Lightweight Key Encapsulation Using LDPC Codes on FPGAs

Jingwei Hu, Marco Baldi, Paolo Santini, Neng Zeng, San Ling, Huaxiong Wang

Abstract—In this paper, we present a lightweight hardware design for a recently proposed quantum-safe key encapsulation mechanism based on QC-LDPC codes called LEDAkem, which has been admitted as a round-2 candidate to the NIST post-quantum standardization project. Existing implementations focus on high speed while few of them take into account area or power efficiency, which are particularly decisive for low-cost or power constrained IoT applications. The solution we propose aims at maximizing the metric of area efficiency by rotating the QC-LDPC code representations amongst the block RAMs in digit level. Moreover, optimized parallelized computing techniques, lazy accumulation and block partition are exploited to improve key decapsulation in terms of area and timing efficiency. We show for instance that our area-optimized implementation for 128-bit security requires 6.82×10^5 cycles and 2.26×10^6 cycles to encapsulate and decapsulate a shared secret, respectively. The area-optimized design uses only 39 slices (3% of the available logic) and 809 slices (39% of the available logic) for key encapsulation and key decapsulation respectively, on a small-size low-end Xilinx Spartan-6 FPGA.

Index Terms—Post-Quantum Cryptography, Key Encapsulation Mechanism, QC-LDPC Code, FPGA Implementation.

1 Introduction

Key encapsulation mechanisms (KEMs) are a class of encryption techniques designed to secure symmetric cryptographic key material for transmission using asymmetric (public-key) algorithms. Efficient and robust quantum-safe KEM design is a crucial and urgent topic in the cryptographic community. Recent years witnessed the NIST call for efficient and secure post-quantum KEMs within the post-quantum cryptography standardization competition [1]. The construction of a commercial quantum computer in not-so-distant future is a desperate threat to quantum-vulnerable primitives, which rely on the hardness of the integer factorization or discrete logarithm problems such as the Diffie-Hellman key exchange, the Rivest-Shamir-Adleman (RSA) cryptosystem and Elliptic Curve Cryptography. Shor's algorithm [2] can be deployed on a quantum computer to solve both the integer factorization problem and the discrete logarithm problem in polynomial time. Code-based cryptosystems, which build their security on the hardness of decoding general linear codes, are among the most promising quantum-resistant candidates for which no known polynomial time attack running on a quantum computer exists.

McEliece proposed the first code-based cryptosystem in 1978 [3], which uses Goppa codes [4] as the underlying coding system. Goppa code-based schemes yield large public keys, which limit the deployment of such systems to resource-constrained scenarios. Niederreiter proposed in 1986 another code-based system [5] exploiting the same trapdoor, but using syndromes instead of codewords and parity-check matrices instead of generator matrices to construct the ciphertext and the key, respectively. It has been proved that, when based on the same code family, the Niederreiter and McEliece cryptosystems are equivalent [6] and achieve the same security levels.

J. Hu, N. Zeng , S. Ling and H. Wang are with the Division of Mathematical Sciences, Nanyang Technological University, Singapore. M. Baldi and P. Santini are with the Department of Information Engineering, Marche Polytechnic University, Italy. e-mail: {davidhu, HXWang, lingsan}@ntu.edu.sg, ZENG0106@e.ntu.edu.sg, m.baldi@univpm.it, p.santini@pm.univpm.it.

Active research is focused on replacing Goppa codes with other families of structured codes that might lead to key size reduction. Nevertheless, this attempt may also compromise the system security. For example, some McEliece variants based on low-density parity-check (LDPC) codes [7], quasi-cyclic low-density parity-check (QC-LDPC) codes [8], quasi-dyadic (QD) codes [9], convolutional codes [10], generalized Reed-Solomon (GRS) codes [11], and rank-metric codes [12], [13] have been successfully attacked. Nevertheless, some variants exploiting QC-LDPC and quasi-cyclic moderate-density parity-check (QC-MDPC) codes [14], [15], [16] have been shown to counteract existing attacks while achieving compact keys. A variety of KEMs built on the Niederreiter cryptosystem and LDPC/MDPC codes such as LEDAkem [17], CAKE [18], BIKE [19], have also started to appear in the literature.

Recently, a new statistical attack, called reaction attack [20], [21], [22] has been devised to recover the key by exploiting the information leaked from decoding failures in QC-LDPC and QC-MDPC code-based systems. The reaction attack is further enhanced in [23], where an error pattern chaining method is introduced to generate multiple undecodable error patterns from an initial error pattern, thus improving the performance of such a reaction attack against instances with only indistinguishability under chosen-plaintext attack (IND-CPA).

1.1 Related work

Cryptographic hardware for the classical McEliece/Niederreiter schemes based on Goppa codes has been extensively studied in the last decade. In 2009, the first FPGA-based implementation of the McEliece cryptosystem was proposed targeting a Xilinx Spartan-3 FPGA and encrypting and decrypting data in 1.07 ms and 2.88 ms, respectively, using security parameters achieving 80-bit security [24]. The authors of [25] presented another accelerator for McEliece encryption with binary Goppa codes on a more powerful Virtex5-LX110T, which is capable to encrypt and decrypt a block in 0.5 ms and 1.3 ms, respectively, providing a similar level of security. The solution in [26] based on hardware/software co-design on a Spartan3-1400AN decrypts a block in 1 ms at 92 MHz with the same level of security. Heyse and Güneysu in [27] report that a Goppa code-based Niederreiter decryption operation consumes 58.78 μs on a Virtex6-LX240T FPGA for 80-bit security (with parameters $n = 2048$ and $t = 27$). Wang *et.al.* presented in [28] an FPGA-based key generator for the Goppa code-based Niederreiter cryptosystem, and later in [29] a full implementation that includes modules for encryption, decryption, and key generation. Their designs decrypt the ciphertext in 60 μs for 256-bit security on a Stratix V FPGA.

The first implementation of the MDPC code-based McEliece cryptosystem on embedded devices was presented in [30] in 2013. For 80-bit security, it is reported to run decryption in 125 μs with over 10,000 slices on Xilinx Virtex-6. A lightweight MDPC code-based McEliece system has been implemented on FPGAs by sequentially manipulating cyclic rotations of the private key in block RAMs [31]. This lightweight design achieves circuit compactness requiring 64 slices for encryption and 148 slices for decryption on a low-end Xilinx Spartan-6 device. An area-time efficient MDPC code-based Niederreiter system has been implemented on FPGAs exploiting a resource balanced MDPC decoding unit [32]. Experimental results show that such an architecture decrypts a message in 65 μs by using about 8,000 slices on a Virtex-6 FPGA.

1.2 Contribution

This paper presents the first efficient and scalable FPGA-based cryptographic hardware for a post-quantum KEM using LDPC codes (LEDAkem). The contributions include:

- a generic and scalable hardware description of LEDAkem for all recommended parameter sets ranging from 128-bit to 256-bit security;
- a new digit-level quasi-cyclic rotation for key encapsulation on embedded hardware. This approach is useful for any quasi-cyclic code and more efficient in terms of processing speed and area footprint;
- optimized block partitioning and lazy accumulation techniques to achieve optimal timing efficiency of key decapsulation;
- a concrete experimental instance for 128-bit security that demonstrates the efficiency of our proposed methods on both high-end and low-end FPGAs.

1.3 Outline

The paper is organized as follows. In Section 1 we describe LEDAkem. In Section 2 we present design considerations for implementing LEDAkem on reconfigurable devices. In Section 3 a generic lightweight hardware architecture for LEDAkem is presented and discussed. In Section 4 experimental results on Xilinx FPGAs are shown to demonstrate the efficiency of the proposed solutions. In Section 5 we draw some conclusions.

TABLE 1: LEDAkem parameters for each NIST security category [17], [33].

Category	$\mathbf{n_0}$	Original Submission					**Revised**				
		p	$\mathbf{d_v}$	m	**t**	**DFR**	**p**	$\mathbf{d_v}$	m	**t**	**DFR**
1	2	27,779	17	[4,3]	224	$\approx 8.3 \cdot 10^{-9}$	14,939	11	[4,3]	136	$\approx 1.16 \times 10^{-8}$
	3	18,701	19	[3,2,2]	141	$\leq 10^{-9}$	7,853	9	[4,3,2]	86	$\leq 10^{-9}$
	4	17,027	21	[4,1,1,1]	112	$\leq 10^{-9}$	7,547	13	[2,2,2,1]	69	$\leq 10^{-9}$
2-3	2	57,557	17	[6,5]	349	$\leq 10^{-8}$	24,693	13	[5,3]	199	$\approx 2 \times 10^{-9}$
	3	41,507	19	[3,4,4]	220	$\leq 10^{-8}$	16,067	11	[4,4,3]	127	$\leq 10^{-9}$
	4	35,027	17	[4,3,3,3]	175	$\leq 10^{-8}$	14,341	15	[3,2,2,2]	101	$\leq 10^{-9}$
4-5	2	99,053	19	[7,6]	474	$\leq 10^{-8}$	36,877	11	[7,6]	267	$\leq 10^{-9}$
	3	72,019	19	[7,4,4]	301	$\leq 10^{-8}$	27,437	15	[4,4,3]	169	$\leq 10^{-9}$
	4	60,509	23	[4,3,3,3]	239	$\leq 10^{-8}$	22,691	13	[4,3,3,3]	134	$\leq 10^{-9}$

2 Preliminaries of LEDAkem

A novel code-based key encapsulation mechanism using LDPC codes, called LEDAkem [17], was proposed by Baldi *et. al* in 2018. The use of a QC-LDPC code $C(n, k, p)$ in LEDAkem allows key size reduction with respect to unstructured codes. LEDAkem has been originally been proposed as a candidate for the NIST call for Post-Quantum Cryptography Standardization that defines 5 security categories, numbered from 1 to 5 and characterized by increasing security levels [34]. The authors proposed nine instances of LEDAkem (as shown in the left part of Table 1), with key sizes starting from 3480 bytes at the lowest security level. New parameters were announced in March 2019, with tight key sizes starting at 1868 bytes, as shown in the right part of Table 1. The use of quasi-cyclic codes brings a beneficial reduction of memory storage required for public keys, for which only the first row/column of each circulant block needs to be stored (the remaining part can be recovered through cyclic rotations), thus resulting in public key sizes considerably smaller than those of other candidates relying on different families of codes. LEDAkem currently is one of the 17 round-2 candidate algorithms of the NIST post-quantum cryptography standardization process [17].

The LEDAkem primitives are described next. Algorithm 1 describes the key generation part of the scheme. Both private and public keys consist of binary quasi-cyclic matrices which are all formed by $p \times p$ circulant blocks, being p an integer properly chosen. The private key is formed by the two matrices H and Q. The public key M corresponds to the systematic form of $L = HQ$, which is obtained through multiplication by the inverse of the rightmost circulant block L_{n_0-1} of L. Since M is systematic, it suffices to use the left part M_l as the public key.

Algorithm 2 describes how to encapsulate an ephemeral random secret e. Note that this random secret acts as the binary error vector with weight t. Bob encapsulates the shared secret e by fetching the public key M_l and then computing $s = [M_l|I]e^T$. The public syndrome vector s representing the encapsulated secret is then sent out.

Algorithm 3 describes the decapsulation of the received ciphertext s into the original error vector e. The syndrome s can be written as $s = Me^T = L_{n_0-1}^{-1}HQe^T$. First, Alice computes $s' = L_{n_0-1}s = HQe^T$. Then, an algorithm called Q-decoder is exploited to perform QC-LDPC decoding through H and Q, in order to recover e from s'.

The public code in LEDAkem is defined by the parity-check matrix $L = HQ$, which is a QC-MDPC matrix.

```
Input: $n$, $p$, $n_0$, and the public key $M_l$
Output: key pairs $M_l$, $\{H, Q\}$
1 randomly generate $n_0$ sparse circulant matrices $H_i$ with size $p \times p$ and
    row/column weight $d_v$ to formulate the secret LDPC matrix as:
      $H = [H_0|H_1|H_2|\dots|H_{n_0-1}]$
2 randomly generate $n_0^2$ sparse circulant blocks $Q_{i,j}$ to formulate the secret
    sparse matrix $Q$ as:
      $Q = \begin{bmatrix} Q_{0,0} & Q_{0,1} & \cdots & Q_{0,n_0-1} \\ Q_{1,0} & Q_{1,1} & \cdots & Q_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{n_0-1,0} & Q_{n_0-1,1} & \cdots & Q_{n_0-1,n_0-1} \end{bmatrix}$
    where the row/column weight of each block $Q_{i,j}$ is fixed as:
      $wt(Q) = \begin{bmatrix} m_0 & m_1 & \cdots & m_{n_0-1} \\ m_{n_0-1} & m_0 & \cdots & m_{n_0-2} \\ \vdots & \vdots & \ddots & \vdots \\ m_1 & m_{n_0-1} & \cdots & m_0 \end{bmatrix}$
3 the secret key (SK) of LEDAkem is formed by $\{H, Q\}$
4 compute the matrix $L$ from $H$ and $Q$ as:
      $L = HQ = [L_0|L_1|L_2|\dots|L_{n_0-1}]$
5 compute the matrix $M$ after inverting $L_{n_0-1}$ as:
      $M = L_{n_0-1}^{-1}L = [M_0|M_1|M_2|\dots|M_{n_0-2}|I] = [M_l|I]$
6 the public key (PK) of LEDAkem is formed by $M_l$
7 return PK and SK
```
Algorithm 1: LEDAkem Key Generation [17]

```
Input: the public key $M_l$
Output: ciphertext $s$
1 randomly generate a binary vector $e$ where $len(e) = n = n_0p$, and
    $wt(e) = t$
2 compute $s = [M_l|I]e^T$
3 return $s$
```
Algorithm 2: LEDAkem Encapsulation [17]

```
Input: the secret key $\{H, Q\}$ and $L_{n_0-1}$, the last circulant block of the
    matrix $HQ$
Output: shared secret $e$
1 compute $s' = L_{n_0-1}s$
2 decode $s'$ to $e$ by Q-decoder: $e$ = Q-decoder$_{H,Q}(s')$
3 return $e$
```
Algorithm 3: LEDAkem Decapsulation [17]

The public code is a QC-MDPC code also in BIKE, which is another round-2 candidate to the NIST post-quantum cryptography standardization process. However, differently from BIKE and other QC-MDPC code-based schemes, the private code structure employed in LEDAkem facilitates decoding. In fact, the so-called Q-decoder used in LEDAkem takes into account the multiplication by Q^T in obtaining the error vector e' to be corrected through decoding from the error vector e used for encryption, that is, $e' = eQ^T$. Thus, decoding the private QC-LDPC code through the Q-decoder allows achieving very fast decoding while keeping the decryption failure rate (DFR) very small (in the order of 10^{-9} or less).

TABLE 2: Timing diagram for FLIP to update one bit of e

	READ_Ltr	READ_S′	SHIFT_8bit+SHIFT_1bit	WRITE_S′
cycle counts	1	1	#shift+1	1

3 Design decisions for lightweight oriented configurable devices

In this section, we introduce some techniques to implement the main functions of LEDAkem (with the revised parameter sets) on small, embedded systems. In particular, we thoroughly describe the manipulation of digit-level rotations in block RAMs (*i.e.,* step-2 in Algorithm 2), the Q-decoder, and some other techniques for key decapsulation.

```
Input: a cyclic matrix stored in RAM $B[\cdot]$ where $B[\cdot]$ has $m = \lceil \frac{p}{d} \rceil$
    entries, and len$(B[0]) = \Delta$, len$(B[i, i \neq 0]) = d$
Output: $B[\cdot]$ rotated by one bit
1 cache $\leftarrow 0$
2 for $i \leftarrow m-1$ to 0 do
3     $B[i] \leftarrow (B[i] << 1)|$cache
4     cache $\leftarrow$ MSB$(B[i])$
                              // cache the highest bit of $B[i]$
5 $B[m-1] \leftarrow B[m-1]|$cache
                              // update the lowest bit of $B[m-1]$
6 return $B[\cdot]$
```
Algorithm 4: Cyclic rotation by one bit (Rot_Bit)

```
Input: a cyclic matrix stored in RAM $B[\cdot]$ where $B[\cdot]$ has $m = \lceil \frac{p}{d} \rceil$
    entries, and len$(B[0]) = \Delta$, len$(B[i, i \neq 0]) = d$
Output: $B[\cdot]$ rotated by one digit
1 cache$_1 \leftarrow B[m-1]$
2 for $i \leftarrow m-2$ to 1 do
3     cache$_2 \leftarrow B[i]$
4     $B[i] \leftarrow$ cache$_1$
5     cache$_1 \leftarrow$ cache$_2$
6 $B[m-1] \leftarrow B[0] << (d-\Delta) +$ cache$_1 >> \Delta$
      // update $B[m-1]$ by concatenating $B[0]$ and the
    highest $d-\Delta$ bits of cache$_1$
7 $B[0] \leftarrow$ cache$_1 \& (2^\Delta - 1)$
      // update $B[0]$ by truncating cache$_1$ to its lowest $\Delta$
    bits
8 return $B[\cdot]$
```
Algorithm 5: Cyclic rotation by one digit (Rot_Digit)

```
Input: a cyclic matrix stored in RAM $B[\cdot]$ where $B[\cdot]$ has $m = \lceil \frac{p}{d} \rceil$
    entries, and len$(B[0]) = \Delta$, len$(B[i, i \neq 0]) = d$
Output: $B[\cdot]$ rotated by $n$ digits
1 for $i \leftarrow n$ to 0 do
2     cache$[i] \leftarrow B[i]$
              // cache the leading $n+1$ entries in $B[\cdot]$
3 for $i \leftarrow 0$ to $n-1$ do
                      // update $B[i, i = 1, 2, \cdots, m-n-1]$
4     $j \leftarrow m-1-i$
5     cache$_1 \leftarrow B[j]$
6     while $j > n$ do
7         cache$_2 \leftarrow B[j-n]$
8         $B[j-n] \leftarrow$ cache$_1$
9         cache$_1 \leftarrow$ cache$_2$
10        $j \leftarrow j-n$
11 $B[0] \leftarrow B[n] \& (2^\Delta - 1)$
12 cache$_1 \leftarrow B[n] >> \Delta$
13 for $i \leftarrow 0$ to $n-1$ do
                      // update the last $n$ entries in $B[\cdot]$
14    cache$_2 \leftarrow$ cache$[n-1-i] \& (2^\Delta - 1)$
15    $B[m-1-i] \leftarrow$ cache$_2 << (d-\Delta) +$ cache$_1$
16    cache$_1 \leftarrow$ cache$[n-1-i] >> \Delta$
17 return $B[\cdot]$
```
Algorithm 6: Multiple cyclic rotation by one digit (Rot_MDigit)

```
Input: a cyclic matrix stored in RAM $B[\cdot]$
Output: $B[\cdot]$ rotated by $l$ bit
1 for $i \leftarrow 0$ to $l-1$ do
2     $B[\cdot] \leftarrow$ Rot_Bit($B[\cdot]$)
3 return $B[\cdot]$
```

Algorithm 7: Matrix cyclic rotation in bit-by-bit fashion

```
Input: a cyclic matrix stored in RAM $B[\cdot]$ where len($B[i, i \neq 0]$) $= d$
Output: $B[\cdot]$ rotated by $l$ bit
1 $q \cdots l' \leftarrow l \div d$
2 for $i \leftarrow 0$ to $q-1$ do
3     $B[\cdot] \leftarrow$ Rot_Digit($B[\cdot]$)
4 for $i \leftarrow 0$ to $l'-1$ do
5     $B[\cdot] \leftarrow$ Rot_Bit($B[\cdot]$)
6 return $B[\cdot]$
```

Algorithm 8: Matrix cyclic rotation in digit-by-digit fashion

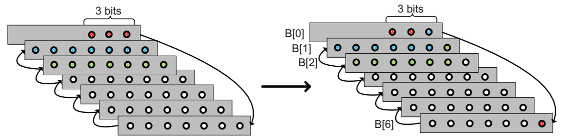
```
Input: a cyclic matrix stored in RAM $B[\cdot]$ where len($B[i, i \neq 0]$) $= d$
Output: $B[\cdot]$ rotated by $l$ bit
1 $q_1 \cdots l' \leftarrow l \div nd$
2 $q_2 \cdots l'' \leftarrow l' \div d$
3 for $i \leftarrow 0$ to $q_1-1$ do
4     $B[\cdot] \leftarrow$ Rot_MDigit($B[\cdot]$)
5 for $i \leftarrow 0$ to $q_2-1$ do
6     $B[\cdot] \leftarrow$ Rot_Digit($B[\cdot]$)
7 for $i \leftarrow 0$ to $l''-1$ do
8     $B[\cdot] \leftarrow$ Rot_Bit($B[\cdot]$)
9 return $B[\cdot]$
```

Algorithm 9: Matrix cyclic rotation in multiple digit-by-digit fashion

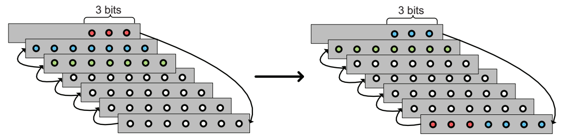
3.1 Cyclic rotation in digit-by-digit fashion

The most important operation regarding key encapsulation/decapsulation is the quasi-cyclic rotation of rows of the private/public keys. Due to the quasi-cyclic nature of the code, it is natural to deposit the first row of each quasi-cyclic block of the key and to retrieve the other rows by rotating the first row. For instance, the computation of the public syndrome $s = [M_l|I]e^T$ is performed by extracting the corresponding rows of M_l successively and then accumulating them. In the previous implementations [30], [32], [35], the first row of the private/public keys was initially stored in block RAMs and loaded to a large quasi-cyclic shifting array for rotating it to the correct form. The problem with this approach is that a single row is still so lengthy that one has to utilize registers of long bit widths to store and perform arithmetic on them. This downside considerably increases usage of the reconfigurable resource, *e.g.* slice utilization, and hence these designs cannot be implemented adequately on some low-end FPGA platforms.

On the other hand, the idea to operate the quasi-cyclic rotation entirely in block RAMs is proposed in [31] to overcome the above issue. Unlike loading and processing one row of long bits at a time, only a small portion of the row (7 bits in the actual implementations) is quasi-cyclically shifted and written back to the block RAMs, which permits to implement a very compact and low power design that handles every 7 bits at one time (see Fig. 1a). To increase the memory access efficiency as much as possible, the read_first mode of Xilinx FPGA BRAM is exploited to enable reading and writing an identical address within the same time interval.



(a) rotate the public/secret key in bit-by-bit fashion



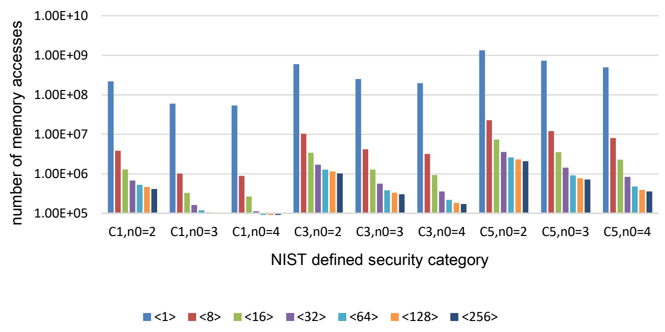
(b) rotate the public/secret key in digit-by-digit fashion

Fig. 1: Example of bit-by-bit versus digit-by-digit rotation. The first address of the memory $B[\cdot]$, *i.e.,* $B[0]$ stores 3 bits and other addresses store 7 bits.

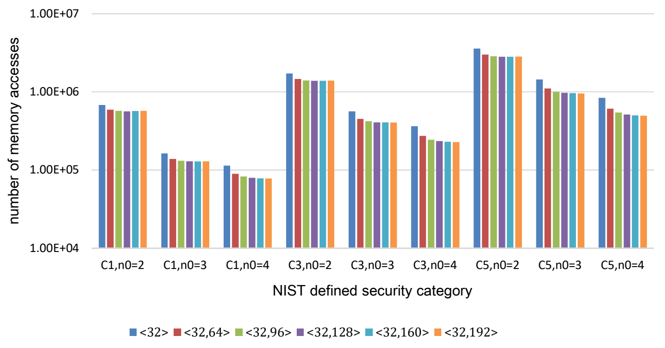
In this work, we propose an alternative approach that accelerates the rotation as in [31] while maintaining the lightweight design feature. This approach is commonly useful for all quasi-cyclic code-based schemes. We use the big-endian notation to store the cyclic matrix. For instance, a cyclic matrix B can be characterized by its first row/ column vector $(b_0, b_1, \ldots, b_{p-1})$: then, in such a case, the first address $B[0]$ stores $(b_{p-1}, b_{p-2}, b_{p-3})$, the second address $B[1]$ stores $(b_{p-4}, b_{p-5}, \ldots, b_{p-10})$ and *etc.* The idea behind our approach is to use the same memory organization as in [31] (in other terms, the first row $B[\cdot]$ of a quasi-cyclic matrix is arranged in RAM using big-endian format), but rotate multiple bits (called digit) each time. Note that $B[\cdot]$ is organized in digits, in other terms, each entry of $B[\cdot]$ except the first one has a digit $d = 7$ bits, while the first entry has a fractional digit $\Delta = p \bmod d$ bits. It is natural to rotate these entries by digit since this operation is straightforwardly done by overwriting each entry by the one below it. Fig. 1b depicts the changes of the first three entries $B[0], B[1], B[2]$ in the digit-by-digit fashion. Here a digit has 7 bits. Before rotation, $B[0], B[1], B[2]$ locate at line 0, 1, 2, respectively. After rotation, B[2] moves to B[1], the least significant 3 bits of B[1] move to B[0], the original B[0] concatenated with the most significant 4 bits of B[1] moves to B[6].

For better clarity, we describe the rotation operations for bit-by-bit and digit-by-digit modes in Algorithms 4 and 5. We point out that our bit-by-bit version deviates slightly from [31]. In [31], the first entry B[0] is restricted to 1 bit for ease of cyclic rotation, while our version is not limited by this restriction. Algorithm 6 describes the faster multiple-digits rotation mode that appears to be efficient if the distance between subsequent rotation indices is large. In this case, it is beneficial to rotate multiple digits each time to approach the target rotation faster. The price we have to pay is the control overhead and extra caches to deposit temporary data: as we show in Algorithm 6, n+1 additional caches are required to operate the n-digit rotation.

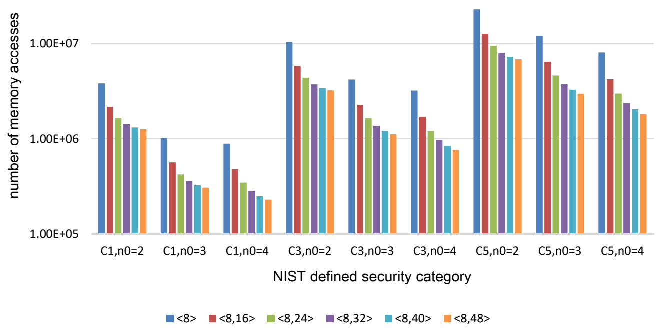
Based on the previously described three rotation modes (Rot_Bit, Rot_Digit, Rot_MDigit), Algorithms 7, 8 and 9 are designed to rotate a matrix by a specific number of positions l. Algorithm 7 describes the basic bit-by-bit rotation using Rot_Bit() l times. Algorithm 8 describes the digit-by-digit



(a) rotation by single digit (8,16,32,64,128,256-bits)



(b) rotation by multiple digits (32-bits)



(c) rotation by multiple digits (8-bits)

Fig. 2: Performance evaluation on distinct rotation modes for $M_i \cdot e_i^T$

rotation which uses Rot_Digit() q times and then switches to Rot_Bit() l' times. Algorithm 9 describes the multiple digit-by-digit rotation that first uses Rot_MDigit() q_1 times and then switches to Rot_Digit() q_2 times, and finally to Rot_Bit() l'' times. It is noteworthy to highlight that if l is smaller than the digit size d, Algorithms 8 and 9 boil down to Algorithm 7 and thus do not improve the rotation efficiency. In other words, Algorithms 8 and 9 are only useful if the rotation step is relatively large ($l > d$).

We study the optimal digit size d for cyclic rotation in RAMs in terms of timing efficiency. This parameter is critical as d determines the data width of an operand (for example, the data width of block memory, registers, field addition, *etc.*) manipulated in hardware. For simplicity, the number of memory accesses is measured to indicate the timing efficiency. Rot_Bit(), Rot_Digit() and Rot_MDigit() process each entry B[i] in B[·] and must read and write B[i] at least once. Consequently, Rot_Bit(), Rot_Digit() and Rot_MDigit() are all assigned to $m = \lceil p/d \rceil$ times of memory accesses. With this basic assumption, let us consider the cyclic rotation in key encapsulation shown in step-2, Algorithm 2 as

$$\begin{aligned} s &= [M_l|I] \cdot e^T \\ &= [M_0|M_1|M_2|\ldots|M_{n_0-2}|I] \cdot [e_0|e_1|\ldots|e_{n_0-2}|e_{n_0-1}]^T \\ &= \sum_{i=0}^{n_0-2} M_i e_i^T + e_{n_0-1}^T \end{aligned}$$

where M_i is a $p \times p$ cyclic matrix and e_i is a $1 \times p$ sparse vector. The multiplication of M_i by e_i^T is equivalent to the accumulation of cyclic rotations of the first column of M_i with respect to the non-zero elements in e_i. Note that the average weight of e_i is $wt(e_i) = t/n_0$ as the errors are uniformly distributed in the vector e. The total number of memory accesses for cyclic rotations in $M_i e_i^T$ is estimated heuristically through Montecarlo simulations. In particular, the digit size d is restricted to the form of 2^k as this form of d eliminates the integer division required in Rot_Digit() and Rot_MDigit(), and for each value of d, one million simulations are run to calculate the average number of memory accesses. Fig. 2a depicts how the performance differs given some typical values of d if the basic digit-level rotation (Alg. 8) is applied to compute $M_i e_i^T$ for all parameters proposed in [36]. We observe that the performance improves rapidly when d increases. However, if d overcomes a particular value, here, $d = 32$, the performance gain becomes negligible as d continues to increase. The performance between the basic and the multiple digit-by-digit rotation is also compared and a comprehensive set of experimental results ($d = 32$ versus $(2d, 3d, 4d, 5d, 6d)$) is reported in Fig. 2b. We observe that the multiple digit-by-digit mode does not always perform well for the LEDAkem parameter sets. The performance gains for six category 1 and 3 instances are negligible. On the other hand, if the digit d is set to a small value, for example, $< 8, 16 >$ (first rotate 16 bits, switch to rotate 8 bits if the remaining offset is smaller than 16), Algorithm 9 exhibits approximately 50% memory access reduction, as shown in Fig. 2c. To conclude, we suggest using the $d = 32$ basic digit-by-digit mode for LEDAkem hardware, as this value balances timing performance and implementation costs. For extreme cases that demand area efficiency or power efficiency, we recommend using the $d = 8$ multiple digit-by-digit mode.

3.2 Q-decoder

LEDAkem proposes to use an improved bit flipping decoder, called Q-decoder, to decode the public syndrome s' into the secret error vector e. The Q-decoder allows recovering e directly though H while taking into account the effect of the multiplication of e by Q. Algorithm 10 describes the operations performed by the Q-decoder. Note that the threshold values $b^{(i)}(s^{(i)})$ are predefined to provide good decoding performance for the Q-decoder. $s^{(i)}$ is a $1 \times p$ row vector and $\hat{e}^{(i)}$ is a $1 \times n$ row vector used in the i-th iteration of the Q-decoder, with initial values $s^{(0)} = s'^T, \hat{e}^{(0)} = 0$. The operand $*$ denotes multiplication performed in the integer domain $\mathbb{Z}$.

TABLE 3: Example of control logic for cyclic rotation in block RAMs

clk count	wea	web	addra	addrb	douta	doutb	cache	dina	dinb
0	$\underline{\text{r}}$	$\underline{\text{r}}$	m-1	m-2	—	—	—	—	—
1	w	r	m-2	m-3	B[m-1]	B[m-2]	—	douta	—
2	w	r	m-3	m-4	—	B[m-3]	B[m-2]	cache	—
⋮	⋮	⋮	⋮	⋮	⋮				
m-2	w	r	1	0	—	B[1]	B[2]	cache	—
m-1	$\underline{\text{w}}$	$\underline{\text{w}}$	0	m-1	—	B[0]	B[1]	cache&(2^r-1)	{doutb,cache $>>r$}

(a) Timing diagram for Rot_Digit

clk count	wea	web	addra	addrb	douta	doutb	cache	dina	dinb
0	$\underline{\text{r}}$	$\underline{\text{r}}$	m-1	m-2	—	—	—	—	—
1	w	w	m-1	m-2	B[m-1]	B[m-2]	—	{douta&$(2^{d-1}-1)$,0}	{doutb&$(2^{d-1}-1)$,douta>> $(d-1)$}
2	r	r	m-3	m-4	—	—	B[m-2]	—	—
⋮	⋮	⋮	⋮	⋮	⋮				
m-2	w	w	2	1	B[2]	B[1]	B[3]	{douta&$(2^{d-1}-1)$,cache>> $(d-1)$}	{doutb&$(2^{d-1}-1)$,douta>> $(d-1)$}
m-1	r	r	0	m-1	—	—	B[1]	—	—
m	w	w	0	m-1	B[0]	B[m-1]	B[1]	{douta&$(2^{r-1}-1)$,cache>> $(d-1)$}	{doutb>> 1,douta>> $(r-1)$}

(b) Timing diagram for Rot_Bit

TABLE 4: Q-decoder decision thresholds used in our LEDAkem design

$\mathbf{C_1, n_0=2}$		$\mathbf{C_1, n_0=3}$		$\mathbf{C_1, n_0=4}$		$\mathbf{C_{2,3}, n_0=2}$		$\mathbf{C_{2,3}, n_0=3}$		$\mathbf{C_{2,3}, n_0=4}$		$\mathbf{C_{4,5}, n_0=2}$		$\mathbf{C_{4,5}, n_0=3}$		$\mathbf{C_{4,5}, n_0=4}$	
w_j	b_j	w_j	b_j	w_j	b_j	w_j	b_j	w_j	b_j	w_j	b_j	w_j	b_j	w_j	b_j	w_j	b_j
0	41	0	43	0	49	0	61	0	64	0	71	0	74	0	86	0	88
2919	42	1089	44	1840	50	4230	62	3954	65	3146	72	5521	75	6369	87	4779	89
4401	43	2212	45	2333	51	7022	63	5039	66	4261	73	9830	76	8307	88	6564	90
5178	44	2717	46	2648	52	8501	64	5663	67	4859	74	12004	77	9436	89	7596	91
5648	45	3003	47	2850	53	9423	65	6070	68	5263	75	13363	78	10162	90	8223	92
		3214	48	2978	54	10023	66	6383	69	5568	76	14241	79	10692	91	8707	93
		3355	49			10452	67	6602	70	5771	77	14915	80	11113	92	9047	94
						10777	68	6773	71	5924	78	15384	81	11419	93	9314	95
										6061	79	15770	82	11664	94	9522	96
												16069	83	11861	95	9709	97
												16300	84				

```
    Input: the secret key $\{H, Q\}$ and the syndrome
           $s' = HQe^T$
    Output: shared secret $e$ or decoding failure
1   for $i \leftarrow 1$ to $l_{max}$ do
2       $\Sigma^{(i)} = [\delta_1^{(i)}, \delta_2^{(i)}, \cdots, \delta_n^{(i)}] = s^{(i-1)} * H$
3       $R^{(i)} = [\rho_1^{(i)}, \rho_2^{(i)}, \cdots, \rho_n^{(i)}] = \Sigma^{(i)} * Q$
4       Define $\mathfrak{F}^{(i)} = \{v \in [1, n] | \rho_v^{(l)} > b^{(i)}(s^{(i)})\}$
5       Update $\hat{e}^{(i)} = \hat{e}^{(i-1)} + 1_{\mathfrak{F}^{(i)}}$, where $1_{\mathfrak{F}^{(i)}}$ is $1 \times n$
          vector with non-zero entries indexed by $\mathfrak{F}^{(i)}$
6       Update $s^{(i)} = s^{(i-1)} + \sum_{v \in \mathfrak{F}^{(i)}} q_v H^T$, where $q_v$ is
          the $v$-th row of $Q^T$
7       if $wt(s^{(i)})$ == 0 then
8           return $\hat{e}^{(i)}$
9   return dec_fail
```

Algorithm 10: LEDAkem Q-decoder [17]

The Q-decoder returns the shared secret e within l_{max} iterations. The DFR is extremely low while employing a significantly smaller number of decoding iterations with respect to classic bit flipping, if appropriate threshold values $b^{(1)}(s^{(1)}), b^{(2)}(s^{(2)}), \ldots, b^{(i_{max})}(s^{(i_{max})})$ are used. In [17], an adaptive choice of decision threshold values is proposed, according to which $b^{(i)}(s^{(i)})$ is computed per each iteration as a function of the syndrome weight $wt(s^{(i)})$. This approach can be implemented efficiently by populating a lookup table with the pairs $\{w_j, b_j\}$ listed in sequential order, where w_j denotes the syndrome weight and b_j corresponds to the flipping threshold. During an iteration (i-th iteration in Algorithm 10), the syndrome weight $wt(s^{(i)})$ is first computed, then the largest w_j is searched in the look-up table such that $w_j < wt(s^{(i)})$, and the corresponding b_j is used as the threshold $b^{(i)}(s^{(i)})$. Table 4 reports the look-up table values for all the considered parameter sets, which have been computed according to the algorithm mentioned in [17, Section 2.4]. The DFR achieved by each one of the considered instances has been estimated through Montecarlo simulations based on a Q-decoder using the aforementioned thresholds. For the Category 1 parameter set with n_0 = 2, we obtained 14 failures over 10^9 decryption simulations, pointing to a DFR $\approx 1.16 \times 10^{-8}$ and for the Category 3 parameter set with n_0=2, we obtained 2 failures out of 10^9 decryptions, pointing to a DFR $\approx 2 \times 10^{-9}$. We have encountered no failures after running 10^9 decoding instances for the remaining parameter sets in Category 1/3/5, thus the DFR can be approximately bounded by the reciprocal of the number of simulations, i.e., DFR $\lesssim 10^{-9}$.

Resistance to side channel attacks of the Q-decoder module should be examined. The two most popular side channel attacks against practical cryptographic implementations are timing and power analysis attacks. Note that, since LEDAkem employs ephemeral keys, it provides a natural resistance against non-profiled power analysis attacks, as a significant amount of power trace collections with the same key is mandatory before the key is eventually revealed. Concerning timing attacks, the current implementation of the Q-decoder is not characterized by constant-time execution. However, we observe that for all proposed parameter sets, the number of iterations made by the Q-decoder is between 3 and 5, with a significant bias towards 4. It is simple to thwart the timing analysis if the Q-decoder always runs for the maximum needed amount of iterations without sacrificing the desired DFR. In addition, LEDAkem exploits ephemeral keys, while typical timing attacks are based on the observation of a large number of decryption instances performed with the same key. This also suggests that the LEDAkem is naturally immune to timing attacks.

3.3 Computation in sparse representation

In general, key decapsulation is more complex than key encapsulation as more matrix multiplications are involved. Unlike the matrix multiplication $s = Me^T$ performed in key encapsulation, the matrix multiplications used in key decapsulation, i.e., $s' = L_{n_0-1}s$, $s^{(i-1)} * H$ and $\Sigma^{(i)} * Q$, keep the form of a sparse circulant matrix multiplied by a dense vector. However, our proposed digit-by-digit rotation is not directly applicable to this specific non-standard form, since our method requires the standard form in which the vector is sparse and then fast rotation is possible by indexing the non-zero entries in the sparse vector.

In this subsection we propose an efficient computation approach that exploits the extremely sparse representation of L_{n_0-1}, H, Q to accelerate key decapsulation. First consider $s' = L_{n_0-1}s$. define $s = [s_0, s_1, \ldots, s_{p-1}]^T$ and $L_{n_0-1} = \begin{bmatrix} l_0 & l_1 & \cdots & l_{p-1} \\ l_{p-1} & l_0 & \cdots & l_{p-2} \\ \vdots & \vdots & \ddots & \vdots \\ l_1 & l_2 & \cdots & l_0 \end{bmatrix}$. Then for each entry s'_i of $s' = [s'_0, s'_1, \ldots, s'_{p-1}]$, we apparently have $s'_i = \sum_{i=0}^{p-1} l_{(p-i) \bmod p} s_i$. Define $\mathfrak{L} = \{v \in [0, p-1] | l_v \neq 0\}$ and $\mathfrak{L}_i$ as the i-th entry of $\mathfrak{L}$, then s'_i is reformulated as

$s'_i = \sum_{j=0}^{md_v-1} s_{(\mathfrak{L}_j+i) \bmod p}$. In other terms, the computation of each s'_i is obtained by only md_v reads of $s_{i,i\in\mathfrak{L}}$, and a total of pmd_v reads is needed to compute the vector s'. Next consider $\Sigma^{(i)} = s^{(i-1)} * H$. Define $\mathfrak{H} = \{v \in [0, p-1] | h_v \neq 0\}$, then we have $\delta_i^{(i)} = \sum_{j=0}^{d_v-1} s_{(\mathfrak{H}_j+i) \bmod p}$, which implies a total number of nd_v reads needed to compute the vector $\Sigma^{(i)}$. Similarly, a total of nm reads is needed for computing $R^{(i)} = \Sigma^{(i)} * Q$.

With the above estimation, we can evaluate the timing performance of key decapsulation, which helps identifying possible performance bottlenecks. Let #iter, #flip denote the average number of decoding iteration and the number of bit flips occurring in each iteration, respectively, then the total number of memory reads for key decapsulation is:

$$pmd_v + \#\text{iter} \cdot (nd_v + nm + \#\text{flip} \cdot md_v)$$

The Monte-Carlo simulation used in Section 2.1 is useful to estimate #iter, #flip. According to our experiments, for all nine LEDAkem instances we have #iter={3.020, 3.134, 2.973, 3.856, 3.022, 3.014, 3.221, 3.022, 3.012} and #flip={83.070, 49.649, 40.869, 102.426, 81.913, 63.988, 167.859, 111.423, 88.436}. We compare the number of memory accesses, *i.e.*, the timing performance between the key encapsulation and decapsulation. Results are reported in Fig. 3a, from which we observe some asymmetry in the key decapsulation performance, which drives us to introduce two new acceleration techniques in order to reduce such a performance gap.

3.3.1 Parallelization of block partition

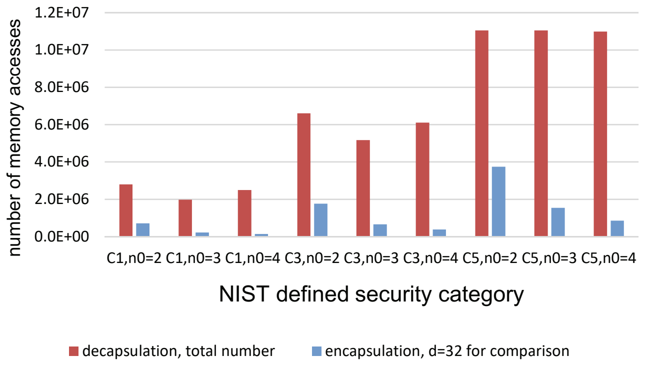
The first technique we use is based on parallel computing for partitioned blocks. Note that the computation of $\Sigma^{(i)} = s^{(i-1)} * H$ and $R^{(i)} = \Sigma^{(i)} * Q$ in the Q-decoder essentially consist of n_0 blocks:

$$\begin{aligned} s^{(i-1)} * H &= s^{(i-1)} * [H_0 | H_1 | \dots | H_{n_0-1}] \\ &= [s^{(i-1)} * H_0 | s^{(i-1)} * H_1 | \dots | s^{(i-1)} * H_{n_0-1}] \end{aligned}$$

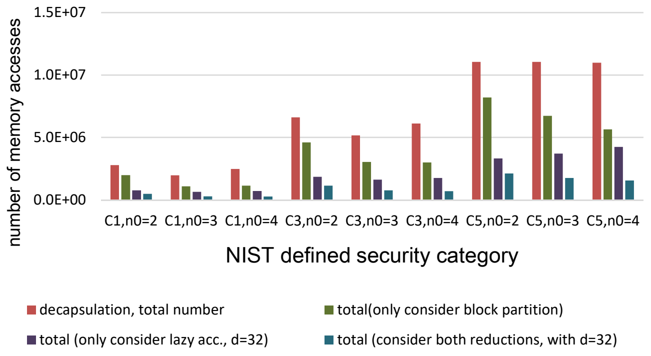
$$\begin{aligned} \Sigma^{(i)} * Q &= [\Sigma_0^{(i)} | \dots | \Sigma_{n_0-1}^{(i)}] * \begin{bmatrix} Q_{0,0} & \cdots & Q_{0,n_0-1} \\ Q_{1,0} & \cdots & Q_{1,n_0-1} \\ \vdots & \ddots & \vdots \\ Q_{n_0-1,0} & \cdots & Q_{n_0-1,n_0-1} \end{bmatrix} \\ &= \left[\sum_{j=0}^{n_0-1} \Sigma_j^{(i)} * Q_{j,0} \middle| \cdots \middle| \sum_{j=0}^{n_0-1} \Sigma_j^{(i)} * Q_{j,n_0-1} \right] \end{aligned}$$

If all these blocks can be processed in parallel, a performance improvement by a factor n_0 is achieved. Indeed, if n_0 copies of the vector $s^{(i-1)}$ are allowed, it is obvious to parallelize all blocks of $s^{(i-1)} * H$. Furthermore, if the blocks of Q are carefully scheduled, it is also possible to parallelize all blocks of $\Sigma^{(i)} * Q$. Table 5 lists the detailed schedule to compute $\Sigma^{(i)} * Q$ with only one copy of $\Sigma^{(i)}$ and Q. The idea is to assign distinct partitions $\Sigma_j^{(i)}$ of the vector $\Sigma^{(i)}$ to each computation block so that computation for each block is performed independently from the others and thus one copy of $\Sigma^{(i)}$ is sufficient. To summarize, if the parallelized block computation is applied, then the total number of memory accesses for key decapsulation is:

$$pmd_v + \#\text{iter} \cdot (pd_v + pm + \#\text{flip} \cdot md_v)$$



(a) Performance evaluation for key decapsulation, compared with key encryption



(b) Performance evaluation for optimized key decapsulation, compared with original key decapsulation

Fig. 3: Estimated memory accesses for LEDAkem key decapsulation

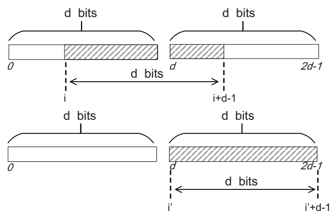
TABLE 5: Parallelization by a factor n_0 of $R^{(i)} = \Sigma^{(i)} * Q$ in the Q-decoder

	compute $\sum_{j=0}^{n_0-1} \Sigma_j^{(i)} * Q_{j,0}$	compute $\sum_{j=0}^{n_0-1} \Sigma_j^{(i)} * Q_{j,1}$	$\cdots$	compute $\sum_{j=0}^{n_0-1} \Sigma_j^{(i)} * Q_{j,n_0-1}$
step 1	$\Sigma_0 * Q_{0,0}$	$\Sigma_1 * Q_{0,1}$	$\cdots$	$\Sigma_{n_0-1} * Q_{0,n_0-1}$
step 2	$\Sigma_1 * Q_{1,0}$	$\Sigma_2 * Q_{1,1}$	$\cdots$	$\Sigma_0 * Q_{1,n_0-1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
step n_0	$\Sigma_{n_0-1} * Q_{n_0-1,0}$	$\Sigma_0 * Q_{1,1}$	$\cdots$	$\Sigma_{n_0-2} * Q_{n_0-1,n_0-1}$

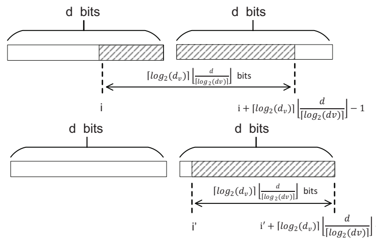
3.3.2 Optimization of lazy accumulation

The second technique we propose is called lazy accumulation. Recall the memory structure to store L_{n_0-1}, $\Sigma^{(i)}$ and $R^{(i)}$. Multiple bits, or more precisely, d bits of data are stored at each memory address, thus it is preferable to process d bits in parallel as the hardware always obtains d bits of data whenever it accesses the memory.

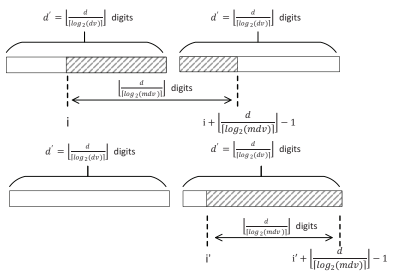
To be specific, let us first consider a simple example concerning $s' = L_{n_0-1}s$. As discussed earlier in this subsection, for a particular entry of s', let us say the i-th entry $s'_i = \sum_{j=0}^{md_v-1} s_{(\mathfrak{L}_j+i) \bmod p}$, the subsequent entry is s'_{i+1}, $s'_{i+1} = \sum_{j=0}^{md_v-1} s_{(\mathfrak{L}_j+i+1) \bmod p}$. It occurs with probability $1 - 1/d$ that $s_{(\mathfrak{L}_j+i) \bmod p}$ and $s_{(\mathfrak{L}_j+i+1) \bmod p}$ reside in the same address, and therefore s'_i and s'_{i+1} can be computed in parallel without additional memory accesses. Table 6 lists the detailed schedule to compute s'_i and s'_{i+1} in parallel. In step 1, one reads the memory once and gets $s_{(\mathfrak{L}_0+i) \bmod p}$ and $s_{(\mathfrak{L}_0+i+1) \bmod p}$. In step 2, one reads the memory again and gets $s_{(\mathfrak{L}_1+i) \bmod p}$ and $s_{(\mathfrak{L}_1+i+1) \bmod p}$ to accumulate



(a) Extract the target d bits from $2d$ bits of s, and then align the d bits


d bits
d bits
$\lceil log_2(d_v)\rceil \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor$ bits
i
$i + \lceil log_2(d_v)\rceil \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor - 1$
d bits
d bits
$\lceil log_2(d_v)\rceil \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor$ bits
i'
$i' + \lceil log_2(d_v)\rceil \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor - 1$

(b) Extract the target $\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ digits from $2d$ bits of $s^{(i-1)}$, and then align the $\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ bits


$d' = \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor$ digits
$d' = \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor$ digits
$\lfloor \frac{d}{\lceil log_2(mdv)\rceil} \rfloor$ digits
i
$i + \lfloor \frac{d}{\lceil log_2(mdv)\rceil} \rfloor - 1$
$d' = \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor$ digits
$d' = \lfloor \frac{d}{\lceil log_2(dv)\rceil} \rfloor$ digits
$\lfloor \frac{d}{\lceil log_2(mdv)\rceil} \rfloor$ digits
i'
$i' + \lfloor \frac{d}{\lceil log_2(mdv)\rceil} \rfloor - 1$

(c) Extract the target $\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ digits from $2\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ digits of $\Sigma^{(i)}$, and then align the $\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ digits

Fig. 4: Lazy accumulation to compute multiple bits of $s' = L_{n_0-1}s, s^{(i-1)} * H, \Sigma^{(i)} * Q$

TABLE 6: Parallelization to compute 2 bits of $s' = L_{n_0-1}s$ by lazy accumulation

	step 1	step 2	$\cdots$	step md_v
compute s'_i	$s_{(\mathfrak{L}_0+i) \bmod p}$	$s_{(\mathfrak{L}_1+i) \bmod p}$	$\cdots$	$s_{(\mathfrak{L}_{md_v-1}+i) \bmod p}$
compute s'_{i+1}	$s_{(\mathfrak{L}_0+i+1) \bmod p}$	$s_{(\mathfrak{L}_1+i+1) \bmod p}$	$\cdots$	$s_{(\mathfrak{L}_{md_v-1}+i+1) \bmod p}$

TABLE 7: Optimized shift value l in lazy accumulation for $s' = L_{n_0-1}s$

	let $d=8$			let $d=32$					let $d=64$					
	l=1	l=2	**l = 4**	l=1	l=2	l=4	**l = 8**	l=16	l=1	l=2	l=4	**l = 8**	l=16	l=32
shift number	4.50	2.50	**2.25**	16.51	8.50	5.25	**5.12**	8.05	32.48	16.50	9.25	**7.12**	9.07	16.03
$\rho_s = \frac{d}{\text{shift number}}$	3.56			6.25					8.99					

respectively, and finally in step md_v, one computes s'_i and s'_{i+1}. Indeed, if one reads two memory addresses at once by using dual-port RAM, in other words, if one gets $2d$ bits of s, he can always compute d bits of s' in parallel. It is convenient to compute d bits of s' since s' is also organized to have d bits at one memory addresses. An illustrative example of the computation of d bits of s' is shown in Fig 4a. Assume that in this example we have obtained $2d$ bits of s in two d-bit registers. The beginning index of the target d bits of s stored in memory is $i \in (0, 1, \ldots, d-1)$. Then the ending index should be $i + d - 1$. The target d bits must be aligned right from i to i' such that accumulation only occurs in the last d bits of the register. To achieve this alignment with better efficiency, in other terms, to shift to $i' = d$ from position i, we propose to use two-layered shift registers similarly to Alg. 8, where a shift by l bits is performed first, with $l > 1$, and a one-bit shift is performed then. Note that the index i is uniformly distributed since the secret key H is generated from a uniform PRNG, the shift value $i' - i$ is also uniform in $(1, 2, \ldots, d)$. We run simulations for $d = 8, 32, 64$ to determine the value of l such that the total shift number is minimized. The results are collected in Table 7. To evaluate the performance improvement introduced by lazy accumulation, we compute the acceleration ratio as $\rho_s = \frac{d}{\text{shift number}}$. This ratio measures the average number of bits from s we can generate per shift.

Next consider $\Sigma^{(i)} = s^{(i-1)} * H$. Note that every entry $\delta_j^{(i)}$ of the vector $\Sigma^{(i)}$ is $\lceil log_2(d_v)\rceil$ bits long and this means one memory address stores at most $\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ digits (or equivalently, $\lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ bits) for $\Sigma^{(i)}$ if we impose that d bits of data are stored at each memory address. Lazy accumulation attempts to extract the desired $\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ digits from the total $2d$ bits of data, as shown in Fig. 4b. Note that this time, we aim to align right to the second register by shifting to $i' = 2d - \lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ from position $i \in (0, 1, \ldots, d-1)$. In other terms, the shift value $i' - i$ is uniform in $(d - \lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor + 1, \ldots, 2d - \lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor)$. Simulations are run to determine the best values of l, which are reported in Table 8. Note that in an actual implementation, $\lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor$ is not always in the form of 2^k, and thus it is advisable to truncate this number to the nearest power of two using the function $f_{2^k}()$. Therefore, in odd and even rows of Table 8 we list two sets of data, the former is based on the maximum allowed $d' = \lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor$, the latter is based on the truncated $d' = f_{2^k}(\lceil log_2(d_v)\rceil\lfloor d/\lceil log_2(d_v)\rceil\rfloor)$. Let us define the acceleration ratio $\rho_\Sigma = \frac{d'}{\text{shift number}}$. For all values of d', we observe that $\rho_\Sigma > 1$, which indicates that some improvement is achieved by using lazy accumulation.

Finally, let us consider $R^{(i)} = \Sigma^{(i)} * Q$. Every entry $\rho_j^{(i)}$ of the vector $R^{(i)}$ consists of $\lceil log_2(md_v)\rceil$ bits and this means one memory address stores at most $d'' = \lfloor d'/\lceil log_2(md_v)\rceil\rfloor$ digits for $R^{(i)}$. Lazy accumulation attempts to extract the desired d' bits from the total $2d'$ bits of data, as shown in Fig. 4c. In Table 9, the shift number is grouped into two sets: $[(C_1), (C_3, n_0 = 2)]$ and $[(C_3, n_0 = 3, 4), (C_5)]$, and the acceleration ratio $\rho_R = \frac{\lfloor d/\lceil log_2(md_v)\rceil\rfloor}{\text{shift number}}$ is always greater

TABLE 8: Optimized shift value l in lazy accumulation for $\Sigma^{(i)} = s^{(i-1)} * H$

	$d' = \lfloor\frac{8}{\lceil log_2(d_v)\rceil}\rfloor$ $d' = f_{2^k}(\lfloor\frac{8}{\lceil log_2(d_v)\rceil}\rfloor)$			$d' = \lfloor\frac{32}{\lceil log_2(d_v)\rceil}\rfloor$ $d' = f_{2^k}(\lfloor\frac{32}{\lceil log_2(d_v)\rceil}\rfloor)$					$d' = \lfloor\frac{64}{\lceil log_2(d_v)\rceil}\rfloor$ $d' = f_{2^k}(\lfloor\frac{64}{\lceil log_2(d_v)\rceil}\rfloor)$					
	l=1	l=2	**l = 4**	l=1	l=2	l=4	**l = 8**	l=16	l=1	l=2	l=4	**l = 8**	l=16	l=32
shift number	4.50	2.50	**2.25**	16.50	8.50	5.25	**5.12**	8.06	32.50	16.50	9.25	**7.13**	9.06	16.04
	4.50	2.50	**2.25**	16.50	8.50	5.25	**5.12**	8.06	32.50	16.50	9.25	**7.13**	9.06	16.04
$\rho_\Sigma = \frac{d'}{\text{shift number}}$	3.56			6.25					8.97					
	3.56			6.25					8.97					

TABLE 9: Optimized shift value l in lazy accumulation for $R^{(i)} = \Sigma^{(i)} * Q$

	$d'' = \lfloor\frac{8}{\lceil log_2(md_v)\rceil}\rfloor$ $d'' = f_{2^k}(\lfloor\frac{8}{\lceil log_2(md_v)\rceil}\rfloor)$	$d'' = \lfloor\frac{32}{\lceil log_2(md_v)\rceil}\rfloor$ $d'' = f_{2^k}(\lfloor\frac{32}{\lceil log_2(md_v)\rceil}\rfloor)$				$d'' = \lfloor\frac{64}{\lceil log_2(md_v)\rceil}\rfloor$ $d'' = f_{2^k}(\lfloor\frac{64}{\lceil log_2(md_v)\rceil}\rfloor)$				
		l=1	l=2	**l = 4**	l=8	l=1	l=2	l=4	**l = 8**	l=16
$C_1, n_0 = 2, 3, 4$	1	8.50	4.50	**3.25**	4.12	15.50	8.00	5.00	**5.00**	8.00
$C_3, n_0 = 2$	1	8.50	4.50	**3.25**	4.12	16.50	8.50	5.25	**5.12**	8.05
$C_3, n_0 = 3, 4$	1	8.50	4.50	**3.25**	4.12	16.50	8.50	5.25	**5.12**	8.05
$C_5, n_0 = 2, 3, 4$	1	8.50	4.50	**3.25**	4.12	16.50	8.50	5.25	**5.12**	8.05
$\rho_R = \frac{d''}{\text{shift number}}$	1	1.23, 1.23				1.8, 1.56				
	1	1.23, 1.23				1.56, 1.56				

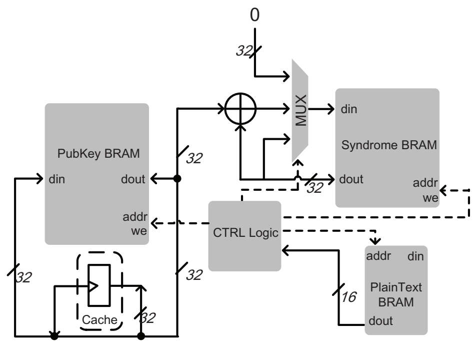


Fig. 5: Generic architecture of key encapsulation engine

than 1, except for $d = 8$, and thus lazy accumulation is mostly advantageous in this case.

To summarize, lazy accumulation is applicable to $s' = L_{n_0-1}s$, $\Sigma^{(i)} = s^{(i-1)} * H$, and $R^{(i)} = \Sigma^{(i)} * Q$ with acceleration factor ρ_s, ρ_Σ, ρ_R, respectively. In other terms, after the optimization of lazy accumulation, the total number of memory accesses is estimated as:

$$pmd_v/\rho_s + \#\text{iter} \cdot (nd_v/\rho_\Sigma + nm/\rho_R + \#\text{flip} \cdot md_v)$$

Fig. 3b compares the timing performance with and without acceleration for $d = 32$. It is seen that the proposed acceleration techniques help improving the performance of key decapsulation by at least four times, and this result even outperforms that of key encapsulation. Note that this comparison is based on a rough estimation that does not consider the timing overhead of memory read/write for intermediate variables and control logic transition. We will present a precise timing estimation for these optimizations in the next section. Moreover, lazy accumulation introduces significantly better acceleration ratio than parallelized block partition without increasing the memory or routing overhead. For parameter sets with small n_0, *i.e.,* $C_{1,3,5}$ with $n_0 = 2$, we suggest using lazy accumulation only; for other parameters, we suggest using both techniques for high-speed applications on high-end FPGAs while using lazy accumulation only for low-end FPGAs.

4 Proposed lightweight architectures for key encapsulation at 128-bit security level

This section describes the proposed key encapsulation and decapsulation engine for the 128-bit security level C_1 with $n_0 = 2, p = 14939, t = 136$. These architectures comply with the techniques proposed in the previous section. We implement the architecture for the 128-bit security level (Category 1, $n_0 = 2$ in Table 1) as a case study, which is easily extensible to higher security levels.

4.1 Key encapsulation engine

Our key encapsulation engine performs the computations shown in Algorithm 2. In particular, the technique we propose to rotate QC-LDPC codes in block RAMs is deployed in this design. Fig. 5 provides an overview of the key encapsulation engine. The shared secret e is stored in the form of t non-zero indices to save memory. The generation of the ciphertext is split into two stages. In the first stage, the second half e_1 of the shared secret e is generated and accumulated to the ciphertext s stored in Syndrome BRAM. e_1 is sparse with average weight $t/2$ and thus we write only the non-zero bits to Syndrome BRAM. Note that s stored in Syndrome BRAM is organized in digits, we instantiate a shift register with initial value 0x0001 to flip the specific bit position in s. Only $t/2$ bit flips are required before e_1 is fully accumulated to s.

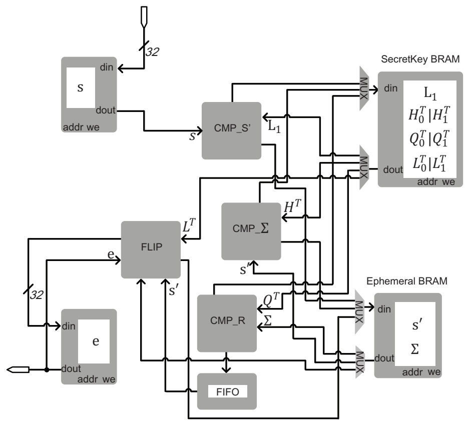


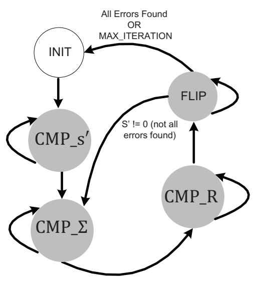
Fig. 6: Generic architecture of key decapsulation engine

The second stage manipulates the cyclic rotation of the public key H stored in PubKey BRAM according to Algorithm 8 and then accumulates the results to Syndrome BRAM. Let $B[\cdot]$ denote the data stored in PubKey BRAM with $m = \lceil p/d \rceil$ entries. True dual-port RAM is exploited to enable Rot_Digit() and Rot_Bit() which owns two independent sets of reading/writing ports, *i.e.,* port A (wea, addra, dina, douta) and port B (web, addrb, dinb, doutb). In general, Rot_Digit() takes m clock cycles and Rot_Bit() takes $m + 1$ cycles. A detailed description for implementing Rot_Bit() and Rot_Digit() on hardware is provided in Table 3. This operation takes one clock cycle delay and a cache register is used to deposit doutb for one more clock cycle to correctly write the rotated data to dina or dinb.

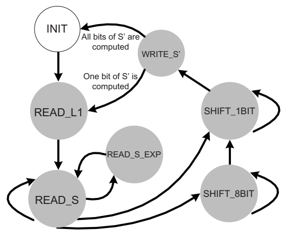
4.2 Key decapsulation engine

Our key encapsulation engine performs the computations corresponding to Algorithm 3 and Algorithm 10. The top level architecture is depicted in Fig. 6. Unlike the key encapsulation engine, the sparse form of secret keys, *i.e.,* indices of non-zero bits of $L_1, H^T = [H_0^T|H_1^T], Q = [Q_0^T|Q_1^T], L^T = [L_0^T|L_1^T]$ are stored in RAMs. The ciphertext input is stored in RAM s and the final shared secret output is stored in RAM e. Critical computing components that include CMP_s', CMP_Σ, CMP_R, and FLIP execute the logic transition peculiarly designed for the Q-decoder (Alg. 10), shown in Fig. 7a.

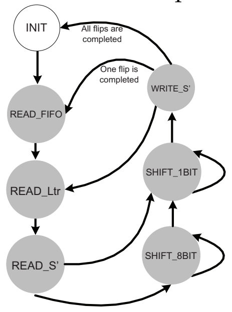
CMP_s' computes $s' = L_{n_0-1}s$ only once in the key decapsulation. Then the decapsulation engine iteratively behaves as the Q-decoder: CMP_Σ computes $\Sigma^{(i)} = s^{(i)} * H$ and CMP_R computes $R^{(i)} = \Sigma^{(i)} * Q$. Eventually, the FLIP block flips the bit positions of the shared secret e and updates the syndrome $s^{(i)}$ according to the thresholds reported in Table. 4.


All Errors Found OR MAX_ITERATION
INIT
FLIP
S' != 0 (not all errors found)
CMP_s'
CMP_R
CMP_Σ

(a) top layer control logic transition

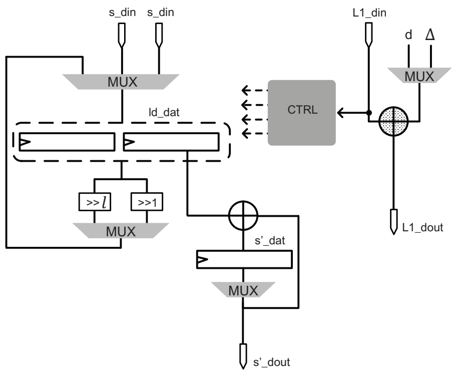


(b) logic transition to compute $s' = L_1 \cdot s$

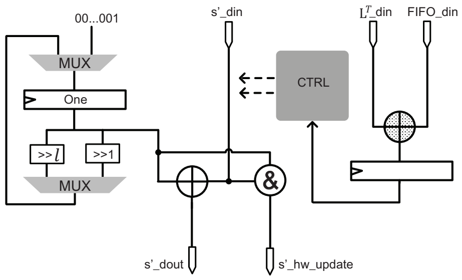


(c) logic transition to update s' via bit flipping

Fig. 7: Typical finite-state machines used in the key decapsulation engine



(a) Internal Structure of CMP_s'



(b) Internal Structure of FLIP

Fig. 8: Critical computing components used in the key decapsulation engine

As illustrated in Section 3.3, the lazy accumulation technique is applied to CMP_s', CMP_Σ, and CMP_R for enhancing the timing performance. The detail of $s' = L_1 s$ is depicted in Fig. 8a. As dual-port RAM is instantiated, we are able to load two successive data of s in one clock cycle to register ld_dat. Two-layer rotating module shifts ld_dat fast such that the lower half of ld_dat includes the target d bits of s according to the values of L_1. Note that we only store the indices of the non-zero entries in the first row of L_1 and these values are added by d in each round (added by Δ in the last round), where d bits of s' are computed. The timing diagram for CMP_L_1, which follows the logic transition in Fig. 7b, is reported in Table 10. This diagram repeats $\lceil pmd_v/d \rceil$ times and thus the total cycle count for CMP_s' is estimated as follows:

TABLE 10: Timing diagram for CMP_L_1 to compute d bits of $s' = L_{n_0-1}s$

	READ_L1	READ_S	SHIFT_8bit+SHIFT_1bit	WRITE_S′	READ_S_EXP
cycle counts	1	1+1	#shift+1	1	1

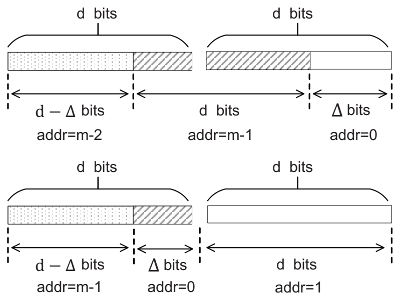


Fig. 9: read_s_exp that occurs when the first line of s is loaded

$$\lceil pmd_v/d \rceil \cdot (\#\text{shift} + 5) + \lceil p/d \rceil + md_v$$

The additional $\lceil p/d \rceil$ represents the cycle count of writing s' back to RAM (WRITE_S′), and md_v represents the extra cycle count needed for the state READ_S_EXP. READ_S_EXP occurs when the first line of s is read out to ld_dat. Two cases may occur in READ_S_EXP to guarantee that $2d$ bits of valid data are always available in ld_dat, as shown in Fig. 9. In the first case, Σ bits and d bits from S[0] and S[m-1] are loaded, and then in the next cycle, the remaining $d - \Sigma$ bits are loaded from S[m-2]. In the second one, d bits and Σ bits from S[1] and S[0] are loaded, and then in the next cycle, the remaining $d - \Sigma$ bits are loaded from S[m-1]. In the case of $C_1, n_0 = 2$, the average number of shifts is 5.12 according to Table 7 with the system parameters $p = 14939, m = 7, d_v = 11, d = 32$, and thus the exact cycle count is $(14939 * 7 * 11 * (5.12 + 5) + 14939)/32 \approx 3.64 * 10^5$.

Likewise, the cycle count for CMP_Σ is estimated as

$$\lceil n_0 p d_v/d \rceil \cdot (\#\text{shift} + 5) + 8n_0 \lceil p/d \rceil$$

Note that CMP_Σ computes $d = 32$ bits of Σ each time but writes the data back to RAM $\Sigma[\cdot]$ using $d/f_{2^k}(\lfloor d/\lceil log_2 d_v \rceil \rfloor) = 8$ clock cycles, which contributes the extra term $8n_0\lceil p/d \rceil$. In the case of $C_1, n_0 = 2$, #shift=5.12 and thus the total count is $2*14939*(11*(5.12+5)+8)/32 \approx 1.11 * 10^5$.

TABLE 11: Timing diagram for FLIP to update one bit of e

	READ_Ltr	READ_S′	SHIFT_8bit+SHIFT_1bit	WRITE_S′
cycle counts	1	1	#shift+1	1

TABLE 12: Implementation results of our LEDAkem engines on a Xilinx Virtex-6 XC6VLX240T FPGA and a Xilinx Spartan-6 XC6SLX75L FPGA after the place and route process.

Aspect	Key Encapsulation Engine		Key Decapsulation Engine	
	Virtex-6	Spartan-6	Virtex-6	Spartan-6
FFs	53	62	658	660
LUTs	104	110	2222	2181
Slices	33	39	870	809
BRAM	1	1	13	17
DSPs	0	0	0	0
Frequency	235 MHz	140 MHz	140 MHz	85 MHz
Time/Op	2.9 ms	4.9 ms	16.1 ms	26.6 ms
Compute e_1	$\approx$ 1144 cycles		—	
Compute He_0^T	$\approx 6.81 \times 10^5$ cycles		—	
Compute s'	—		3.64×10^5 cycles	
Compute $\Sigma^{(i)}$	—		1.11×10^5 cycles	
Compute $R^{(i)}$	—		4.61×10^5 cycles	
Update s'	—		5.76×10^4 cycles	
Overall	6.82×10^5 cycles		2.26×10^6 cycles	

The cycle count for CMP_R is estimated as:

$$\lceil n_0pm/d \rceil \cdot (\text{\#shift} + 5) + 4n_0\lceil p/d \rceil$$

Note that CMP_R calculates $f_{2^k}(\lfloor d/\lceil log_2 md_v \rceil \rfloor) = 4$ digits of R each time but selects the valid ones by scanning these digits one by one. This operation costs the extra $4n_0\lceil p/d \rceil$ cycles. In the case of $C_1, n_0 = 2$, #shift=3.25, d=4 and thus the total count is $2*14939*(7*(3.25+5)+4)/4 \approx 4.61*10^5$.

The FLIP component operates in a different mode other than CMP_s', CMP_Σ, and CMP_R do. As shown in Fig. 7c and 8b, FLIP only updates one bit of the shared secret e according to the value in the vector R in one round of logic transition. A very limited number of positions in R is above the flip threshold, which are stored in the FIFO. For each valid value r_i in FIFO, the r_i-th row of L^T is extracted and then summed to s'. Meanwhile, the Hamming weight of s' is updated on-the-fly by a simple AND of the register *one* and the previous value of s' (s'_din). The critical timing diagram is shown in Table 11, which describes the basic operation of extracting one entry in L^T and then summing to update one bit of $s'(e)$. This timing diagram repeats for #flip $\cdot$ md_v times, which is equivalent to updating $\hat{e}^{(i)} = \hat{e}^{(i-1)} + 1_{\mathfrak{F}^{(i)}}$ and $s^{(i)} = s^{(i-1)} + \sum_{v \in \mathfrak{F}^{(i)}} q_v H^T$, as shown in Algorithm 10.

Thus, the cycle count for FLIP is computed as

$$\text{\#flip} \cdot md_v \cdot (\text{\#shift} + 4) + \text{\#flip}$$

The additional #flip represents the number of FIFO reads. In the case of $C_1, n_0 = 2$, #shift=5, #flip=83.07 and thus the total count is $83.07 * (7 * 11 * (5 + 4) + 1) \approx 5.76 * 10^4$.

To sum up, the total cycle count for key decapsulation is computed as:

$$\text{CMP\_}s' + \text{\#iter} \cdot (\text{CMP\_}\Sigma^{(i)} + \text{CMP\_}R^{(i)} + \text{FLIP})$$

In the case of $C_1, n_0 = 2$, this number equals $2.26 * 10^6$.

5 FPGA IMPLEMENTATION RESULTS AND COMPARISON

In this section, we present our lightweight QC-LDPC KEM implementation results in FPGA. All the results are obtained post place-and-route for a Xilinx high-end FPGA device — Virtex XC6VLX240T and a low-end FPGA device — Spartan XC6SLX16 FPGA using Xilinx ISE 14.7, which demonstrate the compactness of our design on both platforms. In contrast to this work, the prior KEM implementations [25], [29], [30], [32], [38] cannot fit into low-end FPGAs including Spartan-6. As shown in Table 12, our key encapsulation engine on the Virtex-6 device runs at 235 MHz and generates one ciphertext in 6.82×10^5 cycles, that is 2.9 ms of operation time. It runs on the Spartan-6 device at 140 MHz and generates one ciphertext in 4.9 ms. The area footprint is extremely low (33 slices and 39 slices on Virtex-6 and Spartan-6 respectively) as our proposed digit-level QC rotation is entirely executed through memory read and write, which consumes few hardware resources.

On the other hand, our key decapsulation engine runs at 140 MHz and 85 MHz on the Virtex-6 and Spartan-6 device respectively, and decapsulates the shared secret in approximately 2.26×10^6 cycles. This cycle count is equivalent to 16.1 ms and 26.6 ms of operating time for Virtex-6 and Spartan-6, respectively. It is noteworthy to mention that the timing performance of decapsulation is comparable to that of encapsulation, though the operations done in decapsulation include decoding, which is the most complex operation. Such a performance gap is narrowed because the proposed lazy accumulation parallelizes the decapsulation algorithm. Nevertheless, lazy accumulation introduces more complicated logic control and registers, two-layered rotation, and accumulation. The slice usage (870 and 809) reflects this architectural characteristic. The secret key size is also much larger than the public key, together with some ephemeral variables s' and Σ which lead to denser memory consumption: 13 and 17 block RAMs used in Virtex-6 and Spartan-6, respectively.

In the following, we compare our work with other code-based schemes on FPGAs as shown in Table 13. First, we compare with the MDPC code-based schemes that also exploit the quasi-cyclic structure of the codes. Very recently, the BIKE team included some hardware implementation results concerning data encapsulation and key generation in the specification document [19]. A comparison with those data shows that at the same security level of 128 bits, our LEDAkem encapsulation engine runs significantly faster and also uses less slice and memory footprint than the BIKE

TABLE 13: Performance comparison of our lightweight FPGA implementations of LEDAkem with other code-based cryptographic hardware.

Scheme	SL [bit]	Platform	f[MHz]	Time/Op	Cycles	FFs	LUTs	Slices	BRAM
LDPC code:									
Ours (encap.)	**128**	Xilinx Virtex-6	**235**	**2.9ms**	6.82×10^5	**53**	**104**	**33**	**1**
Ours (decap.)			**140**	**16.1ms**	2.26×10^6	**658**	**2,222**	**870**	**13**
MDPC code:									
[19] (encap.)			162.6	39.9ms	6.5×10^6	103	366	125	2
[19] (encap, opt. lv1)	128	Xilinx Artix-7	161.3	20.5ms	3.3×10^6	100	518	190	3
[19] (encap, opt. lv2)			151.5	10.6ms	1.6×10^6	89	668	213	3
[31] (enc.)	80	Xilinx Virtex-6	334	2.2 ms	7.4×10^5	119	226	64	1
[31] (dec.)			318	13.4 ms	4.3×10^6	412	568	148	1
[32] (enc.)	80	Xilinx Virtex-6	400	3.86μs	1,542	10,031	9,886	3,371	2
[32] (dec.)			310	65.76μs	20,384	37,789	24,688	8,781	0
[30] (enc.)	80	Xilinx Virtex-6	351.3	13.66μs	4,800	14,426	8,856	2,920	0
[30] (dec.)			222.5	125.38μs	27,897	32,974	36,554	10,271	0
Goppa code:									
[29] (enc.)				10μs	1498				
[29] (dec.)	103	Xilinx Virtex-5	180	30μs	5864	—	—	6,660	68
[29] (kgen.)				8.35ms	1,503,927				
[37] (enc.)				71μs	2720				
[37] (dec.)	128	Xilinx Artix-7	38.1	410μs	15,638	49,383	25,327	—	68
[37] (kgen.)				42ms	1,599,882				
[25] (enc.)				0.50ms	81,500				
[25] (dec.)	103	Xilinx Virtex-5	163	1.29ms	210,300	—	—	14,537	75
[25] (kgen.)				90 ms	14,670,000				
[38] (dec.)	128	Xilinx Virtex-6	162	0.18ms	28,887	—	—	3,307	15

encapsulation engine does, while our decapsulation engine performs almost at the same level as the BIKE decapsulation does in terms of cycle counts and operation time. The lightweight design from [31] is currently the smallest design based on MDPC codes. However, this design utilizes system parameters for 80-bit security, which below the target posed by NIST for the standardization of post-quantum cryptosystems. Another drawback of the solution proposed in [31] is the high DFR (≈ 0.00506, see [30]). The DFR is a significant performance metric and we believe such a high value is impractical for real deployment. Our design achieves 128-bit security with comparable timing and area consumption, but DFR around 10^{-8}. Moreover, our key encapsulation engine achieves even smaller costs in regard to area and memory consumption. The high-speed design of an MDPC code-based system (actually a primitive version of BIKE) in [30], [32] yields a considerable increase in hardware resources in order to perform fast encryption/decryption. Moreover, it is difficult to scale such an approach to higher security levels with larger key size. The reason is that these designs load and operate the entire key in the register. It is bearable for parameters at 80-bit security level as the public key is a few thousand bits long, but the design performance deteriorates significantly if a higher security level (with typical key sizes of tens of thousands of bits) is considered. Conversely, our methodology maintains the lightweight characteristic for 128-bit security and even higher. Higher security levels only increase the memory consumption of our design since the same arithmetic/logic core is simply iteratively invoked more times in our design.

Second, we compare our work with another popular scheme based on Goppa codes [25], [29], [37], [38]. As a reference implementation, we consider the one reported in [29], where scalability is tested with different system parameters and FPGA platforms. We extract from [29] the experimental results for 103-bit security on a Xilinx FPGA to provide a direct and fair comparison. It is seen that for almost the same security level, Goppa code-based systems require much more memory to store the public/secret key and the arithmetic core that handles Goppa polynomials is also notably complex. Despite this area and memory utilization, Goppa code-based schemes exhibit a scalable and stable timing performance over all NIST security categories. Conversely, our work provides a scalable lightweight design that can be deployed on low-end FPGAs with extremely limited computing resources.

6 Conclusion

This paper presented a lightweight hardware design for a post-quantum key encapsulation mechanism based on LDPC codes, called LEDAkem, which is a second-round candidate to the NIST post-quantum cryptography standardization process. Digit-level cyclic rotation was proposed to accelerate key encapsulation with negligible logic overhead. Lazy accumulation and block partition techniques have also been proposed to optimize the computations needed for performing key decapsulation. The methods above are generic and applicable to other code-based schemes, on condition that the underlying codes are quasi-cyclic.

Acknowledgement

This work was partially supported by Singapore Ministry of Education under Research Grant MOE2016-T2-2-014(S) and the National Research Foundation, Prime Ministers Office, Singapore under its Strategic Capability Research Centres Funding Initiative.

References

[1] L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography," *National Institute of Standards and Technology Internal Report*, vol. 8105, 2016.

[2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM journal on computing*, vol. 26, no. 5, pp. 1484–1509, 1997.

[3] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory," *DSN progress report*, vol. 42, no. 44, pp. 114–116, 1978.

[4] V. D. Goppa, "A new class of linear correcting codes," *Problemy Peredachi Informatsii*, vol. 6, no. 3, pp. 24–30, 1970.

[5] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Problems Of Control and Information Theory-Problemy Upravleniya I Thorii Informatsii*, vol. 15, no. 2, pp. 159–166, 1986.

[6] Y. X. Li, R. H. Deng, and X. M. Wang, "On the equivalence of McEliece's and Niederreiter's public-key cryptosystems," *IEEE Transactions on Information Theory*, vol. 40, no. 1, pp. 271–273, 1994.

[7] C. Monico, J. Rosenthal, and A. Shokrollahi, "Using low density parity check codes in the McEliece cryptosystem," in *Information Theory, 2000. Proceedings. IEEE International Symposium on*. IEEE, 2000, p. 215.

[8] A. Otmani, J.-P. Tillich, and L. Dallot, "Cryptanalysis of two McEliece cryptosystems based on quasi-cyclic codes," *Mathematics in Computer Science*, vol. 3, no. 2, pp. 129–140, 2010.

[9] R. Misoczki and P. S. Barreto, "Compact McEliece keys from Goppa codes," in *International Workshop on Selected Areas in Cryptography*. Springer, 2009, pp. 376–392.

[10] C. Löndahl and T. Johansson, "A new version of McEliece PKC based on convolutional codes," in *International Conference on Information and Communications Security*. Springer, 2012, pp. 461–470.

[11] M. Baldi, M. Bianchi, F. Chiaraluce, J. Rosenthal, and D. Schipani, "Enhanced public key security for the McEliece cryptosystem," *Journal of Cryptology*, vol. 29, no. 1, pp. 1–27, 2016.

[12] P. Loidreau, "A new rank metric codes based encryption scheme," in *International Workshop on Post-Quantum Cryptography*. Springer, 2017, pp. 3–17.

[13] P. Gaborit, A. Otmani, and H. T. Kalachi, "Polynomial-time key recovery attack on the Faure–Loidreau scheme based on Gabidulin codes," *Designs, Codes and Cryptography*, vol. 86, no. 7, pp. 1391–1403, 2018.

[14] M. Baldi, M. Bianchi, and F. Chiaraluce, "Optimization of the parity-check matrix density in QC-LDPC code-based McEliece cryptosystems," in *Communications Workshops (ICC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 707–711.

[15] M. Baldi, M. Bodrato, and F. Chiaraluce, "A new analysis of the Mceliece cryptosystem based on QC-LDPC codes," in *Security and Cryptography for Networks*. Springer, 2008, pp. 246–262.

[16] R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. Barreto, "MDPC-McEliece: New McEliece variants from moderate density parity-check codes," in *IEEE International Symposium on Information Theory Proceedings (ISIT), 2013*. IEEE, 2013, pp. 2069–2073.

[17] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAkem: a post-quantum key encapsulation mechanism based on QC-LDPC codes," in *International Conference on Post-Quantum Cryptography*. Springer, 2018, pp. 3–24.

[18] P. S. Barreto, S. Gueron, T. Gueneysu, R. Misoczki, E. Persichetti, N. Sendrier, and J.-P. Tillich, "Cake: code-based algorithm for key encapsulation," in *IMA International Conference on Cryptography and Coding*. Springer, 2017, pp. 207–226.

[19] N. Aragon, P. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Guneysu, C. A. Melchor *et al.*, "Bike: Bit flipping key encapsulation," 2019.

[20] T. Fabšič, V. Hromada, P. Stankovski, P. Zajac, Q. Guo, and T. Johansson, "A reaction attack on the QC-LDPC McEliece cryptosystem," in *International Workshop on Post-Quantum Cryptography*. Springer, 2017, pp. 51–68.

[21] Q. Guo, T. Johansson, and P. Stankovski, “A key recovery attack on MDPC with CCA security using decoding errors,” in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2016, pp. 789–815.

[22] P. Santini, M. Battaglioni, F. Chiaraluce, and M. Baldi, “Analysis of reaction and timing attacks against cryptosystems based on sparse parity-check codes,” in *Code-Based Cryptography*, M. Baldi, E. Persichetti, and P. Santini, Eds. Cham: Springer International Publishing, 2019, pp. 115–136.

[23] A. Nilsson, T. Johansson, and P. S. Wagner, “Error amplification in code-based cryptography,” *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 238–258, 2019.

[24] T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar, “Microeliece: McEliece for embedded devices,” in *Cryptographic Hardware and Embedded Systems-CHES 2009*. Springer, 2009, pp. 49–64.

[25] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert, “A novel cryptoprocessor architecture for the McEliece public-key cryptosystem,” *Computers, IEEE Transactions on*, vol. 59, no. 11, pp. 1533–1546, 2010.

[26] S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede, “A speed area optimized embedded co-processor for McEliece cryptosystem,” in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*. IEEE, 2012, pp. 102–108.

[27] S. Heyse and T. Güneysu, “Towards one cycle per bit asymmetric encryption: code-based cryptography on reconfigurable hardware,” in *Cryptographic Hardware and Embedded Systems–CHES 2012*. Springer, 2012, pp. 340–355.

[28] W. Wang, J. Szefer, and R. Niederhagen, “FPGA-based key generator for the Niederreiter cryptosystem using binary Goppa codes,” in *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, pp. 253–274.

[29] ——, “FPGA-based Niederreiter cryptosystem using binary Goppa codes,” in *International Conference on Post-Quantum Cryptography*. Springer, 2018, pp. 77–98.

[30] S. Heyse, I. Von Maurich, and T. Güneysu, “Smaller keys for code-based cryptography: QC-MDPC McEliece implementations on embedded devices,” in *Cryptographic Hardware and Embedded Systems–CHES 2013*. Springer, 2013, pp. 273–292.

[31] I. von Maurich and T. Güneysu, “Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices,” in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 38.

[32] J. Hu and R. C. Cheung, “Area-time efficient computation of Niederreiter encryption on QC-MDPC codes for embedded hardware,” *IEEE Transactions on Computers*, 2017.

[33] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, “LEDAcrypt: Low-density parity-check code-based cryptographic systems,” March 2019. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions

[34] “Post-Quantum Cryptography Standardization.” computer security research center, NIST, 2017. [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography

[35] J. Hu and R. C. Cheung, “Toward practical code-based signature: Implementing fast and compact QC-LDGM signature scheme on embedded hardware,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2017.

[36] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, “Design of LEDAkem and LEDApkc instances with tight parameters and bounded decryption failure rate.”

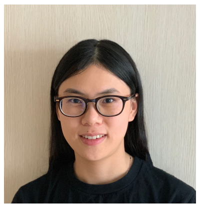
[37] D. J. Bernstein, T. Chou, T. Lange, R. Misoczki, R. Niederhagen, E. Persichetti, P. Schwabe, J. Szefer, and W. Wang, “Classic McEliece: conservative code-based cryptography,” 2019.

[38] P. M. C. Massolino, P. S. Barreto, and W. V. Ruggiero, “Optimized and scalable co-processor for McEliece with binary Goppa codes,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, p. 45, 2015.

Jingwei Hu received the BSc degree in electronic engineering from Dalian Maritime University, Dalian, in 2011, and the MEng degree in computer engineering from Tianjin University, Tianjin, in 2014, and the PhD degree in electronic engineering from City University of Hong Kong, Hong Kong, in 2018. He is currently a postdoctoral research fellow in the School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. His research interest includes embedded cryptographic system, lightweight cryptographic hardware and side channel security.

Marco Baldi received the Laurea degree (Hons.) in electronic engineering and the Ph.D. degree in electronic, computer and telecommunications engineering from the Università Politecnica delle Marche, Ancona, Italy, in 2003 and 2006, respectively. Since 2016, he has been a tenure-track Assistant Professor with the same university. He has co-authored over 150 scientific papers, one book, and three patents. His research is focused on coding and cryptography for information security and reliability. He received the Italian National Scientific Habilitation (ASN) as an Associate Professor of telecommunications engineering in 2017. He currently serves as an Associate Editor for IEEE Communications Letters, the EURASIP Journal on Wireless Communications and Networking, and MDPI Information.

Paolo Santini received the Master degree (Hons.) in Electronic Engineering from the Università Politecnica delle Marche, Ancona, Italy, in 2016, respectively. Since 2016, he has been a PhD student with the same university. His research is focused on coding theory, security and cryptography.



Neng Zeng currently is a Ph. D student (since 2016) in Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore. Her research is focused on coding theory and code-based cryptography.

San Ling received the B.A. degree in mathematics from the University of Cambridge and the Ph.D. degree in mathematics from the University of California, Berkeley. He is currently President’s Chair in Mathematical Sciences, at the Division of Mathematical Sciences, School of Physical and Mathematical Sciences, Nanyang Technological University, Singapore, which he joined in April 2005. Prior to that, he was with the Department of Mathematics, National University of Singapore. His research fields include: arithmetic of modular curves and application of number theory to combinatorial designs, coding theory, cryptography and sequences.

Huaxiong Wang received a PhD in Mathematics from University of Haifa, Israel in 1996 and a PhD in Computer Science from University of Wollongong, Australia in 2001. He is currently an Associate Professor and the Deputy Director of Strategic Centre for Research in Privacy-Preserving Technologies & Systems (SCRIPTS) at Nanyang Technological University in Singapore. He has more than 20 year experience of research in cryptography and information security.