Efficient and Scalable FPGA Design of GF(2^m) Inversion for Post-Quantum Cryptosystems

Andrea Galimberti*, Gabriele Montanaro, and Davide Zoni

Abstract—Post-quantum cryptosystems based on QC-MDPC codes are designed to mitigate the security threat posed by quantum computers to traditional public-key cryptography. The polynomial inversion is the core operation of key generation in such cryptosystems and the adoption of ephemeral keys imposes the execution of key generation for each session. To this end, there is a need for efficient and scalable hardware implementations of the binary polynomial inversion operation to support the key generation primitive across a wide range of computational platforms. This manuscript proposes an efficient and scalable architecture implementing the binary polynomial inversion at the hardware level. Our solution can deliver a performance-optimized implementation for the large polynomials used in post-quantum code-based cryptosystems and for each FPGA of the mid-range Xilinx Artix-7 family. The effectiveness of the proposed solution was validated by means of the BIKE and LEDAcrypt post-quantum QC-MDPC cryptosystems as representative use cases. Compared to the C11- and the optimized AVX2-based software implementations of LEDAcrypt, instances of the proposed architecture targeting the Artix-7 200 FPGA show an average performance improvement of 31.7 and 2.2 times, respectively. Moreover, the proposed architecture delivers a performance improvement up to 18.1 and 21.5 times for AES-128 and AES-192 security levels, respectively, compared to the BIKE hardware implementation.

Index Terms—QC-MDPC cryptosystems, binary polynomial inversion, code-based cryptography, post-quantum cryptography, applied cryptography, FPGA, hardware design.

1 Introduction

Today, public-key cryptography (PKC) is the standard solution to key exchange over an insecure channel. The security of well-established and widely adopted PKC schemes, such as RSA [1], Diffie-Hellman [2], and elliptic-curve cryptosystems [3], relies on the hardness of factoring large integers and of computing discrete logarithms in a cyclic group. However, quantum computers are expected to solve these problems in polynomial time by means of algorithms such as Shor's [4], threatening to make traditional PKC obsolete in the next decades. To cope with the security risk posed by the advancements in quantum computing, the US National Institute of Standards and Technology (NIST) started in 2016 a standardization process to identify a set of post-quantum algorithms to replace current public-key cryptosystems [5]. Submitted proposals span over a wide portion of the state of the art in computational theory, including algebraic geometry [6], coding theory [7], and lattice theory [8]. Despite the theoretical differences, each proposal must satisfy two requirements. First, post-quantum cryptosystems must leverage computationally hard problems for which even quantum computers cannot compute a solution in polynomial time. Second, NIST requires efficient software and hardware implementations targeting Intel Haswell *x86_64* CPUs and Xilinx Artix-7 FPGAs as representative architectures. The choice of targeting FPGAs prevents the adoption of ASIC-specific technology optimizations, thus ensuring a fair comparison of the hardware implementations.

From the theoretical point of view, code-based cryptography has a remarkably good security track, dating back to the McEliece cryptosystem [7] proposed in 1978, and thus motivating its adoption by several proposals to the NIST post-quantum cryptography (PQC) standardization process. However, the strong security and performance of traditional McEliece cryptosystems that employ binary Goppa codes [9] comes at the cost of large memory requirements, in the order of megabytes, to store the key pairs. Quasi-cyclic moderate-density parity-check (QC-MDPC) codes [10] emerged as an effective alternative to binary Goppa codes, reducing the key size of code-based cryptosystems to tens of kilobytes while maintaining security against quantum attacks. The BIKE [11] and LEDAcrypt [12] proposals to the NIST PQC competition rely on QC-MDPC codes.

From the computational point of view, NIST requires not only software but also efficient hardware implementations, since supporting the most computationally intensive parts of each cryptosystem through dedicated accelerators is the key to ensure the wide adoption of post-quantum security solutions across the embedded devices at the edge. To this end, several hardware implementations of BIKE and LEDAcrypt targeting the Xilinx Artix-7 FPGA family have been presented to efficiently support the encapsulation [13], [14], the decapsulation [15], [16], [17], [18], or both [19]. To the best of our knowledge, [19] represents the sole proposal that also provides hardware support for the key generation primitive of modern QC-MDPC cryptosystems, but delivers a hardware architecture that is strongly tailored to low-area FPGAs. Moreover, the solution is not proven to scale effectively to chips with more available resources. In particular,

* *Corresponding author: Andrea Galimberti*

- *A. Galimberti, G. Montanaro and D. Zoni were with the Dipartimento di Elettronica Informazione e Bioingegneria, Politecnico di Milano, 20133 Milano, Italy.
E-mail: {andrea.galimberti, davide.zoni}@polimi.it, gabriele.montanaro@mail.polimi.it*

Manuscript received March 1, 2021; revised XXX.

the hardware is meant to support BIKE instances up to AES-192-equivalent security, i.e., with polynomial length up to 24659, while AES-256 security requires polynomials with length equal to 40973, that is 66% larger, thus exceeding the size of single BRAMs available on the FPGAs targeted by the NIST PQC competition.

TABLE 1: Performance of the software implementations of the BIKE and LEDAcrypt cryptosystems.

Cryptosystem	Performance (10^3 clock cycles) KeyGen	Encaps	Decaps
BIKE [20] AES-128 security level AVX2, Intel Core i7-1065G7	600	220	2220
LEDAcrypt-KEM-CPA [21] AES-128 security level, $n_0 = 2$ AVX2, Intel Core i5-6600	1241	110	785

In contrast, we note that the hardware implementation of the key generation primitive represents a critical component to ensure the efficiency of any post-quantum code-based cryptosystem for three reasons. First, the use of ephemeral private/public key pairs to transmit the session key requires the execution of the key generation primitive at each run. Indeed, the key generation primitive requires a significant fraction of the total execution time (see Table 1), thus motivating its optimization. Second, the size of the operands of key generation is in the order of tens of thousands of bits, thus imposing a careful design of the hardware implementation. Third, key generation must exhibit a constant-time implementation to prevent timing-based side-channel attacks, thus further increasing the design complexity.

1.1 Contributions

This manuscript presents an FPGA-optimized design methodology to implement efficient and scalable hardware support for polynomial inversion in $GF(2^m)$. Inversion dominates the computational complexity of the key generation procedure of QC-MDPC cryptosystems, taking more than 90% of the execution time [20], [21]. The proposed inversion architecture is based on a known algorithm, based on Fermat's little theorem, that iterates exponentiations and multiplications of binary polynomials [22]. The crucial contribution of our research is the efficient and scalable implementation of a parametric hardware accelerator to support the key generation primitive in QC-MDPC cryptosystems across the entire Xilinx Artix-7 FPGA family, that is the hardware target of the NIST PQC competition. Our proposal adds two relevant contributions to the state of the art:

- **Efficiency** - The proposed architecture is optimized to efficiently compute the time-consuming binary polynomial inversion, by employing a parallel architecture for exponentiation and multiplication and an optimal hardware scheduling. Considering the implementation of our solution on the Xilinx Artix-7 200 FPGA, we observed an average performance improvement of 2.2 against the AVX2-based software implementation of LEDAcrypt-KEM-CPA, and a performance improvement of 18.1 and 21.5 times for AES-128 and AES-192 security levels, respectively, compared to the FPGA-based hardware implementation of BIKE.
- **Scalability** - Three parameters allow the designer to optimally select at design time the area-performance trade-off regardless of the polynomial length. Such parameters are *i)* the bandwidth of the architecture's datapath, *ii)* the degree of parallelism for computing the exponentiation, and *iii)* the number of Karatsuba recursions computed in parallel within the multiplication. The exhaustive design space exploration demonstrates the possibility of implementing a performance-optimized inversion module, for each instance of the BIKE and LEDAcrypt-KEM-CPA cryptosystems, over the entire Artix-7 family. Indeed, the smart use of the FPGA block RAM in place of flip-flops to store the inputs, intermediate values, and results allows handling polynomials with a length in the order of tens of thousands of bits even on Artix-7 12, i.e., the smallest FPGA of the Artix-7 family, while still ensuring competitive performance.

1.2 Theoretical background

Quasi-cyclic codes are characterized by parity-check H matrices that are composed of n_0 circulant blocks with size $p \times p$, therefore they can be equivalently represented by the n_0 binary polynomials in $GF(2^p)$ with coefficients equal to the first row of the corresponding circulant blocks. *Moderate-density* codes feature sparse parity-check matrices, i.e., only a small percentage of values are set to 1, allowing for a sparse representation by enumerating the positions of bits set to 1. QC-MDPC codes possess both the *quasi-cyclic* and *moderate-density* properties.

The private key of a QC-MDPC cryptosystem is a parity-check matrix H composed of n_0 circulant blocks H_i (see Equation (1)), while the corresponding public key is computed as in Equation (2). H_i blocks are $p \times p$ circulant matrices, that are equivalent to the polynomials in $\mathbb{Z}_2[x]/(x^p + 1)$ with coefficients corresponding to their topmost rows.

$$H = [H_0|H_1|...|H_{n_0-1}] \tag{1}$$

$$[(H_{n_0-1}^{-1} \cdot H_0)|(H_{n_0-1}^{-1} \cdot H_1)|...|(H_{n_0-1}^{-1} \cdot H_{n_0-2})] \tag{2}$$

As shown in Equation (2), the key generation procedure requires $n_0 - 1$ multiplications, and the first term of each multiplication is the multiplicative inverse of the rightmost circulant block, i.e., H_{n_0-1}. Inversion is thus a critical part of the key generation primitive of QC-MDPC cryptosystems.

The rest of this section overviews the theoretical background of the binary polynomial inversion (see Section 1.2.1) and exponentiation (see Section 1.2.2).

1.2.1 Inversion background

In $GF(2^m)$, a multiplicative inverse for a polynomial $a(x)$, denoted by $a(x)^{-1}$, is a polynomial that when multiplied by $a(x)$ yields the multiplicative identity, i.e., $a(x) \cdot a(x)^{-1} = 1$.

Inversion algorithms can be split in two families, deriving from Euclid's algorithm and from Fermat's little theorem, respectively. Euclid's algorithm allows to compute the greatest common divisor between two polynomials, and polynomial-time algorithms based on it are proposed by [23], [24], [25]. Algorithms based on Fermat's little theorem

Algorithm 1 Inversion procedure from [22]. $a(x)$ is a binary polynomial in $\mathbb{Z}_2[x]/(x^p+1)$ with a multiplicative inverse, where p is a prime such that $ord_2(p) = p-1$. $d(x)$ is the multiplicative inverse of $a(x)$, i.e., $d(x) = a(x)^{-1}$.

```
 1: function [$d(x)$] INVERSION($a(x)$)
 2:     $b(x) = a(x)$;
 3:     $c(x) = a(x)$;
 4:     for $i \in 1 : (\lceil \log_2(p-2) \rceil - 1)$ do
 5:         $d(x) = c(x)^{2^{2^{i-1}}}$;
 6:         $c(x) = d(x) \cdot c(x)$;
 7:         if $(p-2)_2[i] == 1_2$ then
 8:             $d(x) = b(x)^{2^{2^i}}$;
 9:             $b(x) = d(x) \cdot c(x)$;
10:         end if
11:     end for
12:     $d(x) = b(x)^2$;
13:     return $d(x)$;
14: end function
```

date back to the Itoh-Tsujii algorithm (ITA) introduced by [26] and are employed in the software implementations of BIKE [27] and LEDAcrypt [22] and in the hardware implementation of BIKE [19].

The inversion algorithm employed by the software implementation of LEDAcrypt-KEM-CPA [22] is detailed in Algorithm 1. It takes a $\mathbb{Z}_2[x]/(x^p+1)$ binary polynomial $a(x)$ as input and executes a fixed number of iterations to output its multiplicative inverse $d(x) = a(x)^{-1}$. Each iteration (lines 4-11 in Algorithm 1) consists of two exponentiations (lines 5 and 8) and two multiplications (lines 6 and 9). However, lines 8 and 9 of iteration i are executed only when the condition at line 7 is verified, i.e., if the i-th bit of $p-2$ is equal to 1. Finally, a squaring operation produces the inverse polynomial (line 12). Algorithm 1 requires $(\log_2(p-2) + hw(p-2) - 1)$ multiplications and $(\log_2(p-2) + hw(p-2))$ exponentiations, where $hw(y)$ represents the Hamming weight, i.e., the number of bits set to 1, of y. The amount of required operations depends thus not on the input $a(x)$, but exclusively on the polynomial length p, that is a fixed parameter of the QC-MDPC code.

1.2.2 Exponentiation background

Exponentiation in $GF(2^m)$ is the operation that computes $g(x) = f(x)^k$, where the base $f(x)$ and the result $g(x)$ are polynomials in $GF(2^m)$ while the exponent k is a number. If k is a positive integer, then the exponentiation corresponds to iterating k times the multiplication of the base $f(x)$.

Algorithm 2 details the procedure to compute the exponentiation of a binary polynomial in $\mathbb{Z}_2[x]/(x^p+1)$. It takes as inputs the $f(x)$ polynomial and the k non-zero positive integer value, which constitute the base and the exponent, respectively, and it produces the corresponding $g(x)$ polynomial, where $g(x) = f(x)^k$. The exponentiation procedure starts by setting the $g(x)$ polynomial to 0, i.e., its corresponding binary representation is initially constituted by all p bits set to 0 (see line 2 in Algorithm 2). Then, for each i ranging from 0 to $(p-1)$, the algorithm computes the value of the bit in position $i \cdot k \bmod p$ of the $g(x)$ polynomial, i.e., $g(x)[i \cdot k \bmod p]$, as the bit-wise exclusive OR between the

Algorithm 2 Exponentiation procedure. $f(x)$ is a binary polynomial in $\mathbb{Z}_2[x]/(x^p+1)$, where p is a prime such that $ord_2(p) = p-1$. k is a non-zero positive integer, i.e., $k > 0$. $g(x) = f(x)^k$. Note that $a \oplus= b$ is equivalent to $a = a \oplus b$.

```
1: function [$g(x)$] EXPONENTIATION($f(x), k$)
2:     $g(x) = 0$;
3:     for $i \in 0 : (p-1)$ do
4:         $g(x)[(i \cdot k) \bmod p] \oplus= f(x)[i]$;
5:     end for
6:     return $g(x)$;
7: end function
```

values of $g(x)[i \cdot k \bmod p]$ and the i-th bit of the $f(x)$ polynomial, i.e., $f(x)[i]$ (see lines 3-5 in Algorithm 2). Notably, if k and p are coprime, each bit of $g(x)$ is assigned exactly once inside the *for* loop in Algorithm 2, hence each bit of $g(x)$ can be computed independently from the other bits of the same polynomial. Line 2 of Algorithm 2 thus becomes $g(x)[(i \cdot k) \bmod p] = f(x)[i]$. The coprimality condition is verified in the considered application of QC-MDPC codes to cryptography, i.e., the BIKE and LEDAcrypt cryptosystems.

$$f(x) = x^{10} + x^9 + x^3 + x^1 + x^0 = 11000001011_2$$
$$g(x) = f(x)^k = f(x)^4 = \mathbf{00010011011}$$

Time	$f(x)$	$g(x)$
0	11000001011	00000000000
1	11000001011	00000000001
2	11000001011	00000010001
3	11000001011	00000010001
4	11000001011	00000010011
5	11000001011	00000010011
6	11000001011	00000010011
7	11000001011	00000010011
8	11000001011	00000010011
9	11000001011	00000010011
10	11000001011	00000011011
11	11000001011	**00010011011**

Fig. 1: Example of exponentiation.

Figure 1 shows an example of the iterative exponentiation procedure in Algorithm 2 to compute $g(x)$ as the 4-th power of $f(x)$. $f(x)$ and $g(x)$ are polynomials in $\mathbb{Z}_2[x]/(x^p+1)$ represented as p-bit binary values, where k is equal to 4 and p is equal to 11. The procedure takes 12 timesteps. At timestep 0, all bits in $g(x)$ are cleared, i.e., set to 0. One bit of the $f(x)$ polynomial is then processed at each of the subsequent 11 timesteps, with the i-th bit in the $f(x)$ polynomial contributing to generate the bit in position $i \cdot k \bmod p$ in the $g(x)$ polynomial (see line 4 in Algorithm 2), where i ranges from 0 to 10. For each timestep, the processed and generated bits in $f(x)$ and $g(x)$ polynomials are highlighted in red. At the final timestep, the value of $g(x)$ is the result of the exponentiation, i.e., $g(x) = f(x)^4$.

2 Related Works

This section is organized in two parts. Section 2.1 discusses the state of the art related to the binary polynomial inversion, while Section 2.2 overviews the state of the art related to the binary polynomial exponentiation.

2.1 Inversion state of the art

Several algorithms to compute the multiplicative inverses in $GF(2^m)$ and their hardware implementations have been proposed since the 1980s. Fermat's little theorem is at the core of the first state-of-the-art proposals, such as [28] and ITA [26]. Other Fermat-based software and hardware implementations were later introduced by [29], [30], [31], [32], [33], [34], [35]. Euclid's algorithm, that computes the greatest common divisor between two polynomials, was instead first adapted to compute multiplicative inverses in $GF(2^m)$ by [23], that is known as Brunner's algorithm. Subsequent proposals based on Euclid's and Brunner's algorithms were [24], [25], [36]. However, all the previously listed state-of-the-art proposals targeted polynomials with degree in the order of few hundreds at most, due to the lesser requirements of traditional PKC and error control coding schemes.

Only few and more recent proposals target polynomials with degrees in the order of tens of thousands, that are thus suitable to post-quantum QC-MDPC cryptography. They are software and hardware implementations of the BIKE and LEDAcrypt cryptosystems. The software ones target modern *x86_64* CPUs that support custom instructions for carry-less multiplication, while the hardware one targets Artix-7 FPGAs. [27] introduced a constant-time algorithm for polynomial inversion, targeting the software implementation of BIKE and based on Fermat's little theorem. The authors optimized the exponentiation operation and further improved performance by means of a source code targeting the latest Intel Ice Lake CPUs, that support the AVX512 and Vector-PCLMULQDQ instructions. [22] presented a Fermat-based algorithm that is employed in the software implementation of LEDAcrypt and was previously detailed in Section 1.2. [19] presented the FPGA-based implementation of BIKE that employs an inversion algorithm based on [34]. To the best of our knowledge, [19] represents the state-of-the-art hardware implementation of binary polynomial inversion. The employed algorithm differs from the one used in [27], requiring the same number of exponentiations, but slightly less operations if the exponentiations are computed by means of iterated squarings. The algorithms [19], [22], [27] used in BIKE and LEDAcrypt require the same number of exponentiations and multiplications.

2.2 Exponentiation state of the art

Few implementations of the exponentiation algorithm have been proposed in the last decade to efficiently support the key generation algorithm in post-quantum QC-MDPC cryptosystems [19], [22], [27]. [27] performs $GF(2^m)$ exponentiation with two main optimizations. First, the permutation corresponding to a $f(x)^{2^k}$ exponentiation is fully precomputed by storing in a lookup table the positions of bits in the inverse polynomial and indexing them by the original positions in the input polynomial $f(x)$. Lookup tables can be precomputed for all values held by k during the inversion algorithm, which depend exclusively on p. Second, $f(x)^{2^k}$ exponentiations are executed faster as a chain of k squarings, when k is smaller than a threshold value. However, the proposed lookup tables required $p \cdot (\lceil \log_2(p-2) \rceil - 1) \cdot \lceil \log_2 p \rceil$ bits of memory, and may thus not be suitable to constrained devices such as microcontrollers. [22] optimized the memory requirements by using a smaller lookup table, that holds only the $(\lceil \log_2(p-2) \rceil - 1)$ values obtained as $2^i \bmod p$, with $i \in \{1, 2, ..., \lceil \log_2(p-2) \rceil\}$. The position of the j-th coefficient, where $0 \leq j \leq p-1$, of $a(x)^{2^i}$ is instead computed at run-time as $(j \cdot (2^i \bmod p)) \bmod p$, i.e., through a multiplication and a modulus operation. [19] compared three strategies to compute the $f(x)^{2^k}$ exponentiation. The first one iterates k squaring operations, i.e., $f(x)^2$, processed by a squaring module. The second one implements two modules, one computing $f(x)^2$ and the other computing $f(x)^{2^4}$. The latter is used as long as the remaining exponent of the squaring chain is ≥ 4, otherwise the iterative computation is done by the former. The third strategy combines a fixed squaring module computing $f(x)^2$ and a module that computes $f(x)^{2^k}$ exponentiations with arbitrary k. $f(x)^{2^k}$ exponentiations are executed by the latter module when $k \geq BW$, where BW is the width of the architecture datapath, otherwise they are computed by iterative squaring. The third strategy provides the best performance, while occupying slightly more resources than the first one.

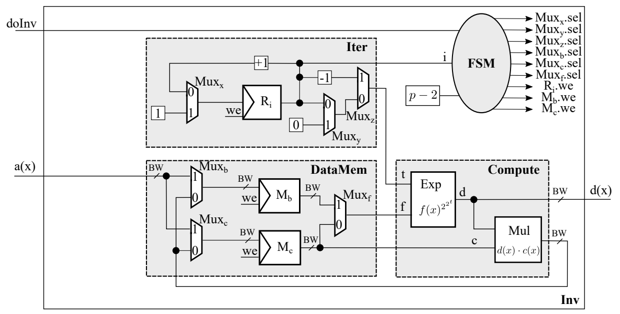
3 Methodology

This section describes the architecture of an efficient and scalable component that computes the multiplicative inverse of a binary polynomial in $\mathbb{Z}_2[x]/(x^p + 1)$. Such arithmetic primitive is the key element employed in the key generation algorithm of QC-MDPC cryptosystems. The efficiency is achieved by means of *i)* a parallel architecture to perform polynomial multiplications and exponentiations and *ii)* an optimal hardware scheduling that allows the concurrent computation of the two operations whenever possible. The scalability is achieved by means of a configurable architecture design that is meant to scale across a wide range of FPGAs rather than being hard-coded to a specific target. The configurable architecture allows to implement the inversion of large binary polynomials on targets ranging from resource-constrained FPGAs up to larger chips that allow for faster execution.

The rest of this section is organized in three parts. Section 3.1 describes the inversion architecture. Section 3.2 presents the architecture of the exponentiation component. Section 3.3 is devoted to its complexity analysis. We note that this manuscript does not aim to optimize the multiplication architecture, since we leverage the scalable and efficient multiplier presented in [14].

3.1 Inversion architecture

The architectural view of the proposed inversion module (`Inv`) is shown in Figure 2a. The module takes as inputs the binary polynomial $a(x)$ to invert and the control signal `doInv` that starts the computation, and outputs the binary polynomial $d(x)$ that is the multiplicative inverse of $a(x)$. The proposed architecture is built upon the inversion algorithm described in Figure 2b.



(a) Inversion architecture

		FSM output control signals								
		Mux.sel						**M/R.we**		
1:	**function** $[d(x)]$ INVERSION$(a(x))$	***x***	***y***	***z***	***b***	***c***	***f***	***i***	***b***	***c***
2:	$b(x) = a(x);$	1	–	–	1	1	–	1	1	1
3:	$c(x) = a(x);$									
4:	**for** $i \in 1 : (\lceil \log_2 (p-2) \rceil - 1)$ **do**	–	–	–	–	–	–	0	0	0
5:	$d(x) = c(x)^{2^{2^{i-1}}};$	–	–	1	–	–	0	0	0	0
6:	$c(x) = d(x) \cdot c(x);$	–	–	–	–	0	–	0	0	1
7:	**if** $(p-2)_2[i] == 1_2$ **then**	–	–	–	–	–	–	0	0	0
8:	$d(x) = b(x)^{2^{2^i}};$	–	0	0	–	–	1	0	0	0
9:	$b(x) = d(x) \cdot c(x);$	–	–	–	0	–	–	0	1	0
10:	**end if**	0	–	–	–	–	–	1	0	0
11:	**end for**	–	–	–	–	–	–	0	0	0
12:	$d(x) = b(x)^2;$	–	1	0	–	–	1	0	0	0
13:	**return** $d(x);$									
14:	**end function**									

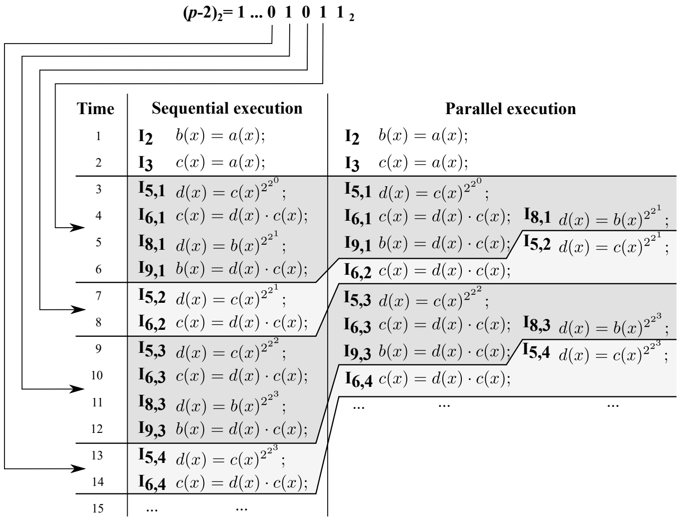
(b) FSM control signals associated to the inversion algorithm

Fig. 2: Top-view architecture of the inversion module, composed of the computational datapath and of the finite state machine that drives the control signals according to the execution of the inversion algorithm (Algorithm 1).

Architectural view - The `Inv` module consists of four submodules, i.e., `Compute`, `DataMem`, `Iter`, and `FSM`. The computational unit (`Compute`) implements the optimized architectures to perform the binary polynomial exponentiation (`Exp`) and multiplication (`Mul`). The memory module (`DataMem`) is meant to efficiently store the input polynomial as well as the intermediate results of the computation. The iteration module (`Iter`) produces the values of the iterator i according to the implemented inversion algorithm (see Figure 2b). Finally, the finite state machine controller (`FSM`) generates the control signals that drive the multiplexers of the datapath and the write enable signals of the registers and memories, depending on the values of the iterator `i`, the code parameter p, and the `doInv` input.

Algorithmic view - The proposed architecture is built upon the inversion procedure described in Algorithm 2b. The input phase starts when the `doInv` input signal is set to 1, storing the binary polynomial `a(x)` received as an input to the `Inv` module in the two memories of the `DataMem` submodule, i.e., $M_{b(x)}$ and $M_{c(x)}$. Such hardware phase corresponds to the execution of the lines 2-3 in Figure 2b. At the end of the input phase, the `Inv` module starts computing the polynomial inverse by iteratively executing the hardware operations corresponding to the instructions at lines 4-11 in Figure 2b. The FSM selectively asserts the selectors of the multiplexers and the write-enable, i.e., `we`, control signals to ensure the correct execution of the inversion procedure. By observing that the value of p is a fixed parameter of the cryptosystem, we note that the FSM only requires the value of the i counter at each iteration to correctly generate the values of the control signals, thus mimicking the execution of the control instructions, i.e., the *for* loop and the *if* conditional statement at lines 4 and 7 of the inversion algorithm. Figure 2b highlights the values of the control signals within the proposed architecture during the hardware execution of the inversion algorithm, where the "$-$" symbol identifies *don't care* values. Once all the iterations have been executed, the FSM forces the final squaring of the $c(x)$ polynomial (see line 12) and subsequently outputs the obtained result $d(x) = a(x)^{-1}$ (see line 13).

Optimized hardware scheduling - To maximize the performance without duplicating the instances of the computational resources, i.e., `Mul` and `Exp`, the proposed inversion architecture is designed to schedule the exponentiations and multiplications to always use the `Exp` and `Mul` modules concurrently whenever possible. Starting from the analysis of the inversion algorithm in Figure 2b, we identified two pairs of instructions for which the computation can be optimized by means of a concurrent execution, since each pair of instructions shows no data dependence. Considering the i-th iteration of the inversion algorithm (see lines 4-11 Figure 2b), the multiplication and the exponentiation instructions at line 6 and 8, respectively, can be concurrently executed on two separate functional units. In a similar manner, the instructions at line 9 of the i-th iteration and at line 5 of the $(i+1)$-th iteration can also be computed at the same time. We note that the concurrent execution of the two pairs of instructions is constrained to the validity of the condition at line 7 of the inversion procedure in Figure 2b, i.e., $(p-2)_2[i] == 1$.



Time	**Sequential execution**	**Parallel execution**	
1	**I$_2$** $b(x) = a(x);$	**I$_2$** $b(x) = a(x);$	
2	**I$_3$** $c(x) = a(x);$	**I$_3$** $c(x) = a(x);$	
3	**I$_{5,1}$** $d(x) = c(x)^{2^{2^0}};$	**I$_{5,1}$** $d(x) = c(x)^{2^{2^0}};$	
4	**I$_{6,1}$** $c(x) = d(x) \cdot c(x);$	**I$_{6,1}$** $c(x) = d(x) \cdot c(x);$	**I$_{8,1}$** $d(x) = b(x)^{2^{2^1}};$
5	**I$_{8,1}$** $d(x) = b(x)^{2^{2^1}};$	**I$_{9,1}$** $b(x) = d(x) \cdot c(x);$	**I$_{5,2}$** $d(x) = c(x)^{2^{2^1}};$
6	**I$_{9,1}$** $b(x) = d(x) \cdot c(x);$	**I$_{6,2}$** $c(x) = d(x) \cdot c(x);$	
7	**I$_{5,2}$** $d(x) = c(x)^{2^{2^1}};$	**I$_{5,3}$** $d(x) = c(x)^{2^{2^2}};$	
8	**I$_{6,2}$** $c(x) = d(x) \cdot c(x);$	**I$_{6,3}$** $c(x) = d(x) \cdot c(x);$	**I$_{8,3}$** $d(x) = b(x)^{2^{2^3}};$
9	**I$_{5,3}$** $d(x) = c(x)^{2^{2^2}};$	**I$_{9,3}$** $b(x) = d(x) \cdot c(x);$	**I$_{5,4}$** $d(x) = c(x)^{2^{2^3}};$
10	**I$_{6,3}$** $c(x) = d(x) \cdot c(x);$	**I$_{6,4}$** $c(x) = d(x) \cdot c(x);$	
11	**I$_{8,3}$** $d(x) = b(x)^{2^{2^3}};$	... ...	...
12	**I$_{9,3}$** $b(x) = d(x) \cdot c(x);$		
13	**I$_{5,4}$** $d(x) = c(x)^{2^{2^3}};$		
14	**I$_{6,4}$** $c(x) = d(x) \cdot c(x);$		
15	... ...		

Fig. 3: Temporal evolution of the sequential and optimized executions of the inversion algorithm for $(p-2) = 459_{10} = 111001011_2$. I$_{x(,y)}$ represents the x-th instruction of the inversion algorithm at the y-th iteration, where $x \in \{1 \ldots 14\}$ and $y \in \{1 \ldots 4\}$.

To demonstrate the effectiveness of the implemented hardware scheduling, Figure 3 shows an example of the execution of the first four iterations of the inversion algorithm, i.e., $i \in \{1, 2, 3, 4\}$, considering $(p-2) = 459_{10} = 111001011_2$. To better highlight the execution speedup due to the proposed optimized hardware scheduling, Figure 3 unrolls the considered *for* loop iterations. In particular, $\mathrm{I}_{x(,y)}$ identifies the instruction at line x of the inversion procedure that is executed during the y-th iteration of the *for* loop. The execution of the inversion algorithm takes advantage of the optimized hardware scheduling for each i-th iteration of the *for* loop such that $(p-2)[i]$ is equal to 1, since the validity of the condition at line 7 (see Figure 2b) allows the concurrent execution of the two identified pairs of multiplication-exponentiation instructions. Considering the example in Figure 3, the optimized hardware scheduling and the non-optimized sequential scheduling execute the four considered iterations in 10 and 14 time units, respectively. The performance speedup of the proposed hardware scheduling is due to the concurrent executions at iterations 1, i.e., $I_{6,1}$-$I_{8,1}$ and $I_{9,1}$-$I_{5,2}$, and 3, i.e., $I_{6,3}$-$I_{8,3}$ and $I_{9,3}$-$I_{5,4}$, respectively (see timesteps 4, 5, 8, and 9 in Figure 3). It is important to note that the actual performance speedup due to the optimized hardware scheduling is a function of the number of ones in the binary encoding of $(p-2)$ (see line 7 in Figure 2b), where p is a parameter of the cryptosystem. However, the selection of the value of p is subject to a set of contrasting requirements to balance the decode failure rate, the performance, and the security of the cryptosystem, thus preventing a choice of its value that only favors the performance of inversion as also highlighted in [20], [21].

Complexity analysis - The time complexity of the inversion procedure (T_{inv}) can be expressed as a function of only the polynomial length p and the execution times of the exponentiation (T_{exp}) and multiplication (T_{mul}). Without considering the proposed scheduling optimization, the inversion procedure requires one exponentiation and one multiplication at each iteration of the *for* loop, and, in addition, one more exponentiation and one more multiplication at each i-th iteration corresponding to an i-th bit of $(p-2)$ that is equal to 1. The number of executed iterations is equal to $\lceil \log_2(p-2) - 1 \rceil$. In addition, one final exponentiation at the power of 2 is performed.

The proposed scheduling optimization reduces the number of operations that are required in the i-th iterations for which $(p-2)_2[i]$ is equal to 1. In such case, an iteration requires two times the execution time of the operation taking the longest between exponentiation and multiplication, instead of the execution time of two exponentiation and two multiplications. The resulting time complexity can therefore be expressed in clock cycles as in Equation 3.

$$\begin{aligned} T_{inv} = & \left((2 \cdot (hw(p-2)-1)) - 1\right) \cdot \max\{T_{exp}, T_{mul}\} \\ & + (zeros(p-2)+1) \cdot (T_{exp} + T_{mul}) \\ & + T_{exp} \end{aligned} \tag{3}$$

Notably, $hw(p-2)$ corresponds to the number of bits of $(p-2)$ set to 1, while $zeros(p-2)$ corresponds to the number of zeros of the binary representation of $(p-2)$, that is equal to $(\lceil \log_2(p-2) \rceil - hw(p-2))$.

For completeness, we also report the time complexity of the multiplier introduced in [14], allowing us to express, together with the time complexity of the exponentiation module discussed later, the execution time of the inversion procedure as a function of only the code parameter p and of the architectural parameters of the implementation. Let PAR_M be the parallelism parameter that expresses how many times the Karatsuba recursion formula is applied, and BW be the bandwidth of the multiplier datapath, then its time complexity can be expressed as in Equation 4.

$$T_{mul} = \left(\sum_{i=0}^{PAR_M} \frac{2}{2^i}\right) \cdot \left\lceil \frac{p}{BW} \right\rceil + \left\lceil \frac{\left\lceil \frac{p}{2^{PAR_M}} \right\rceil}{BW} \right\rceil^2 \tag{4}$$

The first term refers to the data movement between the different layers of Karatsuba recursion, while the second term refers to the execution time required by the 2^{PAR_M} innermost Comba multipliers, each concurrently computing one partial product of the Karatsuba formula.

3.2 Exponentiation architecture

The exponentiation is a critical operation to efficiently perform the polynomial inversion, thus its implementation must be carefully designed to optimize the area-performance trade-off. Starting from the the exponentiation procedure detailed in Algorithm 2, the proposed hardware component leverages the possibility to independently compute each bit of the result polynomial $g(x)$ to deliver a parallel architecture that allows the concurrent computation of PAR_E bits of $g(x)$. The parallel architecture is achieved by employing PAR_E separate hardware memories. In particular, each memory manages the writing of one of the PAR_E bits of $g(x)$. Once all p bits of $f(x)$ have been processed and written to the corresponding PAR_E memories, their bit-wise XOR produces the final $g(x)$ polynomial.

$$f(x) = x^{10} + x^9 + x^3 + x^1 + x^0 = 11000001011_2$$

$$g(x) = f(x)^k = f(x)^4 = \mathbf{00010011011}$$

Time	$f(x)$	$g_1(x)$	$g_0(x)$
0	11000001011	00000000000	00000000000
1	11000001011	00000010000	00000000001
2	11000001011	00000010010	00000000001
3	11000001011	00000010010	00000000001
4	11000001011	00000010010	00000000001
5	11000001011	00000011010	00000000001
6	11000001011	00000011010	00010000001
7	$g(x) = g_1(x) \oplus g_0(x) = \mathbf{00010011011}$		

Fig. 4: Example of parallelized exponentiation.

To demonstrate the performance speedup due to the use of the proposed parallel exponentiation architecture, Figure 4 details the computation of the $g(x)$ polynomial as the 4-th power of the f(x) polynomial using a parallelism of 2, i.e., $PAR_E = 2$. We note that, apart from the parallel computation, the example in Figure 4 performs the computation previously discussed in Section 1.2.2 (see Figure 1). At timestep 0, $f(x)$ holds the input polynomial, while the

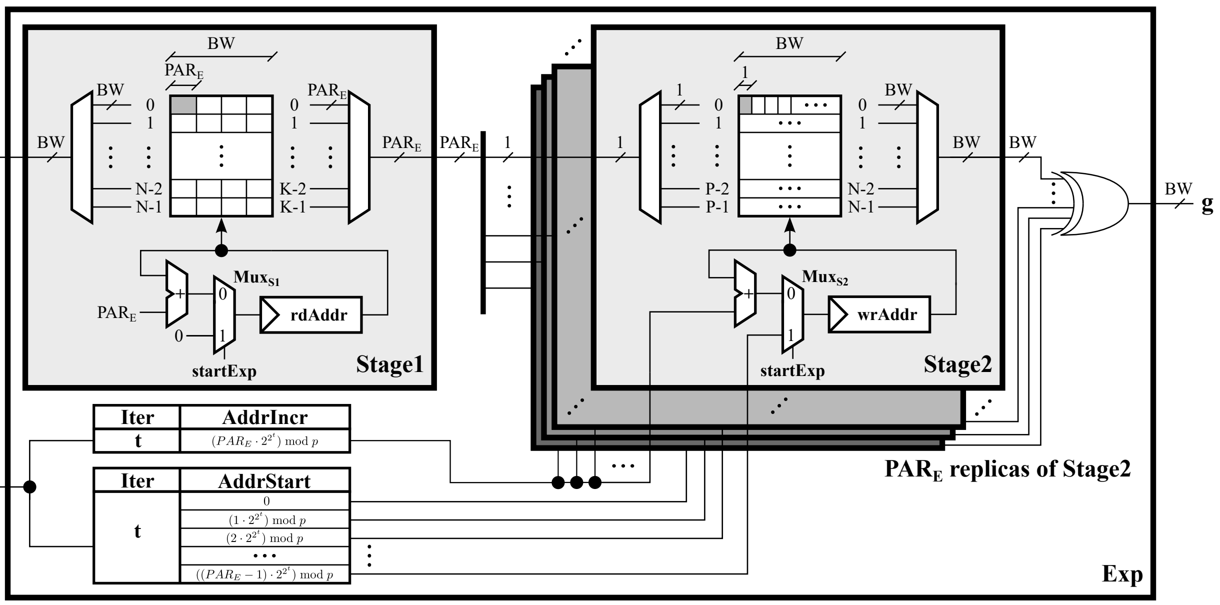

f
BW
PAR$_E$
0
1
N-2
N-1
K-2
K-1
Mux$_{S1}$
rdAddr
startExp
Stage1
Mux$_{S2}$
wrAddr
P-2
P-1
Stage2
g
t
Iter
AddrIncr
$(PAR_E \cdot 2^{2^t}) \bmod p$
AddrStart
$(1 \cdot 2^{2^t}) \bmod p$
$(2 \cdot 2^{2^t}) \bmod p$
$((PAR_E - 1) \cdot 2^{2^t}) \bmod p$
PAR$_E$ replicas of Stage2
Exp

Fig. 5: Detailed view of the proposed exponentiation architecture. ($N = \lceil \frac{P}{BW} \rceil$, $K = \lceil \frac{P}{PAR_E} \rceil$)

PAR_E $g_i(x)$ polynomials, $g_0(x)$ and $g_1(x)$, are set at 0. At each subsequent timestep, PAR_E adjacent bits are read from the $f(x)$ polynomial, and each of them is written to the corresponding $g_i(x)$ polynomial. Blue and red colors to highlight the bits processed at each timestep as well as their positions in the $g_i(x)$ polynomials, where $i \in \{0, 1\}$. Once all p bits of the $f(x)$ polynomial have been read and written in the correct position of the PAR_E $g_i(x)$ polynomials, the $g_i(x)$ polynomials are bit-wise XORed to produce the $g(x)$ result polynomial, which is the 4-th power of $f(x)$.

Architectural view - The `Exp` module in Figure 5 represents our architecture for polynomial exponentiation. It has a `BW`-bit input `f` and an input `t`, corresponding to the base polynomial $f(x)$ and to the exponent 2^{2^t}, respectively, and a `BW`-bit output `g` that corresponds to the resulting polynomial $g(x) = f(x)^{2^{2^t}}$. `BW` is a design-time parameter that defines the datapath bandwidth of the exponentiation module.

The `Exp` module is designed as a two-stage architecture, composed of the `Stage1` and `Stage2` modules. They contain a memory, composed of FPGA BRAMs, that can hold p bits and has a `BW`-bit read/write data bandwidth, and they respectively store the $f(x)$ and $g_i(x)$ polynomials. The PAR_E design-time parameter defines the degree of parallelism within the exponentiation module, i.e., the number of `Stage2` replicas that are instantiated to parallelize the computation. To further improve the efficiency of the proposed architecture, two lookup tables `AddrIncr` and `AddrStart` are populated at compile time to provide the address increment and start values for $g_i(x)$ memories. `AddrIncr` contains $log_2(p-2)$ entries, indexed from 0 to $(log_2(p-2)-1)$, each containing the $(PAR_E \cdot 2^{2^t}) \bmod p$ value, where t is the index of the entry. `AddrStart` contains $log_2(p-2)$ sets of entries, indexed from 0 to $(log_2(p-2)-1)$. Each set of entries contains PAR_E values equal to $(s \cdot 2^{2^t}) \bmod p$ value, where s holds all integer values comprised between 0 and $(PAR_E - 1)$, referring to the corresponding $g_i(x)$ memory, and t is the index of the set of PAR_E entries.

Algorithmic view - The execution of the exponentiation can be seen as organized in three logical phases, i.e., *Input*, *Computation* and *Output*. During the *Input* phase, the `Exp` module stores the p-bit $f(x)$ polynomial into the memory component of the `Stage1` module, passing `BW` bits per clock cycle through the `f` input, while the `Stage2` memory is reset to contain all 0 bits. At the same time, the t value fed through the `t` input is used to index the `AddrIncr` and the PAR_E `AddrStart` values within the two respective lookup tables. The `Stage2` modules share the same `AddrIncr` value, while the `AddrStart` values are correctly dispatched to the instances of the `Stage2` module. Thereafter, the *Computation* phase takes place. At each clock cycle, PAR_E bits are read and output from the memory of the `Stage1` module, from the least to the most significant bits of the p-bit $f(x)$ polynomial. These PAR_E bits are split and each of them is fed as a single-bit signal to one of the replicas of the `Stage2` module. Each single-bit input to a `Stage2` module is written, one per clock cycle, into the `Stage2` memory at a position that starts from the `AddrStart` value and that is incremented (modulo p) at each clock cycle by the `AddrIncr` value. The *Computation* phase ends when all p bits read from the `Stage1` memory have been written to their corresponding positions in the PAR_E `Stage2` memories. Finally, during the *Output* phase, the content of the `Stage2` memories is output, `BW` bits per clock cycle, and the PAR_E `BW`-bit outputs are XORed. We note that p and 2^{2^t} are coprime, i.e., their GCD is 1, thus, it is guaranteed that there can not be any bits set to 1 in two or more different `Stage2` memories, i.e., we cannot have any cancellations due to the XOR operation. The result of the XOR operation corresponds to the actual $g(x)$ polynomial, which is output `BW` bits per clock cycle through the `g` port.

3.3 Exponentiation complexity analysis

This section discusses the time and space complexity of the proposed exponentiation architecture, highlighting the design choices that allow its implementation across a wide range of resource-performance trade-offs.

Time complexity - Equation (5) defines the time required to execute an exponentiation (T_{exp}), expressed in terms of clock cycles.

$$T_{exp} = \left\lceil \frac{p}{BW} \right\rceil \cdot \left\lceil \frac{BW}{PAR_E} \right\rceil \quad (5)$$

It has three parameters. p corresponds to the polynomial length. It is a parameter of the QC-MDPC code and, thus, it can not be controlled by the hardware designer. BW is the bandwidth of the exponentiation datapath expressed in bits and PAR_E is the parallelism implemented in the exponentiation module. Both are configurable parameters of the proposed architecture and can be tuned to explore different area-performance trade-offs.

Equation (5) is the product of two terms. The first term $\lceil \frac{p}{BW} \rceil$ represents the number of memory lines to be read from the input polynomial and written into the output polynomial. The second term $\left\lceil \frac{BW}{PAR_E} \right\rceil$ accounts for the parallel writing on separate BRAMs for the output polynomial. Equation (5) is fully independent from the input polynomial and depends instead exclusively on the p code parameter and on the BW and PAR_E architectural parameters. Since the execution time of the multiplication module is also independent from its input values, and the same holds for the top inversion module, then our implementation guarantees constant-time execution of binary polynomial inversion.

Space complexity - Experimental results showed empirically that LUT and BRAM relative utilization of the available FPGA resources are similar to each other across all hardware instances on the whole Artix-7 family and for all polynomial lengths, with the LUT utilization being slightly larger than the BRAM one on average. At the same time, flip-flops are mostly unused in the proposed architecture. The number of BRAMs is therefore deemed a good metric for the space complexity of the exponentiation module.

Our architecture requires one p-bit memory for the `Stage1` module and one p-bit memory for the `Stage2` module. Due to the parameterized replication of `Stage2` modules, the overall exponentiation module requires $(PAR_E + 1)$ p-bit memories. Equation (6) defines the number of BRAMs of the exponentiation module (M_{exp}).

$$M_{exp} = (PAR_E + 1) \cdot \left\lceil \frac{p}{S_{BRAM}} \right\rceil \cdot \left\lceil \frac{BW}{BW_{BRAM}} \right\rceil \quad (6)$$

It has five parameters. Other than p, BW, and PAR_E, S_{BRAM} represents the size of a BRAM, that may be either 16Kb or 32Kb in Artix-7 FPGAs, while BW_{BRAM} represents the data bandwidth of a BRAM, that may be either 32 bits for 16Kb memories or 64 bits for 32Kb memories.

Equation (6) is the product of three terms. The first term $(PAR_E + 1)$ represents the number of p-bit memories. The second term $\left\lceil \frac{p}{S_{BRAM}} \right\rceil$ accounts for the number of BRAM memories required to store a p-bit polynomial. The third term $\left\lceil \frac{BW}{BW_{BRAM}} \right\rceil$ accounts for the number of BRAM memories necessary to provide the required BW data bandwidth.

4 Experimental Evaluation

This section discusses the area and performance of the proposed inversion architecture in order to highlight its efficiency and scalability. We note that we are not proposing a novel post-quantum QC-MDPC cryptosystem. In contrast, the proposed design methodology is meant to deliver an efficient and scalable hardware support for the binary polynomial inversion, thus accelerating QC-MDPC cryptosystems that employ it within their key generation procedure.

We adopted the LEDAcrypt-KEM-CPA [12] and BIKE [11] key encapsulation mechanisms as representative use cases to demonstrate the validity of the proposed architecture. The inversion module was implemented on all FPGAs of the mid-range Xilinx Artix-7 family, that offers the best price-performance ratio and is the target for the hardware assessment within the NIST PQC standardization process. Performance results of the proposed architecture are compared with two state-of-the-art software implementations running on an Intel Core i7 processor [39] and with a state-of-the-art hardware implementation targeting the Artix-7 FPGA family [38].

The rest of this section is organized in three parts. Section 4.1 overviews the LEDAcrypt and BIKE cryptosystems and their underlying codes. Section 4.2 details the experimental setup, encompassing hardware and software. Finally, Section 4.3 discusses the area and performance results.

4.1 LEDAcrypt and BIKE cryptosystems

We considered the BIKE [20] and LEDAcrypt-KEM-CPA [21] key encapsulation mechanisms as representative use cases for binary polynomial inversion. Both cryptosystems rely on the Niederreiter cryptoscheme [37] and employ a QC-MDPC code. BIKE was selected as an alternate proposal for the third round of the NIST PQC standardization process, while LEDAcrypt also participated to the competition. This part overviews the BIKE and LEDAcrypt code configurations with the goal of demonstrating the wide applicability of our inversion architecture.

TABLE 2: Polynomial length of the BIKE [20] and LEDAcrypt-KEM-CPA [21] cryptosystems.

Code	Security level	Polynomial length p
$L1.4$	AES-128	7187
$L1.3$		8237
$L1.2$		10883
$B1$		12323
$L3.4$	AES-192	13109
$L3.3$		15373
$L3.2$		21011
$B3$		24659
$L5.4$	AES-256	21611
$L5.3$		25603
$L5.2$		35339
$B5$		40973

The QC-MDPC codes underlying BIKE and LEDAcrypt-KEM-CPA feature parity-check matrices H composed of n_0 circulant blocks with size $p \times p$, where p is a large prime number and $n_0 \in \{2, 3, 4\}$ for LEDAcrypt-KEM-CPA, while n_0 is equal to 2 for BIKE. The block size p of the code corresponds to the bitlength of the polynomial which has to be inverted. Table 2 reports all the p code parameters for

BIKE and LEDAcrypt-KEM-CPA. They range from 7187 to 40973. The Bi and $Li.j$ labels refer to BIKE and LEDAcrypt-KEM-CPA instances where i refers to security levels 1, 3, and 5, that correspond to AES-128, AES-192, and AES-256, and j indicates the number of circulant blocks composing the H matrix. The experimental results detailed in the following were obtained for the code parameters in Table 2.

TABLE 3: Available resources on the Artix-7 family FPGAs.

FPGA	LUT	FF	BRAM
Artix-7 12	8000	16000	20
Artix-7 15	10400	20800	25
Artix-7 25	14600	29200	45
Artix-7 35	20800	41600	50
Artix-7 50	32600	65200	75
Artix-7 75	47200	94400	105
Artix-7 100	63400	126800	135
Artix-7 200	134600	269200	365

4.2 Experimental setup

Hardware setup - The inversion architecture discussed in Section 3 was described in SystemVerilog and it was implemented using the Xilinx Vivado 2018.2 hardware design suite. The experimental evaluation was carried out on the FPGAs from the mid-range Xilinx Artix-7 family, for which the available resources are detailed in Table 3.

Each design instance was implemented at a 133 MHz operating frequency. For each considered FPGA and code configuration, we only reported the best hardware implementation, i.e., the feasible one providing the best performance in terms of execution time for an inversion. Such instances were identified after an extensive design space exploration that considered a wide range of values for the configurable parameters of the architecture. We explored two bandwidths BW, 32 and 64 bits, three levels of multiplication parallelism PAR_M, with 1, 2, and 3 Karatsuba recursions computed in parallel, and a large set of levels of exponentiation parallelism PAR_E, with values equal to the powers of 2 between 1 and BW.

TABLE 4: Resource utilization, timing, and performance of the reference hardware instances of BIKE [38].

Code	BW	LUT	FF	BRAM	Freq.	Exec. time
$B1$	32	1776	342	3	100MHz	25.20ms
	64	4162	427	3	80MHz	8.88ms
	128	11721	733	6	74MHz	3.36ms
$B3$	32	1585	311	3	100MHz	110.02ms
	64	4366	493	3	83MHz	35.26ms
	128	12025	660	6	74MHz	12.04ms

The proposed architecture is compared to the reference inversion module extracted from the hardware implementation of BIKE, that targets FPGAs and is freely available online [38]. The state-of-the-art reference was implemented and simulated by employing Vivado 2018.2, targeting Artix-7 FPGAs and using the same synthesis and implementation directives as the ones used for our architecture. We considered only the reference instances implementing the third exponentiation strategy (see Section 2.2), since they show a lower or equal area and higher or equal performance than the other two [19]. The hardware reference implementation of BIKE is available in three bandwidths, i.e., 32, 64, and 128 bits, for the security levels 1 and 3. The instances with 32- and 64-bit bandwidth can be instantiated on an Artix-7 12 FPGA, i.e., the smallest Artix-7 chip, while the 128-bit instances must target an Artix-7 25 or larger FPGA due to the required LUT resources. Resource utilization, maximum clock frequency, and execution time for the reference hardware instances of BIKE are detailed in Table 4.

Software setup - We considered two reference software versions of binary polynomial inversion extracted from the implementation of LEDAcrypt-KEM-CPA. The `C11` version is used as the baseline reference design for performance evaluation, while the optimized software implementation employing the Intel `AVX2` extension is considered as the top-notch reference from the point of view of performance. Both are freely available online [39].

Both software versions were executed on an Intel Core i7-6700HQ CPU, forcing a fixed operating frequency of 3.5 GHz to avoid performance variability due to the power management controller. For each LEDAcrypt-KEM-CPA code configuration, the execution time of the inversion procedure for the `C11` and `AVX2` software implementations was obtained as the average of 30 executions.

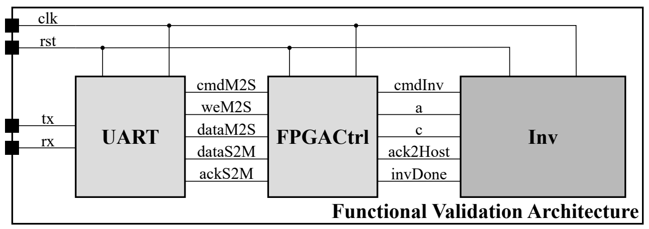
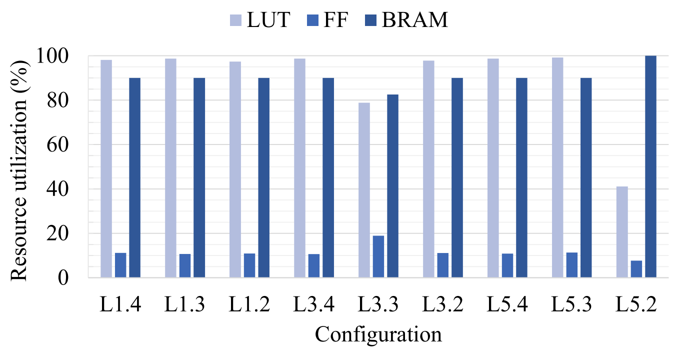


Fig. 6: Hardware setup for the functional assessment of the proposed polynomial inversion architecture.

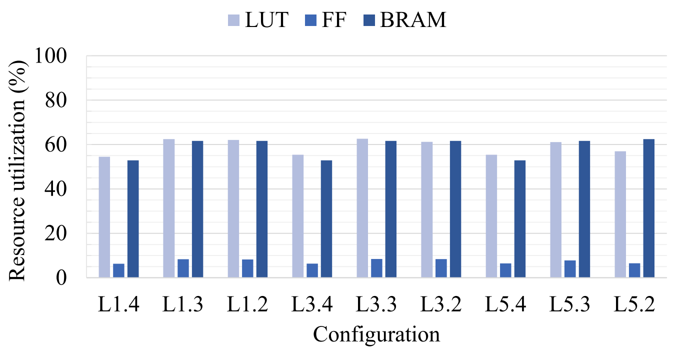
Functional validation - The proposed architecture was functionally validated through both post-implementation timing simulation and board prototype execution, checking the correctness of the obtained results against the inversion procedure extracted from the software implementation of LEDAcrypt-KEM-CPA [39]. For each LEDAcrypt KEM-CPA configuration, i.e., the nine $Li.j$ polynomial lengths reported in Table 2, we collected the results of the software execution of 10000 different inversion procedures.

Post-implementation simulation targeted the Xilinx Artix-7 12 (*xc7a12tcsg325-1*) and 200 (*xc7a200tsbg484-1*) FPGAs, while board prototype execution targeted the Digilent Nexys 4 DDR board, that features an Artix-7 100 (*xc7a100tcsg324-1*) FPGA. In both cases, we implemented a performance-optimized instance of our inversion module for each LEDAcrypt-KEM-CPA polynomial length and each target FPGA. Each inversion instance executed 10000 inversions, and their results were compared with the output of software execution.

Figure 6 describes the functional validation architecture used for both post-implementation simulation and prototype execution. It is made of three parts: the FPGA controller (`Ctrl`) communicates with the host computer to collect the input and return the output, the `UART` module creates a communication channel between the FPGA controller and the host computer, and the `Inv` block represents an instance of the proposed inversion architecture. To perform



(a) Xilinx Artix-7 12



(b) Xilinx Artix-7 200

Fig. 7: Resource utilization of the proposed inversion architecture implemented on the Xilinx Artix-7 12 and 200 FPGAs. The utilization for each resource type is expressed as a percentage of the available resources on the target FPGA.

an inversion operation, the `Ctrl` module drives the `cmdM2S` and `weM2S` signals to collect the input polynomial from the `UART` module. The FPGA controller waits until the UART has sent the required data before closing the communication, that implements a blocking protocol. Once the input has been collected, the `cmdInv` signal is used to load the operand $a(x)$ into `Inv` and to start the inversion. BW bits of $a(x)$ per clock cycle are passed to the inversion module through the `a` signal. The `Inv` module signals the end of the computation through the `invDone` control signal while BW bits of $c(x)$ per clock cycle are loaded into `Ctrl` through the `c` signal. The `Inv` and the `Ctrl` modules exchange data through an acknowledged protocol (see `cmdInv` and `ack2Host` signals). Finally, the `Ctrl` module sends the result back to the `UART` module through the `dataM2S` signal. The `Ctrl` and the `UART` modules also exchange data through an acknowledged protocol (see `cmdM2S` and `ackS2M` signals).

4.3 Experimental results

This section discusses the area and the performance of the proposed inversion template architecture, to demonstrate its efficiency and scalability across the entire Xilinx Artix-7 family of mid-range FPGAs.

Area results - The proposed architecture makes use of the BRAMs of the FPGA as the primary means of storage, allowing the inversion module to fit on tiny FPGAs even for codes with a large block size p. In such a way, the maximum allowed dimension of the dense vectors that store the input, intermediate, and output polynomials is not a function of the available amount of flip-flops, that easily become the scarcest resources on small FPGAs, but it is instead a function of the available BRAM storage capacity. We note that a single BRAM can store up to 36kb and the smallest Artix-7 FPGA features 20 BRAMs and 16000 flip-flops, while the considered polynomial lengths range from 7187 to 40973 bits.

Figure 7 reports the utilization of the look-up table (LUT), flip-flop (FF), and block RAM (BRAM) resources as a percentage of the total available resources on the Artix-7 12 and 200 FPGAs, for polynomial lengths that suit the nine LEDAcrypt-KEM-CPA cryptosystem configurations. Look-up tables are the most used FPGA resource in smaller designs fitting on Artix-7 12 FPGAs. Indeed, most best-performing designs that are still suitable for the smallest Artix-7 FPGA require up to 99% of available LUT resources, while used BRAMs are around 90-95%. Similarly, the majority of Artix-7 200 instances show a slightly higher utilization of LUTs than BRAMs. Regardless of the differences in used FPGA resources, all designs targeting the whole range of Artix-7 FPGAs are characterized by a wide usage of BRAMs, thus significantly minimizing the use of flip-flops. Even if the flip-flop utilization is low, it must be noted that the unused FF resources can not be exploited to further improve the design. For example, on average the FF utilization on the Artix-7 12 is below 15%, while the BRAM utilization is above 90% (see Figure 7a). However, an Artix-7 12 chip features 16000 FFs, thus its storage capacity is lower than a single BRAM and insufficient to store p-bit polynomials. In a similar manner, the FF utilization on Artix-7 200 is lower than 10% for each LEDAcrypt configuration. Even in such scenario, it is impossible to improve the design by leveraging the FF resources.

TABLE 5: Architectural parameters for hardware instances of the proposed architecture on Artix-7 12 and 200 FPGAs.

Code	Artix-7 12			Artix-7 200		
	BW	PAR_E	PAR_M	BW	PAR_E	PAR_M
$L1.4$	64	1	1	64	32	3
$L1.3$	64	1	1	64	64	3
$L1.2$	64	1	1	64	64	3
$B1$	64	1	1	64	64	3
$L3.4$	64	1	1	64	32	3
$L3.3$	32	16	1	64	64	3
$L3.2$	64	1	1	64	64	3
$B3$	64	1	1	64	64	3
$L5.4$	64	1	1	64	32	3
$L5.3$	64	1	1	64	64	3
$L5.2$	32	1	1	64	32	3
$B5$	32	1	1	64	32	3

In contrast, we identified two main limiting factors to a higher grade of parallelism. On the multiplier side, increasing PAR_M parallelism, i.e., implementing parallel computation of 4 or more Karatsuba recursions, demands a number of LUTs and BRAMs that is not available on any FPGA from the Artix-7 family. On the exponentiation side, a high level of PAR_E parallelism (which is nonetheless bounded by the BW bandwidth parameter) may cause timing closure at implementation time to fail, requiring to resort to instances with lower PAR_E that work at the target 133 MHz clock frequency. As shown in Table 5, configurations such as $L5.4$ have a PAR_E value equal to 32, while other ones such as $L5.3$ have a PAR_E value equal to 64, which is the maximum allowed value. A lower exponentiation parallelism results in around 4% and 8% lower LUT and BRAM utilization, respectively, on Artix-7 200 implementations.

TABLE 6: Execution times of C11 and AVX2 software [39] run on a i7-6700HQ CPU and of hardware instances of the proposed architecture on Artix-7 12 and 200 FPGAs.

Code	Software [39]		Proposed architecture	
	C11	AVX2	Artix-7 12	Artix-7 200
$L1.4$	1.80ms	0.20ms	1.18ms	0.10ms
$L1.3$	2.53ms	0.24ms	1.49ms	0.11ms
$L1.2$	4.46ms	0.35ms	2.10ms	0.16ms
$L3.4$	6.25ms	0.50ms	2.71ms	0.24ms
$L3.3$	8.11ms	0.78ms	8.56ms	0.28ms
$L3.2$	16.79ms	0.95ms	5.73ms	0.44ms
$L5.4$	19.58ms	1.24ms	6.09ms	0.51ms
$L5.3$	22.69ms	1.06ms	7.85ms	0.57ms
$L5.2$	49.95ms	2.43ms	47.61ms	1.11ms

Performance results - The performance assessment is achieved by comparing the execution time of the proposed inversion procedure to those of the software implementation of LEDAcrypt and the hardware implementation of BIKE.

In particular, Table 6 reports the performance results for all LEDAcrypt-KEM-CPA configurations, considering the two software references, i.e., `C11` and `AVX2`, and the two hardware instances of the proposed architecture that target the Artix-7 12 and 200 FPGAs. For example, `C11` takes between 1.80 ms and 49.95 ms to complete the inversion, with the two extremes corresponding to the $L1.4$ and $L5.2$ code configurations, respectively.

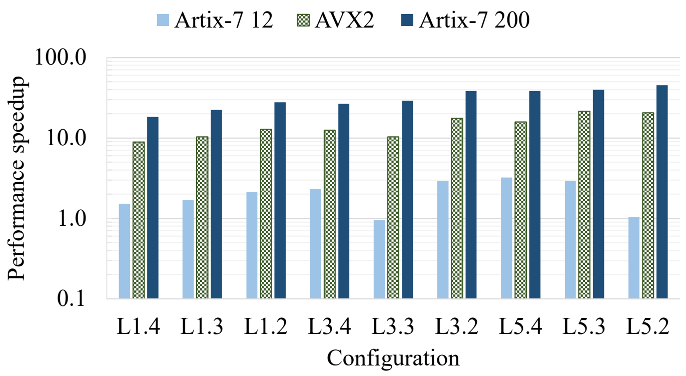


Fig. 8: Performance speedup with respect to C11 software inversion. Software inversion is executed on the i7-6700HQ CPU, while hardware inversion is implemented on the Artix-7 12 and 200 FPGAs.

Figure 8 reports the performance speedup of the AVX2 software and the two hardware implementations, normalized with respect to the `C11` software, highlighting the actual performance improvement across the different implementations of the inversion procedure. The performance speedup of the x implementation is defined as the ratio between the execution time of the C11 software (T_{C11}) and the execution time of x (T_x), where $x \in \{$`AVX2`, `Artix-7 12`, `Artix-7 200`$\}$, as shown in Equation 7.

$$speedup_x = \frac{T_{C11}}{T_x} \tag{7}$$

The inversion modules targeting the low-end Artix-7 12 FPGA show an execution time comprised between 1.18 and 47.61 milliseconds, with a performance speedup between 0.95 and 3.22 (2.08 on average). Notably, the only LEDAcrypt-KEM-CPA configuration for which Artix-7 12 performance is worse than C11 performance is $L3.3$, because of the reduced bandwidth BW due to area constraints (specifically, LUTs). The optimized `AVX2` software implementation shows a performance speedup ranging between 8.9 and 21.4 (14.5 on average), while our inversion modules targeting the Artix-7 200 FPGA show a performance speedup ranging between 18.3 and 45.2 (31.7 on average), compared to the `C11` reference. Moreover, our solution overcomes the `AVX2` software implementation by 2.2 times on average, thus demonstrating the superior capability compared to optimized software solutions that exploit custom instructions offered by recent high-end Intel processors.

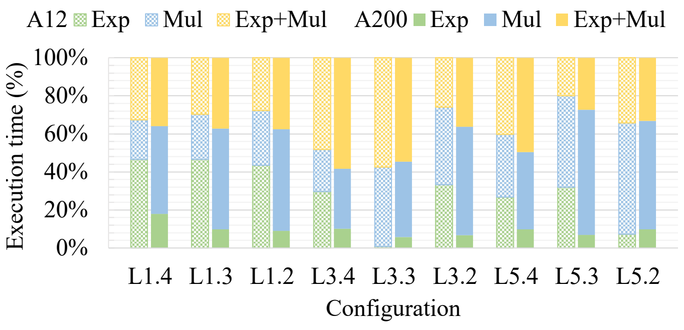


Fig. 9: Breakdown of the execution times of the macro-operations, for instances of the proposed architecture targeting the Artix-7 12 and 200 FPGAs.

Figure 9 shows the breakdown of execution times for the macro-operations scheduled within the inversion procedure, highlighting the time spent computing exponentiations, multiplications, and concurrent exponentiations and multiplications. For each configuration of the LEDAcrypt code, the left and right columns specify the breakdown of the execution times for instances of the inversion targeting the Artix-7 12 and 200 FPGAs, respectively. We note that, for each reported result, the corresponding architectural parameters and performance results are reported in Table 5 and Table 6, respectively. Figure 9 highlights a large fraction of the execution time spent in performing the concurrent execution of the multiplication and the exponentiation. Such value is comprised between 20% and 57% (35% on average) on Artix-7 12 and between 27% and 60% (41% on average) on Artix-7 200 instances. Considering the performance benefit due to the optimized hardware scheduling as well as its theoretical analysis detailed in Section 3, we note that the fraction of the execution time spent performing concurrent exponentiations and multiplications grows higher according to two factors. First, the ratio of 1s constituting the binary encoding of the $(p-2)$ value. Second, the difference in execution time between exponentiations and multiplications, that depends on the level of parallelism for each module.

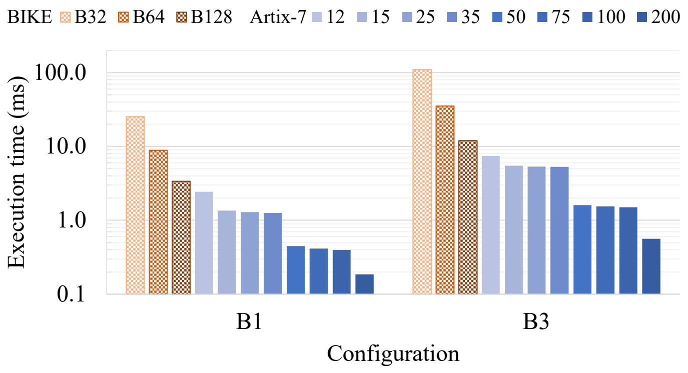


Fig. 10: Execution time of inversion. Results are shown for the reference hardware implementation and for instances of our template architecture on all Artix-7 FPGAs.

The performance achieved by the proposed architecture is also compared to the BIKE reference hardware. Figure 10 reports the execution time to complete the inversion procedure of BIKE for polynomial lengths required to implement AES-128 and AES-192 security, i.e., $B1$ and $B3$ in Table 2. We note that the reference hardware does not support the BIKE configuration with AES-256 security, thus we could not compare our architecture performance with respect to the $B5$ polynomial length. Results are reported for state-of-the-art hardware accelerators with 32-, 64-, and 128-bit bandwidth, and for instances of the proposed architecture targeting each FPGA of the Artix-7 family. We remark that the reference hardware instances of BIKE with 128-bit bandwidth only fit the Artix-7 25 and larger FPGAs. In contrast, each of our inversion instances was chosen to provide the best possible performance while satisfying the resource availability of all the target FPGAs using a 133 MHz operating frequency. Compared to the BIKE reference hardware, our solution provides a speedup ranging from 1.4 to 18.1 for $B1$ and between 1.6 and 21.5 for $B3$. The minimum and maximum speedup are achieved on the Artix-7 12 and 200 FPGAs, respectively, while the other instances of our scalable architecture provide a range of intermediate speedup values.

TABLE 7: Architectural parameters, resources, and performance of inversion instances that target the $B3$ code.

FPGA	BW	PAR_E	PAR_M	LUT	FF	BRAM	Exec. time
Artix-7 12	64	1	1	7954	1782	18	7.44ms
Artix-7 15	64	8	1	10180	2952	25	5.50ms
Artix-7 25	64	16	1	12568	4274	33	5.36ms
Artix-7 35	64	32	1	17291	6899	49	5.29ms
Artix-7 50	64	16	2	26787	7310	69	1.61ms
Artix-7 75	64	32	2	31547	9935	85	1.54ms
Artix-7 100	64	64	2	39319	14945	117	1.50ms
Artix-7 200	64	64	3	81928	24626	225	0.56ms

To further investigate the performance improvements, Table 7 reports the architectural parameters, resource utilization, and performance of the inversion instances on the Artix-7 FPGAs, considering the $B3$ polynomial length (see Table 2). The experimental results confirm that the higher time complexity of the multiplication with respect to the exponentiation (see Section 3.1 and Section 3.3) may suggest favoring the optimization of the former. For example, the execution time decreases from 1.50ms to 0.56ms by increasing the multiplication parallelism PAR_M from 2 to 3 (see lines Artix-7 100 and 200 in Table 7). However, results in Table 7 also highlight the critical contribution to the overall performance of inversion given by optimizing the exponentiation component. For instance, the execution time drops from 7.44ms to 5.29ms by increasing the exponentiation parallelism PAR_E from 1 to 32 (see lines Artix-7 12 and 35 in Table 7).

5 Conclusions

This manuscript presents an FPGA-optimized design methodology to implement efficient and scalable hardware support for polynomial inversion in $GF(2^m)$. The efficiency is achieved by means of *i)* an optimized computing architecture to perform polynomial multiplications and exponentiations and *ii)* an optimized scheduling infrastructure that enables the concurrent computation of the two operations whenever possible. The scalability is attained through a configurable architecture that scales from resource-constrained FPGAs up to larger chips that enable faster execution. We considered the LEDAcrypt and BIKE QC-MDPC post-quantum cryptosystems as representative use cases for the inversion of large binary polynomials. For each code configuration, our template architecture can deliver a performance-optimized inversion implementation while scaling across the whole Xilinx Artix-7 family of mid-range FPGAs. The experimental results demonstrate that the proposed architecture provides hardware support for inversion for polynomials with length of tens of thousands of bits even on the smallest FPGA of the Artix-7 family. Considering our implementation of inversion on the Artix-7 200 FPGA, the experimental results show an average performance improvement of 2.2 times across the full range of the LEDAcrypt configurations, when compared to the LEDAcrypt software implementation employing the Intel AVX2 extension. Compared to the state-of-the-art hardware implementation of BIKE targeting Xilinx Artix-7 FPGAs, the instances of our inversion template architecture show a performance improvement up to 18.1 and 21.5 times for AES-128 and AES-192 security levels, respectively.

References

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, p. 120–126, Feb. 1978. [Online]. Available: https://doi.org/10.1145/359340.359342
[2] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.
[3] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006*, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228.
[4] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 1994, pp. 124–134.
[5] National Institute of Standards and Technology (NIST) - U.S. Department of Commerce, "Post-quantum cryptography," https://csrc.nist.gov/projects/post-quantum-cryptography.
[6] A. Kipnis, J. Patarin, and L. Goubin, "Unbalanced oil and vinegar signature schemes," in *Advances in Cryptology — EUROCRYPT '99*, J. Stern, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 206–222.

[7] R. J. McEliece, "A Public-Key Cryptosystem Based on Algebraic Coding Theory," *DSN Progress Report*, pp. 114–116, 1978.

[8] J. Hoffstein, J. Pipher, and J. H. Silverman, "Ntru: A ring-based public key cryptosystem," in *Algorithmic Number Theory*, J. P. Buhler, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 267–288.

[9] E. Berlekamp, "Goppa codes," *IEEE Transactions on Information Theory*, vol. 19, no. 5, pp. 590–592, 1973.

[10] 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, 6th Int.'l Conference, SCN 2008, Amalfi, Italy, Sep. 10-12, 2008. Proc.*, 2008.

[11] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor, "BIKE website," https://www.bikesuite.org/.

[12] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt website," https://www.ledacrypt.org/.

[13] A. Barenghi, W. Fornaciari, A. Galimberti, G. Pelosi, and D. Zoni, "Evaluating the trade-offs in the hardware design of the ledacrypt encryption functions," in *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Nov 2019, pp. 739–742.

[14] D. Zoni, A. Galimberti, and W. Fornaciari, "Flexible and scalable fpga-oriented design of multipliers for large binary polynomials," *IEEE Access*, vol. 8, pp. 75 809–75 821, 2020.

[15] I. von Maurich and T. Güneysu, "Lightweight code-based cryptography: Qc-mdpc mceliece encryption on reconfigurable devices," in *2014 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–6.

[16] J. Hu, M. Baldi, P. Santini, N. Zeng, S. Ling, and H. Wang, "Lightweight key encapsulation using ldpc codes on fpgas," *IEEE Transactions on Computers*, vol. 69, no. 3, pp. 327–341, 2020.

[17] D. Zoni, A. Galimberti, and W. Fornaciari, "Efficient and scalable fpga-oriented design of qc-ldpc bit-flipping decoders for post-quantum cryptography," *IEEE Access*, vol. 8, pp. 163 419–163 433, 2020.

[18] K. Koleci, P. Santini, M. Baldi, F. Chiaraluce, M. Martina, and G. Masera, "Efficient hardware implementation of the ledacrypt decoder," *IEEE Access*, vol. 9, pp. 66 223–66 240, 2021.

[19] J. Richter-Brockmann, J. Mono, and T. Guneysu, "Folding bike: Scalable hardware implementation for reconfigurable devices," *IEEE Transactions on Computers*, 2021.

[20] N. Aragon, P. S. L. M. Barreto, S. Bettaieb, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, S. Gueron, T. Güneysu, C. A. Melchor, R. Misoczki, E. Persichetti, N. Sendrier, J.-P. Tillich, V. Vasseur, and G. Zémor, "BIKE: Bit flipping key encapsulation - round 3 submission," https://bikesuite.org/files/v4.1/BIKE_Spec.2020.10.22.1.pdf, 2020.

[21] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt: Low-density parity-check code-based cryptographic systems," https://www.ledacrypt.org/documents/LEDAcrypt_v3.pdf, 2020.

[22] A. Barenghi and G. Pelosi, "A comprehensive analysis of constant-time polynomial inversion for post-quantum cryptosystems," in *Proceedings of the 17th ACM International Conference on Computing Frontiers*, ser. CF '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 269–276. [Online]. Available: https://doi.org/10.1145/3387902.3397224

[23] H. Brunner, A. Curiger, and M. Hofstetter, "On computing multiplicative inverses in gf(2/sup m/)," *IEEE Transactions on Computers*, vol. 42, no. 8, pp. 1010–1015, 1993.

[24] K. Kobayashi, N. Takagi, and K. Takagi, "Fast inversion algorithm in gf(2m) suitable for implementation with a polynomial multiply instruction on gf(2)," *IET Comput. Digit. Tech.*, vol. 6, pp. 180–185, 2012.

[25] D. J. Bernstein and B.-Y. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 3, pp. 340–398, May 2019. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/8298

[26] T. Itoh and S. Tsujii, "A fast algorithm for computing multiplicative inverses in gf(2m) using normal bases," *Information and Computation*, vol. 78, no. 3, pp. 171–177, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0890540188900247

[27] N. Drucker, S. Gueron, and D. Kostic, "Fast polynomial inversion for post quantum qc-mdpc cryptography," in *Cyber Security Cryptography and Machine Learning*, S. Dolev, V. Kolesnikov, S. Lodha, and G. Weiss, Eds. Cham: Springer International Publishing, 2020, pp. 110–127.

[28] Wang, Troung, Shao, Deutsch, Omura, and Reed, "Vlsi architectures for computing multiplications and inverses in gf(2m)," *IEEE Transactions on Computers*, vol. C-34, no. 8, pp. 709–717, 1985.

[29] G.-L. Feng, "A vlsi architecture for fast inversion in gf(2/sup m/)," *IEEE Transactions on Computers*, vol. 38, no. 10, pp. 1383–1386, 1989.

[30] N. Takagi, J. Yoshiki, and K. Takagi, "A fast algorithm for multiplicative inversion in gf(2/sup m/) using normal basis," *IEEE Transactions on Computers*, vol. 50, no. 5, pp. 394–398, 2001.

[31] F. Rodriguez-Henriquez, N. Cruz-Cortes, and N. Saqib, "A fast implementation of multiplicative inversion over gf(2/sup m/)," in *International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II*, vol. 1, 2005, pp. 574–579 Vol. 1.

[32] F. Rodríguez-Henríquez, G. Morales-Luna, N. A. Saqib, and N. Cruz-Cortés, "Parallel itoh–tsujii multiplicative inversion algorithm for a special class of trinomials," *Designs, Codes and Cryptography*, vol. 45, no. 1, pp. 19–37, 2007.

[33] C. Rebeiro, S. S. Roy, D. S. Reddy, and D. Mukhopadhyay, "Revisiting the itoh-tsujii inversion algorithm for fpga platforms," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1508–1512, 2011.

[34] J. Hu, W. Guo, J. Wei, and R. C. C. Cheung, "Fast and generic inversion architectures over GF(2^m) using modified itoh–tsujii algorithms," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 4, pp. 367–371, 2015.

[35] L. Li and S. Li, "Fast inversion in gf(2m) with polynomial basis using optimal addition chains," in *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2017, pp. 1–4.

[36] J.-H. Guo and C.-L. Wang, "Systolic array implementation of euclid's algorithm for inversion and division in gf(2/sup m/)," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1161–1167, 1998.

[37] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Probl. Contr. and Inf. Theory*, vol. 15, 1986.

[38] Chair for Security Engineering @ Ruhr-Universität Bochum, "Folding bike: Scalable hardware implementation for reconfigurable devices," https://github.com/Chair-for-Security-Engineering/BIKE.

[39] LEDAcrypt, "Ledacrypt," https://github.com/LEDAcrypt/LEDAcrypt.

Andrea Galimberti, MSc, is a PhD student at Politecnico di Milano, Italy. He received his MSc degree in 2019 in Computer Science and Engineering at Politecnico di Milano. His research interests include computer architectures, hardware-level countermeasures to side-channel attacks and design of hardware accelerators.

Gabriele Montanaro, MSc, received his BSc degree in 2019 in Electronics Engineering and his MSc degree in 2021 in Electronics Engineering at Politecnico di Milano. His research interests target the digital design of complex hardware accelerators.

Davide Zoni, PhD, is assistant professor at Politecnico di Milano, Italy. He published more than 50 papers in journals and conference proceedings. His research interests include RTL design and verification of single- and multi-cores at the edge with emphasis on low power, hardware security, and deep learning. He filed two patents on cyber-security and he is co-founder at Blue Signals, a spin-off of the Politecnico di Milano.