AXI Hardware Accelerator for McEliece on FPGA Embedded Systems

Enrique Cantó-Navarro and Mariano López-García

***Abstract*—This article presents a McEliece hardware accelerator designed to be attached to an AXI infrastructure, addressing the efficient implementation of a flexible post-quantum cryptoprocessor on FPGA-based embedded systems. The complexity of the arithmetic circuits, combined with the adaptability to different applications by configurable parameters and run-time reprogramming, presents challenging issues for integrating the accelerator into these systems. The architecture of the accelerator is based on an application-specific instruction processor, which executes a set of constant-time instructions from an internal register file and memories. The role of the embedded processor is reduced to the initial writing of the instruction memory of the accelerator, the launching of the required set of instructions and configuring the Direct Memory Access controller to retrieve and store data from external memory. The run-time programming of the accelerator provides high flexibility in applications that requires post-quantum cryptography. A set of configurable parameters permits to adapt the security level of the McEliece encryption-decryption and the area-performance tradeoff imposed by the target device. Thus, the accelerator can be implemented from low-cost to high-end FPGAs by configuring the data-width of DMA buses or the parallelism level of the Galois-Field adder-multiplier. Experimental results show the accelerator is suited for implementing efficiently the highest security parameters of the Classic McEliece, achieving a McEliece decryption speed-up from x370 to x556 and occupying a small number of resources on a low-cost FPGA. In high-end FPGAs, the accelerator can be configured using higher security parameters not achieved in previous related cryptoprocessors, providing even higher accelerations.**

***Index Terms*—Public key cryptosystem, real-time and embedded systems, reconfigurable hardware.**

I. Introduction

ASYMMETRIC cryptography provides confidentiality on communications by distributing a public key from a partner which is used to encrypt messages from other parties, but they can only be decrypted by the paired private key. Public key (PK) cryptography is widely used in online services that ensure confidentiality, integrity, authentication, or non-repudiation. However, algorithms for quantum computing have been proposed to defeat the security of widely used PK algorithms, such as the RSA or those based on Elliptic-Curve Cryptography (ECC).

Post-quantum (PQ) algorithms applied on PK cryptography are believed to be secured against quantum computers. They can be classified into five categories: code-based, lattice-based, hash-based, based on multivariate-quadratic-equation, or based on supersingular isogenies of elliptic curves. Each category has its advantages and drawbacks that are under investigation, but generally, the complexity of PQ suffers from efficiency, which has promoted the development of hardware solutions to improve the execution time. Code-based cryptography, represented by the McEliece and its variants, is well-understood and provides reasonably efficient systems that have been developed.

Many implementations have been developed for the Classic McEliece (CMcE), which should not be confused with the original McEliece (McE) from 1978. Despite the name, the Classic McEliece is a NIST standard candidate for Key Encapsulation Mechanism (KEM), initially proposed in 2017 for the NIST competition. It is based on the Niederreiter algorithm, which is a syndrome-based dual variant of the McE. The CMcE replaces the Reed-Solomon (RS) as the error-correction mechanism by binary Goppa codes. Binary Goppa codes are believed to be post-quantum safe, whereas RS has been proven to be insecure. The CMcE provides an efficient KEM, mainly due to shorter keys, simpler key generation, and syndrome-based decoding, when compared with McE. Nevertheless, McE enables larger plaintext blocks (thousands of bits) than in Niederreiter or CMcE (hundreds of bits) using the same security parameters.

Previous works in hardware implementations for McE are quite scarce, whereas developments on CMcE are much larger due to the promising results for the NIST competition. Moreover, most of the previously presented systems are stand-alone cryptoprocessors on FPGAs built from very specific and fixed computational blocks, limiting their adaptability to applications other than KEM. They usually assume that keys are already loaded into internal FPGA memory, enhancing the processing speed by parallel accessing to keys and data. The connectivity with an embedded microprocessor and external memory, permitting changes of keys and algorithms, is not considered in their architecture and experimental results. FPGAs are widely used as prototyping platforms for digital systems. After validation, designs may be ported to ASIC technology if high-volume production is justifiable. However, FPGAs offer the possibility of updating designs if weaknesses are discovered. Furthermore, partially reconfigurable FPGAs permit the mapping of different

Manuscript received 24 July 2023; revised 23 July 2024; accepted 13 August 2024. Date of publication 16 August 2024; date of current version 14 March 2025. This work was supported by MCIN/AEI/ 10.13039/501100011033 under Grant PID2019-107274RB-I00. *(Corresponding author: Enrique Cantó-Navarro.)*

Enrique Cantó-Navarro is with the Electrics, Electronic and Automatic Department, University Rovira i Virgili, 43007 Tarragona, Spain (e-mail: enrique.canto@urv.cat).

Mariano López-García is with the Department of Electronics, Universitat Politécnica de Catalunya, 08800 Vilanova i la Geltrú, Spain (e-mail: mariano.lopez@upc.edu).

Digital Object Identifier 10.1109/TDSC.2024.3445181

accelerators, sharing the same hardware resources in a time-multiplexed manner.

The present work proposes a McE accelerator that provides interfaces to AXI buses to be integrated into an FPGA-based embedded system. The accelerator has been designed from scratch using a Hardware Description Language (HDL) to make low-level architectural decisions focused on reducing the clock cycles and hardware resources. It provides a set of instructions on binary vectors, matrices, Galois-Field $\mathbb{F}_{2^m}$ elements, and polynomials $\mathbb{F}_{2^m}[Z]$ to accelerate the steps typically used in McE encryption-decryption. Internal memories are dedicated to the storing of binary vectors and polynomials that can be read or written from an AXI-Stream interface.

The contributions are summarized as:

1. Hardware accelerator for McEliece that attaches to the AXI infrastructure, addressing limitations of previous works by providing run-time programmability and connectivity to external memory. It is suitable for security-sensitive applications, not restricted to KEM, such as biometric recognition.
2. Efficient use of hardware resources and significant performance improvements are achieved. It provides configurable parameters that allow the selection of the area-performance tradeoff suited to the target FPGA and security levels that overcome previous works.

The next section describes the main related works. Section III describes an overview of the architecture, internal memories, cores, and instruction set of the accelerator. Section IV describes the McE encryption and decryption focusing on the programming of the accelerator. Experiments results are reported and commented on in Section V. Finally, conclusions are presented.

II. Previous Works

Most publications, the cryptoprocessor connectivity with an embedded microprocessor and external memory is not considered. Therefore, the experimental results are theoretical execution times, assuming data and keys are already loaded in the FPGA. Additionally, hardware resources used as interface with external busses are not included. Most CMcE hardware implementations include the key generation since it is required in KEM cryptosystems. On the contrary, McE hardware implementations are devoted to accelerating PK encryption-decryption, relying on an external system to generate keys since it is an intensive task that runs only once for each of the parties and does not benefit as much from the hardware [1].

The system presented in [2] is a Niederreiter cryptoprocessor, which provides high-rate encryption-decryption on Virtex-6 FPGA, but security parameters are limited to 80-bit (n=2048, t=27, m=11). Internal Block RAM (BRAM) memories store the matrices of keys by row-wise addressing, computing an entire row per clock cycle to enhance the hypothetical performance. The reported simulation results show as encryption and decryption take 200 and $14.5{\cdot}10^3$ clock cycles, respectively, assuming that both the public and private keys are already stored in the internal BRAMs. However, this is an unrealistic test since keys must be able to change, and the authors recognize that there is an open problem with the interface's ability to transfer keys into the FPGA memory at such high rates.

Some works optimize the C-code of software implementation of the CMcE, such as the McBits [3], [4] and the specific ARM Cortex-M4 implementation [5]. The authors describe several alternatives to replace internal algorithms and to vectorize calculations, thereby providing efficient execution on microprocessors with SIMD instructions.

The hardware CMcE implementations showed in [6], [7], [8], [9] are based on algorithms taken from the previously commented software optimizations. They presented a cryptoprocessor built from very specialized blocks targeted to accelerate the steps involved in the KEM application. The architectures are described in Verilog, and they are very efficient in execution time for CMcE, but not as much in hardware resources. For instance, the time-optimized version for the highest security level (n=8912, t=128, m=13) [8] reports $123{\cdot}10^3$ LookUp Tables (LUTs), $190{\cdot}10^3$ Flip-Flops (FFs), 589 BRAMs on a high-end Virtex-7, taking $1.28{\cdot}10^6$ clock cycles to generate a key and $6.5{\cdot}10^3$ and $26.2{\cdot}10^3$ clock cycles for encryption and decryption, respectively, assuming data is not read or written to external memory. The area-optimized version reduces the number of FPGA resources to $45{\cdot}10^3$ LUTs and $88{\cdot}10^3$ FFs but still requires a large number of 525 BRAMs. The number of clock cycles is the same for encryption and increases to $48.8{\cdot}10^3$ for decryption. The system [9] accelerates the matrix systemizer during the key generation by processing a portion of the matrix. It is based on the systemizer of [10], but permitting the early aborting when it detects that the matrix is not systemizable. The large number of BRAMs does not allow its implementation on low-cost FPGAs, lacking the flexibility to be adaptable to other applications different than KEM since it does not provide external memory access shared with an embedded microprocessor.

A hardware-software (Hw/Sw) codesign for the CMcE is presented in [11] on a Zynq-UltraScale+ device, a high-end programmable system on chip. The algorithm was described by the C-based High-Level Synthesis (HLS) to get the optimal Hw/Sw partitioning based on the execution times. The HLS synthesises the hardware accelerator and AXI interfaces to the embedded microprocessor and DMA. The experimental results are reported in relative values and percentage of FPGA resources. The execution time includes the overhead of transferring keys and data between external memory and BRAMs. For the highest security level (n=8192, t=128, m=13) the acceleration for key generation reaches x55.2, x3.3 for encoding and x8 for decoding. The total number of devoted FPGA resources is very large, taking about 80% ($219{\cdot}10^3$) of the total available LUTs, 62% ($340{\cdot}10^3$) of the FFs, 38% (958) of the DSPs, and 205% (1870) of BRAMs. Although authors recognise that fully-hardware implementations can naturally surpass the proposed system, the Hw/Sw codesign provides an end-to-end KEM implementation and it has the potential to be adapted to future changes.

The CMcE cryptosystem [12] was implemented in SpinalHDL, improving the execution time and drastically reducing the hardware resources compared to the previously mentioned work for the same security parameters (n=8192, t=128, m=13). However, it does not provide interfaces with a microprocessor

bus and external memory, limiting its adaptability. The experimental results report $1.2{\cdot}10^6$ clock cycles for key generation, $4.5{\cdot}10^3$ and $36.5{\cdot}10^3$ for encryption and decryption, devoting $45.7{\cdot}10^3$ LUTs, $39.6{\cdot}10^3$ FFs, 183 BRAMs on a high-end Virtex UltraScale+.

In general terms, HLS improves productivity by a faster development cycle by isolating the developer from low-level details and decisions. On the other hand, hardware description languages (HDL), such as VHDL or Verilog, require significant hardware design knowledge, but they can be targeted to provide higher performance on lower resources. It can be found research works that embrace HLS versus HDL or vice versa [13], [14]. The efficiency of the HDL design depends on the skill level and development effort, but it has the potential to create more efficient designs than HLS.

A much smaller number of implementations have been developed for the McE encryption-decryption, and most are standalone designs. Moreover, some of them replace the Goppa codes with other codes to improve the implementation efficiency.

The work [10] was the first implementation of McE on FPGA. It is composed of three custom circuits: key generator, encryptor, and decryptor based on the Patterson algorithm. The security parameters were fixed to a low-security level (n=2048, t=50, m=11) on a high-end Virtex-5 FPGA, integrated into a PCI card on a host personal computer (PC). The encryption and decryption take 0.5 ms and 1.29 ms, respectively, including communications with the host PC, achieving x20 and x42 speedup factor at 164 MHz when compared with the software implementation. The cryptoprocessor consumes a significant amount of FPGA resources: 84% of available slices and 50% of BRAMs.

Another work [15] presents a lightweight McE encryptor and decryptor on FPGA, which replaces Goppa codes with quasi-cyclic moderate density parity-check codes (QC-MDPC). The security level provided is limited to 80-bit, and the theoretical encryption-decryption takes 3.4 ms ($73.5{\cdot}10^3$ clock cycles) and 23 ms ($4.27{\cdot}10^6$ clock cycles), employing just 64 and 159 slices on a low-cost Spartan-6 device. The authors claim QC-MDPC provides a very efficient encryption-decryption, but further cryptanalysis for weakness should be investigated. They also recognize encoding and decoding these codes is very different from Goppa codes and cannot be fairly compared with other McE hardware systems.

The McE decryptor shown in [1] is scalable from 80-bit to 256-bit security. It was developed in VHDL, and it can operate, not only with Goppa codes, but also with quasi-dyadic (QD) Goppa codes. QD-Goppa codes provide simpler calculations, but some parameters of these codes have been weakened by cryptoanalysis. The highest security level achieved for Goppa codes is 128-bit (n=3307, t=66, m=12), taking $27{\cdot}10^3$ clock cycles to decrypt at the highest performance and using 16153 LUTs, 2603 FFs, and 25 BRAMs on a low-cost Spartan-3AN. The cryptoprocessor cannot encrypt and does not provide connectivity to external memory, assuming the keys are already loaded in the FPGA programming bitstream.

A quite different approach is the Galois-Field (GF) instruction set extension for RISC-V processors [16], which supports specific computing instructions. The experimental results show encryption and decryption take $4.33{\cdot}10^6$ and $44.2{\cdot}10^6$ clock cycles, respectively, reducing by 8% and 75% the execution time for the CMcE (n=3488, t=64, m=12) running on a RISC-V at 25 MHz. Although this approach provides high flexibility to be used in several applications, not only for CMcE, its performance is far from that obtained by the previous custom cryptoprocessors.

The present work proposes a McE run-time programmable accelerator that provides AXI buses for integration into embedded systems. The design was initially based on the hardware-software codesign presented in [17], which uses 77357 LUTs and 41338 FFs and accelerates decryption by x21 (47.4 ms). The presented accelerator has been redesigned from scratch in VHDL, permitting better control of hardware resources and required clock cycles for computations. It provides a set of constant-time instructions to accelerate computations on Goppa codes and binary vectors and matrices. The microprocessor can write at run-time the set of instructions to accelerate encryption or decryption of the McEliece algorithm, or even other algorithms. Moreover, the accelerator provides a set of parameters that allow configuring the security level and the area-performance tradeoff suitable to the target device.

The McE accelerator can be used in classical applications of PK cryptosystem, such as the encryption/decryption of messages, the exchange of session keys, and digital signatures. Moreover, PK can also be used in blockchain and biometric applications. For instance, the iris-code digital signatures based on RSA[18] or the distributed biometric authentication secured by ECC-based blockchain [19]. A few works focus on biometrics using McE cryptosystems, such as the protection of templates [20] or the non-device-centric facial authentication [21]. In the latter, the authors propose performing the matching in the encrypted domain, providing quantum resistance to the biometric templates and featuring irreversibility, unlinkability, and revocability. The high flexibility provided by the proposed McE accelerator makes it suitable for these applications.

III. Architecture

The accelerator is an application-specific instruction processor. The set of instructions are related to the one of the two provided arithmetic cores: the polynomial $\mathbb{F}_{2^m}[Z]$ unit and the vector core, as shown in Fig. 1. The polynomial $\mathbb{F}_{2^m}[Z]$ arithmetic core is connected to a register file to retrieve or save polynomials. The vector core devotes an internal memory to store the resulting binary vector from a computation involving a source vector or matrix retrieved from external memory. The instructions are stored in an internal memory, which can be programmed at run-time to change the operations launched to the cores. The accelerator provides AXI-Stream interfaces to transfer data efficiently between the external RAM and the internal memories through an AXI-DMA. Finally, the Slave-AXI interface permits the control of the accelerator from a microprocessor, setting up the AXI-Stream interfaces to retrieve or store data into the internal memories, launching a computation, and retrieving the status of the cores.

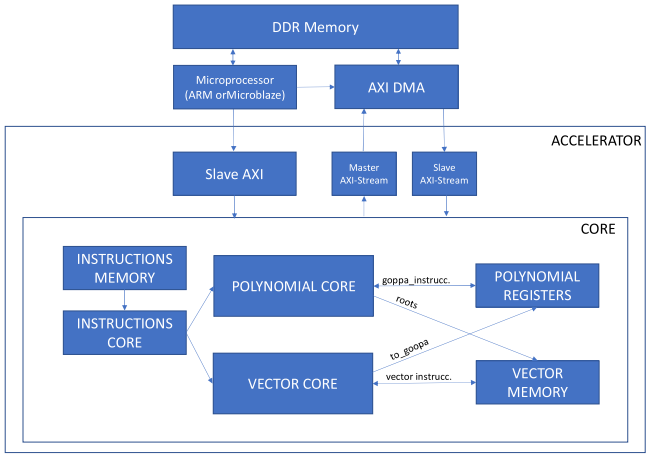


Fig. 1. Overview of the architecture. .

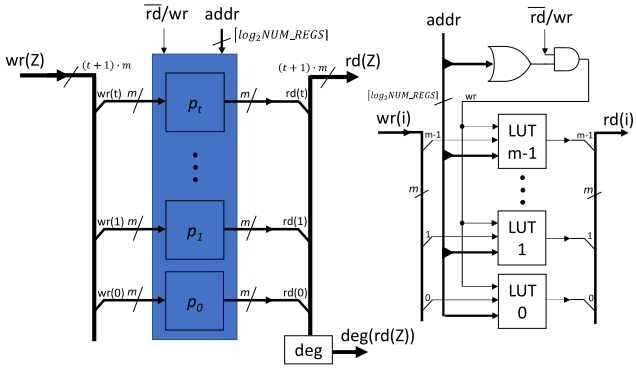


Fig. 2. Distributed memory for the polynomial registers.

TABLE I
IRREDUCIBLE POLYNOMIALS IN GALOIS FIELD

m	$g(x)$
8	$x^8 + x^4 + x^3 + x + 1$
9	$x^9 + x + 1$
10	$x^{10} + x^3 + 1$
11	$x^{11} + x^2 + 1$
12	$x^{12} + x^3 + 1$
13	$x^{13} + x^4 + x^3 + x + 1$
14	$x^{14} + x^5 + 1$
15	$x^{15} + x + 1$

The security parameters (n_{MAX}, m, t) of the McEliece are configured at synthesis time and they affect the polynomial core and its registers. Indeed, the error-correction capability (t) can be any valid value, and the dimension (m) of the Galois-Field $\mathbb{F}_{2^m}$ can be selected from 8 to 15. The code length (n) is programmable at run-time accomplishing $n \leq n_{\text{MAX}}$, where n_{MAX} is a configurable parameter that affects the vector core and memory.

Another set of parameters is used to configuring the depth of the instruction's memory (NUM_INST) and the number of polynomial registers (NUM_REGS). It is recommended that both parameters are configured as powers of two for better use of the internal memories. The default values NUM_INST=256 and NUM_REGS=16 are enough for implementing the McE encryption and decryption.

Finally, the area-performance tradeoff can be managed by selecting the appropriate values for a set of parameters. The width of the AXI-Stream interfaces (AXIS_WIDTH) is selectable to 32/64/128/256-bit. Depending on the external DDRx memories and connection with the FPGA, a higher AXIS_WIDTH improves the read/write throughput but requires higher parallelization and resources. This parameter affects the vector core since 64-bit to 256-bit width data from matrices must be aligned in several 32-bit words to perform properly in parallel. Another parameter is the parallelism factor of the Galois-Field $\mathbb{F}_{2^m}$ adder-multiplier (GF2MULT_FACTOR), which affects the polynomial core.

A. Register File for Polynomials

The architecture provides a register file to read polynomial operands and write results of instructions, eliminating accesses to external DDR memory to perform a computation. Each register stores a polynomial $p(Z) \in \mathbb{F}_{2^m}[Z]$, which is composed of $t+1$ coeficients in Galois-Field $\mathbb{F}_{2^m}$, following the notation $p(Z) = p_t Z^t + \ldots + p_1 Z + p_0$. The register file is a single-port distributed memory since it can be efficiently implemented on LUTs. Fig. 2 shows the architecture of the memory, which provides a single address bus to read/write one of the registers. The degree $\deg(p(Z))$ of a polynomial is the highest index with non-zero coefficient, and it is computed when a polynomial is read. The number of registers is configurable, and in our case, we took NUM_REGS=16 by default, devoting a single LUT to store one bit of one of the coefficients for all the registers. This way, all the registers share the same set of $(t+1) \cdot m$ LUTs.

The registers are named from r0 to r15, permitting polynomial instructions to read operands and store the results in any of these registers. There are two special-purpose registers. The r0 is a read-only register, which allocates the zero polynomial, and it can be used by the instructions as an input operand or to avoid storing a result into the register file. The register r15 stores the irreducible polynomial $g(Z) \in \mathbb{F}_{2^m}[Z]$, which is used by the core when reducing a polynomial. The AXI interfaces provide have access to the polynomial registers from the microprocessor and external memory to read results or to change the private $g(Z)$ at run-time.

B. Polynomial Core

The polynomial core is built around a paralleled $\mathbb{F}_{2^m}$ adder-multiplier. It is composed of $t+1$ independent combinational $\mathbb{F}_{2^m}$ arithmetic circuits, $m_i(y) = op1_i(y) \cdot op2_i(y) + opx_i(y)$, $0 \leq y \leq t$. The reduction polynomial $g(x) \in \mathbb{F}_{2^m}$ is fixed according to Table I.

The $\mathbb{F}_{2^m}$ adder-multiplier can perform $t+1$ addition-multiplications at every clock cycle by attaching three inner registers (po, pq, pb), as depicted in Fig. 3. Each inner register is built of a large set of $(t+1) \cdot m$ flip-flops (FFs) and stores all the $\mathbb{F}_{2^m}$ coefficients of a polynomial $\mathbb{F}_{2^m}[Z]$. Large bit-width

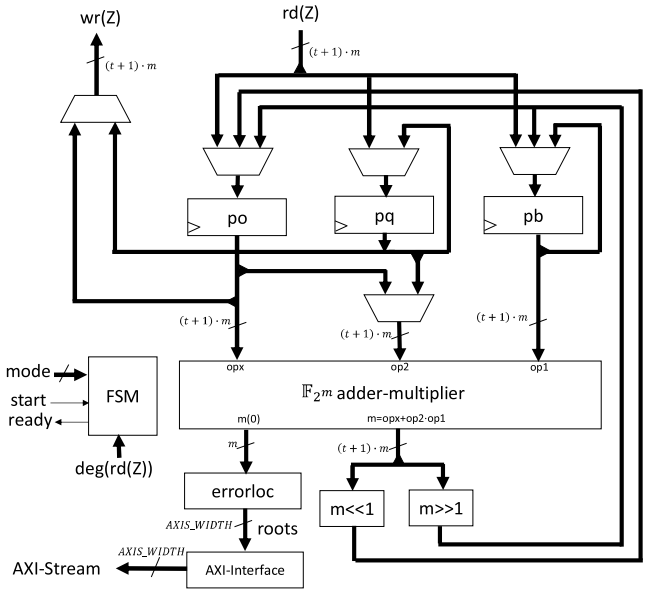


Fig. 3. Overview of the polynomial core..

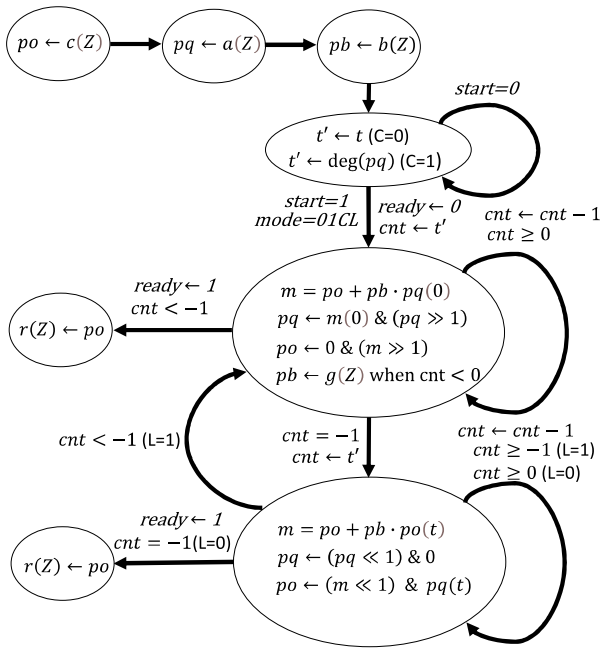


Fig. 4. States of the polynomial operation $r(Z) = a(Z) \cdot b(Z) + c(Z)$.

multiplexers connect the polynomial register file, the inner registers, and adder-multiplier in several ways, depending on the running state. The registers and multiplexers are controlled by a Finite State Machine (FSM), which changes the states based on the starting mode (encoded in the instruction) and an internal counter (cnt). The start and ready ports are used for handshaking with the external circuitry. The number of inputs of the large multiplexers has been minimized since each one is composed of $(t+1) \cdot m$ bits. The output of the $\mathbb{F}_{2^m}$ adder-multiplier (m) is shifted one coefficient to the left ($m << 1$) or to the right ($m >> 1$) before storing it in an inner register.

An inner register p can be written from a polynomial $a(Z) \in \mathbb{F}_{2^m}[Z]$, which is read from the register file in a single clock cycle, denoted as $p \leftarrow a(Z)$. The registers can be shifted one coefficient to the left or right, denoted as $p << 1$ or $p >> 1$ that are equivalent to $a(Z) \cdot Z$ or $a(Z) \cdot Z^{-1}$. Operations over polynomials can be accelerated by the paralleled $\mathbb{F}_{2^m}$ adder-multiplier, and by 1-bit shifting on registers to avoid much more complex barrel shifters. The steps carried out on the po, pq, pb registers and the $\mathbb{F}_{2^m}$ adder-multiplier are kept as similar as possible for the different instructions to avoid incrementing the number of inputs of multiplexers.

For instance, instruction $r8 \leftarrow r7 * r4 + r12$ computes a polynomial addition-multiplication from three registers, returning the result in the fourth register. Concretely, it computes $r(Z) = a(Z) \cdot b(Z) + c(Z)$ in four states, as depicted in Fig. 4.

The instruction starts from an initial idle state by writing the three inner registers from polynomials stored in the register file. Due to the single-port register file, $po \leftarrow c(Z)$, $pq \leftarrow a(Z), pb \leftarrow b(Z)$ require three clock cycles. A fourth cycle is required to start the operation, deasserting the ready port and decoding the operation mode, which is encoded as 01CL (C=0, L=1) for the modular multiplication and addition of polynomials.

In the second state, the $\mathbb{F}_{2^m}$ adder-multiplier performs $po + pb \cdot pq(0)$ and the result is 1-bit shifted to the right ($m >> 1$) before storing it into two registers. The pq is right-shifted, and the lowest coefficient of the result is appended at the left side, denoted as $pq \leftarrow m(0)$ & $(pq \gg 1)$. Concurrently, the register po stores the remaining coefficients of m, appending the coefficient 0 to the left side, according to $po \leftarrow 0$ & $(m \gg 1)$. The state is repeated $t+1$ clock cycles to compute the not reduced result $r'(Z)$. After completing, pq stores the lowest $(t+1)$ coefficients, and po stores the remaining highest t coefficients at the right side. In other words, $\{po, pq\} = r'(Z)$, with $po(t) = 0$. The register pb remains constant, except at the last cycle that loads the irreducible Goppa polynomial $g(Z)$ from the register file, denoted as $pb \leftarrow g(Z)$.

The third state reduces $r'(Z)$ by performing $po + pb \cdot po(t)$, adding to the register po the multiplication of its highest coefficient $po(t)$ with $g(Z)$. The result m is left-shifted before storing it into po, appending the left size coefficient stored in pq, according to $po \leftarrow (m \ll 1)$ & $pq(t)$. Therefore, the degree of the result is reduced when $po(t) \neq 0$ or the result is simply left-shifted when $po(t) = 0$. The reduction state is repeated $t+2$ clock cycles, since at the first cycle $po(t) = 0$. After completing the rest of cycles, $pq = 0$ and po stores the result $r'(Z) \mod g(Z) = r(Z)$ but left-shifted as $po = r(Z) << 1$.

Since $po = r(Z) << 1$ and $pq = 0$, the second state is executed once again to perform the right-shifting of po, finally obtaining $po = r(Z)$ where $deg(r(Z)) < t$. The final state transfers the instruction result, which is stored in po, to the register file and asserts the ready port. Therefore, the total number of clock cycles devoted is $5 + 2 \cdot (t+2)$, since $2 \cdot (t+2)$ cycles are required for the second and third states and 5 clock cycles for the initial and final states.

The previous addition-multiplication instruction works for any input polynomials, resulting in $deg(r(Z)) < t$. In the case of being $a(Z)$ a constant polynomial, the mode changes to 011L (C=1), which slightly changes the execution of the state graph to perform faster, depending on the $deg(a(Z))$. Additionally, the last reduction and right-shifting can be avoided if L=0, resulting in $deg(r(Z)) \leq t$. For instance, assuming $r1$ stores the polynomial $a(Z) = 1$, the reduction of $b(Z)$, which is stored in r7, can be computed by the instruction $r5 \leftarrow r1 * r7 + r0$, taking $5 + 2 \cdot (deg(a(Z)) + 2)$=9 clock cycles since $deg(a(Z)) = 0$ and L=1. Another example, if $r12$ stores $c(Z)$, the addition $b(Z) + c(Z)$ is stored in $r5$ ($deg(r5) \leq t$ when L=0) by executing the instruction $r5 \leftarrow r1 * r7 + r12$, taking $5 + 2 \cdot (deg(a(Z)) + 1)$ =7 clock cycles. A final example is the operation $b(Z) \cdot Z$ by the instruction $r14 \leftarrow r2 * r7 + r0$, assuming $r2$ stores the constant polynomial $a(Z) = Z$, which takes $5 + 2 \cdot (deg(a(Z)) + 1) = 9$ clock cycles since L=0.

Obviously, the described procedure for the addition-multiplication could be reduced by 2 clock cycles by disabling the shifting on m and on the registers po, pq at the last cycle of the second and third states. However, this solution increments significantly the hardware resources due to the additional input inferred on the multiplexers to attach m. Another way to reduce clock cycles is by simultaneously writing to the three inner registers using a three-port memory for the register file. Considering the parameter t is a large number, reducing a few clock cycles does not have a great impact on the overall performance, although it significantly increments the hardware resources. The traditional way to multiply and reduce at each step requires a fourth inner register storing $g(Z)$ during all the cycles, and an additional input at one of the multiplexers. However, the execution time cannot be reduced since a single m can be performed at every clock cycle, and therefore, this solution only provides disadvantages.

The operation $v = roots(c(Z), N)$, which is used to find the roots of the polynomial, is slightly different from others because the result is a binary vector $\mathbb{F}_2^N$ that will be stored in the vector memory (v). The erroloc circuit depicted in Fig. 3 is a simple m-bit comparator attached to a register, which connects to the AXI-Stream bus to share the same interface used to write internal memories. For instance, the instruction $roots(r8,\ 8192)$ computes the roots of the polynomial stored in the register $r8$, resulting in an 8192-bit binary vector.

At the idle state, the instruction initializes the inner register po with the polynomial $c(Z)$, and loads pq with $r0$ to initialize all its coefficients to zero, as shown in Fig. 5. Then, the internal FSM register max is initialized to 8191 to evaluate $c(Z)$ from $Z = 0$ to $Z = 8191$. Since the $\mathbb{F}_{2^m}$ adder-multiplier is composed of independent arithmetic blocks, it can concurrently compute $t + 1$ partial evaluations of $c(Z)$ at each clock cycle. The counter cnt is the Z that is going to be evaluated, requiring $(t + 1)$ clock cycles to complete. As shown in the second state, by right shifting pq and writing cnt at the left side, the register stores $pq(t) = Z, pq(t-1) = Z-1, pq(t-2) = Z-2, \ldots, pq(0) = Z - t$. Since pb stores $m >> 1$ with $pb(t) = 0$ at each clock cycle, the results from the $\mathbb{F}_{2^m}$ adder-multiplier are $m(t) = c_t + Z \cdot 0 = c_t$, $m(t-1) = c_{t-1} + (Z-1) \cdot c_t$, $m(t-2) = c_{t-2} + (Z-2) \cdot (c_{t-1} + (Z-2) \cdot c_t)$, and so on. During the first t clock cycles, the $m(0)$ does not provide a valid value, but after this initial latency $m(0) = c_0 + Z \cdot (c_1 + Z \cdot (c_2 + \cdots (Z \cdot c_{t-1} + c_t) \cdots))$, which is the evaluation $c(Z)$ by the Horner's method. The evaluations $m(0) = c(Z)$, from $Z = 0$ to $Z = N - 1$ are obtained in sequential order, being the throughput of one evaluation per clock cycle. Therefore, the total computation time, including the initial state, is $3 + (t + 1) + N$ clock cycles. The errorloc circuit compares $m(0)$ with $0 \in \mathbb{F}_{2^m}$, resulting in a single bit, which is serially stored in a shift register to interface the AXI-Stream bus that writes the vector memory.

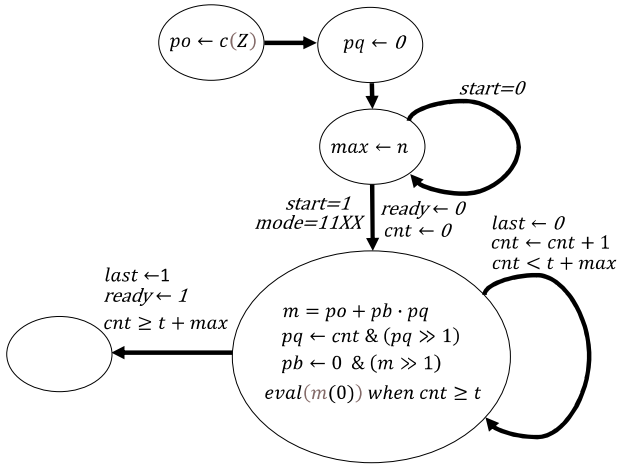


Fig. 5. States of the operation $v = roots(c(Z), N)$.

As it is shown in the previous operations, the procedures taken in the states are very similar. Moreover, all the instructions that can be performed by the polynomial core (Table II) follow a similar approach. All of them reuse the three inner registers and connections to the $\mathbb{F}_{2^m}$ adder-multiplier through the same multiplexers, saving FPGA resources. The main difference is the states carried out by the FSM according to each instruction.

The copy instruction just copies a register into another one. The division $a(Z)/b(Z)$ returns two polynomials accomplishing $q(Z) \cdot b(Z) + r(Z) = a(Z)$. Similarly, the $split(a(Z))$ returns two polynomials composed of the even $e(Z)$ and odd $o(Z)$ coefficients. The GF Mult instruction is the Galois Field $\mathbb{F}_{2^m}$ multiplication of the composing coefficients $r_i = a_i \cdot b_i (0 \leq i \leq t)$. These last two instructions can be used to compute the $\sqrt{a(Z)}$. Finally, the last instruction returns by the AXI interface the degree of the polynomial stored in a register.

The polynomial core devotes a large number of resources to the parallel $\mathbb{F}_{2^m}$ adder-multiplier and its associated multiplexers. The fully parallelized core does not fit in low-cost devices for the larger security parameters (m, t), but the core provides the parameter GF2MULT_FACTOR (denoted as F), which defines the parallelism factor. The number of paralleled arithmetic circuits in the $\mathbb{F}_{2^m}$ adder-multiplier is reduced to $\lceil (t+1)/F \rceil$ ($F \geq 2$), and the width of inputs of the multiplexers depicted in Fig. 3 is reduced to $m \cdot \lceil (t+1)/F \rceil$. The instructions execute the same states described before, but each cycle is divided into F clock cycles. Each of the inner registers are divided into

TABLE II
POLYNOMIAL INSTRUCTION SET

	Polynomial operation Instruction example	Time (T_{CLK})
Add-Mul	$r(Z) = a(Z) \cdot b(Z) + c(Z)$ $r4 \leftarrow r6 * r11 + r10$	$5 + F \cdot 2 \cdot (t + 2)$
Mult	$r(Z) = a(Z) \cdot b(Z)$ $r4 \leftarrow r6 * r11 + r0$	$5 + F \cdot 2 \cdot (t + 2)$
Module	$r(Z) = b(Z)\ mod\ g(Z)$ $r4 \leftarrow r1 * r11 + r0$	$5 + F \cdot 4$
Add	$r(Z) = b(Z) + c(Z)$ $r4 \leftarrow r1 * r11 + r10$	$5 + F \cdot 2$
Mult Z	$r(Z) = b(Z) \cdot Z$ $r4 \leftarrow r2 * r11 + r0$	$5 + F \cdot 4$
Division	$\{r(Z), q(Z)\} = a(Z)/b(Z)$ $\{r4, r3\} \leftarrow r6/r11$	$6 + F \cdot 2 \cdot (t + 1) + F$
Roots	$v = roots(c(Z), N)$ $roots(r6, N)$	$3 + F \cdot (t + 1 + N)$
Copy	$r(Z) = a(Z)$ $r4 \leftarrow r6$	3
GF Mult	$r_i = a_i \cdot b_i\ (0 \le i \le t)$ $r4 = GF(r6 * r11 + r0)$	$5 + F \cdot 2$
Split	$\{e(Z), o(Z))\} = split(a(Z))$ $\{r4, r3\} \leftarrow split(r6)$	$4 + F \cdot 2 \cdot (t + 1)$
Degree	$deg(a(Z))$ $deg(r3)$	4

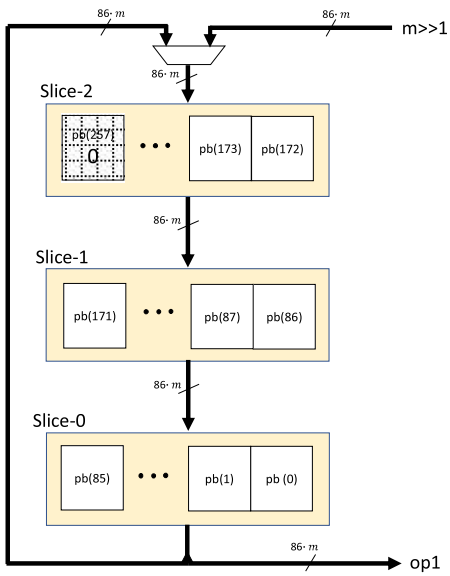


Fig. 6. Slices for the inner register pb for t=256 and F=3.

F slices of $\lceil (t+1)/F \rceil$ coefficients, which are continuously rotated. Fig. 6 shows the slices of the register pb for the case $t = 256$ and $F = 3$. Each slice stores 86 coefficients, padding $pb(j) = 0$ when $j > t$. The slice-0 attaches to the $op1$ port of the $\mathbb{F}_{2^m}$ adder-multiplier. In the next clock cycle the result $m \gg 1$ is stored in the slice-2, and the contents of slices are rotated to perform with the next coefficients, maintaining the same connectivity with the $\mathbb{F}_{2^m}$ adder-multiplier. After completing

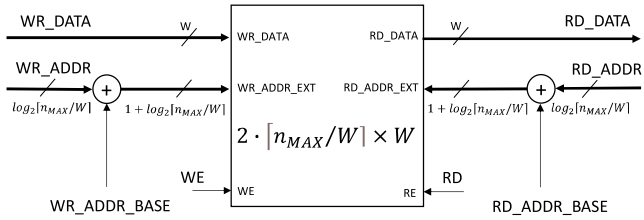


Fig. 7. Vector memory.

the F clock cycles, the inner registers store the same contents as in a single cycle of the fully parallelized core. The obvious inconvenience is the clock cycles needed by the instructions carried out by the $\mathbb{F}_{2^m}$ adder-multiplier are incremented by the factor F. For instance, the instruction for addition-multiplication of polynomials takes $5 + F \cdot 2 \cdot (t + 2)$ clock cycles, or the instruction for finding roots takes $3 + F \cdot (t + 1 + n)$.

C. Vector Memory

The maximum number of bits that can be allocated in the vector memory is a configurable parameter (n_{MAX}), which is provided at synthesis time to be implemented by using internal BRAMs. Instructions that use the vector memory provide the bit-size of operands, such as the code length (n) or the code dimension (k). The bit-size is programmable at run-time, permitting the vector core to work with different code lengths that accomplish $n \le n_{\text{MAX}}$.

The BRAMs are configured as a dual port memory with independent read and write ports, as depicted in Fig. 7. The width of the ports (denoted W) matches the AXI-Stream width (parameter AXIS_WIDTH) to provide faster processing with data retrieved from external DDRx RAM.

The memory is arranged in $2 \cdot \lceil n_{\text{MAX}}/W \rceil$ x W-bit words, providing a total capacity of $2 \cdot n_{\text{MAX}}$ bits to enable double-buffering during processing.

The core automatically swaps the base address of both buffers before starting a new instruction that requires simultaneous read/write access to BRAMs. If n_{MAX} is a power of 2, the addition to obtain the extended address is replaced with a simple bit concatenation of the buffer address with the most significant bit of the base address.

D. Vector Core

Table III resumes the instructions provided by the core, which is depicted in Fig. 8. All the instructions perform a computation from a binary vector v retrieved from the read buffer, writing the resulting vector v in the write buffer of the same memory. Therefore, these instructions do not specify the v operator, but they require specifying an argument $N \le n_{\text{MAX}}$, which is the number of bits considered in the binary vector $v \in \mathbb{F}_2^N$. As in the polynomial core, the procedures carried out by the states to execute instructions are very similar, reducing hardware resources.

TABLE III
VECTOR INSTRUCTION SET

	Vector operation Instruction example	Time (T_{CLK})
Add $V \in \mathbb{F}_2^N$ to $v \in \mathbb{F}_2^N$	$v \leftarrow V + v$ $add\_V(N)$	$\lceil N/W \rceil$
Mult $M \in \mathbb{F}_2^{rows \times N}$ to $v \in \mathbb{F}_2^N$	$v \leftarrow M \cdot v$ $mult\_M(N)$	$rows \cdot \lceil N/W \rceil$
Permute $v \in \mathbb{F}_2^N$ from $P \in \mathbb{F}_{2^{32}}^N$	$v(i) \leftarrow v(P(i))\ (0 \le i < N)$ $permute\_P(N)$	N
Goppa code from $v \in \mathbb{F}_2^N$	$r(Z) \leftarrow to\_goppa(v, N)$ $r4 \leftarrow to\_goppa(N)$	N

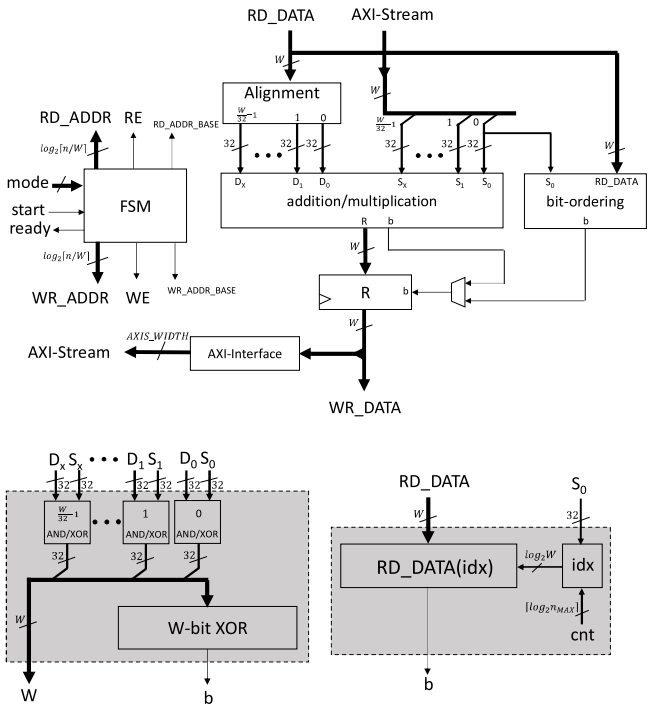


Fig. 8. Architecture of the vector core (top) and the addition/multiplication (bottom-left) and bit-ordering (bottom-right) subcircuits. .

For instance, the instructions for vector addition $v \leftarrow V + v$ and for matrix multiplication $v \leftarrow M \cdot v$ are very similar, sharing the same logic to read data from external memory, 32-bit alignment with retrieved data from BRAMs, and storage of partial results. The result is temporally stored in a W-bit register (R) before writing the result word into BRAM, taking $\lceil N/W \rceil$ clock cycles per row to perform the computation, where N is the number of bits for columns. The alignment circuit is required when the AXI-Stream is larger than 32-bit to align data retrieved from external memory with BRAM into 32-bit words. The external DMA must be properly configured to point to the base address of V or M, and the total number of words to read from DDRx RAM. The real execution time is higher than the theoretical one since the DMA introduces wait-states between burst transactions to access data stored in shared external memory.

The remaining instructions are the permutation $v(i) \leftarrow v(P(i))$ and the transformation of N bits from v to Goppa code $r(Z) \leftarrow to\_goppa(v, N)$. Both instructions are very similar since they basically read a single bit from v at every clock cycle at different locations, serially storing the resulting bits into the temporal register R. The location index of the bit to read from v is retrieved either from a permutation vector P stored in external memory or from the counter cnt for the other instruction. The register is sequentially stored in BRAM for the first instruction, or in one of the registers of the polynomial file for the last instruction.

The permutation of v by using a vector $P \in \mathbb{F}_{2^{32}}^N$ is equivalent to the expression $v \cdot P$, where $P \in \mathbb{F}_2^{N \times N}$ is a permutation binary matrix with exactly one 1 every row and column, but it requires lower accesses to memory.

The instruction to transform v to Goppa code could be executed in a single clock cycle by adding a large multiplexer at the input of the polynomial registers. However, the FPGA resources would be greatly increased, whereas the overall performance of McE decryption is very slightly improved since it is not frequently used.

Notice the benefit in the execution time for the vector addition and matrix multiplication when W is larger, but neither the permutation nor the Goppa transformation. By using similar procedures in the states carried out for all the instructions, hardware resources are minimized.

IV. McEliece

The McEliece (McE) is a code-based PK encryption system. Encryption is simple, but decryption is far more complex since it involves several algorithms based on operations with binary matrices, Galois-Field $\mathbb{F}_{2^m}$ and polynomials $\mathbb{F}_{2^m}[Z]$.

The key generation, encryption, and decryption described in the CMcE differ from the McE. The CMcE is a recent NIST standard candidate for Key Encapsulation Mechanism (KEM), which can efficiently implement the key generation and decoding. It is based on a syndrome-based variant of the McE, which encodes a message on a t-weight error string $\in \mathbb{F}_2^n$, encrypting it as the syndrome $c = H \cdot e \ \in \mathbb{F}_2^{mt}$, since $H \in \mathbb{F}_2^{mt \times n}$ is the parity-check matrix that is used as the public key.

A. Key Generation

Although the presented accelerator is not intended for key generation, we provide a brief description. A binary Goppa code is defined by a polynomial $g(Z)$ and a set of distinct elements L from a finite field, providing a high error correction capacity from a parity-check matrix H [22], [23], which is undistinguishable from random matrices. The key generation starts by computing a random irreducible Goppa polynomial $g(Z) \epsilon \mathbb{F}_{2^m}[Z]$. Then, the generator matrix $G \in \mathbb{F}_2^{k \times n}$ is computed for a Goppa code that meets the minimum distance criterion $d \ge 2t + 1$. The generator matrix is perturbated with two random invertible matrices: the non-singularity matrix $S \in \mathbb{F}_2^{k \times k}$and the permutation matrix $P \in \mathbb{F}_2^{n \times n}$. Finally, the public key $G_{pub} \in \mathbb{F}_2^{k \times n}$ is obtained from the multiplication of the three matrices as $G^{pub} = S \cdot G \cdot P$.

Algorithm 1: Pseudo-C code for Encryption.

Input: $(G^{pub})^T \in \mathbb{F}_2^{n\times k}$, $m \in \mathbb{F}_2^k$, $e \in \mathbb{F}_2^n$ $(t-weight)$
Result: $c \in \mathbb{F}_2^n$
$inst\_enc = \{ mult\_M(k);\ add\_V(n);\}$

McE_LoadVector(m, k);	$//v \leftarrow m$
McE_Launch(inst_enc);	
DMA_Read($(G^{pub})^T, n, k$)	$//v \leftarrow (G^{pub})^T \cdot v$
DMA_Read($e, 1, n$)	$//v \leftarrow e + v$
McE_SaveVector(c, n);	$//c \leftarrow v$

Algorithm 2: Pseudo-C code for Decryption.

Input: $c \in \mathbb{F}_2^n$, $(P_1 \cdot P)^{-1} \in \mathbb{F}_{2^{32}}^n$, $P_1 \in \mathbb{F}_{2^{32}}^n$, $(S^{-1})^T \in \mathbb{F}_2^{k\times k}$
Result: $m \in \mathbb{F}_2^{kn}$, $\varepsilon \in \mathbb{F}_2^n$
$inst\_dec1 = \{ permute\_P(n);\ \}$
$inst\_dec2 = \{add\_V(n); permute\_P(k); mult\_M(k);\ \}$

McE_LoadVector(c, n);	$//v \leftarrow c$
McE_Launch(inst_dec1);	
DMA_Read($(P_1 \cdot P)^{-1}, 1, n$);	$//v \leftarrow v \cdot (P_1 \cdot P)^{-1}$
McE_SaveVector(c', n);	$//c' \leftarrow v$
Patterson( );	$//v \leftarrow Patterson(v)$
McE_SaveVector(ε, n);	$//\varepsilon \leftarrow v$
McE_Launch(inst_dec2);	
DMA_Read($c', 1, n$);	$//v \leftarrow c' + v$
DMA_Read($P_1{}^{-1}, 1, k$);	$//v \leftarrow truncate(v \cdot P^{-1}, k)$
DMA_Read($(S^{-1})^T, k, k$);	$//v \leftarrow (S^{-1})^T \cdot v$
McE_SaveVector(m, k);	$//m \leftarrow v$

We developed the key generation as in the Flexiprovider library, which uses an auxiliary vector $P_1 \in \mathbb{F}_{2^{32}}^n$ to permute the columns of the parity-check matrix H, helping the systematization of G. The transposed parity-check matrix $H^T \in \mathbb{F}_2^{n\times mt}$ is stored as part of the private key along with the transposed inverse matrix $(S^{-1})^T$, the permutation vectors P^{-1}, $(P_1 \cdot P)^{-1}$, and the irreducible Goppa polynomial $g(Z)$.

At the starting time, the embedded microprocessor writes all the instructions that are going to be carried out by the accelerator for encryption and decryption. It also initializes the polynomial registers $r1 = 1 \ \epsilon\ \mathbb{F}_{2^m}[Z]$, $r2 = Z \ \epsilon\ \mathbb{F}_{2^m}[Z]$, $r15 = g(Z) \ \epsilon\ \mathbb{F}_{2^m}[Z]$, and $r14 = \sqrt{Z} \ \epsilon\ \mathbb{F}_{2^m}[Z]$.

B. Encryption

The encryption transforms a plain text $m \in \mathbb{F}_2^k$ and a randomly generated *t*-weight error $e \in \mathbb{F}_2^n$ to a ciphertext $c \in \mathbb{F}_2^n$ by using the public key matrix $G^{pub} \in \mathbb{F}_2^{k\times n}$, according to:

$$c = m \cdot G^{pub} + e = \left(G^{pub}\right)^T \cdot m + e \tag{1}$$

The instruction memory of the accelerator contains two instructions to perform the encryption, pointed by *inst_enc* (Algorithm 1). After loading the internal memory v with m, the microprocessor can launch the encryption on the accelerator. Then, the microprocessor configures the DMA to the addresses that store $(G^{pub})^T$ and e, and sets the number of rows and columns to start the reading of data from external memory. After completing the encryption, v stores the cyphertext c and writes it to external memory.

C. Decryption

After receiving the ciphertext $c \in \mathbb{F}_2^n$, a set of steps are carried out to get the located errors ε and recover the plain text *m*. The steps followed are:

1. $c' = c \cdot (P_1 \cdot P)^{-1}$
2. $\varepsilon = Decode(c')$
3. $y' = c' + \varepsilon$
4. $y = truncate(y' \cdot {P_1}^{-1}, k)$
5. $m = y \cdot S^{-1} = (S^{-1})^T \cdot y$

Since $(P_1 \cdot P)^{-1} = P^{-1} \cdot P_1^{-1}$, the first step removes the permutation *P* from the ciphertext. Notice the resulting vector $c' \in \mathbb{F}_2^n$ is not only required in the next step but also after the decoding. Therefore, c' is stored in external memory for later use, as shown in Algorithm 2. The Patterson algorithm decodes the location of errors, storing them into the binary vector $\varepsilon \in \mathbb{F}_2^n$ of *t*-weight. Then, the previously stored c' is retrieved from external memory and added to ε, obtaining $y' \in \mathbb{F}_2^n$. The auxiliary permutation P_1 is removed from y', resulting in the subvector $y \in \mathbb{F}_2^k$. Finally, the matrix $(S^{-1})^T \in \mathbb{F}_2^{k\times k}$ is multiplied with the subvector $y \in \mathbb{F}_2^k$, writing the computed plaintext $m \in \mathbb{F}_2^k$ into external memory.

1) Patterson Algorithm: The commonly used algorithms for decoding are the Patterson and the Berlekamp-Massey (BM). The BM is faster and simpler, although it can correct up to $t/2$ errors by default. The modified BM algorithm can recover the required t errors by the computation of the double-size parity matrix $H^{(2)}$ and syndrome $S^{(2)}$. The Patterson algorithm can correct the t errors by operating in binary Goppa codes and it has been thoroughly analyzed for side-channel attacks [23]. However, the algorithm is significantly more complex than the BM, requiring a larger variety of computations in Galois-Field $\mathbb{F}_{2^m}$ and polynomials $\mathbb{F}_{2^m}[Z]$.

The Patterson algorithm computes the location of errors from $c' \in \mathbb{F}_2^n$, storing them into $\varepsilon \in \mathbb{F}_2^n$. It is divided into six steps, starting with the syndrome calculation from the parity matrix $H \in \mathbb{F}_2^{mt\times n}$. The syndrome is stored in a register $(r4)$ to perform a set of polynomial operations. The Error Location Polynomial (ELP), denoted as $\sigma(Z)$, is obtained after two executions of the Extended Euclidean Algorithm (EEA) and a square root computation. Finally, the finding of roots provides the locations of the errors, storing them in a binary vector $\varepsilon \in \mathbb{F}_2^n$.

1. $S_{c'}(Z) = H^T \cdot c$'
2. $1 \equiv T(Z) \cdot S_{c'}(Z)\ mod\ g(Z)$ (by EEA)
3. $R(Z) = \sqrt{T(Z) + Z}$
4. $a(Z) \equiv b(Z) \cdot R(Z)\ mod\ g(Z)$ (by EEA)
5. $\sigma(Z) = a(Z)^2 + Z \cdot b(Z)^2$
6. $\varepsilon = roots(\sigma(Z))$

As in the previous codes, the microprocessor (Algorithm 3) launches the execution of two sets of instructions on the

Algorithm 3: Pseudo-C code for Patterson.

Input: $v = c' \in \mathbb{F}_2^n$, $H^T \in \mathbb{F}_2^{mt \times n}$
Result: $v = \varepsilon \in \mathbb{F}_2^n$
$inst\_pat1 = \{\ mult\_M(n);$
$r4 \leftarrow to\_goppa(m \cdot t);\ \}$
$inst\_pat2 = \{\ r3 \leftarrow r3{*}r3;\ r4 \leftarrow r4{*}r4;$
$r3 \leftarrow r2{*}r4 + r3;\ roots(r3, n);\ \}$
$r2 = Z\ \epsilon\ \mathbb{F}_{2^m}[Z]$

McE_Launch(inst_pat1);	
DMA_Read($H^T, m \cdot t, n$);	$//r4 \leftarrow to\_goppa(H^T \cdot v)$
EEA_Inv( );	$//r4 \leftarrow EEA\_Inv(r4)$
SQRT( );	$//r4 \leftarrow \sqrt{r4 + Z}$
EEA_Mod( );	$//\{r3, r4\} \leftarrow EEA\_Mod(r4)$
McE_Launch(inst_pat2);	$//v \leftarrow roots(r4^2 \cdot Z + r3^2,\ n)$

accelerator and programs the DMA when they require external memory access.

2) Roots Finding Algorithm: There are several algorithms that can be used to find the roots of a polynomial $\sigma(Z)\,\epsilon\,\mathbb{F}_{2^m}[Z]$. For instance, the Gao-Matter additive FFT reduces the number of additions and multiplications in the Galois field $\mathbb{F}_{2^m}$ by using a butterfly network, improving the execution time on software.

We apply Horner's method since it can be easily implemented on the paralleled $\mathbb{F}_{2^m}$ adder-multiplier and without additional dedicated circuits, reducing FPGA resources. Moreover, Horner's algorithm is very efficiently performed, achieving a throughput of one evaluation per clock cycle, as commented in Section III.B.

3) Square Root Algorithm: The square root of a polynomial $a(Z)$ is efficiently performed by computing the square root of the $\mathbb{F}_{2^m}$ coefficients and splitting them into even and odd coefficients. The resulting coefficients are combined with the precomputed $\sqrt{Z}\ \epsilon\ \mathbb{F}_{2^m}[Z]$ to obtain the square root of the polynomial [23], according to:

$$\sqrt{a(Z)} = \sum_{i=0}^{\lfloor \frac{t}{2} \rfloor} e_i^{2^{m-1}} \cdot Z^i + \sqrt{Z} \cdot \sum_{i=0}^{\lfloor \frac{t-1}{2} \rfloor} o_i^{2^{m-1}} \cdot Z^i \qquad (2)$$

The computation of $\sqrt{T(Z) + Z}$ (Algorithm 4) starts adding Z to $T(Z)$, storing the result in $r4$. Then, instruction $r4 \leftarrow GF(r4 * r4 + r0)$ is repeated $m - 1$ times, obtaining the square roots. The resulting coefficients splits into the even $e_i^{2^{m-1}}$ and odd $o_i^{2^{m-1}}$ sets, which are stored in registers $r6$ and $r7$, respectively. The last instruction multiplies $\sqrt{Z}$ and $o_i^{2^{m-1}}$, adding $e_i^{2^{m-1}}$ to get the result.

4) Extended Euclidean Algorithm: The applied EEA computes the Greatest Common Divisor (gcd) equation from an input polynomial $p(Z)$ and the irreducible $g(Z)$ along with the Bézout co-polynomial $\mu(Z)$, satisfying:

$$\gcd(g(Z), p(Z)) = a(Z) = \lambda(Z) \cdot g(Z) + \mu(Z) \cdot p(Z) \qquad (3)$$

Algorithm 4: Pseudo-C code for $\sqrt{T(Z) + Z}$.

Input: $r4 = T(Z)\ \epsilon\ \mathbb{F}_{2^m}[Z]$
Result: $r4 = \sqrt{T(Z) + Z}\ \epsilon\ \mathbb{F}_{2^m}[Z]$
$inst\_sqrt = \{\ r4 \leftarrow r1{*}r4 + r2;$
$r4 \leftarrow GF(r4{*}r4 + r0);$
$\ldots$ (Repeated $m - 1$ times)
$\{r6, r7\} \leftarrow split(r4);$
$r4 \leftarrow r14{*}r7 + r6;\ \}$
$r14 = \sqrt{Z}\ \epsilon\ \mathbb{F}_{2^m}[Z]$
$r2 = Z\ \epsilon\ \mathbb{F}_{2^m}[Z]$
$r1 = 1\ \epsilon\ \mathbb{F}_{2^m}[Z]$

McE_Launch(inst_sqrt);	$//r4 \leftarrow \sqrt{r4 + Z}$

The equation is solved by several iterations, starting from the initial condition:

$$\begin{array}{ll} a_0(Z) = p(Z) & \mu_0(Z) = 1 \\ a_{-1}(Z) = g(Z) & \mu_{-1}(Z) = 0 \end{array} \qquad (4)$$

At each iteration $i \geq 1$, new polynomials $a_i(Z)$ and $\mu_i(Z)$ are computed by:

$$\begin{array}{l} q(Z) = a_{i-2}(Z) / a_{i-1}(Z) \\ a_i(Z) = a_{i-2}(Z) \cdot q(Z) + a_{i-1}(Z) \\ \mu_i(Z) = \mu_{i-2}(Z) .q(Z) + \mu_{i-1}(Z) \end{array} \qquad (5)$$

For the EEA_Inv, the process is repeated until $a(Z) = 0$, detected by $deg(a(Z)) < 0$ to accomplish the equability $1 \equiv \mu(Z) \cdot p(Z)\ mod\ g(Z)$. The maximum number of iterations that can be performed is t since the maximum degree of $p(Z)$ is $t - 1$. However, the condition $deg(a(Z)) < 0$ can be achieved before t iterations, reducing the execution time. To implement a constant-time EEA, the algorithm is executed iteratively using a variable j that keeps track of the remaining number of iterations (Algorithm 5). The first execution initializes the registers and performs normally, storing the result in two registers after completing. The second one computes the remaining iterations but does not store the result. In both executions, the condition $j > 0$ is checked and reads $deg(r7)$ from the accelerator, although the condition $deg(r7) \geq -1$ is always true.

For the EEA_Mod the same process is repeated but until $deg(a(Z)) \leq t/2$ or $deg(\mu(Z)) \leq (t-1)/2$ that accomplishes $a(Z) \equiv \mu(Z) \cdot p(Z)\ mod\ g(Z)$. In this case, the maximum number of iterations is $(t + 1)/2$.

V. Experimental Results

The experimental results show the hardware resources and execution times of McE encryption and decryption using the Vivado 2019.1. The McE accelerator is implemented in two different embedded systems: on a low-cost FPGA and on a high-end MPSoC prototyping board. The first system is based on the soft-core MicroBlaze processor on an Artix-7A50T device, which is connected to a 256 MB DDR3 memory [24]. The high-end Zynq UltraScale+ XCZU9EG device contains a quad-core embedded 64-bit ARM-A53 processor, which is connected to a 4 GB DDR4 memory [25].

Algorithm 5: Pseudo-C code for EEA_Inv.

Input: $r4 = p(Z)\ \epsilon\ \mathbb{F}_{2^m}[Z]$,
Result: $r3 = a(Z)\epsilon\ \mathbb{F}_{2^m}[Z]$, $r4 = \mu(Z)\ \epsilon\ \mathbb{F}_{2^m}[Z]$

```
inst_eea1 = { r7 ← r4;                    // r7=a0=p(Z)
              r6 ← r15;                   //r6=a_1=g(Z)
              r10 ← r1;                   //r10=μ0(Z)=1
              r9 ← r0; }                  //r9=μ_1(Z)=0
inst_eea2 = { {r8, r5} ← r6/r7;  //r5=q(Z) r8=ai(Z)
              r11 ← r5*r10 + r9;          //r11=μi(Z)
              r6 ← r7; r7 ← r8;           //r6=ai-1 r7=ai
              r9 ← r10; r10 ← r11; }      //r9=μi-1 r10=μi
inst_eea3 = { r3 ← r6; r4 ← r9;}
r15 = g(Z) ε F2m[Z]
r1 = 1 ε F2m[Z]
```

```
j = t;
McE_Launch(inst_eea1);
while(j > 0 && deg(r7) ≥ 0) {
  McE_Launch(inst_eea2); j = j − 1; }
McE_Launch(inst_eea3);
while(j > 0 && deg(r7) ≥ −1) {
  McE_Launch(inst_eea2); j = j − 1; }
```

The hardware execution time (Hw) of the accelerator includes the programming of the DMA and access of keys, and data from external RAM. The software execution time (Sw) is reported for a constant-time C-code and compiled with the highest optimization for the target microprocessor. The MicroBlaze is configured in 32-bit mode with all the settings for the highest performance, such as cache enabled, barrel-shifter, hardware multiplier, divisor, etc.

Table IV reports the FPGA resources (LUTs, FFs, BRAMs) of the synthesized accelerator and the selected clock frequency (MHz) as well the execution time (ms) for encryption and decryption on Sw and Hw, respectively, and the achieved speed-up (SU). The security parameters have been selected to the highest levels defined on the CMcE ($m = 13,\ t = 119,\ 128,\ n = 6960,\ 6688,\ 8192$). Since n can be changed at run-time, ensuring $n \leq n_{\text{MAX}}$, the synthesis parameter n_{MAX} is fixed to its maximum valid value (2^m).

For the low-cost Artix-7A50T, the parameters GF2MULT_FACTOR (F=4) and AXIS_WIDTH (W=64) are selected to fit at the available resources, since they are shared with the embedded soft-core microprocessor and peripherals. In the high-end Zynq UltraScale+ XCZU9EG, the larger available FPGA resources permit to select the performance parameters for a faster implementation. The W=256 is selected to enhance accessing speed to the external memory and F=1 to implement the fully paralleled $\mathbb{F}_{2^m}$ adder-multiplier.

Unfortunately, the security parameters provided by the McE processors [10] [15] [1] described in Section II are notably lower than those of the proposed accelerator, making them not comparable. The number of hardware resources for the Artix-7 is quite small when compared with the CMcE processors [8] [11] [12], from 12301 to 13200 LUTs, from 8460 to 9160 FFs, and only 5 BRAMs. This fact is explained because the CMcE cyptoprocessors are built from a larger set of computing circuits, each one designed for a specific stage, and keys and data are internally stored in BRAM. On the contrary, in the proposed McE accelerator, computing circuits are reused on several stages and keys are retrieved from external memory. However, it is important to point out that this comparison is not fair since CMcE processors are specifically designed for Key Encapsulation Mechanism (KEM). These cryptoprocessors include the key generation circuitry, which is much more frequently executed when compared to the encryption-decryption operations in McE. The execution time by using the accelerator varies from 5.33 to 8.58 ms on encryption, and from 10.4 to 13.5 ms on decryption, running at 100 MHz. The encryption speed-up against MicroBlaze running at the same clock frequency is around x33.5, meanwhile the decryption speed-up goes from x1033 to x1549. This difference is because encryption time is dominated by a matrix-vector multiplication instruction, which is limited by the reading time of the public key from external memory. However, the decryption executes not only three matrix-vector multiplications but also polynomial instructions. Polynomial instructions are significantly accelerated since the register file is used for reading operands and storing results, performing computations by the paralleled $\mathbb{F}_{2^m}$ adder-multiplier.

Note that encryption (Algorithm 1) solely executes the instructions *add_V(n)* (vector addition) and *mult_M(k)* (matrix-vector multiplication). The decryption (Algorithm 2) requires all the instructions that were implemented, including the two previously commented. Therefore, an accelerator that could only execute decryption would not save hardware resources. Additionally, the design focuses on accelerating the decryption process, which requires much more computing time than encryption.

Due to the selected performance parameters, dedicated resources on the Zynq UltraScale+ are increased when compared with the Artix-7. Since W and 1/F are multiplied by four and the clock frequency doubles, the execution times on the accelerator are reduced roughly by eight. The speed-up is lower than in the Artix-7 since the clock frequency at the ARM-A53 microprocessor (1.2 GHz) is 6 times higher than at the accelerator (200 MHz), achieving from x371 to x556 on decryption. The encryption speed-up is limited to x6.8 due to external memory accesses required for retrieving the public key.

The Zynq UltraScale+ enables larger security parameters. We tested values from $m = 13$ to $m = 15$, $n = 2^m$, and the highest t that accomplishes $m \cdot t \leq n/2$. The performance parameter W=256 is selected to provide high accessing speed to the external memory. In these cases, the F parameter has been tested from 1 to 5 since it greatly affects the area-performance tradeoff of the accelerator. The F significantly affects the number of LUTs but has little effect on the FFs and none in BRAMs, as depicted in Fig. 9(left). The clock frequency is reduced for larger m because of to the added complexity in the combinational $\mathbb{F}_{2^m}$ arithmetic circuits. Although the encryption speed-up is reduced due to the lower clock frequency, independently from F given that it does not execute polynomial instructions, the decryption speed-up increases with t and depends on F. Increasing F leads to a lower

TABLE IV
FPGA RESOURCES & EXECUTION TIMES

Synthesis results									Execution results						
n_{MAX}	m	t	W	F	LUTs	FFs	BRAMs	MHz	n	$T^{SW}_{encrypt}$	$T^{HW}_{encrypt}$	$SU_{encrypt}$	$T^{SW}_{decrypt}$	$T^{HW}_{decrypt}$	$SU_{decrypt}$
Artix-7A50T									SW on MicroBlaze@100 MHz						
8192	13	119	64	4	12301	8460	5	100	6960	199.8	5.97	33.5	13241	10.4	1269
									8192	287.9	8.58	33.6	13549	13.1	1033
8192	13	128	64	4	13200	9160	5	100	6688	177.5	5.33	33.3	16968	10.4	1549
									8192	281.7	8.41	33.5	16461	13.5	1215
Zynq UltraScale+ XCZU9EG									SW on ARM-A53@1.2 GHz						
8192	13	119	256	1	24976	9218	8	200	6960	5.3	0.77	6.8	650.0	1.44	453
									8192	7.5	1.11	6.8	663.1	1.79	371
8192	13	128	256	1	26428	9857	8	200	6688	4.7	0.69	6.8	791.5	1.42	556
									8192	7.4	1.09	6.8	808.5	1.85	436
				1	55255	21959	8	200						4.34	2480
				2	45055	22016	8	200						7.40	1456
8192	13	315	256	3	36775	22103	8	200	8192	4.8	0.70	6.8	10776	10.5	1031
				4	36676	22024	8	200						13.5	798
				5	32682	22174	8	200						16.6	651
				1	103930	42454	11.5	160						18.68	3730
				2	81752	42525	11.5	160						31.7	2193
16384	14	585	256	3	67076	42627	11.5	160	16384	18.6	3.39	5.5	69423	44.7	1554
				4	64986	42599	11.5	160						57.7	1204
				5	59225	42693	11.5	160						70.7	982
				1	210138	83485	19.5	125						84.5	5408
				2	156852	83650	19.5	125						142.0	3211
32768	15	1092	256	3	132300	83638	19.5	125	32768	73.3	17.05	4.3	455100	199.7	2284
				4	121665	83692	19.5	125						257.4	1772
				5	118161	83643	19.5	125						315.1	1448

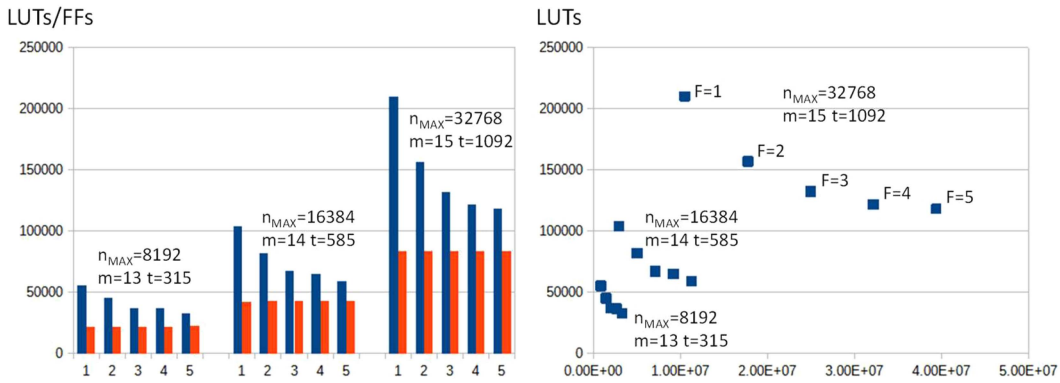


Fig. 9. Number of LUTs and FFs for the three highest security levels (left) from F=1 to F=5. Area-performance tradeoff for the same cases (right), representing the number of clock cycles on decryption at the x-axis.

number of LUT resources needed by the $\mathbb{F}_{2^m}$ adder-multiplier at the cost of augmenting the clock cycles required by polynomial instructions, incrementing the decryption time. Nonetheless, increasing F further will not significantly reduce the number of LUTs but the decryption time is greatly increased, as depicted in Fig. 9(right). For the higher security parameters, changing from F=1 to F=2 reduces the number of LUTs by 25% but increases the decryption time by 68%. Changing from F=4 to F=5 reduces LUTs only by 3%, and decryption time increases by 23%. Finally, changing from F=1 to F=5, LUTs are reduced by 44%, and decryption time increases by 273%. However, as the experimental results show, the speed-up is very noticeable, from x5401 to x1446, and similar behavior is observable for the other reported cases.

The Hw/Sw codesign [17] does not work in constant time, performing faster polynomial instructions when coefficients are zero. Even so, when comparing the proposed accelerator (n=8192, n=13, t=316, P=1 case) with [17], the decryption

TABLE V
SW EXECUTION TIMES ON INTEL I7-13700H

n	m	t	$T^{SW}_{encrypt}$	$T^{SW}_{decrypt}$
6960	13	119	0.29 (100%)	18.76 (1.87%)
8192	13	119	0.42 (100%)	20.16 (2.33%)
6688	13	128	0.26 (100%)	21.81 (1.53%)
8192	13	128	0.44 (100%)	24.90 (2.05%)
8192	13	316	0.27 (100%)	226.0 (0.24%)
16384	14	585	1.04 (100%)	1268 (0.13%)
32768	15	1091	4.65 (100%)	7591 (0.08%)

time is reduced by x11, using 29% fewer LUTs and 47% fewer FFs on the same FPGA.

Uniquely during the execution of the instructions for vector addition, permutation, and matrix-vector multiplication, the DMA accesses data from external memory. The vector core performs calculations from the retrieved data in parallel with the DMA reading the subsequent data from DDRx. As a result, there is no overhead due to data transfer. However, processing time is limited by the data throughput of the external DDRx and the AXI-Stream, interconnected through the DMA. If DDRx throughput is large enough, augmenting the AXI-Stream width (W) reduces the execution time of vector instructions at the cost of increasing the FPGA resources devoted. However, the overall decryption time is not significantly improved since it is largely dominated by the polynomial instructions.

The maximum throughput of the DDR4 of the high-end Zynq UltraScale+ board is 17066 MB/s. The peak throughput at the AXI-Stream can achieve $2{\cdot}f_{CLK}{\cdot}(256/8)$=12800 MB/s since W=256 and the DMA operates at double the accelerator frequency ($2{\cdot}f_{CLK}$ =400 MHz) by using additional circuits for synchronizing both clock domains.

For the case of the Artix-7 board, the maximum achievable throughput of the DDR3 is 1600 MB/s. The AXI-Stream peak throughput is 800 MB/s since W=64 and the system operates on a single clock domain running at f_{CLK}=100 MHz. Increasing W or the clock frequency of the DMA could potentially improve throughput to accelerate vector instructions. However, such changes require increasing the parallelization factor (F) of the polynomial core to fit into the available resources of the low-cost device, which leads to a significant degradation in the overall decryption time. Therefore, on this device, it is preferable to use a single clock domain and maintain W = 64 for a more efficient decryption.

The previous experimental results focus on the Hw accelerator, which is suitable for embedded systems with cost and performance limitations compared to Personal Computers (PCs). Nonetheless, Table V shows execution times on a high-performance core of an Intel i7-13700H running at up to 5 GHz. Alongside the encryption and decryption times (in ms), the table also presents the percentage of processing time devoted to computations over vectors: 100% for encryption and very low for decryption, ranging from 2.33% to 0.08%. The results demonstrate that decryption on FPGAs is faster than on the i7 core for the tested cases, primarily because the processing time is dominated by computations over polynomials. The polynomial core benefits from parallelization of calculations within internal memory. Conversely, encryption on the i7 core is faster because it is vastly dominated by a matrix-vector multiplication. The higher clock frequency and memory throughput of the PC (DDR5-5200, 41600 MB/s) justify these results.

VI. CONCLUSION

An AXI run-time programmable McE accelerator is presented, which provides a set of configurable parameters. The goal is to provide a flexible accelerator for embedded systems that can be adapted to several applications that require constant-time McE encryption-decryption. Depending on the security parameters required by an application, the performance parameters can be selected to achieve the best area-performance tradeoff in the implementation.

The accelerator is designed in HDL for a better control of the FPGA resources and the clock cycles needed by the implementation. It provides a set of application-specific instructions to operate in binary vectors, matrices, and Goppa codes, providing AXI connectivity to an embedded microprocessor. Keys and instructions memory are programmable from such microprocessor at run-time, improving its flexibility.

Experimental results show as the accelerator can implement the highest security parameters of CMcE on low-cost and high-end FPGAs, achieving up to x1549 and x556 decryption speed-up, respectively.

Such security parameters can be noticeably increased to levels not achieved in previous related cryptoprocessors. In the larger tested case, the accelerator provides decryption speed-up from x5408 to x1448 while devoting from 210138 to 118161 LUTs, depending on the selected tradeoff.

The main limitation of the proposed accelerator is the lack of support for key generation. Computing the transposed inverse matrix requires additional circuits and large internal memories that exceed the resources of low-cost devices. Furthermore, key generation is heavily used in KEM (Classic McEliece) but not in PK encryption-decryption (McEliece). Thus, extending the accelerator for key generation instructions in McEliece offers limited benefits from Hw and is unsuitable for low-cost devices.

REFERENCES

[1] P. M. C. Massolino, P. S. L. M. Barreto, and W. V. Ruggiero, "Optimized and scalable co-processor for McEliece with binary goppa codes," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, 2015, Art. no. 45, doi: 10.1145/2736284.

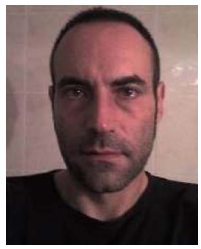
[2] S. Heyse and T. Güneysu, "Towards one cycle per bit asymmetric encryption: Code-based cryptography on reconfigurable hardware," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2012, pp. 340–355, doi: 10.1007/978-3-642-33027-8_20.

[3] D. J. Bernstein, T. Chou, and P. Schwabe, "McBits: Fast constant-time code-based cryptography," in *Proc. Int. Conf. Cryptographic Hardware Embedded Syst.*, 2013, pp. 250–272, doi: 10.1007/978-3-642-40349-1_15.

[4] T. Chou, "McBits revisited: Toward a fast constant-time code-based KEM," *J. Cryptographic Eng.*, vol. 8, no. 2, pp. 95–107, 2018, doi: 10.1007/s13389-018-0186-9.

[5] M. S. Chen and T. Chou, "Classic mceliece on the ARM cortex-M4," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2021, no. 3, pp. 125–148, 2021, doi: 10.46586/tches.v2021.i3.125-148.

[6] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based key generator for the Niederreiter cryptosystem using binary goppa codes," in *Proc. Int. Conf. Cryptographic Hardware Embedded Syst.*, 2017, pp. 253–274, doi: 10.1007/978-3-319-66787-4_13.
[7] W. Wang, J. Szefer, and R. Niederhagen, "FPGA-based niederreiter cryptosystem using binary goppa codes," in *Proc. Int. Conf. Post-Quantum Cryptogr.*, 2018, pp. 77–98, doi: 10.1007/978-3-319-79063-3_4.
[8] Albrecht M. et al., "Classic McEliece: Conservative code-based cryptrography," 2020, [Online]. Available: https://classicmceliece.org/nist
[9] P. J. Chen et al., "Complete and improved FPGA implementation of classic McEliece," *IACR Trans. Cryptographic Hardware Embedded Syst.*, vol. 2022, no. 3, pp. 71–113, 2022, doi: 10.46586/tches.v2022.i3.71-113.
[10] A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert, "A novel cryptoprocessor architecture for the McEliece public-key cryptosystem," *IEEE Trans. Comput.*, vol. 59, no. 11, pp. 1533–1546, Nov. 2010, doi: 10.1109/TC.2010.115.
[11] V. Kostalabros, J. Ribes-González, O. Farràs, M. Moretó, and C. Hernandez, "HLS-based HW/SW co-design of the post-quantum classic McEliece cryptosystem," in *Proc. 31st Int. Conf. Field-Program. Log. Appl.*, 2021, pp. 52–59, doi: 10.1109/FPL53798.2021.00017.
[12] S. Chen, H. Lin, W. Huang, and Y. Huang, "Hardware design and implementation of classic McEliece post-quantum cryptosystem based on FPGA," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2022, pp. 1–6, doi: 10.1109/HPEC55821.2022.9926295.
[13] M. Gurel, "A comparative study between RTL and HLS for image processing applications with FPGAs," Univ. California, San Diego, 2016. [Online]. Available: https://escholarship.org/uc/item/9vx1s37b
[14] R. Millón, E. Frati, and E. Rucci, "A comparative study between HLS and HDL on SoC for image processing applications," *Elektron*, vol. 4, no. 2, pp. 100–106, 2020, doi: 10.37537/rev.elektron.4.2.117.2020.
[15] I. Von Maurich and T. Güneysu, "Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices," in *Proc. Des. Automat. Test Europe*, 2014, pp. 1–6, doi: 10.7873/DATE2014.051.
[16] Y. M. Kuo, F. Garcia-Herrero, O. Ruano, and J. A. Maestro, "RISC-V galois field ISA extension for non-binary error-correction codes and classical and post-quantum cryptography," *IEEE Trans. Comput.*, vol. 72, no. 3, pp. 682–692, Mar. 2023, doi: 10.1109/TC.2022.3174587.
[17] M. López-García and E. Cantó-Navarro, "Hardware-software implementation of a McEliece cryptosystem for post-quantum cryptography," in *Advances in Intelligent Systems and Computing*, Berlin, Germany: Springer, 2020, doi: 10.1007/978-3-030-39442-4_60.
[18] H. M. Therar, E. A. Mohammed, and A. J. Ali, "Biometric signature based public key security system," in *Proc. 3rd Int. Conf. Adv. Sci. Eng.*, 2020, pp. 1–6, doi: 10.1109/ICOASE51841.2020.9436615.
[19] Y. K. Lee and J. Jeong, "Securing biometric authentication system using blockchain," *ICT Exp.*, vol. 7, no. 3, pp. 322–326, 2021, doi: 10.1016/j.icte.2021.08.003.
[20] A. Sharma and D. B. Ojha, "Biometric template security using code base cryptosystem," *Inf. Secur.: An Int. J.*, vol. 26, no. 2, pp. 49–60, 2013.
[21] R. Arjona, P. López-González, R. Román, and I. Baturone, "Post-Quantum biometric authentication based on homomorphic encryption and classic McEliece," *Appl. Sci.*, vol. 13, no. 2, 2023, Art. no. 757, doi: 10.3390/app13020757.
[22] E. R. Berlekamp, "Goppa codes," *IEEE Trans Inf Theory*, vol. IT-19, no. 5, pp. 590–592, Sep. 1973.
[23] R. Safieddine and A. Desmarais, "Comparison of different decoding algorithms for binary goppa codes," 2014. [Online]. Available: http://www.cayrel.net/?Implementation-of-Goppa-codes
[24] "Xilinx artix-7 FPGA 50T evaluation kit," 2014. [Online]. Available: https://www.avnet.com/wps/portal/apac/products/product-highlights/xilinx-artix7/
[25] "Zynq UltraScale+ MPSoC ZCU102 evaluation kit," 2023. [Online]. Available: https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html



Enrique Cantó-Navarro received the MS degree in electronics enginering in 1995 and the PhD is 2001 at the Universidad Politécnica de Catalunya (UPC), in 1995 and 2001. He has been associate professor since 1996 in the UPC, and assistant professor with the Universitat Rovira i Virgili (URV) since 2003. He has participated in several National and International research projects related to smart-cards, FPGAs, hardware accelerators and biometrics. He has published more than 60 research papers in journals and conferences. His research interests include hardware accelerators for biometrics algorithms, cryptography, and run-time reconfigurable embedded systems.

Mariano López-García received the MS and PhD degrees in telecommunication engineering from the Technical University of Catalonia, Barcelona, Spain, in 1996 and 1999, respectively. In 1995, he joined the Department of Electronic Engineering where he became an associate professor in 2001. He currently teaches courses in Microelectronics and Advance Digital Design. He also taught during several years Power Electronics, Analog Electronics and Design of PCB Boards at undergraduate level. He spent one year at Cambridge University, Faculty of Computer Sciences, as visiting scholar collaborating on the implementation of biometric algorithms on low-cost devices. He is currently member of the "Subcomité Español CTN 71 SC37 para Identificación biométrica" of AENOR. His research interests include signal processing, biometrics, embedded systems, hardware-software co-design and FPGAs.