2024 27th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)

Optimised AES with RISC-V Vector Extensions

Mahnaz Namazi Rizi*, Nusa Zidaric*, Lejla Batina‡ and Nele Mentens*†
*Leiden university, The Netherlands Email:{m.namazi.rizi, n.zidaric, n.mentens}@liacs.leidenuniv.nl
†KU Leuven, Belgium Email: nele.mentens@kuleuven.be
‡Radboud Univerity, The Netherlands Email:lejla@cs.ru.nl

2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS) | 979-8-3503-5934-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/DDECS60919.2024.10508919

***Abstract*—With the advent of quantum computers, organizations and users should consider the potential impact of quantum threats on their cryptographic systems and be prepared to adopt Post-Quantum Cryptography (PQC) solutions when needed. However, PQC algorithms are often difficult to implement on standard processors and resource-constrained embedded devices, due to complicated mathematical algorithms and large parameters. The goal of this research is to design efficient HW/SW co-design of the PQC algorithm Classic McEliece (CM) using the RISC-V Instruction Set Architecture (ISA). In the first step, the acceleration of the AES algorithm, which is used as part of the key generation in CM, is explored using RISC-V Vector Extensions version 1.0 (RVV1.0). In this paper, we compare the vector-accelerated AES running on Vicuan coprocessor with the scalar AES running on Ibex.**

***Index Terms*—Classic McEliece, AES, Acceleration, RISC-V, Vector Extensions**

I. Introduction

Nowadays, the security of many systems relies on Public Key Cryptography (PKC) algorithms which employ a pair of keys: a public and a private. Although these keys are mathematically related, it would take too many resources and time to derive one key from the other using classical computers. Thus, breaking a PKC algorithm is supposed to be practically infeasible. However, quantum computers will change the story as they can factorize large composite numbers and solve discrete logarithm problems in polynomial time using Shor's algorithm [1]. To resist quantum computers, the National Institute of Standard and Technology (NIST) initiated a competition in 2017, to evaluate and standardize one or more quantum-resistant Key-Encapsulation Mechanisms (KEM), public-key encryption (PKE) and digital signature schemes. These schemes leverage mathematical structures that are not efficiently solvable by both classical and quantum computers. Based on these structures, PQC algorithms are divided into several categories: lattice-based cryptography, hash-based cryptography, isogeny-based cryptography, multivariate-based cryptography and code-based cryptography. At the time of writing, NIST has finalized four algorithms in its standardization process and put four other algorithms, including Classic McEliece(CM) in the fourth round for further research. This motivates further investigation of efficient implementation of these candidates. The implementation of PQC algorithms is often a challenging task, due to complex computations and large key sizes. Accelerating key generation in PQC is crucial to reduce the time during which the keys might be exposed to potential quantum attacks. As it was stated in the third round documentation of NIST, "Classic McEliece has a very large public key size and fairly slow key generation. This is likely to make CM undesirable in many common settings" [2]. CM has been studied less extensively than lattice-based cryptography algorithms, which have been widely studied after the NIST standardized CRYSTAL-Kyber in the third round. This study aims to accelerate CM (Key Generation, Encapsulation, Decapsulation) for RISC-V based processors by exploiting parallelization through standard and custom RISC-V Vector Extensions version 1.0 (RVV1.0) [3] and various algorithmic optimization techniques. RISC-V is chosen as a popular open-source and royalty-free ISA [4], which gives the opportunity of adding custom instructions for performance reasons. To the best of our knowledge, the use of RVV1.0 for speeding up the whole CM algorithm has not been explored yet. Only in [5], a simulation model of a RISC-V processor supporting RVV1.0 was used to explore the capability of vectorization on Gaussian Elimination in public key generation.

979-8-3503-5934-3/24/$31.00 ©2024 IEEE

II. Approach and Background

In order to propose a design in RISC-V ISA, we first conducted a thorough literature survey of the ISA, its extensions, and available RISC-V processors and vector coprocessors, and investigated their use for cryptography. This survey was partly published in [6]. We chose the combination of Ibex [7] and the Vicuna vector coprocessor [8] as shown in Fig. 1, and Spike, the official RISC-V ISA simulator, to validate our design. A part of the CM Key Generation step is generation of random bytes. We decided to use AES (Advanced Encryption Standard) [9] as a subroutine for this task. AES has namely proven to be suitable non-linear high-speed pseudo-random number generator [10]. So, we investigate the efficiency of AES implementation Tiny-AES [1] using RVV1.0 We propose two vector-accelerated AES variants which are executable on Vicuan vector coprocessor and compare their cycle counts, instruction counts, and memory footprint with the scalar design running on Ibex. Ibex is a 32-bit RISC-V soft-core with two pipeline stages [7]. It supports the RISC-V base Integer(I)/Embedded(E) instruction set RV32I/RV32E, plus Integer Multiplication and Division(M), Compressed(C) and Bit Manipulation(B) extensions. It executes non-vector (scalar) instructions and forwards vector instructions to Vicuna [8].

[1]https://github.com/kokke/tiny-AES-c

Authorized licensed use limited to: DUBLIN CITY UNIVERSITY. Downloaded on April 02,2025 at 15:04:36 UTC from IEEE Xplore. Restrictions apply.

Users can customise Vicuna to have as many parallel vector pipelines as needed, with each pipeline containing one or multiple execution units. This approach can benefit algorithms with high computational intensity. If the memory bandwidth limits the effectively achievable performance, more execution units will not yield more throughput. The pre-defined layouts of Vicuna are listed as follows:

1) Compact: A single pipeline with all execution units.
2) Dual: VLSU, VALU, VIDXU in one pipeline, and VMUL and VLSDU in the second.
3) Triple: VLSU in one pipeline, VALU and VIDXU in the second, and VMUL and VSLDU in the third.
4) Legacy: A layout with each unit in a separate pipeline.

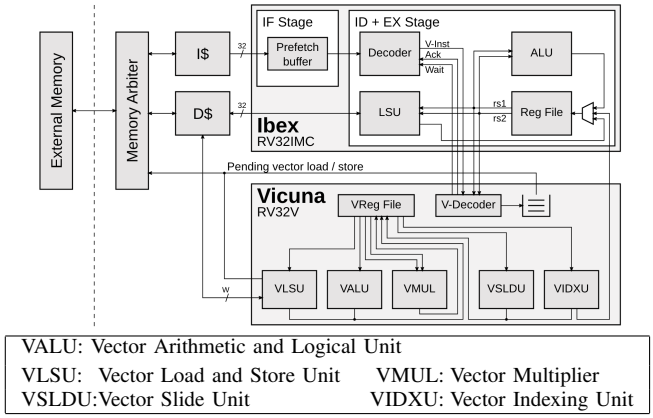

Fig. 1: The overview of Vicuna coupled with Ibex [8]

The security of McEliece, as a code-based public-key cryptosystem, which was introduced by Robert J. McEliece in 1978 [11], is based on the difficult problem of decoding a message from random errors using an error-correcting code from the family of Goppa codes. It seems that quantum computers do not give significant improvement in attacking McEliece, beyond improvements in brute force search possible with Grover's algorithm [12]. Classic McEliece (CM) is designed to combine the advantages of McEliece and Niederreiter. The existing McEliece uses a Generator Matrix (G) for the public key, whereas CM uses the Parity Check Matrix (H) used as the public key in Niederreiter [13]. CM as a Key Encapsulation Mechanism includes three processes: Key generation, Encapsulation and Decapsulation. Ten different parameter sets with different security levels are submitted for CM: two level-1 with public key size around 256 KB, two level-3 with public key size around 512 KB and six level-5 parameter sets with public key sizes around 1 MB and 1.3 MB. Four official software implementations are proposed for each of these ten parameter sets in the third-round submission: one reference implementation in C (ref), one vectorized implementation for 64-bit integers (vec) and two implementations using the Intel/AMD 128-bit vector instructions (sse, avx) [14].

III. Related Work

In [15], some algorithm-hardware co-design schemes are employed to accelerate the key generation in CM, while minimizing the storage overhead caused by large-size keys. Compared to existing designs, their FPGA implementation is around 4x faster with 9∼14x less memory-time. Processing In Memory (PIM) is investigated to accelerate CM in [16]. In the case of transposed public keys, they achieved a speed-up of 12.6x for key generation and 26x for encryption using direct-mapping cache. Sim et al. [13], propose an efficient software implementation for encapsulation and decapsulation of CM on 64-bit ARMv8 processors. Their Encapsulation implementation uses vector registers for 16-byte parallel operations and achieve a performance improvement of up-to 1.99x. The slow public key generation in CM needs a big binary matrix to be inverted. To perform the inversion efficiently, the matrix must be kept in a fast RAM. However, high-speed RAM is expensive and standard smart card controller rarely have RAM bigger than 64k Bytes. So in [17] an approach for outsourcing the expensive matrix inversion algorithm to the host system connected to the security controller is explored.

A constant-time hardware implementation of CM, including Key generation, Encapsulation and Decapsulation is proposed and evaluated on Xilinx Artix7 FPGA in [18]. The results for two variants, High-speed and Lightweight, are reported and compared with the results of existing hardware designs of the code-based KEM schemes, BIKE [19] and the high-level synthesis from C code of HQC in the third-round.

CM implementation on ARM Cortex-M4 was explored in [20] and [21]. Roth et al. [21] tried to overcome the memory constraint of CM by ad-hoc generation of the public key from the (extended) private key. The encapsulation process takes place on the device of a peer which does not have access to the private key. To reduce memory usage of this process, they compute encapsulation operation while public key is streaming from a peer, without buffering it completely. Meanwhile, [20] authors save public key on the flash memory of their board and propose techniques to optimize all three processes of CM.

There have been several attempts to optimize AES since it was introduced. In [22] some platform-dependent instructions are used for AES-128 acceleration. To reduce the number of CPU integer instructions in comparison with a baseline T-table implementation with 68 instructions, they take advantage of the CPU's dedicated instructions.

Stoffelen in [23] provides an optimised assembly implementation of T-Table based AES-128 for RV32I. Encryption of a 16-byte block requires 80 instruction per round. In [24] two custom extensions (saes32.encsm, saes32.encs) are proposed for AES encryption. One round of AES can be implemented by using just 16 SAES32 instructions and a round key load. Marshall et al. [25] propose five different custom ISEs for AES-128 on 32 and 64-bit RISC-V base architectures, optimised for area and latency and evaluate them on SCARV and Rocket cores. A latency-optimised design uses four S-Boxes to achieve one cycle latency for *ShiftRows()* transformation, whereas an area-optimised design uses one S-Box for the transformation. Finally they compare their results with a software-only T-table implementation. In [26], Marshall et al. implement some cryptographic algorithms of the 32-

Authorized licensed use limited to: DUBLIN CITY UNIVERSITY. Downloaded on April 02,2025 at 15:04:36 UTC from IEEE Xplore. Restrictions apply.

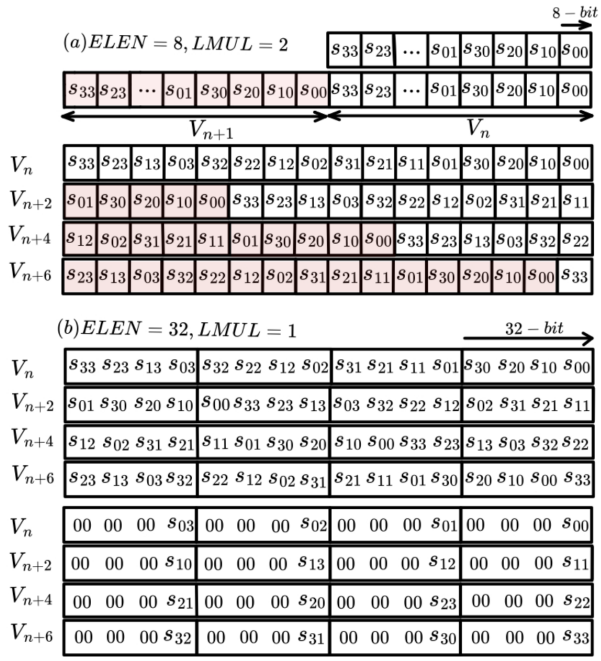

Fig. 2: ShiftRows() transformation in RVV1.0

bit RISC-V scalar cryptography extension v0.7.0, of which version v1.0.1 was ratified in 2022 [27]. Their proposed design for AES is based on [24] and requires 16 dedicated AES instructions per round plus 4 load-word instructions to load the round key. It uses a T-table approach with one S-Box instantiation. Their design evaluation on SCARV core show that AES-128 encryption of a block, can be accomplished in only 296 cycles with 241 instructions using the Crypto ISE instead of 1665 cycles and 1024 instructions in baseline design (RV32IMC). Nisanci et al. in [28] present a software-only implementation of eleven cryptographic algorithms, including AES, using the RV32I ISA and compare their performance to a RISC-V processor with additional hardware modules capable of single-cycle execution. Among the papers mentioned, all use 4 look-up tables to accelerate T-Table-based AES_ECB, while in our case, memory usage must be optimized due to the large key size in CM. So we use Tiny-AES which uses a look-up table only for the S-Box.

IV. Methodology and Progress

In this section, the method for accelerating AES encryption using RVV1.0 is explained. Before applying any transformation on the State matrix, a configuration-setting instruction, *vsetivli*, sets VL to 16, LMUL to 1 and ELEN to 8 (e8m1). Since the State bytes are stored in consecutive addresses in the memory, the 8-bit vector unit-Stride load instruction, (*vle8.v*), can be used to read the whole 16-byte State into a vector register. Vector unit-stride loads/stores can take advantage of the full width of the memory interface to load/store as many elements as possible per transaction if the base address is aligned to the width of the memory interface. If the base address is not suitably aligned, then unit-strided loads/stores are handled like a general strided load/store (with a stride of 1), which also requires one transaction per element [8]. Assuming that all round keys are pre-computed and saved in memory, one needs to load the corresponding key from memory and applies that on the entire State by just one instruction,*vxor.vv*, which executes an XOR operation on two vector register elements.

Since the values of the State bytes are indices for the S-Box table in *SubBytes()*, indexed-unordered load *vluxei8.v* is used to perform the look-ups. With the offset equal to the address of the S-Box, this transformation will be performed in 52 clock cycles just by one instruction. As we do not use any data cache in the Ibex+Vicuna architecture, using look-up tables does not make our design vulnerable to timing attacks.

There are two approaches to perform *ShiftRows()* and *MixColumns()*. One is based on pre-computed indices for shifting State bytes (ν1) and the other implements the shifting process using Vector Slide instructions (ν2). In the first approach of *ShiftRows()*, the State must be saved in memory after being transformed by *SubBytes()*, and re-loaded using *vluxei8.v* like the approach in *SubBytes()*. To minimize memory accesses, this shuffling can be done directly in vector registers, using the *vrgather.vv* instruction. However, this takes more than 100 clock cycles to complete. In the second approach, first the State vector, V_n is copied into the next vector register V_{n+1}, by sliding up V_n for VL=16 elements using a *vslideup.vi* instruction while LMUL=2. The resulting vector group consists of two vector registers V_n and V_{n+1} each containing the State. By applying three *vslidedown.vi* with offsets 5, 10 and 15 to this vector group, we have three shifted versions of the State in vector registers V_{n+2}, V_{n+4} and V_{n+6}, respectively like in Fig. 2(a). The shifted vector groups after changing ELEN to 32, are shown in Fig. 2(b). Since only the least significant byte of every 32-bit element is required, we apply a logical *AND* with 32'hFF to each vector element through the *vand.vi* instruction. *MixColumns()* uses Multiplication in ν1, whereas this is done by a logical shift in ν2, as shown in Fig. 3. By applying a number of logical shift and OR instructions on vectors V_n, V_{n+2}, V_{n+4} and V_{n+6}, one gets vector registers V_i and V_j. Then by running the piece of code in Fig. 3, the output of *MixColumns()* would be in the V_l vector register.

V. Evaluation and Results

As in PQ-RISCV [2], Tiny-AES v1.0.0 is used as the baseline implementation of AES in ECB mode and verified with a key size of 128 bits. Decryption can be optimised like encryption. To compile and build the design, the RISC-V GNU Compiler Toolchain [3] with the -O3 optimization flag, is used. The validation process and performance measurements are done through RTL simulation in Vivado 2021.2 for the PYNQ_Z2 board. First, *Tiny-AES.c*, is compiled for the RV32IMC (Scalar) and then for the RV32IMCV (Vector) architecture. By considering VLEN=128, it is possible to load the whole 16-byte State in a single vector register. Since not all the opcodes for the RVV1.0

[2]https://github.com/scarv/pq-riscv
[3]https://github.com/riscv-collab/riscv-gnu-toolchain

Authorized licensed use limited to: DUBLIN CITY UNIVERSITY. Downloaded on April 02,2025 at 15:04:36 UTC from IEEE Xplore. Restrictions apply.

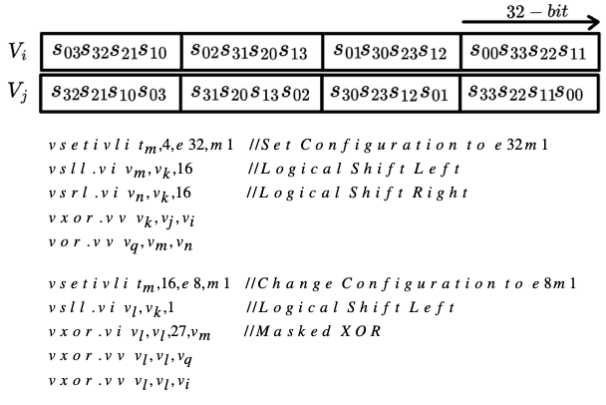

Fig. 3: MixColumns() transformation in RVV1.0

instructions are defined in the Vicuna decoder (e.g., *vlm.v* is not defined), an auto-vectorised design by compiler may not be executable. Thus, some parts of AES are re-written using rvv−intrinsics in C.

TABLE I demonstrates the instruction count and memory footprint which are decreased to about 10% of the scalar design by using RVV1.0. There is no significant difference in static memory for storing the S-Box. Only $\nu 2$ uses 16 bytes more for *ShiftRows()* indices.

TABLE II records the speed-up and area overhead for all variants on different Vicuna configurations. The $\nu 2$, is approximately 2x faster than the scalar design, in the Vicuna compact model, while it reaches 2.5x more performance in the legacy pipeline architecture, where all execution units can operate simultaneously.

AES-128 Variant	ISA	Instruction Count	Program Memory	Static Memory
Scalar	RV32IMC	730	2260	256
$\nu 1$	RV32IMCV	67	250	272
$\nu 2$	RV32IMCV	57	220	256

TABLE I: Instruction count and memory footprint

Ibex+Vicuna Configuration	AES-128 Variant	Cycles (a block)	Cycles/byte	(Ibex+Vicuna) utilization Slice	LUT	FF
No-Vicuna	Scalar	6507	407	1300	3270	2350
compact	$\nu 1$	5961	373	2919	8047	5826
	$\nu 2$	3369	211			
dual	$\nu 1$	5961	373	3507	10521	7193
	$\nu 2$	3162	198			
triple	$\nu 1$	3824	239	4390	14088	8574
	$\nu 2$	2842	178			
legacy	$\nu 1$	5589	350	5027	15885	10454
	$\nu 2$	2715	170			

TABLE II: Cycles and utilization for different configuration

VI. Conclusion

We explore how RISC-V vector extensions can accelerate AES which results in code size reduction and performance enhancement. Our aim in this work is not to compete with other optimised AES implementations such as those using the T-table approach or dedicated hardware units, but to show how AES, as part of Classic McEliece, can be implemented in an optimal way on an existing vector processor. In ongoing work, we started applying this approach to other bottlenecks of Classic McEliece, like Gaussian Elimination in key generation. Our results show that a significant speed-up can be achieved on the Vicuna vector coprocessor in comparison to the baseline Ibex processor. To achieve our goals, we'll add custom vector extensions and change the coprocessor appropriately.

References

[1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, 1997.

[2] G. Alagic et al., "Status report on the third round of the nist post-quantum cryptography standardization process," 2022.

[3] R.-V. Foundation, "Risc-v "v" vector extension , version 1.0," 2021. [Online]. Available: https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf

[4] A. Waterman et al., "Volume I: Unprivileged ISA," p. 238, 2019.

[5] S. Pircher et al., "Exploring the risc-v vector extension for the classic mceliece post-quantum cryptosystem," in *22nd Int. Symp. Quality Electronic Design (ISQED)*, 2021, pp. 401–407.

[6] e. a. Anders, J., "A survey of recent developments in testability, safety and security of risc-v processors," in *2023 IEEE European Test Symposium (ETS)*, 2023.

[7] lowRISC, "Ibex: An embedded 32 bit risc-v cpu core," 2018.

[8] M. Platzer and P. Puschner, "Vicuna: A Timing-Predictable RISC-V Vector Coprocessor for Scalable Parallel Computation," in *33rd Euromicro Conf. on Real-Time Systems*, 2021.

[9] NIST, "Announcing the advanced encryption standard (AES)," 2001.

[10] P. Hellekalek and S. Wegenkittl, "Empirical evidence concerning aes," *ACM Trans. Modeling and Computer Simulation(TOMACS)*, 2003.

[11] R. J. McEliece, "A public-key cryptosystem based on algebraic," *Coding Thv*, 1978.

[12] H. Singh, "Code based cryptography: Classic mceliece," *arXiv preprint arXiv:1907.12754*, 2019.

[13] M. e. a. Sim, "Optimized implementation of encapsulation and decapsulation of classic mceliece on armv8," *IACR Cryptol. ePrint Arch.*, 2022.

[14] T. Chou et al., "Classic mceliece: conservative code-based cryptography," October, 2020.

[15] Y. Zhu et al., "Mckeycutter: A high-throughput key generator of classic mceliece on hardware," in *ACM/IEEE Design Automation Conf.*, 2023.

[16] C. Nugier and V. Migliore, "Acceleration of classic mceliece post-quantum cryptosystem with cache processing," *IEEE Micro*, 2023.

[17] R. Urian and R. Schermann, "Classic mceliece key generation on ram constrained devices," IACR Cryptol. ePrint Arch., 2022.

[18] P.-J. Chen et al., "Complete and improved fpga implementation of classic mceliece," *Cryptol. ePrint Arch.*, 2022.

[19] J. R.-B. et al., "Folding bike: Scalable hardware implementation for reconfigurable devices," Cryptol. ePrint Arch., Paper 2020/897, 2020.

[20] M.-S. Chen and T. Chou, "Classic mceliece on the arm cortex-m4," IACR Cryptol. ePrint Arch., 2021.

[21] J. Roth, E. G. Karatsiolis, and J. Krämer, "Classic mceliece implementation with low memory footprint," in *Smart Card Research and Advanced Application Conf.*, 2020.

[22] D. J. Bernstein and P. Schwabe, "New aes software speed records," in *Int. Conf. on Cryptology in India*. Springer, 2008, pp. 322–336.

[23] K. Stoffelen, "Efficient cryptography on the risc-v architecture," in *Int. Conf. on Cryptology and Information Security in Latin America*. Springer, 2019, pp. 323–340.

[24] M.-J. O. Saarinen, "A lightweight isa extension for aes and sm4," *arXiv preprint arXiv:2002.07041*, 2020.

[25] B. Marshall et al., "The design of scalar aes instruction set extensions for risc-v," *IACR Trans. Crypto. Hardware Embedded Systems*, 2020.

[26] ——, "Implementing the draft risc-v scalar cryptography extensions," in *Proceedings of the 9th Int. Workshop on Hardware and Architectural Support for Security and Privacy*, 2020, pp. 1–8.

[27] R.-V. Foundation, "RISC-V Cryptography Extensions Volume I: Scalar & Entropy Source Instructions , veriosn v1.0.1," p. 105.

[28] G. Nişancı, P. G. Flikkema, and T. Yalçın, "Symmetric cryptography on risc-v: Performance evaluation of standardized algorithms," *Cryptography*, vol. 6, no. 3, p. 41, 2022.

Authorized licensed use limited to: DUBLIN CITY UNIVERSITY. Downloaded on April 02,2025 at 15:04:36 UTC from IEEE Xplore. Restrictions apply.