Article

Efficient Fine-Grained LuT-Based Optimization of AES MixColumns and InvMixColumns for FPGA Implementation

Oussama Azzouzi [1,2], Mohamed Anane [2], Mohamed Chahine Ghanem [3,4,*], Yassine Himeur [5] and Hamza Kheddar [4,6]

[1] Department of Computer Science, Centre Universitaire El Cherif Bouchoucha Aflou, Aflou 03001, Algeria; o_azzouzi@esi.dz
[2] Laboratory of System Design Methods, National Higher School of Computer Science, BP 68M, Algiers 16309, Algeria; m_anane@esi.dz
[3] Cybersecuirty Institute, University of Liverpool, Liverpool L69 3BX, UK
[4] Cyber Secuirty Research Centre, London Metropolitan University, London N7 8DB, UK; kheddar.hamza@univ-medea.dz
[5] College of Engineering and Information Technology, University of Dubai, Dubai 14143, United Arab Emirates; yhimeur@ud.ac.ae
[6] Laboratoire des Systèmes Électroniques Avancés, Department of Electrical Engineering, University of Medea, Medea 26000, Algeria
* Correspondence: mohamed.chahine.ghanem@liverpool.ac.uk

Abstract

This paper presents fine-grained Field Programmable Gate Arrays (FPGA) architectures for the Advanced Encryption Standard (AES) MixColumns and InvMixColumns transformations, targeting improved performance and resource utilization. The proposed method reformulates these operations as boolean functions directly mapped onto FPGA Lookup-Table (LuT) primitives, replacing conventional xor-based arithmetic with memory-level computation. A custom MATLAB-R2019a-based pre-synthesis optimization algorithm performs algebraic simplification and shared subexpression extraction at the polynomial level of Galois Field $GF(2^8)$, reducing redundant logic memory. This architecture, LuT-level optimization minimizes the delay of the complex InvMixColumns stage and narrows the delay gap between encryption (1.305 ns) and decryption (1.854 ns), resulting in a more balanced and power-efficient AES pipeline. Hardware implementation on a Xilinx Virtex-5 FPGA confirms the efficiency of the design, demonstrating competitive performance compared to state-of-the-art FPGA realizations. Its fast performance and minimal hardware requirements make it well suited for real-time secure communication systems and embedded platforms with limited resources that need reliable bidirectional data processing.

Keywords: AES; Rijndael stages; MixColumns/InvMixColumns; LuT-based optimization; Galois Field $GF(2^8)$; FPGA

check for updates

Academic Editors: Jiawei Zhang and Teng Li

Received: 13 November 2025
Revised: 10 December 2025
Accepted: 12 December 2025
Published: 14 December 2025

Citation: Azzouzi, O.; Anane, M.; Ghanem, M.C.; Himeur, Y.; Kheddar, H. Efficient Fine-Grained LuT-Based Optimization of AES MixColumns and InvMixColumns for FPGA Implementation. *Electronics* **2025**, *14*, 4912. https://doi.org/10.3390/electronics14244912

1. Introduction

With the proliferation of secure digital communication, cryptographic algorithms have become fundamental in protecting sensitive information in a wide range of applications, from personal data storage to enterprise-level networking and embedded systems [1–3]. Among these, AES has emerged as the most widely adopted symmetric encryption algorithm due to its strong security guaranties, standardized structure, and suitability for both software and hardware implementations. However, as demand for real-time encryption increases in constrained environments such as Internet of Things (IoT) devices, mobile platforms, and low-power embedded systems, there is a growing need to optimize cryptographic algorithms for performance, power efficiency, and minimal hardware resources [4–6].

The implementation of the AES algorithm on FPGAs is motivated by the combination of parallelism, deterministic timing, and hardware-level flexibility offered by these devices. Unlike general-purpose processors, which rely on sequential instruction execution, FPGAs enable concurrent realization of AES transformations, leading to higher throughput and reduced latency [7,8]. Furthermore, the deterministic execution and energy efficiency of FPGA-based designs are essential for applications operating under strict timing or power constraints, particularly in embedded and IoT systems [9].

From a security perspective, implementing AES directly in reconfigurable hardware reduces exposure to software-based attacks and enables dynamic reconfiguration of encryption cores without modifying the overall architecture [10–12]. Nevertheless, FPGA-based implementations remain vulnerable to physical and side-channel attacks. Power analysis, electromagnetic (EM) emission monitoring, and fault injection attacks can exploit the deterministic and parallel nature of FPGA execution to extract secret keys or disrupt operations [13,14]. Additionally, since FPGA resources such as LuTs and routing channels are shared among logic blocks, attackers may exploit information leakage caused by resource contention or timing variations. Countermeasures such as runtime reconfiguration and masking techniques can mitigate these threats but often introduce trade-offs in area, latency, and power [15,16]. Consequently, while FPGA implementations offer significant performance and flexibility benefits, careful attention to side-channel resistance, fault resilience, and secure synthesis is essential for robust AES deployment in practical systems.

Since the selection of AES by the National Institute of Standards and Technology (NIST), it has remained a central focus of research aiming to improve execution performance, reduce power consumption, and minimize hardware area [17]. Many of these enhancements have been achieved using architectural techniques such as pipelining and parallelism. AES encryption and decryption operate in sequential rounds—ranging from 10 to 14 depending on the key size—each composed of four main operations: ShiftRow/InvShiftRow, MixColumns/InvMixColumns, SubByte/InvSubByte, and AddRoundKey. Among these, the MixColumns operation in encryption and its counterpart InvMixColumns in decryption are the most complex and computationally intensive. Consequently, they represent critical performance bottlenecks in the AES data path [18].

1.1. Research Background and Related Works

The MixColumns and InvMixColumns transformations in AES are matrix multiplications over $GF(2^8)$, defined by fixed matrices in the standard [19]. Because the InvMixColumns matrix contains higher numerical coefficients, decryption typically incurs greater computational complexity and latency than encryption [20]. Although these operations are essential for achieving strong data diffusion by randomizing the AES state columns, they are also among the most power and resource-demanding components of the algorithm, making their optimization a key design objective [21].

Over the past decade, many studies have explored ways to optimize MixColumns and InvMixColumns on FPGAs, aiming to improve throughput, reduce area, and balance encryption and decryption complexity. Early designs often used simple matrix-based or HDL implementations, which limited access to FPGA-specific features. More recent work has applied advanced techniques such as pipelining, composite field arithmetic, and LuT optimization to increase performance and reduce resource usage.

At first, MixColumns and InvMixColumns were implemented as separate hardware blocks. Later research introduced shared-resource designs that reused logic between both transformations, reducing circuit area and redundant computation [18]. Other approaches improved speed through byte-level operations, matrix decomposition, and common subexpression elimination [22]. LuT–based designs also reduced computation time but required more area [23]. Several FPGA-oriented studies proposed low-power solutions using decomposition, precomputation, and parallelism to make AES more energy efficient [21]. Some works further merged both transformations into a single hardware module or restructured InvMixColumns to reuse MixColumns logic, saving additional resources [24,25].

Building on these methods, newer architectures focused on pipelining and fine-grained logic optimization. For example, Rupanagudi et al. [26] optimized MixColumns and S-box stages through datapath restructuring and fine-grained pipelining, achieving about 1.4× faster performance and significant area reduction on Xilinx Artix-7 devices (Xilinx Inc., San Jose, CA, USA). Madhavapandian and MaruthuPandi [27] proposed a modified MixColumns design based on gate replacement and resource sharing for Virtex FPGAs. Their hybrid hardware–software approach demonstrated high operating frequencies and good scalability for network security applications. Prayitno et al. [28] compared Galois Field and LuTs implementations, reporting that their LuT-based MixColumns design used only 28 slice LuTs with a delay of 2.236 ns, while the InvMixColumns required 214 LuTs and 2.7 ns delay. Similarly, Kumar et al. [29] developed an MPPRM-based AES architecture that avoids LuTs in SubBytes/InvSubBytes and integrates MixColumns/InvMixColumns in a high-throughput subpipelined structure. Their Virtex-5 implementation reached about 733 MHz with a delay of 2.095 ns, showing efficient resource utilization across several FPGA families. Despite these improvements, most implementations still rely on high-level synthesis (HLS) and macro-level pipelining, which limits fine-grained control over FPGA resources and low-level boolean optimization.

1.2. Research Gap and Research Question

LuT-based optimization is a common strategy for accelerating cryptographic hardware on FPGAs. In this approach, complex arithmetic operations are reformulated as compact boolean functions mapped directly onto the device's native LuTs, reducing logic depth and intermediate computations. This enables single-cycle evaluation of precomputed outputs, improving latency and throughput while efficiently leveraging FPGA parallelism. Since LuT sizes and architectures vary across FPGA families and synthesis behavior depends heavily on the toolchain, such low-level HDL implementations are naturally device-specific. Nevertheless, porting to another FPGA remains feasible with minor adjustments to accommodate the target LuT architecture.

Despite an extensive body of work on AES MixColumns and InvMixColumns implementations for FPGAs, three recurrent gaps remain. First, many high-performance designs focus on macro-level improvements such as pipelining, parallel datapaths, or composite-field arithmetic, while low-level LuT primitive mapping and boolean-level algebraic simplification receive comparatively little systematic treatment; therefore there is limited empirical evidence linking pre-synthesis algebraic reductions directly to post-place-and-route LuT/slice and timing outcomes. Second, existing resource-sharing strategies typically treat MixColumns and InvMixColumns separately or rely on ad hoc heuristics to identify common subexpressions, reducing the opportunity for systematic, automated LuT consolidation across all output bits. Third, cross-paper comparisons are frequently confounded by differing FPGA families, synthesis/place-and-route tool versions, and timing constraints, making it difficult to determine whether reported benefits stem from algorithmic improvements or platform/tool differences. This work addresses these gaps by (a) expressing MixColumns/InvMixColumns outputs as explicit boolean functions targeted to LuT5/LuT6 primitives, (b) applying an automated pre-synthesis pass to extract and collapse shared subexpressions into LuT-sized input groups, and (c) reporting detailed LuT/slice counts and combinational-delay measurements on a single Xilinx Virtex-5 baseline to enable more direct comparison.

Research question: To what extent can boolean-level, LuT-centric algebraic simplification combined with automated shared-subexpression extraction reduce LuT usage and narrow the encryption/decryption delay gap for AES MixColumns/InvMixColumns on modern Xilinx FPGAs, while preserving practical one- or two-cycle pipeline trade-offs?

1.3. Research Contribution

In this work, we present a hardware-oriented optimization of the MixColumns and InvMixColumns transformations, targeting improved performance and reduced area on Xilinx Virtex-5 FPGAs. The proposed approach minimizes the delay gap between encryption and decryption by exploiting the low-level capabilities of FPGA LuTs and introducing algorithmic optimizations prior to logic synthesis. Main contributions of this work are summarized as follows:

- MixColumns and InvMixColumns are redesigned as boolean functions mapped directly to FPGA LuTs (LuT5/LuT6), replacing xor-based arithmetic with faster and efficient LuT-based computation. This fine-grained LuT-level design, combining algebraic and synthesis optimization, achieves higher efficiency than standard automated synthesis.
- The proposed design lowers the delay of the complex InvMixColumns operation and reduces the gap between encryption (1.305 ns) and decryption (1.854 ns), resulting in a more balanced and energy-efficient AES pipeline. Such balance is especially valuable in real-time communication and secure networking systems, where matched encryption and decryption speeds ensure steady throughput and low latency.
- A custom MATLAB-based algorithm performs algebraic simplification and shared subexpression extraction at the polynomial level of $GF(2^8)$, automatically merging redundant LuTs blocks. This reduces LuT usage from 110×4 to 72×4, achieving about 35% area reduction compared to standard synthesis flows.
- The complete design was implemented and tested on a Xilinx Virtex-5 FPGA. The hardware results confirm superior performance and resource efficiency.

The rest of this paper is organized as follows: Section 2 provides an overview of the AES algorithm discusses arithmetic operations in the $GF(2^8)$ used in AES. Section 3 introduces several low-level optimization strategies for computing the MixColumns and InvMixColumns matrix products. Section 4 presents hardware implementation results on the Xilinx Virtex-5 FPGA along with comparative analysis. Section 5 addresses the ethical considerations. Finally, Section 6 concludes the paper.

2. Preliminaries

2.1. Advanced Encryption Standard

AES is a symmetric block cipher that operates on 128-bit data blocks and employs secret keys of 128, 192, or 256 bits, corresponding to 10, 12, and 14 encryption rounds, respectively [30]. Renowned for its robustness and efficiency, AES is widely implemented across various applications, including router networks, wireless communication systems such as smart meters, and Smart Cards.

Let us now focus on the case of AES with a 128-bit key, as the other cases (AES-192 and AES-256) are similar except for the number of rounds. The plaintext is divided into 128-bit blocks, and each block, along with the encryption key, is transformed into a matrix format. These resulting matrices are square, consisting of 16 elements represented in hexadecimal format, where each element contains one byte. These matrices are referred to as State (for the plaintext) and Key (for the encryption key). Figure 1 illustrates the functional diagram of a single AES round.

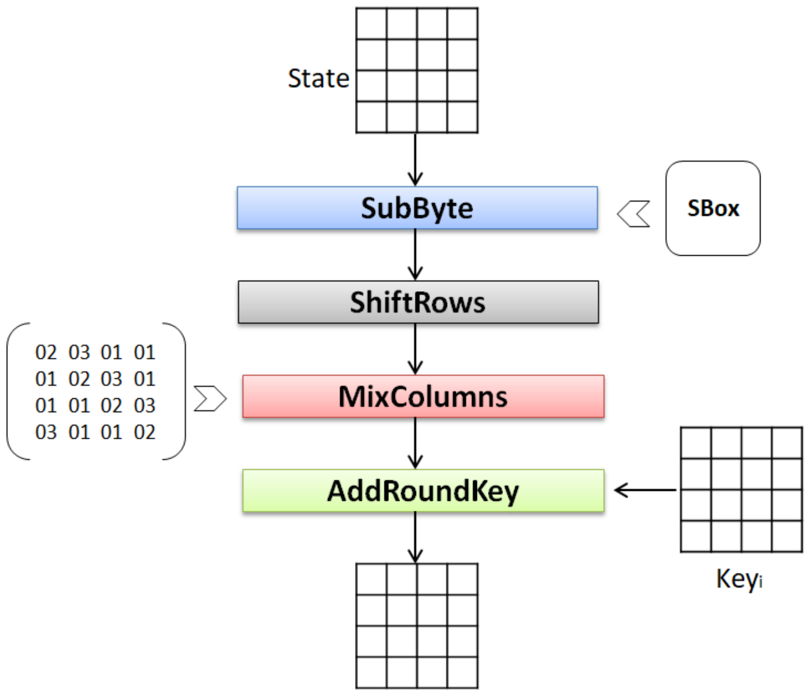


Figure 1. Functional diagram of an AES encryption round.

The algorithm begins with an initial AddRoundKey operation using the encryption key (128 bits), followed by 9 standard rounds, each consisting of four operations: SubBytes, ShiftRows, MixColumns, and AddRoundKey. The final round omits the MixColumns operation. Each round uses its own key, generated through a process called Key Expansion. These keys are either precomputed or calculated at the start of the AES.

The details of the four operations are as follows:

1. SubBytes: This is a substitution in the $GF(2^8)$, where each byte is replaced by another byte from a specific invertible substitution table, the SBox. This nonlinear transformation accounts for 75% of AES's security [30].
2. ShiftRows: This is a transposition step where each element of the State matrix is cyclically shifted to the left by a specific number of columns. Row 0 is not shifted, Row 1 is shifted by 1 byte, Row 2 by 2 bytes, and Row 3 by 3 bytes.
3. MixColumns: This is the most complex operation and requires the most computation in the encryption/decryption process. It involves column-wise matrix multiplication in the $GF(2^8)$, resulting in significant computational complexity. This transformation contributes 15% to AES's security [30].
4. AddRoundKey: This involves the exclusive addition (xor) of each byte in the State matrix with the corresponding byte in a round key RoundKey[i] in the $GF(2^8)$.

In what follows, we focus on the MixColumns and InvMixColumns operations, which are central to this work. We will begin by introducing the arithmetic of the $GF(2^8)$, which is extensively used in the AES and specifically in the targeted operations of this work.

2.2. AES Arithmetic: Galois Field

AES is a byte-oriented encryption algorithm, where each byte consists of 8 bits. The values of these bytes are interpreted as elements in the $GF(2^8)$, a finite field that enables operations like addition and multiplication to be defined over 256 elements. Each element in $GF(2^8)$ can be algebraically represented as a polynomial of degree less than or equal to 7 with binary coefficients [31]. A byte b can be expressed in multiple representations:

- Binary form: $b = \{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$.
- Polynomial form: $b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$, with $b_i \in \{0, 1\}$.
- Hexadecimal form: Two-digit hexadecimal values in $\{00, 01, \ldots, FF\}$.

In this field, addition corresponds to a bitwise xor operation between two bytes, with "00" as the additive identity. Multiplication, however, is more complex; it involves multiplying the corresponding polynomials and reducing the result modulo an irreducible polynomial of degree 8. This reduction ensures the result remains within the 8-bit limit.

The irreducible polynomial used in AES is defined shown in Equation (1).

$$M(x) = x^8 + x^4 + x^3 + x + 1 \quad \text{(hexadecimal: 0x11B)} \tag{1}$$

This polynomial cannot be factored into lower-degree polynomials over $GF(2)$, which ensures the field's algebraic structure remains intact.

AES matrix operations such as MixColumns and InvMixColumns rely heavily on multiplications in $GF(2^8)$ using predefined constant matrices. These matrices are:

$$\text{MixColumns matrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \quad \text{and} \quad \text{InvMixColumns matrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \tag{2}$$

The MixColumns operation involves computing the product of a state column vector with the MixColumns matrix. Similarly, InvMixColumns performs the reverse transformation. These matrix multiplications correspond to solving systems of equations where each output byte S_i is computed as:

$$\begin{cases} S_0 = (\{02\} * E_0) \oplus (\{03\} * E_1) \oplus E_2 \oplus E_3 \\ S_1 = E_0 \oplus (\{02\} * E_1) \oplus (\{03\} * E_2) \oplus E_3 \\ S_2 = E_0 \oplus E_1 \oplus (\{02\} * E_2) \oplus (\{03\} * E_3) \\ S_3 = (\{03\} * E_0) \oplus E_1 \oplus E_2 \oplus (\{02\} * E_3) \end{cases} \tag{3}$$

$$\begin{cases} S'_0 = (\{0e\} * E_0) \oplus (\{0b\} * E_1) \oplus (\{0d\} * E_2) \oplus (\{09\} * E_3) \\ S'_1 = (\{09\} * E_0) \oplus (\{0e\} * E_1) \oplus (\{0b\} * E_2) \oplus (\{0d\} * E_3) \\ S'_2 = (\{0d\} * E_0) \oplus (\{09\} * E_1) \oplus (\{0e\} * E_2) \oplus (\{0b\} * E_3) \\ S'_3 = (\{0b\} * E_0) \oplus (\{0d\} * E_1) \oplus (\{09\} * E_2) \oplus (\{0e\} * E_3) \end{cases} \tag{4}$$

From these equations, it is evident that the core operations required for MixColumns and InvMixColumns are multiplications by constants in $GF(2^8)$ and xor additions.

To implement these multiplications efficiently in hardware, the AES defines specific functions like `xtime_x()`, where x represents a constant multiplier (e.g., 02, 03, 09, 0B). Let B be an arbitrary element in $GF(2^8)$, represented as a polynomial:

$$B(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \tag{5}$$

2.2.1. Multiplication by {02} in $GF(2^8)$ – `xtime_2()`

The function `xtime_2()` multiplies $B(x)$ by x (i.e., by {02}) and reduces the result modulo $M(x)$. This corresponds to a left bit shift followed by a conditional xor with 0x1B if the most significant bit (MSB) is 1. The corresponding hardware implementation, depicted in Figure 2, uses 3 xor gates with a critical path (CP) of 1.

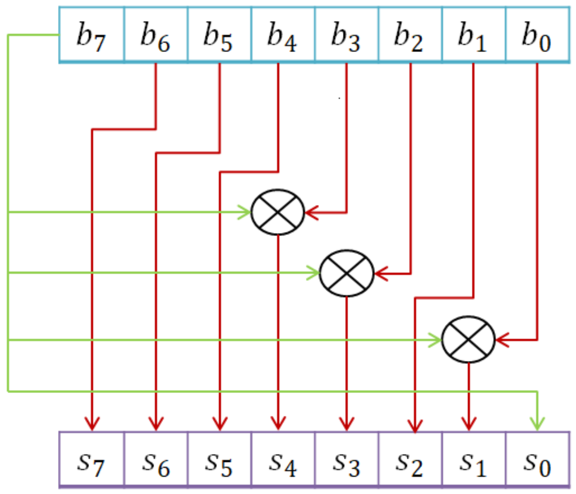


Figure 2. Multiplier by {02} in $GF(2^8)$ (`xtime_2()`).

2.2.2. Multiplication by {03} in $GF(2^8)$ – `xtime_3()`

The function `xtime_3()` is implemented as

$$\texttt{xtime\_3()} = \texttt{xtime\_2()}(B) \oplus B \tag{6}$$

which means multiplying by {02} and then xoring (adding) the original byte. This requires 9 xor gates with a CP of 2, as illustrated in Figure 3.

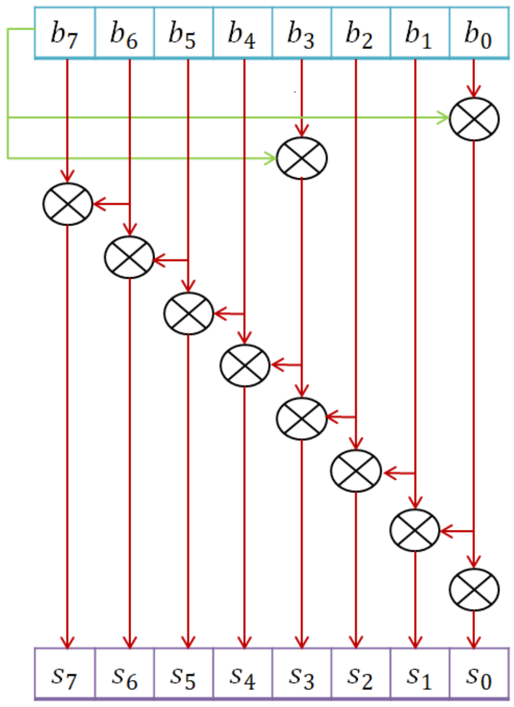


Figure 3. Multiplier by {03} in $GF(2^8)$ (`xtime_3()`).

By generalizing this methodology and composing multiple `xtime_x()` functions, more complex multiplications (e.g., by {09}, {0B}, {0D}, {0E}) used in InvMixColumns can be efficiently implemented. Table 1 summarizes the logic gate complexity and CP delay for each multiplication constant used in the MixColumns and InvMixColumns operations. As shown, the most computationally expensive multiplication is `xtime_0E()`, which requires 16 xor gates and exhibits a CP of 3.

Table 1. Xor gate count and CP for each `xtime_k()` function.

Multiplication	Number of Xor Gates	Critical Path (CP)
xtime_02()	3	1
xtime_03()	9	2
xtime_04()	5	2
xtime_08()	7	2
xtime_09()	13	2
xtime_0B()	16	3
xtime_0C()	12	3
xtime_0D()	16	3
xtime_0E()	16	3

3. Proposed Optimization of MixColumns and InvMixColumns

The main goal of this section is to optimize the MixColumns and InvMixColumns operations to enhance AES hardware performance. Since InvMixColumns involves more complex Galois Field multiplications, it typically incurs higher latency than MixColumns. Therefore, reducing this delay gap is essential to achieve balanced encryption and decryption throughput, particularly in real-time communication and secure networking applications.

The proposed approach exploits the native hardware resources of Xilinx FPGAs—specifically LuTs to efficiently implement both operations. Each LuT acts as a small truth table that stores precomputed logic outputs for all possible input combinations, replacing complex gate-level computations with a single memory access. This results in notable speed improvements, especially for repetitive AES transformations.

The LuT6 component, which is a 2^6-bit memory addressed by 6 bits and returns a single output bit, was selected for this implementation. By expressing each output bit of the MixColumns and InvMixColumns transformations as a boolean function of input bits, these functions can be directly mapped to LuTs.

3.1. Optimized Implementation of MixColumns

MixColumns processes four 8-bit input vectors $\{E_0, E_1, E_2, E_3\}$ to produce four 8-bit outputs $\{S_0, S_1, S_2, S_3\}$. Using Equation (3), the logical expression for each bit of S_0 can be derived. For example, the boolean logic for each bit of S_0 can be derived, as illustrated in Equation (7).

$$\begin{cases} S_0(0) = E_0(7) \oplus E_1(0) \oplus E_1(7) \oplus E_2(0) \oplus E_3(0) \\ S_0(1) = E_0(0) \oplus E_0(7) \oplus E_1(0) \oplus E_1(1) \oplus E_1(7) \oplus E_2(1) \oplus E_3(1) \\ S_0(2) = E_0(1) \oplus E_1(1) \oplus E_1(2) \oplus E_2(2) \oplus E_3(2) \\ S_0(3) = E_0(2) \oplus E_0(7) \oplus E_1(2) \oplus E_1(3) \oplus E_1(7) \oplus E_2(3) \oplus E_3(3) \\ S_0(4) = E_0(3) \oplus E_0(7) \oplus E_1(3) \oplus E_1(4) \oplus E_1(7) \oplus E_2(4) \oplus E_3(4) \\ S_0(5) = E_0(4) \oplus E_1(4) \oplus E_1(5) \oplus E_2(5) \oplus E_3(5) \\ S_0(6) = E_0(5) \oplus E_1(5) \oplus E_1(6) \oplus E_2(6) \oplus E_3(6) \\ S_0(7) = E_0(6) \oplus E_1(6) \oplus E_1(7) \oplus E_2(7) \oplus E_3(7) \end{cases} \quad (7)$$

Bits such as $S_0(0), S_0(2), S_0(5), S_0(6), S_0(7)$ are derived from five inputs and can be directly mapped onto a LuT5. Other bits, such as $S_0(1), S_0(3), S_0(4)$, require seven inputs and need to be optimized or approximated using shared logic or multi-LuT configurations. Figure 4 illustrates a sample implementation of $S_0(0)$ and $S_0(1)$ using LuTs. This implementation requires only one clock cycle per MixColumns transformation, as each output is retrieved via a single memory access.

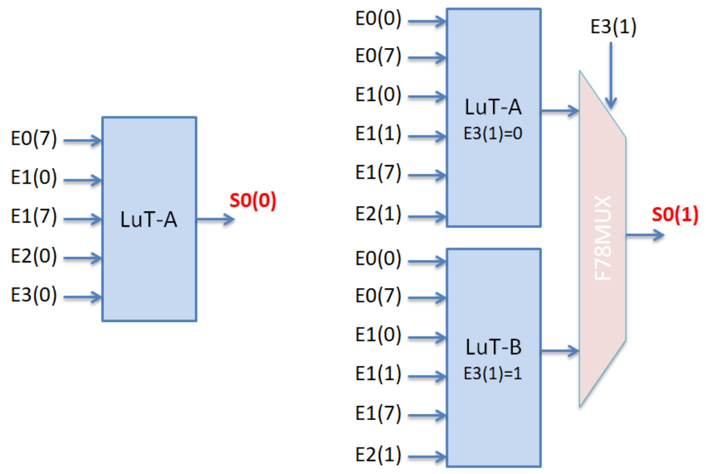


Figure 4. Hardware implementation of $S_0(0)$ and $S_0(1)$ using LuTs for the MixColumns.

3.2. Optimized Implementation of InvMixColumns

The InvMixColumns operation, due to its heavier polynomial multipliers (e.g., {0e}, {0b}, etc.), is more complex than MixColumns. However, a similar LuT-based approach can be applied to optimize its execution. Starting from the boolean expressions derived from Equation (4), each output byte S'_0 can be represented as a function of the input bits.

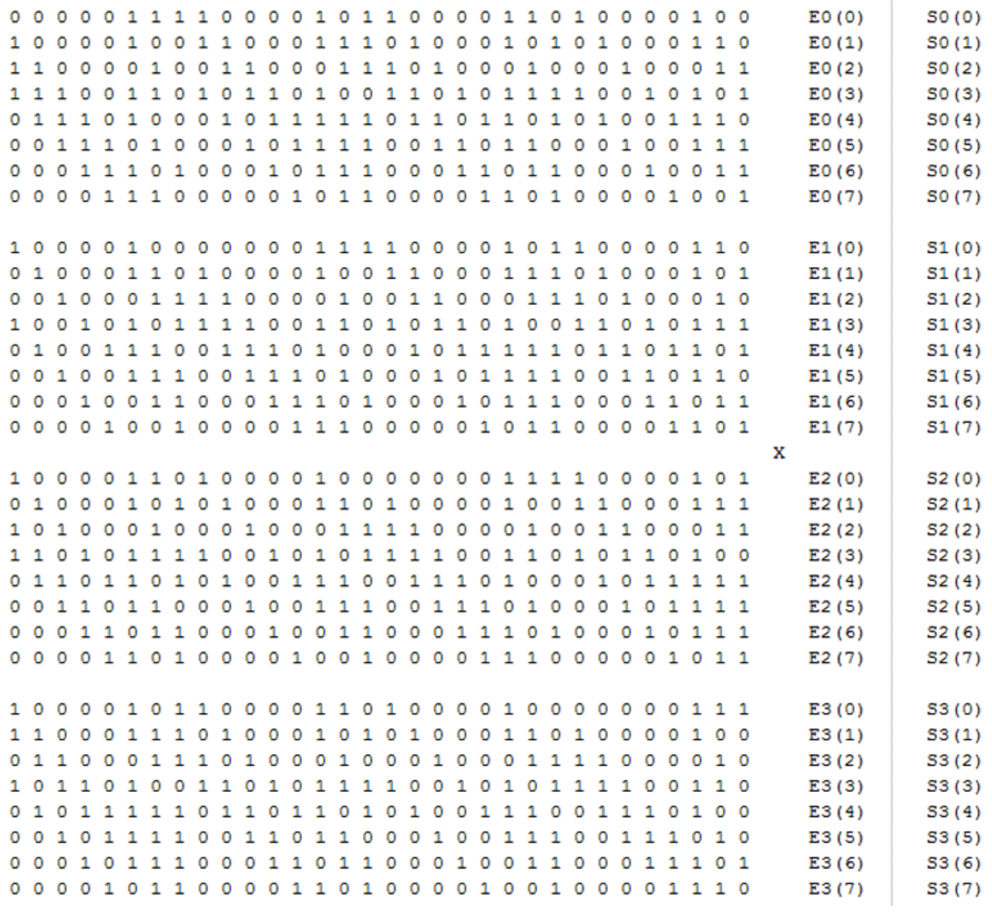
$$\begin{cases} S'_0(0) = (E_0(7) \oplus E_1(0) \oplus E_1(7) \oplus E_2(0) \oplus E_3(0)) \oplus (E_0(6)) \oplus E_2(6))) \oplus (E_0(5) \oplus E_2(5) \oplus E_1(5) \oplus E_3(5)) \\ S'_0(1) = (E_0(0) \oplus E_1(0) \oplus E_1(1) \oplus E_1(7) \oplus E_2(1) \oplus E_3(1)) \oplus E_2(7) \oplus (E_1(6) \oplus E_3(6) \oplus E_0(5) \oplus E_2(5) \\ \qquad \oplus E_1(5) \oplus E_3(5)) \\ S'_0(2) = (E_0(1) \oplus E_1(1) \oplus E_1(2) \oplus E_2(2) \oplus E_3(2)) \oplus ((E_0(0) \oplus E_2(0)) \oplus (E_0(6) \oplus E_2(6) \oplus E_1(7) \oplus E_3(7) \\ \qquad \oplus E_1(6) \oplus E_3(6))) \\ S'_0(3) = (E_0(2) \oplus E_1(2) \oplus E_1(3) \oplus E_2(3) \oplus E_3(3)) \oplus ((E_0(1) \oplus E_2(1) \oplus E_0(6) \oplus E_2(6)) \oplus (E_0(0) \oplus E_2(0) \\ \qquad \oplus E_2(7) \oplus E_1(0) \oplus E_3(0) \oplus E_3(7) \oplus E_0(5) \oplus E_2(5) \oplus E_1(5) \oplus E_3(5))) \\ S'_0(4) = (E_0(3) \oplus E_1(3) \oplus E_1(4) \oplus E_2(4) \oplus E_3(4) \oplus E_1(7)) \oplus ((E_0(2) \oplus E_2(2) \oplus E_2(7)) \oplus (E_0(1) \oplus E_2(1) \\ \qquad \oplus E_1(1) \oplus E_3(1) \oplus E_1(6) \oplus E_3(6) \oplus E_0(5) \oplus E_2(5) \oplus E_1(5) \oplus E_3(5))) \\ S'_0(5) = (E_0(4) \oplus E_1(4) \oplus E_1(5) \oplus E_2(5) \oplus E_3(5)) \oplus ((E_0(3) \oplus E_2(3)) \oplus (E_0(2) \oplus E_2(2) \oplus (E_0(6) \oplus E_2(6) \\ \qquad \oplus E_1(2) \oplus E_1(7) \oplus E_3(2) \oplus E_3(7) \oplus E_1(6) \oplus E_3(6))) \\ S'_0(6) = (E_0(5) \oplus E_1(5) \oplus E_1(6) \oplus E_2(6) \oplus E_3(6)) \oplus ((E_0(4) \oplus E_2(4)) \oplus (E_0(3) \oplus E_0(7) \oplus (E_2(3) \oplus E_2(7) \\ \qquad \oplus E_1(3) \oplus E_1(7) \oplus E_3(3) \oplus E_3(7))) \\ S'_0(7) = (E_0(6) \oplus E_1(6) \oplus E_1(7) \oplus E_2(7) \oplus E_3(7)) \oplus ((E_0(5) \oplus E_2(5)) \oplus (E_0(4) \oplus E_2(4) \oplus E_1(4) \oplus E_3(4))) \end{cases} \tag{8}$$

The proposed implementation theoretically requires only one clock cycle for the MixColumns operation, as each output can be retrieved through a single memory access. This design approach proves even more beneficial for the InvMixColumns operation, which is typically more complex due to heavier arithmetic in $GF(2^8)$. To realize this optimization, an architecture was constructed using xor logic gates, similar to the MixColumns implementation. Each output bit was expressed as a boolean function of the corresponding input bits. For example, the output byte S'_0 is described using the set of equations given in Equation (8).

Ultimately, four such systems are generated—one for each output byte—and can be implemented similarly to the MixColumns process. Initially, the complete implementation required approximately 110 $\times$ 4 LuTs. However, further analysis revealed that several logic expressions shared common inputs, allowing for optimization through logic reuse.

To exploit this, the architecture was reformulated as a matrix multiplication problem, called M. A 32 $\times$ 32 binary matrix, composed of zeros and ones, was used to represent the relationship between input bits $\{E_0(0), E_0(1), \ldots, E_3(7)\}$ and output bits $\{S_0(0), S_0(1), \ldots, S_3(7)\}$. This setup transformed the hardware design into a combinatorial optimization task.

The MATLAB-based optimization aims to reduce the number of 6-input Look-Up Tables (LuTs) needed for the InvMixColumns transformation by identifying redundant logic and reusing common input signals across multiple outputs. The algorithm begins with a 32 × 32 binary matrix, as shown in Figure 5, representing the transformation and an empty list to store the optimized LuTs. It then iteratively processes the outputs that still need to be mapped. In each step, it examines the matrix columns, selects the six most influential input bits, and creates a LuT for them. If no outputs can be mapped with this selection, the algorithm tries the next best combination. Once a LuT is generated, its outputs are removed from the remaining list. This process continues until all outputs are assigned, and the algorithm finally returns the optimized LuTs along with the total number used. Algorithm 1 shows the pseudocode for MATLAB-based InvMixColumns optimization.



```
0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0     E0(0)   S0(0)
1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 0     E0(1)   S0(1)
1 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1     E0(2)   S0(2)
1 1 1 0 0 1 1 0 1 0 1 1 0 1 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 1     E0(3)   S0(3)
0 1 1 1 0 1 0 0 0 1 0 1 1 1 1 1 0 1 1 0 1 1 0 1 0 1 0 0 1 1 1 0     E0(4)   S0(4)
0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 1 0 0 1 1 0 1 1 0 0 0 1 0 0 1 1 1     E0(5)   S0(5)
0 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 0 1 1     E0(6)   S0(6)
0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1     E0(7)   S0(7)

1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0     E1(0)   S1(0)
0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 1     E1(1)   S1(1)
0 0 1 0 0 0 1 1 1 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0     E1(2)   S1(2)
1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 0 1 1 0 1 0 0 1 1 0 1 0 1 1 1     E1(3)   S1(3)
0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 1 1 0 1 1 0 1 1 0 1     E1(4)   S1(4)
0 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 1 0 0 1 1 0 1 1 0     E1(5)   S1(5)
0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 1 1     E1(6)   S1(6)
0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1     E1(7)   S1(7)
                                                                  X
1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0 1     E2(0)   S2(0)
0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1 1     E2(1)   S2(1)
1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 1 0 0 1 1 0 0 0 1 1     E2(2)   S2(2)
1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 0 1 0 1 1 0 1 0 0     E2(3)   S2(3)
0 1 1 0 1 1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 1 1     E2(4)   S2(4)
0 0 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1 1     E2(5)   S2(5)
0 0 0 1 1 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 1 1 1     E2(6)   S2(6)
0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 1 0 1 1     E2(7)   S2(7)

1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1     E3(0)   S3(0)
1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 0     E3(1)   S3(1)
0 1 1 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 1 1 0 0 0 0 1 0     E3(2)   S3(2)
1 0 1 1 0 1 0 0 1 1 0 1 0 1 1 1 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 0     E3(3)   S3(3)
0 1 0 1 1 1 1 1 0 1 1 0 1 1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0 0     E3(4)   S3(4)
0 0 1 0 1 1 1 1 0 0 1 1 0 1 1 0 0 0 1 0 0 1 1 1 0 0 1 1 1 0 1 0     E3(5)   S3(5)
0 0 0 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 1 0 0 1 1 0 0 0 1 1 1 0 1     E3(6)   S3(6)
0 0 0 0 1 0 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 0 0 1 0 0 0 0 1 1 1 0     E3(7)   S3(7)
```

Figure 5. Matrix M.

Algorithm 1 Pseudocode: MATLAB-Based LuT Optimization of InvMixColumns

```
1:  Start with $M$ {32 × 32} binary matrix and a LuT6 inputs
2:  Initialize an empty list called optimized_LuTs
3:  Consider all output bits of $M$ as remaining_outputs
4:  While there are still remaining_outputs do the following steps:
5:      Compute $M$ columns into remaining_outputs and sort them.
6:      Select top 6 inputs and find outputs depending on them.
7:      If no outputs match, try the next top 6-input combination.
8:      Create LuT with selected inputs and map outputs; add to optimized_LuTs.
9:      Remove all mapped outputs from remaining_outputs.
10: Repeat until all outputs are mapped to LuTs.
11: Return optimized_LuTs and total LuTs used.
```

Through this automated optimization, the total number of LuTs was reduced from 110×4 to 72×4, significantly improving area efficiency. The final InvMixColumns implementation requires only two clock cycles, corresponding to two memory access stages, as illustrated in Figure 6.

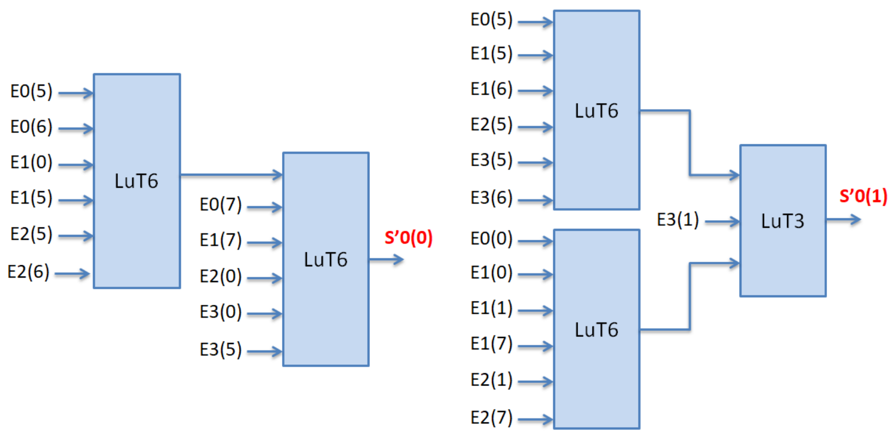


Figure 6. Hardware implementation of $S_0'(0)$ and $S_0'(1)$ using LuTs for the InvMixColumns.

4. Implementation and Result Analysis

The proposed architectures were implemented on a Xilinx Virtex-5 Genesys FPGA using the ISE Design Suite (version 14.7). The hardware was described in VHDL at the LuT level, and its functionality was verified through ISim simulations. The operating frequencies shown in Table 2 correspond to the maximum values obtained after the place-and-route stage, under default timing constraints. A 100 MHz clock frequency was used to ensure fair comparison with previous FPGA-based implementations. Both post-synthesis and post-routing analyses were performed to confirm the accuracy of the frequency and delay results.

Table 2. Comparison of results from MixColumns and InvMixColumns implementations.

Ref.	Platform	Freq. MHz	Architecture	MixColumns			InvMixColumns		
				Slices	LuTs	Times (ns)	Slices	LuTs	Times (ns)
Our	Virtex-5	100	xor	12	42	1.266	30	82	2.233
			LuTs	12	44	1.305	21	73	1.854
[27]	Virtex-6	315.806	HW	38	152	1.762	-	-	-
[28]	MATLAB-FPGA	-	HW	-	28	2.236	-	214	2.7
[26]	Artix-7	-	HW	32	128	1.478	79	316	2.084

The Virtex-5 Genesys FPGA is a high-performance, reconfigurable platform based on 65 nm CMOS technology, providing high logic density, speed, and energy efficiency. It is built around CLBs containing LuTs and flip-flops, which allow efficient implementation of complex digital operations such as cryptographic transformations. The device also includes DSP48E slices, Block RAMs, and Clock Management Tiles (CMTs) for fast arithmetic processing and stable timing. The Genesys board offers multiple I/O interfaces—DDR2 memory, USB, Ethernet, and VGA—making it well suited for both research and prototype development. Supported by the Xilinx ISE Design Suite, it enables complete synthesis, simulation, and timing analysis.

In this work, two hardware architectures were designed for the MixColumns and InvMixColumns operations. The first, unoptimized, is based on xor gates, as defined by Equations (3) and (4). The second, optimized, uses LuT-based implementations, as shown in Figures 4 and 6.

Table 2 presents a detailed comparison between the proposed MixColumns and InvMixColumns implementations and several representative FPGA-based designs reported in the literature. The comparison focuses on three primary criteria: propagation delay (ns), resource utilization (slices and LuTs), and target FPGA platform.

The proposed MixColumns design achieves a propagation delay of 1.305 ns using 44 LuTs on a Virtex-5 FPGA, outperforming the implementation of Rupanagudi et al. [26] (1.478 ns) and showing significant improvement over those of Prayitno et al. [28] (2.236 ns) and Madhavapandian et al. [27] (1.762 ns). Although our system operates at a lower test frequency (100 MHz versus 315.8 MHz in [27]), it achieves a shorter combinational delay thanks to its compact boolean formulation and optimized LuT mapping.

For InvMixColumns, which generally presents higher computational complexity, the proposed LuT-optimized design achieves a delay of 1.854 ns, outperforming [28] (2.7 ns) and [26] (2.084 ns). The reduced timing discrepancy between MixColumns and InvMixColumns (1.305 ns vs. 1.854 ns) leads to a more balanced AES pipeline, improving throughput consistency and minimizing decryption latency overhead.

From an area perspective, the proposed architecture demonstrates notable resource efficiency. The MixColumns transformation requires only 44 LuTs and 12 slices, compared to 128 LuTs and 32 slices in [26] and 152 LuTs and 38 slices in [27]. Likewise, the InvMixColumns implementation occupies just 73 LuTs and 21 slices, representing a reduction of more than 60% relative to previous designs such as [28] (214 LuTs) and [26] (316 LuTs). These results confirm the effectiveness of the LuT-level decomposition and algebraic simplification strategy, which together minimize logic depth and hardware footprint.

Unlike earlier approaches that rely on the direct synthesis of high-level HDL or matrix-multiplication modules, the proposed design performs algorithmic-level optimization prior to synthesis. AES MixColumns transformations are expressed as boolean systems and implemented directly using FPGA-native LuT primitives, ensuring reduced logic depth, balanced routing, and lower power dissipation.

Overall, when delay, area, and architectural balance are considered together, the proposed implementation achieves the most favorable trade-off among the referenced works. It combines low latency and compact area with consistent encryption and decryption performance, making it highly suitable for embedded and reconfigurable cryptographic accelerators where both speed and area efficiency are essential.

To further illustrate this comparison, execution time and area utilization were analyzed, as summarized in Table 2 and depicted in Figure 7. The results clearly show that the LuT-based design outperforms the xor-based one, especially for the InvMixColumns operation. This performance advantage increases with the number of variables involved in each output equation. Figure 7 demonstrates that as the variable count grows, the differences in delay and area between the two architectures become more pronounced, confirming that the LuT-based approach effectively narrows the execution-time gap between MixColumns and InvMixColumns, thereby aligning decryption performance more closely with encryption in the AES algorithm. Comparative results analysis with related works is summarized in Table 3.

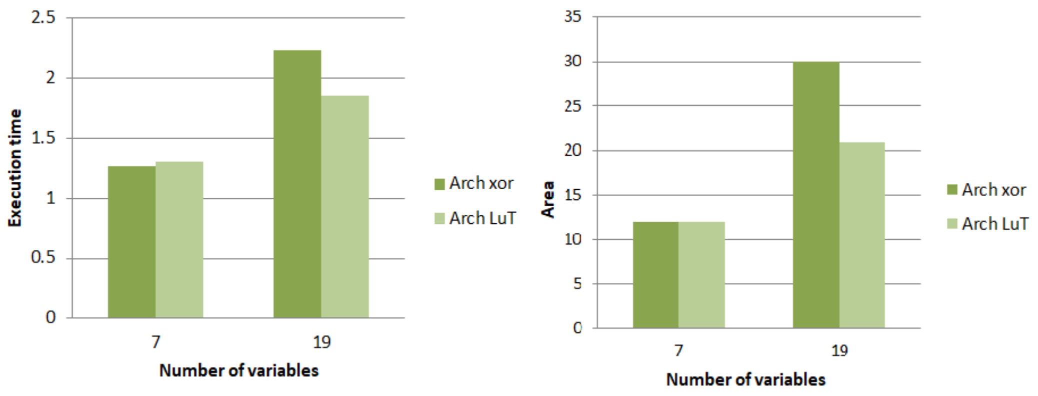


Figure 7. Comparison between our architectures in terms of execution time and occupied area.

Table 3. Comparison of proposed approach with related works.

Ref.	Platform	Approach	Key Reported Results (Delay/LuTs/Slices)
Our Work	Virtex-5	LuT-level boolean mapping + Pre-synthesis	Mix: 1.305 ns, 44 LuTs, 12 slices. Inv: 1.854 ns, 73 LuTs, 21 slices.
[26]	Artix-7	Datapath restructuring, fine pipelining	Mix: 1.478 ns, 128 LuTs, 32 slices; Inv: 2.084 ns, 316 LuTs.
[27]	Virtex-6	Gate replacement and resource sharing	Mix: 1.762 ns, 152 LuTs, 38 slices.
[28]	MATLAB	Direct $GF(2^8)$ vs. LuT comparisons	Mix: 2.236 ns, 28 LuTs; Inv: 2.7 ns, 214 LuTs.
[29]	Virtex-5	MPPRM-based architecture, subpipelined	High freq (reported $\approx$ 733 MHz), delay $\approx$ 2.095 ns.

5. Ethical Considerations

This work presents a hardware-focused optimization of AES MixColumns and InvMixColumns for Xilinx Virtex-5 FPGAs. By redesigning these operations as boolean functions mapped directly to LuT5/LuT6 and replacing xor-based arithmetic with LuT-based computation, the approach reduces the decryption delay and narrows the encryption–decryption gap, creating a more balanced AES pipeline. A MATLAB-based algorithm further simplifies the design and cuts LuT usage by $\approx$ 35%. FPGA implementation confirms faster performance and lower resource use, making it suitable for real-time secure applications.

6. Conclusions

This work presented optimized hardware architectures for the AES MixColumns and InvMixColumns transformations using a fine-grained LuT-based decomposition. The proposed designs improve both speed and area efficiency on Xilinx Virtex-5 FPGAs while significantly reducing the delay gap between encryption and decryption—especially for the more complex InvMixColumns operation. By mapping boolean equations directly onto FPGA LuT primitives, the implementation achieves competitive performance compared to existing designs.

Although the current work targets AES-128, the proposed architectures are easily scalable to AES-192 and AES-256, since the MixColumns and InvMixColumns transformations are identical across all key sizes. Extending the design to longer keys mainly increases area and latency proportionally to the number of rounds.

While the evaluation in this study was limited to the Virtex-5 platform, the proposed architectures are inherently compatible with newer FPGA families such as Artix-7, Zynq, and UltraScale. This compatibility stems from the fact that the design exclusively relies on FPGA-native LuTs and combinational boolean logic, without depending on device-specific hard blocks. The underlying logic and mapping methodology are therefore portable across different Xilinx FPGA generations, ensuring similar synthesis and routing behavior. Synthesis results on more recent FPGA families will be investigated in future work to further validate this portability.

Furthermore, although power analysis was not performed in the current study, it represents an important direction for future work, particularly for embedded and IoT applications where energy efficiency is critical. Future research will include detailed power modeling, adaptive optimization, and AI-assisted design exploration. Additionally, integrating the proposed modules into a complete AES pipeline and validating them in real-time secure communication systems will further demonstrate their practical value.

Author Contributions: Conceptualization, O.A., M.A., M.C.G., Y.H. and H.K.; Methodology, O.A., M.A., M.C.G., Y.H. and H.K.; Software, O.A., M.A., M.C.G., Y.H. and H.K. ; Validation, O.A., M.A., M.C.G., Y.H. and H.K.; Formal Analysis, O.A., M.A., M.C.G., Y.H. and H.K.; Investigation, O.A., M.A., M.C.G., Y.H. and H.K.; Resources, O.A., M.A., M.C.G., Y.H. and H.K.; Data Curation, O.A., M.A., M.C.G., Y.H. and H.K.; Writing—Original Draft, O.A., M.A., M.C.G., Y.H. and H.K.; Writing—Review and Editing, O.A., M.A., M.C.G., Y.H. and H.K.; Visualization, O.A., M.A., M.C.G., Y.H. and H.K.; Supervision, O.A., M.A., M.C.G., Y.H. and H.K.; Project Administration, O.A., M.A., M.C.G., Y.H. and H.K.; Funding Acquisition, O.A. and M.A. All authors have read and agreed to the published version of the manuscript.

Funding: The Open Access (OA) fee for this paper was funded by the University of Liverpool.

Institutional Review Board Statement: This research was deemed not to require the University's Ethical Committee approval, as it does not fall under any of the cases requiring ethical approval.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Silva, C.; Cunha, V.A.; Barraca, J.P.; Aguiar, R.L. Analysis of the cryptographic algorithms in IoT communications. *Inf. Syst. Front.* **2024**, *26*, 1243–1260.
2. Singh, A.; Rathee, G.; Kerrache, C.A.; Ghanem, M.C. A relay-chain-powered ciphertext-policy attribute-based encryption in intelligent transportation systems. *Transp. Eng.* 2025, *In Press*.
3. Ghanem, M.C.; Ratnayake, D.N. Enhancing WPA2-PSK four-way handshaking after re-authentication to deal with de-authentication followed by brute-force attack: A novel re-authentication protocol. In Proceedings of the 2016 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA), London, UK, 13–14 June 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–7.
4. Opilka, F.; Niemiec, M.; Gagliardi, M.; Kourtis, M.A. Performance analysis of post-quantum cryptography algorithms for digital signature. *Appl. Sci.* **2024**, *14*, 4994.
5. Azzouzi, O.; Anane, M.; Koudil, M.; Issad, M.; Himeur, Y. Novel area-efficient and flexible architectures for optimal Ate pairing on FPGA. *J. Supercomput.* **2024**, *80*, 2633–2659.
6. Azzouzi, O.; Anane, M.; Ghanem, M.C.; Himeur, Y.; Wojtczak, D. Flexible and area-efficient codesign implementation of AES on FPGA. *Cryptography* **2025**, *9*, 78.
7. Shahbazi, K.; Ko, S.B. High throughput and area-efficient FPGA implementation of AES for high-traffic applications. *IET Comput. Digit. Tech.* **2020**, *14*, 274–282.
8. Rahmanpour, M.; Zavare, A.A. Very high throughput implementation of advanced encryption standard (AES) algorithm on FPGA. *Signal Process. Renew. Energy* **2017**, *1*, 1–8.
9. Nguyen, V.T.; Le, X.H.; Tran, D.H.; Ngo, Q.T.; Nguyen, H.T. AES-RV: Hardware-efficient RISC-V accelerator with low-latency AES instruction extension for IoT security. *arXiv* **2025**, arXiv:2505.11880.

10. Khairallah, M.; Chattopadhyay, A.; Peyrin, T. Looting the LUTs: FPGA optimization of AES and AES-like ciphers for authenticated encryption. In Proceedings of the International Conference on Cryptology in India, Chennai, India, 10–13 December 2017; Springer: Berlin/Heidelberg, Germany, 2017; pp. 282–301.
11. Abdulsamad, A.A.; Répás, S.R. Application of FPGA devices in network security: A survey. *Electronics* **2021**, *14*, 3894.
12. Oussama, A.; Mohamed, A.; Nassim, H. Software implementation of pairing based cryptography on FPGA. In Proceedings of the International Conference on Computer Science and its Applications, Melbourne, VIC, Australia, 2–5 July 2018; Springer: Berlin/Heidelberg, Germany, 2018; pp. 102–112.
13. Aljuffri, A.; Huang, R.; Muntenaar, L.; Gaydadjiev, G.; Ma, K.; Hamdioui, S.; Taouil, M. The Security Evaluation of an Efficient Lightweight AES Accelerator. *Cryptography* **2024**, *8*, 24.
14. Khalaj Monfared, S.; Forte, D.; Tajik, S. Randohm: Mitigating impedance side-channel attacks using randomized circuit configurations. In Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design, New York, NY, USA, 27–31 October 2024; pp. 1–9.
15. Gigerl, B.; Klug, F.; Mangard, S.; Mendel, F.; Primas, R. Smooth passage with the guards: Second-order hardware masking of the AES with low randomness and low latency. *IACR Trans. Cryptogr. Hardw. Embed. Syst.* **2024**, *2024*, 309–335.
16. Srivastava, A.; Miftah, S.S.; Kim, H.; Pal, D.; Basu, K. PoSyn: Secure Power Side-Channel Aware Synthesis. *arXiv* **2025**, arXiv:2506.08252.
17. Ajagbe, S.A.; Adeniji, O.D.; Olayiwola, A.A.; Abiona, S.F. Advanced encryption standard (AES)-based text encryption for near field communication (NFC) using Huffman compression. *SN Comput. Sci.* **2024**, *5*, 156.
18. Fischer, V.; Drutarovsky, M.; Chodowiec, P.; Gramain, F. InvMixColumn decomposition and multilevel resource sharing in AES implementations. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2005**, *13*, 989–992.
19. Kan, F.J.; Chen, Y.H.; Su, S.H.; Dai, L.L.; Lin, K.S.; Wang, S.Y. The fastest matrices multiplication using involutory matrix in AES MixColumns–InvMixColumns transformation. *Commun. CCISA* **2024**, *30*, 1–19.
20. Ghosal, A.K.; Sardar, A.; Chowdhury, D.R. Differential fault analysis attack-tolerant hardware implementation of AES. *J. Supercomput.* **2024**, *80*, 4648–4681.
21. Sleem, M.; Alkabani, Y.; Rashed, A.; Ibraheem, A. Low power implementation of AES mix columns/inverse mix column on FPGA. *Adv. Mater. Res.* **2013**, *677*, 311–316.
22. Ghaznavi, S.; Gebotys, C.; Elbaz, R. Efficient technique for the FPGA implementation of the AES mixcolumns transformation. In Proceedings of the 2009 International Conference on Reconfigurable Computing and FPGAs, Cancun, Mexico, 8–11 December 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 219–224.
23. Berent, A. Advanced Encryption Standard by Example. ABI Software Development. 2013. Available online: https://img2.helpnetsecurity.com/dl/articles/AESbyExample.pdf (accessed on 12 July 2025).
24. Abraham, N.E.; Thomas, T. FPGA implementation of mix and inverse mix column for AES algorithm. *Int. J. Sci. Res. Dev.* **2013**, *1*, 1981–1984.
25. Li, H.; Friggstad, Z. An efficient architecture for the AES mix columns operation. In Proceedings of the 2005 IEEE International Symposium on Circuits and Systems (ISCAS), Kobe, Japan, 23–26 May 2005; IEEE: Piscataway, NJ, USA, 2005; pp. 4637–4640.
26. Rupanagudi, S.R.; Bhat, V.G.; Padmavathi, P.; Darshan, G.; Gurikar, S.K.; Darshan, S.; Sindhu, N.; Vidya J., V. A further optimized mix column architecture design for the advanced encryption standard. In Proceedings of the 2019 11th International Conference on Knowledge and Smart Technology (KST), Phuket, Thailand, 23–26 January 2019; IEEE: Piscataway, NJ, USA, 2019; pp.181–185.
27. Madhavapandian, S.; MaruthuPandi, P. FPGA implementation of highly scalable AES algorithm using modified mix column with gate replacement technique for security application in TCP/IP. *Microprocess. Microsystems* **2020**, *73*, 102972.
28. Prayitno, R.H.; Sudiro, S.A.; Madenda, S.; Harmanto, S. Hardware implementation of Galois field multiplication for mixcolumn and inversemixcolumn process in encryption–decryption algorithms. *J. Theor. Appl. Inf. Technol.* **2022**, *100*, 5358–5367.
29. Kumar, T.M.; Reddy, K.S.; Rinaldi, S.; Parameshachari, B.D.; Arunachalam, K. A low area high speed FPGA implementation of AES architecture for cryptography application. *Electronics* **2021**, *10*, 2023.
30. Rijmen, V.; Daemen, J. The Advanced Encryption Standard Process. In *The Design of Rijndael: The Advanced Encryption Standard (AES)*; Springer: Berlin/Heidelberg, Germany, 2020.
31. Daemen, J.; Rijmen, V. AES Proposal: Rijndael (Document Version 2). AES Submission. 1999. Available online: https://www.cs.miami.edu/home/burt/learning/Csc688.012/rijndael/rijndael_doc_V2.pdf (accessed on 1 June 2025).