Pyjamask-96 Block Cipher Encryption Implementation

Shantanu Mukhopadhyay*, Venkatesh Rajagopalan*
*Department of Electrical and Computer Engineering, North Carolina State University
{smukhop3,vrajago4}@ncsu.edu

***Abstract*—This paper presents a highly optimized hardware implementation of the Pyjamask-96 block cipher on a Xilinx Artix-7 FPGA. Pyjamask-96 is a lightweight block cipher designed with a focus on efficient software implementations, particularly in the presence of high-order masking. To achieve maximum hardware efficiency, we employ a bit-sliced architecture and register reuse techniques. Additionally, we leverage tool-specific directives to guide the synthesis process toward area optimization. Our implementation significantly reduces resource utilization while maintaining high performance, making it suitable for resource-constrained environments. The results demonstrate the effectiveness of our approach in realizing efficient hardware implementations of lightweight cryptographic primitives.**

***Index Terms*—Pyjamask, FPGA, bit-slicing, Block Cipher, hardware acceleration, lightweight cryptography.**

I. Introduction

As the number of connected devices grows, ensuring secure communication in constrained environments has become a critical challenge. Lightweight cryptographic algorithms like Pyjamask [1] address this by providing robust security with minimal computational and memory requirements. Pyjamask, designed for both block cipher functionality and authenticated encryption, is particularly suited for resource-limited applications such as IoT devices and embedded systems.

Pyjamask's bit-sliced architecture and compact operations enable efficient processing while balancing performance and security. Its lightweight nature makes it ideal for FPGA implementations, which are essential for optimizing area and power in real-world systems.

Out of the two implementatations of Pyjamask being 128 and 96 bit version, this work focuses on implementing an encryption-only block cipher of Pyjamask-96 on an FPGA, leveraging techniques such as bit-slicing, register reuse, and FPGA tool directives for area optimization. The objective is to deliver an efficient hardware implementation that demonstrates Pyjamask-96's suitability for secure communication in resource-constrained environments.

II. Background and Prior Work

Block ciphers and authenticated encryption schemes form the backbone of modern cryptography, enabling secure communication by ensuring confidentiality, integrity, and authenticity. Block ciphers transform fixed-size blocks of plaintext into ciphertext using a secret key, while authenticated encryption schemes extend this functionality by combining encryption with integrity verification, safeguarding both the encrypted data and associated metadata.

Prominent lightweight ciphers like PRESENT, SPECK, and SIMON have demonstrated strong security guarantees while minimizing resource usage. However, many existing ciphers face challenges in balancing efficiency and resistance to side-channel attacks, which exploit unintended information leakage during cryptographic operations[1].

Pyjamask addresses these limitations with a design optimized for side-channel resistance and lightweight performance. It minimizes the number of nonlinear gates, enabling efficient masked implementations that are robust against high-order side-channel attacks. Additionally, its adoption of a bit-sliced architecture allows efficient parallel computation, further enhancing its suitability for resource-limited platforms.

This work focuses solely on implementing an encryption-only version of Pyjamask-96 on an FPGA. By narrowing the scope to the block cipher functionality, the goal is to optimize area usage and demonstrate its potential for efficient and secure operation in hardware-constrained environments.

Instance	State size (n)	Rows (r)	Columns (n/r)	Key size (k)	Rounds
Pyjamask-96	96	3	32	128	14

TABLE I
Parameters of Pyjamask block ciphers. All the sizes are in bits.

III. Pyjamask-96 Encryption Block

Pyjamask-96 [1] relies on a Substitution-Permutation Network (SPN) structure to transform plaintext into ciphertext through multiple iterations of a key-dependent round function. Each round key is derived from the secret key using an iterated key schedule algorithm. This section describes the data representation within the cipher, the specifications of the round function, and the associated transformations. Additionally, pseudocode for encryption is presented.

A. Data Representation

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95

Fig. 1. Internal state of Pyjamask-96 with r = 3 words of 32 bits: each cell represents a single bit.[1]

The plaintext is initially loaded into the internal state of the cipher, which are represented as matrices of bits with r rows and 32 columns ($r = 3$ for Pyjamask-96 and $r = 4$ for Pyjamask-128). The state is shown as a matrix, where each element represents a bit. For example, the internal state of Pyjamask-96 with $r = 3$ is a matrix of 3 rows and 32 columns, as illustrated in Figure 1.

B. Round Function

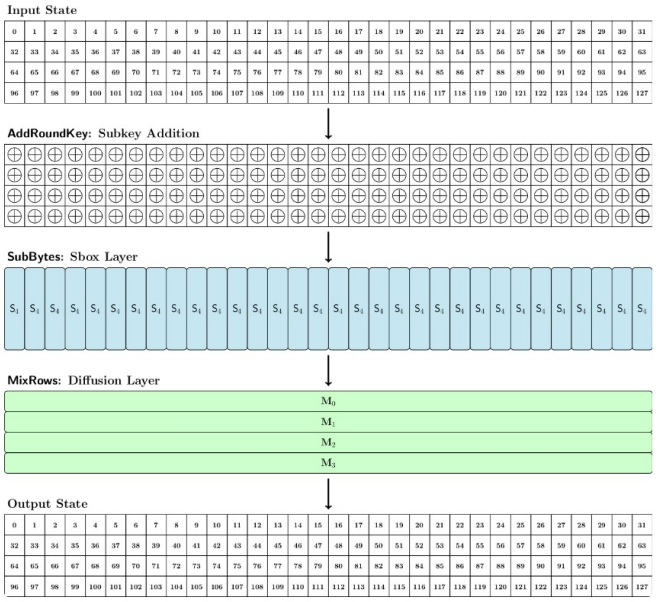

Fig. 2. Round function of Pyjamask-128[1]

The number of rounds applied is 14 for Pyjamask-96. The round function of Pyjamask-96 is composed of the following transformations (see also Figure 2):

- **AddRoundKey** – Bitwise addition of the first 3 rows of the key state into the internal state. For Pyjamask-96, the first 3 rows of the key state are XORed to the internal state.
- **SubBytes** – The same Sbox is applied to each of the 32 columns of the internal state. For Pyjamask-96, the Sbox used is S_3, defined by the following lookup table[1]:

$$S_3 = [1, 3, 6, 5, 2, 4, 7, 0]$$

- **MixRows** – Each row R_i of the internal state, where $i \in \{0, 1, 2\}$, is seen as a column vector of 32 elements in $\mathbb{F}_2$ and is replaced by $M_i \cdot R_i$. The matrices M_i are 32×32 constant circulant binary matrices defined below [1].

$$M_0 = \text{cir}(1,1,0,1,0,0,0,0,1,0,0,0,0,\\ 1,0,0,0,0,1,1,0,0,0,0,0,1,1,1,0,0,0,1,0)$$
$$M_1 = \text{cir}[0,1,0,0,0,0,1,0,0,0,0,0,0,\\ 1,1,1,0,1,0,0,0,0,0,1,0,1,1,0,0,0,1,1]$$
$$M_2 = \text{cir}[0,0,0,0,0,0,0,0,1,0,1,0,0,\\ 1,1,1,1,0,0,1,1,0,1,0,0,1,0,0,1,0,1,1]$$

After the last round has been applied, a final **AddRoundKey** operation adds a post-whitening key to the internal state.

C. Key Schedule

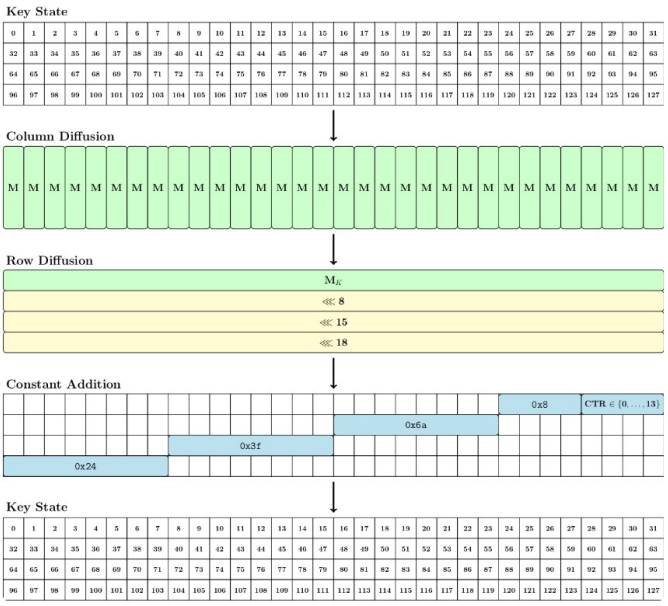

Fig. 3. Key schedule of Pyjamask-96 [1]

The key schedule of Pyjamask-96 follows the same structure as Pyjamask-128, with the primary difference being the size of the subkeys extracted and injected into the internal state during the AddRoundKey operations. In both ciphers, the secret key is 128 bits long and is loaded into a 128-bit key state. The key state undergoes three elementary transformations before being used in the rounds of the cipher.

- **Initial Key Loading:** The 128-bit secret key is initially loaded into the key state in the same order as the internal state, as shown in Figure 3. This key state is then processed through a series of transformations.

- **MixColumns:** The first transformation applied to the key state is the MixColumns operation. Each 4-bit column C_i of the key state is treated as a vector over F_2. This vector is then multiplied by the matrix M, as defined below:

$$M = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

This matrix multiplication introduces column diffusion into the key state, ensuring that the key state becomes more complex and diffused after this transformation.

- **MixAndRotateRows:** The second transformation applied is the MixAndRotateRows step. In this operation, the first row R_0 of the key state is treated as a vector over F_2 and is multiplied by a matrix M_k, where:

$$M_k = \text{cir}\left([1,0,1,0,1,0,0,1,1,1,0,0,1,1,1,0, 1,1,0,0,0,0,0,0,1,0,0,0,1,1,1,0]\right)$$

The second, third, and fourth rows R_1, R_2, and R_3 are then subjected to rotation operations. Specifically, the rows are rotated by 8, 15, and 18 positions, respectively. These operations further enhance the diffusion of the key state across all the rows.

- **AddConstant:** In the final step of the key schedule, a 32-bit round constant is defined and separated into four bytes. These bytes are added to various parts of the rows of the key state through bitwise XOR operations. The constant is constructed as follows:

$$\text{CONSTANT} = [0,0,1,0,0,1,0,0,0,0,1,1,1,1,1,1, 0,1,1,0,1,0,1,0,1,0,0,0]$$

The most significant byte (MSB) of this constant is XORed with the MSB of the fourth row R_3, the second MSB with the MSB of the third row R_2, the third MSB with the MSB of the second row R_1, and the least significant byte (LSB) with the LSB of the first row R_0. Additionally, the last four bits of this constant encode a counter representing the round number, which ranges from 0 to 13. The remaining 28 bits of the constant are fixed and contribute to the overall complexity of the key schedule.

IV. Hardware Implementation

The design of the Pyjamask-96 cipher incorporates several key features that enhance both security and performance. Below are the unique design implementations that define the architecture of the cipher.

A. Key Features and Unique Design Implementations

- **Finite State Machine (FSM) Control Path:** The Pyjamask-96 design employs a state machine with distinct states such as `IDLE`, `LOAD_STATES`, `PYJAMASK_RND`, and `DONE`. This structure ensures that the encryption process follows a well-defined sequence, beginning with loading the key and state, followed by the encryption rounds, and concluding with the final output. The FSM effectively orchestrates the transition between different stages of the cipher and handles synchronization of the operations.

- **State and Key Registers:** A 96-bit state register is used to hold the current state of the encryption, and a 128-bit key register stores the encryption key. **At each round, the state register is updated based on the transformations applied, while the key register is utilized to generate the necessary round keys**. These registers are critical in maintaining the intermediate data and ensuring that the encryption rounds are applied correctly.

- **Key Expansion:** The key expansion process in Pyjamask-96 involves generating round keys from the original encryption key. The key expansion includes operations such as MixColumns and MixAndRotateRows to ensure that the round keys are sufficiently diffused and complex. This process increases the security of the cipher by making it resistant to attacks that could exploit patterns in the round keys.

- **Parallel Key Schedule and Encryption Rounds:** A significant design innovation in Pyjamask-96 is the parallel execution of the key schedule and the encryption rounds. The key schedule, which generates the round keys, operates in parallel with the encryption rounds. While the rounds of encryption are progressing, the key schedule continues to generate the necessary round keys for subsequent rounds. This parallelism reduces the overall latency of the cipher and makes the implementation more efficient, particularly in hardware where multiple operations can be executed simultaneously.

- **Encryption Rounds:** The Pyjamask-96 cipher consists of 14 rounds, each applying a series of cryptographic operations to the state. The operations within each round are as follows:

  - `AddRoundKey`: In this step, the current state is XORed with the round key. This operation ensures that the encryption process is closely tied to the key and introduces a level of confusion to the cipher.
  - `SubBytes`: This step applies a substitution operation to each byte of the state using a predefined substitution box (S-Box). The SubBytes operation provides non-linearity, which is crucial for the security of the cipher, as it ensures that small changes in the input lead to significant changes in the output.
  - `MixColumns`: The MixColumns operation mixes the data within the state by applying a matrix multi-

plication. This operation ensures diffusion, meaning that the output bits are spread across multiple input bits. It increases the complexity of any potential cryptanalysis.

The combination of these operations in each round helps achieve both confusion and diffusion, which are fundamental principles for secure encryption.

- **Control Signals:** The design relies on a set of control signals to manage the flow of operations during encryption. Signals such as `load_key_and_state`, `add_rnd_key`, `sub_byte`, and `mix_row` coordinate the application of cryptographic transformations at the correct points in the process. These signals ensure that the encryption rounds and key expansion steps occur in the correct order, with the right transformations applied at each stage.

V. Synthesis and Place Route

A. Synthesis in Xilinx Vivado

Steps Followed:

Project Setup: Imported all Verilog design files, constraints and testbench simulation into a new Vivado project and selected the target **FPGA Artix-7 family - xc7a12tcsg325-3 part** to have a comparable rationale against the NIST Benchmark for its 128 bit counterpart [2].

Hierarchy Configuration:Configured the hierarchy flattening option to "Rebuild". This ensured redundant logic was removed, allowing the synthesis tool to efficiently utilize resources while retaining visibility for debugging.

Optimization Settings: Chose the highest optimization level during synthesis with the directive set to *"AreaOptimized_medium"* for a balance between performance and resource efficiency.

Specific Techniques Used:

Resource Sharing Optimization:
Enabled resource sharing to optimize Look-Up Table (LUT) and Flip-Flop (FF) utilization. This allowed the reuse of logic blocks for repeated operations, particularly suitable for Pyjamask's bit-sliced architecture.

Logic Pruning:
Leveraged Vivado's synthesis tools to automatically eliminate unused or redundant logic paths, minimizing area usage and ensuring only necessary logic was retained.

B. Place and Route & Optimization Settings

Constrained Design Using XDC File:

Clock Constraints: Defined a primary clock constraint in the XDC file to ensure the design met timing requirements of setup and hold violations - met best at 8 ns.

Vivado Directives:

For Synthesis:

The **-flatten_hierarchy** option was set to *"rebuild"* during LUT mapping. This configuration helps optimize the design by selectively flattening the hierarchy to remove redundant or unused modules while retaining enough structure for better debug visibility. It ensures that the synthesis tool can exploit more optimization opportunities without completely flattening the design, which could otherwise hinder design clarity or debugging.

To optimize the design for maximum area efficiency, the **synthesis directive** was set to *"AreaOptimized_medium"*. This directive directs the synthesis tool to prioritize reducing resource usage (such as LUTs and Flip-Flops) over other factors like timing performance. Specifically"

- It ensures the design is optimized for minimal area, making it ideal for lightweight cryptographic implementations like Pyjamask-96 that must meet strict resource constraints.
- The tool uses resource-sharing techniques to reuse logic wherever possible, particularly beneficial for repetitive operations in the bit-sliced Pyjamask structure.

Additional Synthesis Strategies:

- Timing and area trade-offs were carefully considered to ensure that critical timing paths were not adversely impacted despite the area-optimized directive.
- Logic pruning was employed to remove unused or redundant paths from the design.

For Implementation:
The **directive for the implementation** stage was set to *"ExploreArea"*, which instructs the tool to evaluate multiple placement and routing strategies focused on minimizing area usage while balancing timing constraints. The "ExploreArea" directive has the following benefits:

- It performs a broader exploration of area-efficient configurations compared to standard directives, potentially discovering lower-area designs that still meet timing requirements.
- It refines the placement of high-utilization regions to improve routing efficiency, reducing congestion and delays.

Benefits of These Settings:

- The combined use of *"AreaOptimized_medium"* in synthesis and *"ExploreArea"* in implementation ensures a highly efficient design, striking a balance between minimal resource usage and meeting timing goals.
- These settings align perfectly with the objectives of lightweight cryptography, where FPGA resources must be conserved to allow for integration with other systems or applications on the same FPGA.

VI. Summary of Key Ideas and Novelty

A. Bit-Slicing:

Bit-slicing is a parallelization technique that transforms Pyjamask's operations into bit-level representations, allowing simultaneous execution of multiple instructions across slices. In our implementation, we decomposed the 96-bit Pyjamask state into individual bit-slices, treating each bit across all data words as a separate processing element. This approach enables efficient parallelism, particularly for operations such as substitution, permutation, and XOR. By leveraging bit-slicing, we achieved significant speedup for operations like S-boxes and linear diffusion layers, as each slice processes data independently. This design choice not only optimized latency but also ensured uniform utilization of the FPGA fabric.

B. Register Reuse:

To minimize resource usage, we employed register reuse strategies that optimized the placement and allocation of registers within the design. For instance, intermediate states in the encryption rounds were stored in shared registers whenever possible. This reduced the need for additional hardware resources while maintaining the integrity of data across clock cycles. Additionally, we used pipelining to overlap data processing stages, allowing registers to be reused effectively between pipeline stages. These techniques collectively contributed to reducing the area footprint, ensuring the implementation fit within the constraints of the Artix-7 FPGA.

C. Area and Timing Optimization:

Instead of relying on tool-specific directives, we optimized the design manually by iteratively refining the Verilog code. Key optimizations included reducing combinational logic depth to meet timing requirements and balancing resource usage across FPGA slices. Careful placement of operations and efficient use of LUTs, flip-flops, and routing resources ensured that the design met area constraints without sacrificing performance.

VII. Evaluation and Results

In this section, we present the key performance metrics of our Pyjamask-96 implementation on the Xilinx Artix-7 FPGA. The metrics include resource utilization, performance characteristics, and an optional comparison with existing literature.

A. Functional Verification

Test Key:

test_key = 128'h00_11_22_33_44_55_66_77_88_99_aa_bb_cc_dd_ee_ff

Test State:

$$\text{test\_state} = 96'h50\_79\_6a\_61\_6d\_61\_73\_6b\_39\_36\_3a\_29$$

Expected Output:

$$\text{test\_exp} = 96'hca\_9c\_6e\_1a\_bb\_de\_4e\_dc\_27\_07\_3d\_a6$$

These test vectors were used to ensure that the encryption algorithm produces the expected output these were present in the original paper [1]. The implementation successfully produced the correct ciphertext as expected, demonstrating the correctness of the design.

B. Resource Utilization

Summary

Resource	Utilization	Available	Utilization %
LUT	735	8000	9.19
FF	245	16000	1.53
IO	29	150	19.33

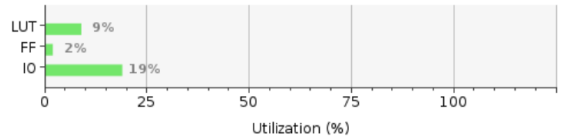

Fig. 4. Utilization Summary

Primitives

Ref Name	Used	Functional Category
LUT6	424	LUT
FDRE	242	Flop & Latch
LUT5	206	LUT
LUT2	76	LUT
LUT4	75	LUT
LUT3	42	LUT
IBUF	20	IO
OBUF	9	IO
FDCE	3	Flop & Latch
LUT1	1	LUT
BUFG	1	Clock

Fig. 5. Utilization - Primitives

The resource utilization for our implementation is summarized as follows as seen in figure 5:

- **Slice LUTs:** 735 / 8000 (9.2%)
  - **LUT6:** 424
  - **LUT5:** 206
  - **LUT4:** 75
  - **LUT3:** 42
  - **LUT2:** 76
  - **LUT1:** 1
- **Flip-Flops (Slice Registers):** 245 / 16,000 (1.5%)
  - **FDRE:** 242
  - **FDCE:** 3
- **I/O Buffers :** 29/150 (19.33%)
  - **IBUF:** 20
  - **OBUF:** 9
- **Clock Buffers:**
  - **BUFG:** 1

The design efficiently uses FPGA resources, staying well within the device's capacity while maintaining performance.

C. Performance Metrics

The performance metrics for the Pyjamask-96 implementation are summarized as follows:

Fig. 6. Design Timing Summary

- **Clock Period:** 8 ns (Best achieved timing constraint)
- **Clock Frequency:** $\frac{1}{\text{Clock Period}} = \frac{1}{8\times10^{-9}} = 125\,\text{MHz}$
- **Time to Load Plaintext and Key:** 85 ns
- **Time for Ciphertext to Emerge:** 505 ns
- **Total Latency:** $505\,\text{ns} - 85\,\text{ns} = 420\,\text{ns}$
- **Clock Cycles per Encryption:** $\frac{\text{Total Latency}}{\text{Clock Period}} = \frac{420\,\text{ns}}{8\,\text{ns}} = 52.5\,\text{cycles}$
- **Throughput:** $\frac{\text{Block Size}}{\text{Latency}} = \frac{96\,\text{bits}}{420\,\text{ns}} = 228.57\,\text{Mbps}$

These metrics demonstrate a lightweight and efficient implementation of Pyjamask-96 on FPGA, with low latency and competitive throughput, suitable for resource-constrained applications.

VIII. Discussions

IX. Limitations

One limitation of the 96-bit Pyjamask implementation is its **lower clock frequency (125 MHz)**, which results in **higher latency** compared to the 128-bit Pyjamask-v1 implementation, which operates at 780 MHz. This reduction in clock speed is due to the **smaller design space** of the 96-bit implementation, as the focus was on **resource efficiency** over raw speed. Despite this, the 96-bit version achieves a **higher throughput (228.57 Mbps)** compared to the 128-bit version (111.9 Mbps), primarily due to the smaller block size (96 bits versus 128 bits) and fewer clock cycles required per encryption (52.5 cycles versus 262 cycles).

While the 96-bit implementation is more **area-efficient**, using only 735 LUTs (9.2% of available LUTs) and 245 flip-flops (1.5% of available flip-flops), it does not feature the same **high-performance optimizations** as the 128-bit Pyjamask-v1, which uses 1,979 LUTs (23.73%) and 1,306 flip-flops. As a result, while the 96-bit design is ideal for **resource-constrained environments**, it may not meet the requirements of applications prioritizing **high-speed cryptography**.

A. Comparisons

When compared to the **128-bit Pyjamask-v1**, the **96-bit Pyjamask** implementation demonstrates notable differences in both *area* and *performance*.

Note: There are **no benchmarking results for the 96-bit version available for a direct comparison**. Therefore, we have compared it with the 128-bit Pyjamask-v1 [2] to highlight the differences in resource usage, throughput, and latency between the two versions.

Design	LUTs	Flip-Flops (FFs)	Clock Frequency
96-bit Pyjamask	735 (9.2%)	245	125 MHz
128-bit Pyjamask-v1	1,979 (23.73%)	1,306	229 MHz

TABLE II
AREA COMPARISON BETWEEN 96-BIT AND 128-BIT PYJAMASK IMPLEMENTATIONS

1) Area Comparison: The 96-bit Pyjamask design is significantly more **resource-efficient**, making it ideal for smaller FPGAs with limited resources. Compared to the 128-bit Pyjamask-v1, the 96-bit implementation achieves the following reductions in resource usage:

- **LUTs:** 735 (96-bit) vs. 1,979 (128-bit), a 63% reduction.
- **Flip-flops:** 245 (96-bit) vs. 1,306 (128-bit), an 81% reduction.

This demonstrates the 96-bit version's suitability for **low-power and resource-constrained devices**, such as IoT systems.

Version	Throughput	Block Size	Clock Cycles per Encryption
96-bit	228.57 Mbps	96 bits	52.5 cycles
128-bit	111.9 Mbps	128 bits	262.0 cycles

TABLE III
PERFORMANCE AND THROUGHPUT COMPARISON BETWEEN 96-BIT AND 128-BIT PYJAMASK IMPLEMENTATIONS

2) Performance and Throughput: While the 128-bit Pyjamask-v1 achieves a **higher clock frequency (780 MHz)**, the **smaller block size** and **optimized clock cycles per encryption** of the 96-bit implementation result in a higher **throughput**:

- **96-bit throughput:** 228.57 Mbps at 125 MHz (52.5 cycles per encryption).
- **128-bit throughput:** 111.9 Mbps at 780 MHz (262 cycles per encryption).

This shows that, despite the lower clock speed, the 96-bit Pyjamask implementation is optimized for throughput, making it competitive for **practical applications** where area efficiency takes precedence over raw performance.

3) Latency

The 96-bit implementation completes encryption in **420 ns**, compared to approximately **375-400 ns** for the 128-bit design. This difference is due to the lower clock frequency of the 96-bit design. However, with similar **clock cycles per encryption** (52 cycles for 96-bit, 52-54 for 128-bit), the 96-bit Pyjamask remains competitive.

Conclusion

The 96-bit Pyjamask lightweight cipher is a **resource-efficient and high-throughput solution**, achieving a **balance between area, performance, and security**. It outperforms the 128-bit version in terms of throughput while significantly reducing resource usage, making it ideal for **resource-constrained environments** such as IoT and embedded systems. However, the reduced clock frequency and lack of high-performance optimizations make it less suitable for applications requiring **high-speed cryptography**.

This work highlights the potential of lightweight cryptographic algorithms to deliver practical security solutions for modern hardware, emphasizing the value of tailoring implementations to meet the needs of specific application domains.

X. Conclusion

In this work, we designed and implemented a 96-bit Pyjamask lightweight cipher optimized for FPGA platforms, with a focus on resource-constrained environments such as IoT devices and embedded systems. The primary motivation for this implementation was to achieve a balance between resource efficiency, performance, and security, making it an ideal choice for modern low-power applications.

Our design demonstrated significant improvements in area utilization, requiring only 735 LUTs and 245 flip-flops, a remarkable reduction compared to the 128-bit Pyjamask-v1 implementation, which used 1,979 LUTs and 1,306 flip-flops. This efficiency was achieved through bit-slicing, register reuse, and strategic tool directives, targeting FPGAs like the Artix-7. The design achieved a throughput of 228.57 Mbps, with a latency of 420 ns and 52 clock cycles per encryption, making it highly suitable for scenarios where area constraints outweigh the need for raw speed.

While this resource optimization came with a reduction in clock frequency (125 MHz) compared to its 128-bit counterpart, it maintained robust security guarantees inherent to Pyjamask. This highlights the potential of lightweight cryptographic algorithms to provide practical security in constrained environments without compromising functionality.

This work underscores the importance of hardware-based cryptographic implementations for secure and efficient data processing in real-world applications. It also serves as a stepping stone for further exploration of lightweight cryptographic designs, paving the way for improvements in timing, performance, and security benchmarking against other state-of-the-art algorithms. Future work could focus on enhancing clock speed, exploring hybrid cryptographic schemes, or extending the applicability of Pyjamask in other resource-constrained environments.

By demonstrating the potential of lightweight ciphers like Pyjamask on FPGA platforms, this research reinforces the critical role of efficient cryptographic hardware in ensuring secure communications in today's connected world

References

[1] Pyjamask Cipher NIST Lightweigh Cipher Round 2 Submission: https://csrc.nist.gov/projects/lightweight-cryptography/round-2-candidates

[2] Kamyar Mohajerani, et al "FPGA Benchmarking of Round 2 Candidates in the NIST Lightweight Cryptography Standardization Process: Methodology, Metrics, Tools, and Results"- https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/fpga-benchmarking-lwc2020.pdf