A Symmetric Key Cryptosystem and Hardware Encryption Module using Non-linear Reversible Cellular Automata

Kamalika Bhattacharjee[1] and Shlok Shelat[2]

[1]Department of Computer Science and Engineering, National Institute of Technology, Tiruchirappalli, 620015, Tamilnadu, India.
[2]Department of Computer Science, Ahmedabad University, Ahmedabad, 380009, Gujrat, India.

Contributing authors: kamalika@nitt.edu; shlok.s1@ahduni.edu.in;

Abstract

This paper presents an encryption and decryption method using Cipher Block Chaining (CBC) combined with Large Cycle Reversible Non-Uniform Cellular Automata. The proposed approach utilizes an 8-size rule vector that ensures both substantial cycle length and randomness, which are critical for effective cryptographic operations. The 8-bit cellular automaton (CA) is scaled to any length cellular automaton (CA) by repeating specific rules, minimizing the need for additional gates and maintaining cost efficiency. This CA serves as an encryption key, transforming the plaintext into an encrypted message devoid of discernible patterns. We propose two designs with block sizes of 64 and 128 bits. The encrypted output undergoes rigorous randomness testing using the Dieharder, NIST, and TestU01 test suites. An analysis of known and chosen plaintext attacks shows that these are not feasible. In addition, a visual test was conducted to confirm the randomness of the ciphertexts used. Finally, we propose a prototype hardware model for encryption using a **64**-bit key that is robust and scalable.

Keywords: Non-linear Non-uniform Elementary Cellular Automata, Reversibility, Cryptographic Module, Encryption, Randomness Testbeds, Hardware

1 Introduction

Cryptography plays a crucial role in securing digital communications by protecting sensitive information from unauthorized access and tampering. As technology advances, the demand for more efficient and secure cryptographic systems is growing. Cellular automata (CA), particularly nonlinear reversible CAs, have emerged as promising candidates for cryptographic applications owing to their inherent properties of long cycle lengths, chaotic behavior, and ability to generate complex pseudo-random sequences, which are essential for creating robust cryptographic keys [1–7, 10, 23].

Cellular automata are discrete, abstract computational systems that have demonstrated significant potential in various fields, including in cryptography. Their ability to generate complex pseudo-random sequences makes them suitable for cryptographic operations, such as encryption and random number generation. Nonlinear reversible cellular automata (CA), in particular, offer enhanced security features owing to their complex evolution and reversibility, which are essential for creating robust cryptographic keys.

Cellular automata (CA) are classical nature-inspired computational models. They were introduced by Ulam and vonNeumann in the 1940s to model self-reproducing systems, and have since been used to mimic complex physical and biological processes. By using CAs for encryption, we leverage these well-known emergent behaviors to produce high-quality pseudo-randomness in a hardware-friendly way.

In this study, we used non-uniform elementary reversible CAs with large cycles to design a robust symmetric key cryptosystem. In the case of a reversible cellular automata, its global evolution rule is bijective, so the previous configuration can be uniquely determined from the present one. We used the cipher-chain blocking (CBC) mode to implement this cryptosystem. The basic idea of this cryptosystem is as follows: The initial configuration of the n-cell CA is a plaintext block of n bits. For encryption, the CA evolves for t-time steps in the forward direction, and the t^{th} configuration becomes the encrypted ciphertext. Again, for decryption, the CA evolves t times in the backward direction to obtain the plaintext. We use a function called *prevConf* to obtain the previous configuration of an input configuration based on a given CA rule vector for decryption.

The implementation of such a cryptographic module by cipher-chain blocking (CBC) mode, considering a block size of 128 bits, was previously demonstrated in Ref. [11], where a 128-cell CA serves as the encryption key to transform any plaintext into ciphertext. Although this approach offers a secure and robust encryption mechanism, it is hindered by concerns regarding efficiency in terms of speed and scalability. Furthermore, the selection of rules as the key for the cryptosystem was heuristic. This work is an extension of that in Ref. [11], we address these limitations by reducing the encryption time and introducing a scalable key-construction method. This enhanced approach allows the encryption process to adapt to various plaintext block sizes based on requirements. For instance, an appropriate key size can be chosen, such as a 32-bit plaintext block, and a 256-bit plaintext block can be encrypted effectively by creating respective encryption keys of the same size by iteratively applying specific rules from an 8-bit module, thereby enhancing both efficiency and adaptability.

However, our main motivation for this study stems from the need to develop a hardware cryptographic module that leverages the advantages of nonlinear elementary CAs. Traditional cryptographic systems often rely on complex algorithms that require substantial computational resources, making them less suitable for hardware implementations. By utilizing reversible nonlinear non-uniform elementary CA, we aim to design a cryptographic module that not only provides high security but also maintains low hardware complexity.

2 Related Work

Cellular automata (CA) have been extensively investigated as cryptographic primitives because of their straightforward rule-based evolution and inherent parallelism. Early research by Wolfram demonstrated that one-dimensional CA (Rule 30) can generate complex pseudorandom sequences suitable for encryption [1]. These CA-based generators can function as stream ciphers, integrating the plaintext with CA states at each step [5, 7]. Traditional ciphers, such as AES or RSA, typically rely on algebraic complexity and sequential operations; however, evolving cryptanalytic techniques and hardware requirements necessitate the development of alternative models. CA schemes can be efficiently implemented in hardware (e.g., FPGAs) and offer dynamic, nonlinear transformations. Notably, reversible CA rules (i.e., bijective global evolution) enable ciphertext blocks to be uniquely inverted back to plaintext, rendering them appealing for block cipher design [4, 6]. These attributes, including long cycle lengths, high nonlinearity, and hardware-friendly architecture, have inspired numerous CA-based encryption schemes.

Numerous CA-based ciphers have been proposed. For example, Das and Roy-Chowdhury's CAR30 is a scalable CA stream cipher that combines rule 30 with a maximum-length linear CA. CAR30 uses a 128-bit key (with a 120-bit IV) and offers a high throughput comparable to Grain or Trivium, owing to its flexible, hardware-efficient design [23]. On the block cipher side, Jeyaprakash *et al.* introduced KAMAR, a lightweight Feistel cipher for sensor networks using a one-dimensional reversible CA [20]. KAMAR uses radius-1 CA rules (including RCA rules 30 and 45) as nonlinear S-boxes on 128-bit blocks, achieving a high clock frequency and low area on the FPGA. Similarly, Ostapov *et al.* developed two CA-based symmetric ciphers: a 3D-CA-augmented AES-like block cipher and a CA-based stream cipher [19]. Their block cipher embeds 3D CA permutations into an SP network around an AES S-box, while their stream cipher uses CA-driven permutation functions. Beyond pure cipher design, CA has been integrated into broader security systems. For instance, Corona-Bermúdez *et al.* proposed a CA-based framework that first uses a zero-knowledge protocol on a NP-hard secret and then encrypts data with a two-dimensional CA [8]. These studies illustrate the versatility of CA in cryptography, from purely algebraic stream ciphers to hybrid block ciphers and CA-driven authentication schemes.

Many recent studies have focused specifically on reversible and hybrid CA for hardware encryption. Kotulski *et al.* present a multi-layer symmetric cipher built entirely from reversible CAs on FPGA [18]. Their design processes 128-bit plaintext blocks through five cascaded reversible-CA layers with a 256-bit secret key governing dynamic rule configurations. This fully CA implementation achieves high diffusion and is optimized for parallel hardware. Other researchers have theoretically analyzed the cryptographic properties of CA. For instance, Mariot and Leporati studied the algebraic nonlinearity and resiliency of CA global rules, showing that linear CA correspond to cyclic codes and that permutive local rules yield 1-resilient global rules [10]. In practice, however, only a handful of elementary CA rules are reversible in isolation; therefore, schemes often use higher-order or nonuniform CA constructions. For example, Janz *et al.* and others have constructed reversible CA rules by combining two elementary rules (one for cell=0 and one for cell=1) to ensure bijectivity [12]. These specialized reversible CA classes (e.g., based on Rule 137/118) can achieve very long cycles, even on small blocks. Overall, related work spans linear and additive CAs to complex nonlinear and hybrid CA systems, with many designs demonstrating high hardware efficiency and resistance to classical attacks.

3 Synthesis Algorithm and Essential Criteria

To build a secure CA-based cipher, we first need cryptographic keys whose CA rules are reversible and have very large cycle lengths (ideally close to 2^n). In this section, we describe the stochastic synthesis algorithm (from Ref. [13]) were used to generate candidate non-uniform CAs with these properties. We also defined the key criteria—reversibility, long cycles, and good randomness—that each candidate must satisfy.

This study uses a one-dimensional 3-neighbourhood 2-state CA, named elementary CA (ECA), over a null boundary condition with lattice size $n = \mathbb{Z}/n\mathbb{Z}$ where the cells are numbered from 0 to $n-1$. The CA is non-uniform, that is, each cell may use a different rule to update its state, and at least one of the rules used in the CA is nonlinear. A *rule vector* $\mathcal{R} = \langle R_0, \cdots R_{n-1} \rangle$ represents the nonuniform ECA, where each R_i is the rule used in cell i, $0 \leq i \leq n-1$. Hereafter, if not explicitly mentioned, a CA shall be interpreted as only a non-linear, non-uniform elementary CA.

We consider only the reversible CAs, that is, starting from any configuration, the evolution traverses back to that configuration. In Ref. [13], a stochastic synthesis algorithm is given that uses Table 1 to generate reversible non-uniform ECAs such that the cycle length of the CA is large. In our work, by '*large cycle length*', we mean that the CA has at least one cycle of length at least 2^{n-1}. For ease of understanding, we reproduce the algorithm (Algorithm 1) from Ref. [13].

As this is a stochastic algorithm, there can be cases of false positives, indicating that the CAs generated do not produce long cycle lengths. However, to be used as a cryptographic primitive, along with reversibility, a substantially large cycle length as close to 2^n as possible, and a good randomness property, are essential. To obtain a primary assessment of the randomness quality of the CA, we perform a *visual test* by plotting the space-time diagrams of the generated CAs to determine whether small changes in the initial state can lead to vastly different outcomes over time. This is indicated by the appearance of zero-to-less patterns in the space-time diagrams representing the evolution of the CA. These visual representations highlight the intricate

Algorithm 1 Stochastic Algorithm for Generating Large Cycle Reversible CA

```
Step 1: Partition each class $m$ of Table 1 into different sets $\mathtt{C}^{\mathtt{m}}_{\mathtt{q}}$ as $\mathtt{C}^{\mathtt{m}}_{\mathtt{q}} = \{R | R$ belongs to class $m$ and $\mathtt{P}(R) = q$ using Table 4$\}$ where $q \in \{0.25, 0.5, 1\}, m \in \{I, II, III, IV, V, VI\}$.

Step 2: Select $R_0$ uniformly at random from $\{5, 10, 9, 6\}$ (see Table 2). From $R_0$, the class $m$ of $R_1$ is identified using Table 2 and

Step 3: Repeat Steps 4, 5, and 6 for $i$ (1 to $n-2$) to synthesize a reversible CA.

Step 4: Generate a random number ($X$) between 0 and 1.

Step 5: If $X < 0.35$
            if $\mathtt{C}^{\mathtt{m}}_{0.5} \neq \emptyset$,
                select $R_i$ uniformly at random from $\mathtt{C}^{\mathtt{m}}_{0.5}$
            otherwise
                select $R_i$ uniformly at random from $\mathtt{C}^{\mathtt{m}}_{1}$
        Else If $X < 0.75$
            select $R_i$ uniformly at random from $\mathtt{C}^{\mathtt{m}}_{1}$
        Otherwise
            If $\mathtt{C}^{\mathtt{m}}_{0.25} \neq \emptyset$,
                select $R_i$ uniformly at random from $\mathtt{C}^{\mathtt{m}}_{0.25}$
            otherwise
                select $R_i$ uniformly at random from $\mathtt{C}^{\mathtt{m}}_{1}$

Step 6: Determine class $m$ of $R_{i+1}$ based on $R_i$, (using Table 1).

Step 7: Select $R_{n-1}$ uniformly at random from the set {5, 20, 65, 80} (using Table 3).
```

interactions within the system. Space-time diagrams of cellular automata exhibiting an extended cycle length while demonstrating randomness provide valuable insights into the dynamic patterns and complexities inherent in their evolution. Therefore, our primary selection criterion is to be utilized as a cryptographic key, and the synthesized CAs by Algorithm 1 must possess two essential qualities:

- **Substantial Cycle Length:** The cycle length should exceed 2^{n-1}. This implies that the automaton goes through a significantly high number of unique states before eventually repeating, which is indicative of a highly complex and expansive evolution of the state.
- **Good Randomness Quality:** The CA should exhibit good randomness quality to ensure unpredictability. Only those CAs that do not display discernible patterns are considered random and suitable for cryptographic applications.

Algorithm 1 (Stochastic CA-Key Synthesis) generates an n-cell reversible, nonlinear CA whose global state-cycle is astronomically long. Its core steps are:

Rule Classification: All 256 elementary 3-neighbour binary rules are grouped into four classes based on their dependency on neighbors (fully, partially, weakly, or not at all). Fully and partially dependent rules form the "preferred" pool for maximal nonlinearity and mixing in the model.

Biased Random Assembly: Starting at the left end, pick each cell's rule at random—with a strong bias toward the fully/partially dependent classes, a smaller chance for weakly dependent, and zero chance for independent. At the first and

Table 1: Class relationship of R_i and R_{i+1}

Class of R_i	R_i	Class of R_{i+1}
I	51, 204, 60, 195	I
	85, 90, 165, 170	II
	102, 105, 150, 153	III
	53, 58, 83, 92, 163, 172, 197, 202	IV
	54, 57, 99, 108, 147, 156, 198, 201	V
	86, 89, 101, 106, 149, 154, 166, 169	VI
II	15, 30, 45, 60, 75, 90, 105, 120, 135, 150, 165, 180, 195, 210, 225, 240	I
III	51, 204, 15, 240	I
	85, 105, 150, 170	II
	90, 102, 153, 165	III
	23, 43, 77, 113, 142, 178, 212, 232	IV
	27, 39, 78, 114, 141, 177, 216, 228	V
	86, 89, 101, 106, 149, 154, 166, 169	VI
IV	60, 195	I
	90, 165	IV
	105, 150	V
V	51, 204	I
	85, 170	II
	102, 153	III
	86, 89, 90, 101, 105, 106, 149, 150, 154, 165, 166, 169	VI
VI	15, 240	I
	105, 150	IV
	90, 165	V

Table 2: First Rule

Rules for $\mathcal{R}_0$	Class of $\mathcal{R}_1$
3, 12	I
5, 10	II
6, 9	III

Table 3: Last Rule

Rule class for $\mathcal{R}_{n-1}$	Rule set for $\mathcal{R}_{n-1}$
I	17, 20, 65, 68
II	5, 20, 65, 80
III	5, 17, 68, 80
IV	20, 65
V	17, 68
VI	5, 80

last cells (and each intermediate step), small compatibility tables are enforced so that the global CA map remains bijective.

Cycle-Length Verification: Once the length-n rule vector is built, the CA is evolved from a fixed seed, and the number of steps it takes to return is counted. If the cycle length falls below the cryptographically significant threshold (on the order of 2^{n-1}), the vector is discarded and the sampling is repeated.

Because each accepted vector is both bijective and packed with high-dependency rules, the resulting CA combines provable reversibility, strong nonlinearity, and an

Table 4: Categories of R_i with $P(R_i)$

Category	R_i	$P(R_i)$
Completely Dependent	90, 165, 150, 105	1
Partially Dependent	30, 45, 75, 120, 135, 180, 210, 225, 86, 89, 101, 106, 149, 154, 166, 169	0.5
Weakly Dependent	53, 58, 83, 92, 163, 172, 197, 202, 54, 57, 99, 108, 147, 156, 198, 201, 23, 43, 77, 113, 142, 178, 212, 232, 27, 39, 78, 114, 141, 177, 216, 228	0.25
Independent	51, 204, 85, 170, 102, 153, 60, 195, 15, 240	0

exponentially large period, making it an excellent key-schedule or pseudo-random generator for secure block ciphers.

The following section details the process of selecting a suitable CA for cryptographic purposes, including the generation of initial rule vectors, scaling procedures, and randomness assessment.

4 Selection of Proper CAs and the Scale-up

In cryptographic applications, the selection of good keys plays a pivotal role in ensuring the security and efficiency of the cryptographic module. This study aims to generate an efficient, fast, secure, and scalable cryptographic module. Hence, we start our search with good rule vectors of cell length 8.

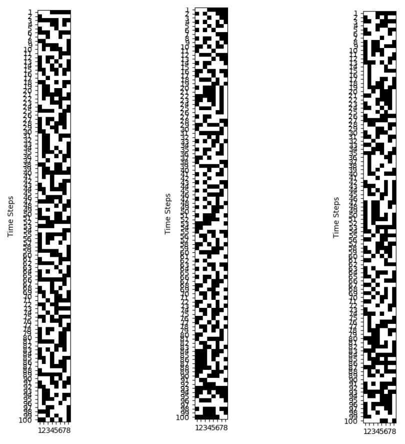
4.1 Generating 8-bit Rule Vectors

We initiated our study with 8-bit CAs to establish a foundation for scalability. This approach allows us to extend these 8-bit CAs to larger lattice sizes, such as 16-bit, 32-bit, and beyond, which will subsequently be employed as keys in the encryption process. Additionally, the choice of an 8-bit CA is advantageous because of the relative ease of computing various parameters, such as the cycle length.

We employed Algorithm 1 to generate rule vectors of size 8. Each synthesized CA was subjected to a verification process to ensure that its cycle length exceeded 128. For enhanced security, we selected CAs with cycle lengths greater than 200. This threshold provides a robust margin and ensures that the generated CAs can produce sequences that are sufficiently complex and unpredictable. Over a run of 5000 iterations, several CAs were identified that exhibited large cycle lengths. Table 5 lists some sample results of good CAs with corresponding cycle lengths. These CAs are then tested over a space-time diagram, and those with the fewest patterns are chosen for the next stage. Fig. 1 shows three space-time diagrams of some of the CAs chosen for scale-up.

Table 5: Sample 8-bit CAs satisfying our criteria

8-bit Rule Vector	Cycle Length
⟨5, 90, 89, 165, 105, 90, 105, 5⟩	239
⟨9, 150, 75, 147, 105, 150, 165, 65⟩	206
⟨5, 150, 169, 90, 105, 165, 90, 5⟩	204
⟨10, 105, 90, 45, 165, 150, 65, 5⟩	255
⟨5, 105, 165, 135, 154, 90, 90, 5⟩	222
⟨5, 120, 106, 105, 165, 150, 150, 80⟩	235
⟨5, 150, 90, 150, 165, 90, 90, 5⟩	217
⟨5, 150, 154, 165, 90, 90, 150, 80⟩	219



(a) (b) (c)

Fig. 1: Sample space-time diagrams for 8-bit CAs. (a) ⟨5, 90, 89, 165, 105, 90, 105, 5⟩, (b) ⟨9, 150, 75, 147, 105, 150, 165, 65⟩ and (c) ⟨5, 150, 169, 90, 105, 165, 90, 5⟩

4.2 Scaling up to 16-bit Rule Vectors

An 8-bit CA is quite compact and, although useful for initial testing and verification, is insufficient for practical encryption applications. To employ CA for encryption, the length of the CA must be increased to provide a more complex and secure key. We first scaled the 8-bit CA to a 16-bit CA by repeating the middle 4 rules. However, an arbitrary repetition is not guaranteed to follow Algorithm 1. To generate any $4{+}4k, ; k \in \mathbb{N}$, size rule vector $\mathcal{R}_{4+4k}$, we need to ensure that the rules in the chosen 8-bit rule vector $\mathcal{R}8 = \langle R_0, R_1, R_2, R_3, R_4, R_5, R_6, R_7\rangle$ are selected carefully. In particular, R_2 must be included in the next class of R_5 so that it can appear at cell 6 in the scaled-up CA $\mathcal{R}4 + 4k = \langle R_0, R_1, (R_2, R_3, R_4, R_5)^k, R_6, R_7\rangle$. Simultaneously, it preserves the class relationship of Table 1, that is, the next class of R_2 at the 7^{th} cell contains R_3 and so on. If all rules are found in their designated classes throughout, the rule vector adheres to Algorithm 1, because the initial rules are chosen following this algorithm and no new rules are introduced. If any rule fails to belong to the specified class at any point in time, the rule vector is discarded. For instance, if the rule vector for an 8-bit CA is $\langle 5, 90, 89, 165, 105, 90, 105, 5\rangle$, we create a 16-bit CA by tripling the middle four rules, resulting in $\langle 5, 90, 89, 165, 105, 90, 89, 165, 105, 90, 89, 165, 105, 90, 105, 5\rangle$. The repetition of the central 4 bits enables the preservation of reversibility and follows Algorithm 1, allowing the transformation of an 8-cell reversible CA into a 16-cell reversible CA.

We conduct experiments for all 8-bit CAs that meet the criteria in Section 4.1. Each CAs is scaled to a 16-bit CA by repeating its middle four rules, as described above. We then verify whether each scaled rule vector still satisfies all conditions of Algorithm 1 and has a cycle length greater than 2^{15}. Only the 16-bit rule vectors that met both criteria were selected. Based on our experimental results, we assert that all scaled versions of these qualifying 16-bit rule vectors will also follow Algorithm 1. This approach also facilitates scalability, as the same hardware/software module can be replicated to develop an encryption module capable of processing larger key sizes according to the application requirements. Some of the selected 16-cell CAs with their cycle lengths are reported in Table 6.

Table 6: Some selected 16-bit rule vectors

16-bit Rule Vector	**Cycle Length**
⟨5, 90, 89, 165, 105, 90, 89, 165, 105, 90, 89, 165, 105, 90, 105, 5⟩	29536
⟨5, 120, 106, 105, 165, 150, 106, 105, 165, 150, 106, 105, 165, 150, 150, 80⟩	60237
⟨5, 150, 90, 150, 165, 90, 90, 150, 165, 90, 90, 150, 165, 90, 90, 5⟩	65535
⟨5, 150, 154, 165, 90, 90, 154, 165, 90, 90, 154, 165, 90, 90, 150, 80⟩	54379
⟨10, 75, 90, 150, 165, 150, 90, 150, 165, 150, 90, 150, 165, 150, 101, 80⟩	59483
⟨6, 105, 105, 89, 150, 90, 105, 89, 150, 90, 105, 89, 150, 90, 165, 20⟩	37619
⟨5, 105, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 149, 80⟩	35447
⟨9, 105, 45, 105, 90, 150, 45, 105, 90, 150, 45, 105, 90, 150, 90, 65⟩	64030
⟨9, 86, 105, 165, 165, 90, 105, 165, 165, 90, 105, 165, 165, 90, 165, 20⟩	56628
⟨6, 178, 165, 105, 89, 105, 165, 105, 89, 105, 165, 105, 89, 105, 165, 20⟩	45256
⟨6, 169, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 150, 20⟩	54655
⟨10, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 80⟩	33731
⟨5, 150, 169, 90, 105, 165, 169, 90, 105, 165, 169, 90, 105, 165, 90, 5⟩	61162
⟨6, 169, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 150, 20⟩	54655
⟨6, 178, 165, 105, 89, 105, 165, 105, 89, 105, 165, 105, 89, 105, 165, 20⟩	45256

4.3 Scaling up to 32-bit and 64-bit Rule Vectors

We now scale up the chosen 16-bit rule vectors to 32-bit rule vectors such that they follow Algorithm 1. According to the algorithm, there is a high likelihood that the maximum cycle length exceeds 2^{31}. However,

Then, to further assess the randomness quality of these CAs, we test their performance over the Dieharder test suite [14] by generating a 1GB output `.bin` file produced as a concatenation of output configurations[1]. The CAs that pass at least 60 out of the 114 tests are scaled up to 64 bits. These 64-bit rule vectors were examined for cycle length by running the experiment for 2 h (Table 7 shows some results). The CAs with substantial cycle lengths are then tested over Dieharder (see Table 8 for some sample results), and those that pass at least 75 out of the 114 tests are considered for the encryption module.

We have identified a few potential candidates The results are displayed in Table 9. The cycle lengths for these rules are extremely large. For example, the actual cycle length observed for the rule vector γ over 32-bits with an initial configuration of all zeros except the middle bit set to 1 is $2,51,46,83,067 > 2^{31}$, which we obtain by evolving the CA for 6 h. Similarly, for 64-cell size version of γ, even after running for 10 h with $2,69,39,88,610$ evolutions, the initial configuration of all zeros except the middle bit set to 1 is not returned. This indicates that the cycle length is indeed very

[1]For testing by any test suite, the seed is considered as 19650218.

Table 7: Number of unique configurations generated for scaled-up version of some rule vectors in 32-bit and 64-bit in 1hr and 2 hrs, respectively

16-bit Rule Vector	32-bit version	64-bit version
[5, 105, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 149, 80]	Returned to initial configuration after 265117393 evolutions	Did not return to the initial configuration after 2 h. Total evolutions in 2 hours: 590022679
[10, 75, 90, 150, 165, 150, 90, 150, 165, 150, 90, 150, 165, 150, 101, 80]	Did not return to the initial configuration after 1 h. Total evolutions in 1 hour: 391678348	Did not return to the initial configuration after 2 hours. Total evolutions in 2 hours: 585246246
[6, 105, 105, 89, 150, 90, 105, 89, 150, 90, 105, 89, 150, 90, 165, 20]	Did not return to the initial configuration after 1 hour. Total evolutions in 1 hour: 392088396	Did not return to the initial configuration after 2 hours. Total evolutions in 2 hours: 581214085
[9, 105, 45, 105, 90, 150, 45, 105, 90, 150, 45, 105, 90, 150, 90, 65]	Returned to initial configuration after 60 evolutions	Returned to initial configuration after 124 evolutions
[9, 86, 105, 165, 165, 90, 105, 165, 165, 90, 105, 165, 165, 90, 165, 20]	Did not return to the initial configuration after 1 hour. Total evolutions in 1 hour: 390848819	Did not return to the initial configuration after 2 hours. Total evolutions in 2 hours: 576526465
[6, 178, 165, 105, 89, 105, 165, 105, 89, 105, 165, 105, 89, 105, 165, 20]	Returned to initial configuration after 333665725 evolutions	Did not return to the initial configuration after 2 hours. Total evolutions in 2 hours: 587427949
[6, 169, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 150, 20]	Did not return to the initial configuration after 1 hour. Total evolutions in 1 hour: 385004622	Did not return to the initial configuration after 2 hours. Total evolutions in 2 hours: 594027072
[10, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 80]	Returned to initial configuration after 1210860 evolutions	Did not return to the initial configuration after 2 hours. Total evolutions in 2 hours: 590639981

large, exceeding 2^{n-1} for this n. The rule vectors in Table 9 can also be scaled up to 128-bits. For example, when the rule vector γ is scaled up to 128-bits and tested over Dieharder, its result is: out of 114 tests, 93 passed, 3 weak, and 18 failed. This shows that our scheme for generating a rule vector from scaling is indeed able to find very good keys.

5 The Proposed Symmetric-Key Cryptosystem

Our proposed symmetric key encryption system employs the Cipher Block Chaining (CBC) mode of operation. In CBC mode, each plaintext block is encrypted by XORing it with the previous ciphertext block, with an initialization vector (IV) used for the first block. This method strengthens the security through a chaining mechanism. The IV is a block-sized sequence of randomly generated bits created by a good pseudo-random number generator (PRNG). Here, we use the Mersenne Twister (`MT19937`

Table 8: Dieharder test result of sample CAs over 32 and 64 cell lengths

16-bit Rule Vectors	**32-bit version**	**64-bit version**
[6, 178, 165, 105, 89, 105, 165, 105, 89, 105, 165, 105, 89, 105, 165, 20]	Passes: 64 Fails: 41 Weak: 9	Passes: 79 Fails: 32 Weak: 3
[10, 75, 90, 150, 165, 150, 90, 150, 165, 150, 90, 150, 165, 150, 101, 80]	Passes: 69 Fails: 38 Weak: 7	Passes: 79 Fails: 32 Weak: 3
[9, 105, 45, 105, 90, 150, 45, 105, 90, 150, 45, 105, 90, 150, 90, 65]	Passes: 66 Fails: 40 Weak: 8	Passes: 81 Fails: 30 Weak: 3
[5, 105, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 149, 80]	Passes: 64 Fails: 40 Weak: 10	Passes: 78 Fails: 31 Weak: 5
[9, 86, 105, 165, 165, 90, 105, 165, 165, 90, 105, 165, 165, 90, 165, 20]	Passes: 67 Fails: 40 Weak: 7	Passes: 78 Fails: 31 Weak: 5
[6, 105, 105, 89, 150, 90, 105, 89, 150, 90, 105, 89, 150, 90, 165, 20]	Passes: 66 Fails: 41 Weak: 7	Passes: 76 Fails: 32 Weak: 6
[10, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 80]	Passes: 63 Fails: 42 Weak: 9	Passes: 78 Fails: 31 Weak: 5
[6, 169, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 150, 20]	Passes: 68 Fails: 39 Weak: 7	Passes: 78 Fails: 31 Weak: 5

Table 9: Candidate rule vectors for encryption denoted by different variables

Variable	**Rule Vector**
α	[10, 75, 90, 150, 165, 150, 90, 150, 165, 150, 90, 150, 165, 150, 101, 80]
β	[6, 105, 105, 89, 150, 90, 105, 89, 150, 90, 105, 89, 150, 90, 165, 20]
γ	[5, 105, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 149, 80]
δ	[9, 105, 45, 105, 90, 150, 45, 105, 90, 150, 45, 105, 90, 150, 90, 65]
ϵ	[9, 86, 105, 165, 165, 90, 105, 165, 165, 90, 105, 165, 165, 90, 165, 20]
ζ	[6, 178, 165, 105, 89, 105, 165, 105, 89, 105, 165, 105, 89, 105, 165, 20]
θ	[6, 169, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 150, 20]
ι	[10, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 90, 169, 165, 105, 80]

[2]), a widely recognized PRNG known for its speed, statistical properties, and use in various programming languages and applications. Other reliable PRNGs can also be used instead of `MT19937`.

The plaintext is split into equal-sized blocks, each n bits in size, where n can be set according to the application requirements. To facilitate parallel processing within the CBC mode, we use *BLOCKS_PER_GROUP*, which represents the number of plaintext blocks allocated to each thread. These blocks are grouped into clusters

[2]Here, we use `MT19937` [15] implementation in Python as `std::mt19937_64` for testing.

of *BLOCKS_PER_GROUP*, with each group operating in CBC mode independently, using its IV, and running concurrently with separate threads.

The key of the cryptosystem is a large, non-linear, reversible CA generated through Algorithm 1 using the mechanism proposed in Section 4. Encryption occurs in two layers: the first layer uses a nonlinear large-cycle CA, followed by a linear CA layer. To begin, the block is advanced k time-steps through the key using the *nextConf* method. Then, it undergoes n steps through a linear uniform CA with an ECA 153. For decryption, the steps are reversed: the block first advances n steps through the linear layer, followed by k steps through the key's reverse. This step is equivalent to moving k steps backward through the original key, which is accomplished using the *prevConf* algorithm outlined in Algorithm 3. Because the same key is used for both encryption and decryption, this approach qualifies as a symmetric-key cryptosystem. Figure 2 illustrates the overall structure of the proposed cryptosystem. Next, we outline the key components of our system: *nextConf, prevConf, blockEncrypt, blockDecrypt, threadEncrypt, threadDecrypt, encrypt* and *decrypt* as discussed in Ref. [11]

5.1 nextConf

The primary function of the *nextConf* method is to generate the next configuration of the given block. This function accepts two parameters: the current configuration (represented as a binary sequence) and the rule vector. The rule vector is generated using the method described in Section 4. The function iterates over all cells (bit positions) in the current configuration, and for each position, retrieves the states of the neighboring left and right cells to form the neighborhood. The next state for the current position is then determined based on the neighborhood and the corresponding rule for that position. The process of the nextConf function is outlined in Algorithm 2.

Algorithm 2 next configuration given a rule vector

```
procedure NEXTCONF(conf, ruleVector)
    for $pos \leftarrow 0$ to $len(conf) - 1$ do
        $neighbourhood \leftarrow conf[pos - 1 : pos + 1]$
        $nextConf[pos] \leftarrow ruleVector[pos] >> neighbourhood \;\&\; 1$
    end for
    return $nextConf$
end procedure
```

5.2 prevConf

The *prevConf* function calculates and returns the previous configuration based on the given current configuration and a set of rules provided by the rule vector. The function operates in a series of steps, involving both left-to-right and right-to-left passes over the bit positions (or cells). The steps are shown in Algorithm 3.

During the left-to-right pass, the function processes each bit position, determines its value, and identifies any dependencies. A set of candidate RMTs is created by

Algorithm 3 previous configuration given a rule vector

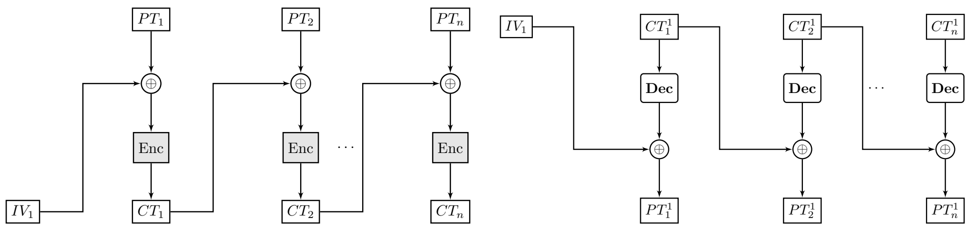
```
procedure PREVCONF(conf, ruleVector)
    for $pos \leftarrow 0 \quad to \quad BLOCK\_SIZE - 1$ do                ▷ Forward pass
        $candidateRMTs \leftarrow empty\ list$
        for $RMT \leftarrow 0 \quad to \quad 7$ do
            if $RMT$ obeys discovered dependence rules and has resolved bits then
                candidateRMTs.append(RMT)
            end if
            Resolve bits by finding common bits in all rmts from candidateRMTs
            Capture dependencies among RMTs from candidateRMTs
        end for
    end for
    for $pos \leftarrow BLOCK\_SIZE, \ldots, 0$ do                ▷ Backward pass
        if pos is dependent on prevPos in $[pos - 1, pos - 2]$ then
            resolve prevPos based on the dependence rule goto prevPos
        end if
    end for
    return $prevConf$
end procedure
```

checking all possible values and analyzing the relationships between bits, updating the dependency information as needed. Based on these resolved dependencies and candidate bit patterns, the function continues to resolve the dependencies and set the bits accordingly.

In the subsequent right-to-left pass, the function uses the resolved dependencies to fill in any missing bits in the frame. It iterates through the bits and identifies those that are unset and dependent on others. By following the dependency chain, it sets the missing bits based on the information gathered during the left-to-right pass phase. Once all computations are completed, the function returns the previous configuration as a bitstring. The whole process takes $\mathcal{O}(n)$ time to find the previous configuration of the current n-bit configuration.

5.3 blockEncrypt

This function is responsible for encrypting a given plaintext block and returning the corresponding cipher-text block. The n-bit block first undergoes a transformation through the nonlinear nonuniform CA layer given by the key. The block is evolved k steps using the *nextConf* function using the key. The resulting output block is then passed through a linear layer that features a uniform CA with rule 153, where the block advances for n time steps. The ciphertext produced by this process is then returned. This multilayer encryption method adds security through the repeated evolution of the CA, with the two layers using distinct sets of ECA rules to further obfuscate the plaintext. Figure 2a illustrates



(a) Encrypt block in CBC mode (b) Decrypt block in CBC mode

Fig. 2: Encryption and Decryption considering one thread

5.4 threadEncrypt

To enable multithreading, this module operates as a single-thread instance that manages the encryption of a predefined number of blocks grouped together. Each thread is responsible for processing $BLOCKS\_PER\_GROUP$ blocks of the main plaintext. The initialization vector (IV) is initialized for the current group and remains global for all blocks within that group. The *blockEncrypt* function is applied to all the blocks in the group. For each block, the ciphertext is generated by XORing the result of the current plaintext block with that of the previous ciphertext block.

5.5 encrypt

This outlines the entire encryption process. The plaintext blocks are divided into groups for concurrent encryption, with each group being processed by a separate thread. Each thread is responsible for handling a group and calling the threadEncrypt() function. The final ciphertext is returned after all the groups are processed. The overall encryption process is presented in Algorithm 4.

Algorithm 4 encryption function

```
procedure ENCRYPT(PTblocks)
    $groups \leftarrow len(PTblocks)\ /\ BLOCKS\_PER\_GROUP$        ▷ groups :
num_of_threads
    $CTblocks \leftarrow$ empty list
    $CipherTextBlocks \leftarrow len(PTblocks) + groups$
    for $group \leftarrow 0 \quad to \quad groups$ do
        new thread is generated to encrypt the current group
        $CTblock \leftarrow$ each individual block in a group is encrypted by using nextConf
        $CTblocks.append(CTblock)$
    end forreturn CTblocks
end procedure
```

5.6 blockDecrypt

As illustrated in Figure 2b, the decryption process retraces the encryption path to recover the plaintext. It begins by passing through the linear layer with uniform ECA rule 153 for n time intervals. Next, the block runs k time steps through the non-linear layer, using the *prevConf* function over the same key by advancing backward.

5.7 threadDecrypt

This function operates similarly to the *blockEncrypt* function. It handles the decryption of a given set of blocks, with each block being decrypted using the *blockDecrypt* function.

5.8 decrypt

This is the overall decryption module for the ciphertext. The function creates a predefined number of threads, with each thread responsible for decrypting a group of data. Each group consists of several blocks of ciphertext. Once all the threads complete their execution, the resulting plaintext is returned. The steps are shown in Algorithm 5.

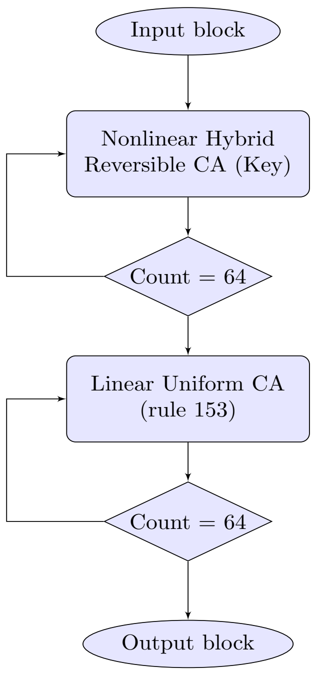
Algorithm 5 decryption function

```
procedure DECRYPT(CTblocks)
    $groups \leftarrow len(PTblocks)\ /\ BLOCKS\_PER\_GROUP$                ▷ groups :
num_of_threads
    $PTblocks \leftarrow$ empty list
    $CipherTextBlocks \leftarrow len(PTblocks) + groups$
    for $group \leftarrow 0$ to $groups$ do
        new thread is generated to decrypt the current group
        $CTblock \leftarrow$ each individual block in a group is decrypted by using prevConf
        $PTblocks.append(PTblock)$
    end forreturn PTblocks
end procedure
```

5.9 Example: 64-bit CA-based Cryptographic Module

Let us consider the block size as 64-bit. The encryption module (see Fig. 2a) accepts plaintext as input and changes it to encrypted text by running it through two encryption layers. The encryption process includes performing an XOR operation on the plaintext with an IV generated by a PRNG. After the XOR operation, the outcome undergoes two layers of encryption. The initial step consists of evolving the xored output $n = 64$ times with our key, which is the 64-bit CA (say, by the rule vector γ). Next, the transformed outcome undergoes a second round of encryption, in which it undergoes $k = 64$ evolutions according to ECA 153. The final output is the encrypted message (Fig. 3a).

The decryption module (see Fig. 2b) accepts the ciphertext as input and changes it to plaintext by running it through the two decryption layers. The decryption process involves retracting the large cycle reversible CA to obtain the plaintext. The ciphertext is first evolved forward for 64 time steps using a uniform CA with ECA Rule 153. The result of the linear layer is evolved backwards for 64 time steps using a nonlinear reversible CA key with the help of *prevConf*. To recover the original text, the output of the linear layer is XORed with the IV.


Input block
Nonlinear Hybrid Reversible CA (Key)
Count = 64
Linear Uniform CA (rule 153)
Count = 64
Output block
(a) Flow diagram of encryption process

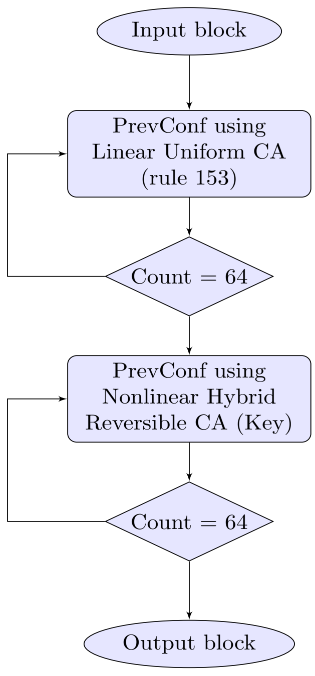

Input block
PrevConf using Linear Uniform CA (rule 153)
Count = 64
PrevConf using Nonlinear Hybrid Reversible CA (Key)
Count = 64
Output block
(b) Flow diagram of decryption process

Fig. 3: Encryption and Decryption taking 64-bit Block

6 Results and Analysis

Now we analyse the strength of our scheme as well as test the selected CAs for randomness quality to find the best one for cryptography.

6.1 Design considerations

Our cryptosystem is designed using two parameters n, k: n can be chosen as 64, 128, etc., and accordingly k can also be set. A Larger value of k indicates more time to perform the encryption and subsequently the decryption. An optimized value of k is identical to n. The cryptosystem employs Cipher Block Chaining (CBC) mode for encryption, chosen over Substitution Permutation Network (SPN) mode to defend against linear and differential cryptanalytic attacks that could exploit static S-boxes.

In the linear layer, ECA 153 is used, which is an equal-length cycle CA. The cycle length of an n-cell uniform CA following ECA 153 is given by $l = 2^{\lfloor \log n \rfloor + 1}, n \geq 2$ [5]. For example, in a 64-cell CA, the cycle length is 128. Therefore, in both encryption and decryption, advancing 64 time steps ensures that the configuration returns to the original input after 128 time steps. This is why n time steps are used in the linear layer during encryption and decryption.

In the linear layer, we use ECA 153, which is known to produce an equal-length cycle. In fact, an n-cell uniform CA under rule153 has cycle length $2^{\lfloor \log n \rfloor + 1}$. For example, a 64-cell CA returns to its initial state after 128 iterations. Hence, in encryption/decryption, we advance exactly 64 steps in this layer, ensuring that we traverse a complete cycle. We place this linear (rule-153) layer after the nonlinear CA layer so that even if the nonlinear layer by itself has a shorter cycle, the combined system still enjoys the full equal-length cycle property. This adds an additional security layer, as all configurations are guaranteed to be part of the fixed 128-step cycle.

The non-linear layer, containing the large-cycle reversible CA as the key, advances the input by k steps during encryption and reverses k steps using *prevConf* during decryption. This balance between the number of time steps and the security-performance trade-off is crucial for optimizing both the security and computational efficiency of the cryptosystem.

The cryptosystem uses a single n-cell CA as the symmetric key for all the blocks across the threads. A known plaintext attack aimed at exposing the CA's cyclic structure would require the encryption of $2^n - 1$ n-bit blocks, which is computationally infeasible for large n such as $n = 128$.

The strength of the cryptosystem depends heavily on the key, which is a single n-cell, non-uniform, non-linear, reversible CA. To ensure robustness and security, we must choose a *good* CA as the key. The Following subsections test the candidature of the keys selected in Section 4.3, considering the block size as 64-bits. Using a 64-bit block instead of the 128-bit version in Ref. [11] is a good trade-off with speed without compromising security (see Section 6.3). However, a user may choose to use a 128-bit block size for better security.

Table 10: Results of statistical tests for 64-bit block size along with the total gate count

Rule Vector	**Dieharder (P/F/W)**	**NIST (P/F)**	**SmallCrush (P/F)**	**Total Gates**
α	114/0/0	All/None	All/None	136
β	109/1/4	All/None	All/None	162
γ	**112/1/1**	**All/None**	**All/None**	**114**
δ	110/0/4	All/None	All/None	158
ϵ	113/0/1	All except 1/1	All/None	116
ζ	105/2/7	All/None	All/None	155
θ	108/0/6	All/1	All/None	123
ι	108/0/6	All/None	All/None	119

6.2 Results of Statistical Tests

To verify the randomness quality of the encrypted message, we test the 64-bit encrypted module by choosing different keys over different testbeds, such as Dieharder [14], NIST [16], and SmallCrush of TestU01 [17]. For Dieharder (resp. NIST), the pattern $0^{32}10^{31}$ is repeatedly used as plaintext to generate the 1GB (resp. 131MB) `.bin` file. For the SmallCrush test, the seed is initially set to 0 and incremented by 1 with each call to the encryption module. CAs that demonstrate strong performance in all three testbeds are considered suitable for use as encryption keys. Table 10 shows the results of the rule vectors of Table 9. Here, we observe that all CAs pass a minimum of 105 tests, fail a maximum of two tests, and show weakness in at most seven tests in Dieharder. For SmallCrush, all the selected candidates passed all tests, and at NIST, at most one test failed. Some CAs, such as α, pass all tests in each testbed, indicating the efficacy of our scheme.

6.3 Resistance against Different Attacks

We now test the encryption scheme against theoretical attacks. We show the situation when the block size is at least 64 bits. For a higher block size, the situation is similar.

6.3.1 Resistance against Brute Force Attack

The high entropy and effective diffusion of the initialization vector (IV) in CA, combined with its reversibility, make it a strong defence against statistical or analytical attacks. This indicates that the most effective attack technique might be a brute-force attack. Two brute-force attack techniques can be employed. First, the attacker may opt to rigorously construct the 64-bit key. However, to do this, (s)he has to verify all possible 64 cell keys following Algorithm 1. This is impractical because it requires at least $\Omega(2^{64})$ time for a key of length 64. Similarly, following Table 1, if one tries to build a reversible key of 64 bits, (s) will fall into a similar scenario. The other approach might be to infer the cyclic structure of the encryption key using a transition diagram. However, this approach is also infeasible, since the chosen keys have an extremely large cycle length that requires an attacker $O(2^{63})$ to find the 64 cell key. Therefore, a brute-force attack is not computationally feasible.

6.3.2 Resistance against Known Plaintext Attack

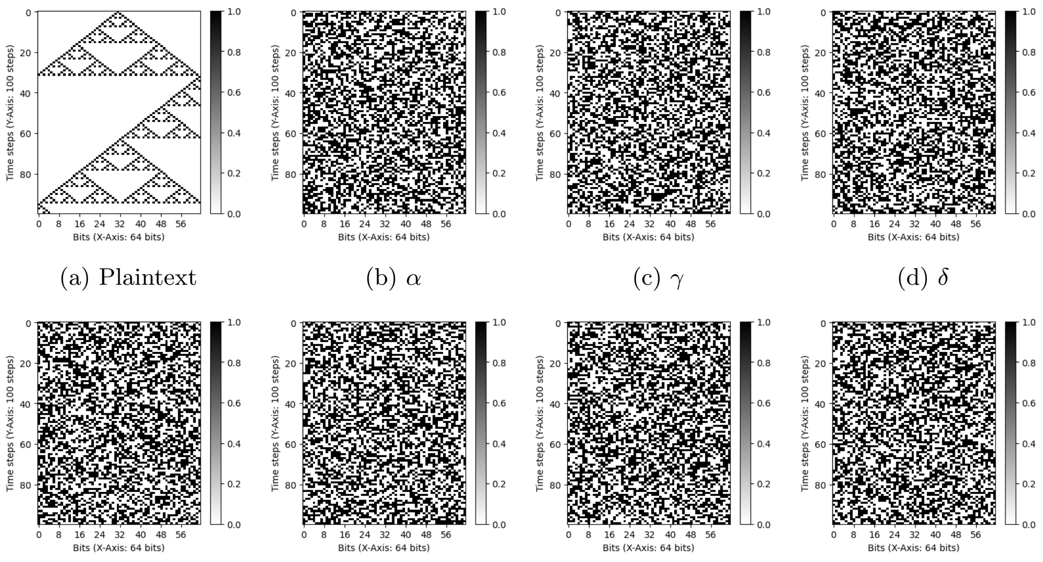
The high degree of nonlinearity and extremely large cycle length of the proposed cryptosystem guarantee that there is no obvious relationship between the plaintext and ciphertext that can be exploited. However, if the chosen plaintext block and the corresponding ciphertext block are within a tiny cycle of the CA (key), the attacker can study the short cycle and learn something about the key. However, this is an unlikely scenario because encryption works in two layers; thus, the ciphertext is not directly generated from the key. Therefore, even if it falls into an apparent small cycle, it does not reveal any information about the key because the attacker does not know which CA is being used as the key. Such a plaintext block must be identified through a brute-force search of 2^{64} configurations, which makes the attack computationally infeasible. Consequently, the proposed cryptosystem is not vulnerable to chosen or known plaintext attacks.

6.3.3 Resistance against Cycle Attack

In a cycle attack, the attacker searches for a fixed-length or small-length cycle in the CA used as key. Such a discovery can damage the cryptosystem and leak information about the key. However, because the keys are chosen to have very large ($O(2^{63})$) cycle lengths, the attacker cannot find such cycles in polynomial time. Furthermore, the use of ECA 153 as the linear layer provides an extra layer of security because of its equal-length cycle generation property. Therefore, the cryptosystem is highly resistant to cycle attacks.

6.4 Visual Test

Finally, we conduct a visual test akin to a space-time diagram to assess any discernible patterns in the encrypted texts. For this analysis, we used the space-time diagram of an ECA 90 (64-bit CA run over 100 time-steps) with pronounced fractal characteristics as the plaintext (see Fig. 4a). Each row of this space-time diagram represents the plaintext, whereas the corresponding row in another diagram illustrates the ciphertext generated by that particular key. Figures 4a and 4b to 4h, reveal that the initially



(a) Plaintext (b) α (c) γ (d) δ

(e) ϵ (f) ι (g) θ (h) ζ

Fig. 4: Ciphertext for 7 keys generated from Table 9 with plaintext of Fig. 4a

predictable patterns in the plaintext space-time diagram are thoroughly randomised after encryption. This also demonstrates that all keys effectively produce a similar randomization effect, demonstrating the significant potential of these keys.

As shown in Figure 4, the ciphertexts appear highly disordered, and the visual inspection is inherently informal. In future work, we plan to apply quantitative chaos metrics. For example, one can compute the *Shannon entropy* of the ciphertext blocks as a statistical measure of randomness. More sophisticatedly, one can measure the sensitivity to initial conditions using the CA *Lyapunov exponent*, defined via a Boolean "damage-spreading" process. Bagnoli *etal.* showed how to compute a maximal Lyapunov exponent for 1D CAs by tracking how a single-bit perturbation evolves. Such measures would automatically gauge the CA's unpredictability, complementing the Dieharder/NIST tests.

However, among these CAs, our next objective is to identify the best key that requires the fewest gates during the hardware implementation process.

6.5 Gate Minimisation for Hardware Implementation

Next, we aim to select only one rule vector as our key, which requires the fewest gates for hardware implementation. This approach aims to optimize the efficiency and effectiveness of hardware implementation by balancing performance with resource utilization, ultimately enhancing the overall system design. Table 10 reports the gate requirements of our candidate CAs with $n = 64$. From this table, we can see that the rule vector γ has the fewest total gates (114), a very large cycle length, and performs very well in the Dieharder and NIST tests, making it the optimal choice for hardware implementation.

Table 11: Results of SAC test

Metric Used	**Metric Value**
Expected Mean	32
Observed Mean	31.89
Variance	19.4979
Std. Deviation	4.4156
Coeff. of Variance	0.139861896

6.6 Strict avalanche criterion test (SAC)

To test the avalanche effect of using our chosen key with γ, we test it over the strict avalanche criterion (SAC). To perform this test, we again consider 64-bit block size. So, 1000 random 64-bit plaintexts are generated using `MT19937` (`std::mt19937_64`). For each plaintext, the following process is applied: for a plaintext, one bit at a time is flipped to create another plaintext; these two plaintexts are then encrypted using our key, and the bit difference between them is calculated. The average of all possible bit differences over the 1000 runs is taken as the final score. As summarized in Table 11, the average SAC score is approximately 32, indicating that the cryptosystem with our chosen key meets the SAC test criteria.

7 Comparative Study

This section presents a comprehensive evaluation of our proposed cellular automata-based block cipher against established symmetric encryption algorithms. The cipher employs reversible cellular automata (RCA) transformations in a novel 2-layer architecture with configurable evolution steps, designed to achieve optimal security-efficiency trade-offs for resource-constrained environments. Our evaluation encompasses both 64-bit and 128-bit variants, systematically comparing cryptographic properties, hardware implementation metrics, and overall performance against contemporary standards including NIST Lightweight Cryptography candidates.

The cellular automata approach leverages the inherent parallelism and local neighborhood interactions to create strong diffusion properties while maintaining computational simplicity. This design philosophy enables efficient hardware implementation with superior resource utilization compared to traditional block ciphers, making it particularly suitable for IoT and embedded security applications.

Table 12 presents the fundamental cryptographic characteristics of our proposed cipher alongside established algorithms.

Table 12: Cryptographic Design Parameters and Security Properties

Cipher	Type	Key (bits)	Block (bits)	Rounds	Avalanche
RCA-BC (64)	Block	64	64	64	49.98 %
RCA-BC (128)	Block	128	128	128	50.00 %
CAR30 [23]	Stream	128	–	–	Good
CAvium [24]	Stream	80	–	–	Good
AES-128 [21]	Block	128	128	10	50 %
PRESENT [27]	Block	80/128	64	31	Good
ASCON-128 [22]	Block/AEAD	128	128	12	50 %
GIFT-COFB [25]	Block/AEAD	128	64	40	Good
Grain-128AEAD [26]	Stream	128	–	–	Good

Table 13 summarizes the hardware implementation results, demonstrating the efficiency advantages of the cellular automata-based approach.

Table 13: FPGA Implementation Performance Metrics

Cipher	FPGA	LUTs/Slices	Freq. [MHz]	Throughput [Mbps]	Eff.
RCA-BC (64)	Virtex-7	238	250	1090	4.6
RCA-BC (128)	Virtex-7	469	250	2180	4.6
AES-128 [21]	Virtex-E	3760	200	1280	0.34
PRESENT [28]	Spartan-III	238 slices	215	411	1.7/slice
ASCON-128 [22]	Virtex-7	2978	250	1090	0.37
GIFT-COFB [25]	Virtex-7	6311	952	615	0.10
Grain-128AEAD [26]	FPGA	Variable	200	1250	9.34

Note: Performance data for comparative algorithms are sourced from their respective publications; unreported metrics are indicated by blank entries.

7.1 Security Properties Analysis

Our cellular automata-based cipher demonstrates excellent avalanche characteristics, achieving near-optimal diffusion rates of 49.98% and 50.00% for the 64-bit and 128-bit variants respectively. These results are comparable to established standards like AES-128 and ASCON-128, confirming that the cellular automata transformations provide the necessary confusion and diffusion properties essential for cryptographic security.

The 2-layer architecture with k=n evolution steps ensures that each bit of the plaintext influences approximately half of the ciphertext bits, satisfying the strict avalanche criterion. This performance validates our design approach where local cellular automata rules, when applied systematically across the cipher structure, produce global cryptographic strength comparable to traditional algorithms.

7.2 Hardware Implementation Efficiency

The cellular automata-based design demonstrates exceptional hardware efficiency on FPGA platforms. The 64-bit variant achieves 1,090 Mbps throughput using only 238 LUTs, resulting in a resource efficiency of 4.6 Mbps/LUT—substantially outperforming conventional block ciphers. This efficiency stems from the natural mapping of cellular automata operations to FPGA lookup tables, where simple Boolean functions can be implemented directly in hardware.

Compared to AES-128, our approach provides approximately 13.5× better resource efficiency while maintaining comparable security properties. Against NIST Lightweight Cryptography finalists, the cipher shows approximately 12.4× better efficiency than ASCON-128 and nearly 46× improvement over GIFT-COFB. The consistent efficiency across both 64-bit and 128-bit variants demonstrates the scalability of the cellular automata approach.

7.3 Cellular Automata Architecture Benefits

The reversible cellular automata framework offers several architectural advantages that translate directly to implementation benefits. The local neighborhood interactions inherent in cellular automata create parallel processing opportunities that FPGA architectures can exploit efficiently. Unlike traditional ciphers that require complex S-boxes and permutation networks, our approach uses simple, uniform transformation rules that reduce hardware complexity.

The 2-layer processing structure with configurable evolution steps provides flexibility in security-performance trade-offs while maintaining the core cellular automata properties. This design philosophy enables efficient scaling from 64-bit to 128-bit variants with linear resource growth, demonstrating the inherent scalability of the approach.

7.4 Implementation Scalability and Optimization

The cellular automata-based cipher exhibits excellent scalability characteristics. Transitioning from 64-bit to 128-bit implementations nearly doubles throughput while maintaining constant efficiency metrics, indicating optimal resource utilization scaling. This linear scalability is particularly valuable for applications requiring different security levels within the same hardware platform.

The uniform nature of cellular automata transformations simplifies both encryption and decryption operations, reducing overall implementation complexity compared to algorithms requiring distinct forward and inverse operations. This symmetry in the transformation process contributes to the observed hardware efficiency while maintaining cryptographic security.

7.5 Performance Limitations and Future Directions

While the cellular automata-based approach demonstrates superior hardware efficiency, several areas require further investigation. Power consumption analysis remains ongoing, with preliminary estimates suggesting higher dynamic power usage compared to ultra-lightweight stream ciphers. Side-channel attack resistance requires comprehensive evaluation, particularly given the increasing emphasis on such vulnerabilities in modern cryptographic standards.

Future work includes ASIC implementation analysis to evaluate energy efficiency in dedicated hardware, development of authenticated encryption modes to compete with AEAD-capable algorithms, and comprehensive cryptanalysis against advanced attack methodologies. These investigations will further establish the cellular automata approach as a viable alternative for resource-constrained cryptographic applications.

7.6 Conclusion

This comparative analysis establishes the cellular automata-based block cipher as a high-performance solution for resource-constrained environments. The combination of strong security properties, exceptional hardware efficiency, and scalable architecture positions this approach as a compelling alternative to established cryptographic standards. The significant efficiency advantage over NIST Lightweight Cryptography standards, coupled with comparable security characteristics, validates the cellular automata paradigm for practical cryptographic applications.

The results demonstrate that cellular automata-based cryptography can achieve the stringent requirements of modern security applications while providing significant implementation advantages. This work contributes to the growing field of alternative cryptographic approaches, offering a mathematically sound foundation for future lightweight cryptographic developments.

8 A Basic Hardware Encryption Model

One of the advantages of ECA is its ease of hardware implementation, with low hardware complexity and interconnection cost. In this section, we describe a prototype of the cryptographic hardware module using the rule vector $\langle$5, 105, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 105, 90, 90, 90, 149, 80⟩ as the key over the encryption process described in Fig. 3. The same can be used to scale up to 128-bits block size while increasing the time steps to 128. We implement this multilayer encryption process using combinational logic components such as XOR gates, multiplexers (MUXes), comparators, and sequential circuit components such as flip-flops and counters.

8.1 Modules

As we want our model to be scalable, we divide the entire encryption process into individual modules and submodules.

8.1.1 XOR Module

This module performs bitwise XOR between two 64-bit inputs: the plaintext and an initialisation vector. The output of this module serves as the input for the first-layer encryption. As `MT19937` does not have hardware, we manually used the IV generated by the code (`std::mt19937_64`) as an input to the XOR module.

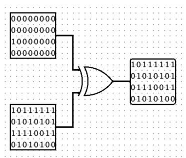


Fig. 5: XOR Module

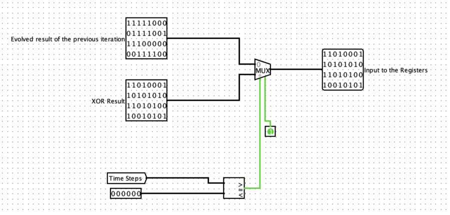


Fig. 6: MUX Module

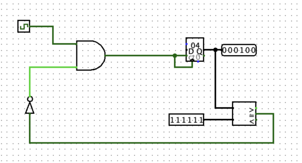


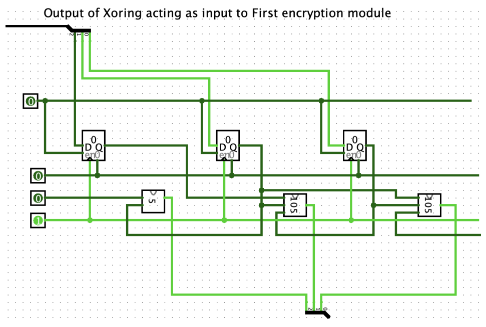
Fig. 7: Counter and Comparator Module

8.1.2 First Layer Encryption Module

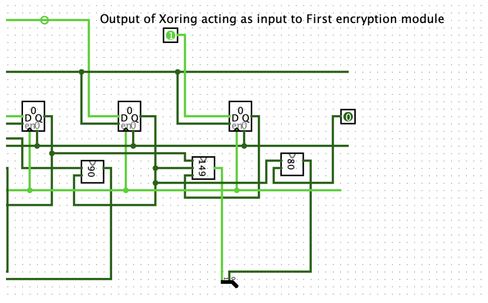
This module takes input from the XOR module and evolves it for 64 time-steps using our 64-bit key. It contains some sub-modules for implementing the key with the scope of scale-up, a counter, a comparator, and a MUX. The present state of each cell is stored in a D flip-flop for an update.

Each rule is implemented using combinatorial logic for the key. We divide the key into three unit sub-modules: the first one representing the first two rules $5, 105$ (see Fig. 8a), and the second representing the last two rules $149, 80$ (Fig. 8b) and the last one is the middle four significant rules $105, 90, 90, 90$ (Fig. 8c). To achieve the 64-bit rule vector used as our key, after the first module, the middle rules' block is duplicated 15 times, followed by the second module of the last rules. This approach provides the potential for scalability and allows the optimized use of our developed hardware.

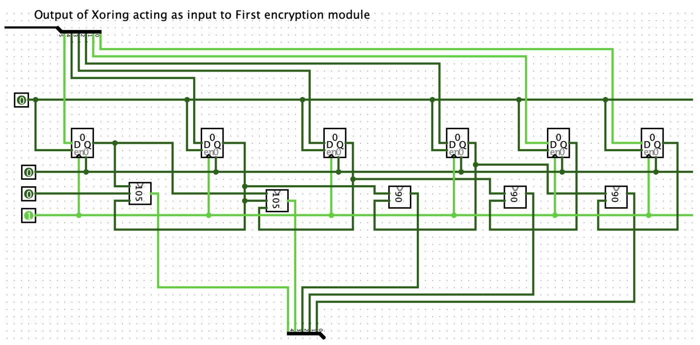
The selection signal from the counter and comparator modulo (Fig. 7) to the MUX (Fig. 6) determines whether the MUX forwards the XOR result (when the counter is at 0) or the evolved result from the previous iteration. The enable signal of the MUX is maintained high to allow operation at each clock cycle. The counter module increments with each clock cycle and tracks the number of evolution cycles. When the counter reaches 64, the comparator outputs a signal to the MUX to halt the evolution.



(a) Module for (R_0, R_1): $(5, 105)$



(b) Module for (R_{n-2}, R_{n-1}): $(149, 80)$



(c) Module of middle 4 rules to replicate: $(105, 90, 90, 90)$

Fig. 8: The three modules of rule vector γ used for scale-up to 64-bit

This ensures that the plaintext undergoes exactly 64 iterations of evolution in every encryption layer.

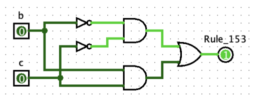
8.1.3 Second Layer Encryption Module

Following the first layer, the evolved result is passed to the second-layer encryption module. Here, the cellular automaton is evolved again for 64 iterations using a new rule vector comprising entirely of ECA 153 (Fig. 9).

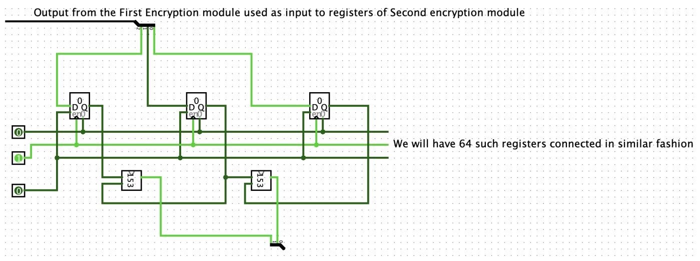
8.1.4 Interconnections of the Modules

The XOR module is connected to the first input of the MUX module, whereas the second input of the MUX module receives the evolved result from the previous iteration. The output of the MUX, a 64-bit value, is then fed into the 64 registers of the first-layer encryption module. The operation of these registers is governed by the output of an AND gate within the counter and comparator module, ensuring that the registers function for precisely 64 time steps.

In the counter and comparator module associated with the first-layer encryption, the comparator output serves as one input to an AND gate, and the second input is the clock signal. The output of this AND gate is subsequently routed to the AND gate within the counter and comparator module of the second encryption layer. The other



(a) Rule 153


Output from the First Encryption module used as input to registers of Second encryption module
We will have 64 such registers connected in similar fashion
153
153

(b) Rule 153 block

Fig. 9: Second layer linear uniform CA block

input to this gate is the inverted output from the comparator in first-layer encryption. This arrangement guarantees that the second-layer encryption commences only after the first-layer encryption is completed, following the 64 iterations in the first layer. The final evolved output from the first encryption layer is passed as the input to the second encryption module.

In the second layer of encryption, the MUX module receives two inputs: the evolved result from the previous iteration and the final evolved output from the first encryption module. The MUX output is then provided to the 64 registers of the second encryption layer. The enable signal for the registers in this second layer is driven by the AND gate in the counter and comparator module of the second layer. After 64 iterations, the final encrypted output is obtained from the registers of the second-layer encryption module.

8.2 Performance Analysis

To implement the hardware model, tt Logisimwas first used for prototyping, and then `Vivado` was used to develop the module in Verilog HDL. For simulation and verification, `Vivado`'s built-in simulator was used to test the functionality of the `Verilog` code. The design was tested for both `64-bit` and `128-bit` implementations. All tests and analyses, including CPU time measurement, power utilization assessment, and device utilization evaluation, were performed in `Vivado`, to optimize and validate the module's performance on the target platform. Tables 14, 15 and 16 report the performance summary of this basic hardware model for 64-bit blocks. Tables 17, 18, and 19 report the performance summary of this basic hardware model for 128-bit blocks.

Here, *elapsed time* is the total duration from the start to the end of a program, including both CPU processing and any waiting times (e.g., I/O operations or resource access). *Power utilization* is the amount of power consumed by a circuit during its operation. where *dynamic power* is the power consumed during active switching between logic states, which occurs owing to signal transitions within the device, and *static power* refers to the power consumed even when the device is in an idle state, primarily owing to leakage currents in the transistors. Table 14 and 17 display the power utilisation for our two modules, and Table 15 and 18 display the on-chip power utilisation. *Slice logic* utilisation refers to the number of logic blocks used in the circuit.

Table 14: Summary of Time, Power and Thermal Metrics for 64-bit blocks

CPU Time	0.07 seconds
Elapsed Time	0.13 seconds
Total On-Chip Power (W)	70.610 (Junction temp exceeded!)
Design Power Budget (W)	Unspecified*
Power Budget Margin (W)	NA
Dynamic (W)	69.375
Device Static (W)	1.235
Effective TJA (C/W)	1.5
Max Ambient (C)	0.0
Junction Temperature (C)	125.0
Confidence Level	Low
Setting File	—
Simulation Activity File	—
Design Nets Matched	NA

Table 15: On-Chip Power Utilisation for Circuit Components for 64-bit block

On-Chip	Power (W)	Used	Available	Utilization (%)
Slice Logic	1.600	439	—	—
LUT as Logic	1.516	238	134600	0.18
Register	0.080	144	269200	0.05
BUFG	0.005	1	32	3.13
Others	0.000	2	—	—
Signals	6.673	498	—	—
I/O	61.102	196	500	39.20
Static Power	1.235			
Total	70.610			

Table 16: Slice Logic Utilization for 64-bit block

Site Type	Used	Fixed	Prohibited	Available	Util (%)
Slice LUTs	238	0	0	134600	0.18
LUT as Logic	238	0	0	134600	0.18
LUT as Memory	0	0	0	46200	0.00
Slice Registers	144	0	0	269200	0.05
Reg as Flip Flop	144	0	0	269200	0.05
Reg as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Such logic blocks include Look-Up Tables (LUTs), flip-flops (FFs), and multiplexers (MUXs) used in our design. Tables 16 and 19 display the device utilization for our 64-bit and 128-bit models, respectively.

A comparison of the performance of both 64-bit and 128-bit versions is shown in Tables 20 and 21). The 128-bit rule vector cryptosystem γ shows exceptional results in all Dieharder, NIST, and SmallCrush tests with minimal weak behavior. However, as expected, it consumes more power and resources and is slower than the 64-bit version.

9 Conclusion

In this study, we designed a generic symmetric-key cryptosystem using a nonlinear, nonuniform elementary reversible large-cycle cellular automaton (CA). To select an appropriate key, we analyzed the effectiveness of expanding various 8-bit CA rule vectors to create 64-bit or 128-bit CAs by repeatedly using the middle four rules for encryption, focusing on randomness tests and gate minimization. A detailed analysis

Table 17: Summary of Time, Power and Thermal Metrics for 128 bits

CPU Time	0.12 seconds
Elapsed Time	0.20 seconds
Total On-Chip Power (W)	137.829
Design Power Budget (W)	Unspecified*
Power Budget Margin (W)	NA
Dynamic (W)	136.713
Device Static (W)	1.116
Effective TJA (C/W)	1.5
Max Ambient (C)	0.0
Junction Temperature (C)	125.0
Confidence Level	Low
Setting File	—
Simulation Activity File	—
Design Nets Matched	NA

Table 18: On-Chip Power Utilisation for Circuit Components for 128 bits

On-Chip	**Power (W)**	**Used**	**Available**	**Utilization (%)**
Slice Logic	3.575	846	—	—
LUT as Logic	3.389	466	134600	0.35
Register	0.181	274	269200	0.10
BUFG	0.005	1	32	3.13
Others	0.000	2	—	—
Signals	10.798	979	—	—
I/O	122.340	388	500	77.60
Static Power	1.116			
Total	137.829			

Table 19: Slice Logic Utilization for 128 bits

Site Type	**Used**	**Fixed**	**Prohibited**	**Available**	**Util (%)**
Slice LUTs	469	0	0	134600	0.35
LUT as Logic	469	0	0	134600	0.35
LUT as Memory	0	0	0	46200	0.00
Slice Registers	274	0	0	269200	0.10
Reg as Flip Flop	274	0	0	269200	0.10
Reg as Latch	0	0	0	269200	0.00
F7 Muxes	0	0	0	67300	0.00
F8 Muxes	0	0	0	33650	0.00

Table 20: Performance Metrics for Rule Vector γ

Parameter	**64-bit block**	**128-bit block**
Dieharder Tests Passed	112/114	111/114
Weak Behaviour in Dieharder	1 tests	3 tests
NIST Tests Passed	All	All except 1
SmallCrush Tests Passed	All	All
Software Output Time	16 ms	28 ms
Hardware Output Time	10 s	20 s
Avg. Encryption Time	15.2 μs	28.3 μs
Avg. Decryption Time	150.7 μs	314.1 μs

of gate minimization for hardware implementation and SAC test identified one rule vector, γ as the optimal choice. Based on this rule vector as a key, a basic hardware encryption modulo is developed to work in the CBC mode, containing two layers.

Table 21: Performance Evaluation of 64-bit and 128-bit Designs

Metric	**64-bit**	**128-bit**
Time (CPU)	4s	12s
Total Power (W)	70.610	137.829
Dynamic Power (W)	69.375	136.713
Slice LUTs	238	469
Slice Registers	144	274
Time Elapsed	10 s	20 s

Finally, this hardware model was simulated in `Vivado`, and its performance was analyzed. Overall, our findings validate that reversible nonlinear nonuniform ECAs are highly effective for generating encrypted data with strong randomness properties and for building a scalable, robust cryptographic module.

From the perspective of natural computing, our results demonstrate that reversible nonlinear CAs can serve as practical, nature-inspired cryptographic primitives, combining emergent complexity with hardware efficiency.

Future studies may explore further optimizations of the hardware model. In addition, for IV, `MT19937` is used, which is not hardware implementable. Instead, a useful approach is to use CA-based good PRNGs, which are implementable in hardware. Moreover, as we repeat some scalable rules, this may eventually result in patterns that could be exploited, leading to vulnerabilities in the cryptographic system. Further analysis of this needs to be performed in the future. In addition, we have only shown the encryption module; the decryption module for the symmetric key approach, which will be similar, can be implemented in the future.

Acknowledgment

This work was partially carried out as a project at the Indian Summer School on Cellular Automata 2024. The authors are grateful to Mr. Tarun Lywait, Mr. Kiran Srinivasan and Mr. Krishnadas Nair for implementing the first version of the work and to Mr. Vishesh Bhatia and Mr. Rushi Moliya for their participation in the initial part of the current work. We would like to extend our deepest gratitude to Prof. Sukanta Das for his insightful advice, encouragement, and unwavering mentorship, and the Indian Summer School of Cellular Automata 2024 for the resources and support needed for the completion of this paper.

This work is partially supported by Start-up Research Grant (File number: SRG/2022/002098), SERB, Department of Science & Technology, Government of India

References

[1] Wolfram, S. (1986). Cryptography using cellular automata. In Advances in Cryptology—CRYPTO'85 Proceedings 5 (pp. 429-432). Springer Berlin Heidelberg.

[2] Wolfram, S.. A New Kind of Science. Wolfram Media, 2002.

[3] Tomassini, M., & Perrenoud, M. (2001). Cryptography with cellular automata. Applied Soft Computing, 1(2), 151-160.

[4] Chaudhuri, P. P., Chowdhury, D. R., Nandi, S., & Chattopadhyay, S. (1997). Additive Cellular Automata: Theory and Applications, Volume 1 (Vol. 43). John Wiley & Sons.

[5] Mukhopadhyay, D., RoyChowdhury, D. (2004). Cellular Automata: An Ideal Candidate for Block Ciphers. In: Ghosh, R.K., Mohanty, H. (eds) Distributed Computing and Internet Technology. ICDCIT 2004. Lecture Notes in Computer Science, Vol. 3347. Springer, Berlin, Heidelberg.

[6] Banerjee, T., Roy Chowdhury, D. (2021). EnCash: an Authenticated Encryption scheme using Cellular Automata. In: Gwizdałła, T.M., Manzoni, L., Sirakoulis, G.C., Bandini, S., Podlaski, K. (eds) Cellular Automata. ACRI 2020. Lecture Notes in Computer Science(), vol. 12599. Springer, Cham.

[7] Nandi, Sukumar, Kar, B., and Pal, Pushkar. (1995). Theory and Applications of Cellular Automata. Computers, IEEE Transactions on. 43. 1346 1357. 10.1109/12.338094.

[8] M. Corona-Bermúdez, E. J. Del-Rey, and M. Valencia-Barragán, "Security Scheme Based on Cellular Automata and Zero Knowledge Protocol," *Electronics*, vol. 13, no. 13, 2022.

[9] Das, S., RoyChowdhury, D. CAR30: A new scalable stream cipher with rule 30. Cryptogr. Commun. 5, 137–162 (2013).

[10] Mariot, L., & Leporati, A. (2018). A cryptographic and coding-theoretic perspective on the global rules of cellular automata. Natural Computing, 17, 487-498.

[11] Lywait, T., Srinivasan, K., Nair, K., Bhattacharjee, K. (2024). A Scheme for Symmetric Cryptosystem Using Large-Cycle Reversible Cellular Automata. In: Bagnoli, F., Baetens, J., Bandini, S., Matteuzzi, T. (eds) Cellular Automata. ACRI 2024. Lecture Notes in Computer Science, Vol. 14978. Springer, Cham.

[12] A. Janz, D. Reinel, and A. Keller, "Construction of reversible CA for cryptographic applications," *Proceedings of the 6th International Conference on Cellular Automata for Research and Industry (ACRI)*, 2006.

[13] Mukherjee, S., Adak, S., Bhattacharjee, K., and Das, S.. Non-uniform Non-linear Cellular Automata with Large Cycles and Their Application in Pseudo-Random Number Generation. *International Journal of Modern Physics C*, 32, 2021. doi:10.1142/S0129183121500911.

[14] Brown, R. G., Eddelbuettel, D., and Bauer, D. Dieharder: A Random Number Test Suite. https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[15] Matsumoto, M. and Nishimura, T. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8, 1 (Jan. 1998), 3–30.

[16] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., and Vo, S. Statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST special publication. revision 1a, volume 800-22. National Institute of Standards and Technology, Technology Administration, U.S. Department of Commerce, 2010.

[17] L'Ecuyer, P. and Simard, R. TestU01: A C library for the empirical testing of random number generators. ACM Transactions on Mathematical Software, 33(4):22:1–22:40, 2007.

[18] Z. Kotulski, M. Kutyłowski, and K. Pawlak, "Multi-Layer Cryptosystem Using Reversible Cellular Automata," *Electronics*, vol. 14, no. 13, pp. 2627–2645, 2024.

[19] A. Ostapov, V. Pilgun, and Y. Mashko, "Cryptographic Services Based on Elementary and Chaotic Cellular Automata," *Electronics*, vol. 11, no. 4, pp. 613–630, 2023.

[20] T. Jeyaprakash, V. Ranganathan, and R. S. Prakash, "KAMAR: A Lightweight Feistel Block Cipher Using Cellular Automata," *Wireless Sensor Network*, vol. 8, pp. 11–23, 2016.

[21] T. Good and M. Benaissa, "Very small FPGA application-specific instruction processor for AES," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 7, pp. 1477–1486, 2006. https://www.researchgate.net/publication/3451286_Very_small_FPGA_application-specific_instruction_processor_for_AES

[22] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "ASCON v1.2: Lightweight authenticated encryption and hashing," *NIST Lightweight Cryptography Standard*, 2023. https://github.com/ascon/ascon-hardware

[23] S. Das and D. Roy Chowdhury, "CAR30: A new scalable stream cipher with rule 30," *Cryptography and Communications*, vol. 5, no. 2, pp. 137–162, 2013, DOI: 10.1007/s12095-012-0079-1. https://www.researchgate.net/publication/257770233_CAR30_A_new_scalable_stream_cipher_with_rule_30

[24] S. Karmakar, D. Mukhopadhyay, and D. Roy Chowdhury, "CAvium Strengthening Trivium stream cipher using cellular automata," *Journal of Cellular Automata*, vol. 7, no. 2, pp. 179–197, 2012. https://www.researchgate.net/publication/267469445_CAvium_-_Strengthening_Trivium_stream_cipher_using_Cellular_Automata

[25] S. Banik *et al.*, “GIFT-COFB v1.1: Lightweight authenticated encryption,” *NIST LWC Finalist*, 2022. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf

[26] M. Hell *et al.*, “Grain-128AEADv2: A lightweight AEAD stream cipher,” *NIST LWC Finalist*, 2022. https://grain-128aead.github.io/

[27] A. Bogdanov *et al.*, “PRESENT: An ultra-lightweight block cipher,” in *Proc. CHES 2007*, LNCS 4727, pp. 450–466, Springer, 2007, DOI: 10.1007/978-3-540-74735-$2_2$8. https://www.iacr.org/archive/ches2007/47270450/47270450.pdf

[28] J. G. Pandey, T. Goel, and A. Karmakar, “An efficient VLSI architecture for PRESENT block cipher and its FPGA implementation,” *Journal of Circuits, Systems and Computers*, vol. 26, no. 3, 2017, DOI: 10.1007/978-981-10-7470-$7_2$7. https://www.researchgate.net/publication/321958506_An_Efficient_VLSI_Architecture_for_PRESENT_Block_Cipher_and_Its_FPGA_Implementation