Real-Time Encryption Using Cellular Automata

by

Jeremy W. Clark
250053182

Aleksander Essex
001661727

Electrical/Computer Engineering Project Report (ECE 416)
Department of Electrical and Computer Engineering
The University of Western Ontario
London, Ontario, Canada

Real-Time Encryption Using Cellular Automata

by

Jeremy Clark and Aleksander Essex

Faculty Advisor: Dr. S. Primak

Electrical/Computer Engineering Project Report (ECE 416)
submitted in partial fulfillment of the requirements
for the degree of Bachelor of Engineering Science

Department of Electrical and Computer Engineering
The University of Western Ontario
London, Ontario, Canada
March 26, 2004

Abstract

A private-key cryptosystem based on cellular automata has been designed and implemented. The initiative of this endeavour is to develop a real-time encryption algorithm for application in high-speed wireless networking. Cellular automata are a class of computational system that can generate chaotic behaviour from simple arrangements. The properties of these systems have been cultivated to produce fast pseudo-random number generators for encipherment. A symmetrical combination of a cipher with a data stream forms the basis of encryption. Consideration is given to known attacks on related systems and preventative measures are formulated. A standardized statistical analysis of cryptographically-secure randomness, as well as statistical measurements of the avalanche effect, are invoked on the cipher and discussed. An implementation of the system in software is offered as proof-of-concept. To improve encryption rates, a hardware design is developed and simulated. Resultant throughput is discussed and compared with current wireless network standards. Finally, recommendations addressing continued work in this project are offered.

Keywords: cellular automata, chaotic dynamical systems, cryptanalysis, cryptography, IEEE 802.11, private-key encryption, pseudo-random number generation, security, stream ciphers, symmetric ciphers, wired equivalent protocol

Acknowledgements

Our gratitude is extended to Dr. Serguei Primak whose wisdom and guidance was instrumental in our receipt of the *Computer Engineering Design Day* award. We were honoured to have had his fine standards as our aim. Additionally, this project was benefit to many insightful contributions by Dr. Christopher Essex, to whom special recognition is given.

Contents

Abstract **ii**

Acknowledgements **iii**

List of Tables **vi**

List of Figures **vii**

List of Programs **viii**

1 Introduction **1**
1.1 Overview and Motivation 1
1.2 Design Considerations 2
1.3 Outline of Report 2

2 Statement of the Problem **3**

3 Literature Review **4**
3.1 Encryption Models 4
3.1.1 Overview 4
3.1.2 Private-Key Stream Ciphers 5
3.2 Pseudo-Random Number Generation 6
3.2.1 Overview 6
3.2.2 Cellular Automata 8
3.3 Cryptosystems using Cellular Automata 9
3.3.1 Wolfram's Cryptosystem 9
3.3.2 Meier-Staffelbach Attack 10
3.3.3 Hybrid Cellular Automata 11

4 Methodology of Solution **12**
4.1 General Overview 12
4.2 Cryptanalysis Techniques 13
4.2.1 FIPS 140-1 Statistical Tests For Randomness 13
4.2.2 Avalanche Effect 14
4.3 Direction from Literature 15
4.4 Shrinking Generator 16
4.5 System Implementation and Acceleration 17

5 Cryptosystem Design **19**
5.1 Introduction . . . 19
5.2 Java Implementation . . . 19
5.2.1 Cellular Automata Class . . . 19
5.2.2 FIPS 140-1 Testfile Generator . . . 20
5.2.3 Cryptosystem Proof of Concept . . . 20
5.3 Hardware Implementation . . . 21
5.4 VLSI Implementation . . . 22

6 Discussion of Results **23**
6.1 Statistical Analysis . . . 23
6.2 Avalanche Effect . . . 24
6.3 Session Length . . . 26
6.4 Encryption Rates . . . 27

7 Conclusions and Recommendations **28**
7.1 Conclusions . . . 28
7.2 Recommendations for Future Work . . . 28
7.2.1 Shrinking Generator Acceleration . . . 28
7.2.2 Full VLSI Layout . . . 29
7.2.3 Key Distribution . . . 29
7.2.4 Hybrid CA Synthesis . . . 30
7.2.5 Pre-Cipher Compression . . . 30
7.2.6 Physical Cryptanalysis . . . 31

A Program Source Code in Java **32**
A.1 Key - Key.java . . . 32
A.2 Cellular Automata - CellAuto.java . . . 33
A.3 Sender - Sender.java . . . 36
A.4 Receiver - Receiver.java . . . 37
A.5 Cellular Automata Pseudo-Random Number Generator - CAprng.java . . . 38

B Circuit Design and Timing Diagrams **39**
B.1 Cellular Automata . . . 39
B.2 Stream Cipher . . . 40

C Circuit Layout in $0.35\mu m$ CMOS **41**
C.1 4-Cell CA (Rule 30) . . . 41

Bibliography **42**

Vitae **44**

List of Tables

4.1 Acceptable Number of Runs of Specified Length 14

6.1 FIPS 140-1: Passed Tests out of 1000 23
6.2 Wireless Network Transfer Rates 27

List of Figures

3.1 Block-Diagram of Private-Key Cryptosystem 5
3.2 Rule 30 from Simple and Random Initial Conditions 9
3.3 Wolfram's Cryptosystem 10
3.4 Meier-Staffelbach Completion Backwards 10

4.1 Block Diagram of Research Trends 15
4.2 CA Shrinking Generator 16

6.1 Percent Change to Keystream - Statistical Results 24
6.2 Visualization of the Avalanche Effect 25

B.1 Cellular Automata Circuit Diagram 39
B.2 Stream Cipher Circuit and Timing Diagram 40

List of Programs

Key.java 32
CellAuto.java 33
Sender.java 36
Receiver.java 37
CAprng.java 38

Chapter 1

Introduction

1.1 Overview and Motivation

The absence of a suitable encryption algorithm for wireless networking is the motivation for our project. The introduction of wireless networking has brought, with its convenience and connectivity, a cascade of security concerns. Digital eavesdropping is difficult on virtual private networks secured with firewalls and routers, but wireless networks have no physical infrastructure to hide behind. Users communicate with their router through an omnidirectional broadcast, and anyone within range can listen in. In a personal setting, this is undesirable. In a corporate setting, it is unacceptable.

Employing encryption on all network traffic can dissuade digital eavesdropping, but developing sufficiently fast algorithms is a surprisingly difficult task. The majority of encryption standards in use today were developed for application on small packets of data like passwords, certificates, and credit card numbers. An attempt to encrypt all network traffic with them is likely to result in catastrophic delay.

Current IEEE 802.11 protocols use a standard of encryption called the *Wired Equivalent Privacy*. It has been demonstrated to be insecure, and utilities which could be employed to compromise the system are widely available online. Although no system is completely secure, recovering the encrypting key should not be accessible with typical computational power in a reasonable period of time.

1.2 Design Considerations

Two primary initiatives will be undertaken: speed and security. It is our intention to develop a cryptosystem that will not introduce latencies if implemented on a wireless network. This will be the motivation in choosing an encryption model to base our system upon, as well as computationally fast systems to perform real-time enciphering operations. Once the system design is finalized, it will be developed and implemented in hardware to further increase encryption rates.

The second prerogative is security. Design decisions will be made to ensure the enciphering operations provide strong cryptographic properties, and are not susceptible to cryptanalysis. Statistical analysis will be used to demonstrate the security of our final design.

The aim of our project is to demonstrate the system's functionality, security, and speed, as proof-of-concept. We hope to attract industry interest so our work may be fully realized as an application-specific integrated circuit, which can be integrated into commercial wireless devices.

1.3 Outline of Report

The report will resume in Chapter 2 with a formal statement of the problem. This will be followed in Chapter 3, with a review of relevant models and algorithms as presented in literature, that we will use a foundation for the design decisions in Chapter 4. The methodologies will be discussed in terms of viable alternatives and a chosen design will be formulated. This final design will be implemented in Chapter 5, and the results of the implementations will be presented and discussed in Chapter 6. Conclusions and a series of recommendations for future work will be offered in Chapter 7. Appending the report is a collection of source code, circuit schematics, and timing diagrams referenced from various chapters of the report.

Chapter 2

Statement of the Problem

To design and implement a cryptosystem with the following properties:

1. Cryptographically-secure pseudo-random number generation

2. Strong sensitivity to keys

3. Counters known attacks on related cryptosystems

4. Encryption rates surpassing 802.11g specifications

Chapter 3

Literature Review

3.1 Encryption Models

3.1.1 Overview

RSA

Public-key cryptosystems were developed to address the problem of key exchange over insecure channels. RSA is the most widely use and tested public-key cryptosystem. Its security is reliant on the assumed computational intractability of factorizing two large prime numbers. The key is asymmetric, whereby the *encrypting* key of a particular person is publicly distributed to whomever wishes to send that person an encrypted message. The person also possesses a corresponding and secret *decrypting* key. The asymmetry of the system makes the determination of the secret key from the public key infeasible. Public-key encryption is much slower than private-key and other symmetric systems. The algorithm itself involves exponential modulo operations on numbers exceeding 1024 bits to be reasonably secure by present standards [1]. This is not conducive to real-time encrypted communication.

Advanced Encryption Standard

The National Institute of Standards and Technology (NIST) of the United States currently uses the Rijndael private key block cipher algorithm as its Advanced Encryption Standard (AES). The algorithm itself performs successive rounds of substitution, shift, mix and addition operations on 128 bit data blocks [2]. The algorithm is usually implemented in software, making efficient use of parallel pipelined processors, allowing it to achieve high encryption rates. Hardware proposals promise further improved encryption rates, however as key size is increased, encryption rates are reduced.

Wired Equivalent Privacy

The *IEEE 802.11* standard for wireless data communication employs the Wired Equivalent Privacy (WEP) protocol for encryption. This standard is currently the most widely used in commercial applications. WEP uses the RC4 encryption algorithm, but its implementation has been discovered to be susceptible to key scheduling weakness. Numerous effective attacks against WEP including [3] have been mounted, rendering it highly insecure.

3.1.2 Private-Key Stream Ciphers

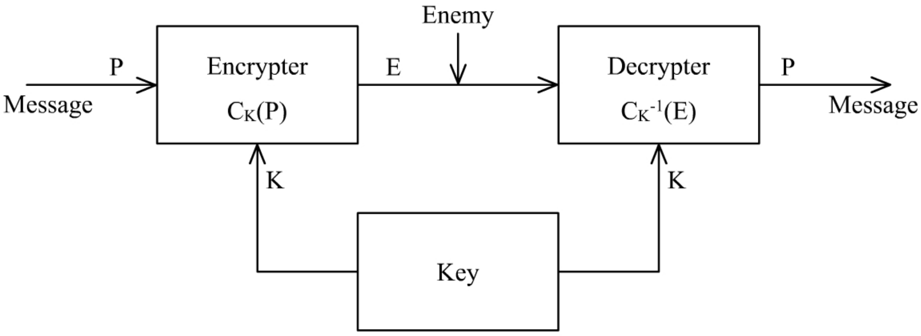


Figure 3.1: Block-Diagram of Private-Key Cryptosystem

The cryptosystem depicted in Figure 3.1 operates with a single, private key. For the immediate discussion, assume this key is held *a priori* by both sender and receiver. The plaintext message is denoted P, and is encrypted by the sender using the transformation $C_K(x)$. The resultant ciphertext E is sent through an insecure channel to the receiver. Thus,

$$E = C_K(P) \tag{3.1}$$

To restore the plaintext, the receiver performs the inverse transformation $C_K^{-1}(x)$ on the ciphertext

$$P = C_K^{-1}(C_K(P)) = C_K^{-1}(E) \tag{3.2}$$

The transformation function is employed individually on successive bits of the plaintext

$$C_K(P) = \{C_{K_1}(p_1), C_{K_2}(p_2), ...\} \tag{3.3}$$

This process, called stream ciphering, is extremely fast and well suited for serial communication. The transformation function is implemented as a logical exclusive-or (XOR) between a plaintext bit and a random cipher bit s_k. The XOR function has the added advantage of being a symmetric transformation. Thus,

$$e_i = C_i(p_1) = p_i \oplus s_{k_i} \tag{3.4}$$

$$p_i = C_i^{-1}(e_i) = p_i \oplus s_{k_i} \oplus s_{k_i} = e_i \oplus s_{k_i} \tag{3.5}$$

The so-called *one-time pad* is defined by

$$e_i = p_i \oplus s_i \qquad for\ i = 1, 2, 3 \ldots, \tag{3.6}$$

where keystream s was generated independently and randomly, and $s_k \equiv s$ in this case. This technique has been proved unconditionally secure [4]. However to encrypt a message of length L, both the sender and receiver are both required a complete and *a priori* knowledge of s, where $|p| = |s| = L$. This poses the problem of secure transport of an L length key to the receiver. As an alternative, the random bit sequence s can be generated by an algorithm, uniquely determined by key k, called a *pseudo-random number generator*. Presumably $|k| << L$, greatly alleviating the problem of key transport.

3.2 Pseudo-Random Number Generation

3.2.1 Overview

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.” These words once spoken by John von Neumann are a reminder that when one speaks of *random* in this context, they in fact mean *pseudo-random*. The sequence generated by a PRNG, although random in appearance, is fully deterministic. Any two PRNG with identical internal connections, with an identical key k, can produce an identical keystream s.

Linear Feedback Shift Registers

A linear feedback shift register (LFSR) of N stages consists of N single-bit registers. The output sequence $s = s_0, s_1, s_2 \ldots$ is fully determined by an iterative process described as

$$s_j = (c_1 s_{j-1} + c_2 s_{j-2} + \ldots + c_N s_{j-N})\ mod\ 2 \qquad for\ j \geq N \tag{3.7}$$

where $c_i \in \{0, 1\}$ and is the connection coefficient of register i. The updated state of register i, denoted by s_i^* at the next time step is

$$s_i^* = s_{(i-1)\ mod\ N} \tag{3.8}$$

The LFSR has been in widespread use over the last half century, finding use in applications requiring pseudo-random sequence generation, such as *Spread-Spectrum* and *Ultra-Wideband* communications. Unfortunately, the inherent linearity of a LFSR allows for effective cryptanalysis [5]. The security of such systems is thus limited.

Chaotic Regions of Differential Equations

Recent advances have been made using chaotic regions of differential equations to act as a cipher mask for a signal. The sender and receiver synchronize onto a chaotic carrier waveform such as the Rössler system denoted by

$$\begin{aligned} \dot{x} &= -y - x, \tag{3.9} \\ \dot{y} &= x + \alpha y, \tag{3.10} \\ \dot{z} &= \alpha + z(x - \mu), \tag{3.11} \end{aligned}$$

This method as outlined in [6] has been shown to possess attractive cryptographic properties. The major obstacle of such methods has been in a viable implementation. Using analog electronics, these chaotic waveforms are readily generated, but this is not suitable for wireless transmission in the framework of modern protocols. Conversely, the system could be evolved on discrete computers requiring numerous floating-point operations. The speed of that method is therefore unsuitable for real-time encryption. A desired system is then one that combines the inherent digital properties of the LFSR with the chaotic behaviour of differential equations.

3.2.2 Cellular Automata

Cellular Automata (CA) are deterministic dynamical systems. They are space-discrete forming a finite array of N elements. Each element, called a cell, is state-discrete, and we will only consider binary cells. Thus,

$$C(t) = [s_0(t), s_1(t), ..., s_{N-1}(t)], \qquad s \in \{0, 1\} \tag{3.12}$$

CA are time-discrete and the entire system updates with every time integral. $C(0)$ is the initial condition, and the new state of each cell is fully determined by the past state of the cell and its immediate neighbours. Thus for cell i in a three neighbourhood system,

$$s_i(t+1) = f(s_{i-1}(t), s_i(t), s_{i+1}(t)) \tag{3.13}$$

Boundary cells do not have a full parent neighbourhood, requiring a more precise neighbourhood definition to resolve the problem. We will consider wrap-boundary systems where a neighbour of cell i is defined as: $(i \pm 1)\text{mod}N$, for a CA of width N. For example,

$$s_0(t+1) = f(s_{N-1}(t), s_0(t), s_1(t)) \tag{3.14}$$

The transformation function f is called a rule and it is uniformly applied to each cell in the system at every time-step. Each possible neighbourhood combination may be placed in a lookup table. The rule is named by decimal equivalent of the outputs. For instance, the following rule is denoted Rule 30, and may be alternatively expressed as a Boolean equation.

$$\begin{gathered} f(111) = 0,\ f(110) = 0,\ f(101) = 0,\ f(100) = 1 \\ f(011) = 1,\ f(010) = 1,\ f(001) = 1,\ f(000) = 0 \end{gathered}$$

$$s_i(t+1) = s_{i-1}(t) \oplus (s_i(t) + s_{i+1}(t)) \tag{3.15}$$

There are 2^{r^N} possible rules for a one-dimensional, r state, N neighbourhood CA. In the case of binary, 3-neighbourhood CA, there are 2^{2^3} or 256 possible rules. Of these rules, a few exhibit chaotic behaviour. Stephen Wolfram was the first to propose CA-based pseudo-random number generators, and he was particularly interested in Rule 30 [7]. CA may be represented on a horizontal lattice and evolved downwards with time. Rule 30 of width $N = 300$ as evolved from a simple initial condition is shown in Figure 3.2(a). This demonstrates that the complexity is intrinsic and not inherited from the initial condition itself. In practice, initial conditions are chosen randomly as shown in 3.2(b).

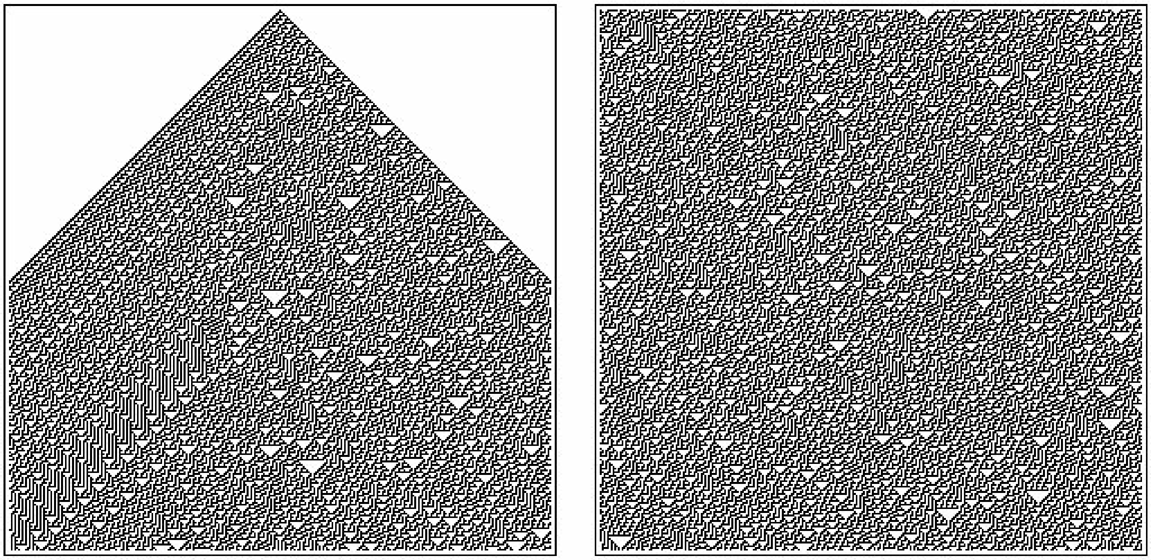


Figure 3.2: Rule 30 from Simple and Random Initial Conditions

3.3 Cryptosystems using Cellular Automata

3.3.1 Wolfram's Cryptosystem

A CA-based cryptosystem was first proposed by Stephen Wolfram in 1985 [8]. The system is based on the stream ciphering technique described in Equations 3.4 and 3.5, and uses a CA to produce the random bit s_{k_i}. The private-key K encapsulates the initial condition of the CA, $C(0)$ in Equation 3.12, which is determined with a true random number generator. The evolution of the CA is governed by Rule 30, as described in Equation 3.15, and the resultant is a chaotic dynamical system.

The cipher is generated by sampling the contents of one cell of the system at each time step. In Figure 3.3, this corresponds to a column of cells. This column is XORed with the plain text as shown on the left to produce the ciphertext shown on the right. As mentioned, the key size is equivalent to width N of the CA, and Wolfram empirically determined that the period length of the cipher approximates

$2^{0.63N}$. Wolfram, in a more rigorous treatise of the topic in 2003 [9], also cautions that a small number of initial conditions will yield simple repetitious behaviour.

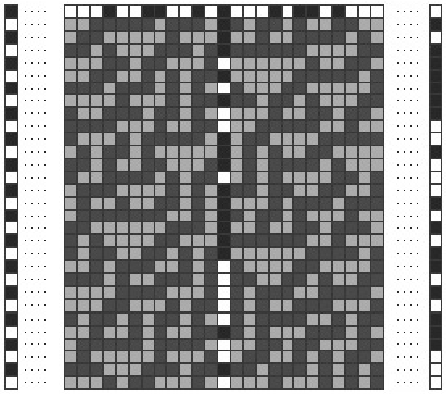


Figure 3.3: Wolfram's Cryptosystem

Wolfram optimistically suggested the complexity of recovering the key from the cipher is $O(2^N)$. As it turns out, a successful cryptanalysis of his system would be found.

3.3.2 Meier-Staffelbach Attack

A method of cryptanalysis against Wolfram's cryptosystem was outlined by Meier and Staffelbach in [10]. Their attack exploits the geometry of the output column along with the partial linear properties of rule 30 itself. Using these properties, combined with the observed output sequence, portions of the internal evolution of the automata can be reconstructed with great likelihood. Their technique called "completion backwards" is illustrated in Figure 3.4.

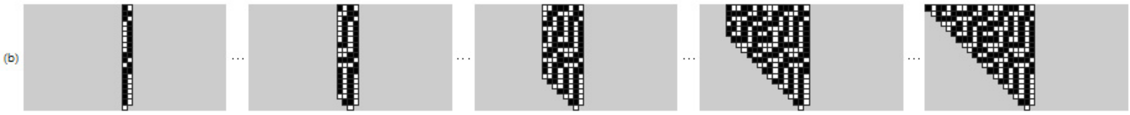


Figure 3.4: Meier-Staffelbach Completion Backwards

The results of experimentation suggest that attackers using dedicated hardware could potentially crack CA based cryptosystems of up to 1000 cells in width, leading to their recommendation of even larger systems. This however is undesirable because of increased hardware cost and key size. An alternative to a larger system that can still effectively counter the Meier-Staffelbach attack is therefore more desirable.

3.3.3 Hybrid Cellular Automata

In wake of the Meier-Steffelbach attack, a series of countermeasures were reflected in literature. A critical reaction to the simplicity of Wolfram's cipher sought more complexity in the rule governing the cellular automata. Nandi *et al* proposed a so-called programmable cellular automata (PCA) in [11], for implementation in block and stream ciphering. Unlike Wolfram's uniform CA, where every cell is governed by the same rule, PCA are hybrid systems capable of dynamically alternating between two chaotic rules.

Two methods of rule selection are proposed: control instructions are generated with a $L \times L$ ROM segmented into I invertible matrices, or with an additional L sized CA. Blackburn *et al* demonstrated insecurities of both systems in [12]. The ROM-generator PCA was demonstrated to have mere polynomial complexity $O(L^2 I)$, while the complete key of the dual-PCA system was recovered in 2^{L+2} trials – a significant reduction of the 2^{3L} trials suggested in [11].

Tomassini *et al* suggest four rule hybrid CA in [13]. Departing from designed rule selectors, the cipher is artificially evolved using a genetic algorithm. Each cell and its three predecessors in time form a population. At every evolutionary time step, the entropy of the population is evaluated, and this influences the cell's fitness score. Cells which evolve into uniformly distributed streams (the conditions for maximum entropy) are considered fittest. Transformations, induced periodically to promote fitness, include bit flipping (mutations) and rule switches (crossover). The result is excellent PRNG and cryptographically strong ciphers. However, the time-cost associated with constant fitness evaluations does not lend the system to real-time applications.

Chapter 4

Methodology of Solution

4.1 General Overview

In Chapter 2, the problem of performing strong encryption in real-time was formulated. Chapter 3 presented relevant theories and models from which our proposed solution will be derived. It is hoped that two initiatives are maintained in reaching this solution – maximizing computational speed while perpetuating a high level of security.

As discussed, stream ciphers are accommodating for high data transfer rates and will thus be implemented in our scheme, as opposed to the design alternatives of public key and block cipher systems. A deterministic method of ciphering is required to ascertain the plaintext's restoration. However to ensure the statistical properties of the plaintext are destroyed, the cipher must also possess random properties to thwart cryptanalysis.

Consideration was given to three dynamical systems that were both deterministic and chaotic. Cellular automata have been shown to possess the computational speed of linear feedback shift registers while exhibiting intrinsic chaotic behaviour like the Rössler equations, and thus are chosen. However, in order to surmount the security problems of similar initiatives, careful consideration of cryptanalysis techniques will precede further design decisions.

4.2 Cryptanalysis Techniques

4.2.1 FIPS 140-1 Statistical Tests For Randomness

Motivation

The first essential step toward cryptanalysis of the system is to perform statistical hypothesis testing on output bitstreams as a valuation of overall randomness. Several basic statistical tests proposed in [5] were selected to form the basis for testing. Instead of personally selecting suitable significance levels for our statistics, it was decided to build testing into the framework of the FIPS 140-1 test of randomness. This test was developed by NIST and provides explicit bounds of significance, assigning a simple pass/fail to 20,000 bit samples produced by a PRNG. The tests and pass-conditions specified by FIPS 140-1 are outlined below.

Monobit Test

It is expected that for a random[1] sequence, the overall number of 0's and 1's in cipher s are approximately equal. Letting n_0, n_1 denote the number of 0's and 1's in s respectively, the monobit statistic is defined as,

$$X_{monobit} = \frac{(n_0 - n_1)^2}{n} \tag{4.1}$$

For a trial performed on 20,000 bits, the test is passed if $X_{monobit} < 23.9432$.

Poker Test

It is expected that for a random sequence, the overall number of m bit combinations, or *poker hands*, in s are approximately equal. Thus s is divided into k non-overlapping sequences of m bits in length. The poker statistic is accordingly defined as,

$$X_{poker} = \frac{2^m}{k}\left(\sum_{i=1}^{2^m} n_i^2\right) - k \tag{4.2}$$

FIPS 140-1 operates on 20,000 bit blocks and 4 bit *poker hands* thereby stipulating $m = 4$ and $k = 5000$. The test is passed if $1.03 < X_{poker} < 57.4$.

[1] It is important to note that the term "random" in this case specifically means "cryptographically-secure pseudo-random." When talking about the monobit test, it should be mentioned that true random sequences can in fact contain biases toward 1's or 0's. The FIPS 140-1 tests merely specify properties that are attractive for the purposes of encryption.

Runs Test

A run of length i is defined as a sequence of i consecutive bits of either all 1's or all 0's. A run of 1's is referred to as a *block*, while a run of 0's is referred to as a *gap*. Let B_i, G_i respectively be the overall number of blocks and gaps of i bits in length. The number of blocks and gaps of length i in a random sequence of length n is expected to be $e_i = (n - i + 3)/2^{i+2}$. The runs statistic is then defined as

$$X_{runs} = \sum_{i=1}^{k} \frac{(B_i - e_i)^2}{e_i} + \sum_{i=1}^{k} \frac{(G_i - e_i)^2}{e_i} \tag{4.3}$$

The test passes if the number of runs that occur within a 20,000 bit sample is within the corresponding interval specified in Table 4.1.

Length of Run	Required Interval
1	2267–2733
2	1079–1421
3	502–748
4	223–402
5	90–223
6+	90–223

Table 4.1: Acceptable Number of Runs of Specified Length

Long Runs Test

A long run is defined as any block or gap greater than 34 bits in length. (i.e. $i > 34$) The test is passed if there are *no* long runs.

4.2.2 Avalanche Effect

An important cryptographic property that our system should exhibit is something known as the avalanche effect, as originally outlined in [14]. We will define the avalanche effect for our purposes as follows: a minor change to the key k must result in significant and random-looking changes to the keystream s. Specifically, for a *good* avalanche effect, a change in one bit of key k results in a random change to about 50% of the keystream s. This property is important in cryptosystem design because it allows an attacker no information with regard to how close an attempted key is to the correct key. A poor avalanche effect will allow an attacker to greatly reduce their keyspace search.

There seems to be a good indication in literature that chaotic CA systems will posses a good avalanche effect. We will nonetheless need to define a statistical test to evaluate the avalanche. Let A represent the entire evolution of an w-cell CA to l steps. Let A_i represent the entire evolution of an identically sized CA in which there are i complemented bits in the initial condition. Additionally let the operation $|x|_1$ denote the overall number of 1's in x. We then define our avalanche statistic as,

$$X_{avalanche} = \frac{|A \oplus A_i|_1}{w \times l} \tag{4.4}$$

This statistic determines the number of bits differing between A and A_i and calculates a ratio of that with respect to the overall number of bits. In accordance with the definition, a *good* avalanche effect is described that for $i = 1$ then $X_{avalanche} \approx 0.5$.

4.3 Direction from Literature

Desirable cryptographic properties have been shown. Statistical analysis ensures the cipher has a high entropy in terms of single bits and a four bit aggregation, while the avalanche effect prevents an attacker from diverging to a solution during a key search. However, as the Meier-Staffelbach attack demonstrated, a third assurance must be given: any linear properties of the system must be dissipated.

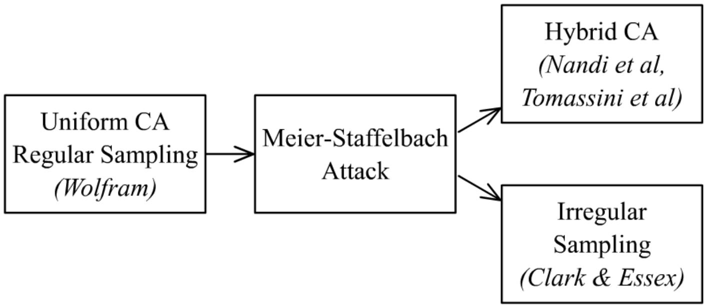


Figure 4.1: Block Diagram of Research Trends

As seen in the previous chapter, the apparent reaction to the Meier-Staffelbach attack on Wolfram's system was to complicate the rules governing the CA. Figure 4.1 depicts this trend in literature, and shows that research is still advancing in this direction as evident in the elaborate CA proposed by Tomassini *et al* in [13].

These systems are still sampling one cell over consecutive time steps to produce the cipher. It is here that we will diverge from the precedent set in literature, and forge a faster alternative than hybrid CA for destroying the partial-linearity in Wolfram's system. In Chapter 7, a recommendation for the synthesis of our methodologies with hybrid CA will be made in hopes of contributing to a true *sui generis* cryptosystem. But for the present, we'll submit a proposal for irregular sampling of the CA.

4.4 Shrinking Generator

A relatively new random sequence generator known as the *shrinking generator* is proposed in [15] whereby the output sequence of a LFSR R_B is used to select a portion of the output sequence of second LFSR R_A. The output sequence is a shrunken version of R_A, also known as an *irregularly decimated subsequence*. Using this idea of sampling a sequence in an irregular fashion, we can effectively eliminate the partial-linearity exploited by the Meier-Staffelbach attack in both a simple and elegant way. Let us now redefine the shrinking generator in the context of a cellular automata. Firstly, we must specify how we derive a serial keystream from cellular automata, which is a parallel process. Let A represent the evolution of a one-dimensional L-cell cellular automata. Let the notation $A_{i,l}$ represent the state of cell index i at line l. We define the serial output sequence s_A of A as,

$$s_A \;=\; A_{1,1}, A_{2,1}, \ldots, A_{L-1,1}, A_{L,1}, A_{1,2}, A_{2,2}, \ldots, A_{L-1,2}, A_{L,2}, \ldots \tag{4.5}$$

Similarly, let B represent the evolution of another one-dimensional cellular automata, and s_B represent the corresponding output sequence. Sequence s_B functions as a controller, irregularly sampling s_A. The keystream s is produced by sampling s_{A_i} when $s_{B_i} = 1$. When $s_{B_i} = 0$, bit s_{A_i} is discarded and does not form part of the keystream s. This process is illustrated in 4.2.

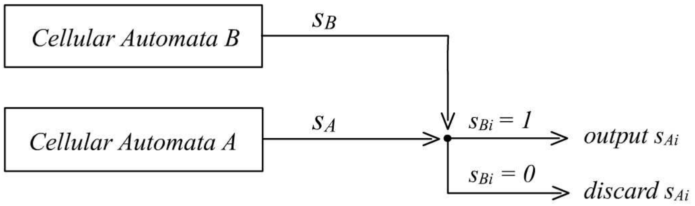


Figure 4.2: CA Shrinking Generator

4.5 System Implementation and Acceleration

In network applications, the implementing of an encryption algorithm is subject to issues of locality and methodology. Application data undergoes a series of transformations before network transmission. The OSI network model describes these transformations as a stack of seven layers, where each layer contains a collection of protocols to ensure proper network functionality. The level at which encryption is employed must be carefully considered.

Encryption is typically engaged at the presentation layer which ensures the data content of a network packet is private but its destination is visible. This allows intermediary devices between the sender and receiver to route the packet. For applications in wireless networks, no device exists between the computer and the router, so security could be improved by implementing encryption at a lower level. However the lower levels offer a collection of protocols aimed at preserving data integrity during transmission. Stream ciphering is contingent on synchronization between the sender and receiver – a lost packet, or even bit, will prevent all subsequent decryption. It is therefore advisable to implement our system at the presentation layer, above the transport layer protocols, like TCP, which offer guaranteed delivery.

Given their high positioning in the OSI model, encryption algorithms are usually programmed in software to allow easy interfacing with networking protocols. Software cryptosystems may be executable from the operating system, but a typical implementation for networking would reside in the firmware of the network card or router. In either event, the CPU of the device is burdened with the task of encryption and decryption. In situations where large amounts of data must be encrypted, software systems running off the CPU will introduce catastrophic latencies. So-called cryptoaccelerators are hardware implementations designed to alleviate this processing burden.

Hardware in this case is much faster solution over software. For example, the two-input XOR in Equation 3.4 would take over 2.6 clock cycles[2] in Intel 8086 assembly-level software, compared to the cost of one gate delay. Furthermore, the processor can never fully dedicate itself to the process of encrypting and decrypting. It is required to

[2] A 16-bit XOR routine and associated execution time [16]:

Instruction	*Clock Cycles*
IN AX, [Port]	12
MOV BX, AX	1
IN AX, [Port]	12
XOR AX, BX	1
OUT [Port], AX	16

compute the random numbers for the cipher, as well as any demands of the operating system and other running applications. In wireless networking, all network data must be encrypted without sacrificing throughput. Therefore a hardware implementation of our system is intended to meet these real-time demands. Our intention is to implement an application-specific integrated circuit to provide fast and dedicated encryption.

Chapter 5

Cryptosystem Design

5.1 Introduction

There were three levels of realization of this cryptosystem. Firstly, a full implementation was written in Java. This was mainly for the purposes of statistical testing and proof-of-concept. For reasons previously discussed, the cryptosystem is not speed-optimal in software. Consequently, a hardware simulation of the system was done in Altera as a *field gate programmable array* (FPGA). Several speed-accelerating design techniques were explored. The original intention however for this system had always been an implementation as an *application-specific integrated circuit* (ASIC). Accordingly, a basic CMOS circuit was developed to support speed claims suggested by the FPGA implementation. Some specifics of our design are discussed in the following.

5.2 Java Implementation

5.2.1 Cellular Automata Class

A generalized program that computes the evolution of a one-dimensional CA was required to base the cryptosystem upon. Refer to *CellAuto.java* in the appendix for a complete listing. This program is an implementation of equation 3.14 which instantiates a CA with a user-specified evolving rule, cell size, neighbourhood, and initial condition. The evolving rule is stored in the array *ruleIndex* functioning as a look-up table (LUT). When a cell $s_i(t+1)$ is being computed, its parent neighbourhood N_i, as specified by

$$N_i \;=\; (\ldots, s_{i-1}, s_i, s_{i+1}, \ldots), \quad s \in 0,1 \tag{5.1}$$

is converted into its decimal value $N_{i_{10}}$. Accounting for wrap boundary conditions as outlined in equation 3.14, the state of cell $s_i(t+1)$ is assigned the binary bit value stored in location *ruleIndex($N_{i_{10}}$).* The system evolution is performed one bit at a time in the order specified by equation 4.5. Public methods are provided to deliver to the user both *bit* and *byte-wise* output.

5.2.2 FIPS 140-1 Testfile Generator

For statistical testing of the cryptosystem, a software utility that performs the FIPS 140-1 tests as specified in section 4.2.1 was used to test the output of the cellular automata class. The utility accepts as input a data file on which to preform the tests. The file represents the keystream output of a PRNG arranged into bytes. A program was then required to generate the keystream output of a CA in byte[1] form, and store that information in a file. Accordingly, the program *CAprng.java* was written to supply the utility with properly formatted data files. It accepts user input defining the parameters of a CA, the number of bytes to generate, and the filename to which the data is output. These data files were used later by the utility.

5.2.3 Cryptosystem Proof of Concept

The encryption and decryption operations specified by equations 3.4 and 3.5 were respectively implemented in programs *sender.java* and *reciever.java* to demonstrate the functionality of the overall cryptosystem. These programs both use the same instance of *cellAuto.java* to generate identical keystreams for encryption/decryption. The encrypted data is transmitted between two separate networked computers using TCP sockets. This protocol was necessary to provide what is called *guaranteed delivery*, which prevents lost or scrambled packets from causing a loss of synchronization to the receiver's decryption operation. Several test files were encrypted and transmitted over an insecure channel (i.e. the network) and successfully decrypted. This exercise demonstrated practical operation of our cryptosystem. However, because of inherent speed limitations of a software implementation, the encryption/decryption

[1] 8 steps in the evolution of a CA, representing 8 data bits, is stored as one byte

operations were performed outside of a real-time environment. Speed considerations are therefore deferred to the hardware implementation.

5.3 Hardware Implementation

As described in the previous chapter, our intention is to implement our cryptosystem in hardware. Reconfigurable devices like field-programmable gate arrays (FPGA) offer a flexible environment for circuit design. Although our ultimate intention is for an ASIC implementation, VLSI layouts are difficult to modify and are often reserved as the final step in the design process. As a proof-of-concept, we developed our cryptosystem in an FPGA simulator to establish a functional and elegant design.

Cellular automata are computational systems and thus well-suited for hardware implementation. Figure B.1 in Appendix B shows our design. Recall from section 3.2.2 that CA form a space-discrete array of N cells. This can be formed with a series of N interconnected flip-flops called a ring register, where the output of the last flip-flop is connected to the input of the first. This forms a logical ring which rotates every clock cycle. Each time-step in the CA is fully determined by the previous time-step, and so only two ring registers are needed to maintain the system.

In literature, the CA rule defined in equation 3.15 is always implemented as a boolean function. As an alternative, we propose that it form a lookup table in RAM. Each output bit of the rule would be stored in memory at the address of its associated neighbourhood. This would allow instant access to a rule's solution, as well the ability to reprogram the system with any rule.

In the circuit diagram, the RAM chip is denoted *LUT* and the two registers, *ring*_16. Upon startup, the *Load* pin is enabled and the initial condition is loaded into the top ring from input $InitVec$. Once loaded, three neighbours are sampled from the top register and placed on the address bus of the RAM. The direct memory addressing scheme of the lookup table returns the new bit. On the next clock-cycle, this value is written to the bottom ring, while the top ring rotates.

This process continues until the entire ring has been read. This event is marked by the most significant bit of a counter, counting up to $2N-1$, changing. An elegant design using several multiplexors swaps the functionality of the rings. For the next N cycles, the bottom ring is read from and the top ring is written to. This functionality switches back and forth after every line, *ad infinitum*.

The output of the CA is used to cipher the plaintext in Figure B.2 using a dual-input XOR gate. The associated waveform diagram is shown, assuming an initial condition and rule has been pre-initialized by the system. As evident from the timing diagram, our design is capable of enciphering one bit per clock cycle.

The final element of our design, the shrinking generator, is not yet implemented. It appears at first to be a trivial addition of a transmission gate governed by a second CA, as perhaps suggested by the block diagram in Figure 4.2. In actuality, this would produce an irregular output, introducing a series of synchronization issues. While we felt a rigorous solution to this problem falls outside the scope of this project, a recommendation will be made for future work in this area.

5.4 VLSI Implementation

In keeping with the spirit of an ASIC implementation, a basic 4-cell layout of rule 30 was done in $0.35\mu m$ CMOS using the *Cadence Virtuoso* design platform. While a full ASIC implementation was far beyond the scope of an undergraduate project, this basic version provides a verification of our intended high-speed design goals. The circuit, as illustrated in C.1, consists of 4 D-type flip-flops which store the current state of each cell. The flip-flops provide storage, updating, and initial loading of cell states. The input of each flip-flop is connected to the output of its own boolean logic implementation of equation 3.15. This circuit differs from the other implementations in that it is a fully parallel CA. This means that each cell has its own hardwired and dedicated rule unit to compute the cell's next state, as opposed to the Java and FPGA versions in which all the cells share one LUT. Effectively 4 parallel keystream bits are generated every clock cycle. The circuit was clocked and tested at 500 MHz. A practical full scale implementation however would be undertook using the LUT style architecture, only producing 1 bit serially per clock cycle. As the circuit was demonstrated to be functional at 500 MHz, it can be so concluded that speeds of 500 Mbps are achievable.

Chapter 6

Discussion of Results

6.1 Statistical Analysis

For the purposes of testing for statistical randomness of our cryptosystem, several test files were generated by the program described in section 5.2.1 and run through the FIPS140-1 test utility. For each trial, a 1000-cell rule 30 CA with a randomly chosen key was evolved, generating a 2.5Mb of test data. FIPS operates on 20,000 bit blocks by assigning each block a pass/fail grade. There were consequently 1000 tests performed for each key/trial. The overall pass statistics are presented in table 6.1.

Trial	Monobit	Poker	Runs	Long Runs
1	993	1000	998	1000
2	995	1000	998	1000
3	991	1000	1000	1000
4	992	1000	999	1000
5	994	1000	999	1000
6	994	1000	999	1000
7	995	1000	996	1000
8	995	1000	999	1000
9	994	1000	998	1000
10	993	1000	998	1000
Average	99.36%	100%	99.84%	100%

Table 6.1: FIPS 140-1: Passed Tests out of 1000

The results show a 99.36% pass rate of the *monobit* test, and a 99.84% pass rate of the *runs* test. When a test was failed, it was four times more likely to be monobit than runs. Additionally through the course of testing, our cryptosystem did not fail the *poker* or *long-runs* test once.

6.2 Avalanche Effect

A statistical test for assessing the avalanche effect of a CA system was proposed in section 4.2.2 whereby two identical CA systems differing only by 1 bit are evolved. The number of differing bits in the evolution is calculated and expressed as a percent of the total bits generated. The expectation of a good-avalanche effect is for 50% change to the keystream. To test for this property, one thousand trials were performed in *Mathematica* in which a 500-cell CA with a randomly chosen key generated 1,000,000 bits of data. The same system, with the key differing by one bit was similarly evolved.

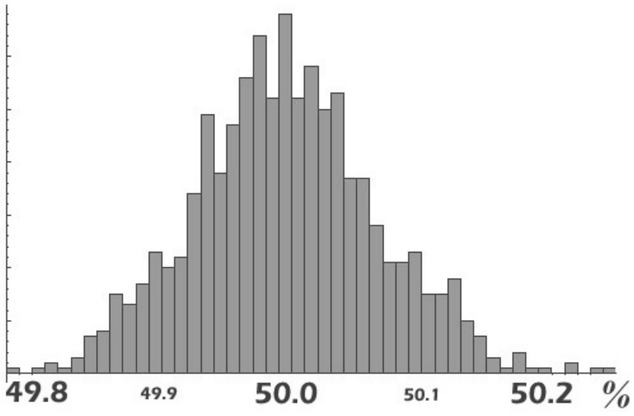


Figure 6.1: Percent Change to Keystream - Statistical Results

The number of changed bits between the two systems was counted and a statistic generated. A histogram of the trials show a tightly bound centering around the expected 50%, thereby indicating a good avalanche effect. Although the statistics are favourable, a certain property of the system must be taken into account to make proper use of the avalanche effect. Consider Figure 6.2 which is a visual representation of the avalanche effect. As the CA evolves downward, the darker region demonstrates how a one bit error in the key (top line) propagates outward through the system. Recall that for our purposes, a CA evolution wraps at the boundaries as is evident half way down in the figure where the error wraps from right to left. The avalanche effect is said to have fully taken hold where the error finally wraps around and meets, as is shown about 3/4 of the way down in the figure.

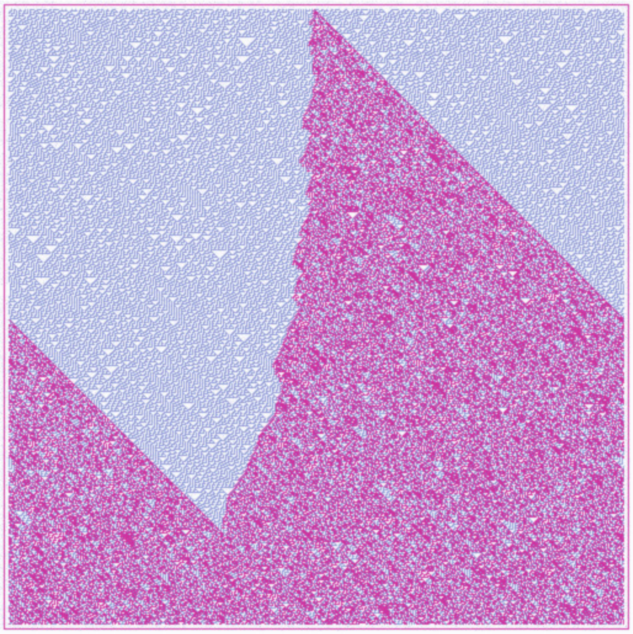


Figure 6.2: Visualization of the Avalanche Effect

In design of the cryptosystem, we do not wish to allow bits of the CA to form part of the keystream until the avalanche effect has fully taken hold – the point at which information about the key is effectively lost. This implies that bits should be discarded at start-up until the avalanche effect has fully taken hold. How long does it take for this to happen? A good measure of the sensitive dependence on the initial conditions is the so-called *Lyapunov* exponents of the system. It represents the averaged rate of divergence of two neighbouring trajectories (e.g. CA evolutions with keys differing by one bit). If we define the term d_k/d_0 to represent the distance between trajectories at iteration k, we can express the Lyapunov exponents of the system as

$$\lambda \ = \ \lim_{k \to \infty} \frac{1}{k} \ln \left(\frac{d_k}{d_0} \right) \quad . \tag{6.1}$$

This is a measure of the mean exponential rate of divergence of trajectories. Visually, this represents the slopes of the diverging error in Figure 6.2. Wolfram claims that for rule 30 CA systems, typically the system expands at a rate of $\lambda_{left} = 0.25$ and $\lambda_{right} = 1$ [8]. This means that on average, for each line in a CA, the avalanche will grow 1 cell to the right and 0.25 cells to the left. Although λ_{right} was found to be a constant throughout our testing, λ_{left} seemed to vary greatly between trials. For the purposes of estimating necessary start-up bits to be thrown away, we will assume that in the worst case scenario, $\lambda_{left} = 0$ and $\lambda_{right} = 1$. In this situation, an N-cell CA should discard the first N lines of its evolution to allow the avalanche effect to fully take hold. This corresponds to our recommendation that the first N^2 bits of the evolution of a CA are discarded at start-up.

6.3 Session Length

CA are ultimately periodic systems. This property could become vulnerable to statistical attack if the transfer session being encrypted is longer than the period of the CA. The period length of an N-cell CA, evolved by rule 30 (equation 3.15), was found by Wolfram[9] to approximate

$$L = 2^{0.63N} \tag{6.2}$$

where L is the number of lines until the system repeats. For systems with a larger cell size (and corresponding initial condition) one expects a longer period. To form a direct comparison between our system and 128-bit WEP, consider a CA of equivalent keysize, that is, a 128-cell CA (i.e. $N = 128$). Additionally assume a shrinking generator scheme in which half the bits are discarded. The amount of data bits L_{CA} that may be encrypted by the key before the keystream is likely to repeat itself is expressed by

$$L_{CA} \;=\; \frac{1}{2} \cdot N \cdot 2^{0.63N} \;\approx\; 10^{26}\, bits, \qquad N = 128 \tag{6.3}$$

Thus for a 128 bit CA cryptosystem, we can safely encrypt about 10^{26} bits, or about 10^{25} bytes until the key requires changing. To compare the session length of our cryptosystem against WEP's, we first must come up with a reasonable estimate. It was proposed that given new methodologies, an effective attack against WEP could be made on an average computer after collecting about 5,000,000 packets of data from a target network [3]. Conservatively estimating the average packet size to be 500 bytes on a given network, we find the corresponding session length, L_{WEP}, to be about 10^9 bytes. Comparing these two estimates

$$\frac{L_{CA}}{L_{WEP}} \;=\; \frac{10^{25}}{10^{9}} \;=\; 10^{16} \tag{6.4}$$

we see that the see that for a CA cryptosystem with an equivalent keysize to WEP, the session length is about 10^{16} times longer.

6.4 Encryption Rates

As indicated in Section 5.2, the hardware implementation of our cryptosystem is capable of encrypting 1 bit of data per clock cycle. This is a fundamental limit for serial computation. If higher encryption rates are required, parallel computation could be induced by implementing several systems on one chip – the implementation in B.2 requires only 2% of a 10 kb FPGA. Alternatively, a RAM lookup table could be implemented for each cell. This would allow the entire ring register to update every clock-cycle, yielding N bits per cycle. In either case, two remarks require mentioning. The shrinking generator will discard about 50% of the bits, effectively cutting practical throughput in half. Secondly, the output will be a block of bits which will require a parallel-to-serial conversion for single channel communication.

For the scope of this project, we are interested in serial ciphering and thus will consider encrypting only one bit per clock cycle. The CMOS implementation in the previous chapter indicates that clock speeds of 500 Mz are realizable. This gives a theoretical encryption rate of 500 megabits per second, however employing a shrinking generator may reduce it to 250 Mbits/s. A recommendation will be given on how to avoid this penalty in the following chapter.

Standard	Max. Transfer Rate	Data Throughput
IEEE 802.11b	11 Mbits/s	5.5 Mbits/s
IEEE 802.11g	54 Mbits/s	24.7 Mbits/s

Table 6.2: Wireless Network Transfer Rates

Current wireless standards [17] are shown in Table 6.2. Data throughput, as distinguished from transfer rates, does not include the packaging which encapsulates presentation layer data. As indicated in Section 4.5, this represents the real-time deadline of our system. With dedicated hardware, it has been shown that our system is capable of meeting this demand and is in fact an order of magnitude faster.

Chapter 7

Conclusions and Recommendations

7.1 Conclusions

The goal of this project was to create a secure, real-time cryptosystem. Statistical analysis was employed using the FIPS 140-1 tests for evaluating the cryptographic randomness of our cipher, and the high pass rate indicates a secure cipher. Our cipher also shows a good avalanche effect after N^2 bits which demonstrates that a keysearch attack will not diverge to a solution. With the use of a shrinking generator, our system is resistant to the only known attack on CA-based stream ciphers. Additionally, our system is more secure than WEP with a 10^{16} times longer period for an equivalent keysize. It may be deduced from these results that the prerogative of security was actualized. Furthermore, a hardware implementation of our system can realize encryption rates 10 times faster than IEEE 802.11g data throughput, demonstrating that our system meets the real-time demands of wireless networking without introducing latencies. Therefore, in conclusion, our system exhibits all the properties desired in the problem statement. We submit that this technology shows considerable promise and should be considered by industry for product development. Recommendations for evolving our proof-of-concept into a marketable solution, as well as cautions of potential difficulties with this endeavor, will now be offered.

7.2 Recommendations for Future Work

7.2.1 Shrinking Generator Acceleration

It has been mentioned that the shrinking generator was not implemented in hardware. Dropping bits from the keystream could be achieved by routing the output through a transmission gate, however it would also create aperiodic output. This is still a realizable solution but it would require active intervention from its host device to keep the transfer synchronized. We have come up with an alternative solution to keep this issue internal so the controlling device may assume a more passive role. It was infeasible due to time constants for us to implement, but we offer it as a recommendation for future work.

In section 6.1.4, parallel computing methods were discussed. In particular, the concept of having several lookup tables is utilizable here. By producing several bits in parallel every cycle, we can essentially create a buffer. When a bit is dropped, several subsequent bits have already been calculated and are ready for immediate output. This method would have to be complemented by alterations to the ring register to allow it to rotate by more than one position per cycle. The size of the buffer is a topic for further investigation and statistical analysis.

7.2.2 Full VLSI Layout

As mentioned in section 5.4, a full ASIC implementation was beyond the scope of this project. However, it has been our intention from the beginning that the entire system be implemented in VLSI. An FPGA implementation offers the same functionality as an ASIC but are generally much slower and consume more power. For this reason, a recommendation for a full VLSI implementation is offered. Our proof-of-concept chip layout, in appendix 3, was developed in CMOS but consideration may be given to BiCMOS or domino logic to further speed gains.

7.2.3 Key Distribution

For the purposes of this project, it has been assumed that the sender and receiver have an *a priori* knowledge of the private key. In practice, a secure method of negotiation is required to exchange this key. Historically, this was done in person. Today, public key encryption standards like RSA, described in section 3.3.1 and in [18], offer a secure channel for key exchange. As mentioned, these systems are generally slower than stream ciphers, so routing the entire message this way is inadvisable. However, they are sufficient for small packets of data like keys.

What is left for further research is a system by which these keys may be distributed. It could be the prerogative of a central distributor, or alternatively the sender could generate the key and send it securely to the receiver prior to an encrypted session. The key should be determined by a *true* random number generator, and it should be verified before usage, since certain initial conditions do not result in insecure behaviour. It is also recommended that the key be periodically refreshed to ensure it never extends beyond its period length (see section 6.3.1).

7.2.4 Hybrid CA Synthesis

In section 4.3, an overview was given of past initiatives to destroy the inherent partial-linearity of Wolfram's system, to counter the Meier-Staffelbach attack. Solutions in literature revolve around the use of hybrid CA. We proposed using a shrinking generator instead, however this solution is not mutually exclusive. Further study into a synthesis of our ideas with the systems discussed in section 3.3.3 is recommended. This endeavour would likely sacrifice the speed of our system for greater complexity, but such a system may have applications in other fields of cryptography where real-time deadlines are not as critical.

7.2.5 Pre-Cipher Compression

An important concept in Shannon's model of cryptography[4] is a quantity known as the unicity distance. In simple terms, it represents a threshold in the amount of a ciphertext needed to be intercepted for a successful brute force attack, given unlimited time and computational power. For binary stream ciphering of keysize N, the unicity distance may be approximated by,

$$L_0 \approx \frac{N}{r} \tag{7.1}$$

The term r represents the percentage redundancy of a L-bit ciphertext. It is defined as,

$$r = 1 - \frac{H(P)}{L \log 2} \tag{7.2}$$

An increase in entropy of the plaintext, $H(P)$, would result in a decrease of redundancy and an increase of the unicity distance. To exploit this principal for our advantage, data compression could be employed on the plaintext before ciphering to increase its entropy, making the system more secure against brute force attacks. The design of a real-time data compression scheme is left for future work in this project. We especially encourage consideration be given to CA-based compaction methods, as outlined in [9].

7.2.6 Physical Cryptanalysis

The scope of this project, in relation to security, was to prevent statistical cryptanalysis. However, as cryptoaccelerators become more prevalent, a range of attacks have been developed to expose weaknesses of the underlining hardware. One method of attack involves the introduction of malicious faults designed to leak the key. This is possible with a reprogrammable device like an FPGA but not with an ASIC. A second method, the so-called side-channel attack, attempts to establish a correlation between the data being operated on and measurements of a cryptosystem's timing, power consumption, or electromagnetic radiation. It is recommended that precautions be taken against physical attacks in future work on this project.

Appendix A

Program Source Code in Java

A.1 Key - Key.java

```
import java.io.*;
import java.lang.Integer.*;
import java.util.*;

public class Key {

    private BitSet initCond;
    private int rule;

    public Key() {
        String prng = ”011000100111001101001010011001
01010100011100001010101001110111100110001100110100100
00011001101011101100101100000111000011000101100101100
11111111011001001010011101100101101010111100101100111
10010111100000100111010110001010110111010010101001001
10101001011110000100000111000001111111000000111010100
01011111110110100000110010100001001110001010111010011
01100111000001011101000010110011100111001111110000100
11001101101011101110010110001001110001100000000011110
01111101111011011000000110111101111111000011011111000
11110111010101101101110001011100001010100010011010001
10011000110011011110000010110000110000101010100100110
00011010101001010001110010010010001111111010011011101
10010100100011101100000010001010011000101111100001000
0
10000010110001101100001001000100101111010100000001010
11100001011000110010000001001011101001111000001011110
1
11001101100111011011100101010010101011000001001101110
10000111100111010101111000111011011111110110001010100
01111100011111010000100111110111000110011010010010100
0100000111000110110101110000100011100011”;
        initCond = new BitSet(1024);
        for (int i=0; i<1024; i++) {
            if (prng.charAt(i)==’1’)
                initCond.set(i);
        }
        rule = 30;
    }

    public int getRule(){
        return rule;
    }

    public BitSet getInitCond(){
        return initCond;
    }
}
```

A.2 Cellular Automata - CellAuto.java

```
import java.io.*;
import java.util.*;
import java.math.*;

public class CellAuto{

    protected int cells;
    protected int neighb;
    protected BitSet ruleIndex;

    public BitSet currentLine;
    public BitSet nextLine;

    protected int maxCellOffset;
    protected int currentBitIndex;

    public CellAuto (int rule, int cells, int neighb, BitSet initCond){
        this.cells = cells;
        this.neighb = neighb;
        ruleIndex = decToBin(rule, (int)Math.pow(2, neighb));
        currentLine = (BitSet)initCond.clone();
        nextLine = new BitSet(cells);
        currentBitIndex = 0;
        maxCellOffset = (int)Math.floor(neighb/2);
    }

    public CellAuto (BitSet rule, int cells, int neighb, BitSet initCond){
        this.cells = cells;
        this.neighb = neighb;
        ruleIndex = (BitSet)rule.clone();
        currentLine = (BitSet)initCond.clone();
        nextLine = new BitSet(cells);
        currentBitIndex = 0;
        maxCellOffset = (int)Math.floor(neighb/2);
    }

    private BitSet decToBin (int dec, int size){
        BitSet bin = new BitSet(size);
        bin.clear();
        for (int i = size; i>=0; i--){
            if ((dec>=Math.pow(2, i))) {
                dec = dec - (int)Math.pow(2, i);
                bin.set(i);
            }
        }
        return bin;
    }

    private int bitsToDec (int currentCell){
        int dec = 0;
        BitSet bitConfig = new BitSet(neighb);

        if (currentCell - maxCellOffset < 0){
            int spillOver = maxCellOffset - currentCell;

            for(int i = 0; i<spillOver; i++){
                if(currentLine.get(cells - spillOver + i))
                    dec = dec + (int)Math.pow(2, i);
            }

            for(int i = spillOver; i<neighb; i++){
                if(currentLine.get(currentCell - maxCellOffset + i))
                    dec = dec + (int)Math.pow(2, i);
            }
        }

        else if ((currentCell + maxCellOffset) >= cells){
            int spillOver = currentCell + maxCellOffset - cells + 1;

            for(int i = 0; i<neighb-spillOver; i++){
                if(currentLine.get(currentCell - maxCellOffset + i))
                    dec = dec + (int)Math.pow(2, i);
            }

            for(int i = (neighb-spillOver); i<neighb; i++){
                if(currentLine.get(i - neighb + spillOver))
                    dec = dec + (int)Math.pow(2, i);
            }
        }

        else{
            for(int i = 0; i<neighb; i++){
                if(currentLine.get(currentCell - maxCellOffset + i))
                    dec = dec + (int)Math.pow(2, i);
            }
        }

        return dec;
    }

    public BitSet generateLine(){
        for (int i=0; i<cells; i++){
            nextLine.set(i, ruleIndex.get(bitsToDec(i)));
        }
        currentLine = (BitSet)nextLine.clone();
        return currentLine;
    }

    public boolean generateBit(){

        if (currentBitIndex < cells-1){
            currentBitIndex++;
            return currentLine.get(currentBitIndex-1);
        }
        else{
            boolean temp = currentLine.get(cells-1);
            currentBitIndex = 0;
            generateLine();
            return temp;
        }
    }

    public int generateByte(){
        int dump = 0;
        if (generateBit())
```

```
            dump = dump | 0x80;
        if (generateBit())
            dump = dump | 0x40;
        if (generateBit())
            dump = dump | 0x20;
        if (generateBit())
            dump = dump | 0x10;
        if (generateBit())
            dump = dump | 0x08;
        if (generateBit())
            dump = dump | 0x04;
        if (generateBit())
            dump = dump | 0x02;
        if (generateBit())
            dump = dump | 0x01;
        return dump;
        }
}
```

A.3 Sender - Sender.java

```
import java.io.*;
import java.lang.Integer.*;
import java.util.*;
import java.net.*;

public class Sender {
    public static void main(String[] args)
        throws java.io.IOException {
        DataInputStream input = new DataInputStream(System.in);
        Socket TCPConnect = new Socket(”sun20.eng.uwo.ca”,6789);
        DataOutputStream output = new DataOutputStream(TCPConnect
           .getOutputStream());
        Key key = new Key();
        CellAuto mask = new CellAuto(key.getRule(),1024,3,key.
           getInitCond());
        int currentByte;
        while((currentByte = input.read()) != -1){
            int plainByte = currentByte;
            int maskByte = mask.generateByte();
            int cipherByte = plainByte ^ maskByte;
            output.write(cipherByte);
        }
        TCPConnect.close();
        output.close();
        input.close();
    } //main class
} //Sender.java
```

A.4 Receiver - Receiver.java

```
import java.io.*;
import java.lang.Integer.*;
import java.util.*;
import java.net.*;

public class Receiver {
    public static void main(String[] args)
        throws java.io.IOException {
        ServerSocket TCPConnect = new ServerSocket(6789);
        int currentByte;
        int counter = 1;

        while(true) {
            Socket connection = TCPConnect.accept();
            System.out.println("\nConnection #" + counter + ":");
            System.out.println("\t" + connection.toString());
            DataInputStream input = new DataInputStream(
                connection.getInputStream());
            DataOutputStream output = new DataOutputStream(new
                FileOutputStream("file" + counter));
            Key key = new Key();
            CellAuto mask = new CellAuto(key.getRule(),1024,3,key
                .getInitCond());
            while((currentByte = input.read()) != -1){
                int cipherByte = currentByte;
                int maskByte = mask.generateByte();
                int plainByte = cipherByte ^ maskByte;
                output.write(plainByte);
            }
            output.close();
            input.close();
            System.out.println("\tData written to file"+counter);
            System.out.println("Connection Closed.");
            counter ++;
        }
    } //main class
} //Sender.java
```

A.5 Cellular Automata Pseudo-Random Number Generator - CAprng.java

```java
// Cellular Automata Pseudo-Random Number Generator
// Program used to generate the evolution of a cellular automata in the form of byte data.
// The data files output by this program are directly used by the FIPS 140-1 test suite.
// Usage: java CAprng <rule> <cells> <neighbourhood> <bytes> <outputFileName>
// EXAMPLE: java CAprng 30 1024 3 10000 output1
// Generates rule 30, 1024 cell, 3 neighbourhood, 10000 byte bit stream sent to file output1

import java.io.*;
import java.util.*;
import java.math.*;

public class CAprng {

        public static void main(String[] args) throws java.io.IOException{

                int rule    = Integer.parseInt(args[0]);
                int cells   = Integer.parseInt(args[1]);
                int neighb  = Integer.parseInt(args[2]);

                DataOutputStream out = new DataOutputStream(new FileOutputStream(args[4]));
                BitSet initCond = new BitSet(cells);

                for (int i = 0; i<cells; i++){
                        if(Math.random()<0.5)
                        initCond.set(i);
                        }
                cellularAutomata doCA = new cellularAutomata(rule, cells, neighb, initCond);

                for(int k=0;k<Integer.parseInt(args[3])+1;k++){
                        out.write(doCA.generateByte());
                        }
                out.close();
        }
}
```

Appendix B

Circuit Design and Timing Diagrams

B.1 Cellular Automata

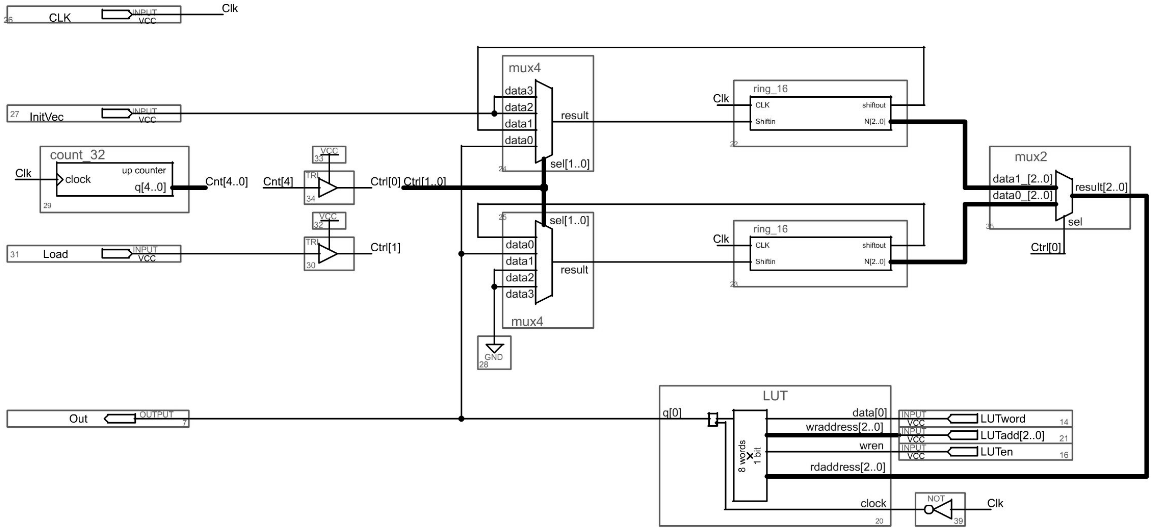


Figure B.1: Cellular Automata Circuit Diagram

B.2 Stream Cipher

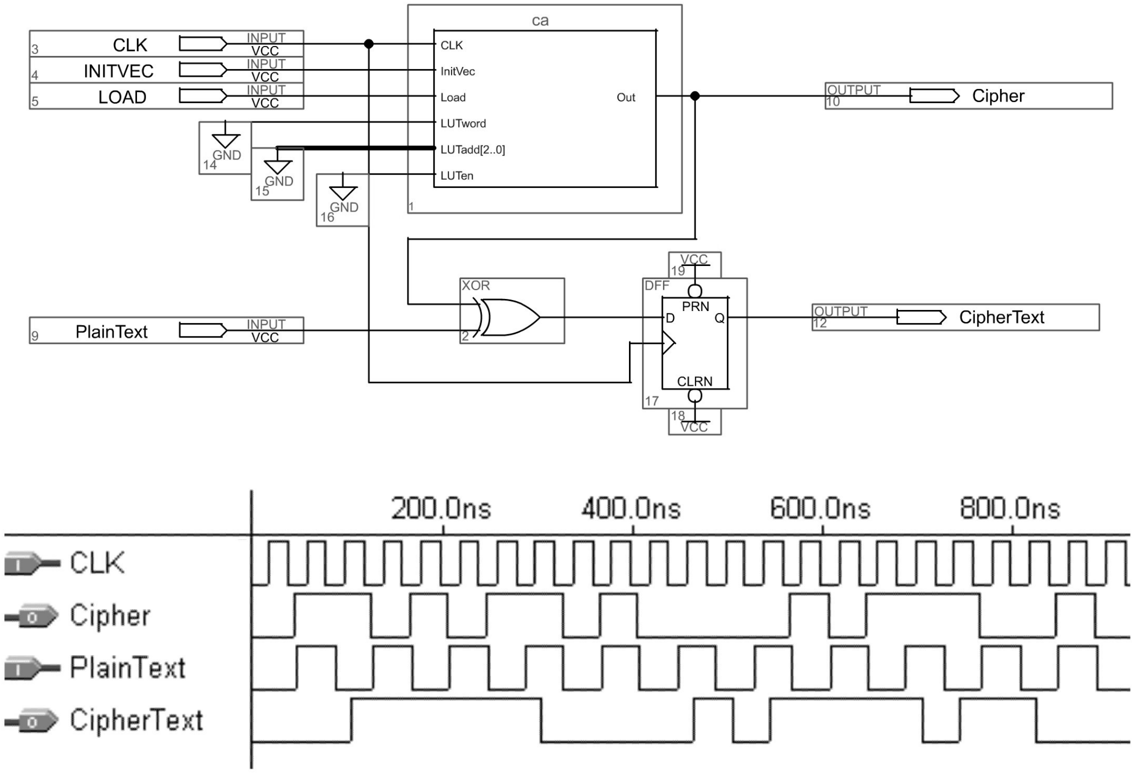
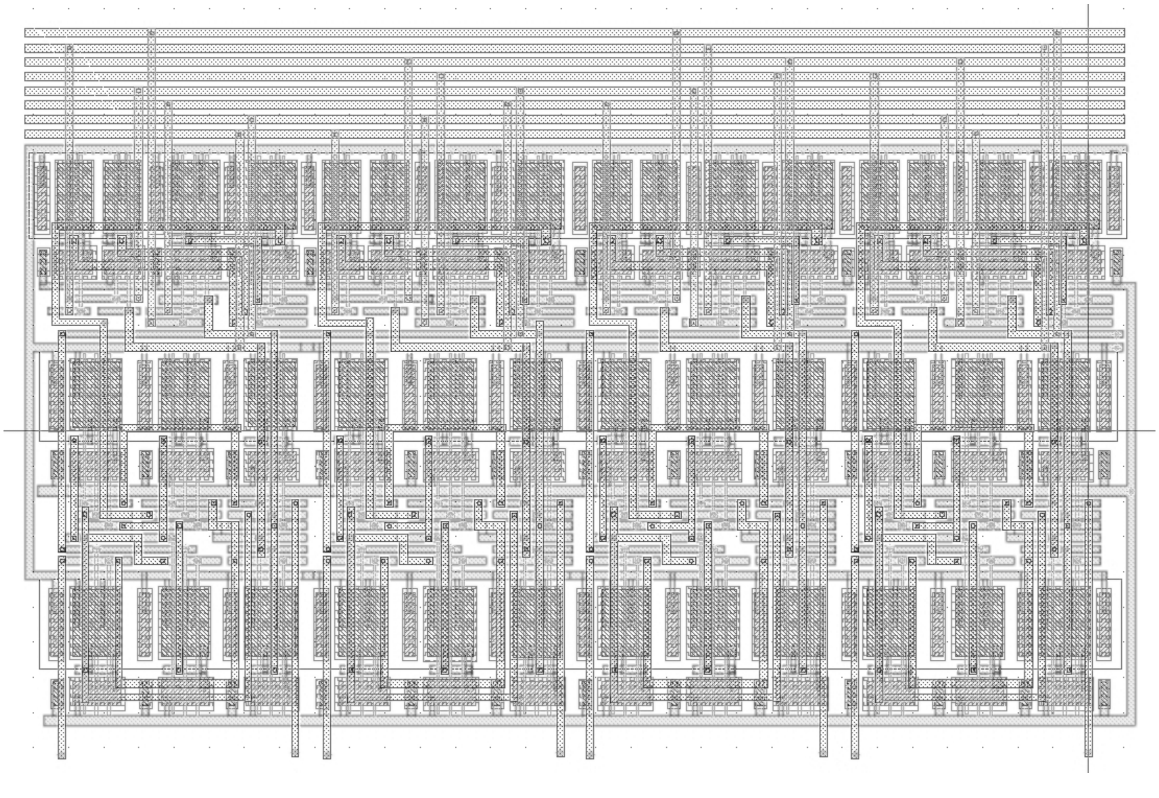


Figure B.2: Stream Cipher Circuit and Timing Diagram

Appendix C

Circuit Layout in $0.35\mu m$ CMOS

C.1 4-Cell CA (Rule 30)



Bibliography

[1] Rsa from wikipedia. *http://en.wikipedia.org/wiki/RSA*.

[2] J. Daemen and V. Rijmen. Aes proposal: Rijndael. *AES Algorithm Submission*, 1999. Online: *http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf*.

[3] A. Stubblefield, J. Ioannidis, and A. Rubin. Using the fluhrer, mantin, shamir attack to break wep. Technical Report TD-4ZCPZZ, Rev. 2, AT-T Labs, 2001.

[4] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[5] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, New York, NY, 1997.

[6] Weiguang Yao. Improving security of communication using chaoitc synchronization. page 38. Doctoral Thesis, The University of Western Ontario, 2002.

[7] Stephen Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics*, 7(2):123–169, 1986.

[8] Stephen Wolfram. Cryptography with cellular automata. In H. C. Williams, editor, *Advances in Cryptology - CRYPTO 85, Proceedings*, pages 429–432. Springer Verlag, LNCS Nr. 218, 1986.

[9] Stephen Wolfram. *A New Kind of Science.* Wolfram Media, Champaign, IL, 2002.

[10] W. Meier and O. Staffelbach. Analysis of pseudo random sequences generated by cellular automata. In *Advances in Cryptology - EUROCRYPT 91, Proceedings*, pages 186–189. Springer-Verlag, LNCS Nr. 547, 1992.

[11] S. Nandi, B. K. Kar, and P. Pal Chaudhuri. Theory and applications of cellular automata in cryptography. *IEEE Transactions on Computers*, 43(12):1346–1357, 1994.

[12] Blackburn, Murphy, and Paterson. Comments on “theory and applications of cellular automata in cryptography”. *IEEETC: IEEE Transactions on Computers*, 46, 1997.

[13] M. Tomassini and M. Perrenoud. Cryptography with cellular automata. *Applied Soft Computing*, 1(2):151–160, 2001.

[14] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, 1973.

[15] D. Coppersmith, H .Krawczyk, and Y. Mansour. The shrinking generator. In *Advances in Cryptology - CRYPTO '93, Proceedings*, pages 22–39. Springer-Verlag, LNCS Nr. 773, 1994.

[16] Mohaned Rafiquzzaman. *Microprocessors and Microcomputer-Based System Design*. CRC Press, Boca Raton, second edition, 1995.

[17] Ieee 802.11g white paper. Technical Report 802.11g-WP104-R, Broadcom, 2003.

[18] M. Hellman. An overview of public key cryptography. *Communications Magazine, IEEE*, 16(6):24–32, 1978.

Vitae

NAME:	Jeremy William Clark
PLACE OF BIRTH:	Hanover, Ontario
YEAR OF BIRTH:	1981
SECONDARY EDUCATION:	John Diefenbaker Secondary School (1995-2000)
HONOURS & AWARDS:	Honourable Mention, Ontario Engineering Competition (2004)
	Dean's Honour List (2001, 2003)
	Ontario Scholar (1996, 1997, 1998, 1999, 2000)

NAME:	Aleksander Edwin Essex
PLACE OF BIRTH:	Toronto, Ontario
YEAR OF BIRTH:	1980
SECONDARY EDUCATION:	Sir Frederick Banting Secondary School (1994-1998)
HONOURS & AWARDS:	UWO Computer Engineering Design Day Award (2004)
	Honourable Mention, Ontario Engineering Competition (2004)
	Ontario Scholar (1998)