Efficient Implementation of "Large" Stream Cipher Systems

Palash Sarkar
Centre for Applied Cryptographic Research
Department of Combinatorics and Optimization
University of Waterloo
200 University Avenue West,
Waterloo, Ontario,
Canada N2L 3G1
e-mail: psarkar@cacr.math.uwaterloo.ca

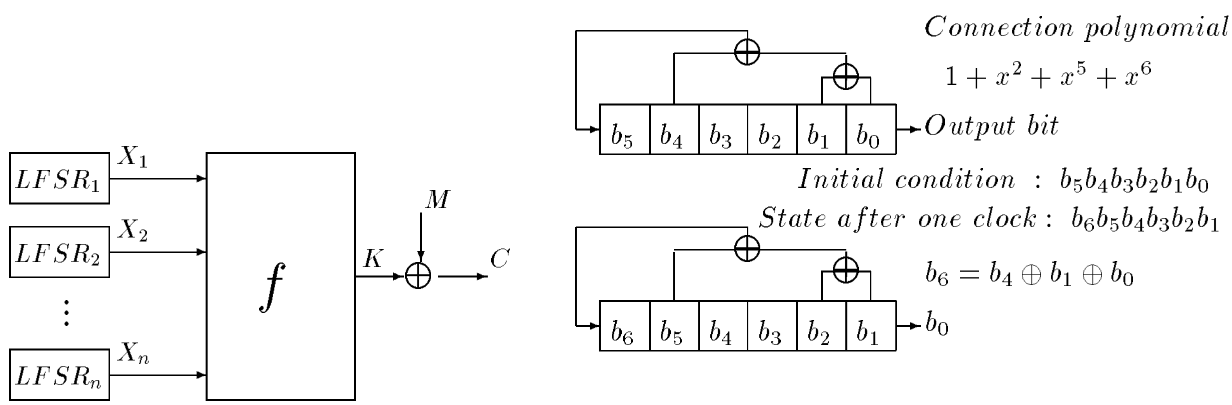
Subhamoy Maitra*
Computer & Statistical Service Centre
Indian Statistical Institute
203, B.T. Road, Calcutta 700 035, INDIA
e-mail: subho@isical.ac.in

Abstract

A standard model of stream cipher combines the outputs of several independent Linear Feedback Shift Register (LFSR) sequences using a nonlinear Boolean function to produce the keystream. Here we present a low cost hardware architecture for such secret-key cryptosystems using a relatively large number of LFSRs. We propose implementation of the LFSRs using Cellular Automata in VLSI. This provides a regular and uniform two dimensional array of flip flops with only local interconnections. The main bottleneck in the implementation of stream ciphers using a relatively large number of LFSRs is the implementation of the combining Boolean function. We show that this bottleneck can be removed and it is feasible to implement "large" cryptographically secure Boolean functions using a reconfigurable pipelined architecture.

Keywords : Stream Ciphers, Boolean functions, Linear Feedback Shift Registers, Cellular Automata, Reconfigurable Hardware, Pipelined Architecture.

*Corresponding author.



(a) LFSR based encryption scheme *(b) LFSR : One step evolution*

Figure 1: Stream Cipher System

1 Introduction

In the most common model of stream ciphers, the outputs of several independent Linear Feedback Shift Registers (LFSRs) are combined using a nonlinear Boolean function (see Figure 1). The initial conditions of the LFSRs constitute the secret key of the system. See [3] for more details about LFSR. In such a system, n bits from the n different LFSR's are generated at each clock. These n bits are provided as n input values to the combining function. That is, the LFSRs provide the input bit streams $X_1, X_2, \ldots, X_n$ to the combining Boolean function f. The output of the combining function is the key stream (K) which is XORed with the message stream (M) to obtain the cipher stream (C).

The combining Boolean functions must possess certain cryptographic properties for the overall system to be secure. Design of proper Boolean function have received a lot of attention in recent times as evidenced by the papers [4, 8, 10, 12]. This has answered many theoretical questions on the design of Boolean functions for stream cipher applications. It is now time to turn to the implementation issues of such Boolean functions and their actual use in stream cipher cryptography.

LFSR based stream cipher systems are generally implemented using a Boolean function on a small number of variables, typically 8 to 10. The main reason being the difficulty in efficiently implementing a Boolean function on a large number of variables (say 20 or more variables). However, if one were to use such a function with properly selected parameters, then none of the currently known attacks would have even a remote chance of success.

The VLSI area used in implementing stream cipher systems have two components.

1. The area used to implement the LFSRs.

2. The area used to implement the Boolean function.

Suppose the system uses an n-input Boolean function and (for simplicity) assume the length of all the LFSRs are same (say L). Then the area used to implement the LFSRs is proportional to $L \times n$ while the area used to implement the Boolean function can be proportional to 2^n. Consequently, while the area required by the LFSRs increase linearly with n, the area required by the Boolean function can be exponential in n. Thus, by increasing the number of inputs to the Boolean function, the main hurdle would be in implementing the Boolean function and not the LFSRs.

Let us compute some real parameters to get a feel of the problem. Suppose a 32-variable combining function is used where the length of the LFSRs is 64 bits long on average (shortly we will discuss why we will not use equal length LFSRs). Then the number of flip-flops required to implement the LFSRs is only 2048, while a direct implementation of the Boolean function can require area proportional to 2^{32}. The key size of such an LFSR system is estimated as follows. The secret key of the system are the initial states of the LFSRs and hence account for $32 \times 64 = 2048$ key bits. While this is a large key, it should be noted that currently RSA systems are also being advocated with 2048 bit keys.

In this paper we tackle the implementation issue of Boolean functions on a large number of variables. (Here we consider a Boolean function on 24 or more variables to be a "large" one.) There is no general purpose implementation method and implementation is dependent on the specific design of the Boolean function. We present an algorithm and hardware description of the recursive construction method presented in [4]. The functions in [4] are built recursively. Thus a function F of n variables is built up from a function h of k variables. It is important to note that, if we use a function h which is optimum with respect to the parameters algebraic degree, order of resiliency and nonlinearity, then the function F is also optimum with respect to these parameters [9]. We describe an algorithm which uses the function h as a black box (an oracle) and computes the output of F on an n-bit input in time linear in $n - k$. The space required by the algorithm is $O(1)$ plus the space required to implement h.

In an LFSR based stream cipher system, an n-bit input is provided to the function at each clock cycle. Thus our algorithm cannot be directly translated into a hardware circuit. Instead we use a regular pipelined architecture to map the algorithm to hardware. The pipeline takes $n - k$ cycles to fill up and after that, it can handle an n-bit input at each clock cycle. There are $n - k$ stages to the pipeline which are all similar to each other providing a uniform design. Implementation of each stage can be done by a circuit or look up table of constant size. The total space required to implement F is the space required to implement h plus an additional $O(n - k)$ size circuit. Usually the number of variables k of the function h will be significantly less than the number of variables n of the function F, space required to implement F is of the same order as the space required to implement h. This makes it feasible to implement functions of 24 or more variables with nominal cost.

An important parameter is the linear complexity of the generated key sequence. To obtain the maximum possible linear complexity of the key sequence, we need to use LFSRs whose lengths are coprime to each other [1]. Thus the LFSRs are going to be of different lengths. A direct implementation of such different length LFSRs is going to produce a very irregular VLSI structure. To obtain a more regular structure, we suggest the use of a uniform two dimensional array of flip flops connected in a suitable fashion. Some of the flip flops in the two dimensional

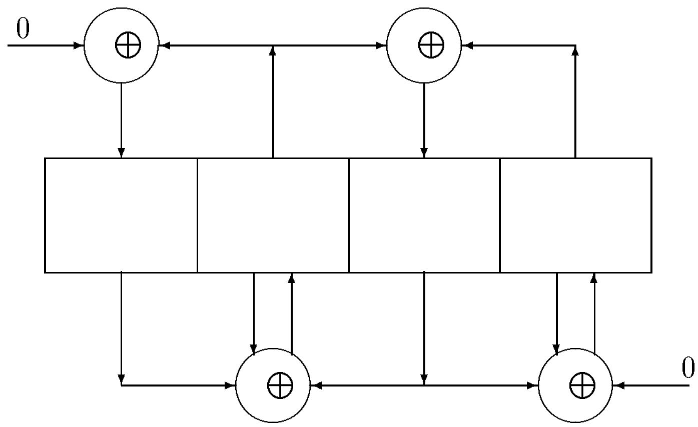


Figure 2: A $< 90, 150, 90, 150 >$ CA

array will not functional. This is the price to pay for obtaining uniformity in the design.

Now consider implementing the different length LFSRs on this two dimensional structure. Each LFSR must have a large number of tap cells to resist cryptanalytic attacks [15]. Further, each of the LFSRs are going to have a long feedback connection. Thus the overall connection pattern on the two dimensional array is going to be highly irregular. This is also considered to be a disadvantage in VLSI implementation.

Here we suggest the use of cellular automata (CA) to replace the LFSRs. The class of CA we suggest are algebraically equivalent to LFSRs. Hence the security of the system is not affected by this change. The advantage would be that a CA based design would provide a uniform and regular structure with only local interconnections, which is very attractive from VLSI point of view. CA based architectures have been proposed for many traditional LFSR applications (see [2]).

In Section 2, we briefly outline the necessary details of CA required to replace LFSRs in stream cipher cryptography. The cryptographically useful Boolean functions from [4, 9, 6] are described in Section 3. We summarize the main points and gloss over the cryptographic properties since our purpose here is to discuss the implementation of these functions. The actual algorithm and hardware description is presented in Section 4. Finally we conclude with some remarks on future work in Section 5.

2 Cellular Automata

A cellular automaton is a finite array of cells, where each cell can store a bit of information. The collection of values of the cells constitute the global state of the CA, whereas the state of a cell is called its local state. The CA evolves globally in discrete time steps, with the state of each cell changing at each time step. The change is affected by the values of the two neighbouring cells and also optionally itself. This is pictorially depicted in Figure 2. The cell at the left end

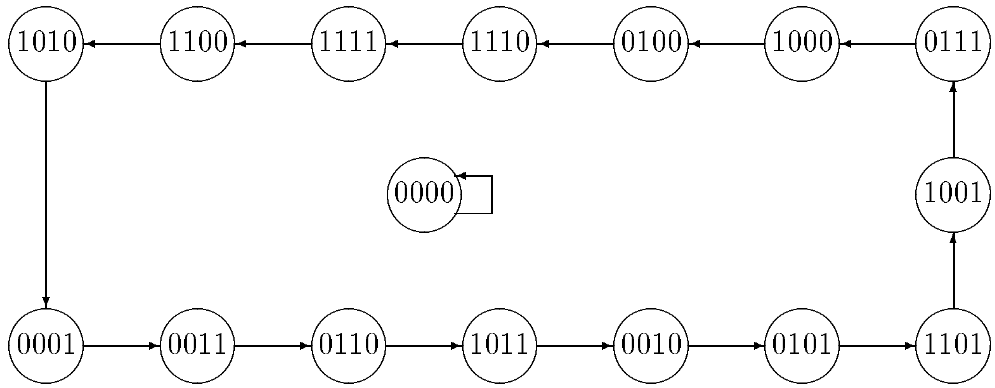


Figure 3: STD for the CA in Figure 2.

does not have a left neighbour and one at the right end does not have a right neighbour. If the next state of a cell depends on its two neighbours and itself, then the cell is said to follow rule 150. If the next state of the cell depends only on its two neighbours and not on itself then it is said to follow rule 90. (See [16] for an explanation and nomenclature of CA rules). A CA having cells which use only rules 90 and 150 is called a 90/150 CA. In the rest of the discussion we will be interested in only 90/150 CA. The next state evolution of a CA can be totally described by a tridiagonal matrix as follows. Consider a 4-cell CA with rules $< 90, 150, 90, 150 >$ (see Figure 2). If the current state is (x_0, x_1, x_2, x_3), then the next state (y_0, y_1, y_2, y_3) is given by

$$(x_0, x_1, x_2, x_3)\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} = (y_0, y_1, y_2, y_3).$$

Thus starting from an initial configuration, the CA evolves in discrete time state under the action of the state transition matrix. See Figure 3 for the next state behaviour of the $< 90, 150, 90, 150 >$ CA. The initial configuration is loaded in parallel into the CA cells. In our setup this initial configuration is the secret key for the CA being used. The output of the CA can be taken as the output of any particular cell of the CA. The sequence generated by any cell is same as any other cell except for a circular shift in the sequence. Note that unlike the LFSRs the amount of shift between two consecutive cells may be more than 1.

The tridiagonal matrix which governs the behaviour of the CA is called the state transition matrix. It is known [2] that *if the characteristic polynomial of this matrix is primitive over $GF(2)$ then an n-cell CA will cycle through all the possible $2^n - 1$ non null states.* The characteristic polynomial of the $< 90, 150, 90, 150 >$ CA is $x^4 + x + 1$, which is primitive over $GF(2)$ and hence the CA cycles through all the non zero states as shown in Figure 3. The output sequence of the CA is completely determined by the characteristic polynomial of the state transition matrix. This is the basis for replacing LFSRs by CA.

Given a primitive polynomial it is easy to design an LFSR which has this polynomial as its connection polynomial. On the other hand the design method for CA is not straightforward. One approach is to form the companion matrix and then use the Lanczos tridiagonalization over $GF(2)$. This approach has been carried out in [11]. However, a simpler and a more elegant algorithm has been presented by Tezuka and Fushimi [13]. The matter that interests us here is the fact that *given any primitive polynomial it is possible to design a CA whose state transition matrix has this primitive polynomial as its characteristic polynomial.*

From this discussion it is clear that the use of CA does not alter the stream cipher system in any essential way and hence the security of the system remains unaltered. The only advantage to be gained is the simplicity in VLSI implementation. The use of CA over LFSR has been suggested for several advantages in VLSI design. The local connection structure of CA makes it a regular and cascadable architecture. On the other hand, the long feedback connection of LFSRs introduce delays and is also undesirable from a VLSI layout point of view (see [2]). Also see [7] for a survey on CA.

2.1 CA Based Implementation

As mentioned in the Section 1, the main difficulties in the implementation of the LFSRs are the following.

1. To have the maximum linear complexity, the LFSRs lengths need to be pairwise relatively prime. A direct implementation would have to use registers of different lengths resulting in a non uniform structure.

2. The connection pattern for an LFSR is highly irregular. The tap points of an LFSR are in general not regularly placed. In addition, the number of taps in the LFSR must be high to resist against certain types of attacks [15]. Further, an LFSR has a long feedback connection and the length of this feedback connection can be equal to the length of the LFSR.

Thus a direct implementation of the LFSRs leads to an irregular and non uniform design. This is considered to be a distinct disadvantage in VLSI implementation. We discuss how the above problems can be tackled.

To tackle the first problem, we suggest the use of a two dimensional array $n \times L$ of flip flops, where n is the number of inputs to the Boolean function and L is the maximum degree of a connection polynomial (say 128). In each row of this structure, the connection pattern for a single polynomial is implemented. Thus in each row, some of the flip flops will not be functional. The cost incurred due to this would be offset by the design advantage in using a uniform two dimensional array.

The solution to the second problem is to use 90/150 CA to implement the LFSRs. Corresponding to a primitive polynomial we will be able to get the corresponding 90/150 CA using the algorithm provided in [13]. Each cell in such a CA will be connected to its left and right neighbours. Further if the rule for the cell is 150, then it will also be connected to itself. Thus all connections are local and regular. Also the long feedback connection of the LFSR is eliminated.

In the two dimensional array of flip flops, for each row, the number of flip flops used is equal to the length of the corresponding CA. The outputs of all the CA are taken in a bit slice manner from one end (say the right end) of the two dimensional array. In this case the non functional flip flops will be towards the left end. Thus the overall design is a two dimensional array of flip flops with only local connections and the output is provided in a bit slice manner by the rightmost column of the two dimensional array. Such a structure will be simple to implement in VLSI and will also provide easy reconfigurability using standard structures like FPGA.

3 Cryptographically Useful Boolean Functions

We present a brief overview of the various cryptographic properties that a Boolean function must satisfy in order to be used for stream cipher systems. Since our purpose in this paper is implementation, we briefly mention the properties. For more detailed definitions we refer to [4, 8].

An n-variable function is said to be *balanced* if the output column of its truth table has equal number of zeros as ones. It is said to be *m-resilient*, if the probability of the output being one is half even if atmost m of the inputs are fixed to constant values. The algebraic normal form of a Boolean function is its canonical sum of products representation as a multivariate polynomial over $GF(2)$. The degree of the polynomial is called the *algebraic degree* or simply degree of the function. Functions of degree atmost one are called affine functions. Given a Boolean function, its *nonlinearity* is its Hamming distance to the set of affine function, i.e., its Hamming distance to its best affine approximation.

A method of designing cryptographically useful Boolean functions is to start from an initial good function and recursively build up the desired function. Several such recursive methods have been proposed [4, 12]. The method proposed in [4] is simple, though it does not always result in the best function. The reason being the use of an unbalanced, highly nonlinear initial function which was not optimized with respect to nonlinearity, algebraic degree and order of resiliency. However, for a suitable initial function, the method of [4] produces optimized functions. These initial functions have to belong to one of the *saturated sequences* discussed in [9]. For example, if we use a 7-variable, 2-resilient, degree 4, nonlinearity 56 function [6], then the resulting sequence of functions constructed using the method of [4] are the best possible with respect to nonlinearity, algebraic degree and order of resiliency. Thus we restrict ourselves to implementation of the recursive method of [4], noting that the initial function h must be an optimized function with respect to nonlinearity, algebraic degree and order of resiliency.

Suppose an n-variable function $F(X_n, \ldots, X_1)$ is to be used in the stream cipher system. Following the method of [4], this F is represented by a sequence $(h, S_1, \ldots, S_t)$, where h is the initial function of k variables $X_k, \ldots, X_1$ and S_i's are the recursive operators used to build up the function F. Each $S_i \in \{Q, R\} \times \{r, c, rc\}$, where the action of S_i is described as follows. Let $F_0 = h$ and F_i be the function produced after application of S_i. Suppose $S_i = (\Psi_i, \tau_i)$, where $\Psi_i \in \{Q, R\}$ and $\tau_i \in \{r, c, rc\}$.
If $\Psi_i = Q$ then,

$$\begin{aligned} &F_i(X_{i+k}, X_{i+k-1}, \ldots, X_{k+1}, X_k, \ldots, X_1) \\ &\qquad = (1 \oplus X_{i+k}) F_{i-1}(X_{i+k-1}, \ldots, X_{k+1}, X_k, \ldots, X_1) \end{aligned}$$

$$\oplus X_{i+k}(a \oplus F_{i-1}(b \oplus X_{i+k-1}, \ldots, b \oplus X_{k+1}, b \oplus X_k, \ldots, b \oplus X_1)).$$

If $\Psi_i = R$, then

$$\begin{aligned} & F_i(X_{i+k}, X_{i+k-1}, \ldots, X_{k+1}, X_k, \ldots, X_1) \\ &= (1 \oplus X_{i+k-1})F_{i-1}(X_{i+k}, X_{i+k-2} \ldots, X_{k+1}, X_k, \ldots, X_1) \\ &\quad \oplus X_{i+k-1}(a \oplus F_{i-1}(b \oplus X_{i+k}, b \oplus X_{i+k-2} \ldots, b \oplus X_{k+1}, b \oplus X_k, \ldots, b \oplus X_1)). \end{aligned}$$

The value of τ_i determine the values of a and b in the following manner. If $\tau_i = r$, then $a = 0, b = 1$. If $\tau_i = c$, then $a = 1, b = 0$ and if $\tau_i = rc$, then $a = b = 1$.

It is important to note that at each step either $\tau_i \in \{r, c\}$ or $\tau_i \in \{rc, c\}$. This is required to increase the order of resiliency by 1 at each step (see [4]). The actual set of possible values for τ_i is determined recursively as follows. If the order of resiliency of h is even then $\tau_1 \in \{r, c\}$, else $\tau_1 \in \{rc, c\}$. In general, if the order of resiliency of F_{i-1} is even then $\tau_i \in \{r, c\}$, else $\tau_i \in \{c, rc\}$.

In this paper, we will solely be concerned with the implementation of F as represented by the sequence $(h, S_1, \ldots, S_t)$. For cryptographic properties we refer the reader to [4, 9]. Note that $n = k + t$, and $F = F_t$. If h has the order of resiliency m_1, then F has the order of resiliency $m = m_1 + t$. The algebraic degree of F and h are same and the nonlinearity of F is 2^t times the nonlinearity of h.

4 Boolean Function Implementation

In this section we provide algorithms and hardware for resilient functions on large number of input variables. The algorithm we present needs one step for initialization and then t steps in loop to generate the output. For LFSR based stream ciphers, the LFSRs output one bit at each clock and hence an n-bit input is presented to the non linear combining function at each clock. Thus an algorithm which takes more that one clock cycle to compute the output of the Boolean function will introduce delays into the system leading to a degradation of performance. We solve this problem by using a pipelined architecture to map the algorithm to hardware. The pipeline takes t clock cycles to fill up and from then on provides a bit of output at each clock cycle. Thus the total delay for obtaining all the key bits is t clock cycles instead of a delay of t clock cycles for each key bit. Thus the pipeline ensures that there is no effective degradation in the performance of the system.

4.1 Algorithm

We present an algorithm to compute the output of a function F on an n-bit input $(X_n, \ldots, X_{k+1}, X_k, \ldots, X_1)$. The function F is represented by $(h(X_k, \ldots, X_1), S_1, \ldots, S_{n-k})$, where h is presented as a black box and can be implemented either by a combinational circuit or by a look up table. The algorithm requires both time and space linear in m.

Let F be represented by $(h, S_1, \ldots, S_{n-k})$, where h is a function of k variables. Define $F_0 = h$ and F_i to be a function represented by $(h, S_1, \ldots, S_i)$. Then $F_{n-k} = F$. We will refer to the recursive definition of F_i provided in Section 3. First we present an inefficient but obvious algorithm to compute $F_t = F_{n-k} = F(X_n, \ldots, X_1)$ based on the recursive definition in Section 4.

$recCompute(F_i(X_{i+k}, \ldots, X_1))$

1. if $(i = 0)$ return $h(X_k, \ldots, X_1)$.
2. if $(\Psi_i = Q)$ $\{X = X_{i+k}.\}$
3. else $\{X = X_{i+k-1}; X_{i+k-1} = X_{i+k}\}$.
4. if $(X = 0)$ return $recCompute(F_{i-1}(X_{i+k-1}, \ldots, X_1))$.
5. else
6. if $(\tau_i = c)$ return $1 \oplus recCompute(F_{i-1}(X_{i+k-1}, \ldots, X_1))$.
7. if $(\tau_i = c)$ return $recCompute(F_{i-1}(1 \oplus X_{i+k-1}, \ldots, 1 \oplus X_1)$.
8. if $(\tau_i = rc)$ return $1 \oplus recCompute(1 \oplus F_{i-1}(X_{i+k-1}, \ldots, 1 \oplus X_1)$.
9. end if.

end.

Steps 2 and 3 of the above algorithm interchanges the variables X_{i+k} and X_{i+k-1} if $\Psi_i = R$. The rest of the algorithm works according to the recursive definition of F_i. Note that the recursive approach is top down, i.e., it starts processing the variable X_n first and then descends to lower numbered variables. It is easy to see that the algorithm takes time $O(t)$. However, the stack depth of the algorithm is also $O(t)$, which is undesirable. Hence we map it to an iterative algorithm. There are a few key observations to do this.

1. There is no need to carry the variables $X_{k-1}, \ldots, X_1$ through the algorithm. If $\Psi_1 = Q$, then let $Y = X_k$ else $Y = X_{k+1}$. Set $v_0 = h(Y, X_{k-1}, \ldots, X_1)$ and $v_1 = h(1 \oplus Y, 1 \oplus X_{k-1}, \ldots, 1 \oplus X_1)$. Then we will ultimately have to output v_i or $1 \oplus v_i$, depending on the variables $X_n, \ldots, X_k$.

2. At each recursive call, depending on the value of τ_i we either complement the input or the output or both. Thus at each stage it is sufficient to record whether the input/output of the next evaluation has to be complemented. This is kept track of by two bit variables a and b. The variable a records whether the output needs to be complemented and the variable b records whether the input needs to be complemented.

Based on these observations, we next present the algorithm $computeTD(.)$, which converts the recursive algorithm $recCompute()$ to an iterative algorithm.

$computeTD(X_n, \ldots, X_1)$ {
    if $(\Psi_1 = Q)$ then $Y = X_k$;
    if $(\Psi_1 = R)$ then $Y = X_{k+1}$;
    $v_0 = h(Y, X_{k-1}, \ldots, X_1)$; $v_1 = h(1 \oplus Y, 1 \oplus X_{k-1}, \ldots, 1 \oplus X_1)$;
    $a = 0$; $b = 0$;
    for $i = t$ downto 1 do {
        (1*) if $(\Psi_i = Q)$ then $X = X_{i+k}$;
        (2*) if $(\Psi_i = R)$ then { $X = X_{i+k-1}; X_{i+k-1} = X_{i+k}$; }
        if $(b \oplus X = 1)$ then {
            if $(\tau_i = c)$ then $a = a \oplus 1$;
            if $(\tau_i = r)$ then $b = b \oplus 1$;

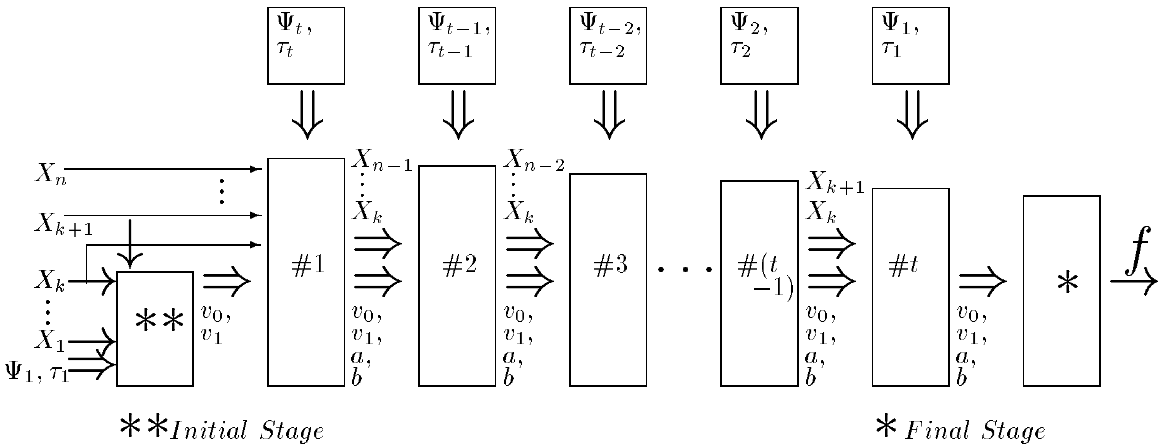


Figure 4: Pipelined Implementation of $computeTD(.)$

```
        if (τ_i = rc) then {a = a ⊕ 1; b = b ⊕ 1; }
      }
   }
   return a ⊕ v_b;
}
```

Based on the previous discussion, we get the following result.

Theorem 4.1 *The algorithm $computeTD(X_n, \ldots, X_1)$ correctly computes $F(X_n, \ldots, X_1)$ in $O(t)$ time.*

4.2 Hardware Implementation of $computeTD(.)$

Here we show how very low cost pipelined hardware can be developed where the output of F on successive tuples of n-bit input is available at each clock pulse after initial t clocks, i.e. starting from $(t+1)$-th clock.

In the hardware description, we will be manipulating Ψ, τ_i as binary values. To do this we need to describe how they will be encoded as bits. If $\Psi_i = Q$, then this is encoded by putting $\Psi_i = 0$. If $\Psi_i = R$, then this is encoded by putting $\Psi_i = 1$. If $\tau_i = c$, then this is always coded by putting $\tau_i = 1$. On the other hand $\tau_i = 0$ codes $\tau_i = r$ or $\tau_i = rc$ according as $i \not\equiv m_1 \bmod 2$ or $m_1 \equiv t \bmod 2$, where m_1 is the order of resiliency of the initial function h (see Section 3).

The pipeline has t stages numbered $\#1$ to $\#t$ (see Figure 4). Stage $\#i$ stores the current values of $X_k, \ldots, X_{n-i+1}$. The two bits v_0 and v_1 are present at each stage along with the two other work bits a and b.

The initial stage (Figure 5a) of the algorithm performs the computation required to get the values v_0, v_1. For this the function h needs to be evaluated twice. We are assuming that each evaluation of the function h takes one clock cycle and hence h is implemented either as a look

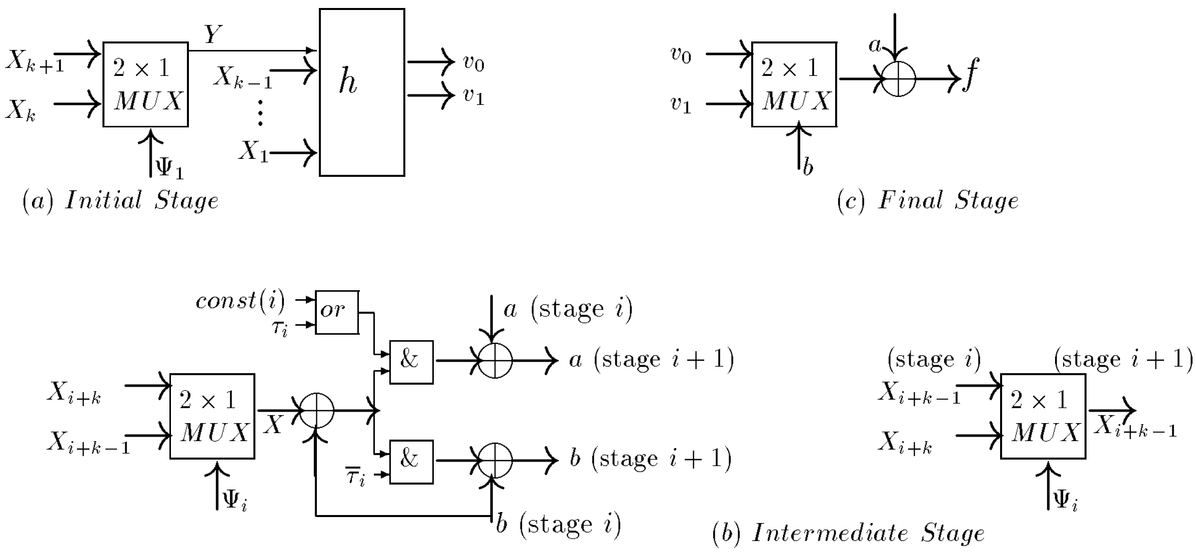


Figure 5: Components of Top Down Architecture

up table or by a small depth computational circuit.

The intermediate stages of the pipeline perform the task of variable interchange and updation of the bits a and b (see Figure 5b). The bits v_0, v_1 are carried forward unchanged. If $\Psi_i = R$ the value of X_{i+k} and X_{i+k-1} should be properly interchanged for the next stage as in lines (1*) and (2*) of the algorithm. The 2×1 multiplexer ensures that the output is X as required by the algorithm. If X and b are unequal, then the two & gates are activated, otherwise a and b are carried forward unchanged to the next stage. If $\tau_i = 0$, then τ_i represents r or rc and the input has to be complemented. The & of $(X \oplus b)$ and $\overline{\tau}_i$ ensures this. If $\tau_i = 1$, then τ_i is c and the output certainly needs to be complemented. Also if $\tau_i = 0$ but represents rc, then also the output needs to be complemented. But τ_i can represent rc only if $i - m_1 \equiv 0 \bmod 2$. The value of the function $const(i)$ is $(i - m_1 + 1) \bmod 2$, and the combination of the *or* and & gate ensures that a is updated as required.

The final stage (Figure 5c) is simple. Depending on the value of b, it outputs either v_0 or v_1 and a is simply EXORed with the output of the 2×1 multiplexer.

The whole circuit operates as follows. At each clock, stage $\#i$ forwards the values of the variables to the next stage and updates the values of work bits a, b for the next stage. The values v_0 and v_1 are forwarded unchanged. It is important to understand the need for generation of v_0, v_1 at the first stage and carrying them through all the t stages. We need these two bits only at the end for the final circuit (Figure 5c). However, the values of v_0, v_1 are generated from the variables X_1 to X_{k+1}. It is more efficient to carry two bits v_0, v_1 through the t stages instead of carrying the $k+1$ bits $X_1, \ldots, X_{k+1}$. Since there are t stages, the whole pipeline takes t clock cycles to be completely filled up. Hence the first output appears at $(t+1)$th clock and consequently a bit of output appears at each clock.

We use both the rising and falling edge of the clock. Each stage stores two buffers, one input and another output (see Figure 6). At the leading edge the values of the input buffer registers of

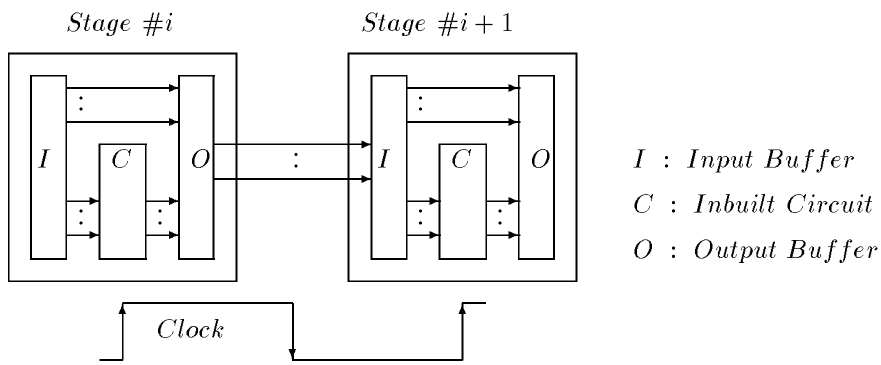


Figure 6: Input Output Latching for Intermediate Stages

stage $\#i$ are latched to the output buffer registers of the same stage. The signals $X_k, \ldots, X_{i+k-2}$ and v_0, v_1 go directly from input buffer to output buffer. The other three signals X_{i+k-1}, a, b are generated through the inbuilt combinational circuit (Figure 5b) from X_{i+k}, X_{i+k-1}, a, b and Ψ_i, τ_i. That is, the stage C in Figure 6 contains the circuit of Figure 5c. At the falling edge of the clock, the output buffer registers of stage $\#i$ are latched to the input buffer registers of the stage $\#(i+1)$. The inbuilt combinational circuit being small enough, it is justified to consider that the delay of the circuit is much less than the clock width and hence there is no problem in using both the leading and falling edge of the clock in the hardware. The inbuilt combinational circuit blocks in this architecture can also be implemented using small lookup table.

Note that the Boolean function is reconfigurable as if we can load a new set of values for $(\Psi_t, \tau_t), \ldots, (\Psi_1, \tau_1)$, then the function F will change, keeping the cryptographic parameters same. This will help in accessing the elements of a large set of Boolean functions with minimum possible change, changing the pattern of $2t$ bit values.

We also address the issue of synchronization at this point. The same kind of system will be available to both the sender and receiver. Once both sides start with a specific key, the first output comes after a delay of t clock cycles, i.e. starting from the $(t+1)$-th clock. Now consider the case when the key of the system is going to be changed. In that case, when the new key is loaded, then the pipeline will contain some data generated from the earlier key. The data coming from the next key will be operational after t clock cycles after it is loaded in the LFSRs. This is the same case in both the sender and receiver end. Hence, there is no additional requirement for synchronization in this setup.

4.3 A Specific Example

Consider the implementation of a function F on 24 variables. We take the initial function h to be a 16-variable function, with order of resiliency 8, algebraic degree 7 and nonlinearity $2^{15} - 2^9$ [6]. The function h is optimized with respect to the parameters considered here. Now we use a pipeline of 8 levels to get the 24-variable function F with order of resiliency 16, algebraic degree

7, and nonlinearity $2^{23} - 2^{17}$. These are also a set of best possible parameters. The user has an option of selecting $2 \times 8 = 16$ bits for $(\Psi_8, \tau_8), \ldots, (\Psi_1, \tau_1)$, to get a fairly wide range (2^{16}) of choices for F. Further, it is possible to design a suitable architecture so that the values of these 16 bits can be programmable and the design can be implemented using an FPGA structure. Thus it is possible to design a reconfigurable structure which can be programmed to implement any one of the 2^{16} possible 24-variable Boolean functions F. The VLSI area required to implement the reconfigurable structure is rougly equal to the VLSI area required to implement the 16-variable initial function h. An overall delay of only 8 clock cycles is introduced in the system due to the pipeline. Note that the delay is a constant 8 clock cycles and is independent of the length of the key stream.

In this system, we will have 24 different LFSRs. We implement the LFSRs by CA. Depending on the requirement of the total key size, we need to choose the length of the CAs, where the lengths of any two CAs are coprime. Let us consider the maximum length of an CA will be less than 128. As a specific example, consider that the lengths will be the following values : $41, 43, 47, 53, 57, 59, 61, 67, 71, 73, 74 = 2 \times 37, 77 = 7 \times 11, 79, 83, 89, 93 = 3 \times 31, 97, 101, 103, 107, 109, 113, 115 = 5 \times 23, 119$. The total summation of these lengths, i.e. the key size of the system is 1931. Since the resiliency of F is 16, the first affine function to which F will have non zero correlation must be non degenerate on 17 variables. Hence, the best possible correlation attack will need estimating an equivalent polynomial of length x from the cipher text or the key sequence. Let x be the sum of the first 17 values in the above sequence. Here $x = 1164$, and hence any known attack is infeasible. It is also important to note that the connection polynomial of the equivalent LFSR is the product of the connection polynomials of the individual LFSRs.

The two dimesional array of flip flops required to implement the CAs is going to be an 24×128 array. Thus the total number of flip flops is going to be 3072. Note that of these only 1931 are going to be used. This is a small trade off to obtain a uniform design. Also note that this number of flip flops are going to be non functional irrespective of whether CA or LFSR is used. The use of CA ensures that the connection structure of the array is going to be uniform.

5 Conclusion

In this paper we have proposed LFSR systems employing large Boolean functions. We have described hardware implementation of large Boolean functions constructed using the recursive method of [4], with optimized function as an initial one [9]. The main point we have tried to make is that LFSR systems employing large Boolean functions are feasible to implement in hardware. We provide a reconfigurable pipelined architecture for the large Boolean function and propose the use of cellular automata for a regular VLSI structure of different length LFSRs. Given the known attacks (see [14] and the references in this paper) and the current advancement of the computer systems, it is improbable that this kind of system will be vulnerable in near future. To the best of our knowledge this is the first effort to consider this problem. Several questions remain as to the best possible implementation and the implementation of Boolean functions constructed using other recursive methods. We feel these can be possible future research topics.

References

[1] C. Ding, G. Xiao, and W. Shan. *The Stability Theory of Stream Ciphers.* Number 561 in Lecture Notes in Computer Science. Springer-Verlag, 1991.

[2] P. P. Chaudhuri. Additive cellular automata: theory and applications, volume 1. IEEE Press, NJ, 1997.

[3] S. W. Golomb. Shift Register Sequences. Aegean Park Press, 1982.

[4] S. Maitra and P. Sarkar. Highly nonlinear resilient functions optimizing Siegenthaler's inequality. In *Advances in Cryptology - CRYPTO'99*, number 1666 in Lecture Notes in Computer Science, pages 198–215. Springer Verlag, August 1999.

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

[6] E. Pasalic, S. Maitra, T. Johansson and P. Sarkar. New constructions of resilent and correlation immune boolean functions achieving upper bounds on nonlinearity. In *Proceedings of the Workshop on Cryptography and Coding Theory*, Paris, 2001.

[7] P. Sarkar. A brief history of cellular automata. *ACM Computing Surveys* Volume 32, Issue 1 (2000), Pages 80-107.

[8] P. Sarkar and S. Maitra. Construction of nonlinear Boolean functions with important cryptographic properties. In *Advances in Cryptology - EUROCRYPT 2000*, number 1807 in Lecture Notes in Computer Science, pages 491–512. Springer Verlag, 2000.

[9] P. Sarkar and S. Maitra. Nonlinearity bounds and constructions of resilient boolean functions. In *Advances in Cryptology - CRYPTO 2000*, number 1880 in Lecture Notes in Computer Science, pages 515–532. Springer Verlag, 2000.

[10] J. Seberry, X. M. Zhang, and Y. Zheng. On constructions and nonlinearity of correlation immune Boolean functions. In *Advances in Cryptology - EUROCRYPT'93*, pages 181–199. Springer-Verlag, 1994.

[11] M. Serra and T. Slater. A Lanczos algorithm in a finite field and its applications. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 1990.

[12] Y. V. Tarannikov. On resilient Boolean functions with maximum possible nonlinearity. *Proceedings of INDOCRYPT 2000*, volume 1977 of LNCS, pages 19-30.

[13] S. Tezuka and M. Fushimi. A method of designing cellular automata as pseudorandom number generators for built-in self-test for VLSI. In *Finite Fields: Theory, Applications and Algorithms*, Contemporary Mathematics, AMS, pages 363–367, 1994.

[14] T. Johansson and F. Jonsson. Fast Correlation Attacks through Reconstruction of Linear Polynomials. *Proceedings of CRYPTO 2000*, volume 1880 of LNCS, pages 300-315.

[15] W. Meier and O. Staffelbach. Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1:159–176, 1989.

[16] S. Wolfram. Theory and applications of cellular automata: including selected papers 1983-1986. World Scientific, NJ, 1986.