Low Cost Hardware Architecture for Secure Stream Cipher Cryptosystems *

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, Kolkata 700 035, INDIA
e-mail: subho@isical.ac.in

Abstract

We design low cost hardware architecture for secure stream cipher cryptosystems. Our architecture provides the first practical method of implementing state-of-the-art theoretical constructions of secure Boolean functions suitable for stream ciphers. Using a pipelined architecture, we show that it is possible to implement systems which use Boolean functions of a relatively large number (around 24) of variables. Our architecture is reconfigurable and provides a universal circuit for a certain class of secure Boolean functions. We also propose the use of Cellular Automata to replace the Linear Feedback Shift Registers in stream ciphers. This provides certain design advantages without affecting the overall security of the system. We present concrete real-life designs of an 8-variable and a 24-variable stream cipher system.

Index Terms : Cryptography, Stream Cipher, Boolean function, Linear Feedback Shift Register, Cellular Automaton, Reconfigurable Hardware, Pipelined Architecture.

*This is a significantly revised and expanded version of the paper *Efficient Implementation of "Large" Stream Cipher Systems* presented at the *3rd Workshop on Cryptographic Hardware and Embedded Systems*, CHES 2001, May 13 - 16, Paris, France.

1 Introduction

Stream cipher cryptography is a classical method of secure information exchange. In this method, the message is considered to be a bit stream. Encryption is done by bitwise XORing with a pseudo random bit stream. This gives the cipher bit stream. Decryption is done by again bitwise XORing the original pseudo random bit stream to the cipher bit stream.

Let $(M_i)_{i\geq 0}$, $(K_i)_{i\geq 0}$ and $(C_i)_{i\geq 0}$ respectively be the message, pseudo random and cipher bit streams. The enciphering operation is the following.

$$C_i = M_i \oplus K_i, \quad i \geq 0.$$

The bit stream C_i is transmitted. At the receiving end, deciphering is done in the following manner.

$$C_i \oplus K_i = M_i \oplus K_i \oplus K_i = M_i, \quad i \geq 0.$$

Stream cipher systems provide high speed encryption and hence are used in defence communications all over the world. One of the popular models of hardware based stream ciphers is shown in Figure 1. In this model, the outputs of several independent Linear Feedback Stream Registers (LFSRs) are combined using a Boolean function F to produce the pseudo random bit stream. At each clock cycle, each of the n LFSRs produce a bit of output. These n-bits are combined by the Boolean F function to produce a pseudo random bit. Thus one pseudo random bit is produced at each clock cycle and hence the rate of encryption is also one bit per clock cycle. The secret key of the system consists of the initial conditions of all the LFSRs.

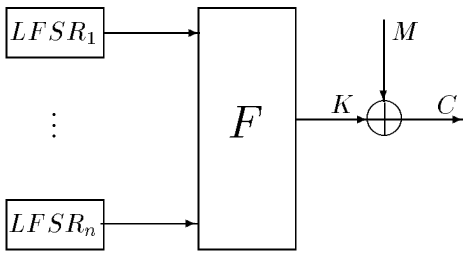


Figure 1: Stream Cipher System.

The model in Figure 1 has been studied extensively in the literature (see for example [17, 18, 11, 6, 14, 15, 4, 19, 10]). The combining Boolean functions must possess certain cryptographic properties for the pseudo random bit stream to be secure. Attacks on the model [8, 4, 2, 18] have shown the necessity for these properties. On the other hand, active research has been conducted in designing secure Boolean functions as evidenced by the recent papers [6, 14, 15, 19, 10]. Currently, it is well accepted in the cryptography community that using Boolean functions with suitable parameters will ensure security against all the known attacks.

The hardware area used in implementing the model in Figure 1 have two components.

1. The area used to implement all the LFSRs.

2. The area used to implement the Boolean function.

The area used to implement all the LFSRs is linear in the number of LFSRs while the area required to implement the Boolean function can be exponential in the number of LFSRs. Let us compute some parameters to get a feel of the problem. Suppose a 24-variable combining function is used where the lengths of the LFSRs are 64 bits on an average (we later indicate why equal length LFSRs are not used). Then the number of flip-flops required to implement the LFSRs is only 1536, while a direct implementation of the Boolean function can require area proportional to 2^{24}. Thus a straightforward implementation of a 24-variable system is prohibitively costly.

Many mathematical constructions of secure Boolean functions are known. However, the actual implementation issues of the state-of-the-art constructions [6, 14, 19, 10] have not been tackled. For the mathematical theory to be useful, it is important to translate the theoretical constructions into actual hardware circuits. There is no general purpose method for doing this. Here we provide an efficient, low cost method for implementing the recursive construction presented in [6]. Using our method, Boolean functions of a large number of variables (around 24) can be easily implemented in hardware.

We provide specific designs of 8-variable and 24-variable systems in Section 5. These specific designs are an important part of the paper and are more valuable than mere illustrative examples. The designs have carefully thought out parameters and are meant to be directly used by manufacturers of stream ciphers. To the best of our knowledge, currently no other stream cipher system provides 24-variable security.

The functions in [6] are built recursively. A function F of n variables is built up from a function h of $k(< n)$ variables. We first describe an algorithm which uses a subroutine for the function h and computes the output of F on an n-bit input. The time required to compute the output of F is linear in $t(= n - k)$, assuming that the output of h can be computed in constant time. The space required by the algorithm is $O(1)$ plus the space required to implement the subroutine for h.

A direct hardware implementation of the algorithm requires t clock cycles to produce one pseudo random bit. This is clearly unacceptable. We require a pseudo random bit per 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 t cycles to be filled up and after that it can handle an n-bit input at each clock cycle. In our 8 and 24-variable designs, $t = 3$ and 14 respectively. Thus there is no effective degradation in the throughput of the system.

There are t similar stages to the pipeline providing a uniform design. The combinational circuit of each stage is implemented by a very small, constant size circuit. The size of the pipeline is a small fraction of the size of the circuit required to implement h. In the 24-variable system given in Section 5, h is implemented using a look-up table of size 2048 and the rest of the pipeline requires only 148 flip flops, 99 gates and thirty 2×1 MUXes.

Our design produces a reconfigurable architecture. This means that the same circuit can be used to implement a large class of stream cipher systems. In fact, in Section 6, we show that the architecture provides a universal circuit for a certain class of secure functions.

The overall implementation of the stream cipher system require the implementation of both the LFSRs and the Boolean function. We propose the replacement of LFSRs by Cellular Automata (CA), without affecting the security of the system. The interconnection structure of a CA is regular and uniform, which is a distinct advantage in VLSI design. Additionally, the use of CA provides added reconfigurability to the design. This aspect is discussed in Section 6. The use of CA has been suggested for many other traditional LFSR applications (see [1]).

The organization of the paper is the following. In Section 2, we present preliminaries on LFSR and CA, the cryptographic properties of Boolean functions used in Figure 1 and the recursive construction of Boolean functions from [6]. In Section 3, we present the CA based design of the set of LFSRs. The algorithm and hardware design for the Boolean function are described in Section 4. Two specific designs are presented in Section 5. Reconfigurability features of the system are described in Section 6. Finally, Section 7 concludes the paper with discussion for possible future work.

2 Preliminaries

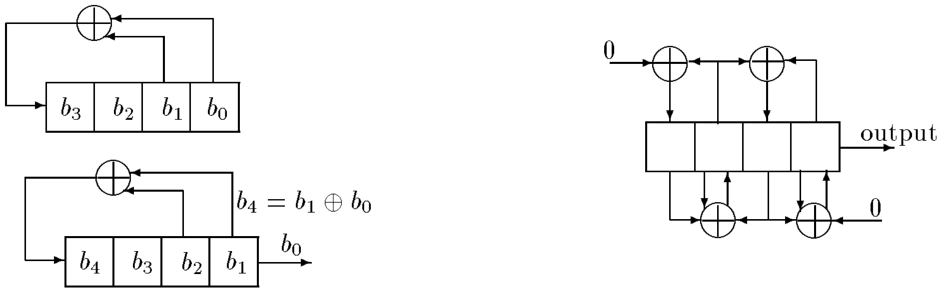
2.1 LFSR

LFSRs are simple hardware devices which generate a bit stream satisfying a linear recurrence. An LFSR consists of a register of flip flops, which are set to some initial conditions. We explain the operation of an LFSR by an example. Suppose $(b_t)_{t\geq 0}$ is a bit stream satisfying the recurrence $b_t = b_{t-3} \oplus b_{t-4}$ for $t \geq 4$. The recurrence needs the string $b_3b_2b_1b_0$ to start. This string is the initial condition. The LFSR in Figure 2(a) produces the bit stream $(b_t)_{t\geq 0}$. See [3] for more details on LFSR. The following cryptographic design criteria are required of the LFSRs in Figure 1.

1. The connection polynomial of each LFSR is primitive. This ensures maximum possible period for the output bit stream of each individual LFSR.
2. The length of each LFSR is at least 40.
3. Each LFSR must have a large number of tap connections [8].
4. The lengths of the LFSRs must be distinct from each other. This ensures maximum linear complexity of the pseudo random stream K in Figure 1 (see [12, Theorem 3]).

2.2 CA

It is known [1], that a certain class of CA are algebraically equivalent to LFSRs. Our motivation in this section is to describe this equivalence. This will show that replacing CA by LFSR will not affect the security of the system.



(a) LFSR with connection poynomial $x^4 + x + 1$

(b) A (90,150,90,150) CA with characteristic polynomial $x^4 + x + 1$.

Figure 2: Examples of LFSR and CA.

A cellular automaton is a register of flip flops where each flip flop is called a cell. At each clock pulse, the state of each cell changes. For an l-cell CA, let c_i^j be the state of the ith cell at the jth clock cycle, $1 \leq i \leq l$, $j \geq 0$. Then

$$c_i^j = c_{i-1}^{j-1} \oplus u_i c_i^{j-1} \oplus c_{i+1}^{j-1}. \tag{1}$$

If $u_i = 0$, then the ith cell is said to follow rule 90 and if $u_i = 1$, the ith cell is said to follow rule 150. (See [1] for an explanation of CA rules). A 4-cell 90/150 CA is pictorially depicted in Figure 2(b). The cell at the left end does not have a left neighbour and one at the right end does not have a right neighbour. The output of the CA is taken as the output of the rightmost cell.

The collection of values of the cells at each clock cycle is called the state of the CA. The change of values of the cells determine the next state of the CA. The next state evolution of a 90/150 CA can be totally described by a tridiagonal matrix, called the state transition matrix. We explain by an example.

Consider the 4-cell $(90, 150, 90, 150)$ CA of Figure 2(b). 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

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

Given the initial configuration, the behaviour is totally determined by the state transition matrix.

It is known [1] that *if the characteristic polynomial of the state transition matrix is primitive over $GF(2)$, then the output of an l-cell CA initialised by a non null configuration will have period* $2^l - 1$. The characteristic polynomial of the $(90, 150, 90, 150)$ CA is $x^4 + x + 1$, which is primitive over $GF(2)$ and hence the output of the CA is 110101111000100, which has period 15.

The output bit stream of the CA is completely determined by the characteristic polynomial of the state transition matrix. This is the basis for replacing LFSR by CA. Suppose $(b_t)_{t \geq 0}$ is a bit stream generated by an LFSR whose connection polynomial $p(x)$ is primitive. Then the same sequence (modulo a shift) is generated by a CA whose state transition matrix has $p(x)$ as its characteristic polynomial. Thus the structures are algebraically equivalent. As in the case of LFSR, the secret key is the initial configuration of the 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. However, methods for doing this are known [16, 20]. 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 the above discussion, it is clear that replacing LFSR by CA does not alter the security of the system. The advantages to be gained are uniformity and reconfigurability in the design. See [1] for other VLSI applications where the CA based design has replaced LFSR based designs. Also see [13] for a survey on the general theory of CA.

2.3 Properties and Construction of 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 [17, 6, 14].

Definitions :

- An n-variable function is said to be *balanced* if the output column of its truth table has equal number of zeros as ones.
- A Boolean function is said to be *m-resilient*, if the probability of the output being one is half even if at most m of the inputs are fixed to constant values.
- The *algebraic normal form* of a Boolean function is its canonical sum of products representation using XOR and AND gates which is 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 at most 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.

There are several known methods [6, 14, 19, 10] for the design of Boolean functions possessing a secure combination of the above mentioned properties, namely, number of variables, order of resiliency, algebraic degree and nonlinearity. Further, the results of [15] identifies the class of Boolean functions which achieve the best possible trade-off among these properties. Many of these best functions can be constructed by the methods of [6, 14, 19, 10].

There are two approaches to the construction - direct and recursive. Here we show how to implement functions obtained by the recursive construction method presented in [6]. The advantage of the method of [6] is that it is simpler than the other methods [14, 19, 10]. We next provide a description of the construction method of [6].

Suppose an n-variable function $F(X_n, \ldots, X_1)$ is to be used in the stream cipher system. Following the method of [6], 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,

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

- If $\Psi_i = R$, then

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

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 $a = 1, b = 0$
- if $\tau_i = rc$, then $a = b = 1$.

It is important to note that at each step either $\tau_i \in \{r, c\}$ or $\tau_i \in \{rc, c\}$ (see [6]). 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 $\tau_i \in \{c, rc\}$.

Note that $n = k+t$, and $F = F_t$. If h has order of resiliency m_1, then F has 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 (see [6]).

The construction method produces a class of functions and not just a single one. There are two things to be noted.

- The choice of the function h is not unique. The values of the parameters - number of variables, resiliency, algebraic degree and nonlinearity - are specified. One can choose any h which satisfies these values.
- For a fixed h, the construction produces $2^{2t} = 4^t$ possible functions F. At each stage, there are two possible choices for each of Ψ and τ.

Example : We provide an example to illustrate the construction method described above. Suppose we want to construct an 8-variable, 4-resilient, degree 3 and nonlinearity 96 function F. Any function with these values of the parameters achieves an optimal trade-off among the mentioned parameters [15].

We start the recursive construction using a 5-variable, 1-resilient, degree 3 and nonlinearity 12 function h. These values also achieve an optimal trade-off among the concerned parameters [15].

We provide a choice of h from [14]. (See also [7] for a different construction.)

$$\begin{aligned} h(x_5, \ldots, x_1) &= (x_5 \oplus 1)(x_4 \oplus 1)(x_1 \oplus x_2) \oplus (x_5 \oplus 1)x_4(x_1 \oplus x_3) \\ &\quad \oplus x_5(x_4 \oplus 1)(x_2 \oplus x_3) \oplus x_5 x_4(x_1 \oplus x_2 \oplus x_3) \end{aligned}$$

Given h, there are $4^3 = 64$ possible functions F which can be constructed from h. Two examples are

1. F represented by $(h, (Q, rc), (R, r), (Q, c))$.

2. F represented by $(h, (R, c), (Q, r), (R, rc))$.

Next we provide a detailed description of the function F represented by $(h, (Q, rc), (R, r), (Q, c))$.

$$\begin{aligned} F_0(x_5, \ldots, x_1) &= h(x_5, \ldots, x_1) \\ F_1(x_6, \ldots, x_1) &= (1 \oplus x_6)F_0(x_5, \ldots, x_1) \oplus x_6(1 \oplus F_0(1 \oplus x_5, \ldots, 1 \oplus x_1) \\ F_2(x_7, \ldots, x_1) &= (1 \oplus x_6)F_1(x_7, x_5, \ldots, x_1) \oplus x_6(1 \oplus F_1(1 \oplus x_7, 1 \oplus x_5, \ldots, 1 \oplus x_1) \\ F_3(x_8, \ldots, x_1) &= (1 \oplus x_8)F_2(x_7, \ldots, x_1) \oplus x_8(1 \oplus F_2(x_7, \ldots, x_1)) \\ &= x_8 \oplus F_2(x_7, \ldots, x_1) \end{aligned}$$

The function F_3 is the desired function F. Note that, while obtaining F_3 from F_2, an algebraic simplification is possible. No such simplification is in general possible in the first two steps. In general, the algebraic normal form of the function will be quite complicated [6]. ■

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 [6, 15].

3 CA Based Design for the LFSRs of Figure 1

Here we discuss the replacement of the LFSRs in Figure 1 by CA. Recall from Section 2.1 the design criteria on the LFSRs of Figure 1. The main difficulties in the implementation of the LFSRs are the following.

1. The LFSR lengths need to be distinct. A direct implementation would have to use registers of different lengths resulting in a non uniform structure.

2. The tap points of an LFSR are in general not regularly placed. In addition, the number of taps in the LFSR must be high. Further, an LFSR with a primitive connection polynomial, necessarily has a long feedback connection and the length of this feedback connection is 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. We discuss how the above problems can be tackled.

To tackle the first problem, we suggest the use of an $n \times L$ two dimensional array 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 64). 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 of [20]. 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.

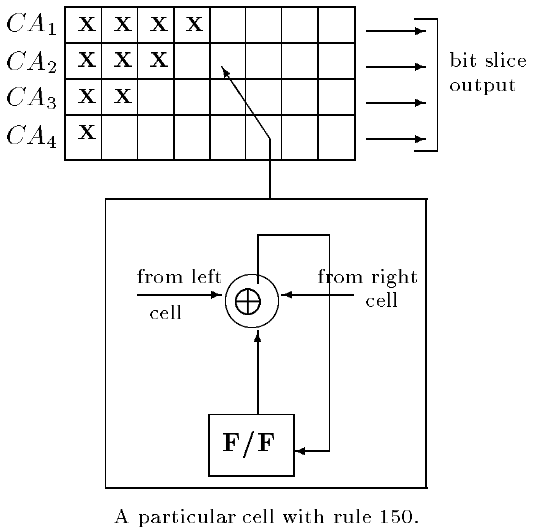


Figure 3: Example of a 4×8 array of flip-flops implementing 4 CA having lengths 4,5,6,7.

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 CAs are taken in a bit slice manner from the right end of the two dimensional array. The non functional flip flops are 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 regular and uniform architecture is well suited for VLSI design. See Figure 3 for a small example of the two dimensional CA structure described here. The flip flops marked X are non functional.

4 Boolean Function Implementation

The crucial problem is to design circuits for the Boolean functions described in Section 2.3. The requirement on any such circuit is two fold.

- The size of the circuit implementing F must not be much larger than the size of the circuit implementing h.
- The circuit must be able to compute a bit of output per clock cycle.

Thus we have to compute the output of F represented by $(h, S_1, \ldots, S_t)$ on an n-bit input $X_n, \ldots, X_1$. We first obtain a recursive algorithm based on the recursive description in Section 2.3. Then we eliminate the recursion to obtain an iterative algorithm. This iterative algorithm requires t steps to compute the output. Thus the algorithm cannot be directly implemented. We bypass the problem by mapping the algorithm into a pipelined architecture. The pipeline takes t clock cycles to fill up and after that produces a bit of output at each clock cycle. The total delay for obtaining all the pseudo random 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

Let F be represented by $(h, S_1, \ldots, S_{n-k})$, where h is a function of k variables and let $t = n - k$. We will refer to the recursive definition of F_i provided in Section 2.3. As before, $F_0 = h$ and F_i is the function represented by $(h, S_1, \ldots, S_i)$. Then $F_t = F$.

First we present a recursive algorithm to compute $F_t = F_{n-k} = F(X_n, \ldots, X_1)$.

$recCompute(F_i(X_{i+k}, \ldots, X_1))$
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 $(\tau_i = r)$ return $recCompute(F_{i-1}(1 \oplus X_{i+k-1}, \ldots, 1 \oplus X_1)$;
8. if $(\tau_i = rc)$ return $1 \oplus recCompute(F_{i-1}(1 \oplus X_{i+k-1}, \ldots, 1 \oplus X_1)$;
9. end if

end

The call $recCompute(F_t(X_n, \ldots, X_1))$ will correctly return the value of $F(X_n, \ldots, X_1)$.

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. The main properties of $recCompute()$ are as follows.

1. It takes t steps to compute a bit of output.

2. The stack depth is $O(t)$.

Having a large stack depth makes the algorithm inefficient to implement. Fortunately, the recursion in $recCompute()$ is a case of tail recursion and can be removed. 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 one of $v_0, v_1, v_0 \oplus 1, v_1 \oplus 1$, 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 managed 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, ..., X_1) {
    if (Ψ_1 = Q) then Y = X_k;
    if (Ψ_1 = R) then Y = X_{k+1};
    v_0 = h(Y, X_{k-1}, ..., X_1);  v_1 = h(1 ⊕ Y, 1 ⊕ X_{k-1}, ..., 1 ⊕ X_1);
    a = 0; b = 0;
    for i = t downto 1 do {
        (1*) if (Ψ_i = Q) then X = X_{i+k};
        (2*) if (Ψ_i = R) then { X = X_{i+k-1}; X_{i+k-1} = X_{i+k}; }
        if (b ⊕ X = 1) then {
            if (τ_i = c) then a = a ⊕ 1;
            if (τ_i = r) then b = b ⊕ 1;
            if (τ_i = rc) then {a = a ⊕ 1; b = b ⊕ 1; }
        }
    }
    return a ⊕ v_b;
}
```

We provide an explanation for testing the condition $b \oplus X = 1$. At any stage of the algorithm, the variable X is the one on which the function F_i is projected. The variable b records whether the input needs to be complemented. If $X = b = 0$, then the values of a and b do not need to be changed. Also if $X = b = 1$, then we will have to complement the value of the input X and thus again get 0. In this case also, the values of a and b do not need to be changed. In the other two cases, the values of a and b need to be updated based on the value of τ_i. The rest of the algorithm is straightforward. The time taken by $computeTD()$ to compute the output is $O(t)$. Thus *algorithm* $computeTD(X_n, \ldots, X_1)$ *correctly computes* $F(X_n, \ldots, X_1)$ *in* $O(t)$ *time.*

Example : We provide an example of the behaviour of algorithm $computeTD()$. We use the example of Section 2.3. Let F be an 8-variable function represented by $(h, (Q, rc), (R, r), (Q, c))$. In this case $k = 5$, $n = 8$ and $t = 3$.

Suppose we want to compute the output of F on the input $101X_5X_4X_3X_2X_1$. Thus here we have $X_8 = 1$, $X_7 = 0$ and $X_6 = 1$.

initialization step : Since $\Psi_1 = Q$, we obtain $Y = X_5$. This gives $v_0 = h(X_5, X_4, X_3, X_2, X_1)$ and $v_1 = h(1 \oplus X_5, 1 \oplus X_4, 1 \oplus X_3, 1 \oplus X_2, 1 \oplus X_1)$. The variables a and b are set to 0.

step $i = 3$: At this point we have $a = 0$, $b = 0$.

- $\Psi_3 = Q$ and so we obtain $X = X_8 = 1$.
- Since $X_8 \oplus b = 0$ and $\tau_3 = c$, we update the value of a to 1. The value of b remains 0.

step $i = 2$: At this point we have $a = 1$, $b = 0$.

- $\Psi_2 = R$ and so we obtain $X = X_6 = 1$. Also the value of X_6 is set to that of X_7, i.e., the value of X_6 now becomes 0.
- Since $b \oplus X = 1$ and $\tau_2 = r$, we update the value of b to 1. The value of a remains 1.

step $i = 3$: At this point we have $a = 1$, $b = 1$.

- $\Psi_1 = Q$ and so we obtain $X = X_6 = 0$. Note that the value of X_6 has been changed to 0 in the previous step.
- Since $b \oplus X = 1$ and $\tau_1 = rc$, we update the values of both a and b. Both are changed to 0.

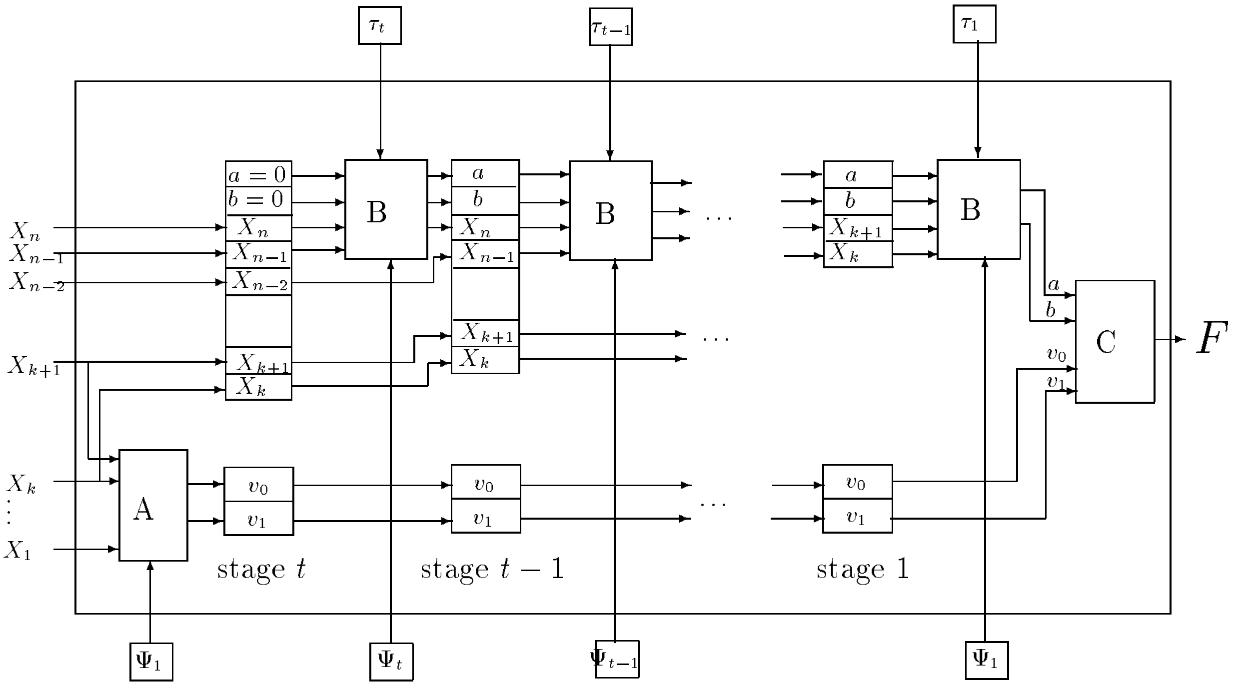
final step : The value $a \oplus v_b$ is returned. At this point $a = b = 0$, and so v_0 is returned. ■

4.2 Hardware Architecture for *computeTD*(.)

As mentioned before, a direct implementation of algorithm *computeTD*() will mean that t clock cycles are required to produce one bit of output. This will lead to unacceptable degradation in the performance of the system. Here we show how a low cost pipelined architecture can be developed to implement the circuit for F. The pipeline takes t clock cycles to fill up. The output of F on successive tuples of n-bit input is available at each clock pulse after the initial t clocks pulses, i.e., starting from $(t + 1)$-th clock pulse.

In the hardware description, we will be manipulating Ψ_i, 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 $\tau_i = 1$.



A: initial circuit; B: intermediate circuit; C: final circuit;

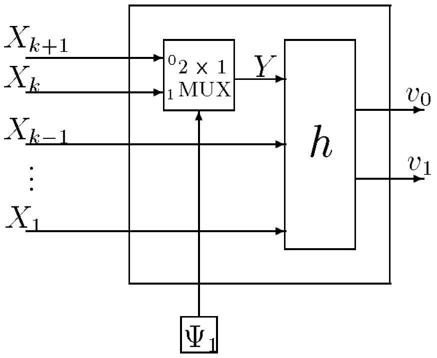
Figure 4: Pipelined Architecture.

- 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 $i \equiv m_1 \bmod 2$, where m_1 is the order of resiliency of the initial function h (see Section 2.3).

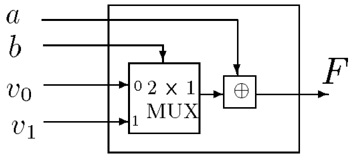
The pipeline has t internal stages numbered 1 to t (see Figure 4). Stage i stores the current values of $X_k, \ldots, X_{k+i}$. The two bits v_0 and v_1 are present at each stage along with the two other work bits a and b.

The initial circuit (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. This is tackled in the following manner. The function h is implemented by a look-up table. Corresponding to an input $X_k, \ldots, X_1$, the look-up table stores the values $h(X_k, \ldots, X_1)$ and $h(1 \oplus X_k, \ldots, 1 \oplus X_1)$. Thus in one clock cycle the look-up table provides the values of v_0, v_1. The size of the look-up table is 2^{k+1}. We provide some precise parameters in Section 5.

The intermediate stages of the pipeline perform the task of variable interchange and updation of the bits a and b (see Figures 6 and 7). The bits v_0, v_1 are carried forward without being changed. If $\Psi_i = R$ the values of X_{i+k} and X_{i+k-1} are properly interchanged for the next stage as in lines (1*) and (2*) of algorithm *computeTD*(). The 2×1 multiplexer ensures that the output X has the proper value. 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 & operation 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. If $\tau_i = 0$ but represents rc, then also the output needs to be complemented. But τ_i can represent rc only if $i + m_1 \equiv 0 \bmod 2$ (see



(a) Initial Circuit



(b) Final Circuit

Figure 5: Initial and Final Circuits.

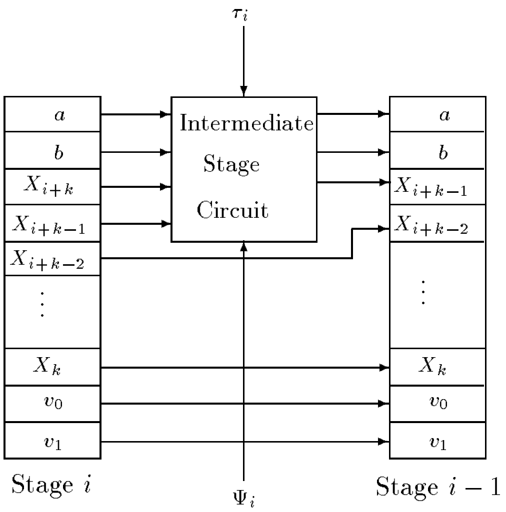


Figure 6: Transition from stage i to stage $i-1$.

Section 2.3). The value of the function $const(i)$ is $(i + m_1 + 1) \bmod 2$, and the combination of the *or* and & gates ensures that a is updated as required.

The final circuit (Figure 5b) is simple. Depending on the value of b, it outputs either v_0 or v_1 and a is simply XORed 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 5b). However, the values of v_0, v_1 are generated from the variables X_1 to X_{k+1}. By carrying the two bits v_0, v_1 through the t stages we can avoid carrying the $k-1$ bits $X_1, \ldots, X_{k-1}$. Only the bits X_k, X_{k+1} are carried. This reduces the number of flip flops required at any intermediate

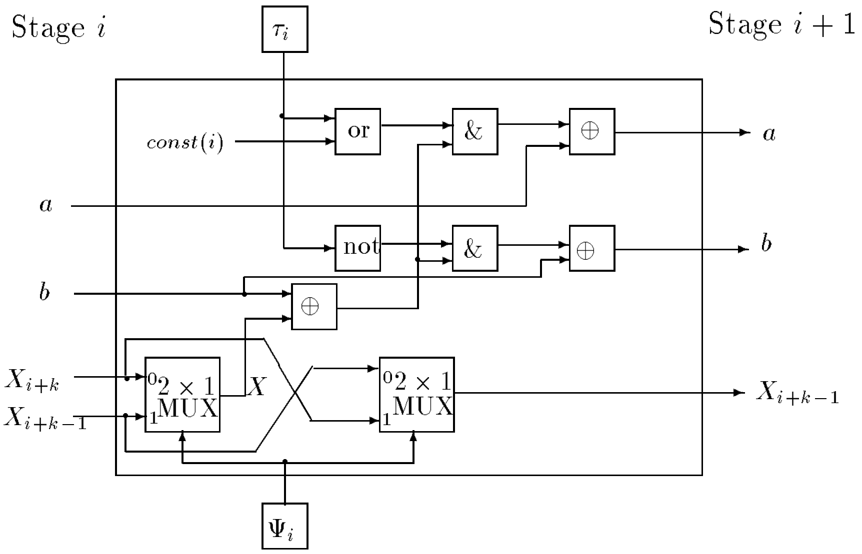


Figure 7: Intermediate Circuit.

stage in the pipeline.

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.

4.2.1 Size of the Pipeline

The size of the pipeline in Figure 4 is the size of the look up table (or circuit) to compute h plus the additional gates and flip flops shown in Figures 5 and 7. We provide an estimate of the size.

1. The initial and final circuits each require a 2×1 MUX. Additionally, the final circuit requires a XOR gate and the initial circuit requires a flip flop to store the value of Ψ_1.
2. Each of the i $(1 \le i \le t)$ intermediate circuits require
   - two 2×1 MUX circuits;
   - two & gates; one *or* gate; one *not* gate; three XOR gates;
   - $i+1$ flip flops to store the values of $X_{k+i} \ldots X_k$ and two flip flops to store the values of Ψ_i and τ_i.

Thus the total counts of the component are the following.

component	t-stage	3-stage	6-stage	9-stage	12-stage	14-stage
2×1 MUX	$2t+2$	8	14	20	26	30
gates	$7t+1$	22	43	64	85	99
flip flop	$2t + \frac{(t+1)(t+2)}{2}$	16	40	73	115	148

As the number of stages grows, the cost will be dominated by the area required to store the values of the different X_j's. However as the above table shows, for a moderate number of stages, the total size is quite small.

4.2.2 Key Synchronization

The Boolean function is part of the stream cipher system. The working of the system requires a secret key to be shared between the sender and the receiver. A particular key is used to generate a fixed number of bits, after which it is replaced by a new key. The sender and the receiver know exactly the points at which the new key is to be used.

In our case, the secret key consists of the initial conditions of all the CAs. The use of a pipeline seem to suggest that when a new key is used the pipeline has to be flushed. We explain that this is not the case.

The same 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 changed. When the new key is loaded, the pipeline will still contain some data generated from the earlier key. The data coming from the new key will be operational only after t clock cycles from the time it is loaded.

This is the same situation for both the sender and the receiver. Both the sender and the receiver load the new keys at the same time. During the time the pipeline gets filled up with data from the new key, the bits from the old key continue to be used. Hence, there is no additional requirement for synchronization in this setup. The operation proceeds without any break in the generation of the pseudo random bits.

5 Specific Designs

We provide two specific real-life designs of stream cipher systems. These designs have well thought out parameters and are meant to be used directly by manufacturers. Our designs are of an 8-variable and a 24-variable system. Current stream ciphers typically use around 8-variable functions, though not necessarily the theoretically best possible functions. In this case, our design provides a low cost method to implement a theoretically best possible function. On the other hand, to the best of our knowledge none of the currently known stream ciphers provide 24-variable security. Thus in this case, our design provides the first practical method of obtaining 24-variable security.

5.1 The 8-Variable System

The 8-variable function F is built from a 5-variable function h as described in Section 2.3. This requires a 3-stage pipeline.

Direct Implementation Size : The size of direct implementation of F is $\simeq 2^8 = 256$ gates/flip flops.

Pipelined Implementation Size : The pipelined implementation requires

– 64 flip flops to implement the look-up table for h.

– Eight 2×1 MUXes, 22 gates and 16 flip flops.

The total delay in the system is 3 clock cycles. Thus, it is clear that the pipelined implementation is going to require lesser size at no effective degradation in throughput.

The lengths of the CA are to be distinct [12, Theorem 3]. We choose the following lengths.

$$41, 42, 43, 44, 45, 46, 47, 48.$$

Primitive polynomials of these degrees along with the algorithm of [20] will provide the connection pattern for the CAs. The total sum of these is the length of the secret key. Thus the secret key is of size 356 bits. The CAs are implemented in a uniform 8×48 array of flip flops. The size of this array is 384. Thus a total of $384 - 356 = 28$ flip flops will be non functional. This is a small price for obtaining a regular design.

5.2 The 24-Variable System

We use a 24-variable function F which is built from a 10-variable function h. The 10-variable function h is chosen to have order of resiliency 4, algebraic degree 5 and nonlinearity 480. Such a function h can be constructed using the method described in [10]. Our target 24-variable function F has order of resiliency 18, algebraic degree 5 and nonlinearity $2^{14} \times 480$. Both h and F achieve optimal trade-off among the mentioned parameters [15]. The function F will be built from h using the method described in Section 2.3. One possible representation for F is

$$\begin{gathered}(h, (Q, r), (P, rc), (P, c), (Q, rc), (P, r), (Q, c), (P, r), (P, rc), \\ (Q, r), (Q, rc), (P, r), (Q, rc), (P, r), (P, rc)).\end{gathered}$$

Implementation of F requires a 14-stage pipeline. If h is fixed (i.e, implemented by a combinational circuit) there are 4^{14} possible functions F which can be implemented by the 14-stage pipeline. We now compare the sizes of direct and pipelined implementation for F.

Direct Implementation Size : The size of direct implementation of F is $\simeq 2^{24}$ gates/flip flops.

Pipelined Implementation Size : The pipelined implementation requires.

– 2048 flip flops to implement the look-up table for h.

– Thirty 2×1 MUXes , 99 gates and 148 flip flops.

The total delay in the system is 14 clock cycles. A direct implementation of F is prohibitively expensive. On the other hand, the pipelined implementation is not only feasible but requires only moderate cost.

We now describe the lengths of the CAs. The lengths of the CAs are to be distinct [12, Theorem 3]. The following values for the lengths are chosen.

$$41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64.$$

As before, primitive polynomials of corresponding lengths along with the algorithm of [20] will provide the connection pattern for the CAs. The total sum of these lengths is the key size of the system. Hence the key is 1260 bits long. The two dimensional array of flip flops required to implement the CAs is going to be an 24×64 array. Thus the total number of flip flops is going to be 1536. Out of these only 1260 are going to be used. This is the trade off for obtaining a uniform design.

6 Reconfigurability

We discuss the reconfigurability of the CA structure and the pipeline separately.

6.1 Reconfigurability of the CA Structure

Recall from Section 2 that the characteristic polynomial of the state transition matrix for the CA completely determines the bit stream produced by the CA. Also this polynomial is taken to be a primitive polynomial to maximise the period of the generated bit stream.

Given any primitive polynomial $p(x)$, it is possible to get a 90/150 CA such that the characteristic polynomial of its state transition matrix is $p(x)$ [20]. The structure of any l-cell 90/150 CA can be easily encoded by a bit string $u = u_1 \dots u_l$ of length l using the following rule.

"If $u_i = 1$, then the ith cell has rule 150, else it has rule 90."

Equation (1) in Section 2.2 describes how the string u determines the change of state of each cell.

The values $u_1, \dots, u_l$ are stored in a control register as shown in Figure 8. Thus changing a primitive polynomial will simply mean changing the values of the control register. In this manner, the CA can be reconfigured to implement any primitive polynomial of length l. This technique is similar to programmable CA [1]. *Here we note that this kind of reconfigurability cannot be obtained if LFSRs are used.*

6.2 Reconfigurability of the Boolean Function

The pipelined architecture provides a natural way to reconfigure the hardware. The values of Ψ_i and τ_i are stored in flip flops (see Figure 4). Consider these flip flops to be organised into two registers - the Ψ-register and the τ-register. The Ψ-register stores the values $(\Psi_1, \Psi_t, \dots, \Psi_1)$ and the τ-register stores the values $(\tau_t, \dots, \tau_1)$.

The function F is completely defined by h and the sequence of values $(\Psi_1, \tau_1), \dots, (\Psi_t, \tau_t)$. Thus it is easy to reconfigure the pipeline. We have to change the following two things.

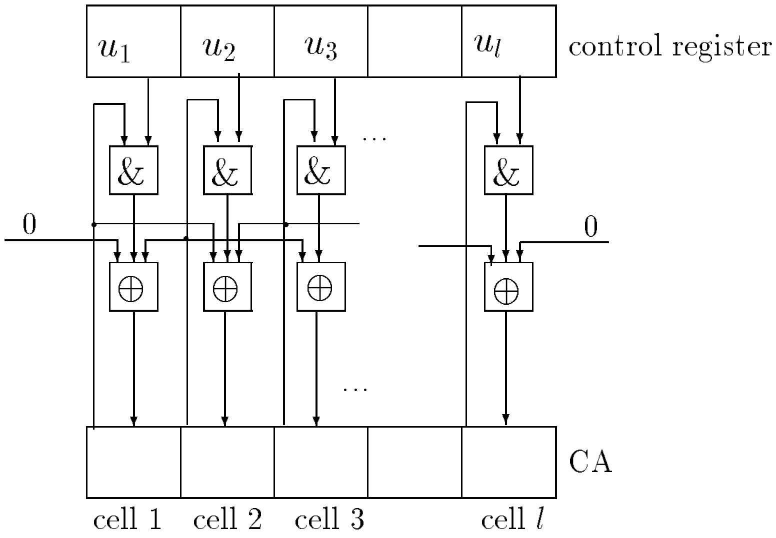


Figure 8: A Reconfigurable l-cell CA.

- Change the look up table for the function h.
- Change the values of the Ψ-register and the τ-register.

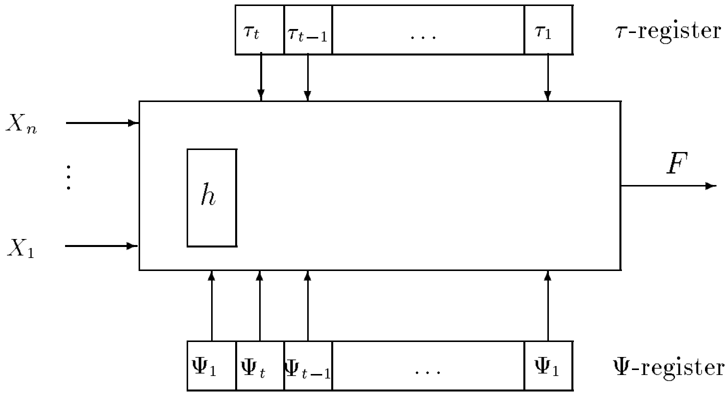


Figure 9: Reconfigurable Boolean function F.

This allows the same circuit to be easily reconfigured to implement any function constructed using the method of Section 2.3 and having a fixed set of parameters (see Figure 9). *Thus the pipeline architecture is a* **universal circuit** *for the class of secure Boolean functions described in [6].*

If the function h is implemented using a combinational circuit, then the first operation cannot be carried out. In this case, reconfigurability is obtained from the second condition. The circuit can be reconfigured to implement any of the 4^t possible functions F. This still provides a large choice of functions.

7 Conclusion

In this paper, we have designed low cost hardware architecture to implement secure stream cipher systems. Our design uses the Boolean functions constructed by the recursive construction of [6]. We have presented the first design of stream cipher cryptosystems providing 24-variable security. To the best of our knowledge, this is the first paper to provide a practical design method for the stream cipher systems employing state-of-the-art cryptographically strong Boolean functions [6].

Several questions remain as to the best possible implementation and the implementation of Boolean functions constructed using other methods [14, 19, 10]. We feel these are future research topics.

Acknowledgement : The authors are grateful to Dr. Anwar Hasan, Prof. Bhabani P. Sinha and Dr. Nabanita Das for reading the paper and providing suggestions that helped to improve the presentation of the paper.

References

[1] P. P. Chaudhuri, D. R. Chowdhury, S. Nandi and S. Chattopadhyay. Additive cellular automata: theory and applications, volume 1. IEEE Press, NJ, 1997.

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

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

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

[5] D. Kahn. The Codebreakers; The Comprehensive History of Secret Communication from Ancient Times to the Internet, 1996.

[6] 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.

[7] S. Maitra and P. Sarkar. "Cryptographically significant Boolean functions with five valued Walsh spectra". *Theoretical Computer Science*, to appear.

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

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

[10] E. Pasalic, S. Maitra, T. Johansson and P. Sarkar. "New Constructions of Correlation Immune and Resilient Boolean Functions Achieving Upper Bounds on Nonlinearity". *Workshop on Coding and Cryptography*, Paris, 2001.

[11] R. A. Rueppel. Analysis and Design of Stream Ciphers Springer-Verlag, 1986.

[12] R. A. Rueppel and O. J. Staffelbach. "Products of linear recurring sequences with maximum complexity". *IEEE Transactions on Information Theory*, volume IT-33, number 1, pp. 124-131, 1987.

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

[14] 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.

[15] 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.

[16] M. Serra and T. Slater. "A Lanczos algorithm in a finite field and its applications". *Journal of Combinatorial Mathematics and Combinatorial Computing*, volume 7, 1990.

[17] T. Siegenthaler. "Correlation-immunity of nonlinear combining functions for cryptographic applications". *IEEE Transactions on Information Theory*, IT-30(5):776–780, September 1984.

[18] T. Siegenthaler. "Decrypting a class of stream ciphers using ciphertext only". *IEEE Transactions on Computers*, C-34(1):81–85, January 1985.

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

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