# Efficient Implementation of Cryptographically Useful "Large" Boolean Functions

Palash Sarkar and Subhamoy Maitra

**Abstract**—We present low cost hardware architecture for 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 of variables. Our architecture is reconfigurable and provide a universal circuit for a certain class of secure Boolean functions.

**Index Terms**—Boolean function, cryptography, pipelined architecture, reconfigurable hardware, stream cipher.

## 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 performed by bitwise XORing the message bit stream with a pseudorandom bit stream. This gives the cipher bit stream. Decryption is performed by bitwise XORing the original pseudorandom 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, pseudorandom, and cipher bit streams. The enciphering operation is the following:

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

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

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

One of the popular models of hardware-based stream ciphers is shown in Fig. 1. In this model, the outputs of several independent Linear Feedback Shift Registers (LFSRs) are combined using a Boolean function $F$ to produce the pseudorandom bit stream. At each clock cycle, each of the $n$ LFSRs produce a bit of output. These $n$-bits are combined by the Boolean function $F$ to produce a pseudorandom bit. Thus, one pseudorandom 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.

The model in Fig. 1 has been studied extensively in the literature (see, for example, [12], [13], [8], [1], [11], [5], [10], [2], [3], [4], [14]). The combining Boolean functions must possess certain cryptographic properties for the pseudorandom bit stream to be secure. Attacks on the model [6], [4], [13], [2], [3] have shown the necessity for these properties. On the other hand, active research has been conducted in designing secure Boolean functions (see, for example, [1], [11], [5], [10], [14]). 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 Fig. 1 has 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 for the problem. Suppose a 24-variable combining function is used, where the lengths of the LFSRs are 64 bits on average. Then, the number of flip-flops required to implement the LFSRs is only 1,536, 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. 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 [5]. Using our method, Boolean functions of a large number of variables can be easily implemented in hardware.

The functions in [5] 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 pseudorandom bit. This is clearly unacceptable. We require a pseudorandom

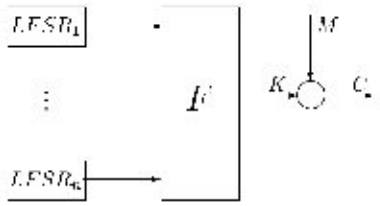


Fig. 1. Stream cipher system.

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. For small $t$, 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 $h$. For the 24-variable Boolean function example given in Section 4, $h$ (a 10-variable Boolean function) is implemented using a look-up table of size $2,048$ (see Remark 1 in Section 3.2) and the rest of the pipeline requires only 148 flip flops, 99 gates, and 30 $2 \times 1$ MUXes.

Our design produces a reconfigurable architecture. This means that the same circuit can be used to implement a large class of Boolean functions. In Section 5, we show that the architecture provides a universal circuit for a certain class of secure functions.

The organization of the paper is the following: In Section 2, we present preliminaries on the cryptographic properties of Boolean functions used in Fig. 1 and the recursive construction of Boolean functions from [5]. The algorithm and hardware design for the Boolean function are described in Section 3. A specific example of a 24-variable Boolean function is presented in Section 4. Reconfigurability features of the circuit are described in Section 5. Finally, Section 6 concludes the paper with discussion for possible future work.

## 2 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 details, see [1], [11], [5], [14].

**Definition 1.**

- *An $n$-variable function is said to be* balanced *if the output of $f$ is equal to 1 for exactly $2^{n-1}$ inputs.*
- *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 [1], [11], [5], [10], [14] 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 [10] identify the class of Boolean functions which achieve the best possible tradeoff among these properties. Many of these best functions can be constructed by the methods of [1], [11], [5], [9], [14].

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 [5]. The advantage of the method of [5] is that it is simpler than the other methods [11], [9], [14]. We next provide a description of the construction method of [5].

Suppose an $n$-variable function $F(X_n, \ldots, X_1)$ is to be used in the stream cipher system. Following the method of [5], 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

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

- If $\Psi_i = R$, then,

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

The value of $\tau_i$ determines 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 [5]). The actual set of possible values for $\tau_i$ 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 the same and the nonlinearity of $F$ is $2^t$ times the nonlinearity of $h$ (see [5]).

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 $\Psi$ and $\tau$.

**Example 1.** 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 tradeoff among the mentioned parameters [10].

We start the recursive construction using a 5-variable, 1-resilient, degree 3, and nonlinearity 12 function $h$. These values also achieve an optimal tradeoff among the concerned parameters [10].

We provide a choice of $h$ from [9].

$$h(x_5,\ldots,x_1) = (x_5 \oplus 1)(x_4 \oplus 1)(x_1 \oplus x_2) \oplus (x_5 \oplus 1)x_4 (x_1 \oplus x_3) \oplus x_5(x_4 \oplus 1)(x_2 \oplus x_3) \oplus x_5x_4(x_1 \oplus x_2 \oplus x_3).$$

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 $(h,(Q,rc),(R,r),(Q,c))$.

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

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 [5].

In this paper, we will solely be concerned with the implementation of the function $F$ as represented by the sequence $(h,S_1,\ldots,S_t)$. For cryptographic properties we refer the reader to [5], [10].

## 3 Boolean Function Implementation

The crucial problem is to design circuits for the Boolean functions described in Section 2. 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. 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 itself up and, after that, produces a bit of output at each clock cycle. The total delay for obtaining all the pseudorandom 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.

### 3.1 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. 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

Step 2 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 $F_i$. In fact, it is easy to verify that the call $recCompute(F_t(X_n,\ldots,X_1))$ will correctly return the value of $F(X_n,\ldots,X_1)$. 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 $\tau_i$, 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 $b$ are to be unchanged. In the other two cases, the values of $a$ and $b$ need to be updated based on the value of $\tau_i$. 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 2.** We provide an example of the behavior of algorithm $computeTD()$. We use the example of Section 2. 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 $X_8 \oplus b = 1$ 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.

### 3.2 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 the $(t+1)$-th clock pulse.

In the hardware description, we will be manipulating $\Psi_i, \tau_i$ 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 $\tau_i = rc$ accordingly 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).

The pipeline has $t$ internal stages numbered 1 to $t$ (see Fig. 2). 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 $b$.

**Remark 1.** The initial circuit (Fig. 3a) 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

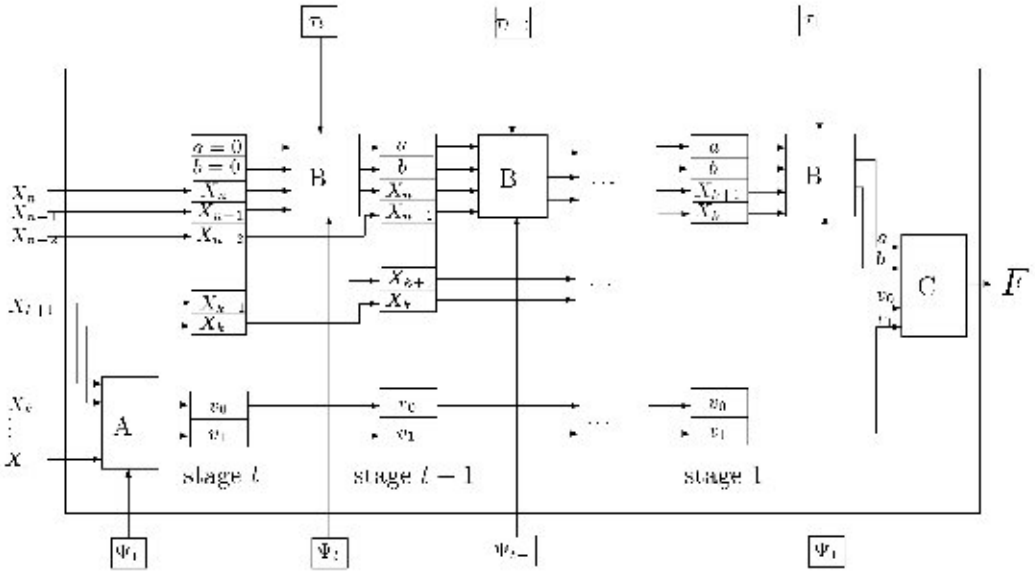


Fig. 2. Pipelined architecture. A: Initial circuit. B: Intermediate circuit. C: Final circuit.

$v_0, v_1$. The size of the look-up table is $2^{k+1}$. We provide some precise parameters in Section 4.

The intermediate stages of the pipeline perform the task of variable interchange and updation of the bits $a$ and $b$ (see Figs. 4 and 5). 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 $2 \times 1$ 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 $\tau_i$ 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 $\tau_i$ is $c$ and the output certainly needs to be complemented. If $\tau_i = 0$ but represents $rc$, then the output also needs to be complemented. But, $\tau_i$ can represent $rc$ only if $i + m_1 \equiv 0 \bmod 2$ (see Section 2). 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 (Fig. 3b) is simple. The $2 \times 1$ MUX and the XOR gate ensure that the output is $a \oplus v_b$.

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 (Fig. 3b). 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 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.

### 3.2.1 Size of the Pipeline

The size of the pipeline in Fig. 2 is the size of the look up table (or circuit) to compute $h$ plus the additional gates and

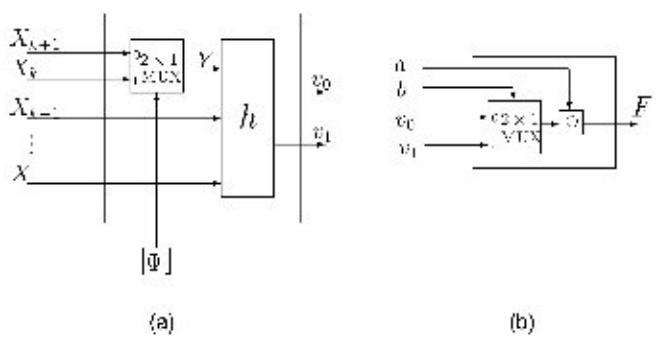


Fig. 3. Initial and final circuits. (a) Initial circuit. (b) Final circuit.

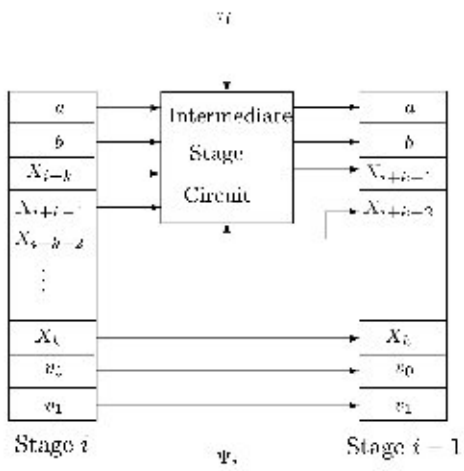


Fig. 4. Transition from stage $i$ to stage $i-1$.

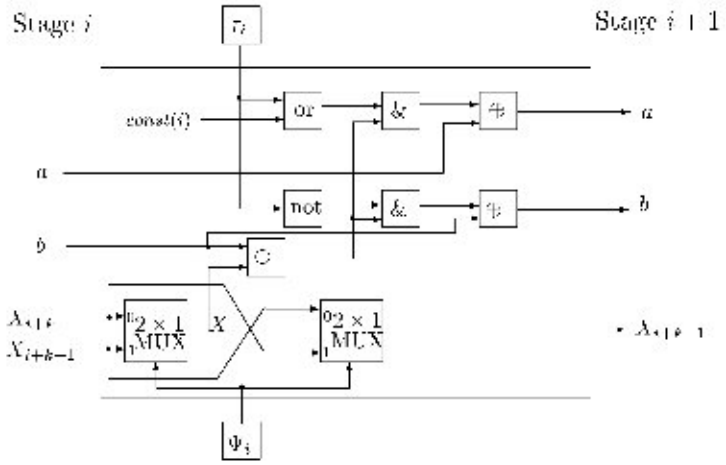


Fig. 5. Intermediate circuit.

flip flops shown in Figs. 3 and 5. We provide an estimate of the size.

1. Each of the initial and final circuits requires a $2 \times 1$ MUX. Additionally, the final circuit requires an XOR gate and the initial circuit requires a flip-flop to store the value of $\Psi_1$.
2. Each of the $i$ $(1 \leq i \leq t)$ intermediate circuits require
   - two $2 \times 1$ 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 $\Psi_i$ and $\tau_i$.

Thus, the total count of the components is as given in Table 1.

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 Table 1 shows, for a moderate number of stages, the total size is quite small.

### 3.2.2 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 LFSRs. 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 pseudorandom bits.

## 4 An Example

We describe 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 [7]. 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 tradeoff among the mentioned parameters [10]. The function $F$ will be built from $h$ using the method described in Section 2. One possible representation for $F$ is

$$(h,(Q,r),(R,rc),(R,c),(Q,rc),(R,r),(Q,c),(R,r),$$
$$(R,rc),(Q,r),(Q,rc),(R,r),(Q,rc),(R,r),(R,rc)).$$

Implementation of $F$ requires a 14-stage pipeline. If $h$ is fixed (i.e., implemented by a combinational circuit), there

TABLE 1
Total Count of the Components

| component | 1-stage | 3-stage | 6-stage | 9-stage | 12-stage | 14-stage |
|---|---|---|---|---|---|---|
| $2 \times 1$ 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 |

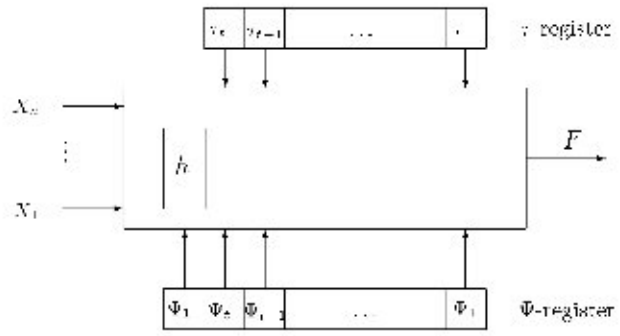


Fig. 6. Reconfigurable Boolean function $F$.

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:

- $2,048$ flip flops to implement the look-up table for $h$.
- $30\ 2 \times 1$ 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.

## 5 Reconfigurability of the Boolean Function

The pipelined architecture provides a natural way to reconfigure the hardware. The values of $\Psi_i$ and $\tau_i$ are stored in flip-flops (see Fig. 2). Consider these flip-flops to be organized into two registers—the $\Psi$-register and the $\tau$-register. The $\Psi$-register stores the values $(\Psi_1, \Psi_t, \ldots, \Psi_1)$ and the $\tau$-register stores the values $(\tau_t, \ldots, \tau_1)$. Note that the bit $\Psi_1$ is required twice—for the initial circuit and also the final circuit—hence, it is more convenient to store the bit $\Psi_1$ twice in the $\Psi$-register.

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

- Change the look up table for the function $h$.
- Change the values of the $\Psi$-register and the $\tau$-register.

This allows the same circuit to be easily reconfigured to implement any function constructed using the method of Section 2 and having a fixed set of parameters (see Fig. 6). *Thus the pipeline architecture is a* **universal circuit** *for the class of secure Boolean functions described in [5].*

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.

## 6 Conclusion

In this paper, we have designed low cost hardware architecture to implement "large" Boolean functions. Our design uses the Boolean functions constructed by the recursive construction of [5]. Several questions remain as to the best possible implementation and the implementation of Boolean functions constructed using other methods [11], [9], [14], [7]. We feel these are future research topics.

## Acknowledgments

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

## References

[1] P. Camion, C. Carlet, P. Charpin, and N. Sendrier, "On Correlation Immune Functions," *Proc. Advances in Cryptology—CRYPTO '91*, pp. 86-100, 1992.
[2] A. Canteaut and M. Trabbia, "Improved Fast Correlation Attacks Using Parity Checks Equations of Weight 4 and 5," *Proc. Advances in Cryptology—EUROCRYPT 2000*, pp. 573-588, 2000.
[3] V. Chepysov, T. Johansson, and B. Smeets, "A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers," *Proc. Fast Software Encryption—FSE 2000*, pp. 181-195, 2001.
[4] T. Johansson and F. Jonsson, "Fast Correlation Attacks through Reconstruction of Linear Polynomials," *Proc. Advances in Cryptology—CRYPTO 2000*, pp. 300-315, 2000.
[5] S. Maitra and P. Sarkar, "Highly Nonlinear Resilient Functions Optimizing Siegenthaler's Inequality," *Proc. Advances in Cryptology—CRYPTO '99*, pp. 198-215, 1999.
[6] W. Meier and O. Stafflebach, "Fast Correlation Attacks on Certain Stream Ciphers," *J. Cryptology*, vol. 1, pp. 159-176, 1989.
[7] E. Pasalic, S. Maitra, T. Johansson, and P. Sarkar, "New Constructions of Correlation Immune and Resilient Boolean Functions Achieving Upper Bounds on Nonlinearity," *Proc. Workshop Coding and Cryptography—WCC 2001*, 2001.
[8] R.A. Rueppel, *Analysis and Design of Stream Ciphers*. Springer-Verlag, 1986.
[9] P. Sarkar and S. Maitra, "Construction of Nonlinear Boolean Functions with Important Cryptographic Properties," *Proc. Advances in Cryptology—EUROCRYPT 2000*, pp. 491-512, 2000.
[10] P. Sarkar and S. Maitra, "Nonlinearity Bounds and Constructions of Resilient Boolean Functions," *Proc. Advances in Cryptology—CRYPTO 2000*, pp. 515-532, 2000.
[11] J. Seberry, X.M. Zhang, and Y. Zheng, "On Constructions and Nonlinearity of Correlation Immune Boolean Functions," *Proc. Advances in Cryptology—EUROCRYPT '93*, pp. 181-199, 1994.
[12] T. Siegenthaler, "Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications," *IEEE Trans. Information Theory*, vol. 30, no. 5, pp. 776-780, Sept. 1984.
[13] T. Siegenthaler, "Decrypting a Class of Stream Ciphers Using Ciphertext Only," *IEEE Trans. Computers*, vol. 34, no. 1, pp. 81-85, Jan. 1985.
[14] Y.V. Tarannikov, "On Resilient Boolean Functions with Maximum Possible Nonlinearity," *Proc. INDOCRYPT 2000*, pp. 19-30, 2000.

**Palash Sarkar** received the Bachelor of Electronics and Telecommunication Engineering degree in 1991 from Jadavpur University, Calcutta, and the Master of Technology in Computer Science degree in 1993 from the Indian Statistical Institute, Calcutta. He received the PhD degree from the Indian Statistical Institute in 1999. Currently, he is an associate professor at the Indian Statistical Institute. His research interests include theoretical computer science and cryptology.

**Subhamoy Maitra** received the Bachelor of Electronics and Telecommunication Engineering degree in 1992 from Jadavpur University, Calcutta, and the Master of Technology in Computer Science degree in 1996 from the Indian Statistical Institute, Calcutta. He received the PhD from the Indian Statistical Institute in 2001. Currently, he is a faculty member at the Indian Statistical Institute. His research interests are in cryptology and digital watermarking.