High-Performance Hardware Implementation for RC4 Stream Cipher

Sourav Sen Gupta, Anupam Chattopadhyay, *Member*, *IEEE*,
Koushik Sinha, *Member*, *IEEE*, Subhamoy Maitra, and Bhabani P. Sinha, *Fellow*, *IEEE*

Abstract—RC4 is the most popular stream cipher in the domain of cryptology. In this paper, we present a systematic study of the hardware implementation of RC4, and propose the fastest known architecture for the cipher. We combine the ideas of hardware pipeline and loop unrolling to design an architecture that produces 2 RC4 keystream bytes per clock cycle. We have optimized and implemented our proposed design using VHDL description, synthesized with 130, 90, and 65 nm fabrication technologies at clock frequencies 625 MHz, 1.37 GHz, and 1.92 GHz, respectively, to obtain a final RC4 keystream throughput of 10, 21.92, and 30.72 Gbps in the respective technologies.

Index Terms—Cryptography, hardware accelerator, high throughput, loop unrolling, pipelining, RC4, stream cipher

1 Introduction

Stream Ciphers are broadly classified into two parts depending on the platform most suited to their implementation; namely software stream ciphers and hardware stream ciphers. RC4 is one of the widely used stream ciphers that is mostly implemented in software. This cipher is used in network protocols such as SSL, TLS, WEP, and WPA. The cipher also finds applications in Microsoft Windows, Lotus Notes, Apple AOCE, Oracle Secure SQL, etc. Though several other efficient and secure stream ciphers have been discovered after RC4, it is still the most popular stream cipher algorithm due to its simplicity, ease of implementation, and speed. In spite of several cryptanalysis attempts on RC4 (see [3], [4], [6], [13], [14], [15], [18], [19], [20], [21], [24] and references therein), the cipher stands secure if used properly.

In this paper, we study several aspects of the hardware implementation of RC4, with respect to its efficient implementation, and present two new hardware designs which allow fast generation of RC4 keystream. The better of the two is the *fastest* known hardware implementation of the cipher till date. To motivate our contribution, we would first like to discuss the basic framework of the cipher. A short note on RC4 follows.

1.1 RC4 Stream Cipher

The RC4 stream cipher was designed by Ron Rivest for RSA Data Security in 1987. It uses S-box S, an array of length N, where each location of S stores 1 byte (typically, $N = 256$). A secret key k of size l bytes is used to scramble this permutation (typically, $5 \leq l \leq 16$). Array K of length N holds the main key, with secret key k repeated as $K[y] = k[y \bmod l]$, for $0 \leq y \leq N-1$.

RC4 has two components, namely the Key Scheduling Algorithm (KSA) and the Pseudo-Random Generation Algorithm (PRGA). The KSA uses the key K to generate a pseudorandom permutation S of $\{0, 1, \ldots, N-1\}$ and PRGA uses this pseudorandom permutation to generate arbitrary number of pseudorandom keystream bytes. The procedures are as in Algorithms 1 and 2, respectively.

Algorithm 1. Key Scheduling Algorithm
1: **procedure** KSA (Secret Key K)
2: Initialize $S \leftarrow \{0, 1, \ldots, N-1\}$ and $j \leftarrow 0$
3: **for** $i = 0, \ldots, N-1$ **do**
4: Increment: $j \leftarrow j + S[i] + K[i]$
5: Swap: $S[i] \leftrightarrow S[j]$
6: **end for**
7: **return** S-box S
8: **end procedure**

Algorithm 2. Pseudo-Random Generation Algorithm
1: **procedure** PRGA (S-box S)
2: Initialize indices: $i \leftarrow 0$, $j \leftarrow 0$
3: **while** TRUE **do**
4: Increment: $i \leftarrow i + 1$, $j \leftarrow j + S[i]$
5: Swap: $S[i] \leftrightarrow S[j]$
6: **output** $Z \leftarrow S[S[i] + S[j]]$
7: **end while**
8: **end procedure**

Any arithmetic addition used in context of RC4 is in general "addition modulo N," unless specified otherwise. The output keystream Z is XOR-ed with the plaintext (byte per byte) to generate the ciphertext at the sender end ($C = M \oplus Z$), and is XOR-ed back with the ciphertext to get back the plaintext at the receiver end ($M = C \oplus Z$).

- *S. Sen Gupta and S. Maitra are with ASU, Indian Statistical Institute, 203 B T Road, Kolkata 700 108, India. E-mail: sg.sourav@gmail.com, subho@isical.ac.in.*
- *A. Chattopadhyay is with MPSoC Architectures, UMIC Research Centre, RWTH Aachen University, Mies-van-der-Rohe Str. 15, Aachen 52074, Germany. E-mail: anupam@umic.rwth-aachen.de.*
- *K. Sinha is with the Hewlett Packard Labs, Bangalore 560 030, India. E-mail: sinha_kou@yahoo.com.*
- *B.P. Sinha is with ACMU, Indian Statistical Institute, 203 B T Road, Kolkata 700 108, India. E-mail: bhabani@isical.ac.in.*

Manuscript received 24 Aug. 2011; revised 3 Jan. 2012; accepted 10 Jan. 2012; published online 16 Jan. 2012.
Recommended for acceptance by G. Qu.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-08-0571.
Digital Object Identifier no. 10.1109/TC.2012.19.

1.2 Motivation

Efficiency in terms of "keystream throughput" has always been a benchmarking parameter for stream ciphers. The efficiency of the RC4 obviously depends on the efficiency of KSA and PRGA. While the KSA invokes a fixed cost for generating the initial pseudorandom state S, the PRGA incurs a variable cost in terms of the number of keystream bytes to be generated. An efficient implementation of RC4 would aim to minimize the cost for per round of KSA and PRGA to provide better throughput. The software implementation of RC4 is simple, and detailed comparison of the software performance of eSTREAM portfolio and RC4 is given in [1].

In this paper, we focus on efficient hardware implementation of the cipher. The main motivation is to test the limits to which RC4, the popular "software" stream cipher, can compete in hardware performance with the state-of-the-art hardware stream ciphers. If it can, we will have a stronger case in support of this time-tested cipher. Furthermore, systematic study of the exploitable fine-grained parallelism aids software developers to attempt better performance in modern parallel processors.

Though there are already a few attempts to propose efficient hardware implementation [5], [9], [11] of RC4, the basic issue remained ignored that the design motivation should be initiated by the following question:

> *"In how many clock cycles can a keystream byte be generated at the PRGA stage in an RC4 hardware?"*

To the best of our knowledge, this line of thought has never been studied and exercised in a disciplined manner in the literature, which in fact, is quite surprising.

A 3-cycle-per-byte efficient implementation of RC4 on a custom pipelined hardware was first proposed by Kitsos et al. [10] in 2003. In the same year, a patent by Matthews Jr. [16] was disclosed, which provided a similar three cycles per byte architecture using hardware pipelining. After a gap of 5 years, another patent by Matthews Jr. [17] was disclosed in 2008, which proposed a new design for RC4 hardware using pipeline architecture. This could increase the efficiency of the cipher to obtain 1-byte-per-cycle in RC4 PRGA. To the best of our knowledge, no further efficiency improvement for RC4 hardware has been proposed in the existing literature.

1.3 Our Contribution

We present two new designs for RC4 hardware targeted toward improved efficiency in terms of its throughput.

Design 1. We propose an RC4 architecture that produces 1 byte per cycle, that is the same throughput as in the design by Matthews [17]. However, our model does not use hardware pipeline approach to obtain this data rate. The main contribution of our work is to take a new look at RC4 hardware design and introduce the idea of *loop unrolling* in this context. We combine consecutive pairs of cycles in a pipelined fashion, and read off the values of one state of the S-box from previous or later rounds of the cipher. To the best of our knowledge, the idea of *loop unrolling* in RC4 has never been exploited in designing an efficient hardware. We present the comprehensive design strategy and analysis of the circuit in Section 2.

Design 2. One may note that Design 1, based on loop unrolling, is completely independent of the design idea of hardware pipelining in case of RC4. Thus, we propose a completely new design of RC4 hardware using *efficient hardware pipeline* and *loop unrolling* simultaneously. This model provides a throughput of 2 bytes per cycle in RC4 PRGA, without losing the clock performance. A detailed account of the design strategy and circuit analysis is presented in Section 3.

Implementation. The implementation of both the designs have been done using VHDL description, synthesized with 90 and 65 nm technologies using Synopsys Design Compiler in topographical mode. Design 2 has also been synthesized with 130 nm technology, for comparison purpose. With strict clock period constraints, we could device a model based on Design 2 that offers the best throughput in hardware implementation:

- 10 Gbps (i.e., 1.25 GBps) on 130 nm technology;
- 21.92 Gbps (i.e., 2.74 GBps) on 90 nm technology; and
- 30.72 Gbps (i.e., 3.84 GBps) on 65 nm technology.

The experimentation with clock period constraints and the final architecture is described in Section 4.

The throughput of Design 2, our final design, is approximately *six times* that of the designs proposed in [10] and [16], and approximately *twice* that of the design proposed in [17] in terms of cycles-per-byte for keystream generation. The area could not be compared with [10] as it is implemented on FPGA, and [16], [17] do not clearly mention any area figures at all. Design 2 also stands quite well in throughput comparison with modern hardware stream ciphers like Grain128, MICKEY, and Trivium, but consumes larger area in general.

1.4 Organization of the Paper

The main content of this paper is organized in the next four sections, summarized as follows:

- Section 2 presents Design 1, a novel RC4 hardware that provides a keystream throughput of 1 byte-per-cycle using the idea of loop unrolling. We detail the design in terms of its components, timing and throughput analysis, and implementation ideas.
- Section 3 presents Design 2, the most efficient hardware design for RC4 that provides a keystream data rate of 2 bytes-per-cycle by combining the strategies of hardware pipeline and loop unrolling. We present the complete schematic and relevant implementation details to prove the claimed efficiency.
- Section 4 presents the final implementation results of Designs 1 and 2, including some intermediate design points. The optimization includes experimentations with strict clock period constraints, and some restructuring of the original model. The final architecture offers the best throughput.
- Section 5 discusses the scope for efficiency improvement using further loop unrolling, and illustrates the limitations regarding this approach. We also discuss relevant issues with storage access in terms of further hardware pipeline, and illustrate the conditions and limitations thereof.

Finally, Section 6 concludes the paper.

TABLE 1
Two Consecutive Loops of RC4 Stream Generation

First Loop	Second Loop
$i_1 = i_0 + 1$	$i_2 = i_1 + 1 = i_0 + 2$
$j_1 = j_0 + S_0[i_1]$	$j_2 = j_1 + S_1[i_2] = j_0 + S_0[i_1] + S_1[i_2]$
Swap $S_0[i_1] \leftrightarrow S_0[j_1]$	Swap $S_1[i_2] \leftrightarrow S_1[j_2]$
$Z_1 = S_1[S_0[i_1] + S_0[j_1]]$	$Z_2 = S_2[S_1[i_2] + S_1[j_2]]$

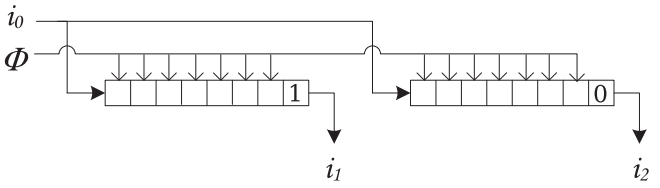


Fig. 1. [Circuit 1] Circuit to compute i_1 and i_2.

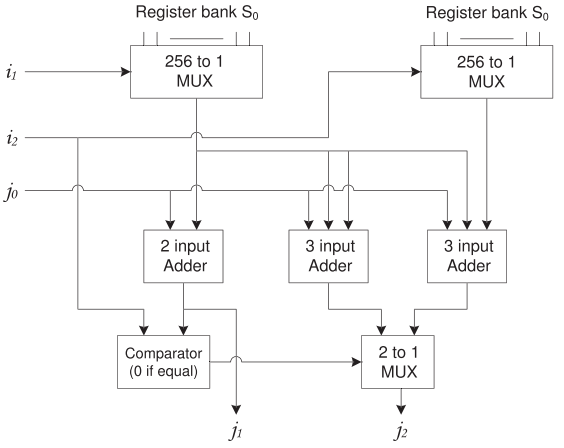


Fig. 2. [Circuit 2] Circuit to compute j_1 and j_2.

2 Design 1: One Byte per Clock

We consider the generation of two consecutive values of Z together, for the two consecutive plaintext bytes to be encrypted. Assume that the initial values of the variables i, j, and S are i_0, j_0, and S_0, respectively. After the first execution of the PRGA loop, these values will be i_1, j_1, and S_1, respectively, and the output byte is Z_1, say. Similarly, after the second execution of the PRGA loop, these will be i_2, j_2, S_2, and Z_2, respectively. Thus, for the first two loops of execution to complete, we have to perform the operations shown in Table 1.

2.1 Design of Individual Components

To store S-array in hardware, we use a bank of 8-bit registers, 256 in total. The output lines of any one of these 256 registers can be accessed through a 256 to 1 Multiplexer (MUX), with its control lines set to the required address i_1, j_1, i_2, or j_2. Thus, we need four such 256 to 1 MUX units to simultaneously read $S[i_1]$, $S[i_2]$, $S[j_1]$, and $S[j_2]$. Before that, let us study how to compute the increments of i and j at each level.

Step 1: Calculation of i_1 and i_2. Incrementing i_0 by 1 and 2 can be done by the same clock pulse applied to two synchronous 8-bit counters. The counter for i_1 is initially loaded with 00000001 and the counter for i_2 is loaded with 00000010, the initial states of these two indices. This serves the purpose for the first two rounds of RC4, in both KSA and PRGA.

Thereafter, in every other cycle, the clock pulse is applied to all the flip-flops except the ones at the LSB position for both the counters, as shown in Fig. 1. This will result in proper increments of i_1 that assumes only the odd values $1, 3, 5, \ldots$, and that of i_2 assuming only the even values $2, 4, 6, \ldots$, as required in RC4. This is assured as the LSB of i_1 will always be 1 and that of i_2 will always be 0, as shown in Fig. 1.

Step 2: Calculation of j_1 and j_2. The values of j_1 and j_2 will be computed and stored in two 8-bit registers. To compute j_1, we need a 2-input parallel adder unit. It may be one using a carry lookahead adder, or one using *scan* operation as proposed by Sinha and Srimani [23], or one using *carry-lookahead-tree* as proposed by Lynch and Swarzlander, Jr. [12]. For computing j_2, there are two special cases:

$$j_2 = j_0 + S_0[i_1] + S_1[i_2] = \begin{cases} j_0 + S_0[i_1] + S_0[i_2] \text{ if } i_2 \neq j_1, \\ j_0 + S_0[i_1] + S_0[i_1] \text{ if } i_2 = j_1. \end{cases}$$

The only change from S_0 to S_1 is the swap $S_0[i_1] \leftrightarrow S_0[j_1]$, and hence we need to check if i_2 is equal to either of i_1 or j_1. Now, i_2 cannot be equal to i_1 as they differ only by 1 modulo 256. Therefore, $S_1[i_2] = S_1[j_1] = S_0[i_1]$ if $i_2 = j_1$, and $S_1[i_2] = S_0[i_2]$ otherwise. In both the cases, three binary numbers are to be added.

Let us denote the kth bit of $j_0, S_0[i_1]$, and $S_1[i_2]$ (either $S_0[i_2]$ or $S_0[i_1]$) by a_k, b_k, and c_k, respectively, where $0 \leq k \leq 7$. We first construct two 9-bit vectors R and C, where the kth bits ($0 \leq k \leq 8$) of R and C are given by

$$R_k = \text{XOR}(a_k, b_k, c_k) \text{ for } 0 \leq k \leq 7,\ R_8 = 0,\ C_0 = 0,$$
$$C_k = a_{k-1}b_{k-1} + b_{k-1}c_{k-1} + c_{k-1}a_{k-1} \text{ for } 1 \leq k \leq 8.$$

In RC4, all additions are done at modulo 256. Hence, we can discard the 9th bit ($k = 8$) of the vectors R, C while adding them together, and carry out normal 8-bit parallel addition considering $0 \leq k \leq 7$. Therefore, one may add R and C by a parallel full adder as used for j_1. The circuit to compute j_1 and j_2 is as shown in Fig. 2.

Step 3: Swapping the S values. In Table 1, the two swap operations in the third row results in one of the following eight possible data transfer requirements among the registers of the S-register bank, depending on the different possible values of i_1, j_1, i_2, and j_2. We have to check if i_2 and j_2 can be equal to i_1 or j_1 (we only know that $i_2 \neq i_1$). All the cases in this direction can be listed as in Table 2. A more detailed explanation for each case is presented as follows:

Case 1: $i_2 \neq j_1$ and $j_2 \neq i_1$ and $j_2 \neq j_1$.

These data transfers are symbolically represented by the following permutation on data in S_0

$$\begin{pmatrix} i_2 & j_2 \\ j_2 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}.$$

This involves four simultaneous registers to register data transfers, as follows: $S_0[i_1] \rightarrow S_0[j_1]$, $S_0[j_1] \rightarrow S_0[i_1]$, $S_0[i_2] \rightarrow S_0[j_2]$ and $S_0[j_2] \rightarrow S_0[i_2]$.

Case 2: $i_2 \neq j_1$ and $j_2 \neq i_1$ and $j_2 = j_1$.

In this case, the data transfers are represented by

$$\begin{pmatrix} i_2 & j_1 \\ j_1 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}.$$

TABLE 2
Different Cases for the Register-to-Register Transfers in the Swap Operation

#	Condition	Register-to-Register Transfers
1	$i_2 \neq j_1$ & $j_2 \neq i_1$ & $j_2 \neq j_1$	$S_0[i_1] \rightarrow S_0[j_1]$, $S_0[j_1] \rightarrow S_0[i_1]$, $S_0[i_2] \rightarrow S_0[j_2]$, $S_0[j_2] \rightarrow S_0[i_2]$
2	$i_2 \neq j_1$ & $j_2 \neq i_1$ & $j_2 = j_1$	$S_0[i_1] \rightarrow S_0[i_2]$, $S_0[i_2] \rightarrow S_0[j_1] = S_0[j_2]$, $S_0[j_1] \rightarrow S_0[i_1]$
3	$i_2 \neq j_1$ & $j_2 = i_1$ & $j_2 \neq j_1$	$S_0[i_1] \rightarrow S_0[j_1]$, $S_0[i_2] \rightarrow S_0[i_1] = S_0[j_2]$, $S_0[j_1] \rightarrow S_0[i_2]$
4	$i_2 \neq j_1$ & $j_2 = i_1$ & $j_2 = j_1$	$S_0[i_1] \rightarrow S_0[i_2]$, $S_0[i_2] \rightarrow S_0[i_1] = S_0[j_1] = S_0[j_2]$
5	$i_2 = j_1$ & $j_2 \neq i_1$ & $j_2 \neq j_1$	$S_0[i_1] \rightarrow S_0[j_2]$, $S_0[j_2] \rightarrow S_0[j_1] = S_0[i_2]$, $S_0[j_1] \rightarrow S_0[i_1]$
6	$i_2 = j_1$ & $j_2 \neq i_1$ & $j_2 = j_1$	$S_0[i_1] \rightarrow S_0[j_1] = S_0[i_2] = S_0[j_2]$, $S_0[j_1] \rightarrow S_0[i_1]$
7	$i_2 = j_1$ & $j_2 = i_1$ & $j_2 \neq j_1$	Identity permutation, no data transfer.
8	$i_2 = j_1$ & $j_2 = i_1$ & $j_2 = j_1$	Impossible, as it implies $i_1 = i_2 = i_1 + 1$.

This involves three data transfers: $S_0[i_1] \rightarrow S_0[i_2]$, $S_0[i_2] \rightarrow S_0[j_1] = S_0[j_2]$, and $S_0[j_1] \rightarrow S_0[i_1]$.

Case 3: $i_2 \neq j_1$ and $j_2 = i_1$ and $j_2 \neq j_1$.

In this case, the data transfers are represented by

$$\begin{pmatrix} i_2 & i_1 \\ i_1 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}.$$

This involves three data transfers: $S_0[i_1] \rightarrow S_0[j_1]$, $S_0[i_2] \rightarrow S_0[i_1] = S_0[j_2]$ and $S_0[j_1] \rightarrow S_0[i_2]$.

Case 4: $i_2 \neq j_1$ and $j_2 = i_1$ and $j_2 = j_1$.

In this case, the data transfers are represented by

$$\begin{pmatrix} i_2 & i_1 \\ i_1 & i_2 \end{pmatrix} \circ \begin{pmatrix} i_1 & i_1 \\ i_1 & i_1 \end{pmatrix}.$$

This involves two data transfers: $S_0[i_1] \rightarrow S_0[i_2]$ and $S_0[i_2] \rightarrow S_0[i_1] = S_0[j_1] = S_0[j_2]$.

Case 5: $i_2 = j_1$ and $j_2 \neq i_1$ and $j_2 \neq j_1$.

In this case, the data transfers are represented by

$$\begin{pmatrix} j_1 & j_2 \\ j_2 & j_1 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}.$$

This involves three data transfers: $S_0[i_1] \rightarrow S_0[j_2]$, $S_0[j_2] \rightarrow S_0[j_1] = S_0[i_2]$, and $S_0[j_1] \rightarrow S_0[i_1]$.

Case 6: $i_2 = j_1$ and $j_2 \neq i_1$ and $j_2 = j_1$.

In this case, the data transfers are represented by

$$\begin{pmatrix} j_1 & j_1 \\ j_1 & j_1 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}.$$

This involves two data transfers: $S_0[i_1] \rightarrow S_0[j_1] = S_0[i_2] = S_0[j_2]$ and $S_0[j_1] \rightarrow S_0[i_1]$.

Case 7: $i_2 = j_1$ and $j_2 = i_1$ and $j_2 \neq j_1$.

In this case, the data transfers are represented by

$$\begin{pmatrix} j_1 & i_1 \\ i_1 & j_1 \end{pmatrix} \circ \begin{pmatrix} i_1 & j_1 \\ j_1 & i_1 \end{pmatrix}.$$

This is the identity permutation, and hence it does not involve any data transfer.

Case 8: $i_2 = j_1$ and $j_2 = i_1$ and $j_2 = j_1$.

This case cannot occur, as it implies $i_1 = i_2$, which is impossible because $i_2 = i_0 + 2 = i_1 + 1$.

After the swap operation is completed successfully, one obtains S_2 from S_0. From the point of view of the receiving registers (in the S-register bank) in case of the above-mentioned register-to-register transfers, we can summarize the cases as follows:

- $S_2[i_1]$ receives data from $S_0[i_1], S_0[j_1]$ or $S_0[i_2]$,
- $S_2[j_1]$ receives from $S_0[i_1], S_0[j_1], S_0[i_2]$ or $S_0[j_2]$,
- $S_2[i_2]$ receives from $S_0[i_1], S_0[j_1], S_0[i_2]$ or $S_0[j_2]$,
- $S_2[j_2]$ receives from $S_0[i_1], S_0[i_2]$ or $S_0[j_2]$.

In view of the above discussions, the input data (1 byte) for each of the 256 registers in the S-register bank will be taken from the output of an 8 to 1 MUX unit, whose data inputs are taken from $S_0[i_1], S_0[j_1], S_0[i_2], S_0[j_2]$, and the control inputs are taken from the outputs of three comparators comparing 1) i_2 and j_1, 2) j_2 and i_1, 3) j_2 and j_1. The circuit to realize the swap is as shown in Fig. 3.

For the simultaneous register-to-register data transfer during the swap operation, we propose the use of Master-Slave JK flip-flops to construct the registers in the S-register bank. This way, the read and write operations will respect the required order of functioning, and the synchronization can be performed at the end of each clock cycle to update the S-state.

Step 4: Calculation of Z_1 and Z_2. The main idea to get the most out of loop unrolling in RC4 is to completely bypass the generation of S_1, and move directly from S_0 to S_2, as discussed right before. However, note that we require the state S_1 for computing the output byte $Z_1 = S_1[S_0[j_1] + S_0[i_1]]$. We apply the following trick of cross-loop look back to resolve this issue.

In step 4 of Algorithm 2, we can rewrite the output $Z_1 = S_1[S_1[i_1] + S_1[j_1]] = S_1[S_0[j_1] + S_0[i_1]]$ as

$$Z_1 = \begin{cases} S_2[i_2], & \text{if } S_0[j_1] + S_0[i_1] = j_2, \\ S_2[j_2], & \text{if } S_0[j_1] + S_0[i_1] = i_2, \\ S_2[S_0[j_1] + S_0[i_1]], & \text{otherwise.} \end{cases}$$

Computing Z_1 involves adding $S_0[i_1]$ and $S_0[j_1]$ first, which can be done using a 2-input parallel adder. The 256 to 1 MUX, which is used to extract appropriate data from S_2, will be controlled by another 4 to 1 MUX. This 4 to 1 MUX is in turn controlled by the outputs of two comparators comparing 1) $S_0[j_1] + S_0[i_1]$ and i_2, and 2) $S_0[j_1] + S_0[i_1]$ and j_2, as illustrated in the circuit of Fig. 4.

Computation of Z_2, however, involves adding $S_1[i_2]$, $S_1[j_2]$, as in the following formula:

$$Z_2 = S_2[S_2[i_2] + S_2[j_2]] = S_2[S_1[j_2] + S_1[i_2]].$$

In this case, we unwrap one cycle of RC4 and gather the values of $S_1[i_2]$ and $S_1[j_2]$ from the S_0 state. $S_1[i_2]$ and $S_1[j_2]$ receive the values from the appropriate registers of S_0 as given below, depending on the following conditions:

- $i_2 \neq j_1, j_2 \neq i_1, j_2 \neq j_1 : S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_2]$,
- $i_2 \neq j_1, j_2 \neq i_1, j_2 = j_1 : S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[i_1]$,
- $i_2 \neq j_1, j_2 = i_1, j_2 \neq j_1 : S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_1]$,

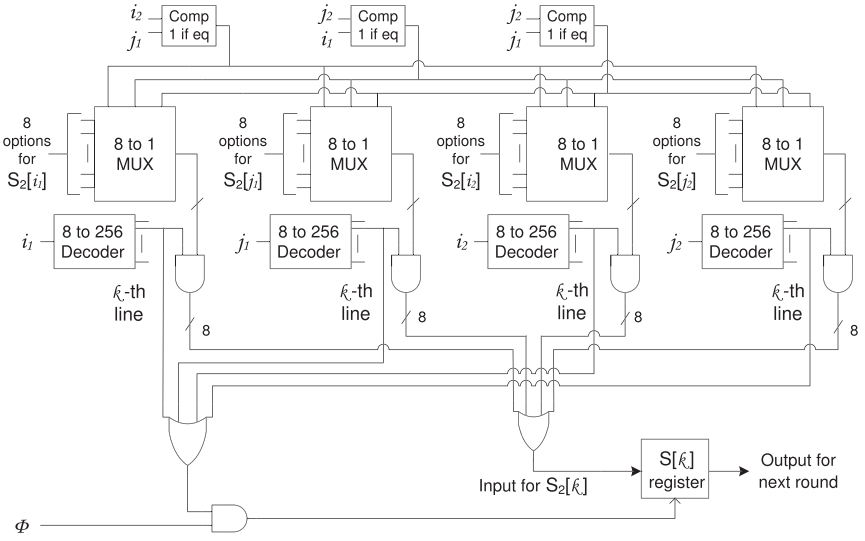


Fig. 3. [Circuit 3] Circuit to swap S values. (Data lines shown only for a fixed k.)

- $i_2 \neq j_1, j_2 = i_1, j_2 = j_1 : S_1[i_2] = S_0[i_2], S_1[j_2] = S_0[j_1]$,
- $i_2 = j_1, j_2 \neq i_1, j_2 \neq j_1 : S_1[i_2] = S_0[i_1], S_1[j_2] = S_0[j_2]$,
- $i_2 = j_1, j_2 \neq i_1, j_2 = j_1 : S_1[i_2] = S_0[i_1], S_1[j_2] = S_0[i_1]$,
- $i_2 = j_1, j_2 = i_1, j_2 \neq j_1 : S_1[i_2] = S_0[i_1], S_1[j_2] = S_0[j_1]$.

These conditions can be realized using an 8 to 1 MUX unit controlled by the outputs of three comparators comparing 1) i_2 and j_1, 2) j_2 and i_1, 3) j_2 and j_1. We can use the same control lines as in case of the swapping operation. The circuit is as shown in Fig. 5.

2.2 Timing Analysis

The timing analysis for the complete PRGA circuit (shown in Fig. 6) is as shown in the three-stage pipeline diagram of Fig. 7. We illustrate the first two iterations, and the rest falls along similar lines. The combinational logics operate between the clock pulses and all read, swap, and increment operations are done at the trailing edges of the clock pulses. The first two bytes Z_1, Z_2 are obtained at the end of the third clock cycle and the next two bytes Z_3, Z_4 are obtained at the fifth clock cycle. A detailed explanation follows.

2.3 The Complete Circuit

The complete circuit diagram for the PRGA algorithm of Design 1 is shown in Fig. 6. We shall henceforth denote the clock by ϕ and its cycles numbered as ϕ_1, ϕ_2, etc., where ϕ_0 refers to the clock pulse that initiates PRGA.

In Fig. 6, L_i denote the latches operated by the trailing edge of ϕ_{2n+i}, i.e., the $(2n+i)$th cycle of the master clock ϕ where $n \geq 0$. For example, the latches labeled L_1 (four of them) are released at the trailing edge of $\phi_1, \phi_3, \phi_5, \ldots$ and the latches labeled L_2 (eight of them) are released at the trailing edge of $\phi_2, \phi_4, \phi_6, \ldots$ etc. In the final implementation, these latches have been replaced by edge-triggered flip-flops which operate at the trailing edge of the clock. Now, we generalize our previous observation to state the following.

Efficiency of PRGA in Design 1:

The hardware proposed for the PRGA stage of RC4 in Design 1, as shown in Fig. 6, produces "one byte per clock" after an initial delay of two clock cycles.

Let us call the stage of the PRGA circuit shown in Fig. 6 the nth stage. This actually denotes the nth iteration of our model, which produces the output bytes Z_{n+1} and Z_{n+2}. The first block (Circuit 1) operates at the trailing edge of ϕ_n, and increments i_n to i_{n+1}, i_{n+2}. During cycle ϕ_{n+1}, the combinational part of Circuit 2 operates to produce j_{n+1}, j_{n+2}. The trailing edge of ϕ_{n+1} releases the latches of type L_1, and activates the swap circuit (Circuit 3). The combinational logic of the swap circuit functions during cycle ϕ_{n+2} and the actual swap operation takes place at the trailing edge of ϕ_{n+2} to produce S_{n+2} from S_n. Simultaneously, the latch of type L_2 is released to activate the Circuits 4 and 5. The combinational logic of these two circuits operate during ϕ_{n+3}, and we get the outputs Z_{n+1} and Z_{n+2} at the trailing edge of ϕ_{n+3}.

This complete block of architecture performs in a cascaded pipeline fashion, as the indices i_2, j_2 and the

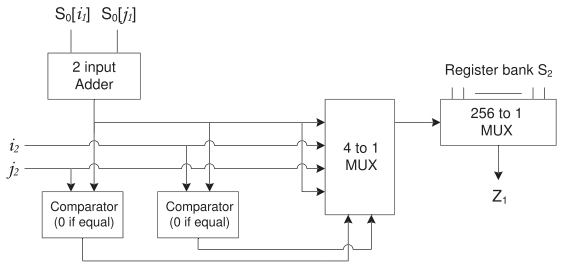


Fig. 4. [Circuit 4] Circuit to compute Z_1.

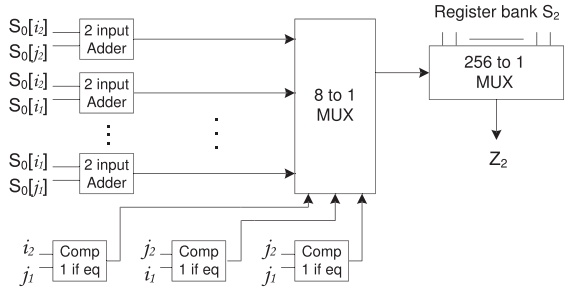


Fig. 5. [Circuit 5] Circuit to compute Z_2.

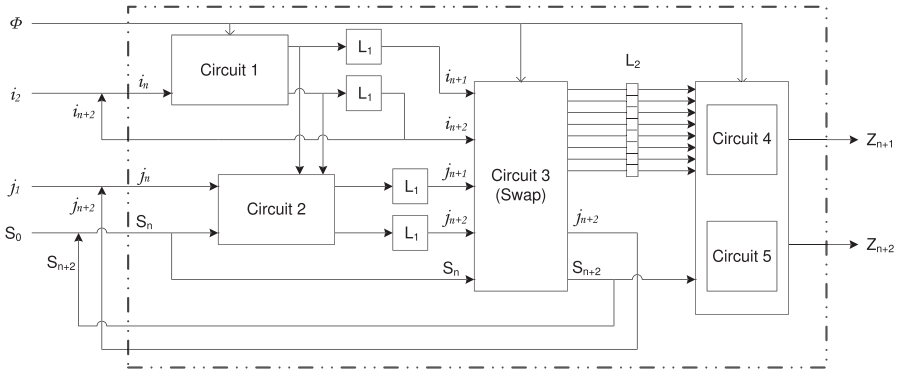


Fig. 6. Circuit for PRGA stage of the proposed RC4 architecture (Design 1).

state S_{n+2} are fed back into the system at the end of ϕ_{n+2} (actually, i_{n+2} is fed back at the end of ϕ_{n+1} to allow for the increments at the trailing edge of ϕ_{n+2}). The operational gap between two iterations (e.g., nth and $(n+2)$th) of the system is thus two clock cycles (e.g., ϕ_n to ϕ_{n+2}), and we obtain two output bytes per iteration.

Hence, the PRGA architecture of Design 1, as shown in Fig. 6, produces $2N$ bytes of output stream in N iterations, over $2N$ clock cycles. Note that the initial clock pulse ϕ_0 is an extra one, and the production of the output bytes lag the feedback cycle by one clock pulse in every iteration (e.g., ϕ_{n+3} in case of nth iteration). Therefore, our model practically produces $2N$ output bytes in $2N$ clock cycles, that is "one byte per clock," after an initial lag of two clock cycles.

Issues with KSA. Note that the general KSA routine runs for 256 iterations to produce the initial permutation of the S-box. Moreover, the steps of KSA are quite similar to the steps of PRGA, apart from the following:

- Calculation of j involves key K along with S and i.
- Computing Z_1, Z_2 is neither required nor advised.

We propose the use of our *loop-unrolled* PRGA architecture (Fig. 6) for the KSA as well, with some minor modifications, as follows:

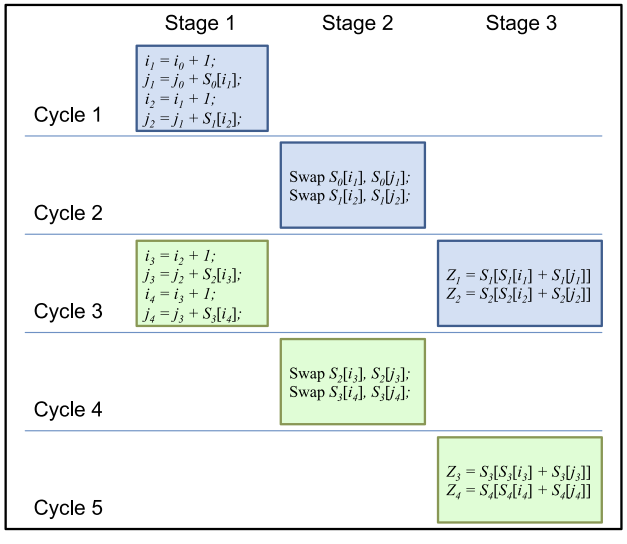


Fig. 7. Pipeline structure for the proposed Design 1.

1. *K-Register bank.* Introduce a new register bank for key K. It will contain l number of 8-bit registers, where $8 \leq l \leq 15$ in practice.
2. *K-Register MUX.* To read key values $K[i_1 \bmod l]$ and $K[i_2 \bmod l]$ from the K-registers, we introduce two 16 to 1 multiplexer unit. The first l input lines of this MUX will be fed data from registers $K[0]$ to $K[l-1]$, and the rest $(16-l)$ inputs can be left floating (recall that $8 \leq l \leq 15$). The control lines of these MUX units will be $i_1 \bmod l$ and $i_2 \bmod l$, respectively, and hence the floating inputs will never be selected.
3. *Modular counters.* To obtain modular indices $i_1 \bmod l$ and $i_2 \bmod l$, we incorporate two modular counters (modulo l) for the indices. These are synchronous counters and the one for i_2 will have no clock input for the LSB position, similar to Fig. 1.
4. *Extra 2-Input parallel adders.* Two 2-input parallel adders are appended to Fig. 2 for adding $K[i_1 \bmod l]$ and $K[i_2 \bmod l]$ to j_1 and j_2, respectively.
5. *No outputs.* Circuits of Figs. 4 and 5 are removed from the overall structure, so that no output byte is generated during KSA. If any such byte is generated, the key K may be compromised.

Using this modified hardware configuration, one can implement two rounds of KSA in two clock cycles, that is "one round per clock," after an initial lag of one cycle. Total time required for KSA is $256 + 1 = 257$ clock cycles.

2.4 Implementation

We have implemented Design 1, the proposed structure for RC4 stream cipher, using synthesizable VHDL description. The S-register box and K-register box are implemented as array of master-slave flip-flops, and are synthesized as standard-cell memory architecture (register-based implementation). The entire VHDL code consists of approximately 1,500 lines.

A major area impact of the circuit originates from the large number of accesses to the S-box and the K-box from the KSA and PRGA circuit. Since the PRGA and KSA will not run in parallel, we shared the read and write ports of S-box and K-box between PRGA and KSA. From KSA, one read access to K-box, two read accesses to S-box, and two write accesses to S-box are needed. From PRGA, six read accesses to S-box and four write accesses to S-box are needed. The two read accesses correspond to simultaneous generation of two Z values at the last step of PRGA. The

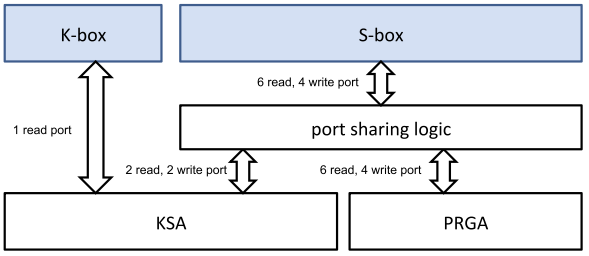


Fig. 8. Access sharing of KSA and PRGA in Design 1.

four read and write accesses correspond to the *double swap* operation. While sharing the mutually exclusive accesses, all the accesses from KSA can be merged among the PRGA accesses. Therefore, the total number of read ports to K-box is 1, the total number of read ports to S-box is 6, and the total number of write ports to S-box is 4. This sharing of storage access is as shown in Fig. 8.

The VHDL code is synthesized with 90 and 65 nm fabrication technologies using Synopsys Design Compiler in topographical mode. The detailed implementation results are presented in Section 4.

2.5 Comparison with Existing Designs

Let us compare the proposed design with the ones that existed for RC4 hardware till date. We only consider existing designs that are focused toward improved throughput, and not any other hardware considerations.

Kitsos et al. [10] and Matthews Jr. [16]. Combining our KSA and PRGA architectures, we can obtain $2N$ output-stream bytes in $2N + 259$ clock cycles, counting the initial delay of 1 cycle for KSA and 2 cycles for PRGA. The hardware implementation of RC4 described in [10] or [16] provides an output of N bytes in $3N + 768$ clock cycles. A formal comparison of the timings is shown in Table 6. One can easily observe that for large N, the throughput of our RC4 architecture is three times compared to that of the designs proposed in [10] and [16].

In terms of the area, exact comparison with [10] is not possible since, we do not have access to the FPGA board for which the area figures of [10] is reported. Considering the design idea, both [10] and [16] modeled their storage using block RAMs. This implementation restricts the number of port accesses per cycle. To overcome that, three 256-byte dual-port RAM blocks are used in [10]. Even then, the design requires three cycles to produce 1 byte of data. An improved design is reported in [16] where only two 256-byte dual-port RAM blocks are used. It may be noted that we have utilized register-based storage for the S and K arrays instead of RAM. This is because a RAM-based storage would incorporate port-access restrictions and latency issues, resulting in a compromise of throughput. An alternative technique to maintain the high throughput with RAM-based implementation may be partitioning the arrays according to the accesses, and optimize accordingly.

Matthews Jr. [17]. This design proposes a 1-byte-per-cycle design of RC4 hardware using a technique that is different from our approach. It achieves the claimed throughput by means of hardware pipeline, and instead of two iterations per cycle, one iteration per cycle in PRGA is performed. The pipeline design is as shown in Table 3 (same as [17, Table 1]). In terms of throughput, this design provides 1-cycle-per-byte output in PRGA and completes KSA in 256 cycles, with an initial lag of three cycles due to the 4-stage pipeline structure (as in Table 3). Thus, N bytes of output is produced in approximately $N + 259$ clock cycles, which is comparable to the performance of Design 1. Detailed comparative results are presented in Table 6.

The obvious advantage of this design is its compactness. It provides the same throughput without resorting to loop unrolling. We have studied this design closely and have implemented similar idea on our own to understand the time and area constraints better. This is required since the documentation in [17] does not report area or timing figures corresponding to any technology node. Instead a figurative summary of the logic structure is reported. We implemented a similar design (named Pipelined-A) which helps us further to fuse the ideas of loop unrolling and hardware pipelining to obtain a better design, as reported in Section 3. It is obvious from the pipeline design proposed in [17] (as shown in Table 3) that, at least three read and two write accesses per cycle is required when the pipeline is full. This is exactly what we achieve after optimization of storage access in Pipelined-C, described in Section 3. However, [17] does neither report the hardware implementation of KSA, nor the port access sharing between PRGA and KSA.

Pipelined-A. Our first design motivated by hardware pipelining is an architecture pipelined in two stages, as in Fig. 9. While [17] proposed a deep pipelining with bypass and data forwarding, our schemes allow a simpler 2-stage pipelining with the same throughput of 1-byte-per-cycle and similar memory port restrictions. The first pipeline stage is devoted for calculation of j and performing the swap, the second pipeline stage computes the value of Z. To minimize the read and write accesses to S-box, the index to be used for

TABLE 3
Pipeline Stages for the Design Proposed in [17]

1st stage	$i_1 = i_0 + 1$	$i_2 = i_1 + 1$	$i_3 = i_2 + 1$			
2nd stage		Read $S[i_1]$ Store $S[i_1]$ into SI $j_1 = j_0 + S[i_1]$	Read $S[i_2]$ Store $S[i_2]$ into SI $j_2 = j_1 + S[i_2]$	Read $S[i_3]$ Store $S[i_3]$ into SI $j_3 = j_2 + S[i_3]$		
3rd stage			Read $S[j_1]$ Store $S[j_1]$ in SJ $t_1 = SI + S[j_1]$ Write SI into $S[j_1]$	Read $S[j_2]$ Store $S[j_2]$ in SJ $t_2 = SI + S[j_2]$ Write SI into $S[j_2]$	Read $S[j_3]$ Store $S[j_3]$ in SJ $t_3 = SI + S[j_3]$ Write SI into $S[j_3]$	
4th stage				Read $S[t_1]$ Store $S[t_1]$ into K Write SJ into $S[i_1]$	Read $S[t_2]$ Store $S[t_2]$ into K Write SJ into $S[i_2]$	Read $S[t_3]$ Store $S[t_3]$ into K Write SJ into $S[i_3]$

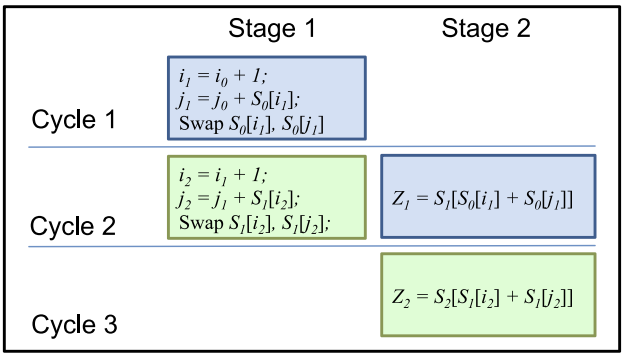


Fig. 9. 1-byte/cycle by Hardware Pipeline (Pipelined-A).

second pipeline is computed at the first stage itself. Note that, the index computation at first stage does not alter the result as $S[i]$ and $S[j]$ are swapped, thus the addition result $S[i] + S[j]$ remains intact.

With the aforementioned structure, the pipeline in PRGA circuit is considerably simplified with respect to Design 1. We further study the circuit in order to improve its area and timing. To that effect, we first reorganized the KSA circuit to merge two iterations of key generation in one cycle. The benefit of this will be to have the S-box created in 128 cycles instead of 256 cycles.

This is done in similar fashion to the unrolling of KSA iterations as per the design discussed in Section 2. The logic for the two consecutive KSA loops in shown in Table 4, and the design idea follows that of Design 1. For this design, the K-box is optimized away as it had constant reset inputs. In the improvements discussed later, the K-box values are controlled from external input. We synthesized the circuit without port sharing, using 90 nm technology at a strict clock frequency, and the synthesis results are as presented in Section 4. The throughput is the same as that of Design 1.

One may prefer Pipelined-A over Design 1 because of its obvious simplicity. However, we look into the possibility of improving the architecture even further.

3 Design 2: Two Bytes per Clock

In this section, we present a novel design for RC4 hardware which provides the best throughput till date. We have already proposed a design in Section 2 to obtain a throughput of 1-byte-per-cycle. We have also designed and studied a hardware pipeline architecture that provides the same. Now we will analyze the two models from a more detailed implementation point of view for potential improvement in the design.

3.1 Area Optimization in Pipelined-A

Note that in the hardware pipeline-based Pipelined-A, as discussed in the previous section (Fig. 9), we had fused the idea of loop unrolling to merge two consecutive rounds of KSA. As a result, the number of read accesses to K-box from KSA grew to 2. The number of read and write accesses from KSA to S-box are both 4 due to the double swap in one cycle.

Pipelined-B. We modified Pipelined-A to implement access sharing (read/write) for S-box between KSA and PRGA. In case of Pipelined-A, KSA contains four read, four write accesses and PRGA contains three read and two write accesses to the S-box. Naturally, all the accesses from PRGA can be shared with accesses from KSA, resulting in total four read and four write accesses. The synthesis indicated a compact circuit with the same throughput.

TABLE 4
Two Consecutive Loops of RC4 Key Scheduling

First Loop	Second Loop
$i_1 = i_0 + 1$	$i_2 = i_1 + 1 = i_0 + 2$
$j_1 = j_0 + S_0[i_1] + K[i_1]$	$j_2 = j_1 + S_1[i_2] + K[i_2]$ $= j_0 + S_0[i_1] + S_1[i_2] + K[i_2]$
Swap $S_0[i_1] \leftrightarrow S_0[j_1]$	Swap $S_1[i_2] \leftrightarrow S_1[j_2]$

Pipelined-C. Another idea, exploited to reduce the circuit size further, is to perform only one iteration of KSA per cycle. In this approach, KSA will require 256 cycles to initialize the S-box. However, the number of both read and write accesses to S-box will become 2 per cycle. By applying access sharing on top of that, total number of read and write accesses to S-box is reduced to 3 and 2, respectively. Furthermore, the number of read accesses to K-box also dropped to 1. The synthesis of this circuit also indicated a more compact design.

We observed a sharp reduction in S-box and KSA areas for Pipelined-C, in comparison with the previous designs. The reduction in the area for S-box is most prominent as Pipelined-C directly reduces the area requirements for the address decoders and multiplexers, due to less number of S-box access ports. However, the reduction in K-box access ports costs us 256 cycles for KSA, instead of 128 as in the previous two designs. Later, we shall present final synthesis results for all the designs to compare the mutual pros and cons.

Next, we extend our analysis on the area and timing improvements of hardware pipelining to propose a completely new and considerably improved design.

3.2 Design Idea for 2 Bytes-Per-Cycle

Recall the design based on the hardware pipeline approach (Pipelined-A) as shown in Fig. 9, and also the main idea of Section 2 (Design 1) where a completely new approach to RC4 hardware design gave rise to a one-byte-per-clock architecture based on the technique of loop unrolling. In case of hardware pipeline, we used the idea of pipeline registers to control the read-after-write sequence during S-box swaps, and that resulted in a natural two stage pipeline model for RC4 PRGA. In case of loop unrolling, this idea of pipeline registers was not used at all, but the same throughput was obtained by merging two consecutive rounds of RC4 PRGA. Two obvious questions in this direction are:

- Can these two techniques be combined?
- Will that provide any better result at all?

This was the main motivation behind the next design, where we answer both the questions in affirmative.

We fused the idea of 2-stage hardware pipeline with that of loop unrolling to generate an RC4 circuit with maximum speed/throughput till date. This is obtained for the case with 2-stage PRGA pipeline and KSA circuit with double

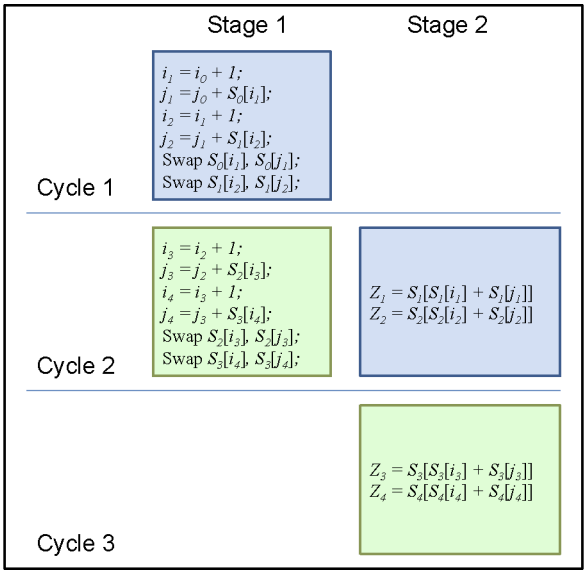


Fig. 10. Pipeline structure for 2-byte-per-clock design.

iterations per cycle in each case. In this case, 128 cycles for S-box preparation is needed at the KSA stage, and then onward after a gap of one cycle, 2 bytes per cycle are generated for encryption purposes. This shows significant improvements over the previously published RC4 implementations in the literature.

3.3 Pipeline Structure

For an intuitive pipeline architecture and timing analysis of this new design, one needs to recall the pipeline structures of the individual designs based on loop unrolling and hardware pipelining. Notice that the loop unrolling approach of Design 1 used a 3-stage pipeline, as in Fig. 7:

1. Increment of indices i and j.
2. Swap operation in the S-register.
3. Read output byte Z from S-register.

Alternatively, the hardware pipeline idea of Pipelined-A, as in Fig. 9, achieved the same using a 2-stage pipeline:

1. Increment of i, j, and Swap in the S-register.
2. Read output byte Z from S-register.

In the design for 2 bytes per clock cycle throughput, we propose a fusion of the two ideas, to generate a 2-stage pipeline architecture, as shown in Fig. 10.

The double swap operation starts at Stage 1 in this case, and takes the help of pipeline registers to maintain the read-after-write ordering during the swap operations. This part of the operation is same as in the hardware pipelined approach for one-byte-per-clock design (Pipelined-A/B/C). The Z values are read from the S-registers after the completion of the double-swap, and using the loop unrolling logic from the first one-byte-per-clock design. That is, two consecutive values of the output byte Z are read from the same state S by using some suitable combinational logic. Similarly, the increment of two consecutive i and j values are done simultaneously using the combinational logic of the original one-byte-per-clock design.

This design obviously provides two output bytes per clock cycle, after an initial lag of one cycle, as is evident

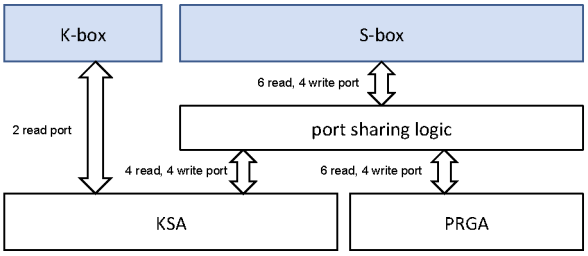


Fig. 11. Read-Write access sharing of KSA and PRGA.

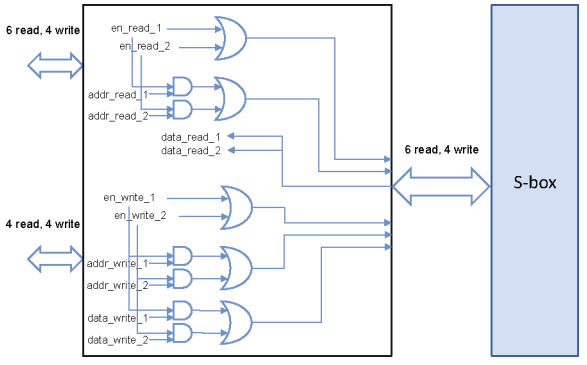


Fig. 12. Port-sharing of KSA and PRGA for S-box access.

from Fig. 10. Thus, for the generation of $2N$ keystream bytes in RC4 PRGA, the circuit has to operate for just $N + 1$ clock cycles, thereby producing an asymptotic throughput of *2-bytes-per-clock*. In KSA, we simply omit Stage 2 of the pipeline structure, and obtain a speed of two KSA rounds per clock cycle. Thus, KSA is completed within 128 cycles in this design. In Section 5, we will discuss about the issues with further pipelining to obtain better throughput using a similar architecture.

3.4 Designing the Storage Access

Since the PRGA and KSA will not run in parallel, we shared the read and write ports of S-box and K-box between PRGA and KSA. From KSA, two read accesses to K-box are required as two loops are merged per cycle. Further, four read and four write accesses to S-box are needed for the double swap operation. From PRGA, six read accesses to S-box and four write accesses to S-box are required. The two read accesses correspond to simultaneous generation of two Z values at the last stage of PRGA, while the four read and four write accesses correspond to the double swap operation. While sharing the mutually exclusive accesses, all the accesses from KSA can be merged among the PRGA accesses. Therefore, the total number of read ports to K-box is 2, the total number of read ports to S-box is 6 and the total number of write ports to S-box is 4. This port-sharing logic is as shown in Fig. 11.

The port-sharing logic reduces the multiplexer area significantly. It should be noted that the multiplexer logic to the register banks claims the major share of area. The port-sharing logic, as shown in Fig. 11, reduces a major share of this combinational area in our design. In Fig. 12, we illustrate the circuit structure for the port-sharing logic that operates with the S-box during KSA and PRGA. Note that K-box accesses are only made by KSA, and there is no question of port sharing in that context. In Fig. 12, we illustrate the port-sharing logic, using just one read and one write port for simplicity.

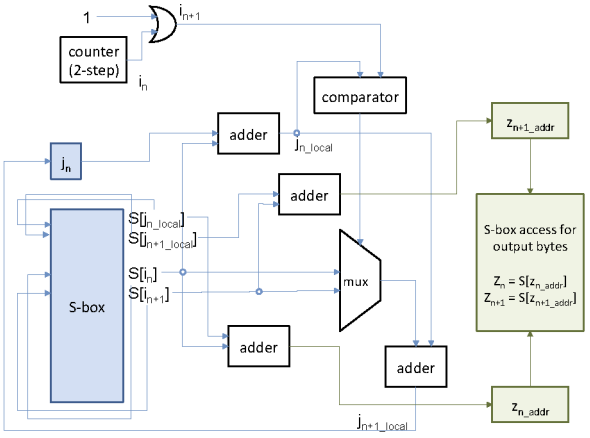


Fig. 13. PRGA circuit structure for proposed architecture.

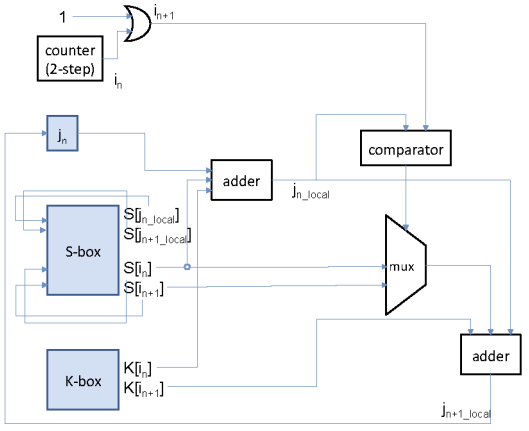


Fig. 14. KSA circuit structure for proposed architecture.

The main storage for the RC4 hardware, as before, is centered around the S-register array and the K-register array. The S-register box comprises of 8 bit registers made of edge-triggered master-slave flip-flops, with a total of 256 such registers to maintain the RC4 states. To accommodate the read and write accesses to the S-box, we use write-access decoders and read-access decoders which in turn control 256-to-1 multiplexer units associated to each location of the state array. The K-register box, that holds the RC4 key, is also designed in a similar fashion, but with the exception that no write accesses are required for the K-registers.

3.5 Structure of PRGA and KSA Circuits

The schematic diagrams for PRGA and KSA circuits in the proposed design are shown in Figs. 13 and 14, respectively.

The PRGA circuit operates as per the 2-stage pipeline structure, where the increment of indices take place in the first stage, and so does the double-swap operation for the S-box. In the same stage, the addresses for the two consecutive output bytes Z_n and Z_{n+1} are calculated as the swap does not change the outcomes of the additions $S[i_n] + S[j_n]$ or $S[i_{n+1}] + S[j_{n+1}]$. In the second stage of the pipeline, the output addresses z_{n_addr} and z_{n+1_addr} are used to read the appropriate keystream bytes from the updated S-box.

The circuit for KSA operates similarly, but has no pipeline feature as the operation happens in a single stage. Here, the increment of indices and swap are done for two consecutive rounds of KSA in a single clock cycle, thereby producing a speed of 2-rounds-per-cycle.

Based on this schematic diagram for the circuits, and the port-sharing logic described earlier, we now attempt the hardware implementation of our new design.

3.6 Implementation

We have implemented the proposed structure for RC4 stream cipher using synthesizable VHDL description. The S-register box and K-register box are implemented as array of master-slave flip-flops, as discussed earlier, and are synthesized as standard-cell memory architecture. The VHDL code is synthesized with 130, 90, and 65 nm fabrication technologies using Synopsys Design Compiler in topographical mode. The synthesis results are presented in Section 4.

4 Implementation Results

In this section, we describe our attempts to optimize our designs and obtain the best throughput in implementation. The gate-level synthesis was carried out using Synopsys Design Compiler Version D-2010.03-SP4, using topographical mode for 130, 90, and 65 nm target technology libraries. The area results are reported using equivalent 2 input NAND gates.

4.1 Hardware Performance of Our Designs

90 nm technology. We experimented with the synthesis of designs in the following order:

- 1-byte/clock design using loop unrolling (Design 1),
- 1-byte/clock by hardware pipelining (Pipelined-A),
- 1-byte/clock by hardware pipelining (Pipelined-B),
- 1-byte/clock by hardware pipelining (Pipelined-C),
- 2-bytes/clock design combining the two (Design 2).

In order to get the best throughput out of our proposed designs, we performed a few experiments with varying clock speed. This included running of all synthesis at 90 nm with strict clock period constraints until no further improvement was possible. The synthesis results are shown in Table 5. The clock period in Pipelined-B is higher than that for Pipelined-A, due to the port-sharing logic that we introduced in Pipelined-B.

Critical path for design 2. After the initial implementations of the designs, we found that the critical path for Design 2 is through the KSA read access of the S-array, followed by the additions for updating j values in the first stage of Fig. 10.

We tried two mechanisms to reduce this critical path. First, we attempted reduced port sharing, as port sharing puts longer delay in the multiplexers. Second, we attempted a modified design with three pipeline stages; the first stage to load the S and K values and put those in pipeline registers, the second stage to perform the additions for j update, and the last one for the swap in KSA. This not only made the design 3-stage pipelined instead of 2-stages, but also required additional bypass logic, which did not help to reduce the critical path. So, we finally kept the 2-stage pipeline as in Fig. 10, and avoided some port sharing along the critical path. This shifted the critical path to the S-box write access from KSA. By removing the port sharing, the clock frequency could be improved even further.

TABLE 5
Synthesis Results for Optimized Designs with Different Target Technology Libraries

Tech. (nm)	Design	Area (NAND gate equivalent)							Max. Clock Freq. (GHz)	KSA (cycles)	PRGA (bytes/cycle)	Speed (Gbps)
		KSA	PRGA	S-box	K-box	Sequential	Combinational	Total				
130	Design 2	310	1000	57435	1180	13819	46106	59925	0.625	256	2	10.00
90	Design 1	310	1199	48669	1084	10942	40320	51262	1.22	256	1	9.76
	Pipelined-A	560	428	52133	1084	10644	43561	54205	1.37	128	1	10.96
	Pipelined-B	527	428	43454	1084	10611	34882	45493	1.25	128	1	10.00
	Pipelined-C	484	612	39231	1084	10801	30610	41411	1.49	256	1	11.92
	Design 2	310	985	52557	1084	10955	43981	54936	1.37	256	2	21.92
65	Design 1	310	1240	53638	1180	14109	42259	56368	1.85	256	1	14.80
	Pipelined-C	550	690	48159	1180	13622	36957	50579	2.22	256	1	17.76
	Design 2	310	927	52998	1180	13484	41931	55415	1.92	256	2	30.72

Currently, the critical path is in the S-box read access from PRGA. That could also be improved by removing some port sharing, but only by causing a heavy increase in area. Therefore, we chose to avoid it.

65 nm technology. We used the same designs which yielded best clock frequencies in 90 nm, and mainly focused at the following three designs:

- 1-byte/clock design using loop unrolling (Design 1),
- 1-byte/clock by hardware pipelining (Pipelined-C), and
- 2-bytes/clock design combining the two (Design 2).

The synthesis results provide us the best throughput for these three designs, obtained by using strict clock period constraints during the implementation, until no further improvement could be made, i.e., until the point that it could generate a valid gate-level netlist. The synthesis results are presented in Table 5.

130 nm technology. In order to benchmark against comparable technology libraries, we have synthesized using 130 nm fabrication technology since, several stream ciphers from eSTREAM project [2] have reported their performance in 130 nm and 250 nm technologies [7], [8]. We used the same designs which yielded best clock frequencies in 65 nm and 90 nm design, and have just implemented our best proposal: Design 2. The synthesis results are presented in Table 5.

Best Throughput. To summarize, the optimized synthesis offers us the best throughput (in gigabits per second) for hardware implementation of RC4 cipher till date.

Design 1 (one-byte-per-clock):

- **9.76 Gbps** in 90 nm technology,
- **14.8 Gbps** in 65 nm technology.

Design 2 (two-bytes-per-clock):

- **10 Gbps** in 130 nm technology,
- **21.92 Gbps** in 90 nm technology, and
- **30.72 Gbps** in 65 nm technology.

4.2 Comparison with Existing Designs

We compare our proposed designs, Design 1 and Design 2 with the ones that existed for RC4 hardware till date. We only consider existing designs that are focused toward improved throughput, and not any other hardware considerations.

Throughput comparison. In Section 2, we have already seen the main RC4 designs proposed in the literature: 3-cycles-per-byte designs of [10] and [16] and 1-byte-per-cycle design of [17]. Section 2 presents our 1-byte-per-cycle architecture Design 1, and in Section 3 we propose the first 2-bytes-per-cycle RC4 architecture Design 2. It requires 257 cycles to complete KSA and generates 2-bytes-per-cycle in PRGA, with an initial lag of two cycles. Thus, Design 2 produces N keystream bytes in approximately $257 + (N/2 + 2) = N/2 + 259$ clock cycles. For large N, this is twice in comparison with Design 1 and the hardware proposed in [17]. Detailed comparison of throughput is presented in Table 6.

Area comparison. As discussed earlier in Section 2, a precise comparison of area requirements with [10] could not be made due to mismatch in implementation platforms, and [16] does not specify any area figures at all. However, [16] uses two 256-byte dual-port RAM blocks for the storage, and for the purpose of comparison we synthesized the RAM using 65 nm technology. This resulted in approximately 11.3 KGates of storage area (without considering the associated circuitry), which is comparable to the total sequential area (approximately 13.5-14.0 KGates for Design 1 and Design 2 in 65 nm technology, as shown in Table 5) of our designs with register-based storage. Fair comparison with [17] could not be done due to lack of relevant area figures.

TABLE 6
Timing Comparison of Design 1 and Design 2 with That of [10], [16], and [17]

Operations	Number of clock cycles required for each operation			
	[10] and [16]	[17]	Design 1	Design 2
Per KSA round	3	1	1	1
Complete KSA	$256 \times 3 = 768$	$256 + 3 = 259$	$256 + 1 = 257$	$256 + 1 = 257$
N bytes of PRGA	$3N$	$N + 3$	$N + 2$	$N/2 + 2$
N bytes of RC4	$3N + 768$	$259 + (N + 3) = N + 262$	$257 + (N + 2) = N + 259$	$257 + (N/2 + 2) = N/2 + 259$
Cycles per byte	$3 + \frac{768}{N}$	$1 + \frac{262}{N}$	$1 + \frac{259}{N}$	$1/2 + \frac{259}{N}$

To put our results in perspective, we surveyed the throughputs of a few popular hardware stream ciphers. The current eSTREAM portfolio of hardware stream ciphers contains three ciphers: Grain_v1, MICKEY_v2, and Trivium. According to a hardware performance evaluation of the ciphers (as in [7] and [8]), these ciphers achieve the following throughputs, with maximum possible optimization and parallelization:

- Grain128: 14.48 Gbps (130 nm), 4.475 Gbps (250 nm),
- MICKEY: 0.413 Gbps (130 nm), 0.287 Gbps (250 nm), and
- Trivium: 22.3 Gbps (130 nm), 18.568 Gbps (250 nm).

In the data above, the current version Grain128_v1 is tested on 130 nm technology, whereas the result for 250 nm technology is with Grain128_v0, as mentioned in [8].

One may observe that in context of the eSTREAM hardware stream ciphers, the optimized implementation of RC4 that we provide fares quite well in terms of throughput (10 Gbps), although RC4 is never claimed to be a hardware cipher. It should also be noted that the area requirements for the proposed RC4 designs (50-60 KGates for Designs 1 and 2) are fairly high compared to those in case of the aforementioned eSTREAM ciphers, as evaluated in [7] (3.2 KGates for Grain128, 5.0 KGates for MICKEY, and 4.9 KGates for Trivium). However, compared to processors or coprocessors in embedded systems, this area is quite reasonable, and is small enough to be integrated in modern embedded processors. The optimization is between the efficiency requirement and the area constraint on the user end. If high throughput is required for the time-tested and widely accepted stream cipher RC4, the user may go for a slightly high-area implementation as we have proposed in this paper, and if the area constraints are stricter, the user may go for the RAM-based implementation proposed in [5], or choose the new lightweight stream ciphers over RC4.

5 Issues with Further Improvements

Based on Design 2, the fastest known hardware implementation of RC4 till date, one may be tempted to push the architecture even further so as to increase its throughput. We tried to venture in this direction as well, and noticed that a better throughput can be obtained via one of the two following avenues:

1. Unroll more loops of the algorithm.
2. Increase the pipeline depth.

First, we shall take a look at issues with further loop unrolling, starting with the idea of Design 1.

5.1 Unrolling Three or More Loops of RC4

In hardware design, the idea of loop unrolling proves to be most effective when the parameters involved in each of the loops are independent. In case of KSA or PRGA in RC4 stream cipher, we are not quite so lucky. The interdependencies between consecutive loops of RC4 originate from the following ordering of steps:

$$\text{Update indices} \rightarrow \text{Swap } S\text{-values} \rightarrow \text{Output } Z.$$

This order has to be obeyed at all circumstances to maintain the correctness of the cipher.

Recall Design 1 (Section 2) where we first introduce the concept of loop unrolling in case of RC4. To implement this idea, we had to take into account all dependencies between two consecutive loops. As the Swap and Output stages depend directly upon the indices i and j, we only needed to consider the interplay between the indices of the two rounds, i.e., i_1, i_2 and j_1, j_2. We had a total of $\binom{4}{2} = 6$ pairs to deal with, but the equality or inequality of 3 of these $(i_1, i_2; i_1, j_1; i_2, j_2)$ did not matter, as they were either impossible to occur, or were anyway expected in the RC4 algorithm. So, we had to worry about the comparison between three pairs of indices

$$(i_1, j_2) \quad \text{and} \quad (i_2, j_1) \quad \text{and} \quad (j_1, j_2)$$

in case of S-box swaps as well as the computations for Z_1 and Z_2. These three pairs gave rise to $2^3 = 8$ choices in each case, and that in turn contributed to the large area for the combinational logic in our architecture.

Now, if we try to unroll another loop of RC4, i.e., three consecutive rounds of the cipher at a time, we will have to deal with six indices i_1, i_2, i_3, and j_1, j_2, j_3. There are $\binom{6}{2} = 15$ pairs, out of which we will have to consider nine pairs for comparison

$$(i_1, j_2), (i_1, j_3), (i_2, j_1), (i_2, j_3), (i_3, j_1), (i_3, j_2) \text{ and } (j_1, j_2), (j_2, j_3), (j_1, j_3).$$

These pairs will give rise to $2^9 = 512$ choices in each case of S-box swaps and output computation, and will require combinational logic to take care of the choices.

In the hardware implementation of Design 1 and Design 2, one may observe that the combinational area already figures quite high. With three loops unrolled, it will be impossible to manage as the logic requirement grows exponentially (eight choices for two loops to 512 choices for three loops). Hence, the idea of further loop unrolling in RC4 do not seem feasible, and we drop the idea.

5.2 Increasing Depth of the Pipeline

In this case, the motivation was to check if deeper hardware pipelining can further improve the throughput for our RC4 architecture. During our experiments with the design, we observed that the critical path in the architecture is due to S-box access, which cannot be improved by whatever deep pipelining one may design.

The only way to reduce the critical path in S-box access logic is to explore either of the following choices:

- Hand-optimizing the multiplexer logic.
- Partitioned S-box to reduce multiplexing logic.

The first option is hard to perform from the RTL and the multiplexer logic is anyway highly optimized by the synthesis tools. One may however investigate the second option further, but most likely such a structure will require predictable access pattern for different partitions of the S-box. This "predictable access pattern" might lead to compromising the cryptographic security, which is not desirable at all.

6 Conclusion and Outlook

The alleged RC4 has been dominant in the arena of stream ciphers since its advent in 1987, and has earned its reputation as the most popular stream cipher till date. Be it academics or the industry, RC4 has been used in numerous forms and shapes in a majority of cryptographic solutions based on stream ciphers. The algorithm for the cipher is intriguingly simple, and one can easily implement it within a few lines of code. This has further promoted RC4 as a natural choice for a software-based stream cipher.

It is rumored in the cryptographic community that an efficient RC4 software implementation can produce the keystream bytes at a rate of three cycles per byte. However, we could not find any documented evidence to this claim. The best software performance of RC4 stated in the literature till date is available at [1], and it provides an account of 7 to 15 cycles per byte of RC4 keystream, depending on the processor and the clock-speed the cipher was tested on.

We take a look at the other side of the coin; the hardware implementation of RC4. There have been a few RC4 hardware designs proposed in the literature, but none provided a complete analysis of the problem. In this paper, we have presented a thorough study of RC4 hardware designing problem from the point of view of throughput, measured in gigabits per second output of the RC4 keystream. We have discussed the issues of loop unrolling and hardware pipelining in RC4 architecture to obtain better throughput, and have experimented extensively to optimize area and performance.

In the process, we have proposed two new designs for RC4 hardware. The first one produces one keystream byte per cycle using the idea of loop unrolling, that has never been exploited in case of RC4 hardware implementation. Our second design tops the first one by combining the idea of loop unrolling with that of efficient hardware pipelining to obtain two keystream bytes per cycle. This is the fastest known RC4 hardware architecture till date, providing a keystream throughput of *30.72 Gbps* in its optimized form, using 65 nm technology.

We have also studied the obvious scopes for further improvement in throughput, using more loop unrolling or better pipelining, and have shown that these tweaks are either not feasible or not optimum in terms of area and speed.

In the future, we will have further look into the hardware architecture for optimizing the area without compromising the runtime performance. It is also an interesting research to look into a flexible hardware platform, which can support multiple fast stream ciphers. Even within the context of RC4, several security enhancements are proposed. While this study concentrated on the basic RC4 implementation, it remains an open, interesting problem to study the security-performance tradeoffs for different RC4 variants.

Acknowledgments

The authors are thankful to the anonymous reviewers for their detailed comments and suggestions, which helped in improving the technical and editorial quality of this paper.

References

[1] Software Performance Results from the eSTREAM Project, eSTREAM, the ECRYPT Stream Cipher Project, http://www.ecrypt.eu.org/stream/perf/#results, 2012.

[2] The Current eSTREAM Portfolio, eSTREAM, the ECRYPT Stream Cipher Project, http://www.ecrypt.eu.org/stream/index.html, 2012.

[3] S.R. Fluhrer and D.A. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator," *Proc. Seventh Int'l Workshop Fast Software Encryption (FSE '00)*, vol. 1978, pp. 19-30, 2000.

[4] S.R. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," *Proc. Eighth Ann. Int'l Workshop Selected Areas in Cryptography (SAC '01)*, vol. 2259, pp. 1-24, 2001.

[5] M.D. Galanis, P. Kitsos, G. Kostopoulos, N. Sklavos, and C.E. Goutis, "Comparison of the Hardware Implementation of Stream Ciphers," *Int'l Arab J. Information Technology*, vol. 2, no. 4, pp. 267-274, 2005.

[6] J. Golic, "Linear Statistical Weakness of Alleged RC4 Keystream Generator," *Proc. Advances in Cryptology EUROCRYPT*, vol. 1233, pp. 226-238, 1997.

[7] T. Good and M. Benaissa, "Hardware Results for Selected Stream Cipher Candidates," eSTREAM, ECRYPT Stream Cipher Project, SASC, Report 2007/023, 2007.

[8] F.K. Gurkaynak, P. Luethi, N. Bernold, R. Blattmann, V. Goode, M. Marghitola, H. Kaeslin, N. Felber, and W. Fichtner, "Hardware Evaluation of eSTREAM Candidates: Achterbahn, Grain, MICKEY, MOSQUITO, SFINKS, Trivium, VEST, ZK-Crypt," eSTREAM, ECRYPT Stream Cipher Project, Report 2006/015, 2006.

[9] P. Hamalainen, M. Hannikainen, T. Hamalainen, and J. Saarinen, "Hardware Implementation of the Improved WEP and RC4 Encryption Algorithms for Wireless Terminals," *Proc. European Signal Processing Conf.*, pp. 2289-2292, 2000.

[10] P. Kitsos, G. Kostopoulos, N. Sklavos, and O. Koufopavlou, "Hardware Implementation of the RC4 Stream Cipher," *Proc. IEEE 46th Midwest Symp. Circuits and Systems*, http://dsmc.eap.gr/en/members/pkitsos/papers/Kitsos_c14.pdf, 2003.

[11] J.-D. Lee and C.-P. Fan, "Efficient Low-Latency RC4 Architecture Designs for IEEE 802.11i WEP/TKIP," *Proc. Int'l Symp. Intelligent Signal Processing and Comm. Systems (ISPACS '07)*, pp. 56-59, 2007.

[12] T. Lynch and E.E. Swartzlander Jr., "A Spanning Tree Carry Lookahead Adder," *IEEE Trans. Computers*, vol. 41, no. 8, pp. 931-939, Aug. 1992.

[13] I. Mantin, "Predicting and Distinguishing Attacks on RC4 Keystream Generator," *Proc. 24th Ann. Int'l Conf. Theory and Applications of Cryptographic Techniques (EUROCRYPT '05)*, vol. 3494, pp. 491-506, 2005.

[14] I. Mantin, "A Practical Attack on the Fixed RC4 in the WEP Mode," *Proc. 11th Int'l Conf. Theory and Application of Cryptology and Information Security*, vol. 3788, pp. 395-411, 2005.

[15] I. Mantin and A. Shamir, "A Practical Attack on Broadcast RC4," *Proc. Eighth Int'l Workshop Fast Software Encryption (FSE '01)*, vol. 2355, pp. 152-164, 2001.

[16] D.P. Matthews Jr., "System and Method for a Fast Hardware Implementation of RC4," US Patent Number 6549622, Campbell, CA, http://www.freepatentsonline.com/6549622.html, Apr. 2003.

[17] D.P. Matthews Jr., "Methods and Apparatus for Accelerating ARC4 Processing," US Patent Number 7403615, Morgan Hill, CA, http://www.freepatentsonline.com/7403615.html, July 2008.

[18] A. Maximov and D. Khovratovich, "New State Recovering Attack on RC4," *Proc. 28th Ann. Conf. Cryptology: Advances in Cryptology*, vol. 5157, pp. 297-316, 2008.

[19] I. Mironov, "(Not So) Random Shuffles of RC4," *Proc. 22nd Ann. Int'l Cryptology Conf. Advances in Cryptology*, pp. 304-319, 2002.

[20] G. Paul and S. Maitra, "On Biases of Permutation and Keystream Bytes of RC4 Towards the Secret Key," *Cryptography and Comm.*, vol. 1, no. 2, pp. 225-268, 2009.

[21] A. Roos, "A Class of Weak Keys in the RC4 Stream Cipher," Two Posts in sci.crypt, http://marcel.wanda.ch/Archive/WeakKeys, 1995.

[22] S. Sen Gupta, K. Sinha, S. Maitra, and B.P. Sinha, "One Byte per Clock: A Novel RC4 Hardware," *Proc. INDOCRYPT '10*, vol. 6498, pp. 347-363, 2010.

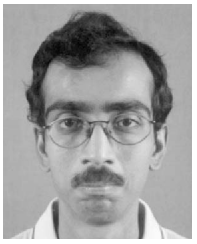
[23] B.P. Sinha and P.K. Srimani, "Fast Parallel Algorithms for Binary Multiplication and Their Implementation on Systolic Architectures," *IEEE Trans. Computers*, vol. 38, no. 3, pp. 424-431, Mar. 1989.

[24] D. Wagner My RC4 Weak Keys, Post in sci.crypt, http://www.cs.berkeley.edu/daw/my-posts/my-rc4-weak-keys, 1995.

Sourav Sen Gupta received the Bachelor of Electronics and Telecommunication Engineering degree from Jadavpur University, Kolkata, India in 2006 and the master of mathematics degree from University of Waterloo, Ontario, Canada in 2008. He is currently working toward the doctoral studies at Indian Statistical Institute, Kolkata, India. His research interests include cryptology and number theory.

Anupam Chattopadhyay received the BE degree from Jadavpur University, India in 2000. He received the MSc degree from ALaRI, Switzerland and PhD degree from RWTH Aachen, Germany, in 2002 and 2008, respectively. During his PhD, he worked on automatic RTL generation from the architecture description language LISA, which was commercialized later by CoWare. He further developed several high-level optimization techniques and verification flow for embedded processors and a language-based modeling, exploration, and implementation framework for partially reconfigurable processors. He has published several technical papers and authored one book in the above research areas. He spent more than three years in different engineering and research positions in industry. He is currently at RWTH Aachen University as an assistant professor in the UMIC research cluster. His research interests include the design and automation of multiprocessor systems for future embedded applications. He is a member of the IEEE.

Koushik Sinha received the MS degree from the Clemson University, South Carolina, and the PhD degree from the Jadavpur University, Calcutta, India in 2003 and 2007, respectively. From 2004 to 2011, he worked at Honeywell Technology Solutions Lab, Bangalore as a lead research scientist. He is currently a scientist at Hewlett Packard Labs, Bangalore, India. He received the Indian Science Congress Young Scientist Award for the year 2008-2009 in the Information and Communication Science and Technology (including Computer Sciences) category. He has served in TPCs and organizing committees of several international conferences. His research interests include the areas of wireless networks, high performance computing, and information retrieval. He is a member of the IEEE.



Subhamoy Maitra received the Bachelor of Electronics and Telecommunication Engineering degree in the year 1992 from Jadavpur University, Calcutta and Master of Technology in Computer Science in the year 1996 from Indian Statistical Institute, Calcutta. He received the PhD degree from Indian Statistical Institute in 2001. Currently he is a professor at Indian Statistical Institute. His research interests include Cryptology and Security.

Bhabani P. Sinha received the MTech and PhD degrees in 1975 and 1979, respectively, both from the University of Calcutta. He joined the faculty of Electronics Unit of the Indian Statistical Institute, Calcutta in 1976, where he became a professor in 1987. From 1993 to 2011, he was the head of the Advanced Computing and Microelectronics Unit of the Indian Statistical Institute, Calcutta. He is currently the dean of Studies at the Indian Statistical Institute, Calcutta. He served as an Indian Space Research Organization (ISRO) Chair Professor during 2000 at the Indian Institute of Technology, Kharagpur, India. He has published more than 130 research papers in various international journals and refereed conference proceedings in the area of computer architectures, algorithms, parallel and distributed computing, mobile computing and wireless communication. His recent research interests include parallel and distributed computing, mobile computing, wireless networks and algorithms. He served as an editor of the *IEEE Transactions on VLSI Systems* during 1998-2000 and as a subject area editor of the *Journal of Parallel and Distributed Computing* till 2011. He is a fellow of the IEEE.