Quad-RC4: Merging Four RC4 States towards a 32-bit Stream Cipher*

Goutam Paul[1], Subhamoy Maitra[1], and Anupam Chattopadhyay[2]

[1] Indian Statistical Institute,
Kolkata 700 108, India.
{goutam.paul,subho}@isical.ac.in
[2] MPSoC Architectures, RWTH Aachen University,
52074 Aachen, Germany.
anupam@ice.rwth-aachen.de

Abstract. RC4 has remained the most popular software stream cipher since the last two decades. In parallel to cryptanalytic attempts, researchers have come up with many variants of RC4, some targeted to more security, some towards more throughput. We observe that the design of RC4 has been changed a lot in most of the variants. Since the RC4 structure is quite secure if the cipher is used with proper precautions, an arbitrary change in the design may lead to potential vulnerabilities, such as the distinguishing attack (Tsunoo et al., 2007) on the word-oriented variant GGHN (Gong et al., 2005). Some variants keep the RC4 structure (Maitra et al., 2008), but is byte-oriented and hence is an overkill for modern wide-word processors. In this paper, we try to combine the best of both the worlds. We keep the basic RC4 structure which guarantees reasonable security (if properly used) and we combine 4 RC4 states tacitly to design a high throughput stream cipher called *Quad-RC4* that produces 32-bit output at every round. The storage requirement for the internal state is only 1024 bits. In terms of speed, this cipher performs much faster than normal RC4 and is comparable with HC-128, the fastest software stream cipher amongst the eSTREAM finalists. We also discuss the issue of generalizing the structure of Quad-RC4 to higher word-width variants.

Keywords: Bias, Keystream, Quad-RC4, RC4 Variant, Stream Cipher.

1 Introduction

Ron's Code 4 or RC4, designed by Ron Rivest, is one of the most popular stream ciphers in symmetric key cryptography. Despite its overwhelmingly simple structure and repeated attempts to cryptanalyze the cipher from cryptologists around the world over more than two decades, the cipher is not yet completely broken. With certain precautions, RC4 can be used safely in any application requiring cryptographic security. Probably this is the reason for wide commercial deployment of RC4 in applications like SSL, TLS, WEP, WPA etc.

Like all stream ciphers, RC4 works in two phases. The Key Scheduling Algorithm (KSA) turns the identity permutation over $\mathbb{Z}_N$ (N is typically 256) into a random-looking permutation with the help of the secret key. The Pseudo Random Generation Algorithm (PRGA) generates a sequence of pseudo-random bytes. This sequence,

* This is a work in progress which began in May 2012. Part of this work was mentioned in a tutorial talk by the second author at INDOCRYPT 2012.

known as the keystream, is bitwise XOR-ed with the plaintext during encryption and with the ciphertext during decryption.

The permutation, which is the main component of the internal state of the cipher, is denoted by an array $S[0 \dots N-1]$. To access the array elements, two indices i and j are used, of which the first one is deterministic and public and the second one is pseudo-random and secret. An array $K[0 \dots N-1]$, which is of the same size as the permutation, holds the secret key $k[0 \dots l-1]$, by repeating it as follows.
$K[y] = k[y \bmod l]$, for $0 \leq y \leq N-1$.

Algorithm RC4 KSA	**Algorithm RC4 PRGA**
Initializing the State:	*Initializing the Indices*:
For $i = 0, \dots, N-1$	$i = j = 0$;
$S[i] = i$;	
$j = 0$;	*Generating Keystream Bytes*
	(*Loop as many plaintext bytes*):
Scrambling the State:	$i = i + 1$; $j = j + S[i]$;
For $i = 0, \dots, N-1$	Swap($S[i]$, $S[j]$);
$j = (j + S[i] + K[i])$;	$t = S[i] + S[j]$;
Swap($S[i], S[j]$);	Output $z = S[t]$;

1.1 Weaknesses of RC4

One important work in RC4 cryptanalysis is recovering the secret key, assuming that the permutation at any stage after the KSA is known. The first work in this direction appeared in [31] and later continued in [4, 2, 3]. All of these works mainly exploit the permutation-key correlations found by [32].

Key recovery from keystream is another milestone in RC4 cryptanalysis that mainly attacks RC4 in the WEP [11, 24, 42, 18, 38] and WPA [35] mode.

Works on state recovery [19, 26] have reduced the state space search from 2^{1700} to 2^{241} and so the use of secret keys of length more than 30 bytes is not recommended any more.

Apart from the above, many biases in RC4 keystream have also been discovered, most of which is short term [32, 22, 34, 21], and a few long-term [13, 10, 23].

1.2 Existing RC4 Variants

Several variants of RC4 exist in the literature. Two variants of RC4 were proposed in FSE 2004. One of them is VMPC [45] and the other one is RC4A [30]. The design of VMPC is based on multilevel nested permutation access, and the structure looks like $P[P_k[P_{k-1}[\dots[P_1[P[x]]]\dots]]]$, $0 \leq x \leq N-1$, where $P_i[x] = (P[x] + i) \bmod N$. RC4A uses two different secret keys and two permutation arrays. After these permutations are scrambled with their respective keys, pairs of elements of one permutation is used as index to pick an output byte from the other permutation.

Some works focus on lifting RC4 from its byte-oriented structure to higher word extensions. The NGG [29] cipher expanded to 32/64 bits with a state size much smaller than 2^{32} or 2^{64}. Subsequently, an updated version of this cipher called GGHN [14] appeared. Py (pronounced Roo) [6] is an RC4-like stream cipher that was submitted to the eSTREAM project [5]. It uses rolling arrays and produces two 32-bit words as output at every step. After some weaknesses were discovered, a series of improvements followed, with the names Pypy [7] and TPypy [8].

In all the variants above, the design is modified to a great extent relative to RC4. In [20], a new variant called RC4$^+$ was presented that keeps the original RC4 structure and adds a few more operations to improve the security of the cipher.

1.3 Motivation for yet another RC4 Variant

RC4 was designed as a byte-oriented software stream cipher. On the other hand, the typical word length of modern processors is mostly 32 bits. For optimal utilization of the resources, a software cipher running on a w-bit machine should ideally produce w bits in every round of keystream generation. Thus, though no attack has been reported on the byte-oriented RC4-like design RC4$^+$ [20], it is not suitable for modern-day processors in terms of resource utilization.

A straight-forward generalization of w-bit RC4 would require a storage of 2^w size permutation over $\mathbb{Z}_{2^w}$. For $w = 32$, storing such a permutation requires 32×2^{32} bits, i.e., 128 Giga bits. Thus, storing the internal state of such a simple extension of RC4 becomes practically infeasible.

The authors of the GGHN stream cipher [14] noticed this gap between the processor word size and RC4 output and proposed a scheme for RC4(n, m) where $N = 2^n$ is the size of the array S in words, m is the word size in bits, $n \leq m$ and $M = 2^m$. However, there are several shortcomings in the design of [14].

1. The basic design of RC4 has been changed to a great extent. We call it a shortcoming, since the time-tested security of RC4 no longer carries forward.
2. The work [41] built a distinguisher on GGHN based on a bias in the first two words of the keystream, associated with approximately 2^{30} samples.
3. In the performance evaluation, they claim that RC4(8, 32) is approximately 3.1 times faster than the original 8-bit RC4 on a 32-bit machine and RC4(8, 64) is 6.2 times faster than RC4 on a 64-bit machine. However no rigorous performance evaluation and comparison with benchmark implementation was not reported.

We focus on building a 32-bit RC4 for a 32-word machine, but we use the basic 8-bit RC4 as a building block. it output at every round of keystream generation. We take a single l-byte key ($16 \leq l \leq 30$) and use it to drive 4 different key schedulings in parallel to obtain 4 different permutations over $\mathbb{Z}_{256}$. We argue that our scheme is more secure than that of basic RC4. We perform extensive performance evaluation and compare with the faster software stream cipher HC-128 in the eSTREAM portfolio.

2 Key Scheduling

We use the key scheduling of RC4$^+$ [20] (for reasons discussed in Section 4), called KSA$^+$, as a building block of our key scheduling scheme. We run four KSA$^+$ on four identity permutations to generate four scrambled permutations over $\mathbb{Z}_{256}$.

Before discussing about multiple RC4 states, let us briefly summarize the KSA$^+$ algorithm. It is executed in three layers, followed by the initialization. The initialization and basic scrambling in the first layer are the same as the original RC4 KSA.

Initialization
For $i = 0, \ldots, N-1$
$\quad S[i] = i;$
$j = 0;$

Layer 1: Basic Scrambling
For $i = 0, \ldots, N-1$
$\quad j = (j + S[i] + K[i]);$
$\quad \text{Swap}(S[i], S[j]);$

In the second layer, the permutation is scrambled using IV's. The deterministic index i moves first from the middle down to the left end and then from the middle up to the right end. An l-byte IV, denoted by an array $iv[0, \ldots, l-1]$, is used from index $\frac{N}{2} - 1$ down to $\frac{N}{2} - l$ during the left-ward movement and the same IV is repeated from index $\frac{N}{2}$ up to $\frac{N}{2} + l - 1$ during the right-ward movement. Here, N is assumed to be even, which is usually the case in standard RC4. For ease of description, we use an array IV of length N with $IV[y] = 0$ for those indices which are not used with IV's.

Layer 2: Scrambling with IV
For $i = \frac{N}{2} - 1$ down to 0
$\quad j = (j + S[i]) \oplus (K[i] + IV[i]);$
$\quad \text{Swap}(S[i], S[j]);$
For $i = \frac{N}{2}, \ldots, N-1$
$\quad j = (j + S[i]) \oplus (K[i] + IV[i]);$
$\quad \text{Swap}(S[i], S[j]);$

Layer 3: Zigzag Scrambling
For $y = 0, \ldots, N-1$
$\quad$ If $y \equiv 0 \bmod 2$ then $i = \frac{y}{2};$
$\quad$ Else $i = N - \frac{y+1}{2};$
$\quad j = (j + S[i] + K[i]);$
$\quad \text{Swap}(S[i], S[j]);$

In the third and final layer, a zig-zag scrambling is performed, where the deterministic index i takes values in the following order: 0, 255, 1, 254, 2, 253, ..., 125, 130, 126, 129, 127, 128. In general, if y varies from 0 to $N-1$ in steps of 1, then $i = \frac{y}{2}$ or $N - \frac{y+1}{2}$ depending on y is even or odd respectively.

In order to run four KSA$^+$ on four identity permutations, we need to expand the l-byte secret key into four l-byte subkeys. We assume l to be even, i.e., $l = 2n$ for some integer n. Then the four sub-keys are formed as follows. The first subkey is the key as it is, i.e.,

$$(k[0], \ldots, k[n-1], k[n], \ldots, k[2n-1]).$$

The second subkey is just the reverse of this one. The third subkey is formed as

$$(k[n-1], \ldots, k[0], k[2n-1], \ldots, k[n]).$$

The fourth subkey is again the reverse of the third subkey. We also mandate a $4l$-byte IV, divided into 4 different l-byte sub-IV's to be used with the individual KSA$^+$.

Note that the above scheme of key scheduling of the proposed cipher is slower than RC4. But it does not matter since the key scheduling would be run only once and our goal is to design a software stream cipher for encrypting long stream of data.

3 Combining Four RC4 States and Keystream Generation

After the end of the key scheduling, we have 4 pseudo-random permutations over $\mathbb{Z}_{256}$. Let us denote the permutations by $S_1, \ldots, S_4$. They are merged into a single array S of size 256, where the i-th entry of S is an 32-bit number, formed by concatenating the 4 bytes $S_1[i]$, ..., $S_4[i]$. Thus, one can visualize S as a two-dimensional structure, where each column is a word of S and each row is a permutation. We use two variables i and j to access a word of S or a byte of the individual permutations.

Input: 1. 4 pseudo-random permutations over $\mathbb{Z}_{256}$.
2. No. of rounds R.
Output: $32R$ pseudo-random keystream bits.

Initializing the Indices:
$i = j = 0$;
Merge Four Permutations in a Single Array
for $i = 0, \ldots, 255$ **do**
  $S[i] = (S_1[i] \ll 24) \mid (S_2[i] \ll 16) \mid (S_3[i] \ll 8) \mid S_4[i]$;
end
Generating Keystream Words
for $r = 1, \ldots, R$ **do**
  $i = (i + 1) \bmod 256$;
  $j = (j + S_4[i]) \bmod 256$;
  Swap($S[i]$, $S[j]$);
  $t = (S[i] + S[j]) \bmod 256$;
  $t_1 = t$ & 0xFF;
  $t_2 = (t \gg 8)$ & 0xFF;
  $t_3 = (t \gg 16)$ & 0xFF;
  $t_4 = (t \gg 24)$ & 0xFF;
  Output $z = S[t_1] \oplus S[t_2] \oplus S[t_3] \oplus S[t_4]$;
  $\{a, b\}$ = Next pair of permutations in turn;
  Swap($S_a[i]$, $S_a[t_a]$);
  Swap($S_b[i]$, $S_b[t_b]$);
end

Algorithm 1: Quad-RC4 PRGA

Like normal RC4, the index i is incremented by 1 modulo 256 and j is incremented pseudo-randomly using only one particular permutation, say S_4.

At every round, we swap the i-th and the j-th word of S; in effect we swap the corresponding entries of each permutation. $S[i]$ and $S[j]$ are added to generate a 32-

bit number t, ignoring the carry. Let the 4 bytes of t be denoted by t_1, ..., t_4. The output keystream word z is simply the bitwise XOR of the words $S[t_1], \ldots, S[t_4]$.

To break the symmetry in the swaps of the individual permutations, we introduce some additional swaps (see Section 4 for its security implications). At every round, we select two permutations S_a and S_b. We then swap the i-th and the t_a-th bytes of S_a and swap the i-th and the t_b-th bytes of S_b. Note that t_a and t_b are the a-th and the b-th bytes of $t = t_1||\ldots||t_4$.

Two permutations out of 4 can be selected in $\binom{4}{2} = 6$ ways. We choose each combination in turn and hence one particular pair of permutations is considered again after 6 rounds. Note that the structure ensures that the individual permutations remain permutations. The complete algorithm for keystream generation is presented in Algorithm 1.

3.1 MAC computation

The same keystream generation algorithm along with an integrity key can be used to compute a 32-bit MAC (Message Authentication Code) of the input plaintext. The idea is similar to the 128-EIA3 algorithm of the ZUC stream cipher [36]. We present the MAC computation in Algorithm 2.

Input: 1. Input message $M_0, \ldots, M_{B-1}$ of B bits.
2. Integrity key of l bytes.
Output: 32-bit MAC.

1 Run quad-RC4 algorithm to generate a keystream of $L = \lceil B/32 \rceil + 2$ words, denoted by $z_0, z_1, \ldots, z_{32L-1}$, where z_0 is the most significant bit of the first keystream word and z_{31} is the least significant bit;
2 **for** $i = 0, \ldots, 32L - 1$ **do**
3 | Set $Z_i = z_i||\ldots||z_{i+31}$;
end
4 Initialize tag $T = 0$;
5 **for** $i = 0, \ldots, B - 1$ **do**
6 | **if** $M_i = 1$ **then**
7 | | $T = T \oplus Z_i$;
| **end**
end
8 $T = T \oplus Z_B$;
9 Output MAC $= T \oplus Z_{32(L-1)}$;

Algorithm 2: Quad-RC4 MAC

4 Security Analysis

The security claim of a stream cipher is always a conjecture. Our design is no different. However, in this section, we put forward several arguments in favour of the strength of the cipher.

4.1 Resisting the Weaknesses Arising from RC4 KSA

According to the analysis presented in [20, Section 3.3], each of the four permutations obtained after the KSA$^+$ is uniformly random, having no bias towards the secret key or IV or absolute values in $\mathbb{Z}_{256}$. Moreover, since the publication of [20], no attack has been presented on the 3-layer KSA. Hence, we can conclude that when we combine the individual permutations to form a big state array, none of the RC4 KSA related weaknesses are carried forward.

According to [16], for stream ciphers using IV's, if the IV is shorter than the key, then the algorithm may be vulnerable against the Time Memory Trade-Off attack. Hence, we choose the IV size as the same as the secret key length.

4.2 Rationale Behind the Additional Swaps

Note that S has a total of 256 words, each of size 32 bits. The indices i and j are the same for the individual permutations. Hence, the swaps between the words $S[i]$ and $S[j]$ effectively causes 4 swaps between the corresponding bytes of the individual permutations. In other words, if we had only these types of swaps, then the relative distances between any two individual permutations would remain invariant for ever. This, in turn, causes the set of the words of S to remain unchanged, leading to potential vulnerabilities. To avoid this situation, we propose the additional swaps, as described in Section 3. One may observe that these additional swaps causes swaps between the bytes of the individual permutations. Hence, the set of the words of S continuously evolve and in principle, any of the 2^{32} possible 32-bit words may become a member of this set.

4.3 Resisting Distinguishing Attacks

The distinguishing attacks aim at identifying events that occur with probability away from that expected in a uniformly random stream. There are several biases [32, 22, 34, 21] in the initial keystream bytes of normal RC4. Since we update the individual RC4 states 1024 more times after the KSA before they are merged, it helps in preventing the propagation of the above biases in the keystream of our RC4 Combiner.

So far, the best long-term distinguisher reported on RC4, which persists even after any amount of RC4 output bytes are thrown away, is due to [23]. According to this work, occurrence of strings of pattern $ABTAB$ (A, B are bytes and T is a string of bytes of small length $G \leq 16$) are more frequent in RC4 keystream than in random stream. It is proved in [23, Theorem 1] that for RC4, the probability of the above event is $\frac{1}{N^2}(1 + \frac{e^{\frac{-4-8G}{N}}}{N})$, which is above $\frac{1}{N^2}$ (the probability of random occurrence). The distinguisher

is constructed by considering the fact that if $j - i = g$, then the entries at the locations i, $i+1$, $i+g$, $i+g+1$, and at the output indices $t = S[i]+S[j]$ corresponding to the rounds i and $i+1$ are not swapped in next few rounds with high probability. In

our case, the output is not selected from the index $t = S[i] + S[j]$. Rather t is split to create 4 indices into the array S. Thus, for the bias of [23] to manifest, more locations in the permutations would have to remain constant. Moreover, the additional swaps break the symmetry in the relative gaps between the permutations, as explained in Section 4.2, making the task of finding a distinguishing attack harder.

To evaluate the randomness of our keystream, we performed extensive experimentation with the NIST test suite [28] and all our keystreams passed all the NIST recommended tests.

4.4 Resisting State Recovery Attacks

The best known state recovery attack [26] on RC4 has complexity 2^{241}. Corresponding to a window of $w + 1$ keystream output bytes, all the j's are assumed to be known. Thus w many $S[i]$ values will be available from the differences between the consecutive values of j's. Then w many equations of type $S{-}1[z] = S[i] + S[j]$ can be formed, where each equation involves only two unknowns. The attack proceeds rounds in a guess and determine manner, following a chain of connected indices across different and it involves occasional guessing of some permutation bytes.

Our design prevents the strategy of [26] to work. We do not expose the permutation entries directly in the output. Each of the 4 words $S[t_1], \ldots, S[t_4]$ are composed of bytes from the individual permutations, but when these 4 words are XOR-ed together, no permutation byte is directly exposed in the keystream. Thus, to know the values of the permutation entries looking at the value of z, there is no other option than to go for all the possible choices.

5 Performance Evaluation

For experiments, we use two kinds of implementation platforms. The first one is general purpose processor. Recognizing the increasing use of stream ciphers in embedded platforms, different embedded processors are chosen as the second implementation platform. The runtime for embedded processors are obtained by running the executables on native simulators. The results of individual implementation and collective overall benchmarking are presented in the following subsections.

5.1 Experiment with Desktop Processor

We benchmark the algorithm presented in this paper on a 64-bit desktop processor running Ubuntu 12.04 on a AMD Phenom$^{\text{TM}}$II X6 1100T running at 3.3 GHz clock. The time is measured by running the *rdtsc* instruction, which returns the built-in time stamp counter. The code is compiled with standard gcc compiler, version 4.1.2. On the C code, few optimizations (such as to perform the computation and variable assignment simultaneously, selective loop unrolling) are implemented. For both the optimized and unoptimized versions of each implementation, the gcc compiler is run

with and without -O3 optimization option. Among the throughput results obtained, the best results are reported here. We report both keystream generation speed and encryption speed separately. The results are presented in the Table 1. For comparison with HC-128 [44], the fastest software stream cipher in the eSTREAM portfolio [5], we downloaded the C code from [43] and then compiled on our available machine.

Table 1. Performance Benchmarking on General Purpose Processors

Algorithm	Keystream Generation		Message Encryption	
	cycles/byte	Throughput (Gbps)	cycles/byte	Throughput (Gbps)
RC4	5.4	4.89	6.1	4.33
Quad-RC4	4.0	6.6	4.7	5.62
HC-128	3.9	6.77	4.5	5.87
HC-128 [44] (Intel Pentium M, 1.6 GHz)	–	–	3.1	4.20
HC-128 [5] (AMD Athlon 64 X2 4200+, 2.2 GHz)	–	–	2.9	6.07

It can be observed from Table 1 that the throughput increases monotonically with increasing instances of RC4 in our combiner model.

5.2 Experiment with Embedded and Customizable Processors

There has been several implementations of RC4 on embedded processors and accelerators [12, 15]. We select a customizable processor design framework [37] for our experiments. Since the prominent embedded processors support up to 32-bit datapath, RC4 and Quad-RC4 are used in these experiments.

In [12], RC4 is executed on a FPGA development board containing ARM922T processor and a keystream generation throughput of 21.24 cycles/byte is obtained. Depending on the CMOS technology library, different clock frequencies are recorded for ARM microprocessors. In a prominent System-on-Chip (SoC) design [39], ARM9 series of processors and ARM Cortex-A8 processors achieve clock frequency of 300 MHz and 500 MHz respectively. On FPGA-based implementation, ARM922T is reported to achieve 200 MHz clock frequency [1]. Considering the highest possible clock frequency, keystream generation throughput of RC4 [12] is 0.19 Gbps.

Customizable application-specific processors are increasingly used in modern SoCs for balancing performance and flexibility constraints. Synopsys Processor Designer [37] provides several starter designs for modeling such processors. We picked up a simple-scalar RISC processor considering the lack of instruction-level parallelism in the RC4 algorithm. It contains 6 pipeline stages, sixteen 32-bit registers, fully bypassed arithmetic logic unit. The instruction-set supports a wide variety of logical, arithmetic, control and load-store instructions. Both the program memory and data

memory are accessed synchronously. The tool-suite, which can be generated from the processor description, includes a C compiler. Synthesizable RTL code can be generated from the description, too.

For a single instance of RC4, the initial unoptimized C code resulted in a keystream generation throughput and message encryption throughput of 36 cycles/byte (0.2 Gbps) and 44 cycles/byte (0.16 Gbps) respectively. Similar to the general purpose processors, Quad-RC4 resulted in an improved keystream generation throughput of 32 cycles/byte (0.23 Gbps). Message encryption speed for Quad-RC4 is 36.2 cycles/byte (0.2 Gbps).

5.3 Overall Benchmarking

The platform, AMD Phenom^{TM}II X6 1100T, is synthesized at 45nm CMOS technology and achieves 9.78 Gbps. This is 2 times faster than plain RC4 keystream generation throughput. It is likely that higher throughput on general purpose computing platforms can be achieved by using more instances of RC4, if wider data-paths are available. For embedded and customizable processors, the maximum achievable RC4 keystream generation throughput is 0.2 Gbps and that for Quad-RC4 is 0.23 Gbps, indicating a similar trend in throughput improvement as observed in general purpose processors. The throughput improvement is little restricted due to the increased number of memory accesses. Given the a-priori knowledge of the memory distribution, memory partitioning can be done to improve the throughput of Quad-RC4 further.

Table 2. Keystream Generation Throughput for Various Computing Platforms

RC4 Instances	Target Platform	CMOS Technology	Clock Frequency	Keystream (Gbps)
Single	AMD Phenom^{TM}II X6 1100T	45nm	2.8 GHz	4.89
Quad	AMD Phenom^{TM}II X6 1100T	45nm	2.8 GHz	6.6
Single	Custom RISC Processor [37]	65nm	906 MHz	0.2
Quad	Custom RISC Processor [37]	65nm	906 MHz	0.23
Single	ASIC	65nm	1.92 GHz	30.72

The fastest reported hardware implementation for RC4 in 65nm CMOS technology achieves a throughput of 30.72 Gbps [33]. It can be noted that the throughput of hardware implementation could be further boosted by scaling down to 45nm CMOS technology and employing the principle of multiple RC4 instances as proposed here. The best results for different platforms are summarized in the Table 2. With increasing number of RC4 instances, the keystream generation throughput increases strongly.

6 How to Combine m RC4 States?

In this section, we propose a model of combining m RC4 states to generate $8m$ keystream bits in every round. For $m = 4$, the generic model yields an instance of Quad-RC4. For $m = 8$, the generic model yields a 64-bit variant which we refer to as *Octa-RC4*.

We start with m pseudo-random permutations over $\mathbb{Z}_{256}$, each generated by m RC4$^+$ key schedulings. Let us denote the permutations by $S_1, S_2, \ldots, S_m$. They are merged into a single array S of size 256, where the i-th entry of S is an $8m$-bit number, formed by concatenating the m many bytes $S_1[i]$, ..., $S_m[i]$.

Input: 1. m pseudo-random permutations over $\mathbb{Z}_{256}$.
2. No. of rounds R.
Output: $8mR$ pseudo-random keystream bits.
Output: $32R$ pseudo-random keystream bits.

Initializing the Indices:
1 $i = j = 0$;

Merge m Permutations in a Single Array
2 **for** $i = 0, \ldots, 255$ **do**
3 $\quad S[i] = (S_1[i] \ll 8(m-1)) \mid (S_{m-2}[i] \ll 16) \mid (S_{m-1}[i] \ll 8) \mid S_m[i]$;
end

Generating Keystream Words
4 **for** $r = 1, \ldots, R$ **do**
5 $\quad i = (i + 1) \bmod 256$;
6 $\quad j = (j + S_m[i]) \bmod 256$;
7 $\quad$ Swap($S[i]$, $S[j]$);
8 $\quad t = (S[i] + S[j]) \bmod 256$;
9 $\quad t_1 = t$ & 0xFF, $t_2 = (t \gg 8)$ & 0xFF, ..., $t_m = (t \gg 8(m-1))$ & 0xFF;
10 $\quad$ Output $z = h(S[t], S[i], S[j], S[t_1], S[t_2], \ldots, S[t_m])$;
11 $\quad \{a, b\}$ = Next pair of permutations in turn;
12 $\quad$ Swap($S_a[i]$, $S_a[t_a]$);
13 $\quad$ Swap($S_b[i]$, $S_b[t_b]$);
end

Algorithm 3: m-RC4: a generalized RC4 combiner

The complete algorithm for keystream generation is called m-RC4 and is presented in Algorithm 3. Like Quad-RC4, the index i is incremented by 1 modulo 256 and j is incremented pseudo-randomly using only one particular permutation, say S_m. At every round, we swap the i-th and the j-th word of S; in effect we swap the corresponding entries of each permutation. $S[i]$ and $S[j]$ are added to generate an $8m$-bit number t, ignoring the carry. Let the m bytes of t be denoted by $t_1, \ldots, t_m$. We propose that in Step 10, the output keystream word z is generated through a suitable function h of the quantities $S[t], S[i], S[j]$, and $S[t_a]$ for $a = 1, \ldots, m$. We have observed that if we simply extend the keystream generation logic of Quad-RC4, i.e., if h returns bitwise XOR of $S[t_a]$'s, $a = 1, \ldots, m$, then for $m = 8$ the keystream is

not perfectly random. How to design h to ensure good randomness properties without compromising throughput is a challenging open problem.

To break the symmetry in the swaps of the individual permutations, we propose to introduce some additional swaps similar to Quad-RC4. At every round, we select two permutations S_a and S_b. We then swap the i-th and the t_a-th bytes of S_a and swap the i-th and the t_b-th bytes of S_b. Note that t_a and t_b are the a-th and the b-th bytes of $t = t_1||\ldots||t_m$.

Two permutations out of m can be selected in $\binom{m}{2} = m(m-1)/2$ ways. We propose that each combination is chosen in turn and hence one particular pair of permutations is considered again after $O(m^2)$ rounds. Note that the structure of m-RC4, in the same way as Quad-RC4, ensures that the individual permutations remain permutations.

6.1 How to Select the Permutation Pairs for Additional Swaps?

Note that at every round of keystream generation, one has to select two permutations in turn and perform additional swaps. Out of m permutations, this can be done in $\binom{m}{2}$ ways. With respect to the combined array S, the choice of two permutations can be represented by a $8m$-bit mask $M = b_1||\ldots||b_m$, each b_i being a byte, where exactly two bytes are FF and all others are 00 (in HEX notation).

We can store all the $\binom{m}{2} = k$ (say) masks in memory and keep a modulo k counter to select each mask in turn. Alternatively, we can generate all the masks on the fly by rotating a small number of masks, what we call *generator* masks. For example, for $m = 4$, the $\binom{4}{2} = 6$ masks are FFFF0000, FF00FF00, FF0000FF, 00FFFF00, 00FF00FF, 0000FFFF. Instead of storing all of them, we can store the two generating masks, namely 0000FFFF and 00FF00FF. Byte-rotations of these two masks can generate all the masks.

We can formalize the above notion as follows. Consider the set of all $\binom{m}{2}$ many $8m$-bit masks to select the two permutations during the PRGA of the RC4 combiner.

Definition 1. *Given a mask M, we define a* **rotation** *operation denoted by $rot(M)$, which rotates the mask to the left by 8 bits.*

Note that left or right rotation does not matter, as long as one follows the same convention. Next, we define a relation ρ on the set of masks.

Definition 2. *Two masks M_1 and M_2 are* **ρ-related***, denoted by $M_1 \rho M_2$, if and only if one can be obtained from the other by a finite number of rotations.*

It is easy to show that ρ is reflexive, symmetric and transitive and hence we can state the following.

Theorem 1. *The relation ρ defined on the set of masks is an equivalence relation.*

Hence ρ will partition the set of masks into disjoint equivalence classes.

Definition 3. *For the set of $8m$-bit masks, any collection of one representative member from each equivalence class of ρ is called a set of* **generator masks**.

The following result gives the number of such generator masks.

Theorem 2. *For the set of $8m$-bit masks, where $m = 2^n$, there are exactly $m/2 = 2^{n-1}$ generator masks.*

Proof. In the masks, denote 'FF' as 1 and '00' as 0. Then the problem is equivalent to counting the number of equivalence classes for m-length bit-strings with exactly two 1's, the rotations being bit-wise. Consider m locations in such a bit-string as m points in a circle and we need to mark two points for the two 1's. After marking one point, the second point can be selected at a gap of 0 or 1 or 2, ..., up to $(m-1)/2$. Each of these $m/2$ selections correspond to one equivalence class. Hence the result follows. □

Thus, for Octa-RC4, we need only 4 generator masks to generate all possible $\binom{8}{2} = 28$ masks.

The exact number of members generated by each generator mask is given by Theorem 3 below.

Theorem 3. *For the set of $8m$-bit masks, there are exactly $m/2-1$ generator masks, each of which can generate m masks, and there is exactly 1 generator mask which can generate $m/2$ masks.*

Proof. Refer to the m-point circle in the proof of Theorem 2. Whenever the gap of two 1's is 0 or 1 or 2, ..., or $(m-1)/2-1$ in a pattern, such a pattern can be rotated m times before a repetition occurs. Hence each of the equivalence classes with the gaps 0 or 1 or 2, ..., or $(m-1)/2-1$ has exactly m members. Whenever the gap of two 1's is $(m-1)/2$, such a pattern can be rotated $m/2$ times before a repetition occurs. The equivalence class corresponding to this pattern has exactly $m/2$ members. Hence the result follows. □

According to Theorem 3, the total number of masks that can be generated by the generator masks is $(m/2-1)\cdot m+1\cdot m/2 = m^2/2-m/2 = \binom{m}{2}$, as expected. The idea of storing a small number of generator masks and simple operations like rotations to generate the other masks can be very useful in efficient hardware implementation of the scheme.

7 Conclusion and Future Work

We propose a generic method of combining 4 RC4 states to produce a 32-bit word-oriented stream cipher Quad-RC4. The scheme is scalable to modern processors of large word-width and is at least as secure as RC4. It also has comparable performance with HC-128.

How to combine an arbitrary (say, m) number of instances of RC4 to generate keystream with higher word-width without compromising security is an interesting open problem. In particular, it would be interesting to explore 64-bit and 128-bit extension of the keystream generator with respect to 64-bit desktop processors and 128-bit SSE registers and its AVX extensions in x86 architectures.

We also plan to study optimized implementation of the proposed model in wide-word SIMD processor as in GPGPU.

References

1. Altera Excalibur, `http://www.altera.com/products/devices/arm/system/exc-arm922t_architecture.html`
2. M. Akgün, P. Kavak, and H. Demirci, "New Results on the Key Scheduling Algorithm of RC4," in *INDOCRYPT*, vol. 5365 of *Lecture Notes in Computer Science*, pp. 40–52, 2008.
3. R. Basu, S. Maitra, G. Paul, and T. Talukdar, "On Some Sequences of the Secret Pseudo-random Index j in RC4 Key Scheduling," in *AAECC*, vol. 5527 of *LNCS*, pp. 137–148, 2009.
4. E. Biham and Y. Carmeli, "Efficient Reconstruction of RC4 Keys from Internal States," in *FSE*, vol. 5086 of *LNCS*, pp. 270–288, 2008.
5. `http://www.ecrypt.eu.org/stream/`
6. E. Biham and J. Seberry, "Py: A Fast and Secure Stream Cipher using Rolling Arrays," April, 2005. Available at `http://www.ecrypt.eu.org/stream/pyp2.html`.
7. E. Biham and J. Seberry, "Pypy: Another version of Py," June, 2006. Available at `http://www.ecrypt.eu.org/stream/pyp2.html`.
8. E. Biham and J. Seberry, "Tweaking the IV Setup of the Py Family of Stream Ciphers - The ciphers TPy, TPypy, and TPy6," January, 2007. Available at `http://www.ecrypt.eu.org/stream/pyp2.html`
9. A. Chattopadhyay and G. Paul, "Exploring Security-Performance Trade-offs during Hardware Accelerator Design of Stream Cipher RC4," in *IFIP/IEEE International Conference on VLSI-SoC*, accepted for publication, 2012.
10. S. R. Fluhrer and D. A. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator," in *FSE*, vol. 1978 of *LNCS*, pp. 19–30, 2000.
11. S. R. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," in *SAC*, vol. 2259 of *LNCS*, pp. 1–24, 2001.
12. N. Fournel, M. Minier and S. Ubeda, "Survey and Benchmark of Stream Ciphers for Wireless Sensor Networks," in *IFIP TC6 /WG8.8 /WG11.2 international conference on Information security theory and practices: smart cards, mobile and ubiquitous computing systems (WISTP)*, Damien Sauveron, Konstantinos Markantonakis, Angelos Bilas, and Jean-Jacques Quisquater (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 202–214, 2007.
13. J. D. Golic, "Linear Statistical Weakness of Alleged RC4 Keystream Generator," in *EUROCRYPT*, vol. 1233 of *LNCS*, pp. 226–238, 1997.
14. G. Gong, K. C. Gupta, M. Hell and Y. Nawaz, "Towards a General RC4-Like Keystream Generator," CISC, pp. 162–174, vol. 3822, Lecture Notes in Computer Science, Springer, 2005.
15. P. Hamalainen, J. Heikkinen, M. Hannikainen and T.D. Hamalainen, "Design of Transport Triggered Architecture Processors for Wireless Encryption," in *8th Euromicro Conference on Digital System Design*, pp. 144–152, 30 Aug.-3 Sept. 2005, DOI: 10.1109/DSD.2005.33.
16. J. Hong and P. Sarkar, " New Applications of Time Memory Data Tradeoffs," ASIACRYPT, pp. 353–372, vol. 3788, Lecture Notes in Computer Science, Springer, 2005.
17. P. Kitsos, G. Kostopoulos, N. Sklavos and O. Koufopavlou, "Hardware Implementation of the RC4 stream Cipher," in *46th IEEE Midwest Symposium on Circuits & Systems*, Cairo, Egypt, 2003.
18. A. Klein, "Attacks on the RC4 stream cipher," in *Des. Codes Cryptography*, vol. 48, no. 3, pp. 269–286, 2008.
19. L. R. Knudsen, W. Meier, B. Preneel, V. Rijmen, and S. Verdoolaege, "Analysis Methods for (Alleged) RC4," in *ASIACRYPT*, vol. 1514 of *LNCS*, pp. 327–341, 1998.

20. S. Maitra and G. Paul, "Analysis of RC4 and Proposal of Additional Layers for Better Security Margin," in *INDOCRYPT*, vol. 5365 of *LNCS*, pp. 27–39, 2008.
21. S. Maitra, G. Paul, and S. Sen Gupta, "Attack on Broadcast RC4 Revisited," in *FSE*, vol. 6733 of *LNCS*, pp. 199–217, 2011.
22. I. Mantin and A. Shamir, "A Practical Attack on Broadcast RC4," in *FSE*, vol. 2355 of *LNCS*, pp. 152–164, 2001.
23. I. Mantin, "Predicting and Distinguishing Attacks on RC4 Keystream Generator," in *EUROCRYPT*, vol. 3494 of *LNCS*, pp. 491–506, 2005.
24. I. Mantin, "A Practical Attack on the Fixed RC4 in the WEP Mode," in *ASIACRYPT*, vol. 3788 of *LNCS*, pp. 395–411, 2005.
25. D. P. Matthews Jr., "Methods and apparatus for accelerating ARC4 processing," US Patent Number 7403615, Morgan Hill, CA, July, 2008. `http://www.freepatentsonline.com/7403615.html`
26. A. Maximov and D. Khovratovich, "New State Recovery Attack on RC4," in *CRYPTO*, vol. 5157 of *LNCS*, pp. 297–316, 2008.
27. I. Mironov, "(Not So) Random Shuffles of RC4," in *CRYPTO*, vol. 2442 of *LNCS*, pp. 304–319, 2002.
28. National Institute of Standards and Technology. "A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications, that describes the test suite," NIST Special Publication 800-22rev1a, April 2010. Available at `http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html`.
29. Y. Nawaz, K. C. Gupta and G. Gong, "A 32-bit RC4-like keystream generator," in *Technical Report CACR 2005-19*, Center for Applied Cryptographic Research, University of Waterloo, 2005. Also appears in IACR Eprint Server, eprint.iacr.org, number 2005/175, June 12, 2005.
30. S. Paul and B. Preneel, "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher," in *FSE*, vol. 3017 of *LNCS*, pp. 245–259, 2004.
31. G. Paul and S. Maitra, "Permutation After RC4 Key Scheduling Reveals the Secret Key," in *SAC*, vol. 4876 of *LNCS*, pp. 360–377, 2007.
32. A. Roos, "A class of weak keys in the RC4 stream cipher." Two posts in sci.crypt, message-id `43u1eh$1j3@hermes.is.co.za` and `44ebge$llf@hermes.is.co.za`, 1995. Available at `http://www.impic.org/papers/WeakKeys-report.pdf`.
33. S. Sen Gupta, A. Chattopadhyay, K. Sinha, S. Maitra and B.P. Sinha, "High Performance Hardware Implementation for RC4 Stream Cipher," in *IEEE Transactions on Computers*, 2012, DOI: 10.1109/TC.2012.19.
34. P. Sepehrdad, S. Vaudenay, and M. Vuagnoux, "Discovery and Exploitation of New Biases in RC4," in *SAC*, vol. 6544 of *LNCS*, pp. 74–91, 2010.
35. P. Sepehrdad, S. Vaudenay, and M. Vuagnoux, "Statistical Attack on RC4 - Distinguishing WPA," in *EUROCRYPT*, vol. 6632 of *LNCS*, pp. 343–363, 2011.
36. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 1: 128-EEA3 and 128-EIA3 Specification. Version 1.7, Dec 30, 2011.
37. Synopsys Processor Designer, `http://www.synopsys.com/Systems/BlockDesign/processorDev`
38. E. Tews and M. Beck, "Practical attacks against WEP and WPA," in *WISEC*, pp. 79–86, ACM, 2009.
39. TI's ARM Portfolio of Products, `http://www.ti.com/lit/ml/sprt539/sprt539.pdf`
40. T. H. Tran, L. Lanante, Y. Nagao, M. Kurosaki, H. Ochi, "Hardware Implementation of High Throughput RC4 Algorithm," IEEE ISCAS, 2012. (abstract available online)
41. Y. Tsunoo, T. Saito, H. Kubo and T. Suzaki, "A Distinguishing Attack on a Fast Software-Implemented RC4-Like Stream Cipher," in *IEEE Transactions on Information Theory*, pp. 3250–3255, vol. 53, issue 9, September 2007.
42. S. Vaudenay and M. Vuagnoux, "Passive-Only Key Recovery Attacks on RC4," in *SAC*, vol. 4876 of *LNCS*, pp. 344–359, 2007.
43. Stream Ciphers HC-128 and HC-256, `http://www3.ntu.edu.sg/home/wuhj/research/hc/index.html`.
44. H. Wu, "The Stream Cipher HC-128," Available at `http://www.ecrypt.eu.org/stream/hcp3.html`.
45. B. Zoltak, "VMPC One-Way Function and Stream Cipher," in *FSE*, vol. 3017 of *LNCS*, pp. 210–225, Springer.