A New Stream Cipher HC-256

Hongjun Wu

Institute for Infocomm Research
21 Heng Mui Keng Terrace, Singapore 119613
hongjun@i2r.a-star.edu.sg

Abstract. Stream cipher HC-256 is proposed in this paper. It generates keystream from a 256-bit secret key and a 256-bit initialization vector. HC-256 consists of two secret tables, each one with 1024 32-bit elements. The two tables are used as S-Box alternatively. At each step one element of a table is updated and one 32-bit output is generated. The encryption speed of the C implementation of HC-256 is about 1.9 bit per clock cycle (4.2 clock cycle per byte) on the Intel Pentium 4 processor.

1 Introduction

Stream ciphers are used for shared-key encryption. The modern software efficient stream ciphers can run 4-to-5 times faster than block ciphers. However, very few efficient and secure stream ciphers have been published. Even the most widely used stream cipher RC4 [25] has several weaknesses [14, 16, 22, 9, 10, 17, 21]. In the recent NESSIE project all the six stream cipher submissions cannot meet the stringent security requirements [23]. In this paper we aim to design a very simple, secure, software-efficient and freely-available stream cipher.

HC-256 is the stream cipher we proposed in this paper. It consists of two secret tables, each one with 1024 32-bit elements. At each step we update one element of a table with non-linear feedback function. Every 2048 steps all the elements of the two tables are updated. At each step, HC-256 generates one 32-bit output using the 32-bit-to-32-bit mapping similar to that being used in Blowfish [28]. Then the linear masking is applied before the output is generated.

In the design of HC-256, we take into consideration the superscalar feature of modern (and future) microprocessors. Without compromising the security, we try to reduce the dependency between operations. The dependency between the steps is reduced so that three consecutive steps can be computed in parallel. At each step, three parallel additions are used in the feedback function and three additions are used to combine the four table lookup outputs instead of the addition-xor-addition being used in Blowfish (similar idea has been suggested by Schneier and Whiting to use three xors to combine those four terms [29]).

With the high degree of parallelism, HC-256 runs very efficiently on the modern processor. We implemented HC-256 in C and tested its performance on the Pentium 4 processor. The encryption speed of HC-256 reaches 1.93 bit/cycle.

This paper is organized as follows. We introduce HC-256 in Section 2. The security of HC-256 is analyzed in Section 3. Section 4 discusses the implementation and performance of HC-256. Section 5 concludes this paper.

2 Stream Cipher HC-256

In this section, we describe the stream cipher HC-256. From a 256-bit key and a 256-bit initialization vector, it generates keystream with length up to 2^{128} bits.

2.1 Operations, variables and functions

The following operations are used in HC-256:

$+$	: $x + y$ means $x + y \bmod 2^{32}$, where $0 \leq x < 2^{32}$ and $0 \leq y < 2^{32}$
$\boxminus$	: $x \boxminus y$ means $x - y \bmod 1024$
$\oplus$	: bit-wise exclusive OR
$\|\|$	: concatenation
$\gg$	: right shift operator. $x \gg n$ means x being right shifted n bits.
$\ll$	: left shift operator. $x \ll n$ means x being left shifted n bits.
$\ggg$	: right rotation operator. $x \ggg n$ means $((x \gg n) \oplus (x \ll (32-n))$ where $0 \leq n < 32$, $0 \leq x < 2^{32}$.

Two tables P and Q are used in HC-256. The key and the initialization vector of HC-256 are denoted as K and IV. We denote the keystream being generated as s.

P	: a table with 1024 32-bit elements. Each element is denoted as $P[i]$ with $0 \leq i \leq 1023$.
Q	: a table with 1024 32-bit elements. Each element is denoted as $Q[i]$ with $0 \leq i \leq 1023$.
K	: the 256-bit key of HC-256.
IV	: the 256-bit initialization vector of HC-256.
s	: the keystream being generated from HC-256. The 32-bit output of the ith step is denoted as s_i. Then $s = s_0\|\|s_1\|\|s_2\|\| \cdots$

There are six functions being used in HC-256. $f_1(x)$ and $f_2(x)$ are the same as the $\sigma_0^{\{256\}}(x)$ and $\sigma_1^{\{256\}}(x)$ being used in the message schedule of SHA-256 [24]. For $g_1(x)$ and $h_1(x)$, the table Q is used as S-box. For $g_2(x)$ and $h_2(x)$, the table P is used as S-box.

$$\begin{aligned}
f_1(x) &= (x \ggg 7) \oplus (x \ggg 18) \oplus (x \gg 3) \\
f_2(x) &= (x \ggg 17) \oplus (x \ggg 19) \oplus (x \gg 10) \\
g_1(x,y) &= ((x \ggg 10) \oplus (y \ggg 23)) + Q[(x \oplus y) \bmod 1024] \\
g_2(x,y) &= ((x \ggg 10) \oplus (y \ggg 23)) + P[(x \oplus y) \bmod 1024] \\
h_1(x) &= Q[x_0] + Q[256 + x_1] + Q[512 + x_2] + Q[768 + x_3] \\
h_2(x) &= P[x_0] + P[256 + x_1] + P[512 + x_2] + P[768 + x_3]
\end{aligned}$$

where $x = x_3||x_2||x_1||x_0$, x is a 32-bit word, x_0, x_1, x_2 and x_3 are four bytes. x_3 and x_0 denote the most significant byte and the least significant byte of x, respectively.

2.2 Initialization process (key and IV setup)

The initialization process of HC-256 consists of expanding the key and initialization vector into P and Q (similar to the message setup in SHA-256) and running the cipher 4096 steps without generating output.

1. Let $K = K_0 || K_1 || \cdots || K_7$ and $IV = IV_0 || IV_1 || \cdots || IV_7$, where each K_i and IV_i denotes a 32-bit number. The key and IV are expanded into an array W_i $(0 \le i \le 2559)$ as:

$$W_i = \begin{cases} K_i & 0 \le i \le 7 \\ IV_{i-8} & 8 \le i \le 15 \\ f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i & 16 \le i \le 2559 \end{cases}$$

2. Update the tables P and Q with the array W.

$$P[i] = W_{i+512} \qquad \text{for } 0 \le i \le 1023$$
$$Q[i] = W_{i+1536} \qquad \text{for } 0 \le i \le 1023$$

3. Run the cipher (the keystream generation algorithm in Subsection 2.3) 4096 steps without generating output.

The initialization process completes and the cipher is ready to generate keystream.

2.3 The keystream generation algorithm

At each step, one element of a table is updated and one 32-bit output is generated. An S-box is used to generate only 1024 outputs, then it is updated in the next 1024 steps. The keystream generation process of HC-256 is given below ("$\boxminus$" denotes "$-$" modulo 1024, s_i denotes the output of the i-th step).

```
$i = 0$;
repeat until enough keystream bits are generated.
{
    $j = i \bmod 1024$;
    if $(i \bmod 2048) < 1024$
    {
        $P[j] = P[j] + P[j \boxminus 10] + g_1(\, P[j \boxminus 3], P[j \boxminus 1023]\,)$;
        $s_i = h_1(\, P[j \boxminus 12]\,) \oplus P[j]$;
    }
    else
    {
        $Q[j] = Q[j] + Q[j \boxminus 10] + g_2(\, Q[j \boxminus 3], Q[j \boxminus 1023]\,)$;
        $s_i = h_2(\, Q[j \boxminus 12]\,) \oplus Q[j]$;
    }
    end-if
    $i = i + 1$;
}
end-repeat
```

2.4 Encryption and decryption

The keystream is XORed with the message for encryption. The decryption is to XOR the keystream with the ciphertext.

3 Security Analysis of HC-256

We start with a brief review of the attakcs on stream ciphers. Many stream ciphers are based on the linear feedback shift registers (LFSRs) and a number of correlation attacks, such as [30, 31, 19, 11, 20, 4, 15], were developed to analyze them. Later Golić [12] devised the linear cryptanalysis of stream ciphers. That technique could be applied to a wide range of stream ciphers. Recently Coppersmith, Halevi and Jutla [6] developed the distinguishing attacks (the linear attack and low diffusion attack) on stream ciphers with linear masking.

The correlation attacks cannot be applied to HC-256 because HC-256 uses non-linear feedback functions to update the two tables P and Q. The output function of HC-256 uses the 32-bit-to-32-bit mapping similar to that being used in Blowfish. The analysis on Blowfish shows that it is extremely difficult to apply linear cryptanalysis [18] to the large secret S-box. The large secret S-box of HC-256 is updated during the keystream generation process and it is almost impossible to develop linear relations linking the input and output bits of the S-box. Vaudenay has found some differential weakness of the randomly generated large S-box [32]. But it is very difficult to launch differential cryptanalysis [2] against HC-256 since it is a synchronous stream cipher for which the keystream generation is independent of the message.

In this section, we will analyze the security of the secret key, the randomness of the keystream, and the security of the initialization process.

3.1 Period

The 65547-bit state of HC-256 ensures that the period of the keystream is extremely large. But the exact period of HC-256 is difficult to predict. The average period of the keystream is estimated to be about 2^{65546} (if we assume that the invertible next-state function of HC-256 is random). The large number of states also completely eliminates the threat of time-memory tradeoff attack on stream ciphers [1, 13].

3.2 The security of the key

We begin with the study of a modified version of HC-256 (with no linear masking). Our analysis shows that even for this weak version of HC-256, it is impossible to recover the secret key faster than exhaustive key search. The reason is that the keystream is generated from a highly non-linear function ($h_1(x)$ or $h_2(x)$), so the keystream leaks very small amount of information at each step. Recovering P and Q requires the partial information leaked from a lot of steps. Because the tables are updated in a highly non-linear way, it is difficult to retrieve the informtion of P and Q from those leaked information.

HC-256 with no linear masking. For HC-256 with no linear masking, the output at the ith step is generated as $s_i = h_1(\, P[i \boxminus 12]\,)$ or $s_i = h_2(\, Q[i \boxminus 12]\,)$. If two outputs generated from the same S-box are equal, then very likely those two inputs to the S-box are equal. According to the analysis on the randomness of the outputs of $h_1(x)$ and $h_2(x)$ given in Subsection 3.3, $s_{2048\times\alpha+i} = s_{2048\times\alpha+j}$ $(0 \le i < j < 1024)$ with probability about 2^{-31}. If $s_{2048\times\alpha+i} = s_{2048\times\alpha+j}$, then at the $(2048 \times \alpha + j)$-th step, $P[i \boxminus 12] = P[j \boxminus 12]$ with probability about 0.5 (31-bit information of the table P is leaked). We note that for every 1024 steps in the range $(2048 \times \alpha, 2048 \times \alpha + 1024)$, the same S-box is used in $h_1(x)$. The probability that there are two equal outputs is $\binom{1024}{2} \times 2^{-31} \approx 2^{-12}$. In average each output leaks $\frac{2^{-12}\times 31}{1024} \approx 2^{-17}$ bit information of the table P. To recover P, we need to analyze at least $\frac{1024\times 32}{2^{-17}} \approx 2^{32}$ outputs. Recovering P from those 2^{32} outputs involves very complicated non-linear equations and solving them is computationally infeasible. Recovering Q is as difficult as recovering P. We note that the table Q is used as S-box to update P, and vice versa. P and Q interact in such a complicated way and recovering them from the keystream cannot be faster than exhaustive key search.

HC-256. The analysis above shows that the secret key of HC-256 with no linear masking is secure. With the linear masking, the information leakage is greatly reduced and it would be even more difficult to recover the secret key from the keystream. We thus conclude that the key of HC-256 cannot be recovered faster than exhaustive key search.

3.3 Randomness of the keystream

In this subsection, we investigate the randomness of the keystream of HC-256. Because the large, secret and frequently updated S-boxes are used in the cipher, the efficient attack is to analyze the randomness of the overall 32-bit words. Under this guideline, we developed some attacks against HC-256 with no linear masking. Then we show that the linear masking eliminates those threats.

3.3.1 Keystream of HC-256 with no linear masking .

The attacks on HC-256 with no linear masking is to investigate the security weaknesses in the output and feedback functions. We developed two attacks against HC-256 with no linear masking.

Weakness of $h_1(x)$ and $h_2(x)$. For HC-256 with no linear masking, the output is generated as $s_i = h_1(P[i \boxminus 12])$ or $s_i = h_2(Q[i \boxminus 12])$. Because there is no difference between the analysis of $h_1(x)$ and $h_2(x)$, we use $h(x)$ to refer $h_1(x)$ and $h_2(x)$ here. Assume that $h(x)$ is a 32-bit-to-32-bit S-box $H(x)$ with randomly generated secret elements and the inputs to H are randomly generated. Because the elements of the $H(x)$ are randomly generated, the output of $H(x)$ is not uniformly distributed. If a lot of outputs are generated from $H(x)$, some values in the range $[0, 2^{32})$ never appear and some appear with probability larger than 2^{-32}. Then it is straightforward to distinguish the outputs from random signal. However each $H(x)$ in HC-256 is used to generate only 1024 outputs, then it gets updated. The direct computation of the distribution of the outputs of $H(x)$ from those 1024 outputs cannot be successful. Instead, we consider the collision between the outputs of $H(x)$. The following theorem gives the collision rate of the outputs of $H(x)$.

Theorem 1. *Let H be an m-bit-to-n-bit S-box and all those n-bit elements are randomly generated, where $m \geq n$ and n is a large integer. Let x_1 and x_2 be two m-bit random inputs to H. Then $H(x_1) = H(x_2)$ with probability about $2^{-n} + 2^{-m}$.*

Proof. If $x_1 = x_2$, then $H(x_1) = H(x_2)$. If $x_1 \neq x_2$, then $H(x_1) = H(x_2)$ with probability 2^{-n}. $x_1 = x_2$ with probability 2^{-m} and $x_1 \neq x_2$ with probability $1 - 2^{-m}$. The probability that $H(x_1) = H(x_2)$ is $2^{-m} + (1 - 2^{-m}) \times 2^{-n} \approx 2^{-n} + 2^{-m}$.

Attack 1. According to Theorem 1, for the 32-bit-to-32-bit S-box H, the collision rate of the outputs is $2^{-32} + 2^{-32} = 2^{-31}$. With 2^{35} pairs of $(H(x_1), H(x_2))$, we can distinguish the output from random signal with success rate 0.761. (The success rate can be improved to 0.996 with 2^{36} pairs.) Note that only 1024 outputs are generated from the same S-box H, so 2^{26} outputs are needed to distinguish the keystream of HC-256 with no linear masking.

Experiment. To compute the collision rate of the outputs of HC-256 (with no linear masking), we generated 2^{39} outputs (2^{48} pairs). The collision rate is $2^{-31} - 2^{-40.09}$. The experiment confirms that the collision rate of the outputs of $h(x)$ is very close to 2^{-31}, and approximating $h(x)$ with randomly generated S-box has negligible effect on the attack.

Remarks. The distinguishing attack above can be slightly improved if we consider the differential attack on Blowfish. Vaudenay [32] has pointed out that the collision in a randomly generated S-box in Blowfish can be applied to distinguish the outputs of Blowfish with reduced round number (8 rounds). The basic idea of Vaudenay's differential attack is that if $Q[i] = Q[j]$ for $0 \leq i, j < 256$, $i \neq j$, then for $a_0 \oplus a'_0 = i \oplus j$, $h_1(a_3||a_2||a_1||a_0) = h_1(a_3||a_2||a_1||a'_0)$ with probability 2^{-7}, where each a_i denotes an 8-bit number. We can detect the collision in the S-box with success rate 0.5 since that S-box Q is used as inputs to $h_2(x)$ to produce 1024 outputs. If $Q[i] = Q[j]$ for $256\alpha \leq i, j < 256\alpha + 256$, $0 \leq \alpha < 4$, $i \neq j$, and x_1 and x_2 are two random inputs (note that we cannot introduce or identify inputs with particular difference to $h(x)$), then the probability that $h_1(x_1) = h_1(x_2)$ becomes $2^{-31} + 2^{-32}$. However the chance that there is one useful collision in the S-box is only $\frac{\binom{256}{2} \times 4}{2^{32}} = 2^{-15}$. The average collision rate becomes $2^{-15} \times (2^{-31} + 2^{-32}) + (1 - 2^{-15}) \times 2^{-31} = 2^{-31} + 2^{-47}$. The increase in collision rate is so small that the collision in the S-box has negligible effect on this attack.

Weakness of the feedback function. The table P is updated with the non-linear feedback function $P[i \bmod 1024] = P[i \bmod 1024] + P[i \boxminus 10] + g_1(P[i \boxminus 3], P[i \boxminus 1023])$. The following attack is to distinguish the keystream by exploiting this relation.

Attack 2. Assume that the $h(x)$ is a one-to-one mapping function. Consider two groups of outputs $(s_i, s_{i-3}, s_{i-10}, s_{i-2047}, s_{i-2048})$ and $(s_j, s_{j-3}, s_{j-10}, s_{j-2047}, s_{j-2048})$. If $i \neq j$ and $1024 \times \alpha + 10 \leq i, j < 1024 \times \alpha + 1023$, they are equal with probability about 2^{-128}. The collision rate is 2^{-160} if the outputs are truely random. 2^{-128} is much larger than 2^{-160}, so the keystream can be distinguished from random signal with about 2^{128} pairs of such five-tuple groups of outputs. Note that the S-box is updated every 1024 steps, 2^{119} outputs are needed in the attack.

The two attacks given above show that the HC-256 with no linear masking does not generate secure keystream.

3.3.2 Keystream of HC-256 .

With the linear masking being applied, it is no longer possible to exploit those two weaknesses separately and the attacks given above cannot be applied directly. We need to remove the linear masking first. We recall that at the ith step, if $(i \bmod 2048) < 1024$, the table P is updated as

$$P[i \bmod 1024] = P[i \bmod 1024] + P[i \boxminus 10] + g_1(P[i \boxminus 3], P[i \boxminus 1023])$$

We know that $s_i = h_1(P[i \boxminus 12]) \oplus P[i \bmod 1024]$. For $10 \leq (i \bmod 2048) < 1023$, this feedback function can be written alternatively as

$$\begin{aligned} s_i \oplus h_1(z_i) = (s_{i-2048} \oplus h_1'(z_{i-2048})) + (s_{i-10} \oplus h_1(z_{i-10})) + \\ g_1(s_{i-3} \oplus h_1(z_{i-3}), s_{i-2047} \oplus h_1'(z_{i-2047})) \end{aligned} \tag{1}$$

where $h_1(x)$ and $h_1'(x)$ indicate two different functions since they are related to different S-boxes; z_j denotes the $P[j \boxminus 12]$ at the j-th step. The linear masking is removed successfully in (1). However, it is very difficult to apply (1) directly to distinguish the keystream. To simplify the analysis, we attack a weak version of (1). We replace all the '+' in the feedback function with '$\oplus$' and write (1) as

$$\begin{aligned} & s_i \oplus s_{i-2048} \oplus s_{i-10} \oplus (s_{i-3} \ggg 10) \oplus (s_{i-2047} \ggg 23) \\ & = h_1(z_i) \oplus h_1'(z_{i-2048})) \oplus h_1(z_{i-10})) \oplus (h_1(z_{i-3}) \ggg 10) \oplus \\ & \quad \oplus (h_1'(z_{i-2047}) \ggg 23) \oplus Q[r_i], \end{aligned} \tag{2}$$

where $r_i = (s_{i-3} \oplus h_1(z_{i-3}) \oplus s_{i-2047} \oplus h_1'(z_{i-2047})) \bmod 1024$. Because of the random nature of $h_1(x)$ and Q, the right hand side of (2) is not uniformly distributed. But each S-box is used in only 1024 steps, these 1024 outputs are not sufficient to compute the distribution of $s_i \oplus s_{i-2048} \oplus s_{i-10} \oplus (s_{i-3} \ggg 10) \oplus (s_{i-2047} \ggg 23)$. Instead we need to study the collision rate. The effective way is to eliminate the term $h_1(z_i)$ before analyzing the collision rate.

Replace the i with $i+10$. For $10 \le i \bmod 2048 < 1013$, (2) can be written as

$$\begin{aligned} & s_{i+10} \oplus s_{i-2038} \oplus s_i \oplus (s_{i+7} \ggg 10) \oplus (s_{i-2037} \ggg 23) \\ & = h_1(z_{i+10}) \oplus h_1'(z_{i-2038})) \oplus h_1(z_i) \oplus (h_1(z_{i+7}) \ggg 10) \oplus \\ & \quad \oplus (h_1'(z_{i-2037}) \ggg 23) \oplus Q[r_{i+10}] \end{aligned} \tag{3}$$

For the left-hand sides of (2) and (3) to be equal, i.e., for the following equation

$$\begin{aligned} & s_i \oplus s_{i-2048} \oplus s_{i-10} \oplus (s_{i-3} \ggg 10) \oplus (s_{i-2047} \ggg 23) = \\ & s_{i+10} \oplus s_{i-2038} \oplus s_i \oplus (s_{i+7} \ggg 10) \oplus (s_{i-2037} \ggg 23) \end{aligned} \tag{4}$$

to hold, we require that (after eliminating the term $h_1(z_i)$)

$$\begin{aligned} & h_1(z_{i-10}) \oplus h_1'(z_{i-2048}) \oplus (h_1(z_{i-3}) \ggg 10) \\ & \quad \oplus (h_1'(z_{i-2047}) \ggg 23) \oplus Q[r_i] \\ & = h_1(z_{i+10}) \oplus h_1'(z_{i-2038}) \oplus (h_1(z_{i+7}) \ggg 10) \\ & \quad \oplus (h_1'(z_{i-2037}) \ggg 23) \oplus Q[r_{i+10}] \end{aligned} \tag{5}$$

For $22 \le i \bmod 2048 < 1013$, we note that $z_{i-10} = z_i \oplus z_{i-2048} \oplus (z_{i-3} \ggg 10) \oplus (z_{i-2047} \ggg 23)$, and $z_{i+10} = z_i \oplus z_{i-2038} \oplus (z_{i+7} \ggg 10) \oplus (z_{i-2037} \ggg 23)$. Approximate (5) as

$$H(x_1) = H(x_2) \tag{6}$$

where H denotes a random secret 106-bit-to-32-bit S-box, x_1 and x_2 are two 106-bit random inputs, $x_1 = z_{i-3}||z_{i-2047}||z_{i-2048}||r_i$ and $x_2 = z_{i+7}||z_{i-2037}||z_{i-2038}$ $||r_{i+10}$. (The effect of z_i is included in H.) According to Theorem 1, (6) holds with probability $2^{-32} + 2^{-106}$. So (4) holds with probability $2^{-32} + 2^{-106}$. We approximate the binomial distribution with the normal distribution. The mean $\mu = Np$ and the standard deviation $\sigma = \sqrt{Np(1-p)}$, where N is the total number of equations (4), and $p = 2^{-32} + 2^{-106}$. For random signal, $p' = 2^{-32}$, the mean $\mu' = Np'$ and the standard deviation $\sigma' = \sqrt{Np'(1-p')}$. If $|u - u'| > 2(\sigma + \sigma')$, i.e. $N > 2^{184}$, the output of the cipher can be distinguished from random signal with success rate 0.9772.

After verifying the validity of 2^{184} equations (4), we can successfully distinguish the keystream from random signal. We note that the S-box is updated every 1024 steps, so only about 2^{10} equations (4) can be obtained from 1024 steps in the range $1024 \times \alpha \le i < 1024 \times \alpha + 1024$. To distinguish the keystream from random signal, 2^{184} outputs are needed in the attack.

The attack above can be improved by exploiting the relation $r_i = (s_{i-3} \oplus h_1(z_{i-3}) \oplus s_{i-2047} \oplus h_1'(z_{i-2047})) \bmod 1024$. If $(s_{i-3} \oplus s_{i-2047}) \bmod 1024 = (s_{i+7} \oplus s_{i-2037}) \bmod 1024$, then (6) holds with probability $2^{-32} + 2^{-96}$ and 2^{164} equations (4) are needed in the attack. Note that only about one equation (4) can now be obtained from 1024 steps in the range $1024 \times \alpha \leq i < 1024 \times \alpha + 1024$. To distinguish the keystream from random signal, 2^{174} outputs are needed in the attack.

We note that the attack above can only be applied to HC-256 with all the '+' in the feedback function being replaced with '$\oplus$'. To distinguish the keystream of HC-256, more than 2^{174} outputs are needed. So we conclude that it is impossible to distinguish a 2^{128}-bit keystream of HC-256 from random signal.

3.4 Security of the initialization process (key/IV setup)

The initialization process of the HC-256 consists of two stages, as given in Subsection 2.2. We expand the key and IV into P and Q. At this stage, every bit of the key/IV affects all the bits of the two tables and any difference in the related keys/IVs results in uncontrollable differences in P and Q. Then we run the cipher 4096 steps without generating output so that the P and Q become more random. After the initialization process, we expect that any difference in the keys/IVs would not result in biased keystream.

4 Implementation and Performance of HC-256

The direct C implementation of the encryption algorithm given in Subsection 2.3 runs at about 0.6 bit/cycle on the Pentium 4 processor. The program size is very small but the speed is only about 1.5 times that of AES [7]. At each step in the direct implementation, we need to compute ($i \bmod 2048$), $i \boxminus 3$, $i \boxminus 10$ and $i \boxminus 1023$. And at each step there is a branch decision based on the value of ($i \bmod 2048$). These operations affect greatly the encryption speed. The optimization process is to reduce the amount of these operations.

4.1 The optimized implementation of HC-256

This subsection describes the optimized C implementation of HC-256 given in Appendix B ("hc256.h"). In the optimized code, loop unrolling is used and only one branch decision is made for every 16 steps. The experiment shows that the branch decision in the optimized code affects the encryption speed by less than one percent.

There are several fast implementations of the feedback functions of P and Q. We use the implementation given in Appendix B because it achieves the best consistency on different platforms. The details of that implementation are given below. The feedback function of P is given as

$$P[i \bmod 1024] = P[i \bmod 1024] + P[i \boxminus 10] + g_1(\,P[i \boxminus 3], P[i \boxminus 1023]\,)$$

A register X containing 16 elements is introduced for P. If ($i \bmod 2048$) < 1024 and $i \bmod 16 = 0$, then at the begining of the ith step, $X[j] = P[(i - 16 +$

j) mod 1024] for $j = 0, 1, \cdots 15$, i.e. the X contains the values of $P[i \boxminus 16], P[i \boxminus 15], \cdots, P[i \boxminus 1]$. In the 16 steps starting from the ith step, the P and X are updated as

$$\begin{aligned}
P[i] &= P[i] + X[6] + g_1(\,X[13], P[i+1]\,);\\
X[0] &= P[i];\\
P[i+1] &= P[i+1] + X[7] + g_1(\,X[14], P[i+2]\,);\\
X[1] &= P[i+1];\\
P[i+2] &= P[i+2] + X[8] + g_1(\,X[15], P[i+3]\,);\\
X[2] &= P[i+2];\\
P[i+3] &= P[i+3] + X[9] + g_1(\,X[0], P[i+4]\,);\\
X[3] &= P[i+3];\\
&\cdots\\
P[i+14] &= P[i+14] + X[4] + g_1(\,X[11], P[i+15]\,);\\
X[14] &= P[i+14];\\
P[i+15] &= P[i+15] + X[5] + g_1(\,X[12], P[(i+1) \bmod 1024]\,);\\
X[15] &= P[i+15];
\end{aligned}$$

Note that at the ith step, two elements of $P[i \boxminus 10]$ and $P[i \boxminus 3]$ can be obtained directly from X. Also for the output function $s_i = h_1(P[i \boxminus 12]) \oplus P[i \bmod 1024]$, the $P[i \boxminus 12]$ can be obtained from X. In this implementation, there is no need to compute $i \boxminus 3$, $i \boxminus 10$ and $i \boxminus 12$.

A register Y with 16 elements is used in the implementation of the feedback function of Q in the same way as that given above.

To reduce the memory requirement and the program size, the initialization process implemented in Appendix B is not as straightforward as that given in Subsection 2.2. To reduce the memory requirement, we do not implement the array W in the program. Instead we implement the key and IV expansion on P and Q directly. To reduce the program size, we implement the feedback functions of those 4096 steps without involving X and Y.

4.2 Performance of HC-256

Encryption Speed. We use the C codes given in Appendix B and C to measure the encryption speed. The processor used in the test is Pentium 4 (2.4 GHz, 8 KB Level 1 data cache, 512 KB Level 2 cache, no hyper-threading). The speed is measured by repeatedly encrypting the same 512-bit buffer for 2^{26} times (The buffer is defined as 'static unsigned long DATA[16]' in Appendix C). The encryption speed is given in Table 1.

The C implementation of HC-256 is faster than the C implementations of almost all the other stream ciphers. (However different designers may have made different efforts to optimize their codes. And the encryption speed may be measured in different ways. So the speed comparison is not absolutely accurate.)

SEAL [26] is a software-efficient cipher and its C implementation runs at the speed of about 1.6 bit/cycle on Pentium III processor. Scream [5] runs at about the same speed as SEAL. The C implementation of SNOW2.0 [8] runs at about 1.67 bit/cycle on Pentium 4 processor. TURING [27] runs at about 1.3 bit/cycle on the Pentium III mobile processor. The C implementation of MUGI [33] runs at about 0.45 bit/cycle on the Pentium III processor. The encryption speed of Rabbit [3] is about 2.16 bit/cycle on Pentium III processor, but it is programmed in assembly language inline in C.

Table 1. The speed of the C implementation of HC-256 on Pentium 4

Operating System	Compiler	Optimization option	Speed (bit/cycle)
Windows XP (SP1)	Intel C++ Compiler 7.1	-O3	1.93
	Microsoft Visual C++ 6.0 Professional (SP5)	-Release	1.81
Red Hat Linux 9 (Linux 2.4.20-8)	Intel C++ Compiler 7.1	-O3	1.92
	gcc 3.2.2	-O3	1.83

Remarks. In HC-256, there is dependency between the feedback and output functions since the $P[i \bmod 1024]$ (or $Q[i \bmod 1024]$) being updated at the ith step is immediately used as linear masking. This dependency reduces the speed of HC-256 by about 3%. We do not remove this dependency from the design of HC-256 for security reason. Our analysis shows that each term being used as linear masking should not have been used in an S-box in the previous steps, otherwise the linear masking could be removed much easier. In our optimized implementation, we do not deal with this dependency because its effect on the encryption speed is very limited on the Pentium 4 processor.

Initialization Process. The key setup of HC-256 requires about 74,000 clock cycles (measured by repeating the setup process 2^{16} times on the Pentium 4 processor with Intel C++ compiler 7.1). This amount of computation is more than that required by most of the other stream ciphers (for example, the initialization process of Scream takes 27,500 clock cycles). The reason is that two large S-boxes are used in HC-256. To eliminate the threat of related key/IV attack, the tables should be updated with the key and IV thoroughly and this process requires a lot of computations. So it is undesirable to use HC-256 in the applications where key (or IV) is updated frequently.

5 Conclusion

In this paper, we proposed a software-efficient stream cipher HC-256. Our analysis shows that HC-256 is very secure. However, the extensive security analysis of any new cipher requires a lot of efforts from many researchers. We thus invite and encourage the readers to analyze the security of HC-256.

Finally we explicitly state that HC-256 is available royalty-free and HC-256 is not covered by any patent in the world.

References

1. S. Babbage, "A Space/Time Tradeoff in Exhaustive Search Attacks on Stream Ciphers", European Convention on Security and Detection, IEE Conference publication, No. 408, May 1995.
2. E. Biham, A. Shamir, "Differential Cryptanalysis of DES-like Cryptosystems", in *Advances in Cryptology – Crypto'90*, LNCS 537, pp. 2-21, Springer-Verlag, 1991.
3. M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius, "Rabbit: A New High-Performance Stream Cipher", in *Fast Software Encryption (FSE'03)*, LNCS 2887, pp. 307-329, Springer-Verlag, 2003.
4. V.V. Chepyzhov, T. Johansson, and B. Smeets. "A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers", in *Fast Software Encryption (FSE'00)*, LNCS 1978, pp. 181-195, Springer-Verlag, 2000.
5. D. Coppersmith, S. Halevi, and C. Jutla, "Scream: A Software-Efficient Stream Cipher", in *Fast Software Encryption (FSE'02)*, LNCS 2365, pp. 195-209, Springer-Verlag, 2002.
6. D. Coppersmith, S. Halevi, and C. Jutla, "Cryptanalysis of Stream Ciphers with Linear Masking", in *Advances in Cryptology – Crypto 2002*, LNCS 2442, pp. 515-532, Springer-Verlag, 2002.
7. J. Daeman and V. Rijmen, "AES Proposal: Rijndael", available on-line from NIST at http://csrc.nist.gov/encryption/aes/rijndael/
8. P. Ekdahl and T. Johansson, "A new version of the stream cipher SNOW", in *Selected Areas in Cryptology (SAC 2002)*, LNCS 2595, pp. 47–61, Springer-Verlag, 2002.
9. S. Fluhrer and D. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator", in *Fast Software Encryption (FSE'00)*, LNCS 1978, pp. 19-30, 2001.
10. S. Fluhrer, I. Mantin, and A. Shamir. "Weaknesses in the Key Scheduling Algorithm of RC4", in *Selected Areas in Cryptography (SAC 2001)*, LNCS 2259, pp. 1-24, Springer-Verlag, 2001.
11. J. D. Golić, "Towards Fast Correlation Attacks on Irregularly Clocked Shift Registers", in *Advances in Cryptography – Eurocrypt'95*, pages 248-262, Springer-Verlag, 1995.
12. J. D. Golić, "Linear Models for Keystream Generator". *IEEE Trans. on Computers*, 45(1):41-49, Jan 1996.
13. J. D. Golić, "Cryptanalysis of Alleged A5 Stream Cipher", in *Advances in Cryptology – Eurocrypt'97*, LNCS 1233, pp. 239 - 255, Springer-Verlag, 1997.
14. J. D. Golić, "Linear Statistical Weakness of Alleged RC4 Keystream Generator", in *Advancesin Cryptology – Eurocrypt'97*, pp. 226 - 238, Springer-Verlag, 1997.
15. T. Johansson and F. Jönsson. "Fast Correlation Attacks through Reconstruction of Linear Polynomials", in *Advances in Cryptology - CRYPTO 2000*, LNCS 1880, pp. 300-315, Springer-Verlag, 2000.
16. L. Knudsen, W. Meier, B. Preneel, V. Rijmen and S. Verdoolaege, "Analysis Methods for (Alleged) RC4", in *Advances in Cryptology – Asiacrypt'98*, LNCS 1514, pp. 327-341, Springer-Verlag, 1998.
17. I. Mantin and A. Shamir, "A Practical Attack on Broadcast RC4", in *Fast Software Encryption (FSE'01)*, LNCS 2355, pp. 152-164, Springer-Verlag, 2002.
18. M. Matsui, "Linear Cryptanalysis Method for DES Cipher", in *Advances in Cryptology – Eurocrypt'93*, LNCS 765, pp. 386-397, Springer-Verlag, 1994.
19. W. Meier and O. Staffelbach, "Fast Correlation Attacks on Certain Stream Ciphers". *Journal of Cryptography*, 1(3):159-176, 1989.

20. M. Mihaljević, M.P.C. Fossorier, and H. Imai, "A Low-Complexity and High-Performance Algorithm for Fast Correlation Attack", in *Fast Software Encryption (FSE'00)*, pp. 196-212, Springer-Verlag, 2000.
21. I. Mironov, "(Not So) Random Shuffles of RC4", in *Advances in Cryptology – Crypto 2002*, LNCS 2442, pp. 304-319, Springer-Verlag, 2002.
22. S. Mister, and S.E. Tavares, "Cryptanalysis of RC4-like Ciphers", in *Selected Areas in Cryptography (SAC'98)*, LNCS 1556, pp. 131-143, Springer-Verlag, 1999.
23. NESSIE, "NESSIE Project Announces Final Selection of Crypto Algorithms", available at https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/press_release_feb27.pdf
24. National Institute of Standards and Technology, "Secure Hash Standard (SHS)", available at http://csrc.nist.gov/cryptval/shs.html
25. R.L. Rivest, "The RC4 Encryption Algorithm". RSA Data Security, Inc., March 12, 1992.
26. P. Rogaway and D. Coppersmith, "A Software Optimized Encryption Algorithm". *Journal of Cryptography*, 11(4), pp. 273-287, 1998.
27. G.G. Rose and P. Hawkes, "Turing: a Fast Stream Cipher". *Fast Software Encryption (FSE'03)*, LNCS 2887, pp. 290-306, Springer-Verlag, 2003.
28. B. Schneier, "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)", in *Fast Software Encryption (FSE'93)*, LNCS 809, pp. 191-204, Springer-Verlag, 1994.
29. B. Schneier and D. Whiting, "Fast Software Encryption: Designing Encryption Algorithms for Optimal Software Speed on the Intel Pentium Processor", in *Fast Software Encryption (FSE'97)*, LNCS 1267, pp. 242-259, Springer-Verlag, 1997.
30. T. Seigenthaler. "Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications". *IEEE Transactions on Information Theory*, IT-30:776-780,1984.
31. T. Seigenthaler. "Decrypting a Class of Stream Ciphers Using Ciphertext Only". *IEEE Transactions on Computers*, C-34(1):81-85, Jan. 1985.
32. S. Vaudenay, "On the Weak Keys of Blowfish", in *Fast Software Encryption (FSE'96)*, LNCS 1039, pp. 27-32, Springer-Verlag, 1996.
33. D. Watanabe, S. Furuya, H. Yoshida, K. Takaragi, and B. Preneel, "A New Keystream Generator MUGI", in *Fast Software Encryption (FSE'02)*, LNCS 2365, pp. 179-194, Springer-Verlag, 2002.

A Test Vectors of HC-256

Let $K = K_0||K_1||\cdots||K_7$ and $IV = IV_0||IV_1||\cdots||IV_7$. The first 512 bits of keystream are given for different values of key and IV.

1. The key and IV are set as 0.

```
8589075b  0df3f6d8  2fc0c542  5179b6a6
3465f053  f2891f80  8b24744e  18480b72
ec2792cd  bf4dcfeb  7769bf8d  fa14aee4
7b4c50e8  eaf3a9c8  f506016c  81697e32
```

2. The key is set as 0, the IV is set as 0 except that $IV_0 = 1$.

```
bfa2e2af  e9ce174f  8b05c2fe  b18bb1d1
ee42c05f  01312b71  c61f50dd  502a080b
edfec706  633d9241  a6dac448  af8561ff
5e04135a  9448c434  2de7e9f3  37520bdf
```

3. The IV is set as 0, the key is set as 0 except that $K_0 = 0x55$.

```
fe4a401c  ed5fe24f  d19a8f95  6fc036ae
3c5aa688  23e2abc0  2f90b3ae  a8d30e42
59f03a6c  6e39eb44  8f7579fb  70137a5e
6d10b7d8  add0f7cd  723423da  f575dde6
```

B The optimized C implementation of HC-256 ("hc256.h")

```
#include <stdlib.h>

typedef unsigned long uint32;
typedef unsigned char uint8;

uint32 P[1024],Q[1024];
uint32 X[16],Y[16];
uint32 counter2048; // counter2048 = i mod 2048;

#ifndef _MSC_VER
#define rotr(x,n)   (((x)>>(n))|((x)<<(32-(n))))
#else
#define rotr(x,n)   _lrotr(x,n)
#endif

#define h1(x,y) {             \
     uint8 a,b,c,d;           \
     a = (uint8) (x);         \
     b = (uint8) ((x) >> 8);  \
     c = (uint8) ((x) >> 16); \
     d = (uint8) ((x) >> 24); \
     (y) = Q[a]+Q[256+b]+Q[512+c]+Q[768+d]; \
}

#define h2(x,y) {             \
     uint8 a,b,c,d;           \
     a = (uint8) (x);         \
     b = (uint8) ((x) >> 8);  \
     c = (uint8) ((x) >> 16); \
     d = (uint8) ((x) >> 24); \
     (y) = P[a]+P[256+b]+P[512+c]+P[768+d]; \
}

#define step_A(u,v,a,b,c,d,m){        \
     uint32 tem0,tem1,tem2,tem3;      \
     tem0 = rotr((v),23);             \
     tem1 = rotr((c),10);             \
     tem2 = ((v) ^ (c)) & 0x3ff;      \
     (u) += (b)+(tem0^tem1)+Q[tem2];  \
     (a) = (u);                       \
     h1((d),tem3);                    \
     (m) ^= tem3 ^ (u) ;              \
}

#define step_B(u,v,a,b,c,d,m){        \
     uint32 tem0,tem1,tem2,tem3;      \
     tem0 = rotr((v),23);             \
     tem1 = rotr((c),10);             \
     tem2 = ((v) ^ (c)) & 0x3ff;      \
     (u) += (b)+(tem0^tem1)+P[tem2];  \
     (a) = (u);                       \
     h2((d),tem3);                    \
     (m) ^= tem3 ^ (u) ;              \
}

void encrypt(uint32 data[])  //each time it encrypts 512-bit data
{
   uint32 cc,dd;
   cc = counter2048 & 0x3ff;
   dd = (cc+16)&0x3ff;

   if (counter2048 < 1024)
   {
      counter2048 = (counter2048 + 16) & 0x7ff;
      step_A(P[cc+0], P[cc+1], X[0], X[6], X[13],X[4], data[0]);
      step_A(P[cc+1], P[cc+2], X[1], X[7], X[14],X[5], data[1]);
      step_A(P[cc+2], P[cc+3], X[2], X[8], X[15],X[6], data[2]);
      step_A(P[cc+3], P[cc+4], X[3], X[9], X[0], X[7], data[3]);
      step_A(P[cc+4], P[cc+5], X[4], X[10],X[1], X[8], data[4]);
      step_A(P[cc+5], P[cc+6], X[5], X[11],X[2], X[9], data[5]);
      step_A(P[cc+6], P[cc+7], X[6], X[12],X[3], X[10],data[6]);
      step_A(P[cc+7], P[cc+8], X[7], X[13],X[4], X[11],data[7]);
      step_A(P[cc+8], P[cc+9], X[8], X[14],X[5], X[12],data[8]);
      step_A(P[cc+9], P[cc+10],X[9], X[15],X[6], X[13],data[9]);
      step_A(P[cc+10],P[cc+11],X[10],X[0], X[7], X[14],data[10]);
      step_A(P[cc+11],P[cc+12],X[11],X[1], X[8], X[15],data[11]);
      step_A(P[cc+12],P[cc+13],X[12],X[2], X[9], X[0], data[12]);
      step_A(P[cc+13],P[cc+14],X[13],X[3], X[10],X[1], data[13]);
      step_A(P[cc+14],P[cc+15],X[14],X[4], X[11],X[2], data[14]);
      step_A(P[cc+15],P[dd+0], X[15],X[5], X[12],X[3], data[15]);
   }
   else
   {
      counter2048 = (counter2048 + 16) & 0x7ff;
      step_B(Q[cc+0], Q[cc+1], Y[0], Y[6], Y[13],Y[4], data[0]);
      step_B(Q[cc+1], Q[cc+2], Y[1], Y[7], Y[14],Y[5], data[1]);
      step_B(Q[cc+2], Q[cc+3], Y[2], Y[8], Y[15],Y[6], data[2]);
      step_B(Q[cc+3], Q[cc+4], Y[3], Y[9], Y[0], Y[7], data[3]);
      step_B(Q[cc+4], Q[cc+5], Y[4], Y[10],Y[1], Y[8], data[4]);
      step_B(Q[cc+5], Q[cc+6], Y[5], Y[11],Y[2], Y[9], data[5]);
      step_B(Q[cc+6], Q[cc+7], Y[6], Y[12],Y[3], Y[10],data[6]);
      step_B(Q[cc+7], Q[cc+8], Y[7], Y[13],Y[4], Y[11],data[7]);
      step_B(Q[cc+8], Q[cc+9], Y[8], Y[14],Y[5], Y[12],data[8]);
      step_B(Q[cc+9], Q[cc+10],Y[9], Y[15],Y[6], Y[13],data[9]);
      step_B(Q[cc+10],Q[cc+11],Y[10],Y[0], Y[7], Y[14],data[10]);
      step_B(Q[cc+11],Q[cc+12],Y[11],Y[1], Y[8], Y[15],data[11]);
      step_B(Q[cc+12],Q[cc+13],Y[12],Y[2], Y[9], Y[0], data[12]);
      step_B(Q[cc+13],Q[cc+14],Y[13],Y[3], Y[10],Y[1], data[13]);
      step_B(Q[cc+14],Q[cc+15],Y[14],Y[4], Y[11],Y[2], data[14]);
      step_B(Q[cc+15],Q[dd+0], Y[15],Y[5], Y[12],Y[3], data[15]);
   }
}

//The following defines the initialization functions

#define f1(x)  (rotr((x),7) ^ rotr((x),18) ^ ((x) >> 3))
#define f2(x)  (rotr((x),17) ^ rotr((x),19) ^ ((x) >> 10))
#define f(a,b,c,d) (f2((a)) + (b) + f1((c)) + (d))

#define feedback_1(u,v,b,c) {      \
   uint32 tem0,tem1,tem2;          \
   tem0 = rotr((v),23); tem1 = rotr((c),10); \
   tem2 = ((v) ^ (c)) & 0x3ff;     \
   (u) += (b)+(tem0^tem1)+Q[tem2]; \
}

#define feedback_2(u,v,b,c) {      \
   uint32 tem0,tem1,tem2;          \
   tem0 = rotr((v),23); tem1 = rotr((c),10); \
   tem2 = ((v) ^ (c)) & 0x3ff;     \
   (u) += (b)+(tem0^tem1)+P[tem2]; \
}

void initialization(uint32 key[], uint32 iv[])
{
   uint32 i,j;

   //expand the key and iv into P and Q
   for (i = 0; i < 8; i++)     P[i] = key[i];
   for (i = 8; i < 16; i++)    P[i] = iv[i-8];

   for (i = 16; i < 528; i++)
       P[i] = f(P[i-2],P[i-7],P[i-15],P[i-16])+i;
   for (i = 0; i < 16; i++)
       P[i] = P[i+512];
   for (i = 16; i < 1024; i++)
       P[i] = f(P[i-2],P[i-7],P[i-15],P[i-16])+512+i;

   for (i = 0;  i < 16;  i++)
       Q[i] = P[1024-16+i];
   for (i = 16; i < 32;  i++)
       Q[i] = f(Q[i-2],Q[i-7],Q[i-15],Q[i-16])+1520+i;
   for (i = 0;  i < 16;  i++)
       Q[i] = Q[i+16];
   for (i = 16; i < 1024;i++)
       Q[i] = f(Q[i-2],Q[i-7],Q[i-15],Q[i-16])+1536+i;

   //run the cipher 4096 steps without generating output
   for (i = 0; i < 2; i++) {
       for (j = 0;  j < 10;   j++)
           feedback_1(P[j],P[j+1],P[(j-10)&0x3ff],P[(j-3)&0x3ff]);
       for (j = 10; j < 1023; j++)
           feedback_1(P[j],P[j+1],P[j-10],P[j-3]);
           feedback_1(P[1023],P[0],P[1013],P[1020]);
       for (j = 0;  j < 10;   j++)
           feedback_2(Q[j],Q[j+1],Q[(j-10)&0x3ff],Q[(j-3)&0x3ff]);
       for (j = 10; j < 1023; j++)
           feedback_2(Q[j],Q[j+1],Q[j-10],Q[j-3]);
           feedback_2(Q[1023],Q[0],Q[1013],Q[1020]);
   }

   //initialize counter2048, and tables X and Y
   counter2048 = 0;
   for (i = 0; i < 16; i++) X[i] = P[1008+i];
   for (i = 0; i < 16; i++) Y[i] = Q[1008+i];
}
```

C Test HC-256 ("test.c")

```
//This program prints the first 512-bit keystream
//then measure the average encryption speed

#include "hc256.h"
#include <stdio.h>
#include <time.h>

int main()
{
    uint32 key[8],iv[8];
    static uint32 DATA[16]; // the DATA is encrypted

    clock_t start, finish;
    double duration, speed;
    uint32 i;

    //initializes the key and IV
    for (i = 0; i < 8; i++) key[i]=0;
    for (i = 0; i < 8; i++) iv[i]=0;

    //key and iv setup
    initialization(key,iv);

    //generate and print the first 512-bit keystream
    for (i = 0; i < 16; i++) DATA[i]=0;
    encrypt(DATA);
    for (i = 0; i < 16; i++) printf(" %8x ", DATA[i]);

    //measure the encryption speed by encrypting
    //DATA repeatedly for 0x4000000 times
    start = clock();
    for (i = 0; i < 0x4000000; i++)  encrypt(DATA);
    finish = clock();

    duration = ((double)(finish - start))/ CLOCKS_PER_SEC;
    speed = ((double)i)*32*16/duration;

    printf("\n The encryption takes %4.4f seconds.\n\
            The encryption speed is %13.2f bit/second \n",\
            duration,speed);
    return (0);
}
```