Cryptanalysis of RC4-Based Hash Function

Mohammad Ali Orumiehchiha **Josef Pieprzyk** **Ron Steinfeld**

Center for Advanced Computing, Algorithms and Cryptography, Department of Computing,
Faculty of Science, Macquarie University, Sydney, NSW 2109, Australia
Email: {mohammad.orumiehchiha,josef.pieprzyk,ron.steinfeld}@mq.edu.au

Abstract

RC4-Based Hash Function is a new proposed hash function based on RC4 stream cipher for ultra low power devices. In this paper, we analyse the security of the function against collision attack. It is shown that the attacker can find collision and multi-collision messages with complexity only 6 compress function operations and negligible memory with time complexity 2^{13}. In addition,we show the hashing algorithm can be distinguishable from a truly random sequence with probability close to one.
Keywords: RC4-Based Hash Function, RC4 Stream Cipher, Cryptanalysis, Collision resistance.

1 Introduction

Cryptographic hash functions are functions that map an input of arbitrary length to a string of a fixed length. It means that the output of a hash function has a fixed length but the input stream can be a string of an arbitrary length (as short as a single bit or as long as several terabytes). Hash functions are indispensable for variety of security applications that include message authentication, integrity verification, and digital signatures. Recent developments in analysis of hash functions have demonstrated that most members of the MD family have many weaknesses that may compromise security of applications in which the hash functions are used. It turns out that for hash functions such as MD5, SHA-0 and SHA-1 (6; 7; 8), there are attacks that allow to find random collisions faster than expected. These advances in cryptanalysis of hashing functions is the main reason for the NIST call for the new SHA-3 cryptographic hash standard (4). SHA-3 is public and has generated a lot of interest from the cryptographic community.

There has been a constant flow of new design ideas and new analysis techniques. One such idea is the usage of stream ciphers to construct new hash functions. The RC4 stream cipher - designed by Rivest in 1987 (5)- seems to be an attractive option to build a fast and light-weight hash function(1; 2). It is a very simple and elegant cipher that can be implemented using relatively modest computing resources. More importantly, RC4 has been studied for many years and its efficiency makes it a good cryptographic tool for building hash functions that can be implemented as a light-weight algorithm. In 2006 Chang, Gupta, and Nandi (2) proposed a hash function that uses RC4 as the building block. The hash function was called RC4-Hash. The compression function in RC4-Hash applies the key scheduling algorithm (KSA) that is one of the main components of RC4. Because of a specific structure of RC4-Hash, the generic attacks (that are so effective against hash functions from the MD family) fail to work. However, in 2008 Idesteege and Preneel (3) have showed that RC4-Hash is not collision resistant.

Recently Yu, Zhang, and Haung (1) came up with an another hash function design that is based on RC4 as well. The function was called the RC4-based hash function and in the paper we are going to call it RC4-BHF. In addition to the KSA function, the RC4-BHF hash function uses also two other RC4 functions, namely KSA* and PRGA*. The aim of the designers was to avoid the attacks by Idesteege and Preneel. The KSA* function is similar to KSA but without the initialization part. The PRGA* is similar to the original pseudorandom generation algorithm (PRGA) of RC4 with a difference that PRGA* does not generate output but changes the internal state. Note that padding of messages in RC4-BHF is different from the one used in RC4-Hash. The brief description of RC4-BHF is given in the next Section. Full details about RC4-BHF can be found in (1). The authors of RC4-BHF argue that their hash function is collision resistant and very efficient. They claim that RC4-BHF is roughly 4.6 times faster than SHA-1 and 16 times faster than MD4 (1).

In this paper, we show that their claim about security of RC4-BHF is not true and we describe how to find collisions. We propose two attacks including collision attack and distinguishing attack. In the first one, by using periodic manner of internal states, we construct colliding message pairs with complexity 2^{13} compress function operations. And also we exploit this attack to make multicollisions. In the second attack, we show that output of RC4-BHF is distinguishable from random sequences.

The rest of the paper is structured as follows. Section 2 gives details of the RC4-BHF construction. Section 3 consists the main results of this work. In this section, after identifying weak points of the algorithm, we present a method to find colliding messages and also show how to construct a distinguisher for the hash function. Section 4 concludes the work.

2 Description of the RC4-BHF hash function

The hash function has been designed by Yu, Zhang and Hung in 2010 and the reader interested in its full description is referred to (1). The hash function uses the building blocks used in the RC4 stream cipher. These blocks, however, are modified by the authors. The blocks in question are:

- KSA (key scheduling algorithm of RC4) – this function takes as an input a 64-byte message $M = (M[0], \ldots, M[63])$ and outputs the internal state $\langle S, i, j \rangle$, where $S = (S[0], \ldots, S[255])$ is a 256-byte sequence and j is a 1-byte index. And also a 1-byte index called i. The function is described in Figure 1.

```
Input:    Message M
Output:   Internal State ⟨S, i, j⟩
    for i = 0  to  255
        S[i] = i;
    end for
    for i = 0  to  255
        j = (j + S[i] + M[i mod 64]) mod 256;
        swap(S[i], S[j]);
    end for
```

Figure 1: KSA Function

  Note that the KSA function is called at the very beginning of the RC4-BHF to initialize the internal state.

- KSA* – the function takes the pair: the message M, the internal state $\langle S, i, j \rangle$ as the input and provides an updated internal state. The full details are given in Figure 2.

```
Input:    Message M and Internal State ⟨S, i, j⟩
Output:   Updated Internal State ⟨S, i, j⟩
    for i = 0  to  255
        j = (j + S[i] + M[i mod 64]) mod 256;
        swap(S[i], S[j]);
    end for
```

Figure 2: KSA* Function

- PRGA* (pseudorandom generation algorithm) – the function takes the pair: an integer len, the internal state $\langle S, i, j \rangle$ as the input and generates an updated internal state on its input. The pseudocode of the function is given in Figure 3.

```
Input:    Integer len, Internal State ⟨S, i, j⟩
Output:   Updated Internal State ⟨S, i, j⟩
    for i = 0 to len
        i = i + 1 mod 256;
        j = (j + S[i]) mod 256;
        swap(S[i], S[j]);
    end for
```

Figure 3: PRGA* Function

The building blocks (functions) are used to create a sequence of compression functions according to the well-known Merkle-Damgård (MD) structure. Given a binary message M of an arbitrary length, the hashing algorithm proceeds through the following steps:

1. **padding** – binary representation of the padding length is appended to the message and then an appropriate number of bits (constant or random) is attached so the number of bits in the resulting message is a multiple of 512. Consequently, the message can be represented as a sequence of $M = (M_1, \ldots, M_n)$, where each M_i is a 512-bit long (or alternatively 64-byte) sequence,

2. **compression** – the message M_1 is used to initialize the internal state $\langle S, i, j \rangle$ as follows
$$\langle S, i, j \rangle \leftarrow KSA(M_1)$$
and then the function PRGA* modifies the state depending on the length len_1 of the message M_1 ($len_1 = M_1 \mod 2^5$)
$$\langle S, i, j \rangle \leftarrow PRGA^*(len_1, \langle S, i, j \rangle).$$
For k; $k = 2, \ldots, n$, the internal states are updated step by step
$$\langle S, i, j \rangle \leftarrow PRGA^*(len_k, KSA^*(M_k, \langle S, i, j \rangle))$$
where $len_k = M_k \bmod 2^5$. Figure 4 illustrates the compression process. Note that the number of rounds applied in PRGA* is controlled by the integer $len_i = (M_i \bmod 2^5)$.

3. **truncation** – the output of the compression step consists of 258 bytes (256 bytes of the state together with 2 index bytes). The final hash value includes the least significant bit of each state byte and the indices. This means that hash value is 272-bit long.

The internal state of RC4-BHF $\langle S, i, j \rangle$, where S indicates internal state of RC4-BHF and (i, j) are the indices used in KSA, KSA*, and PRGA* functions. The state can be divided to four parts S_0, S_1, S_2, S_3, where

$$\begin{aligned}
S_0 &= \{s_k \mid 0 \le k < 64\}, \\
S_1 &= \{s_k \mid 64 \le k < 128\}, \\
S_2 &= \{s_k \mid 128 \le k < 192\}, \\
S_3 &= \{s_k \mid 192 \le k < 256\},
\end{aligned}$$

where s_k is the k-th byte of the internal state.

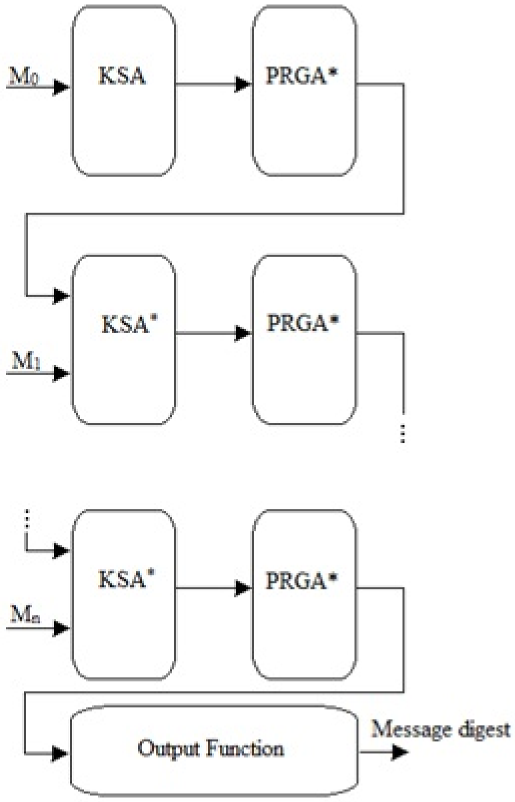

Figure 4: RC4-BHF Scheme

3 Cryptanalysis of RC4-BHF

In this section, we prove that RC4-BHF is not collision resistant. The proposed attack takes 2^{13} compression function operations and negligible memory. To apply collision attack on the algorithm, first we describe the weaknesses of hashing algorithm and then by exploiting these weaknesses, we propose collision attack and also present two distinguishers to tell apart the outputs generated by either RC4-BHF or a random number generator.

3.1 The weaknesses of RC4-BHF

Before describing our attack, we discuss properties of the RC4-BHF that underpin our attack.

1. The internal state is controlled by the input messages and can be manipulated by an appropriate choice of message bytes. In particular, we will show that we can select messages in a such way that the internal state repeats periodically.

2. The execution of the function PRGA* is controlled by the integer len. Note that if $len = M_k \bmod 2^5 = 0$, then the function PRGA* is not executed and can be skipped.

3. The index i is defined to be a byte or integer between 0 and 255. But after each execution of the function KSA*, the index $i = 255$. Similarly, after each execution of PRGA*, the index i can be an integer between 0 and 31. These properties are not used in collision attack but they may be exploited to enhance distinguishing attack on the scheme.

Now, we can describe our collision attack on the RC4-BHF.

3.2 Collision attack on RC4-BHF

The attack takes advantage of the periodicity of the function KSA* as formulated in the following theorem.

Theorem 1 *Given the function KSA* of the RC4-BHF. Let the input internal state be* $S = \langle S_0, S_1, S_2, S_3, 63\rangle$, *the output internal state be* $S' = \langle S'_0, S'_1, S'_2, S'_3, 63\rangle$ *and the message sequence be* $M = (m_0, \ldots, m_{63})$, *where* $m_i = -(s_i - 1) \bmod 256$; $0 \le i < 64$. *Then*

$$KSA^*(\langle S_0, S_1, S_2, S_3, 63\rangle) = \langle S'_0 = S_0, S'_1 = S_2, S'_2 = S_3, S'_3 = S_1, 63\rangle$$

Proof. It can be easily shown by applying KSA* on the internal state or by induction such as a generalisation of Theorem 2 from (3). Denote by $\langle S^{(i)}, j^{(i)}\rangle$ the internal state of RC4-BHF after the i-th step of the compression function KSA*. Note that

$$M[i \bmod 64] = m_{i \bmod 64} = -(s_{i \bmod 64} - 1) \bmod 256.$$

First, we prove by induction that for every $i < 256$, the following equations hold:

$$j^{(i)} = i + 63 \ mod\ 256,\ and$$

$$S^{(i)}[i+1\ mod\ 256] = s_{i+1\ mod\ 64},$$

$$S^{(i)}[i+2\ mod\ 256] = s_{i+2\ mod\ 64},$$

$$\ldots$$

$$S^{(i)}[i+64\ mod\ 256] = s_{i+63\ mod\ 64}.$$

It is clear that this holds before the first step, *i.e.*, for $i = -1$, since $j^{(-1)} = 1$, $S^{(-1)}[0] = S[0] = s_0$ till $S^{(-1)}[63] = S[63] = s_{63}$. Assume that the condition holds after step i ($i < 255$). Then, the update of the pointer j in the $(i+1)$-th step is

$$\begin{aligned} j^{(i+1)} &= j^{(i)} + S^{(i)}[i+1] + M[i\ mod\ 64]\ mod\ 256 \\ &= ((i+63) + s_{i+1})\ mod\ 256 \\ &+ (-(s_{i+1}\ mod\ 64 - 1)\ mod\ 256 \\ &= i + 64\ mod\ 256. \end{aligned}$$

Thus, $S^{(i+1)}$ is found by swapping the $(i+1)$-th and $(i+64)$-th element of $S^{(i)}$. Hence, $S^{(i+1)}[i + 64\ mod\ 256] = S^{(i)}[i+1\ mod\ 256] = s_{i+1\ mod\ 64}$. Of course, $S^{(i+1)}[i+64\ mod\ 256] = S^{(i)}[i+2\ mod\ 256] = s_{i\ mod\ 64}$. This implies that the condition also holds for step $i+1$. After 254 steps, all the elements of S have been rotated as follows:

$$S_0, S_1, S_2, S_3$$
$$S_0, S_2, S_3, S_1$$

■

Observe that if we apply the result of Theorem 1 in three consecutive calls to KSA* ($3 * 256$ steps), then the first state repeats. The situation is illustrated below:

$$
\begin{aligned}
& S_0, S_1, S_2, S_3 \\
\overset{KSA*}{\Longrightarrow}\ & S_0, S_2, S_3, S_1 \\
\overset{KSA*}{\Longrightarrow}\ & S_0, S_3, S_1, S_2 \\
\overset{KSA*}{\Longrightarrow}\ & S_0, S_1, S_2, S_3
\end{aligned}
$$

This means that the application of the function KSA* three times to the state causes that the same state is reached. Note that in addition to the above periodic behaviour of internal states, one can choose other specific messages to achieve the same periodic behaviour with longer periods. In (3), this behaviour of internal states of RC4 stream cipher is investigated and the reader is referred to it for details. Note that the construction of colliding message pairs is easy. To apply attack on RC4-BHF, we need to satisfy two conditions:

$$
\begin{cases}
\text{Condition 1: } j \text{ must be equal 63, and} \\
\text{Condition 2: the least 5 significant bits of } -(s_{63}- \\
\qquad\qquad 1) \ mod\ 256 \text{ must be zero.}
\end{cases} \tag{1}
$$

We expect that these requirements will be satisfied after testing $\approx 2^8 * 2^5$ messages.

3.3 Other Period Properties

As mentioned before, in addition to cycles of length 3, other cycles can be found for the KSA* function. In fact, the relation $M[i \ mod\ 64]$ in the functions KSA and KSA* can be used to apply other input messages to construct internal states with periods 7, 15, 31, 63, 127.

In similar way to Theorem 1, we can formulate appropriate conditions for internal state and the message M. The results are summarized in Table 1.

Using Table 1, we can find other colliding messages. Finding appropriate internal state requires the same effort (given by the time complexity column) for all cycles. Although we present two methods for the cycle equal to 3, these methods can be easily generalized for other cycles different from 3. In next section we show how we can construct colliding messages.

3.4 Finding Collisions

To construct colliding messages, two methods can be used.

- **Method 1.** In this method, after applying message M_0, we obtain the suitable internal state to satisfy the conditions (1). Then, by applying message M_1 three times and padding block, the hash value will be computed. Now, to generate other same hash value, we can repeat the message M_1 as in blocks of 3 and finally apply padding block and compute the final hashing digest. The following relations show how colliding messages can be constructed by method 1.

$$
\begin{aligned}
M^0 &= M_0 \ || \ Padding \\
M^1 &= M_0 \ ||M_P|| \ Padding \\
M^2 &= M_0 \ ||M_P||M_P|| \ Padding \\
&... \\
M^n &= M_0 \ ||M_P||...||M_P|| \ Padding
\end{aligned}
$$

where $M_P = M_1||M_1||M_1$ and $M^i, 0 \leq i \leq n$, are colliding messages.

Table 2: Example for Method 1 including M_0, M_1,M_2 and generated hash value.

	M_0 (64-byte)	M_1 (64-byte)	Hash Value (272-bis)
1	03DE074C6CB1A37 A201C0C8187BA03 6E87A3CCC89C35D F742B14E0D6136F D13986858771176 85ABE130121F415 555ED9D506B5CF4 11DA3B3CF066C04 11DC5548	FF520B5101BFC98 C743E178B6521E7 A30C2E95C43FA77 B25E2E8BB5A3DD0 D9CF299EDA05B11 8CA1A57676E4FB8 041FF520BCED417 8A94D7FCD399347 AA9F5B40	0350EA16 4598FCEC 553FF9C6 9535B628 1F87F266 01D26F48 EEF72985 64265C95 007B
2	004BB7F857C5080 B47B92603AED617 99F14278CAA881C CD997991397E173 9FE27885236CD8A E0DBEF561157C71 0616EA139D1DAF7 5A5C0D9FC3CB222 0D879471	52D5AFD2DA1ACFA B46F514E32F9784 086CB228253A649 BE57835E699275A 799CC8D4F2D7F3D B95F8A21DAA37DD 94E4AC128BB6290 9E0B566560487BA 6EC3EA00	E42DD715 2E9EAB3F 4851B2A0 AFD358F2 B98DF972 0CD285FD CA314801 842ECF4B 0009

We expect that after $2^8.2^5 = 2^{13}$ executions of the compression function for random messages, a suitable M_0 can be found. Table 2 presents two examples of messages M_0, messages M_1 and hash values obtained using Method 1.

Note that changing the length of input message M^i does not effect on padding content. So, we can construct arbitrary number of colliding messages with same hash value. This property can be used to compute multi-collisions.

- **Method 2.** The principle used is the same as in the previous method. We first find two messages M_0 and M_1 which satisfy the condition (1). After these two messages, the messages M_1, M_3 can be made using Theorem 1.Finally, collision pairs can be made by the following relations:

$$
\begin{aligned}
M^0 &= M_0||M_1||M_1||M_1||M_2||Padding \\
M^1 &= M_0||M_2||M_3||M_3||M_3||Padding \\
&...
\end{aligned}
$$

We expect that after $(2^8.2^5)^2$ executions of the compression function for random messages, a suitable M_0 and M_2 can be found. Table 3 shows

Table 1: Properties and conditions to apply collision attack on Algorithm for other cycles

	The Cycle Length	Condition 1	Condition 2	Time Complexity	Relations
1	7	$j = 31$	$-(s_{31} - 1)\ mod\ 64 = 0$	$2^8.2^5$	$m_i = -(s_i - 1)\ mod\ 256$, $m_i = m_{i+32},\ \ 0 \leq i < 32$
2	15	$j = 15$	$-(s_{15} - 1)\ mod\ 64 = 0$	$2^8.2^5$	$m_i = -(s_i - 1)\ mod\ 256$, $m_i = m_{i+16} = m_{i+32} = m_{i+64},\ \ 0 \leq i < 16$
3	31	$j = 7$	$-(s_7 - 1)\ mod\ 64 = 0$	$2^8.2^5$	$m_i = -(s_i - 1)\ mod\ 256$, $m_i = m_{i+8} = m_{i+16} = ... = m_{i+56},\ \ 0 \leq i < 8$
4	63	$j = 3$	$-(s_3 - 1)\ mod\ 64 = 0$	$2^8.2^5$	$m_i = -(s_i - 1)\ mod\ 256$,$m_i = m_{i+4} = m_{i+8} = ... = m_{i+60},\ \ 0 \leq i < 4$
5	127	$j = 1$	$-(s_1 - 1)\ mod\ 64 = 0$	$2^8.2^5$	$m_i = -(s_0 - 1)\ mod\ 256\quad i\ even$ $m_i = -(s_1 - 1)\ mod\ 256\quad i\ odd$
6	255	$j = 0$	$-(s_0 - 1)\ mod\ 64 = 0$	$2^8.2^5$	$m_i = -(s_0 - 1)\ mod\ 256$, $0 \leq i < 64$

two examples of messages M_0, M_1 M_2, M_3, and hash values obtained using Method 2.

Table 3: Example for Method 2 including M_0, M_1,M_2,M_3 and generated hash values.

	M_0 (64-byte)	M_1 (64-byte)	M_2 (64-byte)	M_3 (64-byte)	Hash Value (272-bis)
1	273A4F51 FAA4A7CF 3225E700 0A9ACDBC CABD7CAC 49991F5B B042CF90 80C2B7DC D756756F EFDBC42F E783580C C6CC0A8D BDB335AF AC2460F0 E8B61DA7 3C953096	BAFB22B0 6E1F20C5 0948DF65 A260D573 927B5606 25198784 0044523F 1435862F FC41E3CE BBDDB3D0 A5885890 D759AACB 89CD72D2 C1D3BDAB C7364505 F3EECF80	8FFD0B0A 03E6C6BF 7714E1C0 BF9B71DE 3AED7139 574F6556 57893E71 55E27E14 844B9CE8 B9DBAACC 297B3524 73E36D73 E1C5852D EA475DC6 FCB75F0F 797AA7C2	4459229F 9B50A1E3 F8A3A772 D464CA05 4F5DE628 84295ADC B3260921 0CC0E1A4 DDBBC8AE 71E00A12 43B77EAB 017B1F48 0B4AE795 8A1B4EB8 D2F902FD DAB01900	BEDE F059 71AC F6A3 AF04 5311 0417 28D5 D77E D338 5D58 4085 46A3 040B 5757 67FE 0029
2	7B45A927 E089C366 BB75CB2E 06E9AD05 3F3A007F BF33F060 48597B01 DD73E1F5 D64A55EB 33AEF9D6 31B9094C 1B58562C 6306F784 F1DB3BB2 BBC6E2C9 96178C36	5A154DFF 7B6D869E 3DC2DF25 3F894F68 D2B2F776 1C4674CA 6A8B5B94 4EF6BFAB B5792BC5 D89B7CA9 26D0118C 83698D6B A0BB9D90 61014CB6 8477F8A3 1D6536C0	5CFA81B5 EE3730B8 FB0B01A3 5FB4C45B 78E9ECD3 7CD38830 1059752B 16A0D2B7 C6D2B5E4 001F1C04 E002270C 94C6843D 6A482A03 2DFE4A1D B23882FE AEA65573	587C8C02 5B0B462B B83B3C40 FFEDF472 B6CCD8CF 6299285A 8FCA0768 E2EB787D 36EA2A6C 2E94B301 9103B169 7BC3D057 00313FD4 96C7521F FDC19BC8 59649580	11D3 C922 63F9 EFB1 65B6 370A A78D 6690 79B2 0706 2FE4 1228 2691 9A04 FBDF ED97 0019

3.5 Randomness properties of hash digest

As mentioned in Section 2, the hash value is generated by concatenating the least significant bits of each byte of the final internal state S and two bytes indices i and j. Note that the first 256 bits of the hash value is the least significant bit of the numbers 0 till 255 which are swapped based on three functions KSA, KSA*,and PRGA*. Although the positions of the integers are changed but their values are not modified and it means that the hamming weight of the first 256 bits of hash value for every input message with arbitrary length will be exactly 128.

In addition, index i in the last round just depends to the last input message M_n as $i = M_n\ mod\ 2^5$ and so it will be an integer between 0 and 31. The designers dedicated one byte for index i in the hash value. So first we can see that the three most significant bits for all input messages will be zero and second attacker can change the other five bits of 259-th -263-th bits by changing five least significant bits of the last input message M_n with probability one. Of course, if we consider the effect of padding block in the last round, then the index i will be fixed while padding block does not change. These two weaknesses lead attacker to a strong distinguisher with distinguishing advantage close to 1.

4 Conclusion

We presented collision attack on RC4-BHF. The attack requires negligible memory and time complexity 2^{13} compress function (KSA*) operations. The practicality of the attack has been demonstrated with some colliding messages for RC4-BHF. We also showed the hashing algorithm can be distinguishable from a truly random sequence with probability close to one.

References

[1] Yu Q., Zang C.N., Hung X., "An RC4-Based Hash Function for Ultra-Low Power Devices", 2nd International Conference on Computer Engineering and Technology (ICCET), pp. 323-328, IEEE Publication, 2010.

[2] Chang, D., Gupta, K.C., Nandi, M.: "RC4-Hash: A New Hash Function Based on RC4". In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 80-94. Springer, 2006.

[3] S. Indesteege, and B. Preneel, "Collisions for RC4-Hash," In Information Security - 11th International Conference, ISC 2008, Lecture Notes in Computer Science 5222, C. Lei, V. Rijmen, and T. Wu (eds.), Springer-Verlag, pp. 355-366, 2008.

[4] NIST: Cryptographic hash algorithm competition http://www.nist.gov/hash-competition.

[5] Schneier, B.:" Applied Cryptograph"y, 2nd edn. John Wiley and Sons, Chichester, 1996.

[6] Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19-35. Springer, 2005.

[7] Wang, X., Yu, H., Yin, Y.L.: Efficient Collision Search Attacks on SHA-0. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 1-16. Springer, 2005.

[8] Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17-36, Springer, 2005.