Collisions for RC4-Hash[⋆,⋆⋆]

Sebastiaan Indesteege[1,2,⋆⋆⋆] and Bart Preneel[1,2]

[1] Department of Electrical Engineering ESAT/SCD-COSIC, Katholieke Universiteit Leuven. Kasteelpark Arenberg 10, B-3001 Heverlee, Belgium.
{sebastiaan.indesteege,bart.preneel}@esat.kuleuven.be
[2] Interdisciplinary Institute for BroadBand Technology (IBBT), Belgium.

Abstract. RC4-Hash is a variable digest length cryptographic hash function based on the design of the RC4 stream cipher. In this paper, we show that RC4-Hash is not collision resistant. Collisions for any digest length can be found with an expected effort of less than 2^9 compression function evaluations. This is extended to multicollisions for RC4-Hash. Finding a set of 2^k colliding messages has an expected cost of $2^7 + k \cdot 2^8$ compression function evaluations.

Key words: RC4-Hash, hash functions, collisions, multicollisions.

1 Introduction

Cryptographic hash functions have been receiving much attention from the cryptologic community recently, as several of the widely used hash functions like MD5, SHA-0 and SHA-1, have been broken, or at least shown to be weaker than expected [3,9,10,11]. This is a motivation for the design of new hash functions, based on different design principles. One such proposal is RC4-Hash, which was introduced by Chang, Gupta and Nandi [1] in 2006. The design is inspired by the RC4 stream cipher. The latter was designed by Ron Rivest in 1987, but remained a trade secret until it leaked out in 1994 [8]. The motivation for basing a hash function design on RC4, which is well studied, is to be able to use existing results on RC4 in the security analysis of RC4-Hash [1]. Concerning the performance of RC4-Hash, the designers claim that SHA-1 is roughly 1.5 times faster than RC4-Hash [1].

We focus on the collision resistance of RC4-Hash. Informally, collision resistance means that it should be hard to find two distinct messages

[⋆] The final publication is available at www.springerlink.com.

[⋆⋆] This work was supported in part by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy), and in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

[⋆⋆⋆] F.W.O. Research Assistant, Fund for Scientific Research — Flanders (Belgium).

$m \neq m'$ that hash to the same value, *i.e.*, $h(m) = h(m')$. We show that RC4-Hash is not collision resistant, and give a method to find colliding message pairs with an expected time complexity of less than 2^9 compression function evaluations. We also extend this to multicollisions.

This paper is organised as follows. In Sect. 2, a short description of the RC4-Hash family of cryptographic hash functions is given. Section 3 introduces two distinct methods to construct fixed points of the internal state of RC4-Hash. This is then used in Sect. 4 to construct colliding message pairs for RC4-Hash. In Sect. 5, extensions of the attack, as well as ways to mitigate it, are discussed. Section 6 concludes.

2 Description of RC4-Hash

RC4-Hash follows the "wide pipe" hash function design principle proposed by Lucks [7], which implies that the intermediate state size is (much) larger than the digest size. More specifically, RC4-Hash consists of a compression function $\mathcal{C} : \{0,1\}^w \times \{0,1\}^m \mapsto \{0,1\}^w$, and an output transformation $g_n : \{0,1\}^w \mapsto \{0,1\}^n$. The intermediate state size w is (much) larger than the digest length n. The compression function $\mathcal{C}$ is applied iteratively for every (padded) message block of length m, starting from an initial value. Then, the output transformation g compresses the large internal state down to the required digest length n.

In RC4-Hash, the intermediate state consists of an array S of 256 bytes and a pointer into this array, denoted by j. The array S always represents a permutation of the numbers 0 to 255. The size of the internal state is thus $\log_2(2^8!) + 8 \approx 1692$ bits. The digest length is variable from 16 bytes to 64 bytes, which is much shorter than the internal state size. The length of the message blocks is fixed to 64 bytes.

Padding Rule. A message M is padded in the following way. The 8-bit binary representation of the digest length n (in bytes), $\text{bin}_8(n)$, is prepended to the message. A single "1" bit, v "0" bits and the 64-bit binary representation of the original message length (in bits), $\text{bin}_{64}(|M|)$, are appended to the message. The number v is the least non-negative integer such that $|M| + 73 + v \equiv 0 \bmod 512$. This ensures that the padded message length is an integer multiple of 512 bits, the message block length. Hence, the padded message can be split into t blocks of 512 bits each, denoted by M_1 through M_t.

$$\text{pad}(M) = \text{bin}_8(n) \,||\, M \,||\, 1 \,||\, 0^v \,||\, \text{bin}_{64}(|M|) = M_1 || M_2 || \cdots || M_t \ . \quad (1)$$

```
Input: Internal state $\langle S, j \rangle$, 64-byte message block $X$.
Output: The updated internal state $\langle S, j \rangle$.
1: for $i = 0$ to 255 do
2:     $j \leftarrow j + S[i] + X[r(i)]$
3:     swap$(S[i], S[j])$
4: end for
5: return $\langle S, j \rangle$
```

Fig. 1. The compression function of RC4-Hash, $\mathcal{C}\big(\langle S, j \rangle, X\big)$. All arithmetic is done modulo 256.

Compression Function. The compression function of RC4-Hash, which is denoted by $\mathcal{C}\big(\langle S, j \rangle, X\big)$, is described in Fig. 1. It updates the internal state $\langle S, j \rangle$ in 256 steps. In every step, the pointer j is updated using one byte of the message block X. Then, two elements of the array S are swapped. Each of the 64 bytes of the message block is used in four steps. The order in which they are used is given by the message reordering $r(\cdot)$, see Table 5. This compression function is applied iteratively for every message block M_1 through M_t, starting from the initial state $\big\langle S^{\mathrm{IV}}, 0 \big\rangle$. The initial value permutation S^{IV} is given in Table 6.

Output Transformation. After every block of the padded message has been processed, an output transformation $g_n\big(\langle S, j \rangle\big)$ is applied. This transformation generates the message digest of the required length n from the internal state. First, the permutation S is composed with the initial value permutation S^{IV}. The resulting permutation is saved as T_1. Then, two blank iterations of the compression function $\mathcal{C}$, *i.e.*, using a zero message block, are applied, resulting in T_2. Finally, S is replaced by a composition of the two saved permutations, $T_1 \circ T_2 \circ T_1$, and the message digest is generated using an algorithm similar to RC4's pseudo-random byte generation.

Figure 2 shows the definition of the entire output transformation. In the original description of RC4-Hash [1], the output transformation was further partitioned into the algorithms OWT ("one way transformation") and HBG ("hash byte generation"). These correspond to lines 2–9 and 10–15 of the algorithm in Fig. 2, respectively.

3 Fixed Points of the Compression Function $\mathcal{C}$

In this section, we describe how to construct two distinct types of fixed points for a certain number of iterations of the RC4-Hash compression

```
Input: Internal state $\langle S, j \rangle$ after processing the entire padded message.
Output: The message digest $H$.
 1: $S \leftarrow S^{\mathrm{IV}} \circ S$
 2: // OWT (one way transformation)
 3: $T_1 \leftarrow S$
 4: for $i = 0$ to 511 do
 5:        $j \leftarrow j + S[i]$
 6:        swap$(S[i], S[j])$
 7: end for
 8: $T_2 \leftarrow S$
 9: $S \leftarrow T_1 \circ T_2 \circ T_1$
10: // HBG (hash byte generation)
11: for $i = 0$ to $n$ do
12:        $j \leftarrow j + S[i]$
13:        swap$(S[i], S[j])$
14:        $H[i] \leftarrow S[S[i] + S[j]]$
15: end for
16: return $H$
```

Fig. 2. The output transformation of RC4-Hash, $g_n\big(\langle S, j\rangle\big)$. All arithmetic is done modulo 256.

function $\mathcal{C}$. Each of these constructions is based on one of two types of "partial state rotations", which are introduced in two lemmata, Lemma 1 and Lemma 3.

3.1 Fixed Points of Type I

Lemma 1 (Partial state rotations of type I). *Consider an internal state $\langle S, 0\rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. Denote by $\langle S', j'\rangle$ the internal state reached after applying the compression function $\mathcal{C}$ using the message block $X = \{x, x, \ldots, x\}$ with $x = 1 - s_0 \bmod 256$:*

$$\big\langle S', j' \big\rangle = \mathcal{C}\big(\langle S, j\rangle, X\big) \ . \tag{2}$$

Now, it holds that

$$j' = 0 \qquad \textit{and} \qquad S'[i] = \begin{cases} s_0 & i = 0 \\ s_{i+1} & 1 \le i < 255 \\ s_1 & i = 255 \end{cases} . \tag{3}$$

Proof. Denote by $\big\langle S^{(i)}, j^{(i)} \big\rangle$ the internal state of RC4-Hash after the i-th step of the compression function $\mathcal{C}$. First, we prove by induction that for

Table 1. Partial state rotations of type I.

step i	$j^{(i)}$	$S^{(i)}$								
	0	s_0	s_1	s_2	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
0	1	$\boldsymbol{s_1}$	$\boldsymbol{s_0}$	s_2	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
1	2	s_1	$\boldsymbol{s_2}$	$\boldsymbol{s_0}$	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
2	3	s_1	s_2	$\boldsymbol{s_3}$	$\boldsymbol{s_0}$	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
3	4	s_1	s_2	s_3	$\boldsymbol{s_4}$	$\boldsymbol{s_0}$	$\cdots$	s_{253}	s_{254}	s_{255}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
254	255	s_1	s_2	s_3	s_4	s_5	$\cdots$	s_{254}	$\boldsymbol{s_{255}}$	$\boldsymbol{s_0}$
255	0	$\boldsymbol{s_0}$	s_2	s_3	s_4	s_5	$\cdots$	s_{254}	s_{255}	$\boldsymbol{s_1}$

every $i < 256$ it holds that

$$\begin{cases} j^{(i)} = i + 1 \bmod 256 \ , & \text{and} \\ S^{(i)}[i + 1 \bmod 256] = s_0 \ . \end{cases} \tag{4}$$

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

$$\begin{aligned} j^{(i+1)} &= j^{(i)} + S^{(i)}[i + 1] + X[r(i + 1)] \bmod 256 \\ &= (i + 1) + s_0 + (1 - s_0) \bmod 256 \\ &= i + 2 \bmod 256 \ . \end{aligned} \tag{5}$$

Thus, $S^{(i+1)}$ is found by swapping the $(i + 1)$-th and $(i + 2)$-th element of $S^{(i)}$. Hence, $S^{(i+1)}[i + 2 \bmod 256] = S^{(i)}[i + 1 \bmod 256] = s_0$, *i.e.*, the condition also holds after step $i + 1$.

After 255 steps, all the elements of S have been circularly shifted over one position, *i.e.*, $S^{(254)} = \{s_1, s_2, \ldots, s_{255}, s_0\}$. In the final step, the first and the last element of $S^{(254)}$ are swapped since $j^{(255)} = 0$, resulting in

$$S^{(255)} = S' = \{s_0, s_2, s_3, \ldots, s_{254}, s_{255}, s_1\} \ . \tag{6}$$

From this, the lemma follows. $\square$

Table 1 gives a detailed illustration of Lemma 1. The first column of this table gives the step number i, the second column gives the new value of the pointer j, computed in this step. The last column contains the array S after the step, where the elements that were just swapped are encircled.

Based on this first type of partial state rotations, it is straightforward to construct fixed points for 255 iterations of the compression function $\mathcal{C}$ as is shown in the next theorem.

Theorem 1 (Fixed points of type I). *Consider an internal state $\langle S, 0\rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. After 255 iterations of the compression function $\mathcal{C}$, each using the same message block $X = \{x, x, \ldots, x\}$ with $x = 1 - s_0 \bmod 256$, the same state is reached:*

$$\langle S, 0\rangle = \mathcal{C}^{255}\big(\langle S, 0\rangle, X\big)\ . \tag{7}$$

Proof. The repeated application of Lemma 1 proves the theorem. □

Note that the only requirement for the construction of a fixed point of type I is that the pointer j has to be zero in the starting state. There are no conditions on the contents of the array S. Also, when given a suitable starting state, constructing a fixed point requires only a negligible amount of work, *i.e.*, one subtraction modulo 256 to compute the message byte $x = 1 - s_0 \bmod 256$.

3.2 Fixed Points of Type II

The message reordering $r(\cdot)$ has an interesting property which allows for another type of partial state rotations.

Lemma 2. *The message reordering $r(\cdot)$ does not reorder message bytes with an even index to odd-numbered positions, or vice versa. In other words,*

$$\forall i, 0 \le i < 256 : r(i) \equiv i \pmod 2\ . \tag{8}$$

Proof. The lemma follows in a straightforward way from the definition of $r(\cdot)$ in Table 5. □

Lemma 3 (Partial state rotations of type II). *Consider an internal state $\langle S, 1\rangle$ of RC4-Hash with $S = \{s_0, s_1, \ldots, s_{255}\}$. Denote by $\langle S', j'\rangle$ the internal state reached after applying the compression function $\mathcal{C}$ using the message block $X = \{x_0, x_1, x_0, x_1, \ldots, x_0, x_1\}$ with $x_0 = 1 - s_0 \bmod 256$ and $x_1 = 1 - s_1 \bmod 256$:*

$$\langle S', j'\rangle = \mathcal{C}\big(\langle S, j\rangle, X\big)\ . \tag{9}$$

Now, it holds that

$$j' = 1 \qquad \textit{and} \qquad S'[i] = \begin{cases} s_i & 0 \le i < 2 \\ s_{i+2} & 2 \le i < 254 \\ s_{i-252} & 254 \le i < 256 \end{cases}\ . \tag{10}$$

Proof. Denote by $\langle S^{(i)}, j^{(i)} \rangle$ the internal state of RC4-Hash after the i-th step of the compression function $\mathcal{C}$. Note that, because of Lemma 2 and the definition of X, $X[r(i)] = x_{i \bmod 2} = 1 - s_{i \bmod 2}$. First, we prove by induction that for every $i < 256$ it holds that

$$\begin{cases} j^{(i)} = i + 2 \bmod 256 \ , & \text{and} \\ S^{(i)}[i + 1 \bmod 256] = s_{i+1 \bmod 2} \ , & \text{and} \\ S^{(i)}[i + 2 \bmod 256] = s_{i \bmod 2} \ . \end{cases} \tag{11}$$

It is clear that this holds before the first step, *i.e.*, for $i = -1$, since $j^{(-1)} = 1$, $S^{(-1)}[0] = S[0] = s_0$ and $S^{(-1)}[1] = S[1] = s_1$. Assume that the condition holds after step i ($i < 255$). Then, the update of the pointer j in the $(i + 1)$-th step is

$$\begin{aligned} j^{(i+1)} &= j^{(i)} + S^{(i)}[i + 1] + X[r(i + 1)] \bmod 256 \\ &= (i + 2) + s_{i+1 \bmod 2} + (1 - s_{i+1 \bmod 2}) \bmod 256 \\ &= i + 3 \bmod 256 \ . \end{aligned} \tag{12}$$

Thus, $S^{(i+1)}$ is found by swapping the $(i+1)$-th and $(i+3)$-th element of $S^{(i)}$. Hence, $S^{(i+1)}[i + 3 \bmod 256] = S^{(i)}[i + 1 \bmod 256] = s_{i+1 \bmod 2}$. Of course, $S^{(i+1)}[i+2 \bmod 256] = S^{(i)}[i+2 \bmod 256] = s_{i \bmod 2}$. This implies that the condition also holds for step $i + 1$.

After 254 steps, all the elements of S have been circularly shifted over two position, *i.e.*, $S^{(253)} = \{s_2, s_3, s_4, \ldots, s_{255}, s_0, s_1\}$. Since $j^{(254)} = 0$ and $j^{(255)} = 1$, the swaps made in the last two steps result in the following state

$$S^{(255)} = S' = \{s_0, s_1, s_4, \ldots, s_{255}, s_2, s_3\} \ . \tag{13}$$

From this, the lemma follows. □

Table 2 gives a detailed illustration of Lemma 3. The notations are the same as in Table 1. Based on this type of partial state rotations, fixed points for 127 iterations of the compression function $\mathcal{C}$ can be constructed, as is shown in the next theorem.

Theorem 2 (Fixed points of type II). *Consider an internal state* $\langle S, 1 \rangle$ *of RC4-Hash with* $S = \{s_0, s_1, \ldots, s_{255}\}$. *After 127 iterations of the compression function* $\mathcal{C}$, *each using the same message block* $X = \{x_0, x_1, x_0, x_1, \ldots, x_0, x_1\}$ *with* $x_0 = 1 - s_0 \bmod 256$ *and* $x_1 = 1 - s_1 \bmod 256$, *the same state is reached:*

$$\langle S, 1 \rangle = \mathcal{C}^{127}\big(\langle S, 1 \rangle , X \big) \ . \tag{14}$$

Table 2. Partial state rotations of type II.

step i	$j^{(i)}$	$S^{(i)}$									
	1	s_0	s_1	s_2	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
0	2	$\boldsymbol{s_2}$	s_1	$\boldsymbol{s_0}$	s_3	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
1	3	s_2	$\boldsymbol{s_3}$	s_0	$\boldsymbol{s_1}$	s_4	$\cdots$	s_{253}	s_{254}	s_{255}
2	4	s_2	s_3	$\boldsymbol{s_4}$	s_1	$\boldsymbol{s_0}$	$\cdots$	s_{253}	s_{254}	s_{255}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$
253	255	s_2	s_3	s_4	s_5	s_6	$\cdots$	$\boldsymbol{s_{255}}$	s_0	$\boldsymbol{s_1}$
254	0	$\boldsymbol{s_0}$	s_3	s_4	s_5	s_6	$\cdots$	s_{255}	$\boldsymbol{s_2}$	s_1
255	1	s_0	$\boldsymbol{s_1}$	s_4	s_5	s_6	$\cdots$	s_{255}	s_2	$\boldsymbol{s_3}$

Proof. The repeated application of Lemma 3 proves the theorem. □

Note that, as for fixed points of type I, the only requirement for the construction of a fixed point of type II is that the j pointer has a certain value in the starting state. There are no conditions on the contents of the array S. Constructing a fixed point of type II, when given a suitable starting state, also requires only a negligible amount of work, *i.e.*, two subtractions modulo 256 to compute the message bytes $x_0 = 1 - s_0 \bmod 256$ and $x_1 = 1 - s_1 \bmod 256$.

One could try to further generalise this to longer cyclic patterns. However, the message byte reordering $r(\cdot)$ prevents this as there is no $p > 2$ for which it holds that

$$\forall i, 0 \leq i < 256 : r(i) \equiv i \pmod{p} \ . \tag{15}$$

3.3 Relation to Finney States

A Finney state [4] is an RC4-state where $j = i + 1$ and $S[i] = 1$. From the definition of the RC4 stream cipher, see Fig. 5, it follows that if the current state is a Finney state, the next state must also be a Finney state. Similarly, a Finney state can only arise from a Finney state. In a Finney state, the element "1" is simply moved to the next position in the array S and j is incremented. The initialisation of the RC4 pseudo-random byte generator, see Fig. 5, ensures that the initial state is not a Finney state. Hence, Finney states can never occur in RC4.

In RC4-Hash, however, we can achieve a similar pattern. This is exactly what is done in the case of partial state rotations of type I. The extra freedom coming from the message input is exploited to ensure that the element $S[i]$ is always moved to the next position, such that it is again

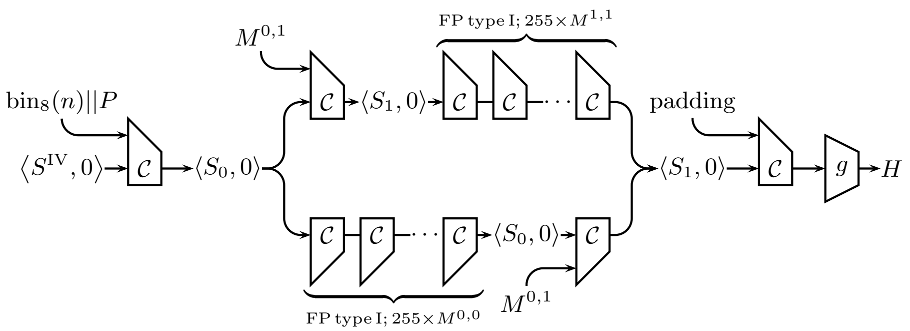

Fig. 3. A collision pair for RC4-Hash using fixed points of type I.

used to update j in the next iteration. Partial state rotations of type II are a generalisation of this, using two elements in an alternating way.

4 Collisions for RC4-Hash

This section describes how to use fixed points for a number of iterations of the compression function $\mathcal{C}$ to construct colliding message pairs for RC4-Hash. In order to be able to construct fixed points, the value of the pointer j in the internal state of RC4-Hash has to be equal to zero (for fixed points of type I) or one (for fixed points of type II), as described in Sect. 3. Although the initial value of j is zero, we cannot make use of the first block because we do not have control over its first byte, which contains the digest length.

Consider fixed points of type I, *i.e.*, we want $j = 0$. Since j can only take 2^8 possible values, we can simply search for a prefix block P which leads to a suitable internal state:

$$\langle S_0, 0\rangle = \mathcal{C}\big(\langle S^{\mathrm{IV}}, 0\rangle, \mathrm{bin}_8(n)||P\big) \ . \tag{16}$$

We expect to find a suitable prefix block after about 2^8 random trials. At this point, we can easily construct a fixed point for this state $\langle S_0, 0\rangle$ by applying Theorem 1. Denote by $M^{0,0}$ the message block that is used 255 times in this fixed point.

Then, we search for an additional message block $M^{0,1}$ which transforms the state $\langle S_0, 0\rangle$ into $\langle S_1, 0\rangle$:

$$\langle S_1, 0\rangle = \mathcal{C}\big(\langle S_0, 0\rangle, M^{0,1}\big) \ . \tag{17}$$

Again, the only condition on $M^{0,1}$ is that the value of the j pointer is not changed by the compression function $\mathcal{C}$. The expected number of random

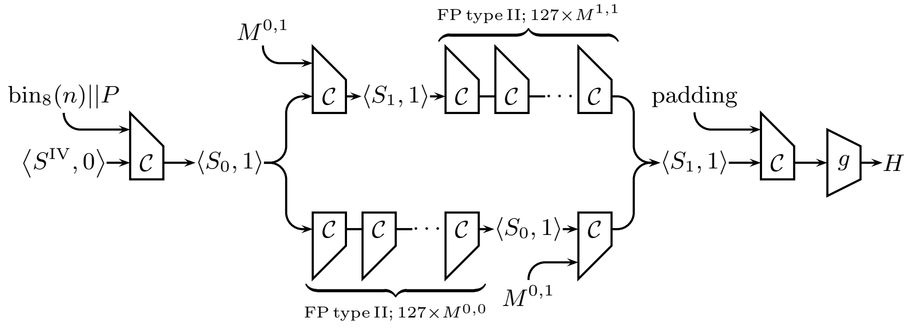

Fig. 4. A collision pair for RC4-Hash using fixed points of type II.

trials required to find a suitable message block is again about 2^8. For the state $\langle S_1, 0\rangle$, it is also possible to construct a fixed point of type I, using Theorem 1. Denote the message block used in this fixed point by $M^{1,1}$. Now, consider the following two messages:

$$\begin{aligned} M &= P||M^{0,1}||\overbrace{M^{1,1}||\cdots||M^{1,1}}^{255}\ , \\ M^\star &= P||\underbrace{M^{0,0}||\cdots||M^{0,0}}_{255}||M^{0,1}\ . \end{aligned} \tag{18}$$

As shown in Fig. 3, these messages form a collision. Indeed, after processing the 257-th block, the internal state of RC4-Hash is $\langle S_1, 0\rangle$ for both messages, *i.e.*, an internal state collision is reached. The extra padding block containing the message length and the output transformation maintain the collision. The expected total time complexity is only 2^9 evaluations of the compression function $\mathcal{C}$. Note that verifying the collision requires about the same effort, since hashing M and $M^\star$ requires two times 258 calls to the compression function $\mathcal{C}$.

Using fixed points of type II, collisions can be found in a completely similar way, as Fig. 4 illustrates. The only differences are that we now require $j = 1$, and that the fixed points only contain 127 iterations of the compression function $\mathcal{C}$. The expected time complexity is also 2^9. If we do not fix in advance which type of fixed points to use, but let this depend on which kind of prefix block is found first, the expected time complexity can be lowered slightly to $2^7 + 2^8$ compression function evaluations.

There is no need to restrict the prefix block P or the message block $M^{0,1}$ to be only a single block. Using multiple blocks does not (significantly) increase the expected time complexity for finding a collision pair,

Table 3. Example collision pair for RC4-Hash$_{64}$, using fixed points of type I.

	M	$M^\star$
block 1 (63 bytes)	`s.␣IndestEEGE␣AnD␣B.␣pReNeEl␣-␣` `cosIc␣-␣cOlLisIoNS␣FoR␣rC4-Hash.`	
block 2 (64 bytes)	`thiS␣MEssAgE␣Is␣pArT␣oF␣a␣colLis` `ion␣EXaMpLe␣for␣RC4-HASH.␣COSIC.`	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
blocks 3–256 (254 $\times$ 64 bytes)	`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` ⋮ `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`	`AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA` ⋮ `AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`
block 257 (64 bytes)	`aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa` `aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`	`thiS␣MEssAgE␣Is␣pArT␣oF␣a␣colLis` `ion␣EXaMpLe␣for␣RC4-HASH.␣COSIC.`
RC4-Hash$_{64}(M)$ = RC4-Hash$_{64}(M^\star)$	`0093b4baefdc64f93d7081978808c49d1286523696e6d4a35ab64f1e42695aff` `79ce81eae91cb47673c4989238fab010f47466906fa65bed88753802c71ae82b`$_x$	

if only the last block of P, resp. $M^{0,1}$, is varied in order to obtain the desired value for the pointer j. Of course, a colliding message pair can always be extended with an equal suffix.

Tables 3 and 4 give examples of colliding message pairs for RC4-Hash$_{64}$, constructed using fixed points of type I and type II, respectively. Additional constraints were imposed to arrive at meaningful messages.

5 Discussion

Kelsey-Schneier Second Preimages. Since fixed points of the compression function of RC4-Hash can be constructed very easily, one may consider to use them to mount a Kelsey-Schneier second preimage attack [6]. This involves building expandable messages, *i.e.*, messages of varying length, which all collide on the intermediate hash result immediately after processing the message. The main problem which makes the Kelsey-Schneier second preimage attack fail for RC4-Hash, is the very large internal state of RC4-Hash. Because of this, the Kelsey-Schneier attack is much slower than exhaustive search in this case.

Multicollisions. A multicollision is a (large) set of messages that all hash to the same value. Multicollisions and their applications were described by Joux [5], although Coppersmith already used them in 1985 [2]. In order to obtain multicollisions for RC4-Hash, we simply concatenate the method

Table 4. Example collision pair for RC4-Hash$_{64}$, using fixed points of type II.

	M	$M^\star$
block 1	`s.␣IndesTeEGE␣ANd␣b.␣pREneEl␣-␣`	
(63 bytes)	`cosIc␣-␣colLISioNS␣For␣Rc4-hAsH.`	
block 2	`thiS␣MesSagE␣IS␣pArT␣of␣a␣collis`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
(64 bytes)	`ioN␣EXAmPle␣FOr␣rc4-HASH.␣COSIC.`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
blocks 3–128	`abababababababababababababababab`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
(126×64 bytes)	`abababababababababababababababab`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
	$\vdots$	$\vdots$
	`abababababababababababababababab`	`aBaBaBaBaBaBaBaBaBaBaBaBaBaBaBaB`
block 129	`abababababababababababababababab`	`thiS␣MesSagE␣IS␣pArT␣of␣a␣collis`
(64 bytes)	`abababababababababababababababab`	`ioN␣EXAmPle␣FOr␣rc4-HASH.␣COSIC.`
RC4-Hash$_{64}(M)$ =	`0023dd337650ef0d9b5e77be533ea644198ff0d8f1d8190628d95b9dd04dadf5`	
RC4-Hash$_{64}(M^\star)$	`d9cd2c1ad8adc8555f03ea3819df4128bc96462a53c7e0cc1afffe78db3bd652`$_x$	

from Sect. 4 several times. Concatenating it k times yields 2^k colliding messages. Actually, only part of the method needs to be repeated k times. Indeed, as the value of the pointer j is maintained by the fixed points, only the search for message blocks $M^{0,1}$ has to be repeated. Thus, the expected time for finding 2^k colliding messages for RC4-Hash is $2^7 + k \cdot 2^8$ compression function evaluations. Naturally, also the method of Kelsey and Schneier [6] to construct multicollisions can be applied, and both methods can even be combined.

Mitigating the Attack. The collision attack described in this paper is built on the existence of two types of fixed points of the compression function of RC4-Hash, which were described in Sect 3. These fixed points use patterns where all the (reordered) message bytes are equal (type I) or alternate between two values (type II). Replacing the message reordering $r(\cdot)$ with a message expansion that guarantees that such patterns can never occur foils the attack. Another approach would be to introduce asymmetry, for instance using intermediate rounds.

6 Conclusion

We have shown that RC4-Hash is not collision resistant. There exist two distinct types of fixed points for a number of iterations of the RC4-Hash compression function $\mathcal{C}$. These can be used to construct colliding message pairs with an expected effort of less than 2^9 compression function evaluations. This also leads to multicollisions, yielding 2^k colliding messages with an expected effort of $2^7 + k \cdot 2^8$ compression function evaluations.

References

1. D. Chang, K. C. Gupta and M. Nandi, *"RC4-Hash: A New Hash Function Based on RC4"* In Progress in Cryptology – INDOCRYPT 2006, LNCS, vol. 4329, pp. 80–94, Springer-Verlag, 2006.
2. D. Coppersmith, *"Another Birthday Attack"*, In Advances in Cryptology – CRYPTO 1985, LNCS, vol. 218, pp. 14–17, Springer-Verlag, 1986.
3. C. De Cannière and C. Rechberger, *"Finding SHA-1 Characteristics: General Results and Applications"*, In Advances in Cryptology – ASIACRYPT 2006, LNCS, vol. 4284, pp. 1–20, Springer-Verlag, 2006.
4. H. Finney, *"An RC4 cycle that can't happen"*, Newsgroup post in `sci.crypt`, September 1994.
5. A. Joux, *"Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions"*, In Advances in Cryptology – CRYPTO 2004, LNCS, vol. 3152, pp. 306–316, Springer-Verlag, 2004.
6. J. Kelsey and B. Schneier, *"Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work"*, In Advances in Cryptology – EUROCRYPT 2005, LNCS, vol. 3494, pp. 474–490, Springer-Verlag, 2005.
7. S. Lucks, *"A Failure-Friendly Design Principle for Hash Functions"*, In Advances in Cryptology – ASIACRYPT 2005, LNCS, vol. 3788, pp. 474–494, Springer-Verlag, 2005.
8. B. Schneier, *"Applied Cryptography"*, Second Edition, John Wiley & Sons, 1996.
9. X. Wang and H. Yu, *"How to Break MD5 and Other Hash Functions"*, In Advances in Cryptology – EUROCRYPT 2005, LNCS, vol. 3494, pp. 19–35, Springer-Verlag, 2005.
10. X. Wang, H. Yu and Y. L. Yin, *"Efficient Collision Search Attacks on SHA-0"*, In Advances in Cryptology – CRYPTO 2005, LNCS, vol. 3621, pp. 1–16, Springer-Verlag, 2005.
11. X. Wang, Y. L. Yin and H. Yu, *"Finding Collisions in the Full SHA-1"*, In Advances in Cryptology – CRYPTO 2005, LNCS, vol. 3621, pp. 17–36, Springer-Verlag, 2005.

Appendix

Table 5. The message reordering $r(\cdot)$.

0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15,	16,	17,	18,	19,	20,	21,
22,	23,	24,	25,	26,	27,	28,	29,	30,	31,	32,	33,	34,	35,	36,	37,	38,	39,	40,	41,	42,	43,
44,	45,	46,	47,	48,	49,	50,	51,	52,	53,	54,	55,	56,	57,	58,	59,	60,	61,	62,	63,		
0,	55,	46,	37,	28,	19,	10,	1,	56,	47,	38,	29,	20,	11,	2,	57,	48,	39,	30,	21,	12,	3,
58,	49,	40,	31,	22,	13,	4,	59,	50,	41,	32,	23,	14,	5,	60,	51,	42,	33,	24,	15,	6,	61,
52,	43,	34,	25,	16,	7,	62,	53,	44,	35,	26,	17,	8,	63,	54,	45,	36,	27,	18,	9,		
0,	57,	50,	43,	36,	29,	22,	15,	8,	1,	58,	51,	44,	37,	30,	23,	16,	9,	2,	59,	52,	45,
38,	31,	24,	17,	10,	3,	60,	53,	46,	39,	32,	25,	18,	11,	4,	61,	54,	47,	40,	33,	26,	19,
12,	5,	62,	55,	48,	41,	34,	27,	20,	13,	6,	63,	56,	49,	42,	35,	28,	21,	14,	7,		
0,	47,	30,	13,	60,	43,	26,	9,	56,	39,	22,	5,	52,	35,	18,	1,	48,	31,	14,	61,	44,	27,
10,	57,	40,	23,	6,	53,	36,	19,	2,	49,	32,	15,	62,	45,	28,	11,	58,	41,	24,	7,	54,	37,
20,	3,	50,	33,	16,	63,	46,	29,	12,	59,	42,	25,	8,	55,	38,	21,	4,	51,	34,	17.		

Table 6. The initial value permutation S^{IV}.

145,	57,	133,	33,	65,	49,	83,	61,	113,	171,	63,	155,	74,	50,	132,	248,
236,	218,	192,	217,	23,	36,	79,	72,	53,	210,	38,	59,	54,	208,	185,	12,
233,	189,	159,	169,	240,	156,	184,	200,	209,	173,	20,	252,	96,	211,	143,	101,
44,	223,	118,	1,	232,	35,	239,	9,	114,	109,	161,	183,	88,	66,	219,	78,
157,	174,	187,	193,	199,	99,	52,	120,	89,	166,	18,	76,	241,	13,	225,	6,
146,	151,	207,	177,	103,	45,	148,	32,	29,	234,	7,	16,	19,	91,	108,	186,
116,	62,	203,	158,	180,	149,	67,	105,	247,	3,	128,	215,	121,	127,	179,	175,
251,	104,	246,	98,	140,	11,	134,	221,	24,	69,	190,	154,	253,	168,	68,	230,
58,	153,	188,	224,	100,	129,	124,	162,	15,	117,	231,	150,	237,	64,	22,	152,
165,	235,	227,	139,	201,	84,	213,	77,	80,	197,	250,	126,	202,	39,	0,	94,
42,	243,	228,	87,	82,	27,	141,	60,	160,	46,	125,	112,	181,	242,	167,	92,
198,	172,	170,	55,	115,	30,	107,	17,	56,	31,	135,	229,	40,	111,	37,	222,
182,	25,	43,	119,	244,	191,	122,	102,	21,	93,	97,	131,	164,	10,	130,	47,
176,	238,	212,	144,	41,	14,	249,	220,	34,	136,	71,	48,	142,	73,	123,	204,
206,	4,	216,	196,	214,	137,	255,	195,	26,	8,	51,	178,	2,	138,	254,	90,
194,	81,	245,	106,	95,	75,	86,	163,	205,	70,	226,	28,	147,	85,	5,	110,

```
Input: Key $K$ consisting of $\kappa$ bytes.
Output: Initial internal state of RC4, $\langle S, i, j \rangle$.
1: // RC4 Key Scheduling Algorithm (KSA)
2: $S \leftarrow \{0, 1, \cdots, 255\}$
3: $j \leftarrow 0$
4: for $i = 0$ to 255 do
5:         $j \leftarrow j + S[i] + K[i \bmod \kappa]$
6:         swap($S[i], S[j]$)
7: end for
8: return $\langle S, 0, 0 \rangle$
```

```
Input: RC4 internal state $\langle S, i, j \rangle$.
Output: One byte of keystream, updated internal state.
1: // RC4 pseudo-random byte generation (PRBG)
2: $i \leftarrow i + 1$
3: $j \leftarrow j + S[i]$
4: swap($S[i], S[j]$)
5: return $S[S[i] + S[j]]$
```

Fig. 5. The RC4 stream cipher, consisting of a key scheduling algorithm (top) and a pseudo-random byte generator (bottom). All arithmetic is done modulo 256. [8]