Cryptanalysis of the CRUSH Hash Function

Matt Henricksen[1] and Lars R. Knudsen[2]

[1] Institute for Infocomm Research,
A*STAR, Singapore
mhenricksen@i2r.a-star.edu.sg
[2] Department of Mathematics,
Technical University of Denmark,
DK-2800 Kgs. Lyngby, Denmark
Lars.R.Knudsen@mat.dtu.dk

Abstract. Iterated Halving has been suggested as a replacement to the Merkle-Damgård construction following attacks on the MDx family of hash functions. The core of the scheme is an iterated block cipher that provides keying and input material for future rounds. The CRUSH hash function provides a specific instantiation of the block cipher for Iterated Halving. In this paper, we identify structural problems with the scheme, and show that by using a bijective function, such as the block cipher used in CRUSH or the AES, we can trivially identify collisions and second preimages on many equal-length messages of length ten blocks or more. The cost is ten decryptions of the block cipher, this being less than the generation of a single digest. We show that even if Iterated Halving is repaired, the construction has practical issues that means it is not suitable for general deployment. We conclude this paper with the somewhat obvious statement that CRUSH, and more generally Iterated Halving, should not be used.

Keywords: CRUSH, Iterated Halving, Hash Functions, Cryptanalysis, Collisions, Second preimages.

1 Introduction

The CRUSH hash function was proposed by Gauravaram, Millan and May [3][5] as an example of a new hash-function design paradigm called Iterated Halving, also described in the same paper. Iterated Halving was based on the designers' almost prescient observation that the repeated use of the Merkle-Damgård (MD) construction within hash function design constituted a single point of failure, and that a forthcoming attack on one member of that family may also apply to others. Of course, this view was vindicated by the quick succession of multi-block collision attacks by Wang, *et al.* on MD4, MD5, SHA-1, etc. [6][7][8]. Following this, Gauravaram *et al.* promoted Iterated Halving, specifically CRUSH, as a viable alternative to MD-type hash functions [3][4]. However CRUSH had until now not received significant analysis. In this paper, we demonstrate severe security and implementation problems with the technique of Iterated Halving, and the CRUSH hash function instantiation.

C. Adams, A. Miri, and M. Wiener (Eds.): SAC 2007, LNCS 4876, pp. 74–83, 2007.

Good hash functions possess at least three primary security properties, including collision resistance, preimage resistance and second preimage resistance. Collision resistance refers to the property whereby it is hard to find two messages x and y, such that for $x \neq y, H(x) = H(y)$ for hash function H. Second preimage resistance refers to the difficulty, given message x and its hash $H(x)$, to find y such that $H(y) = H(x)$. A hash function that is not a collision resistant hash function, meaning that it does not offer both collision resistance and second preimage resistance, has limited application. We show that collisions and second-preimages can be found within CRUSH in ten operations. In fact, the same attack applies even if we substitute the Advanced Encryption Standard (AES) [1] with 192-bit or 256-bit keys for CRUSH.

In Section 2, we describe the CRUSH hash function in the context of Iterated Halving. In Section 3, we show how to use the bijective properties of Iterated Halving to find collisions and second preimages on provided messages for CRUSH or instantiations of Iterated Halving using other block ciphers. In Section 4, we show a persistent weakness in the structure of Iterated Halving with relaxed assumptions about the nature of its block cipher core. In Section 5, we indicate a practical issue that shows Iterated Halving to be inferior to Merkle-Damgård construction from an implementation viewpoint. This issue is likely to prevent the uptake on many platforms, of any hash function implemented in the context of Iterated Halving, even assuming that the security issues could be repaired. Finally, in Section 6, we offer closing remarks.

2 The CRUSH Hash Function

The structure of CRUSH is shown in Figure 1. CRUSH consists of a data buffer, a Bijection Selection Component (BSC), boolean functions $f_i, 1 \leq i \leq 3$, and a bijective function B. B is effectively a block cipher, where the plaintext comes from the data buffer, and the key material from the boolean functions. The function B contains a repeated sub-function F, the details of which do not affect our attack.

The authors of CRUSH suggest that another hash function within the Iterated Halving paradigm can be constructed by replacing B with another block cipher. For example, this could be the AES with 192-bit or 256-bit keys.

2.1 Compressing a Message Using CRUSH

Initially the data buffer contains the entire message to be processed – that is, n 64-bit blocks. The value of n is at least 17, and padding must be applied to short messages to accomplish this. The padding includes length encoding, which means that in this paper, we consider only attacks on messages of equal length.

In each round, a pair of blocks, M_a and M_b, are removed from the head of the buffer for processing by the B function. B also accepts three "keying" elements k_1, k_2 and k_3. The output of B is a pair of blocks y_a and y_b. B is a bijective function. Aside from that, its details are unimportant to our attack.

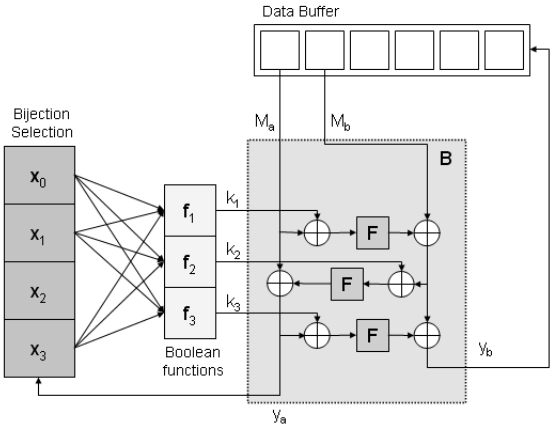

Fig. 1. The CRUSH hash function

Block y_b is appended to the buffer. Thus with each round, the buffer shrinks by one block. After $\frac{n}{2}$ iterations of the B function, there will be $\frac{n}{2}$ blocks in the buffer. After another $\frac{n}{4}$ iterations, the buffer will contain $\frac{n}{4}$ blocks and so on[1]. This is the "halving" aspect of iterated halving. The process is iterated until the point when there are only d blocks remaining in the buffer, where d is the desired digest size in 64-bit blocks. The algorithm terminates, and its output, the digest, is taken as the remaining contents of the buffer. The CRUSH paper suggests a value of $d = 4$, and we analyze CRUSH according to this recommendation. However, the attacks in this paper apply no matter what the value of d.

The BSC holds four 64-bit blocks in its state. In each round, the states in the BSC are shifted such that at time i, block $S_t^i = S_{t+1}^{i-1}$ for $0 \leq t < 3$. State S_0^{i-1} is discarded. The block S_3^i is populated with y_a^i.

Table 1. A toy example of Iterated Halving

Round	Buffer	BSC State	B Function
0	a b c d e f g h	$c_0\ c_1\ c_2\ c_3$	$(a,b) \rightarrow (y_a^1, y_b^1)$
1	c d e f g h y_b^1 –	$y_a^1\ c_0\ c_1\ c_2$	$(c,d) \rightarrow (y_a^2, y_b^2)$
2	e f g h y_b^1 y_b^2 – –	$y_a^2\ y_a^1\ c_0\ c_1$	$(e,f) \rightarrow (y_a^3, y_b^3)$
3	g h y_b^1 y_b^2 y_b^3 – – –	$y_a^3\ y_a^2\ y_a^1\ c_0$	$(g,h) \rightarrow (y_a^4, y_b^4)$
4	y_b^1 y_b^2 y_b^3 y_b^4 – – – –	$y_a^4\ y_a^3\ y_a^2\ y_a^1$	—

A toy example of a message compression illustrates the principle of Iterated Halving below. This example (ignoring padding and message length constraints) compresses an eight-block message $abcdefgh$ into the four-block digest $y_b^1 y_b^2 y_b^3 y_b^4$.

[1] This description does not quite apply if the number of initial blocks is not a power of two. When the number of blocks on the buffer is odd, a dummy block is taken from the BSC to maintain pairing of message blocks. For expediency we do not consider this case in detail.

3 Finding Collisions and Second Preimages

CRUSH implements iterated halving by taking a pair of blocks from the data buffer, processing them concurrently within the B function, returning one element of the pair to the data buffer, and moving the other into the BSC. The BSC and associated boolean functions implement an analog to Key Feedback Mode, whereby one half of the output of one B function round becomes the key for the next. This appears to makes differential cryptanalysis of the function more difficult since the attacker does not have direct control over the key material, which is a highly non-linear function of the input. However, the BSC contains a flaw which allows differential cryptanalysis to be simply applied. Furthermore, the F sub-function within the B block cipher is designed to be bijective so that for x_0, x_1 when $F(x_0) = F(x_1)$, then by necessity $x_0 = x_1$. This is intended to complicate the identification of collisions on the B structure.

However we can easily find collisions and second-preimages by making use of the fact that B is a bijection and reversible. In addition to the bijection property, we use the fact that any difference entering into the BSC endures for only four rounds before it is discarded. By ensuring that differences enter into the BSC rather than the buffer, we need only be concerned with message words for five rounds (ie. ten pairs of words), including the words that introduce the difference.

Recall that at round i, the B block cipher function takes two message words M_a^i and M_b^i as inputs, and produces y_a^i and y_b^i as outputs. Then for the ten pairs of message words $(M_a^i, M_a'^i), (M_b^i, M_b'^i), \ldots, (M_a^{i+4}, M_a'^{i+4})$ and $(M_b^{i+4}, M'i + 4_b)$, we want to use a differential $D1 : (\Delta x_0, \Delta x_1) \rightarrow (\Delta y_0, \Delta 0)$ at all times. This ensures that the difference never goes into the buffer. Without loss of generality, we assume that $i = 0$. However, it can be any value less than $\frac{L}{2} - 5$, where L is the length of the message in 64-bit blocks.

3.1 Identifying Collisions

Because the block cipher B is reversible, it is possible to work backwards through the function by starting with the desired outputs and crafting the message words to satisfy them. To generate a collision, set the outputs of the first call to the block cipher in each message such that $y_a^0 \neq y_a'^0$ and $y_b^0 = y_b'^0$. In the example below, we set $y_a^0 = 0$, $y_a'^0 = 1$, and $y_b^0 = y_b'^0 = 0$. Since the keying material to the block cipher is readily known – the initial state of the BSC is public – it is trivial to calculate M_a^0, M_b^0, $M_a'^0$, $M_b'^0$ (respectively $C51F7BC286ECFA2B$, $877C26794DC08412$ and $BF24001D4DF6E8E5$ and $2DA41C2E7C0DBBCD$ in the example below).

For the following three output words, set $y_a^i = y_a'^i$ and $y_b^i = y_b'^i, 1 \leq i \leq 4$. The only input differences occur in the BSC. In the example below, we set all words equal to 0, but any other value is equally legitimate. Calculate the message words that act as input to B as before. After four rounds, the difference introduced by $y_a^0 \neq y_a'^0$ disappears from the BSC. An internal collision results (ie. the states of both instances of CRUSH are now the same). Of the course, the collision

is maintained by appending identical words to each message, and the resulting digests are identical.

The complexity for finding a collision on any two messages that differ in ten words is just ten decryptions of B (five per message instance). Since all digests, even including a null message, involve the processing of at least seventeen blocks, the collision complexity is at most $\frac{10}{17}$ of generating a single digest.

An example collision on two length-ten messages is shown below.

Message 1 (before padding):

```
C51F7BC286ECFA2B 877C26794DC08412 36E90FCF9B55342D 5CA8F7C75E0DDC9D
64955AF928952224 05C8EFFDC93A2C3B 1009BAE571FC033B 3F9716020BD2918D
AA727095411A8B26 81B4ED776F2ADA41
```

Message 2 (before padding):

```
BF24001D4DF6E8E5 2DA41C2E7C0DBBCD E125E4AFB770AAC9 5C3BEE7C15F27ED0
64955AF928952224 05C8EFFDC93A2C3B  B07155E6E33D428E EAAC9F125F2CF906
BA6A889D411A9B26 2E5A2980EC439EBB
```

Colliding 256-bit digest:

```
AF1E98C1CAFD0CC8 FC0421CCE65C9902 E7478613CD9C2867 D5EAFE307674B3AF
```

3.2 Identifying Second Preimages

The technique for finding second preimages is very similar to that for identifying collisions. It too deals with five rounds and has a complexity of only ten operations.

Generate a digest for message M, taking note of the output of B for rounds $i \ldots i+4$. For the second preimage M', set $y_b'^i = y_b^i$ and $y_a'^i = x, x \neq y_a^i$. Then for the following four rounds for M', set $y_b'^j = y_b^j$ and $y_a'^k = y_a^k$ for $i < j \leq i+4$. Decrypt over B for each $y_a'^i, y_b^i$ to obtain $M_a'^i, M_b^i$.

An example second preimage is shown for the ten-block message containing all zeroes prior to padding. It is easy to find any of the 2^{64} ten-block preimages for this message. The second state word of the BSC is not used as an input to the boolean functions - hence the all zero values in the fifth and sixth words of the preimage.

Message (before padding):

```
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000 0000000000000000 0000000000000000
0000000000000000 0000000000000000
```

256-bit Digest of Message:

```
0923D64E00506F55 FB0B37D9841B5B23 12DEE7CAB0C369B8 1D5F834530C72B22
```

Second preimage (before padding):

```
B0D922EE3E5BF143 D32F46652DA2CEEA 51FE1E43AF1B4800 78F085DFDD4ADC23
0000000000000000 0000000000000000 1136516598C9E675 183DA3ED247AC07D
1030627ADD029C59 C9E9BA651935AE16
```

Digest of Second preimage:

```
0923D64E00506F55 FB0B37D9841B5B23 12DEE7CAB0C369B8 1D5F834530C72B22
```

3.3 Constraints on Messages

These attacks apply to all messages of equal length which are at least ten blocks in length, and differ only in four consecutive words, provided that differences in the BSC are not transferred as dummy blocks to the buffer. This occurs only for the last word of a message with an odd number of blocks, hence is not a major concern to the applicability of our attack. In the majority of cases in which the attack applies, there are 2^{32} choices for the paired values of each word with a difference. These conditions are much less restrictive than the attacks on MD4 [6], etc, which must adhere to specially prescribed bit-conditions. It is very unusual to find such an low complexity attack on a hash function that also has such a large degree of freedom in choice of message words.

4 Corner-Then-Pass: An Observation on Iterated Halving

Our best attack on CRUSH, presented in the previous section, relies on the fact that the B function is bijective and reversible, as is the case whenever B is a block cipher. Here we relax that assumption to make an observation about Iterated Halving that shows that its problems are more deeply ingrained than in the selection of the B function.

In the case where B is not bijective (that is, we replace it with an ideal compression function, using for example, a block cipher with a feed-forward), we are able to generate a collision on a single round by a (standard) birthday attack on B, which is of complexity 2^{64}. For example, find different message words $(M_a^1, M_a'^1)$ and $(M_b^1, M_b'^1)$, such that the outputs $y_a^1 = y_a'^1$ and $y_b^1 = y_b'^1$. Subsequently the internal states for the two messages are equal and a collision is easily found. This attack is far better than a generic birthday collision attack on CRUSH which is of complexity 2^{128} for a 256 bit construction.

Note that in the above collision attack, the inputs to BSC were kept equal in the collision on B. In the defense of CRUSH one may argue that it is possible to find constructions for B such that the complexity of finding these collisions is larger than a standard birthday attack when some of the input is kept constant. However in the following it is demonstrated that iterated halving method contains a fundamental flaw which allows collisions faster than the generic ones for any constructions of B.

We augment differential $D1 : (\Delta x_0, \Delta x_1) \rightarrow (\Delta y_0, \Delta 0)$ from the previous section with $D2 : (\Delta x_2, \Delta x_3) \rightarrow (\Delta 0, \Delta y_1)$. The values of $\Delta x_0, \Delta x_1, \Delta x_2, \Delta x_3, \Delta y_0$

and Δy_1 are immaterial - we are interested in the half collisions $\Delta 0$. Nor do we need to concern ourselves with the internal structure of either B or F to find these half collisions. By the birthday paradox, we can find inputs and outputs that match each differential with an effort[2] of around 2^{32}.

Using differential $D1$ we can keep non-zero (active) differences flowing into the BSC, and using differential $D2$ we can direct active differences into the buffer. In the latter case, the active difference either becomes part of the digest or is processed by B at a later time. If it becomes part of the digest, then no collision is possible. Therefore we want to avoid this outcome in our attack.

On the other hand, using differential $D1$ directs active differences to the BSC, where it will be used in three of the following four rounds as input into the boolean functions. This will affect the subsequent differences into B in a non-linear way, and make them harder to control. We also want to avoid this in our attack.

So it seems that in either case, an active difference hinders our attack. But this is not so. We place message differences very carefully. We then use $D2$ to direct active differences to the buffer until just before the digest is formed, and in the very last round, use the $D1$ differential to route the sole remaining difference to the BSC, removing it from the buffer and creating a collision. The active element is retired to the BSC at just the moment the digest is formed. The algorithm terminates before the boolean functions propagate the difference in the BSC. Therefore the differences coming from the BSC into the B function are zero at all times.

This kind of 'corner-then-pass' technique is possible due to another flaw in Iterated Halving - that the BSC is a sink for differences in message input which may be delayed so that they have no effect on the digest. The technique does not apply to the Merkle-Damgård construction, as it does not have this delay component.

The shortest message pair from which this collision can be generated has a length of twenty 64-bit blocks, due to padding requirements, the need to use both differentials, and on the limitation that the $D1$ differential can only be used to produce the last block of the digest. In this message pair, differences must be introduced only in words 5, 6, 7 and 8.

This can be seen in the diagram below, which shows the differences in the message words within the buffer. For clarity, the message difference $\Delta 0$ is represented by '–', and padding by P. At round 17, the requisite amount of material in the buffer is available and the algorithm terminates.

$\mathrm{D2}_a$ and $\mathrm{D2}_b$ can be generated independently at a cost of 2^{32} operations each. The success of finding a collision on $D2_c$ depends upon the choice of message values made for $\mathrm{D2}_a$ and $\mathrm{D2}_b$, and $\mathrm{D1}_a$ on all three prior message choices. If the

[2] In practice, at the time of publication, the complexity will become slightly higher as storing $2^{32} \times 8$ bytes of material in order to find a match is beyond the capability of most modern hardware. Virtual memory could be used, but this will add a large coefficient to the cost of each operation. In the near future, sufficient volatile memory will be available on commodity machines to avoid this penalty.

Round	Buffer	B
0	$----\ \Delta A\ \Delta B\ \Delta C\ \Delta D\ ----------\ P\ P$	
1	$\Delta A\ \Delta B\ \Delta C\ \Delta D\ ----------\ P\ P\ --$	
2	$\Delta C\ \Delta D\ ----------\ P\ P\ --\ \Delta E$	$D2_a$: $(\Delta A, \Delta B) \rightarrow (\Delta 0, \Delta E)$
3	$----------\ P\ P\ --\ \Delta E\ \Delta F$	$D2_b$: $(\Delta C, \Delta D) \rightarrow (\Delta 0, \Delta F)$
9	$P\ P\ --\ \Delta E\ \Delta F\ ------$	
11	$\Delta E\ \Delta F\ --------$	
12	$--------\ \Delta G$	$D2_c$: $(\Delta E, \Delta F) \rightarrow (\Delta 0, \Delta G)$
16	$\Delta G\ ----$	
17	$----$	$D1_a$: $(\Delta G, \Delta 0) \rightarrow (\Delta H, \Delta 0)$

values do not successfully chain, then backtracking is required. Therefore the complexity of the attack is $2^{33+32+32} = 2^{97}$. This is well below the complexity of finding collisions on the whole hash function by the birthday paradox (ie. 2^{128} for a 256-bit digest), and is an attack that does not depend upon the bijective nature of the embedded block cipher.

5 Iterated Halving Is Not Implementation-Friendly

In [3], it is noted that dedicated hash functions are generally more efficient than hash functions based upon block ciphers. This may be the case in terms of raw throughput, but efficiency is multi-dimensional and Iterated Halving (also [3]), which defines CRUSH as a new dedicated hash function, is an interesting case study in why this statement may not be true in other dimensions, specifically storage costs.

Iterated Halving is embodied by the behaviour of CRUSH's data buffer in accepting one data block for every two that it releases. The buffer is a First-in First-out (FIFO) queue, and blocks must be processed in order. All blocks of the original message must be processed before any blocks produced as output from the B function are reprocessed by B and appended to the buffer. This impacts another dimension of efficiency, since after r rounds, the number of blocks in the buffer will be $min(n - r, d)$. Therefore, assuming streaming of the original message, the buffer must be allocated to contain $\frac{n}{2}$ blocks.

This imposes limits on the length of a message that can be hashed using Iterated Halving. A hardware chip that implements Iterated Halving using x bytes of memory cannot process a $(2 \times x + 1)$-length message. For very large messages, this even applies to general purpose processors aided by virtual memory. The limit imposed by Iterated Halving is far short of the $2^{64} - 1$-bit message limit imposed by the length-encoding of SHA-1.

Consider the performance of SHA-1 on a hardware accelerator. It requires approximately 80 bytes of memory (five 32-bit states, and 512 bits for the current message block), and can handle all real-world messages. It can also be implemented very cheaply, in contrast to a flexible implementation of Iterated Halving, in which the cost is linearly related to the amount of on-board memory and maximum message size. The SHA-1 accelerator does not obsolete faster than its algorithm: it can process a High-Density (HD)-DVD equally as well as

a standard DVD. This is in contrast to an Iterated Halving accelerator carrying 2.5 Gb of onboard memory, which is already substantial and expensive, but incapable of processing in one pass the data on a HD-DVD.

This dimension of efficiency was signposted early on in the history of hash functions. Quoting from Damgård [2],

> ...things seem to get more complex as the length of the messages hashed increase. On the other hand, a hash function is of no use, if we are not allowed to hash messages of arbitrary lengths.

It is important for hash function designers to consider the efficiency of their designs in multiple dimensions, rather than just in terms of raw throughput. Iterated Halving, irrespective of its security flaws, is not suitable for practical deployment.

6 Conclusion

In this paper, we have shown that CRUSH, and more generally Iterated Halving, which mandates a bijective B function, do not satisfy the requirements of good hash functions from either implementation or security viewpoints.

We have shown how to generate collisions and second preimages for Iterated Halving using a block cipher, for messages of equal length in which four words differ. The freedom in choosing those words is very large. The ability to create these collisions and second preimages relies upon the B function of CRUSH being bijective, and upon the BSC rapidly discarding differences. The complexity of these attacks is extremely small, amounting to only ten decryptions of the B function irrespective of the digest size. The attacks apply when any block cipher is used, including the AES with 192-bit or 256-bit keys. It is rare to see practical attacks on symmetric ciphers or hash functions with such a low complexity.

We have also shown that irrespective of the properties of the B function, the structure of Iterated Halving is flawed, because it introduces a delay into the effect of its state upon the message digest. Careful positioning of differences in the message words may result in a collision through the 'corner-and-pass' technique.

Iterated Halving has shown itself to be inefficient relative to the Merkle-Damgård construction, which has a small, fixed memory requirement irrespective of the message length to be hashed. This will almost certainly inhibit the real-life usage of CRUSH and/or Iterated Halving.

CRUSH is not by itself an especially significant hash function. However, the cryptographic community, following the attacks on the MDx family, now has a particular interest in finding alternative constructions to the long established Merkle-Damgård construction. To be useful to industry, such a replacement must meet the efficiency benchmarks set by Merkle-Damgård, and must surpass its level of security. Iterated Halving has been suggested by its designers as such a replacement. As it stands, Iterated Halving as an abstract construction, with CRUSH as a concrete instantiation, is far from the desirable replacement for which the community is searching. We do not recommend the use of CRUSH.

References

1. Daemen, J., Rijmen, V.: The Design of Rijndael: AES—the Advanced Encryption Standard. Springer, Heidelberg (2002)
2. Damgård, I.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
3. Gauravaram, P.: Cryptographic Hash Functions: Cryptanlaysis, Design and Applications. PhD Thesis, Information Security Institute, Faculty of Information Technology, Queensland Unversity of Technology (2007)
4. Gauravaram, P., Millan, W., Dawson, E.P., Viswanathan, K.: Constructing Secure Hash Functions by Enhancing Merkle-Damgård Construction. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP 2006. LNCS, vol. 4058, pp. 407–420. Springer, Heidelberg (2006)
5. Gauravaram, P., Millan, W., May, L.: CRUSH: A New Cryptographic Hash Function Using Iterated Halving Technique. In: Cryptographic Algorithms and Their Uses, QUT, pp. 28–39 (July 2004)
6. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)
7. 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, Heidelberg (2005)
8. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)