Protection of the digital Holy Quran Using SAB hash function

Mohammad A. Ahmad[1,a], Dr. Imad Fakhri Alshaikhli[1,b],

[1] Department of Computer Science, International Islamic University of Malaysia, 53100 Jalan Gombak
Kuala Lumpur, Malaysia, [a]malahmads@yahoo.com, [b]imadf@iium.edu.my

Abstract

Cryptography consists of a set of algorithms and techniques to convert the data into another form so that the contents are unreadable and unexplainable to anyone who does not have the authority to read or write on these data. One of the tools used by cryptography is the hash function. The hash function is used to hash the file so that if anyone tries to modify the text in the file, the number of file obtained from hashing will change. In this paper, we designed a concrete hash function called SAB. SAB hash function uses a permutation *Q* used in Grøstl hash function as a permutation function. Also, it uses the fast wide pipe (FWP) construction designed by Nandi and Paul presented in Indocrypt2010 conference. SAB hash function is designed to protect the data integrity of the digital Holy Quran from alterations or manipulations. SAB hash function outputs *512-bits* as the final digests.

***Keywords*:** FWP, permutation, Digital Holy Quran, SAB hash function

1.Introduction

Allah sent down his book, the Quran, to be the dominant book, the final message, and the law. The Prophet, peace be upon him, had some people write down his revelations, and reviewed it himself, so rest assured regarding the accuracy of what was written. It was the Messenger of Allah, peace be upon him, who forbade writing other than the Quran, such as *hadith* and interpretations of the Quran.

After the death of the Prophet, peace be upon him, a book of Allah and the conservation and care of the religion was manifested through two great incidents, as follows:

First, in the era of the first caliph Abu Bakr, may Allah be pleased with him, while many of the keepers of the Quran died in war, he and a group of senior companions feared that the Quran would be lost. So he ordered the collection of the Quran, by gathering all that was written on wood and leather, as well as what was preserved in the hearts of men. The Quran was collected in one place, supervised by the caliph and his successors after him.

Second, in the era of the third Caliph Othman bin Affan, may Allah be pleased with him, when language differences the led to conflict between Muslims, the Caliph standardized the Muslims on one language, the language of Quraish or Arab tribes, and several copies of the Quran circulated throughout various regions and cities (wikisource, 2009).

The method used to copy the Holy Quran and distribute the copies among the regions was by handwriting. Even with the development of civilization, the progress of the State, and the evolution of technology, handwriting is still the method used in copying and dissemination of the Quran. After writing out the Quran, it is entered into the computers in the form of a scanned image. It can then be printed after auditing by specialists in this area; in recent times, the holy Quran has been printed electronically using the computer, which made the book subject to change and substitution. We need to protect the holy Quran from such alterations.

There are several hash functions can protect the information and files from alterations. This paper discusses the design of a concrete hash function called SAB. SAB hash function uses a permutation *Q* used in Grøstl hash function [1] as a permutation function. Also, it uses the fast wide pipe (FWP) [2] construction designed by Nandi and Paul presented in Indocrypt2010 conference. SAB hash function is designed to protect the data integrity of the digital Holy Quran from alterations or manipulations. SAB hash function outputs *512-bits* as the final digests.

2. Proposed System

SAB is a concrete hash function, takes an input of length up to 2^{64} bits and returns a hash digest with length *512*-bits. We now specify the SAB hash function.

2.1 The hash function construction

The SAB hash functions iterate the compression function *C* which uses the fast wide pipe construction. The fast wide pipe (FWP) construction is the faster version of the wide pipe construction designed by Lucks [3]. The message and chaining input of the wide pipe compression function are *m* and *2n* bits. The wide pipe compression function can be expressed in as $C : \{0, 1\}^{m+2n} \rightarrow \{0, 1\}^{2n}$. But since the FWP is twice faster than the wide pipe construction, its compression function can be expressed as $C : \{0, 1\}^{m+n}$ □$\{0, 1\}^{n}$ where the message and the chaining input to the compression function are $m+n$ and n bits; thus, speeding up the hashing operation by allowing $m+n$ bits of message instead of just m bits per compression function invocation. As Figure 1 describe SAB hash function design, firstly, the message M is padded and split into *l*-bit message blocks $m_1,...,m_L$, and each message block is processed. Two initial *l*-bit values $h_0 = iv_1$ and $h_0^{'} = iv_2$ are defined, and subsequently the message blocks m_i are processed. Algorithm 1 describes FWP construction in more details.

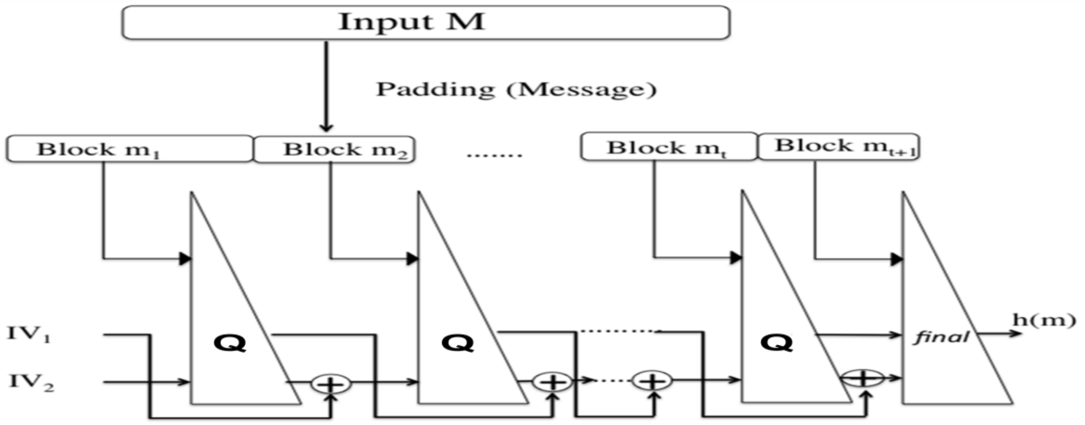

Figure1. SAB hash function

To illustrate algorithm 1, two initial *l*-bit values $h_0 = iv_1 = h_0^{'} = iv_2 = 0^n$ are defined, then, the message M is padded by the padding rule pad(M). The padding rule pad(M) is the execution of the following operation: append *t* zero bits and a 64-bit encoding of |M| to the message M. Select the least integer $t \geq 0$ such that $|M|+t+n+64 = 0 \bmod l$. The length of the original message M is encoded in the last block of message M. FWP_t^C denotes all the compressions functions *C* used in FWP except for the last compression function. In particular, the subfunction in algorithm 1 processes these values ($h_0 = h_{0}^{'}, M_0, M_1, \ldots\ldots, M_{k-2}$) starting from the first compression function and ending in the one before the last compression function. Then, algorithm 1 returns the final truncated hash digest by using the last compression function denoted by *(final)* as shows in Figure 1 and line 4 of algorithm.

Algorithm 1. The FWP mode of operation with the compression function *C* (i.e., FWP^C) [2]

```
Input: Message $M$
Output: Hash output $h$ of size $n$ bits
Initialize: $h_0 = h_0^{'} = 0^n$
1: $M_0||M_1||\ldots||M_{k-1}$=pad$(M)$ where I$M_i$I = $l$ for all $i < k-1$ and I$M_{k-1}$I = $l-n$;
2: ($h_{k-2} = h^{'}_{k-2}$) = $\text{FWP}_t^C$ ($h_0 = h^{'}_{0}, M_0, M_1, \ldots\ldots, M_{k-2}$); /*See Subfunction below*/
```

```
3: $C(h_{k-2} \| h'_{k-2} \| M_{k-1}) = h''_{k-1} \| h'_{k-1}$;
4: return hash output $h = h'_{k-1}$;
Subfunction $\mathrm{FWP}^{C}_{t}$ $(h_0 = h'_{0}, M_0, M_1, \ldots\ldots, M_{k-2})$
5: for $i = 0$ to $k-2$ do
6: $C(h_{i-1} \| M_i) = h_i'' \| h_i'$;
7: $h_i = h_i'' \oplus h'_{i-1}$;
8: end for
9: return $(h_{k-2} = h'_{k-2})$;
```

2.2 The compression function construction

The compression function of SAB is based on the underlying l-bit permutation Q inspired from Grøstl hash function. Hence, the permutation Q of SAB hash function is defined as follows:

$$f_1(h,m) = Q(h \oplus m) \oplus h$$

The construction of f_1 is illustrated in Figure 2. In the next section, we describe how Q is defined.

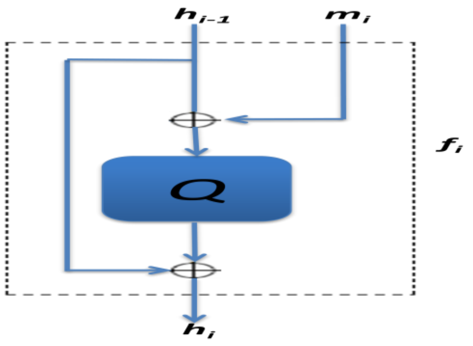

Figure2. The compression functions f_1 of Q permutation

2.3 The output transformation (*final*) m_i

Let φ be the operation that discards all but the trailing n bits. The output transformation Y_1 illustrated in Figure 3 is then defined by

$$Y_1 = trunc_n\, Q \oplus h_{i-2}$$

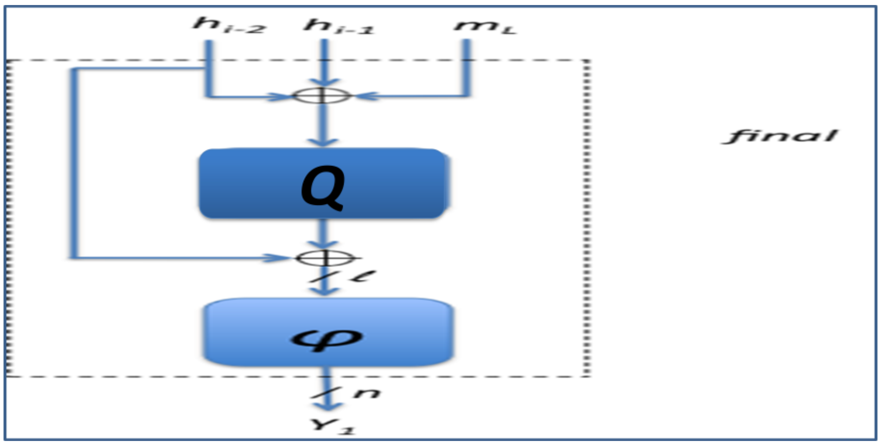

Figure 3. The output transformation φ computes $trunc_n\, Q$

2.4 The design of *Q*

The design of *Q* was inspired from Grøstl [1] which originally inspired by the Rijndael block cipher algorithm [4, 5]. Rijndael design consists of a number of rounds R, which consists of a number of round transformations. Since *Q* is much larger than the 128-bit state size of Rijndael, most round transformations have been redefined. In SAB, a total of four round transformations are defined for each permutation. These are

- AddRoundConstant
- SubBytes
- ShiftBytesWide
- MixBytes.

A round R consists of these four round transformations applied in the above order as illustrated in Figure 4. Hence,

R = MixBytes • SubBytes • SubBytes • AddRoundConstant

We note that all rounds follow this definition. We denote by r the number of rounds. The transformations operate on a state, which is represented as a matrix *A* of bytes (of 8 bits each). The matrix has 8 rows and 16 columns. For the simplicity of exposure, we describe a matrix that has 8 rows and 8 columns, where SAB is the large variant of this concept (8×16) [1].

2.4.1 Mapping from a byte sequence to a state matrix and vice versa

Since SAB operates on bytes, it is generally endianness neutral. However, we need to specify how a byte sequence is mapped to the matrix *A*, and vice versa. This mapping is done in

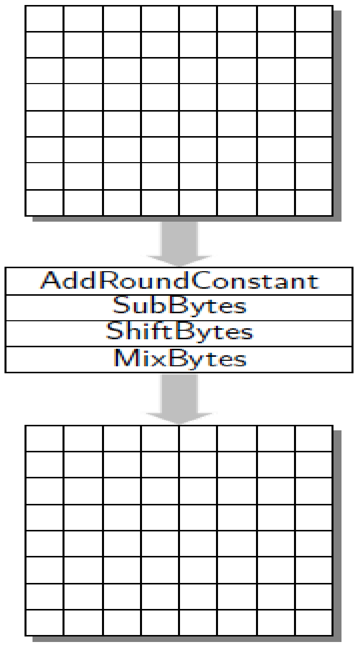

Figure 4. One round of the *SAB* permutations for *Q* is a composition of four basic transformations [1].

similar way as in Rijndael. Hence, the 64-byte sequence 00 01 02 ... 3f is mapped to an 8×8 matrix as [1]

$$\begin{bmatrix} 00 & 08 & 10 & 18 & 20 & 28 & 30 & 38 \\ 01 & 09 & 11 & 19 & 21 & 29 & 31 & 39 \\ 02 & 0a & 12 & 1a & 22 & 2a & 32 & 3a \\ 03 & 0b & 13 & 1b & 23 & 2b & 33 & 3b \\ 04 & 0c & 14 & 1c & 24 & 2c & 34 & 3c \\ 05 & 0d & 15 & 1d & 25 & 2d & 35 & 3d \\ 06 & 0e & 16 & 1e & 26 & 2e & 36 & 3e \\ 07 & 0f & 17 & 1f & 27 & 2f & 37 & 3f \end{bmatrix}.$$

For SAB, we used 8×16 matrix, this method is extended in the natural way. Mapping from a matrix to a byte sequence is simply the reverse operation. From now on, we do not explicitly mention this mapping [1].

2.4.2 AddRoundConstant

The AddRoundConstant transformation adds a round-dependent constant to the state matrix *A*. By addition we mean exclusive-or (XOR). To be precise, the AddRoundConstant transformation in round *i* (starting from zero) updates the state *A* as

$A \leftarrow A\ C[i];$

where *C[i]* is the round constant used in round *i*. The round constants for P_{1024} is [18]

$$P_{1024}: C[i] = \begin{bmatrix} 00 \oplus i & 10 \oplus i & 20 \oplus i & 30 \oplus i & 40 \oplus i & 50 \oplus i & 60 \oplus i & 70 \oplus i & \cdots & \text{f0} \oplus i \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \\ 00 & 00 & 00 & 00 & 00 & 00 & 00 & 00 & \cdots & 00 \end{bmatrix}$$

where *i* is the round number viewed as an 8-bit value.

2.4.3 SubBytes

The SubBytes transformation substitutes each byte in the state matrix by another value, taken from the s-box S. This s-box is the same as the one used in Rijndael as Figure 5 shown. Hence, if $a_{i,j}$ is the element in row *i* and column *j* of *A*, then SubBytes performs the following transformation [1]:

$a_{i,j} \leftarrow S(a_{i,j}),\ 0 \le i < 8,\ 0 \le j < v.$

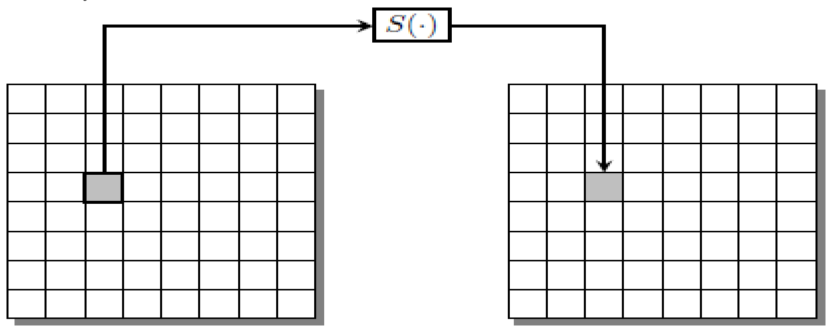

Figure 5. SubBytes substitutes each byte of the state by its image under the s-box S [1].

2.4.4 ShiftBytesWide

ShiftBytesWide cyclically shift the bytes within a row to the left by a number of positions. Let $[\sigma_0, \sigma_1, \ldots, \sigma_7]$ be a list of distinct integers in the range from *0* to *v*-1. Then, ShiftBytesWide moves all bytes in row *i* of the state matrix σ_i positions to the left, wrapping around as necessary. For ShiftBytes in *Q*, we use σ=[0,1,2,3,4,5,6,11] as illustrated in Figure 6 [1].

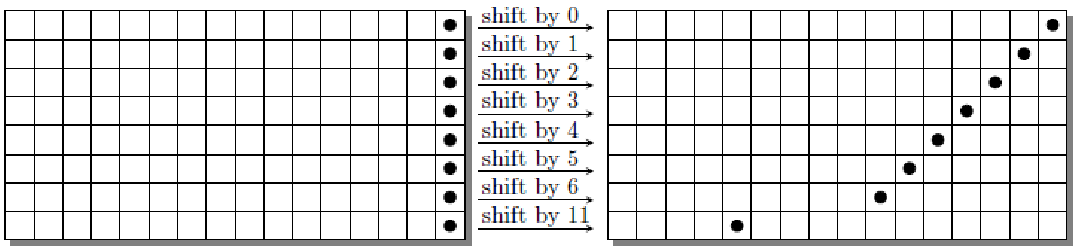

Figure 6. The ShiftBytesWide transformation of permutation Q_{1024} [1].

2.4.5 MixBytes

In the MixBytes transformation, each column in the matrix is transformed independently. To describe this transformation we first need to introduce the finite field F_{256}. This finite field is defined in the same way as in Rijndael via the irreducible polynomial $x^8 \oplus x^4 \oplus x^3 \oplus x \oplus 1$ over F_2. The bytes of the state matrix *A* can be seen as elements of F_{256}, i.e., as polynomials of degree at most 7 with coefficient in {0,1}. The least significant bit of each byte determines the coefficient of x^0, etc [1].

MixBytes multiplies each column of *A* by a constant 8×8 matrix *B* in F_{256}. Hence, the transformation on the whole matrix *A* can be written as the matrix multiplication

$A \leftarrow B \times A.$

The matrix *B* is specified as [18]

$$B = \begin{bmatrix} 02 & 02 & 03 & 04 & 05 & 03 & 05 & 07 \\ 07 & 02 & 02 & 03 & 04 & 05 & 03 & 05 \\ 05 & 07 & 02 & 02 & 03 & 04 & 05 & 03 \\ 03 & 05 & 07 & 02 & 02 & 03 & 04 & 05 \\ 05 & 03 & 05 & 07 & 02 & 02 & 03 & 04 \\ 04 & 05 & 03 & 05 & 07 & 02 & 02 & 03 \\ 03 & 04 & 05 & 03 & 05 & 07 & 02 & 02 \\ 02 & 03 & 04 & 05 & 03 & 05 & 07 & 02 \end{bmatrix}.$$

This matrix is circulant, which means that each row is equal to the row above rotated right by one position. In short, we may write *B* = circ(02, 02, 03, 04, 05, 03, 05, 07) instead. See also Figure 7 [1].

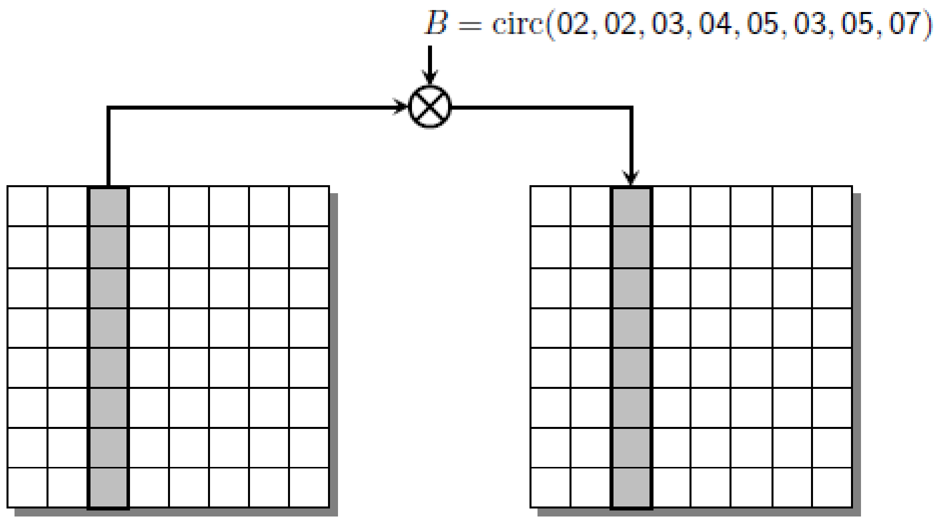

Figure 7. The MixBytes transformation left-multiplies each column of the state matrix treated as a column vector over F_{256} by a circulant matrix *B* [1].

2.4.6 Number of rounds

The number r of rounds is a tunable security parameter. We recommend the following values of r for the four permutations.

Permutations	Digest Size	Recommended value of r
Q_{1024}	512	14

2.5 Initial values

The initial value iv_n of *SAB*-n is the *l*-bit which is *1024-bits* long.

2.6 Padding

As mentioned earlier, the length of each message block is *l*. To be able to operate on inputs of varying length, a padding function pad is defined. This padding function takes a string x of length N bits and returns a padded string x* = pad(x) of a length which is a multiple of *l*.

The padding function does the following. First, it appends the bit ‘1’ to x. Then, it appends w = -N-65mod *l* ‘0’ bits, and finally, it appends a 64-bit representation of $(N + w + 65)/l$. This number is an integer due to the choice of *w*, and it represents the number of message blocks in the final, padded message.

Since it must be possible to encode the number of message blocks in the padded message within 64 bits, the maximum message length is 65 bits short of 2^{64}-1 message blocks. So, the maximum message length in bits is therefore is $1024 \cdot (2^{64}-1) - 65 = 2^{74}-1089$ [1].

2.7 Summary

This section summarizes the complete process of SAB hash function. First, a message M which is to be digested by SAB is padded using the padding function pad. Then, the hash function iterates a compression function $C : \{0, 1\}^l \times \{0, 1\}^l \rightarrow \{0, 1\}^l$ (where *l=1024 -bits)*, which is based on one permutation *Q*. The last compression function *C* is called *final* which truncates the output Y_1 from *1024-bits* to *512-bits*.

3. Conclusion

This paper gives a proposal for new concrete novel design based on permutation hash function named SAB. SAB is based on FWP construction and the permutations *Q* used in SHA-3 finalist Grøstl hash function. SAB hash function is used to protect the digital Holy Quran from alterations or manipulations. The hash digest of SAB is *512-bits* long. Which means, it is infeasible to We leave e software implementation of *SAB* as a future work.

4. References

1) Gauravaram, P., Knudsen, L. R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., & Thomsen, S. S. (2008). Grøstl–a SHA-3 candidate. *Submission to NIST.*

2) Nandi, M. and S. Paul (2010). "Speeding up the wide-pipe: Secure and fast hashing." Progress in Cryptology-INDOCRYPT 2010: 144-162.

3) Lucks, S. (2004). Design principles for iterated hash functions, Cryptology ePrint Archive, Report 2004/253, 2004, http://eprint. iacr. org.

4) J. Daemen and V. Rijmen. AES Proposal: Rijndael. AES Algorithm Submis- sion, September 1999. Available: http://csrc.nist.gov/archive/aes/rijndael/ Rijndael-ammended.pdf (2011/01/15).

5) J. Daemen and V. Rijmen. The Design of Rijndael. Springer, 2002.

6) R.L.Rivest.TheMD4messagedigestalgorithm.InS.Vanstone,editor,AdvancesinCryp- tology - CRYPTO’90, Lecture Notes in Computer Science 537, pages 303–311. Springer Verlag, 1991.

7) R.L. Rivest. The MD5 message-digest algorithm. Request for Comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force, April 1992.

8) NIST. Secure hash standard. FIPS 180-1, US Department of Commerce, Washington D.C., April 1995.

9) H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A strenghened version of RIPEMD. In Gollmann D., editor, Fast Software Encryption, Third International Work- shop, Cambridge, UK, February 1996, Lecture Notes in Computer Science 1039, pages 71–82. Springer Verlag, 1996.

10) D. R. Stinson. Cryptography : Theory and Practice, Second Edition, CRC Press, Inc.
11) Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matt Franklin, editor, Advances in Cryptology- CRYPTO 2004, volume 3152 of Lecture Notes in Computer Science, pages 306–316. Springer, August 15–19 2004.
12) Wang, Xiaoyun, Hongbo Yu, and Yiqun Lisa Yin. "Efficient collision search attacks on SHA-0." *Advances in Cryptology–CRYPTO 2005*. Springer Berlin Heidelberg, 2005.

13) XiaoyunWang, Yiqun Lisa Yin, and Hongbo Yu, 2005.”Finding Collisions in the Full SHA-1, Lecture Notes in Computer Science, Volume 3621, Advances in Cryptology – CRYPTO 2005 Proceedings, pp. 17–36.