Side-Channel Analysis of Six SHA-3 Candidates

Olivier Benoît and Thomas Peyrin

Ingenico, France
forename.name@ingenico.com

Abstract. In this paper we study six 2nd round SHA-3 candidates from a side-channel cryptanalysis point of view. For each of them, we give the exact procedure and appropriate choice of selection functions to perform the attack. Depending on their inherent structure and the internal primitives used (Sbox, addition or XOR), some schemes are more prone to side channel analysis than others, as shown by our simulations.

Keywords: side-channel, hash function, cryptanalysis, HMAC, SHA-3.

1 Introduction

Hash functions are one of the most important and useful tools in cryptography. A n-bit cryptographic hash function H is a function taking an arbitrarily long message as input and outputting a fixed-length hash value of size n bits. Those primitives are used in many applications such as digital signatures or key generation. In practice, hash functions are also very useful for building Message Authentication Codes (MAC), especially in a HMAC [5,33] construction. HMAC offers a good efficiency considering that hash functions are among the fastest bricks in cryptography, while its security can be proven if the underlying function is secure as well [4].

In recent years, we saw the apparition of devastating attacks [38,37] that broke many standardized hash functions [36,30]. The NIST launched the SHA-3 competition [32] in response to these attacks and in order to maintain an appropriate security margin considering the increase of the computation power or further cryptanalysis improvements. The outcome of this competition will be a new hash function security standard to be determined in 2012 and 14 candidates have been selected to enter the 2nd round of the competition (among 64 submissions).

Differential and Simple Power Analysis (DPA and SPA) were introduced in 1998 by Kocher *et al.* [25] and led to a powerful class of attacks called side-channel analysis. They consist in two main steps. First, the power consumption, the electro-magnetic signal [1] or any others relevant physical leakage from an integrated circuit is measured during multiples execution of a cryptographic algorithm. Then, one performs a hypothesis test on subkeys given the algorithm specification, the algorithm input and/or output values and the traces obtained during the first step. This second step requires to compute an intermediary results of the algorithm for a given key guess and all the input/output and analyze correlation [13] with the actual experimental traces. The intermediary result is the output of what we will call hereafter the "selection function".

S. Mangard and F.-X. Standaert (Eds.): CHES 2010, LNCS 6225, pp. 140–157, 2010.

Because of the widely developed utilization of HMAC (or any MAC built upon a hash function) in security applications, it makes sense to consider physical security of hash functions [27,17,28,19,34]. Indeed, those functions usually manipulate no secret and have been at little bit left apart from side-channel analysis for the moment. In practice, the ability to retrieve the secret key that generates the MACs with physical attacks is a real threat that needs to be studied and such a criteria is taken in account by the NIST for the candidates selections [23].

Our contributions. We present a side-channel analysis of six hash functions selected to the 2nd round of the `SHA-3` competition: `ECHO` [7], `Grøstl` [18], `SHAvite-3` [11] (three `AES`-based hash functions), `BLAKE` [3], `CubeHash` [9] and `HAMSI` [35]. This paper aims at finding the appropriate selection function for each SHA-3 candidates in a MAC setting and evaluating the relative efficiency through simulations of the corresponding attacks. Then, we draw conclusions concerning the relative complexity for protecting each candidate against first order side-channel cryptanalysis.

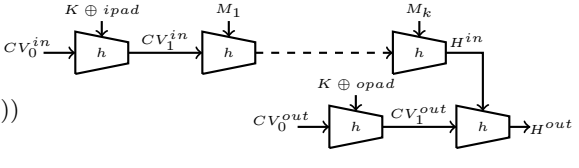
Of course, the intend of this paper is not to show that some particular hash functions can be broken with side-channel analysis, which should be easy in general when the implementation is not protected. However, we believe there are constructions that are naturally harder to attack or easier to implement in a secure and relatively efficient way.

2 How to Perform Side-Channel Attacks on Hash Functions

2.1 Message Authentication Codes with Hash Functions

A Message Authentication Code (MAC) is an algorithm that takes as input a arbitrary long message M and a secret key K and outputs a fixed-length value $V = MAC(M, K)$. One requires that it should be computationally impossible for an attacker to forge a valid MAC without knowing the secret key K, or to retrieve any information about K. This primitive allows the authentication of messages between two parties sharing the same secret key. MACs can be built upon block ciphers (i.e. CBC-MAC [6]) or hash functions in the case of HMAC [5,33]. HMAC instantiated with the hash function H is defined as follows:

$$HMAC(K, M) = H((K \oplus opad)||H((K \oplus ipad)||M))$$



where $||$ denotes the concatenation of bit strings and $\oplus$ represents the bitwise exclusive or (XOR) boolean function. The two words *opad* and *ipad* are two constants of the size of a message block in the iterative hash function. This point

is important: HMAC implicitly considers that H is an iterative hash function. Thus, for each iteration i we have an internal state (so-called chaining variable) CV_i that is updated thanks to a compression function h and a message block $M_i : CV_{i+1} = h(CV_i, M_i)$. The first chaining variable CV_0 is initialized with an initial vector IV and the hash output is the very last chaining variable or a truncated version of it.

It is easy to see that when computing the first hash function call of HMAC $(H^{in} = H((K \oplus ipad)||M)))$, the first iteration is $CV_1 = h(IV, K \oplus ipad)$ and one only needs to guess CV_1 to complete the rest of this hash function computation, whatever the message M. Then, for the second hash function call $(H^{out} = H((K \oplus opad)||H^{in}))$, the same remark applies: one only needs to guess CV_1 to complete the MAC computation whatever H^{in}. We denote by CV_i^{in} the chaining variables for the first hash call and CV_i^{out} the chaining variables for the second one. In practice, one can speed up the implementation by precomputing the CV_1^{in} and CV_1^{out} and starting the two hash processes with those two new initial chaining variables.

Therefore, when attacking HMAC with a side-channel technique, it is very interesting to recover CV_1^{in} and CV_1^{out}. We are now left with the problem of being able to retrieve a fixed unknown chaining variable with random message instances. This will have to be done two times, first for CV_1^{in} and then for CV_1^{out}. The attack process will be identical for each call, so in this article we only describe how to recover the unknown chaining variable with several known random message instances. However, note that attacking CV_1^{in} should be easier than attacking CV_1^{out} because in the former the attacker has full control over the message M, which is not the case in the latter (the incoming message for the second hash call is H^{in}, over which the attacker has no control). For some `SHA-3` candidates, the ability to control the incoming message may reduce the number of power traces needed to recover CV_1^{in}. However, the maximal total improvement factor is at most 2 since the leading complexity phase remains the recovering of CV_1^{out}.

In the case of the so-called stream-based hash functions (for example `Grindahl` [24] or `RadioGatún` [10]), for which the message block size is rather small (smaller than the MAC key and hash output lengths), the HMAC construction is far less attractive and one uses in general the prefix-MAC construction: $MAC(K, M) = H(K||M)$. In order to avoid trivial length-extension attacks (which makes classical Merkle-Damgård [29,14] based hash functions unsuitable for the prefix-MAC construction), the stream-based hash functions are usually composed of a big internal state (much bigger than the hash output length) and define an output function composed of blank rounds (iterations without message blocks incorporation) and a final truncation phase. However, the corresponding side-channel attack for breaking the prefix-MAC construction will not change here and our goal remains to recover the full internal state.

In this paper, we will study the 256-bit versions of the hash functions considered, but in most of the case the analysis is identical for the 512-bit versions. Moreover, when available, the salt input is considered as null. For all candidates

reviewed, the message to hash is first padded and since we only give a short description of the schemes, we refer to the corresponding specification documents for all the details. Finally, since our goal is to recover the internal state before the message digesting phase, there is no need to consider potential output functions performed after all the message words have been processed.

2.2 Side-Channel Attacks

Regardless of the compression function $h(CV, M)$ considered, at some point in the algorithm (usually in the very first stage), the incoming chaining variable CV will be combined with the incoming message block M. The selection functions will be of the form:

$$w = f(cv, m)$$

where cv is a subset of CV and m is a subset from M. Usually w, cv and m have the same size (8 bits in the case of `AES`), but strongly compressing bricks (such as the `DES` Sbox) may impose a smaller w. Ideally the selection function must be non-linear and a modification of 1-bit in one of the input should potentially lead to multiple-bit modifications in the output. For example, the output of a substitution table (Sbox) is a good selection function: block cipher encryption algorithms such as `DES` [16] or AES [15] are very sensitive to side-channel analysis because they both use an Sbox in their construction (a $6 \mapsto 4$-bit Sbox for `DES` and a $8 \mapsto 8$-bit Sbox for AES).

Some algorithms or `SHA-3` candidates (i.e. `BLAKE` or `CubeHash`) do not use such substitution table, while they rely exclusively on modular addition $\boxplus$, rotation $\lll$ and XOR $\oplus$ operations (so-called ARX constructions). In this case, side-channel analysis is still possible but the XOR or modular addition selection functions are less efficient than for the Sbox case. Moreover, it has been theoretically proven that the XOR selection function is less efficient that the modular addition operations [27]. Indeed, the propagation of the carry in the modular addition leads to some non-linearity whereas the XOR operation if completely linear. More precisely, we can quantify the efficiency difference between the `AES` Sbox, the `HAMSI` Sbox, the XOR and the modular addition selection functions by looking at the theoretical correlation results in the so-called hamming-weight model. The rest of this paper exclusively deals with the Hamming weight model since in practice this model leads to good results for the majority of the target devices.

In order to estimate the efficiency of a selection function $f(k, m)$, it is interesting to look at the theoretical correlation $c(j, r)$ between the data set x_i for a key guess j and the data set y_i for an arbitrary real key r. Where $x_i = HW(f(j, m_i))$ and $y_i = HW(f(r, m_i))$, with $i \in [0, \ldots, 2^N - 1]$, N being the number of bits of the selection function input message m and $HW(w)$ being the Hamming Weight of the word w. We also denote by $\overline{x}$ (respectively $\overline{y}$) the average value of the data set x_i (respectively y_i).

$$c(j, r) = \frac{\sum (x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum (x_i - \overline{x})^2} . \sqrt{\sum (y_i - \overline{y})^2}}$$

Of course, when the key guess is equal to the real key ($j = r$), we have $c(j, r) = 1$.

The Figure 1 displays $c(j, 8)$ for $j \in [0, \ldots, 255]$ for the AES selection function $Sbox(k, m)$, for the XOR selection function $k \oplus m$ and for the modular addition selection function $k \boxplus m$. HAMSI is specific because the selection function is using a subset of the Sbox for a given key, therefore, the following table displays $c(j, r)$ for $j \in [0, \ldots, 3]$ and $r \in [0, \ldots, 3]$. Only two bits of the message and two bits of the key are handled in this selection function.

key value \ key guess	$j = 0$	$j = 1$	$j = 2$	$j = 3$
$r = 0$	$+1.00$	-0.17	-0.56	-0.87
$r = 1$	-0.17	$+1.00$	$+0.87$	-0.09
$r = 2$	-0.56	$+0.87$	$+1.00$	$+0.17$
$r = 3$	-0.87	-0.09	$+0.17$	$+1.00$

The efficiency $E(f)$ of the selection function f is directly linked with the correlation contrast c_c between the correct key guess (correlation = 1) and the strongest wrong key guess (correlation = c_w). The higher this contrast, the more efficient the selection function will be to perform a side-channel analysis. Indeed, it will be able to sustain a much higher noise level.

$$c_c = \frac{1-|c_w|}{|c_w|}$$

selection function	AES Sbox	*modular* addition	XOR[1]	HAMSI Sbox
c_w	0.23	0.75	-1	0.87
c_c	3.34	0.33	0	0.15

The values of c_w are extracted from Figure 1 by measuring the highest correlation peak (except the peak with correlation equal to 1 which corresponds to the correct guess). The result of this theoretical/simulation study is the following:

$$E(\texttt{AES Sbox}) > E(\text{modular addition}) > E(\texttt{HAMSI Sbox}) > E(\text{XOR})$$

In the rest of this article, we will search for the best selection function for each SHA-3 candidate analyzed with this conclusion in mind.

3 AES-Based SHA-3 Candidates

In this section, we analyze ECHO [7], Grøstl [18] and SHAvite-3 [11], three AES-based SHA-3 candidates. We recall that the round function of AES is composed of four layers (we use the order AddRoundKey, SubBytes, ShiftRows and MixColumns) and we refer to [31] for a complete specification of this block cipher.

[1] In practice, if the attacker managed to characterize the chip leakage, he eventually can distinguish the wrong guess from the correct guess by taking in consideration the correlation sign (positive or negative). Note that a contrast of zero does not means that the XOR selection function is not yielding any information. Indeed, the attacker have reduced the subkey space from 256 to 2 values (with correlation 1 and -1).

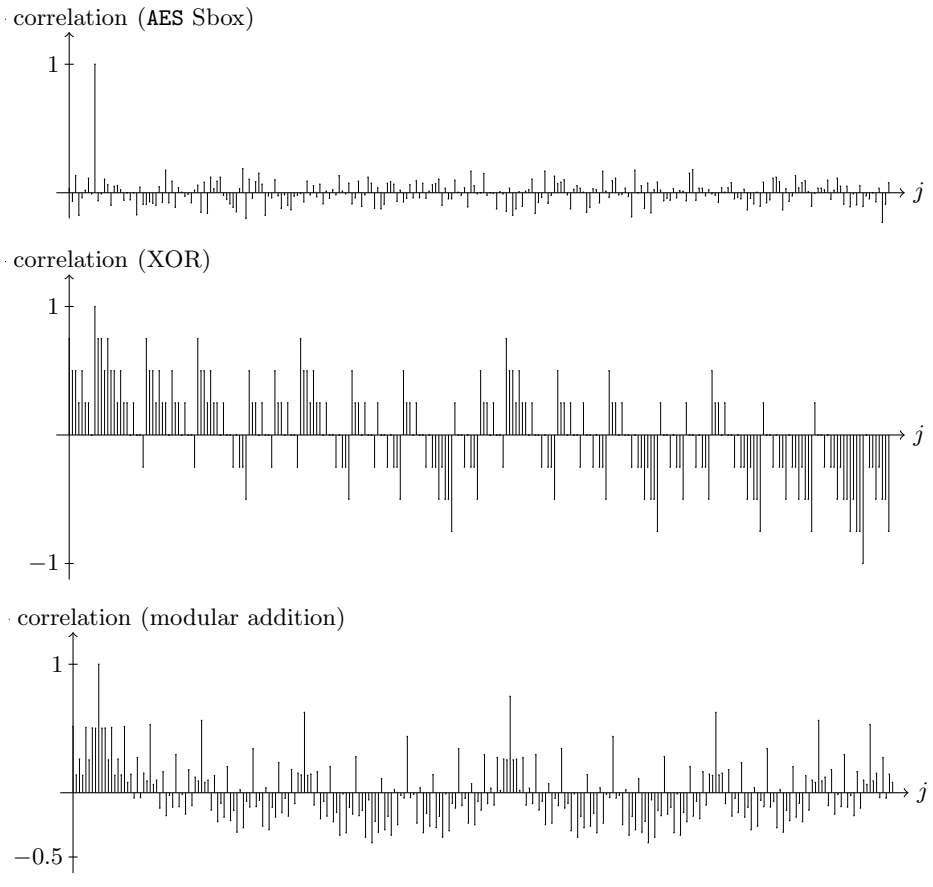


Fig. 1. Correlations $c(j, 8)$ in the Hamming Weight model for the AES Sbox, XOR and modular addition selection function respectively

3.1 ECHO

Description. ECHO [7] is an iterated hash function whose chaining variable CV_i is 512-bit long. Its compression function h maps CV_i and a 1536-bit message block M_i to a new chaining variable CV_{i+1}. More precisely, with CV_i and M_i the compression function h initializes a 2048-bit internal state which is viewed as a 4×4 matrix of 128-bit words (CV_i initializes the first column while M_i fills the three other ones). A fixed-key permutation P_E is applied to this internal state and the output chaining variable is built by calling the shrink_{256} function that XORs all the 512-bit columns together after a feedforward:

$$CV_{i+1} = \mathsf{shrink}_{256}(P_E(CV_i||M_i) \oplus (CV_i||M_i)).$$

The permutation P_E contains 8 rounds, each composed of three functions very similar to the AES ones, but on 128-bit words instead of bytes. First, the BIG.SubBytes

function mimics the application of 128-bit Sboxes on each state word by applying 2 AES rounds with fixed round keys (determined by the iteration and round numbers). Then, BIG.ShiftRows rotates the position in their matrix column of all the 128-bit words according to their row position (analog to the AES). Finally, BIG.MixColumns is a linear diffusion layer updating all the columns independently. More precisely, for one matrix column, it applies sixteen parallel AES MixColumns transformations (one for each byte position in a 128-bit word).

Side-channel analysis. The incoming chaining variable CV fills the first 128-bit words column (denoted cv_i in Figure 2) of the matrix representing the internal state, while the three other columns are filled with the known random incoming message (denoted m_i in Figure 2). The goal of the attacker is therefore to retrieve the words cv_i.

The first layer (BIG.SubBytes) handles each 128-bit word individually. The known and secret data are not mixed yet ($cv_i \mapsto cv'_i$ and $m_i \mapsto m'_i$) and therefore it is not possible to derive a selection function at this point. The same comment applies to the second layer (BIG.ShiftRows) and one has to wait for the third layer (BIG.MixColumns) to observe known and secret data mixing: each column will depend on one secret word cv'_i and three known words m'_i (see Figure 2). More precisely, for each 128-bit word column, BIG.MixColumns applies sixteen parallel and independent AES MixColumns transformations (one for each byte position) and each MixColumns call manipulates one byte of secret and three bytes of known data. Overall, in the end of the first round, every byte $w[b]$ of an internal state word w (we denote $w[b]$ the b-th byte of w) can be written as the following affine equation (see the AES MixColumns definition for the α, β, γ and δ values):

$$w_{i_0}[b] = \alpha \cdot cv'_{i_1}[b] \oplus \beta \cdot m'_{i_2}[b] \oplus \gamma \cdot m'_{i_3}[b] \oplus \delta \cdot m'_{i_4}[b]$$

with $b \in [0, \ldots, 15]$, $i_0 \in [0, \ldots, 15]$, $i_1 \in [0, \ldots, 3]$ and $i_2, i_3, i_4 \in [0, \ldots, 11]$. One could use those $w_i[b]$ as selection functions, but the mixing operation would be the exclusive or. As already explained, the selection function involving an XOR is the least efficient one. It seems much more promising to wait the first layer from the second round of the ECHO internal permutation.

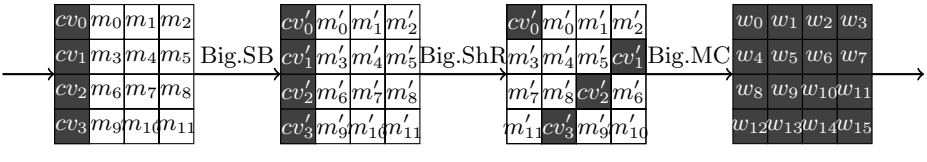


Fig. 2. Recovering the internal state for ECHO. The gray cells represent the words that depends on the initial secret chaining variable. Each cell represents a 128-bit word.

Indeed, the BIG.SubBytes transformation applies directly two AES rounds independently for each words w_i. The first function of the first AES round is the subkey incorporation and in the case of ECHO those subkeys are fully known

constants (we denote them t_i). Then, the second function of the first `AES` round applies the `AES` Sbox to each byte of the internal state. Therefore, we obtain the words w'_i on the output:

$$w'_i[b] = Sbox(w_i[b] \oplus t_i[b]).$$

These equations can be used as selection functions manipulating only the `AES` Sbox which is much more efficient than the XOR case. Overall, one has to perform 64 `AES` Sbox side-channel attacks in order to guess all the words cv'_i byte per byte. By inverting the BIG.SubBytes layer from the words cv'_i, one recovers completely CV. Note that for each byte of cv'_i, one gets 4 selection functions involved. Thus, the overall number of curved can be reduced by a factor 4 at maximum by using and combining this extra information.

3.2 `Grøstl`

Description. `Grøstl` [18] is an iterated hash function whose compression function h maps a 512-bit chaining variable CV_i and a 512-bit message block M_i to a new chaining variable CV_{i+1}. More precisely, two fixed-key permutations P_G and Q_G, only differing in the constant subkeys used, are applied:

$$CV_{i+1} = P_G(CV_i \oplus M_i) \oplus Q_G(M_i) \oplus CV_i.$$

Each permutation is composed of 10 rounds very similar to the `AES` ones, except that they update a 512-bit state, viewed as a 8×8 matrix of bytes (instead of 4×4). Namely, for each round, constant subkeys are first XORed to the state (AddRoundConstant), then the `AES` Sbox is applied to each byte (SubBytes), the matrix rows are rotated with distinct numbers of positions (ShiftBytes) and finally a linear layer is applied to each byte column independently (MixBytes). This is depicted in Figure 3.

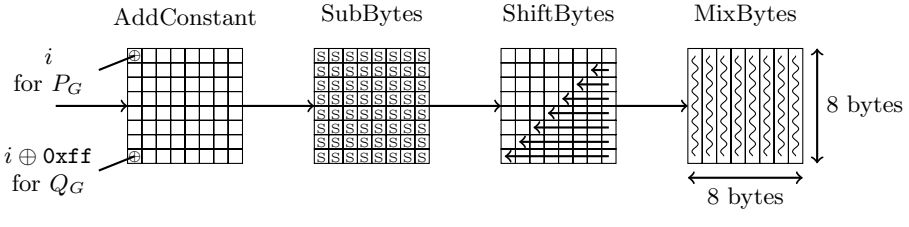


Fig. 3. One round of the internal permutation P_G of `Grøstl`. Each cell represents a byte.

Side-channel analysis. The `Grøstl` case is very simple. One can see that the incoming message block M is processed trough the Q_G permutation. Since the output of this permutation only depends on the message block and not on the incoming chaining variable CV, we can completely ignore this part of the compression function. Then, the permutation P_G takes as input $M \oplus CV$ and

one may be tempted to perform the side-channel attack during this operation. As demonstrated earlier, it is much more convenient to wait for the opportunity to attack the AES Sbox instead. The first layer of the permutation P_G is the AddConstant function which XORs the round number i (the counting starting from 0) to the top left byte of the internal and therefore the operation is fully transparent. Then, the second layer of the first P_G round is the SubBytes function which applies the AES Sbox to every byte of the internal state $w = M \oplus CV$:

$$w[b] = m[b] \oplus CV[b]$$

with $b \in [0, \ldots, 63]$. The output state is denoted w' and we obtain the following selection function which recovers CV byte per byte:

$$w'[b] = Sbox(w[b]).$$

Note that it is possible to improve this attack when dealing with CV_1^{in} (the unknown chaining variable for the first hash call) by choosing appropriately the message. More precisely, one can divide the number of power traces by a factor 64 when choosing all $m[b]$ as equals. Indeed, this allows to perform in parallel the side-channel analysis of the 64 unknown bytes.

3.3 SHAvite-3

Description. SHAvite-3 [11] is an iterated hash function whose compression function h maps a 256-bit chaining variable CV_i and a 512-bit message block M_i to a new chaining variable CV_{i+1}. Internally, we have a block cipher E^S in classical Davies-Meyer mode

$$CV_{i+1} = CV_i \oplus E^S_{M_i}(CV_i).$$

This block cipher derives many subkeys thanks to a key schedule (all subkeys depending on the message M_i only) and is composed of 12 rounds of a 2-branch Feistel scheme (128 bits per branch). The basic primitive in the Feistel rounds is the application of 3 AES rounds with subkeys incoming from the key schedule.

Side-channel analysis. For SHAvite-3, we divide the attack in two phases (see Figure 4). In the first one, we recover the right part (in the Feistel separation) of the incoming chaining variable (CV^R) during the first round. Once this first phase succeeded, we recover the left part of the incoming chaining variable (CV^L) during the second round. The message expansion maps the incoming message M to three 128-bit message words (m_0^j,m_1^j,m_2^j) for each round j. One round j of SHAvite-3 consists in executing sequentially three AES round functions with as input one branch of the current SHAvite-3 state and (m_0^j,m_1^j,m_2^j) as subkeys. Consequently, for the first SHAvite-3 round, the secret vector (CV^R) is mixed with the known message word m_0^1 during the AddRoundKey layer of the first AES round and we note:

$$w[b] = CV^R[b] \oplus m_0^1[b].$$

with $b \in [0, \ldots, 15]$. One could use this equation as the selection function, but it is more appropriate to use the output of the very next transformation instead, i.e. the SubBytes layer:

$$w'[b] = Sbox(w[b]).$$

Before executing the second round of `SHAvite-3`, the left part of the chaining variable (CV^L) is XORed with the output w'' of the three `AES` rounds. Then, this word is mixed with $m_0^2[b]$ just before the first `AES` round of the second `SHAvite-3` round and we note:

$$z[b] = CV^L[b] \oplus w''[b] \oplus m_0^2[b].$$

Obviously, after a successful first phase, it is possible to compute $w''[b]$ and therefore CV^L is the only unknown constant. Once again, one could use this equation as the selection function, but it is more appropriate to use the output of the very next transformation instead, i.e. the SubBytes layer:

$$z'[b] = Sbox(z[b]).$$

Overall, we recover byte per byte the CV^L and CV^R values.

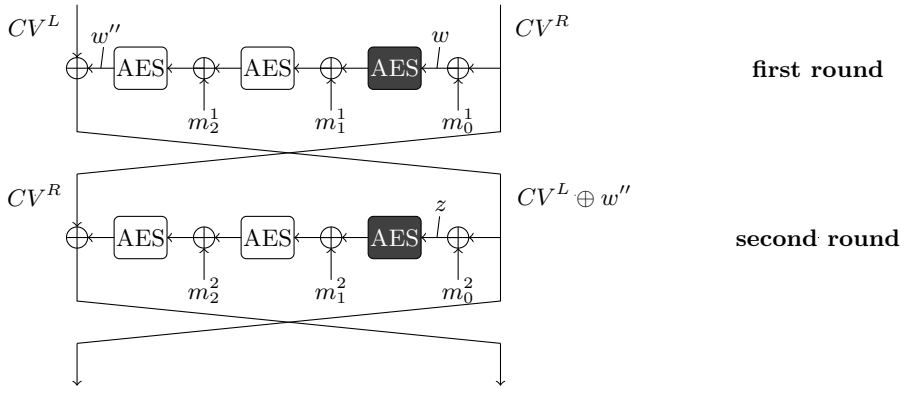


Fig. 4. Recovering the internal state for `SHAvite-3`. The AES rounds we use for recovering the internal state are depicted in black.

4 Other SHA-3 Candidates

In this section, we analyze `BLAKE` [3], `CubeHash` [9] and `HAMSI` [35], three 2nd round `SHA-3` candidates.

4.1 BLAKE

Description. `BLAKE` [3] is an iterated hash function whose compression function h maps a 256-bit chaining variable CV_i and a 512-bit message block M_i to a new

chaining variable CV_{i+1}. Internally, the update is done with a block cipher E^B, keyed with the message block (see Figure 5):

$$CV_{i+1} = final(E^B_{M_i}(init(CV_i)), CV_i).$$

where the *init* function initializes the 512-bit internal state with CV_i and constants. The *final* function computes the output chaining variables according to CV_i, constants and the internal state after the application of E^B. The internal state is viewed as a 4×4 matrix of 32-bit words and the block cipher E^B is composed of 10 rounds, each consisting of the application of eight 128-bit sub-functions G_i. Assume an internal state for `BLAKE` with v_{i+4j} representing the 32-bit word located on row j and column i, one round of E^B is:

$$\begin{array}{cccc} G_0(v_0, v_4, v_8, v_{12}) & G_1(v_1, v_5, v_9, v_{13}) & G_2(v_2, v_6, v_{10}, v_{14}) & G_3(v_3, v_7, v_{11}, v_{15}) \\ G_4(v_0, v_5, v_{10}, v_{15}) & G_5(v_1, v_6, v_{11}, v_{12}) & G_6(v_2, v_7, v_8, v_{13}) & G_7(v_3, v_4, v_9, v_{14}) \end{array}$$

A sub-function G_i incorporates 32-bit message chunks m_i and is itself made of additions, XORs and rotations. During the round r, the function $G_s(a, b, c, d)$ processes the following steps:

$$\begin{array}{l} a \leftarrow (a \boxplus b) \boxplus (m_i \oplus k_j) \\ d \leftarrow (d \oplus a) \ggg 16 \\ c \leftarrow (c \boxplus d) \\ d \leftarrow (b \oplus c) \ggg 12 \\ a \leftarrow (a \boxplus b) \boxplus (m_j \oplus k_i) \\ d \leftarrow (d \oplus a) \ggg 8 \\ c \leftarrow (c \boxplus d) \\ d \leftarrow (b \oplus c) \ggg 7 \end{array}$$

where $\ggg x$ denotes the right rotation of x positions, $i = \sigma_r(2s)$ and $j = \sigma_r(2s = 1)$. The notation σ_r represents a family of permutations on $\{0, \ldots, 15\}$ defined in the specifications document.

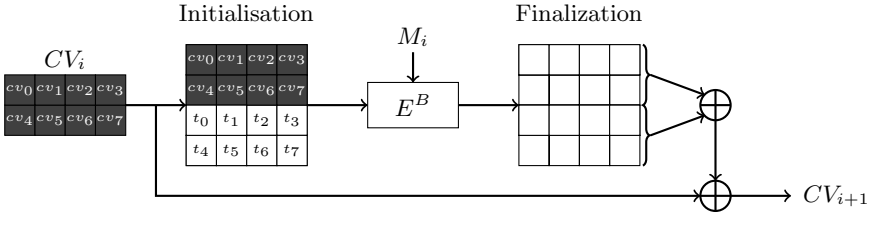


Fig. 5. The compression function of `BLAKE`

Side-channel analysis. The sixteen 32-bit internal state words are initialized with the eight secret chaining value CV words (denoted cv_i) and constants values t_i (see Figure 5). Then, the eight G_i functions during the first `BLAKE` round are applied to the internal state and one can check that the two first parameters of

G_0, G_1, G_2 and G_3 are (cv_0, cv_1), (cv_2, cv_3), (cv_4, cv_5) and (cv_6, cv_7) respectively. Our goal is therefore to recover a_0 and b_0 when applying $G_i(a_0, b_0, c_0, d_0)$ with $0 \leq i \leq 3$. The functions G_i consist in a sequence of eight transformations, the five first being:

$$\begin{aligned} a_1 &= (a_0 \boxplus b_0) \boxplus m_k \\ d_1 &= (d_0 \oplus a_1) \gg 16 \\ c_1 &= c_0 \boxplus d_1 \\ b_1 &= (b_0 \oplus c_1) \gg 12 \\ a_2 &= a_1 \boxplus b_1 \boxplus m_l \end{aligned}$$

In practice, the first transformation will be computed in one of the three following way:

$$\begin{aligned} &\text{first } a \leftarrow a \boxplus b \text{ then } a \leftarrow a \boxplus m_i \\ &\text{first } a \leftarrow a \boxplus m_i \text{ then } a \leftarrow a \boxplus b \\ &\text{first } x \leftarrow b \boxplus m_i \text{ then } a \leftarrow a \boxplus x \end{aligned}$$

For the second and third case, a_0 and b_0 can be found by two side-channel analysis applied successively to the two modular addition selection function (working byte per byte):

$$w_i = cv_i \boxplus m_k \text{ and } z_i = cv_{i+1} \boxplus w_i.$$

For the first case, $a_0 \boxplus b_0$ can be recovered by performing the side-channel analysis on the second modular addition selection function. In order to solve the problem and estimate a_0 and b_0 the attacker has to target the output of the fourth transformation of G_i. However, in this case the selection function would be based on the XOR operation. Therefore it seems more interesting to aim for the fifth transformation of G_i, a modular addition.

4.2 `CubeHash`

Description. `CubeHash` [9] is an iterated hash function whose compression function h maps a 1024-bit chaining variable CV_i and a 256-bit message block M_i to a new chaining variable CV_{i+1}. Internally, the update is done with a permutation P_C:

$$CV_{i+1} = P_C(CV_i \oplus (M_i || \{0\}^{768})).$$

The internal state is viewed as a table of 32 words of 32 bits each. The permutation P_C is composed of 16 identical rounds and one round is made of ten layers (see Figure 6): two intra-word rotation layers, two XOR layers, two 2^{32} modular addition layers and four word positions permuting layers.

Side-channel analysis. The attack is divided into 4 steps. The incoming chaining variable CV fills an internal state represented by a vector of four 256-bit words or eight 32-bit words (denoted cv_i). The known random incoming message M is first XORed with cv_0 and then starts the first round of permutation P_C. Thus, the first selection function is of XOR type and recovers cv_0 byte per byte:

$$w[b] = cv_0[b] \oplus M[b].$$

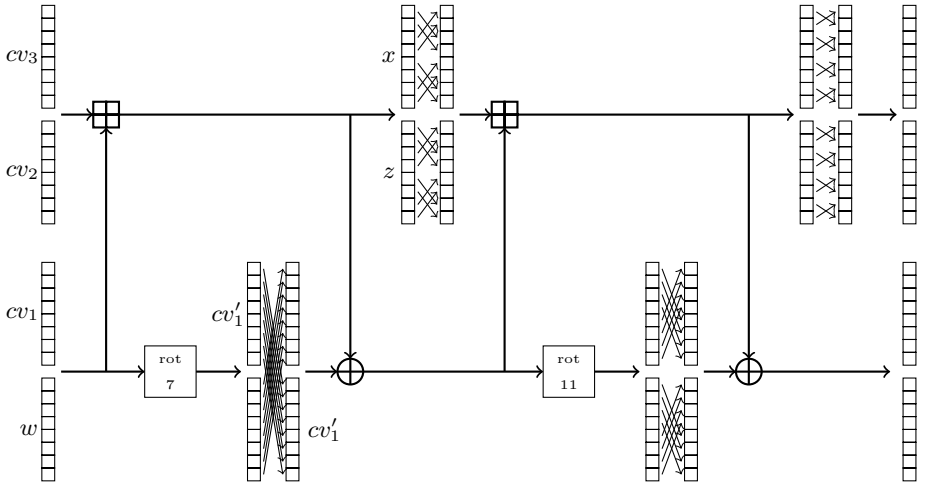


Fig. 6. Recovering the internal state of `CubeHash` during one round of the internal permutation P_C. Each cell represents a 32-bit word.

with $b \in [0, \ldots, 7]$. Once this first step successfully performed, the attacker fully knows w and the first layer of P_C adds each 32-bit words from w to those from cv_2 modulo 2^{32} (and each words from cv_1 to those from cv_3). The second selection function is therefore of modular addition type and recovers cv_2 byte per byte (starting from the LSB of the modular addition):

$$z[b] = cv_2[b] \boxplus w[b].$$

The second layer of P_C applies a rotation to each 32-bit word of w and cv_1 and then XORs z to this rotated version of cv_1 (denoted cv_1'). The selection function for the third step is then of XOR type and recovers cv_1' byte per byte:

$$y[b] = cv_1'[b] \oplus z[b].$$

Finally, going backward in the computation to the first P_C layer, the selection function for the fourth step is of modular addition type and recovers cv_3 byte per byte (starting from the LSB of the modular addition):

$$x[b] = cv_3[b] \boxplus cv_1[b].$$

Note that each step must be successful, otherwise it would compromise the results of the following steps.

4.3 `HAMSI`

Description. `HAMSI` [35] is an iterated hash function whose compression function h maps a 256-bit chaining variable CV_i and a 32-bit message block M_i to a new chaining variable CV_{i+1}. First, the message block is expanded into eight

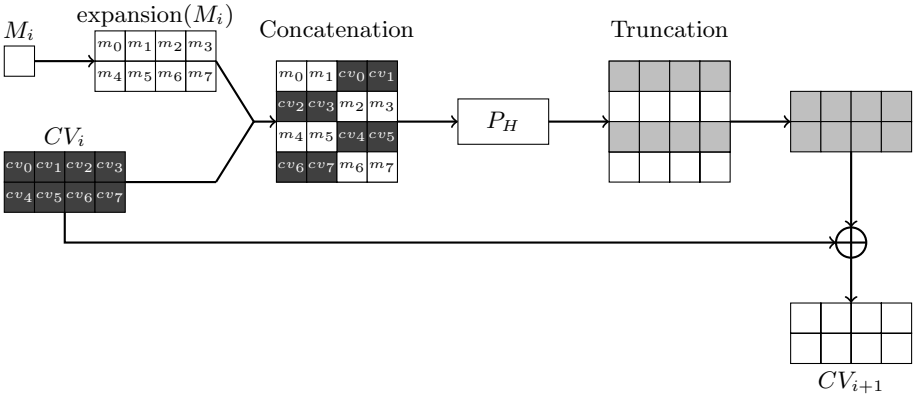


Fig. 7. The compression function of HAMSI. Each cell represents a 32-bit word.

32-bit words m_i that are concatenated to CV_i in order to initialize a 512-bit internal state (viewed as a 4×4 matrix of 32-bit words). Then a permutation P_H is applied to the state and a truncation allows to extract 256 bits. In the end, there is a feedforward of the chaining variable (see Figure 7):

$$CV_{i+1} = trunc(P_H(CV_i||expansion(M_i))) \oplus CV_i.$$

The permutation P_H contains three identical rounds. One round is composed of three layers: constants are first XORed to the internal state, then 4-bit Sboxes are applied to the whole state by taking one bit of each 32-bit word of the same column of the 4×4 matrix and repeating the process for all bit positions. Finally, a linear layer is applied to four 32-bit words diagonals of the 4×4 matrix independently (see Figure 8).

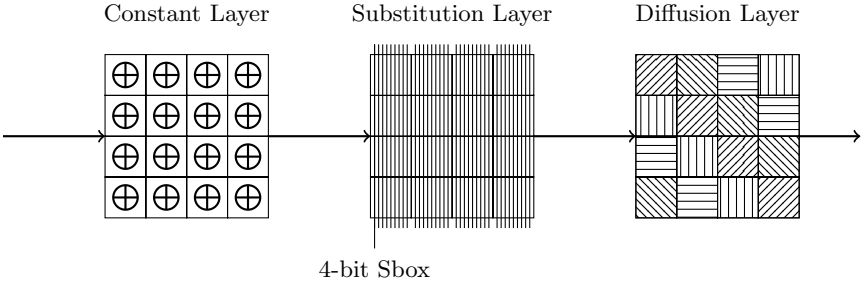


Fig. 8. One round of the internal permutation P_H of HAMSI. Each cell represents a 32-bit word.

Side-channel analysis. The known random message M (after expansion) and the secret chaining variable CV fill the internal state matrix as shown in Figure 7

(32-bit words are denoted m_i and cv_i). The first layer of the permutation P_H XORs each matrix element with a constant t_i:

$$m'_i = m_i \oplus t_i \text{ and } cv'_i = cv_i \oplus t_{i+8}.$$

Then is applied the HAMSI Sbox layer. This Sbox is acting over 4-bits (one bit per word located in the same column of the state matrix). Therefore, the input of each Sbox is composed of 2 known message bits and 2 unknown chaining variable bits. The generic selection function for HAMSI can therefore be written as:

$$w = Sbox(m'_i[b]||cv'_{i+2}[b]||m'_{i+4}[b]||cv'_{i+6}[b])$$

or

$$w = Sbox(cv'_i[b]||m'_{i+2}[b]||cv'_{i+4}[b]||m'_{i+6}[b])$$

for $i \in [0, 1]$ and $b \in [0, \ldots, 127]$ where b is the bit index in a 128-bits word. Overall, one recovers two bits of cv'_i at a time with a total of 4 times 128 correlation computations (with 4 guess each).

5 Conclusion and Discussions

For each hash proposal considered in this article, we described an appropriate selection function for an efficient side-channel attack.

In the case of the AES-based SHA-3 candidates, we did not found significant differences of performance when choosing the selection function. Indeed, in ECHO, Grøstl and SHAvite-3, one has to compute several AES Sbox side-channel attacks in order to retrieve the full secret internal state. Up to a small complexity/number of power traces factor, the three schemes seem to provide the same natural vulnerability to side-channel cryptanalysis. As expected, their situation is therefore very close to the real AES block cipher.

Attacking BLAKE seems feasible in practice since we managed to derive a modular addition selection function for recovering the 256 bits of unknown chaining variable. The modular addition non-linearity is very valuable for the attacker as it increases the correlation contrast. Then, for CubeHash (a typical ARX function) we tried to force the selection function to be of modular addition type as much as possible. Overall, 512 bits can be recovered with modular addition selection function and 512 bits with XOR selection function. In practice, depending on the underlying hardware, it could be challenging to mount the attack. One must notice that the internal state is bigger for CubeHash than for other candidates. This is an additional strength since in practice, if the side-channel attack gives only probabilistic results, the final exhaustive search complexity will be higher. Finally, for HAMSI, the attack would be difficult to mount despite the fact that a substitution table is used. Indeed, the correlation contrast for this primitive is quite low compared to the AES Sbox. We believe that a better selection function involving a modular addition might possibly be found in the inner layer.

Of course, those results concern unprotected implementations and the ranking would be really different if we also considered methods for hardening the side-channel cryptanalysis. For example, in the case of AES-based hash functions, one could perform secure round computations and leverage all the research achieved so far on this subject [2,20]. Also, as ECHO processes parallel AES rounds, we believe it could benefit from secure bit-slice implementations regarding some side-channels attacks [26], while maintaining its normal use efficiency. Finally, ECHO and SHAvite-3 can take advantage of the natural protection inherited from the hardware implementations of the AES round such as the new AES NI instruction set on Intel microprocessor [8].

Side-channel countermeasures for ARX constructions such as BLAKE or CubeHash are of course possible, but they will require to constantly switch from boolean to arithmetic masking. As a consequence, one will observe an important decrease of the speed performance for secure implementations. AES-based hash functions seem naturally easier to attack regarding side-channel cryptanalysis, but are also easier to protect.

References

1. Agrawal, D., Archambeault, B., Rao, J.R., Rohatgi, P.: The EM Side-Channel(s). In: Jr., et al. (eds.) [22], pp. 29–45.
2. Akkar, M.-L., Giraud, C.: An Implementation of DES and AES, Secure against Some Attacks. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 309–318. Springer, Heidelberg (2001)
3. Aumasson, J.-P., Henzen, L., Meier, W., Phan, R.C.-W.: SHA-3 proposal BLAKE. Submission to NIST (2008)
4. Bellare, M.: New Proofs for NMAC and HMAC: Security Without Collision-Resistance. Cryptology ePrint Archive, Report 2006/043 (2006), http://eprint.iacr.org/
5. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
6. Bellare, M., Kilian, J., Rogaway, P.: The Security of Cipher Block Chaining. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 341–355. Springer, Heidelberg (1994)
7. Benadjila, R., Billet, O., Gilbert, H., Macario-Rat, G., Peyrin, T., Robshaw, M., Seurin, Y.: SHA-3 Proposal: ECHO. Submission to NIST (2008), http://crypto.rd.francetelecom.com/echo/
8. Benadjila, R., Billet, O., Gueron, S., Robshaw, M.J.B.: The Intel AES Instructions Set and the SHA-3 Candidates. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 162–178. Springer, Heidelberg (2009)
9. Bernstein, D.J.: CubeHash specification (2.B.1). Submission to NIST, Round 2 (2009)
10. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Radiogatun, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara, August 24-25 (2006), http://radiogatun.noekeon.org/
11. Biham, E., Dunkelman, O.: The SHAvite-3 Hash Function. Submission to NIST, Round 2 (2009), http://www.cs.technion.ac.il/~orrd/SHAvite-3/Spec.15.09.09.pdf

12. Brassard, G. (ed.): CRYPTO 1989. LNCS, vol. 435. Springer, Heidelberg (1990)
13. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, Quisquater (eds.) [21], pp. 16–29
14. Damgård, I.: A Design Principle for Hash Functions. In: Brassard (ed.) [12], pp. 416–427
15. FIPS 197. Advanced Encryption Standard. Federal Information Processing Standards Publication 197, U.S. Department of Commerce/N.I.S.T. (2001)
16. FIPS 46-3. Data Encryption Standard. Federal Information Processing Standards Publication, U.S. Department of Commerce/N.I.S.T. (1999)
17. Fouque, P.-A., Leurent, G., Réal, D., Valette, F.: Practical Electromagnetic Template Attack on HMAC. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 66–80. Springer, Heidelberg (2009)
18. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submission to NIST (2008), http://www.groestl.info
19. Gauravaram, P., Okeya, K.: An Update on the Side Channel Cryptanalysis of MACs Based on Cryptographic Hash Functions. In: Srinathan, K., Pandu Rangan, C., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 393–403. Springer, Heidelberg (2007)
20. Golic, J.D., Tymen, C.: Multiplicative Masking and Power Analysis of AES. In: Jr, et al. (eds.) [22], pp. 198–212
21. Joye, M., Quisquater, J.-J. (eds.): CHES 2004, MA, USA, August 11-13. LNCS, vol. 3156. Springer, Heidelberg (2004)
22. Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.): CHES 2002. LNCS, vol. 2523. Springer, Heidelberg (2003)
23. Kelsey, J.: How to Choose SHA-3, http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Kelsey.pdf
24. Knudsen, L.R., Rechberger, C., Thomsen, S.S.: The Grindahl Hash Functions. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 39–57. Springer, Heidelberg (2007)
25. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
26. Könighofer, R.: A Fast and Cache-Timing Resistant Implementation of the AES. In: Malkin, T.G. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 187–202. Springer, Heidelberg (2008)
27. Lemke, K., Schramm, K., Paar, C.: DPA on n-Bit Sized Boolean and Arithmetic Operations and Its Application to IDEA, RC6, and the HMAC-Construction. In: Joye, Quisquater (eds.) [21], pp. 205–219
28. McEvoy, R.P., Tunstall, M., Murphy, C.C., Marnane, W.P.: Differential Power Analysis of HMAC Based on SHA-2, and Countermeasures. In: Kim, S., Yung, M., Lee, H.-W. (eds.) WISA 2007. LNCS, vol. 4867, pp. 317–332. Springer, Heidelberg (2008)
29. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard (ed.) [12], pp. 428–446
30. National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard (April 1995), http://csrc.nist.gov
31. National Institute of Standards and Technology. FIPS PUB 197, Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce (2001)

32. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a NewCryptographic Hash Algorithm (SHA-3) Family. Federal Register, 27(212):62212–62220 (November 2007), http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf (2008/10/17)
33. NIST. FIPS 198 – The Keyed-Hash Message Authentication Code (HMAC) (2002)
34. Okeya, K.: Side Channel Attacks Against HMACs Based on Block-Cipher Based Hash Functions. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP 2006. LNCS, vol. 4058, pp. 432–443. Springer, Heidelberg (2006)
35. Kücük, Ö.: The Hash Function Hamsi. Submission to NIST (updated) (2009)
36. Rivest, R.L.: RFC 1321: The MD5 Message-Digest Algorithm (April 1992), http://www.ietf.org/rfc/rfc1321.txt
37. 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)
38. 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, Heidelberg (2005)