Linearization Framework for Collision Attacks: Application to CubeHash and MD6*

Eric Brier[1], Shahram Khazaei[2], Willi Meier[3], and Thomas Peyrin[1]

[1] Ingenico, France
[2] EPFL, Switzerland
[3] FHNW, Switzerland

Abstract. In this paper, an improved differential cryptanalysis framework for finding collisions in hash functions is provided. Its principle is based on linearization of compression functions in order to find low weight differential characteristics as initiated by Chabaud and Joux. This is formalized and refined however in several ways: for the problem of finding a conforming message pair whose differential trail follows a linear trail, a condition function is introduced so that finding a collision is equivalent to finding a preimage of the zero vector under the condition function. Then, the dependency table concept shows how much influence every input bit of the condition function has on each output bit. Careful analysis of the dependency table reveals degrees of freedom that can be exploited in accelerated preimage reconstruction under the condition function. These concepts are applied to an in-depth collision analysis of reduced-round versions of the two SHA-3 candidates CubeHash and MD6, and are demonstrated to give by far the best currently known collision attacks on these SHA-3 candidates.

Keywords: Hash functions, collisions, differential attack, SHA-3, CubeHash and MD6.

1 Introduction

Hash functions are important cryptographic primitives that find applications in many areas including digital signatures and commitment schemes. A hash function is a transformation which maps a variable-length input to a fixed-size output, called message *digest*. One expects a hash function to possess several security properties, one of which is *collision resistance*. Being collision resistant, informally means that it is *hard* to find two distinct inputs which map to the same output value. In practice, the hash functions are mostly built from a fixed input size compression function, *e.g.* the renowned Merkle-Damgård construction. To any hash function, no matter how it has been designed, we can always attribute fixed input size compression functions, such that a collision for a derived compression function results in a direct collision for the hash function itself. This way, firstly we are working with fixed input size compression functions rather than varying input size ones, secondly we can attribute compression functions to those hash functions which are not explicitly based on a fixed input size compression function, and

* An extended version is available at `http://eprint.iacr.org/2009/382`

M. Matsui (Ed.): ASIACRYPT 2009, LNCS 5912, pp. 560–577, 2009.

thirdly we can derive different compression functions from a hash function. For example multi-block collision attack [27] benefits from the third point. Our task is to find two messages for an attributed compression function such that their digests are preferably equal (a collision) or differ in only a few bits (a near-collision).

The goal of this work is to *revisit* collision-finding methods using linearization of the compression function in order to find differential characteristics for the compression function. This method was initiated by Chabaud and Joux on SHA–0 [11] and was later extended and applied to SHA–1 by Rijmen and Oswald [26]. The recent attack on EnRUPT by Indesteege and Preneel [15] is another application of the method. In particular, in [26] it was observed that the codewords of a linear code, which are defined through a linearized version of the compression function, can be used to identify differential paths leading to a collision for the compression function itself. This method was later extended by Pramstaller et al. [25] with the general conclusion that finding high probability differential paths is related to low weight codewords of the attributed linear code. In this paper we further investigate this issue.

The first contribution of our work is to present a more concrete and tangible relation between the linearization and differential paths. In the case that modular addition is the only involved nonlinear operation, our results can be stated as follows. Given the parity check matrix $\mathcal{H}$ of a linear code, and two matrices $\mathcal{A}$ and $\mathcal{B}$, find a codeword Δ such that $\mathcal{A}\Delta \vee \mathcal{B}\Delta$ is of low weight. This is clearly different from the problem of finding a low weight codeword Δ. We then consider the problem of finding a conforming message pair for a given differential trail for a certain linear approximation of the compression function. We show that the problem of finding conforming pairs can be reformulated as finding preimages of zero under a function which we call the *condition* function. We then define the concept of *dependency table* which shows how much influence every input bit of the condition function has on each output bit. By carefully analyzing the dependency table, we are able to profit not only from neutral bits [7] but also from probabilistic neutral bits [2] in a backtracking search algorithm, similar to [6, 24, 14]. This contributes to a better understanding of freedom degrees uses.

We consider compression functions working with n-bit words. In particular, we focus on those using modular addition of n-bit words as the only nonlinear operation. The incorporated linear operations are XOR, shift and rotation of n-bit words in practice. We present our framework in detail for these constructions by approximating modular addition with XOR. We demonstrate its validity by applying it on reduced-round variants of CubeHash [4] (one of the NIST SHA-3 [22] competitors) which uses addition, XOR and rotation. CubeHash instances are parametrized by two parameters r and b and are denoted by `CubeHash-`r`/`b which process b message bytes per iteration; each iteration is composed of r rounds. Although we can not break the original submission `CubeHash-`8/1, we provide real collisions for the much weaker variants `CubeHash-`3/64 and `CubeHash-`4/48. Interestingly, we show that neither the more secure variants `CubeHash-`6/16 and `CubeHash-`7/64 do provide the desired collision security for 512-bit digests by providing theoretical attacks with complexities $2^{222.6}$ and $2^{203.0}$ respectively; nor that `CubeHash-`6/4 with 512-bit digests is second-preimage resistant, as with probability 2^{-478} a second preimage can be produced by only one hash evaluation. Our theory can be easily generalized to arbitrary nonlinear operations. We discuss

this issue and as an application we provide collision attacks on 16 rounds of MD6 [23]. MD6 is another SHA-3 candidate whose original number of rounds varies from 80 to 168 when the digest size ranges from 160 to 512 bits.

2 Linear Differential Cryptanalysis

Let's consider a compression function $H = \mathsf{Compress}(M, V)$ which works with n-bit words and maps an m-bit message M and a v-bit initial value V into an h-bit output H. Our aim is to find a collision for such compression functions with a randomly given initial value V. In this section we consider *modular-addition-based* Compress functions, that is, they use only modular additions in addition to linear transformations. This includes the family of AXR (Addition-XOR-Rotation) hash functions which are based on these three operations. In Section 5 we generalize our framework to other family of compression functions. For these Compress functions, we are looking for two messages with a difference Δ that result in a collision. In particular we are interested in a Δ for which two randomly chosen messages with this difference lead to a collision with a high probability for a randomly chosen initial value. For modular-addition-based Compress functions, we consider a linearized version for which all additions are replaced by XOR. This is a common linear approximation of addition. Other possible linear approximations of modular addition, which are less addressed in literature, can be considered according to our generalization of Section 5. As addition was the only nonlinear operation, we now have a linear function which we call $\mathsf{Compress}_{\mathrm{lin}}$. Since $\mathsf{Compress}_{\mathrm{lin}}(M, V) \oplus \mathsf{Compress}_{\mathrm{lin}}(M \oplus \Delta, V) = \mathsf{Compress}_{\mathrm{lin}}(\Delta, 0)$ is independent of the value of V, we adopt the notation $\mathsf{Compress}_{\mathrm{lin}}(M) = \mathsf{Compress}_{\mathrm{lin}}(M, 0)$ instead. Let Δ be an element of the kernel of the linearized compression function, *i.e.* $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = 0$. We are interested in the probability $\Pr\{\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = 0\}$ for a random M and V. In the following we present an algorithm which computes this probability, called the *raw* (or *bulk*) *probability*.

2.1 Computing the Raw Probability

We consider a general n-bit vector $x = (x_0, \ldots, x_{n-1})$ as an n-bit integer denoted by the same variable, *i.e.* $x = \sum_{i=0}^{n-1} x_i 2^i$. The Hamming weight of a binary vector or an integer x, $\mathrm{wt}(x)$, is the number of its nonzero elements, *i.e.* $\mathrm{wt}(x) = \sum_{i=0}^{n-1} x_i$. We use $+$ for modular addition of words and $\oplus, \vee$ and $\wedge$ for bit-wise XOR, OR and AND logical operations between words as well as vectors. We use the following lemma which is a special case of the problem of computing $\Pr\{\big((A \oplus \alpha) + (B \oplus \beta)\big) \oplus (A + B) = \gamma\}$ where α, β and γ are constants and A and B are independent and uniform random variables, all of them being n-bit words. Lipmaa and Moriai have presented an efficient algorithm for computing this probability [19]. We are interested in the case $\gamma = \alpha \oplus \beta$ for which the desired probability has a simple closed form.

Lemma 1. $\Pr\{\big((A \oplus \alpha) + (B \oplus \beta)\big) \oplus (A + B) = \alpha \oplus \beta\} = 2^{-\mathrm{wt}\left((\alpha \vee \beta) \wedge (2^{n-1} - 1)\right)}$.

Lemma 1 gives us the probability that modular addition behaves like the XOR operation. As $\mathsf{Compress}_{\mathrm{lin}}$ approximates Compress by replacing modular addition with

XOR, we can then devise a simple algorithm to compute (estimate) the raw probability $\Pr\{\mathsf{Compress}(M,V)\oplus\mathsf{Compress}(M\oplus\Delta,V)=\mathsf{Compress}_{\mathrm{lin}}(\Delta)\}$. Let's first introduce some notation.

Notation. Let n_{add} denote the number of additions which Compress uses in total. In the course of evaluation of $\mathsf{Compress}(M,V)$, let the two addends of the i-th addition $(1\le i\le n_{\mathrm{add}})$ be denoted by $A^i(M,V)$ and $B^i(M,V)$, for which the ordering is not important. The value $C^i(M,V)=\big(A^i(M,V)+B^i(M,V)\big)\oplus A^i(M,V)\oplus B^i(M,V)$ is then called the carry word of the i-th addition. Similarly, in the course of evaluation of $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$, denote the two inputs of the i-th linearized addition by $\alpha^i(\Delta)$ and $\beta^i(\Delta)$ in which the ordering is the same as that for A^i and B^i. We define five more functions $\mathbf{A}(M,V)$, $\mathbf{B}(M,V)$, $\mathbf{C}(M,V)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$ with $(n-1)n_{\mathrm{add}}$-bit outputs. These functions are defined as the concatenation of all the n_{add} relevant words excluding their MSBs. For example $\mathbf{A}(M,V)$ and $\boldsymbol{\alpha}(\Delta)$ are respectively the concatenation of the n_{add} words $\big(A^1(M,V),\ldots,A^{n_{\mathrm{add}}}(M,V)\big)$ and $\big(\alpha^1(\Delta),\ldots,\alpha^{n_{\mathrm{add}}}(\Delta)\big)$ excluding the MSBs.

Using this notation, the raw probability can be simply estimated as follows.

Lemma 2. *Let* Compress *be a modular-addition-based compression function. Then for any message difference Δ and for random values M and V, $p_\Delta=2^{-\mathrm{wt}\left(\boldsymbol{\alpha}(\Delta)\vee\boldsymbol{\beta}(\Delta)\right)}$ is a lower bound for* $\Pr\{\mathsf{Compress}(M,V)\oplus\mathsf{Compress}(M\oplus\Delta,V)=\mathsf{Compress}_{\mathrm{lin}}(\Delta)\}$.

Proof. We start with the following definition.

Definition 1. *We say that a message M (for a given V) conforms to (or follows) the trail of Δ iff*[1]

$$\big((A^i\oplus\alpha^i)+(B^i\oplus\beta^i)\big)\oplus(A^i+B^i)=\alpha^i\oplus\beta^i,\ \text{for } 1\le i\le n_{\mathrm{add}},\qquad(1)$$

where A^i, B^i, α^i and β^i are shortened forms for $A^i(M,V)$, $B^i(M,V)$, $\alpha^i(\Delta)$ and $\beta^i(\Delta)$, respectively.

It is not difficult to prove that under some reasonable independence assumptions p_Δ, which we call conforming probability, is the probability that a random message M follows the trail of Δ. This is a direct corollary of Lemma 1 and Definition 1 . The exact proof can be done by induction on n_{add}, the number of additions in the compression function. Due to other possible non-conforming pairs that start from message difference Δ and lead to output difference $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$, p_Δ is a lower bound for the desired probability in the lemma. □

If $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$ is of low Hamming weight, we get a near collision in the output. The interesting Δ's for collision search are those which belong to the kernel of $\mathsf{Compress}_{\mathrm{lin}}$, *i.e.* those that satisfy $\mathsf{Compress}_{\mathrm{lin}}(\Delta)=0$. From now on, we assume that $\Delta\neq 0$ is in the kernel of $\mathsf{Compress}_{\mathrm{lin}}$, hence looking for collisions. According to Lemma 2, one needs to try around $1/p_\Delta$ random message pairs in order to find a collision which conforms to the trail of Δ. However in a random search it is better not to restrict oneself

[1] If and only if.

to the conforming messages as a collision at the end is all we want. Since p_Δ is a lower bound for the probability of getting a collision for a message pair with difference Δ, we might get a collision sooner. In Section 3 we explain a method which might find a *conforming* message by avoiding random search.

2.2 Link with Coding Theory

We would like to conclude this section with a note on the relation between the following two problems: (I) finding low-weight codewords of a linear code, (II) finding a high probability linear differential path. Since the functions $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$ are linear, we consider Δ as a column vector and attribute three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ to these three transformations, respectively. In other words we have $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = \mathcal{H}\Delta$, $\boldsymbol{\alpha}(\Delta) = \mathcal{A}\Delta$ and $\boldsymbol{\beta}(\Delta) = \mathcal{B}\Delta$. We then call $\mathcal{H}$ the *parity check* matrix of the compression function.

Based on an initial work by Chabaud and Joux [11], the link between these two problems has been discussed by Rijmen and Oswald in [26] and by Pramstaller et al. in [25] with the general conclusion that finding highly probable differential paths is related to low weight codewords of the attributed linear code. In fact the relation between these two problems is more delicate. For problem (I), we are provided with the parity check matrix $\mathcal{H}$ of a linear code for which a codeword Δ satisfies the relation $\mathcal{H}\Delta = 0$. Then, we are supposed to find a low-weight nonzero codeword Δ. This problem is believed to be hard and there are some heuristic approaches for it, see [10] for example. For problem (II), however, we are given three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ and need to find a nonzero Δ such that $\mathcal{H}\Delta = 0$ and $\mathcal{A}\Delta \vee \mathcal{B}\Delta$ is of low-weight, see Lemma 2. Nevertheless, low-weight codewords Δ's matrix $\mathcal{H}$ might be good candidates for providing low-weight $\mathcal{A}\Delta \vee \mathcal{B}\Delta$, *i.e.* differential paths with high probability p_Δ. In particular, this approach is promising if these three matrices are sparse.

3 Finding a Conforming Message Pair Efficiently

The methods that are used to accelerate the finding of a message which satisfies some requirements are referred to as *freedom degrees use* in the literature. This includes message modifications [27], neutral bits [7], boomerang attacks [16, 20], tunnels [18] and submarine modifications [21]. In this section we show that the problem of finding conforming message pairs can be reformulated as finding preimages of zero under a function which we call the *condition* function. One can carefully analyze the condition function to see how freedom degrees might be used in efficient preimage reconstruction. Our method is based on measuring the amount of influence which every input bit has on each output bit of the condition function. We introduce the dependency tables to distinguish the influential bits, from those which have no influence or are less influential. In other words, in case the condition function does not mix its input bits well, we profit not only from neutral bits [7] but also from probabilistic neutral bits [2]. This is achieved by devising a backtracking search algorithm, similar to [6, 24, 14], based on the dependency table.

3.1 Condition Function

Let's assume that we have a differential path for the message difference Δ which holds with probability $p_\Delta = 2^{-y}$. According to Lemma 2 we have $y = \mathrm{wt}\big(\boldsymbol{\alpha}(\Delta) \vee \boldsymbol{\beta}(\Delta)\big)$. In this section we show that, given an initial value V, the problem of finding a conforming message pair such that $\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = 0$ can be translated into finding a message M such that $\mathsf{Condition}_\Delta(M, V) = 0$. Here $Y = \mathsf{Condition}_\Delta(M, V)$ is a function which maps m-bit message M and v-bit initial value V into y-bit output Y. In other words, the problem is reduced to finding a preimage of zero under the $\mathsf{Condition}_\Delta$ function. As we will see it is quite probable that not every output bit of the Condition function depends on all the message input bits. By taking a good strategy, this property enables us to find the preimages under this function more efficiently than random search. But of course, we are only interested in preimages of zero. In order to explain how we derive the function Condition from Compress we first present a quite easy-to-prove lemma. We recall that the *carry word* of two words A and B is defined as $C = (A + B) \oplus A \oplus B$.

Lemma 3. *Let A and B be two n-bit words and C represent their carry word. Let $\delta = 2^i$ for $0 \le i \le n - 2$. Then,*

$$\big((A \oplus \delta) + (B \oplus \delta)\big) = (A + B) \Leftrightarrow A_i \oplus B_i \oplus 1 = 0\,, \tag{2}$$

$$\big(A + (B \oplus \delta)\big) = (A + B) \oplus \delta \Leftrightarrow A_i \oplus C_i = 0\,, \tag{3}$$

and similarly

$$\big((A \oplus \delta) + B\big) = (A + B) \oplus \delta \Leftrightarrow B_i \oplus C_i = 0\,. \tag{4}$$

For a given difference Δ, a message M and an initial value V, let $\mathbf{A}_k$, $\mathbf{B}_k$, $\mathbf{C}_k$, $\boldsymbol{\alpha}_k$ and $\boldsymbol{\beta}_k$, $0 \le k < (n-1)n_{\mathrm{add}}$, respectively denote the k-th bit of the output vectors of the functions $\mathbf{A}(M, V)$, $\mathbf{B}(M, V)$, $\mathbf{C}(M, V)$, $\boldsymbol{\alpha}(\Delta)$ and $\boldsymbol{\beta}(\Delta)$, as defined in Section 2.1. Let $\{i_0, \ldots, i_{y-1}\}$, $0 \le i_0 < i_1 < \cdots < i_{y-1} < (n-1)n_{\mathrm{add}}$ be the positions of 1's in the vector $\boldsymbol{\alpha} \vee \boldsymbol{\beta}$. We define the function $Y = \mathsf{Condition}_\Delta(M, V)$ as:

$$Y_j = \begin{cases} \mathbf{A}_{i_j} \oplus \mathbf{B}_{i_j} \oplus 1 & \text{if } (\boldsymbol{\alpha}_{i_j}, \boldsymbol{\beta}_{i_j}) = (1,1), \\ \mathbf{A}_{i_j} \oplus \mathbf{C}_{i_j} & \text{if } (\boldsymbol{\alpha}_{i_j}, \boldsymbol{\beta}_{i_j}) = (0,1), \\ \mathbf{B}_{i_j} \oplus \mathbf{C}_{i_j} & \text{if } (\boldsymbol{\alpha}_{i_j}, \boldsymbol{\beta}_{i_j}) = (1,0), \end{cases} \tag{5}$$

for $j = 0, 1, \ldots, y-1$. This equation can be equivalently written as equation (7).

Proposition 1. *For a given V and Δ, a message M conforms to the trail of Δ iff* $\mathsf{Condition}_\Delta(M, V) = 0$.

3.2 Dependency Table for Freedom Degrees Use

For simplicity and generality, let's adopt the notation $F(M, V) = \mathsf{Condition}_\Delta(M, V)$ in this section. Assume that we are given a general function $Y = F(M, V)$ which maps m message bits and v initial value bits into y output bits. Our goal is to reconstruct preimages of a particular output, for example the zero vector, efficiently. More precisely,

we want to find V and M such that $F(M, V) = 0$. If F mixes its input bits very well, one needs to try about 2^y random inputs in order to find one mapping to zero. However, in some special cases, not every input bit of F affects every output bit. Consider an ideal situation where message bits and output bits can be divided into ℓ and $\ell + 1$ disjoint subsets respectively as $\bigcup_{i=1}^{\ell} \mathcal{M}_i$ and $\bigcup_{i=0}^{\ell} \mathcal{Y}_i$ such that the output bits $\mathcal{Y}_j$ $(0 \leq j \leq \ell)$ only depend on the input bits $\bigcup_{i=1}^{j} \mathcal{M}_i$ and the initial value V. In other words, once we know the initial value V, we can determine the output part $\mathcal{Y}_0$. If we know the initial value V and the input portion $\mathcal{M}_1$, the output part $\mathcal{Y}_1$ is then known and so on. Refer to Section 6 to see the partitioning of a condition function related to MD6. This property of F suggests Algorithm 1 for finding a preimage of zero. Algorithm 1 is a backtracking search algorithm in essence, similar to [6, 24, 14], and in practice is implemented recursively with a tree-based search to avoid memory requirements. The values $q_0, q_1, \ldots, q_\ell$ are the parameters of the algorithm to be determined later. To discuss the complexity of the algorithm, let $|\mathcal{M}_i|$ and $|\mathcal{Y}_i|$ denote the cardinality of $\mathcal{M}_i$ and $\mathcal{Y}_i$ respectively, where $|\mathcal{Y}_0| \geq 0$ and $|\mathcal{Y}_i| \geq 1$ for $1 \leq i \leq \ell$. We consider an *ideal behavior* of F for which each output part depends in a complex way on all the variables that it depends on. Thus, the output segment changes independently and uniformly at random if we change any part of the relevant input bits.

Algorithm 1. Preimage finding

Require: $q_0, q_1, \ldots, q_\ell$

Ensure: some preimage of zero under F

0: Choose 2^{q_0} initial values at random and keep those $2^{q'_1}$ candidates which make $\mathcal{Y}_0$ part null.

1: For each candidate, choose $2^{q_1 - q'_1}$ values for $\mathcal{M}_1$ and keep those $2^{q'_2}$ ones making $\mathcal{Y}_1$ null.

2: For each candidate, choose $2^{q_2 - q'_2}$ values for $\mathcal{M}_2$ and keep those $2^{q'_3}$ ones making $\mathcal{Y}_2$ null.

$\vdots$

i: For each candidate, choose $2^{q_i - q'_i}$ values for $\mathcal{M}_i$ and keep those $2^{q'_{i+1}}$ ones making $\mathcal{Y}_i$ null.

$\vdots$

ℓ: For each candidate, choose $2^{q_\ell - q'_\ell}$ values for $\mathcal{M}_\ell$ and keep those $2^{q'_{\ell+1}}$ final candidates making $\mathcal{Y}_\ell$ null.

To analyze the algorithm, we need to compute the optimal values for $q_0, \ldots, q_\ell$. The time complexity of the algorithm is $\sum_{i=0}^{\ell} 2^{q_i}$ as at each step 2^{q_i} values are examined. The algorithm is successful if we have at least one candidate left at the end, *i.e.* $q'_{\ell+1} \geq 0$. We have $q'_{i+1} \approx q_i - |\mathcal{Y}_i|$, coming from the fact that at the i-th step 2^{q_i} values are examined each of which makes the portion $\mathcal{Y}_i$ of the output null with probability $2^{-|\mathcal{Y}_i|}$. Note that we have the restrictions $q_i - q'_i \leq |\mathcal{M}_i|$ and $0 \leq q'_i$ since we have $|\mathcal{M}_i|$ bits of freedom degree at the i-th step and we require at least one surviving candidate after each step. Hence, the optimal values for q_i's can be recursively computed as $q_{i-1} = |\mathcal{Y}_{i-1}| + \max(0, q_i - |\mathcal{M}_i|)$ for $i = \ell, \ell - 1, \ldots, 1$ with $q_\ell = |\mathcal{Y}_\ell|$.

How can we determine the partitions $\mathcal{M}_i$ and $\mathcal{Y}_i$ for a given function F? We propose the following heuristic method for determining the message and output partitions in practice. We first construct a $y \times m$ binary valued table T called *dependency table*.

The entry $T_{i,j}$, $0 \leq i \leq m-1$ and $0 \leq j \leq y-1$, is set to one iff the j-th output bit is highly affected by the i-th message bit. To this end we empirically measure the probability that changing the i-th message bit changes the j-th output bit. The probability is computed over random initial values and messages. We then set $T_{i,j}$ to one iff this probability is greater than a threshold $0 \leq th < 0.5$, for example $th = 0.3$. We then call Algorithm 2.

Algorithm 2. Message and output partitioning

Require: Dependency table T
Ensure: ℓ, message partitions $\mathcal{M}_1, \ldots, \mathcal{M}_\ell$ and output partitions $\mathcal{Y}_0, \ldots, \mathcal{Y}_\ell$.
1: Put all the output bits j in $\mathcal{Y}_0$ for which the row j of T is all-zero.
2: Delete all the all-zero rows from T.
3: $\ell := 0$;
4: **while** T is not empty **do**
5: $\quad \ell := \ell + 1$;
6: $\quad$ **repeat**
7: $\quad\quad$ Determine the column i in T which has the highest number of 1's and delete it from T.
8: $\quad\quad$ Put the message bit which corresponds to the deleted column i into the set $\mathcal{M}_\ell$.
9: $\quad$ **until** There is at least one all-zero row in T OR T becomes empty
10: $\quad$ If T is empty set $\mathcal{Y}_\ell$ to those output bits which are not in $\bigcup_{i=0}^{\ell-1} \mathcal{Y}_i$ and stop.
11: $\quad$ Put all the output bits j in $\mathcal{Y}_\ell$ for which the corresponding row of T is all-zero.
12: $\quad$ Delete all the all-zero rows from T.
13: **end while**

In practice, once we make a partitioning for a given function using the above method, there are two issues which may cause the ideal behavior assumption to be violated:

1. The message segments $\mathcal{M}_1, \ldots, \mathcal{M}_i$ do not have full influence on $\mathcal{Y}_i$,
2. The message segments $\mathcal{M}_{i+1}, \ldots, \mathcal{M}_\ell$ have influence on $\mathcal{Y}_0, \ldots, \mathcal{Y}_i$.

With regard to the first issue, we ideally would like that all the message segments $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_i$ as well as the initial value V have full influence on the output part $\mathcal{Y}_i$. In practice the effect of the last few message segments $\mathcal{M}_{i-d_i}, \ldots, \mathcal{M}_i$ (for some small integer d_i) is more important, though. Theoretical analysis of deviation from this requirement may not be easy. However, with some tweaks on the tree-based (backtracking) search algorithm, we may overcome this effect in practice. For example if the message segment $\mathcal{M}_{i-1}$ does not have a great influence on the output segment $\mathcal{Y}_i$, we may decide to backtrack two steps at depth i, instead of one (the default value). The reason is as follows. Imagine that you are at depth i of the tree and you are trying to adjust the i-th message segment $\mathcal{M}_i$, to make the output segment $\mathcal{Y}_i$ null. If after trying about $2^{\min(|\mathcal{M}_i|,|\mathcal{Y}_i|)}$ choices for the i-th message block, you do not find an appropriate one, you will go one step backward and choose another choice for the $(i-1)$-st message segment $\mathcal{M}_{i-1}$; you will then go one step forward once you have successfully adjusted the $(i-1)$-st message segment. If $\mathcal{M}_{i-1}$ has no effect on $\mathcal{Y}_i$, this would be useless and increase our search cost at this node. Hence it would be appropriate if we backtrack two steps at this depth. In general, we may tweak our tree-based search by setting the number of steps which we want to backtrack at each depth.

In contrast, the theoretical analysis of the second issue is easy. Ideally, we would like that the message segments $\mathcal{M}_i, \ldots, \mathcal{M}_\ell$ have no influence on the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$. The smaller the threshold value th is chosen, the less the influence would be. Let 2^{-p_i}, $1 \leq i \leq \ell$, denote the probability that changing the message segment $\mathcal{M}_i$ does not change any bit from the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$. The probability is computed over random initial values and messages, and a random non-zero difference in the message segment $\mathcal{M}_i$. Algorithm 1 must be reanalyzed in order to recompute the optimal values for $q_0, \ldots, q_\ell$. Algorithm 1 also needs to be slightly changed by reassuring that at step i, all the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$ remain null. The time complexity of the algorithm is still $\sum_{i=0}^{\ell} 2^{q_i}$ and it is successful if at least one surviving candidate is left at the end, *i.e.* $q_{\ell+1} \geq 0$. However, here we set $q'_{i+1} \approx q_i - |\mathcal{Y}_i| - p_i$. This comes from the fact that at the i-th step 2^{q_i} values are examined each of which makes the portion $\mathcal{Y}_i$ of the output null with probability $2^{-|\mathcal{Y}_i|}$ and keeping the previously set output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_{i-1}$ null with probability 2^{-p_i} (we assume these two events are independent). Here, our restrictions are again $0 \leq q'_i$ and $q_i - q'_i \leq |\mathcal{M}_i|$. Hence, the optimal values for q_i's can be recursively computed as $q_{i-1} = p_{i-1} + |\mathcal{Y}_{i-1}| + \max(0, q_i - |\mathcal{M}_i|)$ for $i = \ell, \ell-1, \ldots, 1$ with $q_\ell = |\mathcal{Y}_\ell|$.

Remark 1. When working with functions with a huge number of input bits, it might be appropriate to consider the m-bit message M as a string of u-bit units instead of bits. For example one can take $u = 8$ and work with bytes. We then use the notation $M = (M[0], \ldots, M[m/u-1])$ (assuming u divides m) where $M[i] = (M_{iu}, \ldots, M_{iu+u-1})$. In this case the dependency table must be constructed according to the probability that changing every message unit changes each output bit.

4 Application to CubeHash

CubeHash [4] is Bernstein's proposal for the NIST SHA-3 competition [22]. CubeHash variants, denoted by `CubeHash-`*r*`/`*b*, are parametrized by r and b which at each iteration process b bytes in r rounds. Although `CubeHash-8/1` was the original official submission, later the designer proposed the tweak `CubeHash-16/32` which is almost 16 times faster than the initial proposal [5]. Nevertheless, the author has encouraged cryptanalysis of `CubeHash-`*r*`/`*b* variants for smaller r's and bigger b's.

4.1 CubeHash Description

CubeHash works with 32-bit words ($n = 32$) and uses three simple operations: XOR, rotation and modular addition. It has an internal state $S = (S_0, S_1, \ldots, S_{31})$ of 32 words and its variants, denoted by `CubeHash-`*r*`/`*b*, are identified by two parameters $r \in \{1, 2, \ldots\}$ and $b \in \{1, 2, \ldots, 128\}$. The internal state S is set to a specified value which depends on the digest length (limited to 512 bits) and parameters r and b. The message to be hashed is appropriately padded and divided into b-byte message blocks. At each iteration one message block is processed as follows. The 32-word internal state S is considered as a 128-byte value and the message block is XORed into the first b bytes of the internal state. Then, the following fixed permutation is applied r times to the internal state to prepare it for the next iteration.

1. Add S_i into $S_{i\oplus 16}$, for $0 \leq i \leq 15$.
2. Rotate S_i to the left by seven bits, for $0 \leq i \leq 15$.
3. Swap S_i and $S_{i\oplus 8}$, for $0 \leq i \leq 7$.
4. XOR $S_{i\oplus 16}$ into S_i, for $0 \leq i \leq 15$.
5. Swap S_i and $S_{i\oplus 2}$, for $i \in \{16, 17, 20, 21, 24, 25, 28, 29\}$.
6. Add S_i into $S_{i\oplus 16}$, for $0 \leq i \leq 15$.
7. Rotate S_i to the left by eleven bits, for $0 \leq i \leq 15$.
8. Swap S_i and $S_{i\oplus 4}$, for $i \in \{0, 1, 2, 3, 8, 9, 10, 11\}$.
9. XOR $S_{i\oplus 16}$ into S_i, for $0 \leq i \leq 15$.
10. Swap S_i and $S_{i\oplus 1}$, for $i \in \{16, 18, 20, 22, 24, 26, 28, 30\}$.

Having processed all message blocks, a fixed transformation is applied to the final internal state to extract the hash value as follows. First, the last state word S_{31} is ORed with integer 1 and then the above permutation is applied $10 \times r$ times to the resulting internal state. Finally, the internal state is truncated to produce the message digest of desired hash length. Refer to [4] for the full specification.

4.2 Definition of the Compression Function Compress

To be in the line of our general method, we need to deal with fixed-size input compression functions. To this end, we consider t ($t \geq 1$) consecutive iterations of CubeHash. We define the function $H = \mathsf{Compress}(M, V)$ with an $8bt$-bit message $M = M^0||\ldots||M^{t-1}$, a 1024-bit initial value V and a $(1024 - 8b)$-bit output H. The initial value V is used to initialize the 32-word internal state of CubeHash. Each M^i is a b-byte message block. We start from the initialized internal state and update it in t iterations. That is, in t iterations the t message blocks $M^0, \ldots, M^{t-1}$ are sequentially processed in order to transform the internal state into a final value. The output H is then the last $128 - b$ bytes of the final internal state value which is ready to absorb the $(t + 1)$-st message block (the 32-word internal state is interpreted as a 128-byte vector).

Our goal is to find collisions for this Compress function. In the next section we explain how collisions can be constructed for CubeHash itself.

4.3 Collision Construction

We are planning to construct collision pairs (M', M'') for `CubeHash-`r`/`b which are of the form $M' = M^{\mathrm{pre}}||M||M^t||M^{\mathrm{suf}}$ and $M'' = M^{\mathrm{pre}}||M \oplus \Delta||M^t \oplus \Delta^t||M^{\mathrm{suf}}$. Here, M^{pre} is the common prefix of the colliding pairs whose length in bytes is a multiple of b, M^t is one message block of b bytes and M^{suf} is the common suffix of the colliding pairs whose length is arbitrary. The message prefix M^{pre} is chosen for randomizing the initial value V. More precisely, V is the content of the internal state after processing the message prefix M^{pre}. For this value of V, $(M, M \oplus \Delta)$ is a collision pair for the compression function, *i.e.* $\mathsf{Compress}(M, V) = \mathsf{Compress}(M \oplus \Delta, V)$. Remember that a collision for the Compress indicates collision over the last $128-b$ bytes of the internal state. The message blocks M^t and $M^t \oplus \Delta^t$ are used to get rid of the difference in the first b bytes of the internal state. The difference Δ^t is called the *erasing block difference* and is computed as follows. When we evaluate the Compress with inputs (M, V) and $(M \oplus \Delta, V)$, Δ^t is the difference in the first b bytes of the final internal state values.

Once we find message prefix M^{pre}, message M and difference Δ, any message pairs (M', M'') of the above-mentioned form is a collision for CubeHash for *any* message block M^t and *any* message suffix M^{suf}. We find the difference Δ using the linearization method of Section 2 to applied to CubeHash in the next section. Then, M^{pre} and M are found by finding a preimage of zero under the Condition function as explained in Section 3. Algorithm 4 in the extended version of this article [9] shows how CubeHash Condition function can be implemented in practice for a given differential path.

4.4 Linear Differentials for CubeHash-r/b

As we explained in Section 2, the linear transformation $\mathsf{Compress}_{\text{lin}}$ can be identified by a matrix $\mathcal{H}_{h\times m}$. We are interested in Δ's such that $\mathcal{H}\Delta = 0$ and such that the differential trails have high probability. For CubeHash-r/b with t iterations, $\Delta = \Delta^0||\ldots||\Delta^{t-1}$ and $\mathcal{H}$ has size $(1024-8b)\times 8bt$, see Section 4.2. This matrix *suffers from having low rank*. This enables us to find low weight vectors of the kernel. We then hope that they are also good candidates for providing highly probable trails, see Section 2.2. Assume that this matrix has rank $(8bt-\tau)$, $\tau \geq 0$, signifying existence of $2^\tau - 1$ nonzero solutions to $\mathcal{H}\Delta = 0$. To find a low weight nonzero Δ, we use the following method.

The rank of $\mathcal{H}$ being $(8bt-\tau)$ shows that the solutions can be expressed by identifying τ variables as free and expressing the rest in terms of them. Any choice for the free variables uniquely determines the remaining $8bt-\tau$ variables, hence providing a unique member of the kernel. We choose a set of τ free variables at random. Then, we set one, two, or three of the τ free variables to bit value 1, and the other $\tau-1$, or $\tau-2$ or $\tau-3$ variables to bit value 0 with the hope to get a Δ providing a high probability differential path. We have made exhaustive search over all $\tau + \binom{\tau}{2} + \binom{\tau}{3}$ possible choices for all $b \in \{1,2,3,4,8,16,32,48,64\}$ and $r \in \{1,2,3,4,5,6,7,8\}$ in order to find the best characteristics. Table 1 includes the ordered pair (t, y), *i.e.* the corresponding number of iterations and the $-\log_2$ probability (number of bit conditions) of the best raw probability path we found. For most of the cases, the best characteristic belongs to the minimum value of t for which $\tau > 0$. There are a few exceptions to consider which are starred in Table 1. For example in the CubeHash-3/4 case, while for $t = 2$ we have $\tau = 4$ and $y = 675$, by increasing the number of iterations to $t = 4$, we get $\tau = 40$ and a better characteristic with $y = 478$. This may hold for other cases as well since we only increased t until our program terminated in a reasonable time. We would like to emphasize that since we are using linear differentials, the erasing block difference Δ^t only depends on the difference Δ, see Section 4.3.

Table 1. The values of (t, y) for the differential path with the best found raw probability

$r \setminus b$	1	2	3	4	8	12	16	32	48	64
1	$(14, 1225)$	$(8, 221)^\star$	$(4, 46)$	$(4, 32)$	$(4, 32)$	–	–	–	–	–
2	$(7, 1225)$	$(4, 221)^\star$	$(2, 46)$	$(2, 32)$	$(2, 32)$	–	–	–	–	–
3	$(16, 4238)^\star$	$(6, 1881)$	$(4, 798)$	$(4, 478)^\star$	$(4, 478)^\star$	$(4, 400)^\star$	$(4, 400)^\star$	$(4, 400)^\star$	$(3, 364)^\star$	$(2, 65)$
4	$(8, 2614)$	$(3, 964)$	$(2, 195)$	$(2, 189)$	$(2, 189)$	$(2, 156)$	$(2, 156)$	$(2, 156)$	$(2, 130)$	$(2, 130)$
5	$(18, 10221)^\star$	$(8, 4579)$	$(4, 2433)$	$(4, 1517)$	$(4, 1517)$	$(4, 1244)$	$(4, 1244)$	$(4, 1244)$	$(4, 1244)^\star$	$(2, 205)$
6	$(10, 4238)$	$(3, 1881)$	$(2, 798)$	$(2, 478)$	$(2, 478)$	$(2, 400)$	$(2, 400)$	$(2, 400)$	$(2, 351)$	$(2, 351)$
7	$(14, 13365)$	$(8, 5820)$	$(4, 3028)$	$(4, 2124)$	$(4, 2124)$	$(4, 1748)$	$(4, 1748)$	$(4, 1748)$	$(4, 1748)^\star$	$(2, 447)$
8	$(4, 2614)$	$(4, 2614)$	$(2, 1022)$	$(2, 1009)$	$(2, 1009)$	$(2, 830)$	$(2, 830)$	$(2, 830)$	$(2, 637)$	$(2, 637)$

Second preimage attacks on CubeHash. Any differential path with raw probability greater than 2^{-512} can be considered as a (theoretical) second preimage attack on CubeHash with 512-bit digest size. In Table 1 the entries which do not correspond to a successful second preimage attack, *i.e.* $y > 512$, are shown in gray, whereas the others have been highlighted. For example, our differential path for CubeHash-6/4 with raw probability 2^{-478} indicates that by only one hash evaluation we can produce a second preimage with probability 2^{-478}. Alternatively, it can be stated that for a fraction of 2^{-478} messages we can easily provide a second preimage. The list of differential trails for highlighted entries can be found in the extended version [9].

4.5 Collision Attacks on CubeHash Variants

Although Table 1 includes our best found differential paths with respect to raw probability or equivalently second preimage attack, when it comes to freedom degrees use for collision attack, these trails might not be the optimal ones. In other words, for a specific r and b, there might be another differential path which is worse in terms of raw probability but is better regarding the collision attack complexity if we use some freedom degrees speedup. As an example, for CubeHash-3/48 with the path which has raw probability 2^{-364}, using our method of Section 3 the time complexity can be reduced to about $2^{58.9}$ (partial) evaluation of its condition function. However, there is another path with raw probability 2^{-368} which has time complexity of about $2^{53.3}$ (partial) evaluation of its condition function. Table 2 shows the best paths we found regarding the reduced complexity of the collision attack using our method of Section 3. While most of the paths are still the optimal ones with respect to the raw probability, the starred entries indicate the ones which invalidate this property. Some of the interesting differential paths for starred entries in Table 2 are given in the extended version [9].

Table 3 shows the reduced time complexities of collision attack using our method of Section 3 for the differential paths of Table 2. To construct the dependency table, we have analyzed the Condition function at byte level, see Remark 1. The time complexities are in logarithm 2 basis and might be improved if the dependency table is analyzed at a bit level instead. The complexity unit is (partial) evaluation of their respective Condition function. We remind that the full evaluation of a Condition function corresponding to a t-iteration differential path is almost the same as application of t iterations (rt rounds) of CubeHash. We emphasize that the complexities are independent of digest size. All the complexities which are less than $2^{c/2}$ can be considered as a successful collision attack if the hash size is bigger than c bits. The complexities bigger than 2^{256} have been shown in gray as they are worse than birthday attack, considering 512-bit digest size. The successfully attacked instances have been highlighted.

The astute reader should realize that the complexities of Table 3 correspond to the optimal threshold value, see Section 3.2. Refer to the extended version [9] to see the effect of the threshold value on the complexity.

Practice versus theory. We provided a framework which is handy in order to analyze many hash functions in a generic way. In practice, the optimal threshold value may be a little different from the theoretical one. Moreover, by slightly playing with the neighboring bits in the suggested partitioning corresponding to a given threshold value

Table 2. The values of (t, y) for the differential path with the best found total complexity (Table 3 includes the reduced complexities using our method of Section 3)

$r \setminus b$	1	2	3	4	8	12	16	32	48	64
1	$(14, 1225)$	$(8, 221)$	$(4, 46)$	$(4, 32)$	$(4, 32)$	–	–	–	–	–
2	$(7, 1225)$	$(4, 221)$	$(2, 46)$	$(2, 32)$	$(2, 32)$	–	–	–	–	–
3	$(16, 4238)$	$(6, 1881)$	$(4, 798)$	$(4, 478)$	$(4, 478)$	$(4, 400)$	$(4, 400)$	$(4, 400)$	$(3, 368)^\star$	$(2, 65)$
4	$(8, 2614)$	$(3, 964)$	$(2, 195)$	$(2, 189)$	$(2, 189)$	$(2, 156)$	$(2, 156)$	$(2, 156)$	$(2, 134)^\star$	$(2, 134)^\star$
5	$(18, 10221)$	$(8, 4579)$	$(4, 2433)$	$(4, 1517)$	$(4, 1517)$	$(4, 1250)^\star$	$(4, 1250)^\star$	$(4, 1250)^\star$	$(4, 1250)^\star$	$(2, 205)$
6	$(10, 4238)$	$(3, 1881)$	$(2, 798)$	$(2, 478)$	$(2, 478)$	$(2, 400)$	$(2, 400)$	$(2, 400)$	$(2, 351)$	$(2, 351)$
7	$(14, 13365)$	$(8, 5820)$	$(4, 3028)$	$(4, 2124)$	$(4, 2124)$	$(4, 1748)$	$(4, 1748)$	$(4, 1748)$	$(4, 1748)$	$(2, 455)^\star$
8	$(4, 2614)$	$(4, 2614)$	$(2, 1022)$	$(2, 1009)$	$(2, 1009)$	$(2, 830)$	$(2, 830)$	$(2, 830)$	$(2, 655)^\star$	$(2, 655)^\star$

(Algorithm 2), we may achieve a partitioning which is more suitable for applying the attacks. In particular, Table 3 contains the theoretical complexities for different CubeHash instances under the assumption that the Condition function behaves ideally with respect to the first issue discussed in Section 3.2. In practice, deviation from this assumption increases the effective complexity. For particular instances, more simulations need to be done to analyze the potential non-randomness effects in order to give a more exact estimation of the practical complexity.

According to Section 4.3, for a given linear difference Δ, we need to find message prefix M^{pre} and conforming message M for collision construction. Our backtracking (tree-based) search implementation of Algorithm 1 for CubeHash-3/64 finds M^{pre} and M in 2^{21} (median complexity) instead of the $2^{9.4}$ of Table 3. The median decreases to 2^{17} by backtracking three steps at each depth instead of one, see Section 3.2. For CubeHash-4/48 we achieve the median complexity $2^{30.4}$ which is very close to the theoretical value $2^{30.7}$ of Table 3. Collision examples for CubeHash-3/64 and CubeHash-4/48 can be found in the extended paper [9]. Our detailed analysis of CubeHash variants shows that the practical complexities for all of them except 3-round CubeHash are very close to the theoretical values of Table 3. We expect the practical complexities for CubeHash instances with three rounds to be slightly bigger than the given theoretical numbers. For detailed comments we refer to the extended paper [9].

Comparison with the previous results. The first analysis of CubeHash was proposed by Aumasson et al. [3] in which the authors showed some non-random properties for several versions of CubeHash. A series of collision attacks on CubeHash-1/b and CubeHash-2/b for large values of b were announced by Aumasson [1] and Dai [12].

Table 3. Theoretical $\log_2$ complexities of improved collision attacks with freedom degrees use at byte level for the differential paths of Table 2

$r \setminus b$	1	2	3	4	8	12	16	32	48	64
1	1121.0	135.1	24.0	15.0	7.6	–	–	–	–	–
2	1177.0	179.1	27.0	17.0	7.9	–	–	–	–	–
3	4214.0	1793.0	720.0	380.1	292.6	153.5	102.0	55.6	53.3	9.4
4	2598.0	924.0	163.0	138.4	105.3	67.5	60.7	54.7	30.7	28.8
5	10085.0	4460.0	2345.0	1397.0	1286.0	946.0	868.0	588.2	425.0	71.7
6	4230.0	1841.0	760.6	422.1	374.4	260.4	222.6	182.1	147.7	144.0
7	13261.0	5709.0	2940.0	2004.0	1892.0	1423.0	1323.0	978.0	706.0	203.0
8	2606.0	2590.0	982.0	953.0	889.0	699.0	662.0	524.3	313.0	304.4

Collision attacks were later investigated deeply by Brier and Peyrin [8]. Our results improve on all existing ones as well as attacking some untouched variants.

5 Generalization

In sections 2 and 3 we considered modular-addition-based compression functions which use only modular additions and linear transformations. Moreover, we concentrated on XOR approximation of modular additions in order to linearize the compression function. This method is however quite general and can be applied to a broad class of hash constructions, covering many of the existing hash functions. Additionally, it lets us consider other linear approximations as well. We view a compression function $H = \mathsf{Compress}(M, V) : \{0,1\}^m \times \{0,1\}^v \rightarrow \{0,1\}^h$ as a binary finite state machine (FSM). The FSM has an internal state which is consecutively updated using message M and initial value V. We assume that FSM operates as follows, and we refer to such Compress functions as *binary-FSM-based*. The concept can also cover non-binary fields.

The internal state is initially set to zero. Afterwards, the internal state is sequentially updated in a limited number of steps. The output value H is then derived by truncating the final value of the internal state to the specified output size. At each step, the internal state is updated according to one of these *two* possibilities: either the whole internal state is updated as an affine transformation of the current internal state, M and V, or *only one* bit of the internal state is updated as a *nonlinear* Boolean function of the current internal state, M and V. Without loss of generality, we assume that all of the nonlinear updating Boolean functions (NUBF) have zero constant term (*i.e.* the output of zero vector is zero) and none of the involved variables appear as a pure linear term (*i.e.* changing any input variable does not change the output bit with certainty). This assumption, coming from the simple observation that we can integrate constants and linear terms in an affine updating transformation (AUT), is essential for our analysis. Linear approximations of the FSM can be achieved by replacing AUTs with linear transformations by ignoring the constant terms and NUBFs with linear functions of their arguments. Similar to Section 2 this gives us a linearized version of the compression function which we denote by $\mathsf{Compress}_{\mathrm{lin}}(M, V)$. As we are dealing with differential cryptanalysis, we take the notation $\mathsf{Compress}_{\mathrm{lin}}(M) = \mathsf{Compress}_{\mathrm{lin}}(M, 0)$. The argument given in Section 2 is still valid: elements of the kernel of the linearized compression function (*i.e.* Δ's *s.t.* $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = 0$) can be used to construct differential trails.

Let n_{nl} denote the total number of NUBFs in the FSM. We count the NUBFs by starting from zero. We introduce four functions $\Lambda(M, V)$, $\Phi(\Delta)$, $\Lambda^{\Delta}(M, V)$ and $\Gamma(\Delta)$ all of output size n_{nl} bits. To define these functions, consider the two procedures which implement the FSMs of $\mathsf{Compress}(M, V)$ and $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$. Let the Boolean function g^k, $0 \leq k < n_{\mathrm{nl}}$, stand for the k-th NUBF and denote its linear approximation as in $\mathsf{Compress}_{\mathrm{lin}}$ by g^k_{lin}. Moreover, denote the input arguments of the Boolean functions g^k and g^k_{lin} in the FSMs which compute $\mathsf{Compress}(M, V)$ and $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$ by the vectors x^k and δ^k, respectively. Note that δ^k is a function of Δ whereas x^k depends on M and V. The k-th bit of $\Gamma(\Delta)$, $\Gamma_k(\Delta)$, is set to one iff the argument of the k-th linearized NUBF is not the all-zero vector, *i.e.* $\Gamma_k(\Delta) = 1$ iff $\delta^k \neq 0$. We then define $\Lambda_k(M, V) = g^k(x^k)$, $\Phi_k(\Delta) = g^k_{lin}(\delta^k)$ and $\Lambda^{\Delta}_k(M, V) = g^k(x^k \oplus \delta^k)$. We can then present the following proposition. The proof is given in the full version paper [9].

Proposition 2. *Let* Compress *be a binary-FSM-based compression function. For any message difference Δ, let $\{i_0, \ldots, i_{y-1}\}$, $0 \leq i_0 < i_1 < \cdots < i_{y-1} < n_{\mathrm{nl}}$ be the positions of 1's in the vector $\Gamma(\Delta)$ where $y = \mathrm{wt}\big(\Gamma(\Delta)\big)$. We define the condition function $Y = \mathsf{Condition}_\Delta(M, V)$ where the j-th bit of Y is computed as*

$$Y_j = \Lambda_{i_j}(M, V) \oplus \Lambda_{i_j}^{\Delta}(M, V) \oplus \Phi_{i_j}(\Delta). \tag{6}$$

Then, if Δ is in the kernel of $\mathsf{Compress}_{\mathrm{lin}}$, $\mathsf{Condition}_\Delta(M, V) = 0$ *implies that the pair $(M, M \oplus \Delta)$ is a collision for* Compress *with the initial value V.*

Remark 2. The modular-addition-based compression functions can be implemented as binary-FSM-based compression by considering one bit memory for the carry bit. All the NUBFs for this FSM are of the form $g(x, y, z) = xy \oplus xz \oplus yz$. The XOR approximation of modular addition in Section 2 corresponds to approximating all the NUBFs g by the zero function, *i.e.* $g_{\mathrm{lin}}(x, y, z) = 0$. It is straightforward to show that $\Lambda_k(M, V) = g(\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_k)$ and $\Phi_k(\Delta) = g_{\mathrm{lin}}(\boldsymbol{\alpha}_k, \boldsymbol{\beta}_k, 0)$. We then deduce that $\Gamma_k(\Delta) = \boldsymbol{\alpha}_k \vee \boldsymbol{\beta}_k \vee 0$ and $\Lambda_k^{\Delta}(M, V) = g(\mathbf{A}_k \oplus \boldsymbol{\alpha}_k, \mathbf{B}_k \oplus \boldsymbol{\beta}_k, \mathbf{C}_k \oplus 0)$. As a result we get

$$\begin{aligned} Y_j &= \Lambda_{i_j}(M, V) \oplus \Lambda_{i_j}^{\Delta}(M, V) \oplus \Phi_{i_j}(\Delta) \\ &= (\boldsymbol{\alpha}_{i_j} \oplus \boldsymbol{\beta}_{i_j})\mathbf{C}_{i_j} \oplus \boldsymbol{\alpha}_{i_j} B_{i_j} \oplus \boldsymbol{\beta}_{i_j} A_{i_j} \oplus \boldsymbol{\alpha}_{i_j}\boldsymbol{\beta}_{i_j} \end{aligned} \tag{7}$$

whenever $\boldsymbol{\alpha}_{i_j} \vee \boldsymbol{\beta}_{i_j} = 1$; this agrees with equation (5). Refer to the extended version [9] for more details and to see how other linear approximations could be used.

6 Application to MD6

MD6 [23], designed by Rivest et al., is a SHA-3 candidate that provides security proofs regarding some differential attacks. The core part of MD6 is the function f which works with 64-bit words and maps 89 input words $(A_0, \ldots, A_{88})$ into 16 output words $(A_{16r+73}, \ldots, A_{16r+88})$ for some integer r representing the number of rounds. Each round is composed of 16 steps. The function f is computed based on the following recursion

$$A_{i+89} = L_{r_i,l_i}\big(S_i \oplus A_i \oplus (A_{i+71} \wedge A_{i+68}) \oplus (A_{i+58} \wedge A_{i+22}) \oplus A_{i+72}\big), \tag{8}$$

where S_i's are some publicly known constants and L_{r_i,l_i}'s are some known simple linear transformations. The 89-word input of f is of the form $Q||U||W||K||B$ where Q is a known 15-word constant value, U is a one-word node ID, W is a one-word control word, K is an 8-word key and B is a 64-word data block. For more details about function f and the mode of operation of MD6, we refer to the submission document [23][2]. We consider the compression function $H = \mathsf{Compress}(M, V) = f(Q||U||W||K||B)$ where $V = U||W||K$, $M = B$ and H is the 16-word compressed value. Our goal is to find a collision $\mathsf{Compress}(M, V) = \mathsf{Compress}(M', V)$ for arbitrary value of V. We later explain how such collisions can be translated into collisions for the MD6 hash function.

According to our model (Section 5), MD6 can be implemented as an FSM which has $64 \times 16r$ NUBFs of the form $g(x, y, z, w) = x \cdot y \oplus z \cdot w$. Remember that

[2] In the MD6 document [23], C and L_{r_i,l_i} are respectively denoted by V and g_{r_i,l_i}.

the NUBFs must not include any linear part or constant term. We focus on the case where we approximate all NUBFs with the zero function. This corresponds to ignoring the AND operations in equation (8). This essentially says that in order to compute $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = \mathsf{Compress}_{\mathrm{lin}}(\Delta, 0)$ for a 64-word $\Delta = (\Delta_0, \ldots, \Delta_{63})$, we map $(A'_0, \ldots, A'_{24}, A'_{25}, \ldots, A'_{88}) = 0||\Delta = (0, \ldots, 0, \Delta_0, \ldots, \Delta_{63})$ into the 16 output words $(A'_{16r+73}, \ldots, A'_{16r+88})$ according to the linear recursion

$$A'_{i+89} = L_{r_i,l_i}\big(A'_i \oplus A'_{i+72}\big). \tag{9}$$

For a given Δ, the function Γ is the concatenation of $16r$ words $A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}$, $0 \le i \le 16r-1$. Therefore, the number of bit conditions equals

$$y = \sum_{i=0}^{16r-1} \mathrm{wt}(A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}). \tag{10}$$

Note that this equation compactly integrates cases 1 and 2 given in section 6.9.3.2 of [23] for counting the number of active AND gates. Algorithm 3 in the extended version of this article [9] shows how the Condition function is implemented using equations (6), (8) and(9).

Using a similar linear algebraic method to the one used in Section 4.4 for CubeHash, we have found the collision difference of equation (11) for $r = 16$ rounds with a raw probability $p_\Delta = 2^{-90}$. In other words, Δ is in the kernel of $\mathsf{Compress}_{\mathrm{lin}}$ and the condition function has $y = 90$ output bits. Note that this does not contradict the proven bound in [23]: one gets at least 26 active AND gates.

$$\Delta_i = \begin{cases} \texttt{F6D164597089C40E} & i = 2 \\ \texttt{2000000000000000} & i = 36 \\ 0 & 0 \le i \le 63, i \ne 2, 36 \end{cases} \tag{11}$$

In order to efficiently find a conforming message pair for this differential path we need to analyze the dependency table of its condition function. Referring to our notations in Section 3.2, our analysis of the dependency table of function $\mathsf{Condition}_\Delta(M, 0)$ at word level (units of $u = 64$ bits) shows that the partitioning of the condition function is as in Table 4 for threshold value $th = 0$. For this threshold value clearly $p_i = 0$. The optimal values for q_i's (computed according to the complexity analysis of the same section) are also given in Table 4, showing a total attack complexity of $2^{30.6}$ (partial) condition function evaluation[3]. By analyzing the dependency table with smaller units the complexity may be subject to reduction.

A collision example for $r = 16$ rounds of f can be found in the full version [9]. Our 16-round colliding pair provides near collisions for $r = \mathbf{17}$, $\mathbf{18}$ and $\mathbf{19}$ rounds, respectively, with $\mathbf{63}$, $\mathbf{144}$ and $\mathbf{270}$ bit differences over the 1024-bit long output of f. Refer to [9] to see how collisions for reduced-round f can be turned into collisions for reduced-round MD6 hash function. The original MD6 submission [23] mentions inversion of the function f up to a dozen rounds using SAT solvers. Some slight nonrandom behavior of the function f up to 33 rounds has also been reported [17].

[3] By masking M_{38} and M_{55} respectively with `092E9BA68F763BF1` and `DFFBFF7FEFFDFFBF` after random setting, the 35 condition bits of the first three steps are satisfied for free, reducing the complexity to $2^{30.0}$ instead.

Table 4. Input and output partitionings of the Condition function of MD6 with $r = 16$ rounds

i	$\mathcal{M}_i$	$\mathcal{Y}_i$	q_i	q'_i
0	–	$\emptyset$	0	0
1	$\{M_{38}\}$	$\{Y_1, \ldots, Y_{29}\}$	29	0
2	$\{M_{55}\}$	$\{Y_{43}, \ldots, Y_{48}\}$	6	0
3	$\{M_0, M_5, M_{46}, M_{52}, M_{54}\}$	$\{Y_0\}$	1	0
4	$\{M_j \mid j = 3, 4, 6, 9, 21, 36, 39, 40, 42, 45, 49, 50, 53, 56, 57\}$	$\{Y_{31}, \ldots, Y_{36}\}$	6	0
5	$\{M_{41}, M_{51}, M_{58}, M_{59}, M_{60}\}$	$\{Y_{30}, Y_{51}\}$	2	0
6	$\{M_j \mid j = 1, 2, 7, 8, 10, 11, 12, 17, 18, 20, 22, 24, 25, 26, 29, 33, 34, 37, 43, 44, 47, 48, 61, 62, 63\}$	$\{Y_{52}, \ldots, Y_{57}\}$	6	0
7	$\{M_{27}\}$	$\{Y_{37}, \ldots, Y_{42}\}$	6	0
8	$\{M_{13}, M_{16}, M_{23}\}$	$\{Y_{50}\}$	1	0
9	$\{M_{35}\}$	$\{Y_{49}\}$	1	0
10	$\{M_{14}, M_{15}, M_{19}, M_{28}\}$	$\{Y_{58}, Y_{61}\}$	2	0
11	$\{M_{30}, M_{31}, M_{32}\}$	$\{Y_{59}, Y_{60}, Y_{62} \ldots, Y_{89}\}$	30	0

7 Conclusion

We presented a framework for an in-depth study of linear differential attacks on hash functions. We applied our method to reduced round variants of CubeHash and MD6, giving by far the best known collision attacks on these SHA-3 candidates. Our results may be improved by considering start-in-the middle attacks if the attacker is allowed to choose the initial value of the internal state.

Acknowledgment. The second author has been supported in part by European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II, and the third author by GEBERT RÜF STIFTUNG, project no. GRS-069/07. The authors would like to thank the reviewers of Asiacrypt 2009, Deian Stefan, Martijn Stam, Jean-Philippe Aumasson and Dag Arne Osvik for their helpful comments.

References

1. Aumasson, J.-P.: Collision for CubeHash-2/120 − 512. NIST mailing list, December 4 (2008), `http://ehash.iaik.tugraz.at/uploads/a/a9/Cubehash.txt`
2. Aumasson, J.-P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 470–488. Springer, Heidelberg (2008)
3. Aumasson, J.-P., Meier, W., Naya-Plasencia, M., Peyrin, T.: Inside the hypercube. In: Boyd, C., González Nieto, J. (eds.) ACISP 2009. LNCS, vol. 5594, pp. 202–213. Springer, Heidelberg (2009)
4. Bernstein, D.J.: CubeHash specification (2.b.1). Submission to NIST SHA-3 competition
5. Bernstein, D.J.: CubeHash parameter tweak: 16 times faster, `http://cubehash.cr.yp.to/submission/`
6. 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 2006)
7. Biham, E., Chen, R.: Near-Collisions of SHA-0. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 290–305. Springer, Heidelberg (2004)
8. Brier, E., Peyrin, T.: Cryptanalysis of CubeHash. Applied Cryptography and Network Security. In: Abdalla, M., Pointcheval, D., Fouque, P.-A., Vergnaud, D. (eds.) ACNS 2009. LNCS, vol. 5536, pp. 354–368. Springer, Heidelberg (2009)

9. Brier, E., Khazaei, S., Meier, W., Peyrin, T.: Linearization Framework for Collision Attacks: Application to CubeHash and MD6 (Extended Version). In Cryptology ePrint Archive, Report 2009/382, http://eprint.iacr.org/2009/382
10. Canteaut, A., Chabaud, F.: A new algorithm for finding minimum-weight words in a linear code: application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. IEEE Transactions on Information Theory 44(1), 367–378 (1998)
11. Chabaud, F., Joux, A.: Differential Collisions in SHA-0. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 56–71. Springer, Heidelberg (1998)
12. Dai, W.: Collisions for CubeHash-1/45 and CubeHash-2/89 (2008), http://www.cryptopp.com/sha3/cubehash.pdf
13. eBASH: ECRYPT Benchmarking of All Submitted Hashes, http://bench.cr.yp.to/ebash.html
14. Fuhr, T., Peyrin, T.: Cryptanalysis of Radiogatun. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 122–138. Springer, Heidelberg (2009)
15. Indesteege, S., Preneel, B.: Practical collisions for EnRUPT. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 246–259. Springer, Heidelberg (2009)
16. Joux, A., Peyrin, T.: Hash Functions and the (Amplified) Boomerang Attack. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 244–263. Springer, Heidelberg (2007)
17. Khovratovich, D.: Nonrandomness of the 33-round MD6. Presented at the rump session of FSE 2009 (2009), Slides: http://fse2009rump.cr.yp.to/
18. Klima, V.: Tunnels in Hash Functions: MD5 Collisions Within a Minute. ePrint archive (2006), http://eprint.iacr.org/2006/105.pdf
19. Lipmaa, H., Moriai, S.: Efficient Algorithms for Computing Differential Properties of Addition. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 336–350. Springer, Heidelberg (2002)
20. Manuel, S., Peyrin, T.: Collisions on SHA-0 in One Hour. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 16–35. Springer, Heidelberg (2008)
21. Naito, Y., Sasaki, Y., Shimoyama, T., Yajima, J., Kunihiro, N., Ohta, K.: Improved Collision Search for SHA-0. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 21–36. Springer, Heidelberg (2006)
22. National Institute of Science and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. Federal Register, 72(112) (November 2007)
23. Rivest, R.L., Agre, B., Bailey, D.V., Crutchfield, C., Dodis, Y., Fleming, K.E., Khan, A., Krishnamurthy, J., Lin, Y., Reyzin, L., Shen, E., Sukha, J., Sutherland, D., Tromer, E., Yin, Y.L.: The MD6 hash function — a proposal to NIST for SHA–3. Submission to NIST SHA-3 competition (2008)
24. Peyrin, T.: Cryptanalysis of Grindahl. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 551–567. Springer, Heidelberg (2007)
25. Pramstaller, N., Rechberger, C., Rijmen, V.: Exploiting Coding Theory for Collision Attacks on SHA–1. In: Smart, N.P. (ed.) Cryptography and Coding 2005. LNCS, vol. 3796, pp. 78–95. Springer, Heidelberg (2005)
26. Rijmen, V., Oswald, E.: Update on SHA–1. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 58–71. Springer, Heidelberg (2005)
27. 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)