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 for the condition function. Then, the dependency table concept shows how much influence every input bit of the condition function has on its output bits. Careful analysis of the dependency table reveals degrees of freedom that can be exploited in accelerated preimage reconstruction of 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.
Key words: Hash functions, collisions, differential attack, SHA-3, CubeHash and MD6.

1 Introduction

Hash functions are important cryptographic primitives that find applications in many areas like digital signatures and commitment schemes. A hash function is a transformation which maps a variable-length input to a fixed-size output. 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. We can always attribute a compression function with fixed-size inputs to any hash function for which a collision for the compression function results in a direct collision for the hash function. However, in practice, it is the other way round and the hash functions are based on fixed input size compression functions, *e.g.* the renowned Merkle-Damgård construction. Our task is to find two messages for the attributed compression function such that their outputs are preferably equal (a collision) or differ in a few bits (near-collision). Collisions for a compression function are then directly translated to collisions for the hash function. In contrary, the relevance of near collisions depends on the hash function structure. In most of the cases they provide near collisions for the underlying hash function, but in some cases, like in sponge constructions with a strong filtering at the end or a Merkle-Damgård construction with a strong final transformation, they are of little interest.

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 [8] and was later extended and applied to SHA–1 by Rijmen and Oswald [23]. The recent attack on EnRUPT by Indesteege and Preneel [12] is another application of the method. In particular, in [23] 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. [22] 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 investigate this issue further.

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 $\mathcal{H}$, the parity check matrix 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

⋆ Supported in part by European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II.
⋆⋆ Supported by GEBERT RÜF STIFTUNG, project no. GRS-069/07.

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 output of a function which we call *condition* function. We then define the concept of *dependency table* which shows how much influence every input bit of the condition function has on its output bits. By carefully analyzing the dependency table, we are able to find strategies which accelerate preimage reconstruction for the condition function. This contributes to a better understanding of freedom degrees uses, such as message modifications [24], neutral bits [5], boomerang attacks [13, 17], tunnels [15] or submarine modifications [18].

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 [3] (one of the NIST SHA-3 [19] 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 made 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 hash outputs by providing theoretical attacks with complexities $2^{222.6}$ and $2^{203.0}$ respectively; nor that `CubeHash-6/4` with 512-bit hash outputs 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 [20], another SHA-3 candidate whose original number of rounds varies from 80 to 168 when the output hash size ranges from 160 to 512 bits.

2 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 (near-)collision for such compression functions with a randomly given initial value V. Let us first consider collision attacks, and for the moment *assume that* Compress *uses only modular additions and linear transformations*. This includes the family of AXR (Addition-XOR-Rotation) hash functions which are based on these three operations. We are looking for two messages with a difference Δ that lead to a collision for the compression function. 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. We consider a linearized version of Compress for which all additions are replaced by XOR. This is a common linear approximation of addition. We discuss in Section 5.2 other possible linear approximations of modular addition which are less addressed in literature. As addition was the only nonlinear operation, we now have a linear function which we call $\mathsf{Compress}_{\mathrm{lin}}$. Since $\mathsf{Compress}_{\mathrm{lin}}(M, V) \oplus \mathsf{Compress}_{\mathrm{lin}}(M \oplus \Delta, V)$ 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 *raw* (or *bulk*) *probability*.

2.1 Computing 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 [16]. We are interested in the case $\gamma = \alpha \oplus \beta$ for which the desired probability has a simple closed form.

Lemma 1. (from [16]) $\Pr\{\big((A \oplus \alpha) + (B \oplus \beta)\big) \oplus (A + B) = \alpha \oplus \beta\} = 2^{-\mathrm{wt}\left((\alpha \vee \beta) \wedge (2^{n-1} - 1)\right)}$.

Lemma 1 suggests a simple algorithm to compute (estimate) the raw probability $\Pr\{\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = 0\}$. Let's first introduce some notations.

Notations. 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 ($1 \leq i \leq n_{\mathrm{add}}$) 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 by $\alpha^i(\Delta)$ and $\beta^i(\Delta)$ the two inputs of the i-th linearized addition in which the ordering is the same as that for A^i and B^i. We define five more functions $\mathsf{A}(M, V)$, $\mathsf{B}(M, V)$, $\mathsf{C}(M, V)$, $\mathsf{alpha}(\Delta)$ and $\mathsf{beta}(\Delta)$ with $(n-1)n_{\mathrm{add}}$-bit outputs. These functions are defined as the concatenation of all the n_{add} relevant words excluding their MSB's. For example $\mathsf{A}(M, V)$ and $\mathsf{alpha}(\Delta)$ 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 MSB's. As the functions $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$, $\mathsf{alpha}(\Delta)$ and $\mathsf{beta}(\Delta)$ are linear, we consider Δ as an m-bit column vector and attribute three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ to these three transformations, respectively. The matrix $\mathcal{H}$ has size $h \times m$ and the matrices $\mathcal{A}$ and $\mathcal{B}$ have size $(n-1)n_{\mathrm{add}} \times m$ and satisfy these equations:

$$\mathsf{Compress}_{\mathrm{lin}}(\Delta) = \mathcal{H}\Delta, \mathsf{alpha}(\Delta) = \mathcal{A}\Delta \text{ and } \mathsf{beta}(\Delta) = \mathcal{B}\Delta.$$

Using these notations, the raw probability can be simply estimated as follows.

Lemma 2. *For a compression function* Compress *(which only uses modular addition and linear transformation), let three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ be defined as above. Then for any message difference Δ and for random values M and V, $p_\Delta = 2^{-\mathrm{wt}\left(\mathcal{A}\Delta \vee \mathcal{B}\Delta\right)}$ is a lower bound for* $\Pr\{\mathsf{Compress}(M, V) \oplus \mathsf{Compress}(M \oplus \Delta, V) = \mathcal{H}\Delta\}$.

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*[4]

$$\big((A^i \oplus \alpha^i) + (B^i \oplus \beta^i)\big) \oplus (A^i + B^i) = \alpha^i \oplus \beta^i, \tag{1}$$

for $1 \leq i \leq n_{\mathrm{add}}$, 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 which start from difference Δ and lead to final difference $\mathcal{H}\Delta$, p_Δ is a lower bound for the desired probability in the lemma. □

If $\mathcal{H}\Delta = \mathsf{Compress}_{\mathrm{lin}}(\Delta)$ 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 $\mathcal{H}$, *i.e.* satisfy $\mathcal{H}\Delta = 0$. From now on, we assume that Δ is in the kernel of $\mathcal{H}$, hence looking for collisions. We then call $\mathcal{H}$ the *parity check* matrix of the compression function. 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 to the conforming messages as a collision at the end is all we want. As 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 finding low-weigh codewords of a linear code and finding a high probability linear differential path. Based on an initial work by Chabaud and Joux [8], the problem has been discussed by Rijmen and Oswald in [23] and by Pramstaller et al. in [22] with the general conclusion that finding high probable differential paths is related to low weight codewords of the attributed linear code. In fact the link between these two problems is more delicate. For the earlier, 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 low-weight Δ's. This problem is believed to be hard and there are some heuristic approaches for it, see [7] for example. For the later problem, we are given three matrices $\mathcal{H}$, $\mathcal{A}$ and $\mathcal{B}$ and we are supposed to find Δ's such that $\mathcal{H}\Delta = 0$ an $\mathcal{A}\Delta \vee \mathcal{B}\Delta$ is of low-weight, see Lemma 2. Nevertheless, low-weight codewords Δ's of the parity check 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.

[4] if and only if.

3 Finding a conforming message pair efficiently

In this section we show that the problem of finding conforming message pairs can be reformulated as finding preimages of the zero output of a function which we call condition function. It turns out that the condition function is a useful tool to understand previous attacks on hash functions. Especially, one can analyze the condition function to see how to reconstruct its preimages of zero output efficiently. We then introduce the dependency tables and propose a heuristic algorithm to produce preimages of zero output in case the condition function does not mix well its input bits. Our algorithm can be seen as yet another kind of freedom degrees use but again it contributes to a better understanding of hash functions and freedom degrees speedup methods.

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 $y = \mathrm{wt}\big(\mathsf{alpha}(\Delta) \vee \mathsf{beta}(\Delta)\big)$. 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 find a preimage of zero output for 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 of this function more efficiently than random search. But of course, we are only interested in preimages of zero output. In order to explain how we derive the function Condition from Compress we first present a quite easy-to-prove lemma. We remind 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 $A = \mathsf{A}(M, V)$, $B = \mathsf{B}(M, V)$, $C = \mathsf{C}(M, V)$, $\alpha = \mathsf{alpha}(\Delta)$ and $\beta = \mathsf{beta}(\Delta)$ where A, B, C, α and β are all $(n-1)n_{\mathrm{add}}$-bit vectors. Refer to the notations presented in Section 2.1 for the definitions of these five functions. 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 $\alpha \vee \beta$. We define the function $Y = \mathsf{Condition}_\Delta(M, V)$ as:

$$Y_j = \begin{cases} A_{i_j} \oplus B_{i_j} \oplus 1 \; if \; (\alpha_{i_j}, \beta_{i_j}) = (1,1), \\ A_{i_j} \oplus C_{i_j} \quad if \; (\alpha_{i_j}, \beta_{i_j}) = (0,1) \\ B_{i_j} \oplus C_{i_j} \quad if \; (\alpha_{i_j}, \beta_{i_j}) = (1,0) \end{cases}. \tag{5}$$

for $j = 0, 1, \ldots, y-1$.

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

Proof. Proof is straightforward from Definition 1, Lemma 3 and the definition of Condition function in equation (5).

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 output. However, in some special cases, not every input bit of F effects every output bit. Consider an ideal situation where message bits and output bits can be divided in 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 $\bigcup_{i=1}^{j-1} \mathcal{M}_i$ 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 part $\mathcal{M}_0$, the output portion $\mathcal{Y}_1$ 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 output. Algorithm 1 is a backtracking process in essence, similar to [4, 11, 21], 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 portion depends in a complex way on all the variables which 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 output for F

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

1: For each initial value, choose $2^{q_1 - q'_1}$ values for $\mathcal{M}_1$ and keep $2^{q'_2}$ candidates making $\mathcal{Y}_1$ part 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.

$\vdots$

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 $q'_{i+1} \approx q_i - |\mathcal{Y}_i|$. This comes from the fact that at each step 2^{q_i} values are examined each of which making 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 node 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 $i = \ell + 1, \ell, \ell - 1, \ldots, 1$ with $q_{\ell+1} = 0$.

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 ones 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 behaviour assumption to be violated:

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

With regard to the first issue, we ideally would like that all the message segments $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_i$ 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. The bigger the threshold value th is chosen, the more it helps to increase the effect of the message segments on the relevant output segment. Theoretical analysis of deviation from this requirement may not be easy. However, with some tweaks on the tree-based 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 which is 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 i-th 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 go backward at each depth.

In contrast, the theoretical analysis of the second issue is easy. Ideally, we would like that the message segments $\mathcal{M}_{i+1}, \ldots, \mathcal{M}_\ell$ have no influence on the output segments $\mathcal{Y}_0, \ldots, \mathcal{Y}_i$. The smaller the threshold value th is chosen, the less the influence would be. Let's 2^{-p_i}, $1 \leq i \leq \ell$, denote the probability that changing the message segment $\mathcal{M}_i$ changes at least one 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 have $q'_{i+1} \approx q_i - |\mathcal{Y}_i| - p_i$. This comes from the fact that at each step 2^{q_i} values are examined each of which making 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 $i = \ell+1, \ell, \ell-1, \ldots, 1$ with $q_{\ell+1} = 0$.

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 effect of each message unit on each output bit.

4 Application to CubeHash

CubeHash [3] is Bernstein's proposal for the NIST SHA-3 competition [19]. Although `CubeHash-8/1` is the official submission which at each iteration processes one message byte in 8 rounds, the author has encouraged cryptanalysis of `CubeHash-`*r*`/`*b* variants which at each iteration processes b bytes in r rounds for smaller r's and bigger b's. We refer to [3] for a full specification, but a description is given in Appendix A.

4.1 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 consecutive iterations which start from an internal state which is set to a random initial value V (the randomization of internal state can be done by prepending a few random message blocks). The t message blocks $M^0, \ldots, M^{t-1}$ are processed to make the internal state ready for absorbing the $(t+1)$-st message block. We define the function $H = \mathsf{Compress}(M, V)$ with an $8bt$-bit message $M = M^0 || \ldots || M^{t-1}$

and a 1024-bit initial value V. The output H is the last $(1024 - 8b)$ bits of the final internal state that is ready to absorb the $(t+1)$-st message block. Our goal is to find a collision for this compression function. Once we have a collision, the $(t+1)$-st message block is chosen to erase the differences in the first $8b$ bits of the final internal state. In other words we are looking for messages which collide in the whole internal state. Near-collisions are out of interest as they do neither (directly) help to find a collision for the hash function nor show a weakness. In the next section we explain how high probability differential paths can be found for linearized Compress function. In Appendix G, we present Algorithm 4 which shows how CubeHash Condition function can be implemented in practice for a given differential path.

4.2 Differentials for CubeHash-r/b

As we explained in Section 2, the linear transformation $\mathsf{Compress}_{\mathrm{lin}}$ 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 4.1. This matrix *suffers from having low rank*. This enables us to find good low weight vectors of the kernel which are luckily 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 terminates in a reasonable time.

$r \setminus b$	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	$(4, 2614)$	$(4, 2614)$	$(2, 1022)$	$(2, 1009)$	$(2, 1009)$	$(2, 830)$	$(2, 830)$	$(2, 830)$	$(2, 637)$	$(2, 637)$

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

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 output hash 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 is given in Appendix B.

4.3 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, the time complexity can be reduced to about $2^{58.9}$ (partial) evaluation of its condition function for the path with raw probability 2^{-364}. 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 final complexity of the collision attack. Yet, most of the paths are the optimal ones with respect to the raw probability as well. 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 Appendix C.

$r \setminus b$	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	$(4, 2614)$	$(4, 2614)$	$(2, 1022)$	$(2, 1009)$	$(2, 1009)$	$(2, 830)$	$(2, 830)$	$(2, 830)$	$(2, 655)^\star$	$(2, 655)^\star$

Table 2. The values of (t, y) for the differential path with the best found total complexity.

$r \setminus b$	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

Table 3. Theoretical $\log_2$ complexities of improved collision attacks with freedom degrees use at byte level.

CubeHash instance	$y \setminus th$	0.0	0.025	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.475
3/64	65	16.3	11.3	11.4	10.6	10.2	10.4	9.9	**9.4**	10.4	10.4	11.4	16.5
4/48	134	50.8	38.3	34.6	33.6	32.3	38.4	35.0	32.2	**30.7**	32.3	37.7	38.6
4/32	156	75.2	67.4	67.4	63.7	69.0	62.0	54.7	**53.8**	60.7	66.0	63.9	70.3
3/48	368	99.7	90.6	87.2	80.3	67.6	61.2	65.3	**53.3**	65.1	83.4	102.6	119.3
3/32	400	91.0	63.3	61.5	57.1	61.8	**55.6**	70.9	70.7	79.7	82.4	98.9	113.6

Table 4. Theoretical $\log_2$ complexities of improved attacks with freedom degrees use at byte level versus threshold value.

Table 3 shows the improved time complexities of collision attack using dependency table and freedom degrees use at byte level for the differential paths of Table 2 for the optimal threshold value, see Section 3.2.

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 output hash 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 output hash size. The successfully attacked instances have been highlighted. To see the effect of the threshold value th on the complexity, we focus on five instances of CubeHash: CubeHash-3/64,-4/48,-3/48,-3/32 and-4/32. The first two instances are the ones for which the theoretical complexities are practically reachable and we have managed to find their real collision examples. The other three instances are the ones whose theoretical complexities are just above the practically reachable values and are most probably the ones for which real collisions will be found in near future either using more advanced methods or utilizing a huge cluster of computers. Table 4 shows the effect of threshold value on the complexity for these six instances.

Real collisions for CubeHash-3/64 and -4/48. We use the 2-iteration message difference $\Delta = \Delta^0||\Delta^1$ of relation (12) given in Appendix B for CubeHash-3/64 and of relation (13) given in Appendix C for CubeHash-4/48. Remember that the difference Δ^2 in a third iteration is used to erase the difference in the internal state caused by the differences Δ^0 and Δ^1 in the previous two iterations. What we need to do is to find three message blocks M^{-1}, M^0 and M^1 such that the corresponding $Y = \mathsf{Condition}_{\Delta}(M, V)$ function ($M = M^0||M^1$) produces a zero output vector. The message block M^{-1} is chosen for randomizing the internal state V and will be the common first block of the colliding message pairs, *i.e.* V is the content of the internal state after processing the message block M^{-1}. Once we find M^{-1}, M^0 and M^1 message blocks, any message pairs (M, M') where $M = M^{-1}||M^0||M^1||M^2||M''$ and $M' = M^{-1}||(M^0 \oplus \Delta^0)||(M^1 \oplus \Delta^1)||(M^2 \oplus \Delta^2)||M''$ collide for any message block M^2 and any message suffix M'' of arbitrary length. Appendix D includes the values of M^{-1}, M^0 and M^1 for collision on CubeHash-3/64 with 512-bit output hash value, whereas Appendix E provides the corresponding values for collision constructing on CubeHash-4/48 with 512-bit output hash values. Note that a collision pair/attack for a given r and b is also a collision pair/attack for the same r and bigger b's.

Practice versus theory. We provided a framework which is handy in order to analyze many hash functions in a generic way. In practice, the practical optimal threshold value may be a little different from the theoretical one. Moreover, by slightly playing with the neighbouring bits in the suggested partitioning corresponding to a given threshold value we may achieve a partitioning which is more suitable for applying the attacks. In particular, Tables 3 and 4 contain 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 makes the practical complexity increase. For particular instances, more simulations need to be done to analyse the potential non-randomness effects in order to give a more exact estimation of the practical complexity. In the following we compare the practical complexities with the theoretical values for some cases which their complexities are practically reachable. Moreover, for some CubeHash instances which their complexities are practically unreachable we try to give a more precise estimation of their practical complexities.

Our tree-based search implementation for the CubeHash-3/64 case with $th = 0.3$ has median complexity 2^{21} instead of the $2^{9.4}$ of Table 4. The median decreases to 2^{17} by backtracking three steps at each depth instead of one, see section 3.2. We expect the practical complexities for other instances of CubeHash with three rounds be slightly bigger than the theoretical numbers in Table 3. These cases need to be more investigated.

Our detailed analysis of CubeHash-4/32, CubeHash-4/48 and CubeHash-4/64 show that these cases perfectly match with theory. According to Table 3, for CubeHash-4/64 (with $th = 0.33$) and CubeHash-4/48 (with $th = 0.30$) we have the theoretical complexities $2^{28.8}$ and $2^{30.7}$, respectively. We experimentally achieve median complexities $2^{28.3}$ and $2^{30.4}$ respectively. For CubeHash-4/32 (with $th = 0.25$) the theoretical complexity is $2^{54.7}$. In the tree-based search algorithm, we need to satisfy 44 bit conditions at step 18, *i.e.* $|\mathcal{Y}_{18}| = 44$. This is the node which has the highest cost and if it is successfully passed, the remaining steps will easily be followed. Our simulations show that on average we need about 2^{10} (partial) evaluations of the condition function per one surviving candidate which arrives at depth 18. Hence, our estimation of the practical complexity is $2^{10} \times 2^{44} = 2^{54}$ which agrees with theory.

In CubeHash-5/64 case (with $th = 0.24$), the costliest node is at depth 20 for which 70 bit conditions must be satisfied, *i.e.* $|\mathcal{Y}_{20}| = 70$. Only one surviving candidates from this node suffices to make the remaining condition bits null with little cost. Our simulation shows that on average about $2^{7.0}$ (partial) evaluations of the condition function is required per one surviving candidate which arrives at depth 20. Hence, our estimation of the practical complexity is about $2^{7.0+70} = 2^{77.0}$, versus theoretical value $2^{71.7}$.

In CubeHash-6/16 case (with $th = 0.15$), the costliest node is at depth 9 for which 198 bit conditions must be satisfied, *i.e.* $|\mathcal{Y}_9| = 198$. We need 2^{26} candidates successfully pass this node, *i.e.* $q'_{10} = 26$, see Algorithm 1. Our simulation shows that on average about 2^5 (partial) evaluations of the condition function is required per one surviving candidate which arrives at depth 9. Hence, our estimation of the practical complexity is about $2^{26+5+198} = 2^{229}$, versus theoretical value $2^{222.6}$.

In CubeHash-7/64 case (with $th = 0.255$), the costliest node is at depth 24 for which 201 bit conditions must be satisfied, *i.e.* $|\mathcal{Y}_{24}| = 201$. Only one surviving candidates from this node suffices to make the remaining condition bits null with much less cost. Our simulation shows that on average about 2^7 (partial) evaluations of the condition function is required per one surviving candidate which arrives at depth 24. Hence, our estimation of the practical complexity is about $2^{7+201} = 2^{208}$, versus theoretical value $2^{203.0}$.

We emphasis that for these later cases we did not attempt to play with the neighbouring bits in the partitioning. We believe, in general, complexities can get very close to the theoretical ones if one tries to do so.

Comparison with the previous results. The first analysis of CubeHash was proposed by Aumasson et al. [2] 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 [9]. Collision attacks were later investigated in deep by Brier and Peyrin [6]. Our results improve on all existing ones as well as attacking some untouched variants.

5 Generalization

In Sections 2 and 3 we considered compression functions using 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 quite general and can be applied to a broad class of hash constructions which covers a lot of existing hash functions. Besides, it lets us consider other linear approximations as well. We view a compression function $H = \mathsf{Compress}(M, V) : \{0,1\}^m \times \{0,1\}^v \to \{0,1\}^h$ as a binary finite state machine (FSM) [5]. The FSM has an internal state which is consecutively updated using message M and initial value V. We assume that FSM operates as follows.

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). As we will see, 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 AUT's with linear transformations by ignoring the constant terms and NUBF's 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 NUBF's in the FSM. We introduce three functions $\Lambda(M, V)$, $\Gamma(\Delta)$ and $\Phi(\Delta)$ of output size n_{nl} bits. Let $\Lambda(M, V)$ include the output value of all the n_{nl} NUBF in the course of evaluation of $\mathsf{Compress}(M, V)$ through the execution of FSM. Similarly in the

[5] This model covers even more practical hash constructions. One might also consider the FSM over extensions of binary fields. For example quite a few of SHA-3 candidates are based on AES components, that is, they can be modeled with FSM's over GF(2^8). Here we only deal with the binary field.

course of evaluation of $\mathsf{Compress}_{\mathrm{lin}}(\Delta)$ through linearized version of FSM, let $\Phi(\Delta)$ include the output value of all the n_{nl} linearized NUBF's. The function $\Gamma(\Delta)$ is an analogy to the function $\mathsf{alpha}(\Delta) \vee \mathsf{beta}(\Delta)$ which appears in Lemma 2. The i-th bit of $\Gamma(\Delta)$, $\Gamma_i(\Delta)$, is set to zero iff all the inputs of the i-th linearized NUBF are zero. The role of Γ can be explained as follows. Assume that a message pair with difference Δ follows the differential trail until just before applying the i-th NUBF. If $\Gamma_i(\Delta) = 0$ it means that the path will be satisfied until before applying the $(i+1)$-st NUBF with probability one as the input arguments of i-th NUBF are the same for both messages. However, $\Gamma_i(\Delta) = 1$ indicates that the input arguments of the i-th NUBF differ for the two messages and the differential path would not be surely followed after applying this NUBF. It should now be clear why we need the NUBF's to be free of linear terms. The reason is that if the difference is only in linear terms, applying the i-th NUBF still allows the path to follow.

In case $\Gamma_i(\Delta) = 1$, all we require is to ensure that the differential path is followed after applying the i-th NUBF. This is exactly the way we construct the condition function. To make it more formal, let's introduce another function $\Lambda^{\Delta}(M, V)$ of n_{nl} output bits. The i-th bit of Λ^{Δ} simulates the output of the i-th NUBF for the message $M \oplus \Delta$ assuming that the message pair $(M, M \oplus \Delta)$ conforms to the path until before applying the i-th nonlinear update function. In other words, $\Lambda_i^{\Delta}(M, V)$ is built by applying the i-th NUBF on the difference of the arguments of the functions Λ_i and Φ_i. Clearly, the path is followed after applying the i-th NUBF iff the difference between $\Lambda_i(M, V)$ and $\Lambda_i^{\Delta}(M, V)$ is equal to the output of linearized NUBF, *i.e.* $\Phi_i(\Delta)$. To summarize, similar to the statement in the proof of Lemma 2, under some independence assumptions we can state that $2^{-\mathrm{wt}\left(\Gamma(\Delta)\right)}$ is the probability that a random message pair with difference Δ for a random initial value conforms to the differential trail. The generalization of the condition function in Section 3 is then straightforward to validate Proposition 1 in this case as well. Let $y = \mathrm{wt}\big(\Gamma(\Delta)\big)$ where $\{i_0, \ldots i_{y-1}\}$, $0 \le i_0 < i_1 < \cdots < i_{y-1} < n_{\mathrm{nl}}$, are the positions of 1's in the vector $\Gamma(\Delta)$. The condition function $Y = \mathsf{Condition}_{\Delta}(M, V)$ which maps m-bit message M and v-bit initial value V into y-bit output Y is constructed according to the following relation

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

for $j = 0, 1, \ldots, y-1$.

5.1 Modular addition case

Let's review the compression functions involving only linear transformations and modular addition of n-bit words. We deeply studied this subject in Sections 2 and 3. The modular addition $Z = X + Y$ can be computed by considering one bit memory c for the carry bit. Let $X = (x_0, \ldots, x_{n-1})$, $Y = (y_0, \ldots, y_{n-1})$ and $Z = (z_0, \ldots, z_{n-1})$. We have

$$\begin{aligned} c_{i+1} &= c_i x_i \oplus c_i y_i \oplus x_i y_i \quad \text{for } 0 \le i \le n-2 \\ z_i &= x_i \oplus y_i \oplus c_i \qquad \text{for } 0 \le i \le n-1, \end{aligned}$$

where $c_0 = 0$. It can be argued that a compression function which uses only linear transformations and n_{add} modular additions can be implemented as a FSM. With regard to the notations in Section 5, the FSM has $n_{\mathrm{nl}} = (n-1)n_{\mathrm{add}}$ NUBF's, all of the form $g(x, y, z) = xy \oplus xz \oplus yz$. Remember the notation $A_i = \mathsf{A}_i(M, V)$, $B_i = \mathsf{B}_i(M, V)$, $C_i = \mathsf{C}_i(M, V)$, $\alpha_i = \mathsf{alpha}_i(\Delta)$ and $\beta_i = \mathsf{beta}_i(\Delta)$; see notations in Section 2.1 and the note after Lemma 3). As the input arguments of the i-th NUBF are (A_i, B_i, C_i) we have $\Lambda_i(M, V) = g(A_i, B_i, C_i)$. The XOR approximation of modular additions corresponds to approximating g with the zero function. Therefore $\Phi(\Delta) = 0$ and moreover, the input arguments of the i-th linearized NUBF are $(\alpha_i, \beta_i, 0)$ and hence $\Lambda_i^{\Delta}(M, V) = g(A_i \oplus \alpha_i, B_i \oplus \beta_i, C_i)$. Note that the input arguments of the i-th linearized NUBF, $(\alpha_i, \beta_i, 0)$, are not all zero iff $\alpha_i \vee \beta_i = 1$. That is $\Gamma_i(\Delta) = \alpha_i \vee \beta_i$ which is in agreement with Lemma 2. According to the general equation (6) it follows that for $(\alpha_{i_j}, \beta_{i_j}) \neq (0, 0)$

$$\begin{aligned} Y_j &= \Lambda_{i_j}(M, V) \oplus \Lambda_{i_j}^{\Delta}(M, V) \oplus \Phi_{i_j}(\Delta) \\ &= g(A_{i_j}, B_{i_j}, C_{i_j}) \oplus g(A_{i_j} \oplus \alpha_{i_j}, B_{i_j} \oplus \beta_{i_j}, C_{i_j}) \\ &= (\alpha_{i_j} \oplus \beta_{i_j}) C_{i_j} \oplus \alpha_{i_j} B_{i_j} \oplus \beta_{i_j} A_{i_j} \oplus \alpha_{i_j} \beta_{i_j} \end{aligned} \tag{7}$$

which agrees with equation (5).

5.2 Note on the different linear approximations

Different combinations of different linear approximations of the NUBF's provide different linear approximations of the compression function. However, one should be careful to avoid approximations which might lead

to contradictions due to dependency between different approximations. In fact the probability $2^{-\Gamma(\Delta)}$ would not be a good estimate in this case if there are strong correlations between approximations. In the case of linear approximation of modular addition of n-bit words we have $(n-1)$ NUBF's for the carry bits, out of which $n-2$ are of the form $xy \oplus xc \oplus yc$ and one of the form xy (corresponding to the carry of the LSB). This shows the possibility of $4 \times 8^{(n-2)}$ different linear approximations. For one particular linear approximation, if the difference of the two addends are α and β the output difference γ is uniquely determined. In [16] the notation of "good" differential is introduced to distinguish those differentials which can happen with non-zero probability. A differential $\alpha, \beta \rightarrow \gamma$ is not "good" iff for some $i \in [0, n-1]$, $\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1} \neq \alpha_i \oplus \beta_i \oplus \gamma_i$ [16]. The exact probability of "good" differentials can be computed from Algorithm 2 of [16]. In general, it might not be easy to take redundancies into account. However, a cryptanalyst should try to do her best. We also would like to emphasize that although there exists an exponential number of linear approximations (in terms of n_{nl}) for the compression function, it would be better in practice to concentrate on those for which highly probable linear differential paths are found easily. For example, by approximating the NUBF's with the zero function or sparse linear functions, the $h \times m$ matrix $\mathcal{H}$ which satisfies $\mathsf{Compress}_{\mathrm{lin}}(\Delta) = \mathcal{H}\Delta$ is more likely sparser, making it easier to find differential paths with good raw probability.

6 Application to MD6

MD6 [20], 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 $(A_{16r}, \ldots, A_{16r+88})$ for some integer r representing the number of rounds. Each round is composed of 16 steps. The function f is computed according to the following recursion

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

where S_i's are some publicly known constants and g_{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[6], 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 [20]. 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 in Section 5, MD6 can be implemented as an FSM which has $64 \times 16r$ NUBF's of the form $g(x, y, z, w) = x \cdot y \oplus z \cdot w$. Remember that the NUBF's must not include any linear part or constant term. We focus on the case where we approximate all NUBF's 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 $(A'_0, \ldots, A'_{88}) = 0||\Delta = (0, \ldots, 0, \Delta_0, \ldots, \Delta_{63})$ into the 16 output words $(A'_{16r}, \ldots, A'_{16r+88})$ according to the linear recursion

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

For a given Δ, the function Γ is the concatenation of $16r$ words $A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}$, $0 \leq i \leq 16r-1$. 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 [20] for counting the number of active AND gates. For pedagogical reasons, in Appendix G we present Algorithm 3 which shows how equation (6) is implemented in practice in order to compute the condition function.

Using a similar linear algebraic method to the one used in Section 4.2 for CubeHash, we have found the following collision difference 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 [20]: one gets at least 26 active AND gates.

[6] In the MD6 document, the control word is denoted by V.

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

Table 5. Dependency table for Condition function of MD6 with $r = 16$ rounds.

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

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 5 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 5, showing a total attack complexity of $2^{30.6}$ (partial) condition function evaluation[7]. By analyzing the dependency table with smaller units the complexity may be subject to reduction.

Having set V to zero (which corresponds to choosing null values for the key, the node ID and the control word in order to simplify things), we found a message M, given in the Appendix F, which makes the condition function null. In other words, the message pairs M and $M \oplus \Delta$ are colliding pairs for $r = 16$ rounds of f. This 16-round colliding pair provides near collisions for $r = \mathbf{17}, \mathbf{18}$ and $\mathbf{19}$ rounds with $\mathbf{63}, \mathbf{144}$ and $\mathbf{270}$ bit differences respectively over the 1024-bit long output of f.

Now, let's discuss the potential of providing collisions for full MD6. The MD6 mode of operation is optionally parametrized by an integer L, $0 \leq L \leq 64$, which allows a smooth transition from the default tree-based hierarchical mode of operation (for $L = 64$) down to an iterative mode of operation (for $L = 0$). When $L = 0$, MD6 works in a manner similar to that of the well-known Merkle-Damgård construction (or the HAIFA method). Since in the iterative Merkle-Damgård the first 16 words of the message block are used as a chaining value, and as our difference in equation (11) is non-zero in the first 16 words, we do not get a collision but a pseudo-collision. Nevertheless, for 16-round MD6 in the tree-based hierarchical mode of operation (*i.e.* for $1 \leq L \leq 64$) we get a hash collision. We emphasize that one must choose node ID U and control word W properly in order to fulfill the MD6 restriction on these values as opposed to the null values which we chose. This is the first real collision example for 16-round MD6.

The original MD6 submission [20] mentions inversion of the function f up to a dozen rounds using SAT solvers. Some slight nonrandom behaviour of the function f up to 33 rounds has also been reported [14].

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.

[7] 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

References

1. J-P. Aumasson. Collision for CubeHash-2/120 − 512. NIST mailing list, 4 Dec 2008, 2008. http://ehash.iaik.tugraz.at/uploads/a/a9/Cubehash.txt.
2. J-P. Aumasson, W. Meier, M. Naya-Plasencia and T. Peyrin. Inside the hypercube. In C. Boyd and J. González Nieto, editors, *Australasian Conference on Information Security and Privacy – ACISP 2009*, volume 5594 of *Lecture Notes in Computer Science*, pages 202–213. Springer-Verlag, 2009.
3. D.J. Bernstein. CubeHash specification (2.b.1). Submission to NIST SHA-3 competition, 2008.
4. G. Bertoni, J. Daemen, M. Peeters and G. Van Assche. Radiogatun, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara (August 24-25, 2006). See http://radiogatun.noekeon.org/.
5. E. Biham and R. Chen. Near-Collisions of SHA-0. In M.K. Franklin, editor, *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 290–305. Springer-Verlag, 2004.
6. E. Brier and T. Peyrin. Cryptanalysis of CubeHash. In D. Pointcheval and M. Abdalla, editors, *Applied Cryptography and Network Security – ACNS 2009*, volume 5536 of *Lecture Notes in Computer Science*, pages 354-ï£¡368. Springer-Verlag, 2009.
7. A. Canteaut and F. Chabaud. 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. In *IEEE Transactions on Information Theory, 44(1):367–378, january 1998.*
8. F. Chabaud and A. Joux. Differential Collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.
9. W. Dai. Collisions for CubeHash-1/45 and CubeHash-2/89. Available online, 2008. http://www.cryptopp.com/sha3/cubehash.pdf.
10. eBASH: ECRYPT Benchmarking of All Submitted Hashes. http://bench.cr.yp.to/ebash.html
11. T. Fuhr and T. Peyrin. Cryptanalysis of Radiogatun. In O. Dunkelman, editor, *Fast Software Encryption – FSE 2009*, *Lecture Notes in Computer Science*. Springer-Verlag, 2009.
12. S. Indesteege and B. Preneel. Practical collisions for EnRUPT. In O. Dunkelman, editor, *Fast Software Encryption – FSE 2009*, *Lecture Notes in Computer Science*. Springer-Verlag, 2009.
13. A. Joux and T. Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In A. Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, volume 4622 of *Lecture Notes in Computer Science*, pages 244–263. Springer-Verlag, 2004.
14. D. Khovratovich. Nonrandomness of the 33-round MD6. Presented at the rump session of FSE'09, 2009. Slides are available online at http://fse2009rump.cr.yp.to/.
15. V. Klima. Tunnels in Hash Functions: MD5 Collisions Within a Minute. ePrint archive, 2006 http://eprint.iacr.org/2006/105.pdf.
16. H. Lipmaa and S. Moriai. Efficient Algorithms for Computing Differential Properties of Addition. In M. Matsui, editor, *Fast Software Encryption – FSE 2001*, volume 2355 of *Lecture Notes in Computer Science*, pages 336–350. Springer-Verlag, 2001.
17. S. Manuel and T. Peyrin. Collisions on SHA-0 in One Hour. In K. Nyberg, editor, *Fast Software Encryption – FSE 2008*, volume 5086 of *Lecture Notes in Computer Science*, pages 16–35. Springer-Verlag, 2008.
18. Y. Naito, Y. Sasaki, T. Shimoyama, J. Yajima, N. Kunihiro and K. Ohta. Improved Collision Search for SHA-0. In X. Lai and K. Chen, editors, *Advances in Cryptology – ASIACRYPT'06*, volume 4284 of *Lecture Notes in Computer Science*, pages 21–36. Springer-Verlag, 2006.
19. 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.*
20. R.L. Rivest, B. Agre, D.V. Bailey, C. Crutchfield, Y. Dodis, K.E. Fleming, A. Khan, J. Krishnamurthy, Y. Lin, L. Reyzin, E. Shen, J. Sukha, D. Sutherland, E. Tromer and Y.L. Yin. The MD6 hash function — a proposal to NIST for SHA–3. Submission to NIST SHA-3 competition, 2008.
21. T. Peyrin. Cryptanalysis of Grindahl. In Kaoru Kurosawa, editor, *Advances in Cryptology – ASIACRYPT'07*, volume 4833 of *Lecture Notes in Computer Science*, pages 551–567. Springer, 2007.
22. N. Pramstaller, C. Rechberger and V. Rijmen. Exploiting Coding Theory for Collision Attacks on SHA–1. In *Cryptography and Coding, 10th IMA International Conference*, volume 3796 of *Lecture Notes in Computer Science*, pages 78–95. Springer, 2005.
23. V. Rijmen and E. Oswald. Update on SHA–1. In A. Menezes, editor, *Topics in Cryptology – CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 58–71. Springer-Verlag, 2005.
24. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, 2005.

A CubeHash description

CubeHash, designed by Dan Bernstein [3], 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 $r \in \{1, 2, \dots\}$ and $b \in \{1, 2, \dots, 128\}$. The internal state S is set to a specified value which depends on the output hash 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[8]. 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 exclusive 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. CubeHash-8/1 is the official submission to SHA-3 competition. Implementations of CubeHash-8/2, CubeHash-8/4 and CubeHash-8/8 have been benchmarked as well as part of the eBASH [10] project, though.

B Highest raw probability differential paths found for CubeHash

Here we give differential trials for highlighted entries of Table 1. Remember that $\Delta = \Delta^0 || \Delta^1 || ... || \Delta^{t-1}$ is the kernel of linearized version of CubeHash in t iterations mapping t message differences each of b bytes into the last $128 - b$ bytes of the final state before absorbing the $(t+1)$-st message difference. The difference Δ^t is then used to erase the difference in the first b bytes of the state. Note that a differential path for a given r and b is also a differential path for the same r and bigger b's. Therefore we introduce the path for the smallest valid b.

B.1 Differential paths for CubeHash-1/*

CubeHash-1/4 with $y = 32$:

$$(\Delta^0, \dots, \Delta^4) = (\texttt{00000001}, \texttt{00000000}, \texttt{40400010}, \texttt{00000000}, \texttt{00000010})$$

CubeHash-1/3 with $y = 46$:

$$(\Delta^0, \dots, \Delta^4) = (\texttt{010000}, \texttt{000000}, \texttt{104040}, \texttt{000000}, \texttt{100000})$$

CubeHash-1/2 with $y = 221$:

$$(\Delta^0, \dots, \Delta^8) = (\texttt{0080}, \texttt{0000}, \texttt{2200}, \texttt{0000}, \texttt{228A}, \texttt{0000}, \texttt{0280}, \texttt{0000}, \texttt{2000})$$

B.2 Differential paths for CubeHash-2/*

CubeHash-2/4 with $y = 32$:

$$(\Delta^0, \Delta^1, \Delta^2) = (\texttt{00000001}, \texttt{40400010}, \texttt{00000010})$$

[8] The first message byte into the least significant byte of S_0, the second one into the second least significant byte of S_0, the third one into the third least significant byte of S_0, the fourth one into the most significant byte of S_0, the fifth one into the least significant byte of S_1, and so forth until all b message bytes have been exhausted.

CubeHash**-2/3 with $y = 46$:**

$$(\Delta^0, \Delta^1, \Delta^2) = (010000, 104040, 100000)$$

CubeHash**-2/2 with $y = 221$:**

$$(\Delta^0, \ldots, \Delta^4) = (0040, 1100, 1145, 0140, 1000)$$

B.3 Differential paths for CubeHash-3/*

CubeHash**-3/64 with $y = 65$:**

$$\begin{aligned}
\Delta^0 &= 40000000000000000000000000000000000000000000000000\\
&\quad\ 00000000000000000000080000000000000008000000000\\
&\quad\ 00000000000000000000000000000000\\
\Delta^1 &= 00000000004100000000000000100000000000000000\\
&\quad\ 80002080000000800800020808000000000000000000000\\
&\quad\ 41000010000000000140000000000000\\
\Delta^2 &= 00000000000000000000000000000000000000000002000\\
&\quad\ 00000000000020000000000000000000000000000000000\\
&\quad\ 00000000000000000000000000000010
\end{aligned} \tag{12}$$

CubeHash**-3/48 with $y = 364$:**

$$\begin{aligned}
\Delta^0 &= 00000000000000000000000000000000000010000000000\\
&\quad\ 00001000000000000000000000000000000000000000000\\
\Delta^1 &= 00020280000200000002028000020000000000000000000\\
&\quad\ 00000000000000004000010100000000400001010000000\\
\Delta^2 &= 20202200000000002020220000000000400000500000000\\
&\quad\ 40000005000000004044004040440040404400404044040\\
\Delta^3 &= 08000000080002080800000008000208000000000000000\\
&\quad\ 00000000000000000040000000000000040000000000000
\end{aligned}$$

CubeHash**-3/12 with $y = 400$:**

$$\begin{aligned}
\Delta^0 &= 00020000000000000020000\\
\Delta^1 &= 00000000000000000000000\\
\Delta^2 &= 800A2A8200000000800A2A82\\
\Delta^3 &= 00000000000000000000000\\
\Delta^4 &= 00002000000000000002000
\end{aligned}$$

CubeHash**-3/4 with $y = 478$:**

$$(\Delta^0, \ldots, \Delta^4) = (00000100, 00000000, 41400515, 00000000, 00000010)$$

CubeHash**-3/3 with $y = 798$:**

$$(\Delta^0, \ldots, \Delta^4) = (000280, 000000, 0A2A8A, 000000, 080020)$$

B.4 Differential paths for CubeHash-4/*

CubeHash**-4/36 with $y = 130$:**

$$\begin{aligned}
\Delta^0 &= 00200000000000000020000000000000000000000000000\\
&\quad\ 00000000000000000001000\\
\Delta^1 &= 00022002000000000002200200000000000000000000000\\
&\quad\ 00000000000000001000110\\
\Delta^2 &= 00002000000000000002000000000000000000000000000\\
&\quad\ 00000000000000000000010
\end{aligned}$$

CubeHash**-4/12 with $y = 156$:**

$$\begin{aligned}\Delta^0 &= 04000000000000000004000000\\ \Delta^1 &= 00440040000000000000440040\\ \Delta^2 &= 00040000000000000000040000\end{aligned}$$

CubeHash**-4/4 with $y = 189$:**

$$(\Delta^0, \Delta^1, \Delta^2) = (00001000, 01000110, 00000010)$$

CubeHash**-4/3 with $y = 195$:**

$$(\Delta^0, \Delta^1, \Delta^2) = (000200, 200022, 000002)$$

B.5 Differential paths for CubeHash-5/*

CubeHash**-5/64 with $y = 205$:**

$$\begin{aligned}\Delta^0 &= 04000000000000000000000000000000000000000000000000\\ &\quad\ 00000000000000000000000800000000000000008000000000\\ &\quad\ 00000000000000000000000000000000\\ \Delta^1 &= 00000000000400500000000000001040000000000000000000\\ &\quad\ 00022202000202802020220008202000000000000000000000\\ &\quad\ 01400510000000000040040100000000\\ \Delta^2 &= 00000000000000000000000000000000000000000000002000\\ &\quad\ 00000000000020000000000000000000000000000000000000\\ &\quad\ 00000000000000000000000000000010\end{aligned}$$

B.6 Differential paths for CubeHash-6/*

CubeHash**-6/36 with $y = 351$:**

$$\begin{aligned}\Delta^0 &= 00010000000000000001000000000000000000000000000000\\ &\quad\ 00000000000000000800000\\ \Delta^1 &= 40051541000000004005154100000000000000000000000000\\ &\quad\ 00000000000000020A0828A\\ \Delta^2 &= 00001000000000000000100000000000000000000000000000\\ &\quad\ 00000000000000000000008\end{aligned}$$

CubeHash**-6/12 with $y = 400$:**

$$\begin{aligned}\Delta^0 &= 00020000000000000020000\\ \Delta^1 &= 800A2A8200000000800A2A82\\ \Delta^2 &= 00002000000000000002000\end{aligned}$$

CubeHash**-6/4 with $y = 478$:**

$$(\Delta^0, \Delta^1, \Delta^2) = (00000100, 41400515, 00000010)$$

B.7 Differential paths for CubeHash-7/*

CubeHash**-7/64 with $y = 447$:**

$$\begin{aligned}\Delta^0 &= 00008000000000000000800000000000000000000000000000\\ &\quad\ 00000000000000000000040000000000000000000000000000\\ &\quad\ 00000000000000000000000000000000\\ \Delta^1 &= A880A2888800808000000000000000050040114000000000\\ &\quad\ 15400400000000000000000004510400000000040011000\\ &\quad\ 00000000000000000880A288A08000888\\ \Delta^2 &= 00000000000000000000000000000000000000000000000000\\ &\quad\ 00000000000001000000000000000000000000000000000000\\ &\quad\ 00000000000020000000000000002000\end{aligned}$$

C Best total attack complexity differential paths found for CubeHash

Here we give the differential trials for some stared entries of Table 2. Please see the note on Appendix B.

CubeHash**-3/48 with $y = 368$:**

$$
\begin{aligned}
\Delta^0 &= \texttt{00000000000000000000000000000004000000000000000}\\
&\quad\ \texttt{40000000000000000000000000000000000000000000000}\\
\Delta^1 &= \texttt{80002080000000808000208000000080000000000000000}\\
&\quad\ \texttt{00000000000000004040001000000004040001000000000}\\
\Delta^2 &= \texttt{88080008000000008808000800000000040011000000000}\\
&\quad\ \texttt{00400110000000001100101011001010110010101100100}\\
\Delta^3 &= \texttt{00000028000020200000002800002020000000000000000}\\
&\quad\ \texttt{00000000000000001000000000000001000000000000000}
\end{aligned}
$$

CubeHash**-4/36 with $y = 134$:**

$$
\begin{aligned}
\Delta^0 &= \texttt{00000008000000000000000800000000000000000000000}\\
&\quad\ \texttt{00000000000000004000000}\\
\Delta^1 &= \texttt{88008000000000088008000000000000000000000000000}\\
&\quad\ \texttt{00000000000000000440040}\\
\Delta^2 &= \texttt{08000000000000000800000000000000000000000000000}\\
&\quad\ \texttt{00000000000000000040000}
\end{aligned}
\tag{13}
$$

CubeHash**-7/61 with $y = 455$:**

$$
\begin{aligned}
\Delta^0 &= \texttt{04000000000000000000000000000000000000000000000}\\
&\quad\ \texttt{00000000000000000000000800000000000000800000000}\\
&\quad\ \texttt{00000000000000000000000}\\
\Delta^1 &= \texttt{00000000450040100000000001400010000000000000000}\\
&\quad\ \texttt{88A880A288008080880A288A880800080000000000000000}\\
&\quad\ \texttt{01451040000000005001440000}\\
\Delta^2 &= \texttt{00000000000000000000000000000000000000000000002}\\
&\quad\ \texttt{00000000000000020000000000000000000000000000000}\\
&\quad\ \texttt{00000000000000000000000001}
\end{aligned}
$$

D Collision for CubeHash-3/64

$$
\begin{aligned}
M^{-1} &= \texttt{9B91E97363511AC3AF950F54DBCFD5DF91BC26BDD759104DF15B37847A4F7015}\\
&\quad\ \texttt{E15A8844ABA3075A3816AE13E583F2764019331772446464 9F9BE819EB582ECC}\\
M^0 &= \texttt{B22A98139CC0C8606525818EE6DD7775CF25B34196DC51F4641E56ACB918296B}\\
&\quad\ \texttt{BD082AD01D7481EECC950B6C176C45B623CFE1E2638B16255F61E806F34DE91C}\\
M^1 &= \texttt{4D9E9CD62ED12CBDBA1E0B631856DCFE5BD996571CFF6E94A52242382E154FA6}\\
&\quad\ \texttt{AEB44AC0A247CB298550C7B82BDCA924E81D5E51E997CA67FBDD86FF15D04A0D}
\end{aligned}
$$

E Collision for CubeHash-4/48

$$
\begin{aligned}
M^{-1} &= \texttt{741B87597F94FF1CC01761CA0D80B07CC2E6E760C95DF9A5}\\
&\quad\ \texttt{08FFCBABDA11474E2CCEA7AC62A7C822BE29EDCBA99D476C}\\
M^0 &= \texttt{1D30F8022F4AE8DBD477FA1F7DE37C1AF2516BC6FA4657F9}\\
&\quad\ \texttt{E51539C10EC114DA3B8264DD9361FE07C3D56E88E8512201}\\
M^1 &= \texttt{014A11BFE2FF346FC306D1E430EE80268785A9F841562C9A}\\
&\quad\ \texttt{88A6BF5858E95362F541ACF41C2FDCC1C49470DF1DFAEFDC}
\end{aligned}
$$

F Colliding message for MD6 reduced to 16 rounds

5361E9B8579F7CD1, 8B29C52CA2AB51E4, 0BCF2F1E1B116898, 022C254B88191A11,
F0F1CE9D9A7F63B8, 9FB5B2CE87B7D7F5, E7C78F28EEB4F5C7, C5E8C19CEFC07365,
F88B84529ED90209, 8FACF593AE7390CF, 03A93466247C6B54, B12C70C10904143D,
D92EE67244C300D6, 35EEA586ECCC8A77, 9DCF031C64B528F8, C84807607ADCD418,
367E95EE3CB0FC67, 578A2C716FCC5016, B0C30EA5521F61EF, 7F665B24762D5894,
4196BAF0596A7784, ED5F9A8F183B4BCC, 6077463601FCFE46, 495366B1273E119B,
6E11A21AE5B3A48F, 38082264A0F68F93, 4ED510C2DFA9FF98, 35C5ACEC5E9A1756,
1F6731C861879ECD, 8CECD7B4F761CE82, 332A50854FDA8FE6, 588498B1021E9C23,
CB1FFA21CF89C7A5, 63A6871C77848410, 92A550CB4607F31C, 97024803F162E055,
E2D6EA5A57D2DBF3, AEB418A0F1F01CC5, 090A9304040038C1, 5417960E3D9A06A5,
714215C196813F35, BABAD7A4C154F2C3, 71AF3FD02B543940, FA08624B825648DD,
730D61FF48759275, CF85BA5A06D6AED4, 2E12B3150452C65A, 93C7A9FC314220B4,
81B128A4EF361456, BFE652098170C212, 77540989DC246845, 796F353D07721071,
D82776A3CBFEC586, 1132E4391152F408, CE936924CFFB22AA, D338852F80450282,
4F41AB82E790EEF6, F05378CB6BD36203, 5E506F47C6EC4617, FE6FB5A03BDE8E1C,
AB33EA511EEBAEDC, 7D40F8D4F0C62BF4, 1174E2B748B9CC2E, 1EB743671A31547D

G Condition function for CubeHash and MD6

Algorithms 3 and 4 respectively show how the Condition function can be constructed for MD6 and CubeHash. It is presumed that the input Δ is in the the kernel of their respective condition functions. Note that, in case the Condition function needs to be evaluated several times for a fixed differential path Δ, as is the case for the tree-based search algorithm, we can precompute that part of the function which is independent of the input message M.

Algorithm 3 : Condition function for MD6

Require: r, Δ, M and V
Ensure: $y, Y = \mathsf{Condition}_{\Delta}(M, V)$
1: $(A_0, \ldots, A_{88}) := Q||V||M$
2: $(A'_0, \ldots, A'_{88}) := 0||\Delta$
3: $j := 0$
4: **for** $i = 0, 1, \ldots, 16r - 1$ **do**
5: $\quad A_{i+89} := L_i\big(S_i \oplus A_i \oplus (A_{i+71} \wedge A_{i+68}) \oplus (A_{i+58} \wedge A_{i+22}) \oplus A_{i+72}\big)$
6: $\quad A'_{i+89} := L_i\big(A'_i \oplus A'_{i+72}\big)$
7: $\quad D := A'_{i+71} \vee A'_{i+68} \vee A'_{i+58} \vee A'_{i+22}$
8: $\quad T := (A_{i+71} \wedge A_{i+68}) \oplus (A_{i+58} \wedge A_{i+22})$
9: $\quad T' := \big((A_{i+71} \oplus A'_{i+71}) \wedge (A_{i+68} \oplus A'_{i+68})\big) \oplus \big((A_{i+58} \oplus A'_{i+58}) \wedge (A_{i+22} \oplus A'_{i+58})\big)$
10: $\quad$ **for** all bit positions $k = 0, 1, \ldots, 63$ such that $D_k = 1$ **do**
11: $\quad\quad Y_j := T_k \oplus T'_k$
12: $\quad\quad j := j + 1$
13: $\quad$ **end for**
14: **end for**
15: $y := j$

Algorithm 4 : Condition function for CubeHash

Require: $r, b, t, \Delta = \Delta^0 || \ldots || \Delta^{t-1}$, $M = M^0 || \ldots || M^{t-1}$ and $V = (V_0, \ldots, V_{31})$
Ensure: $y, Y = \mathsf{Condition}_{\Delta}(M, V)$

$(S_0, \ldots, S_{31}) := V$
$(S'_0, \ldots, S'_{31}) := (0, \ldots, 0)$
$j := 0$
for t' from 0 to $t-1$ **do**
  Xor $M^{t'}$ into the first b bytes of the state S {see footnote 8}
  Xor $\Delta^{t'}$ into the first b bytes of the state S' {see footnote 8}
  for *round* from 1 to r **do**
    for i from 0 to 15 **do**
      $\alpha := S'_i$, $\beta := S'_{i \oplus 16}$, $A := S_i$, $B := S_{i \oplus 16}$
      Add S_i into $S_{i \oplus 16}$ and xor S'_i into $S'_{i \oplus 16}$
      $D := \alpha \vee \beta$
      $C := A \oplus B \oplus S_{i \oplus 16}$ {carry word}
      $T = \big((\alpha \oplus \beta) \wedge C\big) \oplus (\alpha \wedge B) \oplus (\beta \wedge A) \oplus (\alpha \wedge \beta)$ {see equation 5 or 7}
      for all bit positions $k = 0, 1, \ldots, 31$ such that $D_k = 1$ **do**
        $Y_j := T_k$
        $j := j + 1$
      end for
    end for
    for $0 \leq i \leq 15$ **do** rotate S_i and S'_i to the left by seven bits **end for**
    for $0 \leq i \leq 7$ **do** swap S_i and $S_{i \oplus 8}$ as well as S'_i and $S'_{i \oplus 8}$ **end for**
    for $0 \leq i \leq 15$ **do** xor $S_{i \oplus 16}$ into S_i and $S'_{i \oplus 16}$ into S'_i **end for**
    for $i \in \{16, 17, 20, 21, 24, 25, 28, 29\}$ **do** swap S_i and $S_{i \oplus 2}$ as well as S'_i and $S'_{i \oplus 2}$ **end for**
    for i from 0 to 15 **do**
      $\alpha := S'_i$, $\beta := S'_{i \oplus 16}$, $A := S_i$, $B := S_{i \oplus 16}$
      Add S_i into $S_{i \oplus 16}$ and xor S'_i into $S'_{i \oplus 16}$
      $D := \alpha \vee \beta$
      $C := A \oplus B \oplus S_{i \oplus 16}$ {carry word}
      $T = \big((\alpha \oplus \beta) \wedge C\big) \oplus (\alpha \wedge B) \oplus (\beta \wedge A) \oplus (\alpha \wedge \beta)$ {see equation 5 or 7}
      for all bit positions $k = 0, 1, \ldots, 31$ such that $D_k = 1$ **do**
        $Y_j := T_k$
        $j := j + 1$
      end for
    end for
    for $0 \leq i \leq 15$ **do** rotate S_i and S'_i to the left by eleven bits **end for**
    for $i \in \{0, 1, 2, 3, 8, 9, 10, 11\}$ **do** swap S_i and $S_{i \oplus 4}$ as well as S'_i and $S'_{i \oplus 4}$ **end for**
    for $0 \leq i \leq 15$ xor $S_{i \oplus 16}$ **do** into S_i and $S'_{i \oplus 16}$ into S'_i **end for**
    for $i \in \{16, 18, 20, 22, 24, 26, 28, 30\}$ **do** swap S_i and $S_{i \oplus 1}$ as well as S'_i and $S'_{i \oplus 1}$ **end for**
  end for{r rounds}
end for{t iterations}
$y := j$