J Cryptol (2015) 28:879–898
DOI: 10.1007/s00145-014-9186-9

Journal of
CRYPTOLOGY

Collision Attack on Grindahl

Thomas Peyrin
Nanyang Technological University, Singapore, Singapore
thomas.peyrin@gmail.com

Communicated by Preneel

Received 2 November 2007
Online publication 28 May 2014

Abstract. Hash functions have been among the most scrutinized cryptographic primitives in the previous decade, mainly due to the cryptanalysis breakthroughs on MD-SHAfamily and the NIST SHA-3 competition that followed. Grindahl is a hash function proposed at FSE 2007 that inspired several SHA-3 candidates. One of its particularities is that it follows the AES design strategy, with an efficiency comparable to SHA-256. This paper provides the first cryptanalytic work on this scheme and we show that the 256-bit version of Grindahl is not collision resistant. Our attack uses byte-level truncated differentials and leverages a counterintuitive method (reaching an internal state where all bytes are active) in order to ease the construction of good differential paths. Then, by a careful utilization of the freedom degrees inserted every round, and with a work effort of approximatively 2^{112} hash computations, an attacker can generate a collision for the full 256-bit version of Grindahl.

Keywords. Grindahl, AES, Hash functions, Collision, Cryptanalysis.

1. Introduction

Cryptographic hash functions are fundamental primitives in information security used in a variety of applications such as message integrity, authentication schemes, or digital signatures. Mathematically speaking, a hash function maps $\{0, 1\}^*$, the set of all finite length bit strings, to $\{0, 1\}^n$, where n is the fixed size of the hash value. Ideally, a cryptographic hash function H should possess the following properties [33]:

- *collision resistance*: finding a pair $x \neq x' \in \{0, 1\}^*$ such that $H(x) = H(x')$ should require $2^{n/2}$ operations;
- *2nd preimage resistance*: for a given $x \in \{0, 1\}^*$, finding an $x' \neq x$ such that $H(x) = H(x')$ should require 2^n operations;
- *preimage resistance*: for a given $y \in \{0, 1\}^n$, finding an $x \in \{0, n\}^*$ such that $H(x) = y$ should require 2^n operations.

Generally, hash functions are built upon a *compression function* and a *domain extension algorithm*. A compression function h has the same security requirements as a hash function but takes fixed-length inputs instead. Then, a domain extension method allows the hash function to handle arbitrary length inputs by defining an (often iterative) algorithm using the compression function as a black box. The pioneering work of Merkle and Damgård [12,34] provided designers an easy way in order to turn collision resistant compression functions onto collision resistant hash functions. After dividing the message to hash (appropriately padded) into blocks m_i, one simply has to update iteratively a state cv_i (called *chaining variable*) with each message block: $cv_{i+1} = h(cv_i, m_i)$. The first state is defined by the *initial value* $cv_0 = IV$. Finally, after having processed every message block, the final state is the output of the hash function. Even if preserving collision resistance, it has been shown that this iterative process presents flaws [13,24–26], and new algorithms [2,6] with better security properties have been proposed.

Almost all published hash functions define a compression function that can be used with any hash domain extension algorithm. There are three different ways of building a compression function. First, one can relate the security of h to a hard problem, such as factorization [9], finding small vectors in lattices [3], syndrome decoding [1], or solving multivariate quadratic equations [7]. The usually bad efficiency of these schemes is compensated by the proofs of security they provide. Another very active domain is the construction of secure compression functions based on block ciphers, which would allow, for example, to build AES-based compression functions. The problem of building a secure n-bit compression function from an ideal n-bit block cipher is more or less resolved [8,42,43] and due to a need of bigger output size, the cryptographic community is now concentrating on the problem of building a secure $(k \times n)$-bit compression function from an ideal n-bit block cipher [20,41,45]. Finally, the most common and efficient way of building a compression function is from scratch, for example, the well known and previously standardized SHA-1 [37] or MD5 [44]. Usually, the dedicated compression functions follow the Davies-Meyer mode that turns a block cipher into a compression function, and a block cipher is, therefore, built from scratch for that purpose. Nevertheless, most hash standards (based on addition-rotation-XOR operations) use this type and they have been broken by novel cryptanalysis results [46–49].

In order to anticipate further improvements of the attacks, NIST has initiated, in 2008, an effort [36] to develop the next hash standard through a public competition, similar to the development process for the Advanced Encryption Standard [38]. This competition finally ended with the choice of KECCAK [4] as the new SHA-3 standard. As precursors of many SHA-3 candidates, some hash functions have been published before the competition, such as LAKE [23], FORK-256 [21], Radio-Gatùn [5], or Grindahl [29].

Despite using known parts of AES for its round transformation, Grindahl cannot be considered as a conservative proposal. This 256-bit hash function does neither use the famous Merkle-Damgård paradigm nor the Davies-Meyer construction and the designers preferred a new configuration: an internal state much bigger than the output size, updated by message words thanks to a fast round function. As output function, blank rounds without incoming message words precede the final truncation of the internal state. Regarding implementation, it requires not much memory and runs faster than SHA-256. The idea underlying this construction is that a big internal state will make it harder for an attacker to build internal collisions (collisions happening before the blank rounds), while collisions due to the final truncation are very unlikely to be forced because of the large number of cryptographic operations during the blank rounds. The designers of Grindahl claimed a collision security of 2^{128} operations as for an ideal 256-bit hash function[1], and security arguments were provided with regard to the number of active Sboxes in a differential path. However, we show in this article that one can find a collision with a work effort of only 2^{112} hash computations. Our method utilizes truncated differences which are very helpful for simplifying the differential analysis of AES-based primitives. In order to further facilitate the search for good differential paths, we will intentionally reach internal states entirely filled with differences and build the collision path backward starting from a colliding state. Finally, once the differential path set, we leverage the freedom degrees available at each round in order to reduce the collision attack complexity as much as possible.

The paper is organized as follows. In Sect. 2, we briefly recall the specification of the Grindahl hash function and in Sects. 3 and 4 we begin the analysis with various observations on the scheme and the general methodology that allows us to build a differential path. Then, in Sect. 5, we provide the first collision attack on Grindahl. Finally, we discuss possible patches in Sect. 6 and we conclude in Sect. 7.

Since the first publication of our attack at the ASIACRYPT 2007 conference [39], several attacks were built on our results. First, our reasoning was likely to apply to the 512-bit version of Grindahl as well, even if the much bigger internal state hardens the attacker's task (his ability to control the differential transitions of the MixColumns operations is reduced). Using our findings, Khovratovich [27] described the first collision attack on the 512-bit version of Grindahl by starting with potentially less interesting truncated differential paths, but for which whole structures of inputs (instead of pairs) can be built in order to greatly reduce the attack complexity. Moreover, our idea to apply truncated differential analysis for the study of AES-based cryptography primitives (only reasoning on the MixColumns truncated differential transitions) looks very promising, as confirmed by the later discovery of rebound attacks [32] and its numerous variations [17, 30,31] that broke many SHA-3 candidates during the competition. Finally, it is to be noted that our freedom degrees fixing technique is quite efficient against stream-cipher-oriented hash functions, as shown, for example, on Radio-Gatùn [16].

On the constructive side, several SHA-3hash functions proposals [19,22,35] took care of our attacks during the design phase. In particular, the designers of FUGUE [19] managed to prove lower bounds on the complexity of our techniques when applied to their proposal. In general, preventing this type of attacks without an important efficiency drop is not trivial, and it seems that slightly increasing the internal state size is a good solution.

2. The Grindahl Family of Hash Functions

Grindahl is a family of hash functions based on the so-called *Concatenate-Permute-Truncate* strategy, where in our case the permutation uses the design principles of

[1] Concerning second-preimage and preimage, the authors also claimed a resistance up to 2^{128} computations, which is lower than what one expects from an ideal 256-bit hash function.

Rijndael [10], well known for being the winning candidate of the Advanced Encryption Standard (AES) process [38]. Two algorithms are defined, a version with a 256-bit output and a 512-bit one. Also, a compression function mode is given, taking only fixed-length inputs, to be used with any hash domain extension algorithm. We give in this section a brief description of the Grindahl hash function with a 256-bit output. For a more detailed specification of the algorithm, we refer to [29].

Let $n = 256$ be the number of output bits of the hash function H, with an *internal state* IS of 48 bytes (384 bits), and let M be the message (appropriately padded) to be hashed. M is split into m blocks $M_1, \dots, M_m$ of 4 bytes each (32 bits). At each iteration k, the message block M_k is used to update the internal state IS_{k-1}. We call *extended internal state* EIS_k the concatenation of the message block M_{k+1} and the internal state IS_k, i.e., $EIS_k = M_{k+1}||IS_k$ and we thus have $|EIS_k| = 416$ bits. We denote by $trunc_t(x)$ the rightmost t bits of x. Let $P : \{0,1\}^{416} \longmapsto \{0,1\}^{416}$ be a non-linear permutation, and let IS_0 be the *initial internal state* defined by $IS_0 = \{0\}^{384}$. Then, for each iteration k with $0 < k < m$, we have $IS_k = trunc_{384}(P(EIS_{k-1}))$. For the last iteration, the truncation is omitted: $EIS_m = P(EIS_{m-1})$. Finally, we apply eight *blank rounds* $EIS_k = P(EIS_{k-1})$, for $m < k \leq m + 8$, and the final output of the hash function is $trunc_{256}(EIS_{m+8})$.

The description is not complete since P has not yet been defined. This permutation follows the design principle of AES (the reader is expected to be familiar with the transformation defined in the AES specifications) and thus the extended state EIS is viewed as a matrix of bytes. However, instead of a (4, 4) byte matrix, we have a matrix α of 4 rows and 13 columns in the case of the 256-bit version of Grindahl. The entry of the matrix α located at the i-th row and the j-th column is a byte denoted by $\alpha_{i,j}$. Consequently, we have

$$\alpha = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,12} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,12} \\ \alpha_{2,0} & \alpha_{2,1} & \cdots & \alpha_{2,12} \\ \alpha_{3,0} & \alpha_{3,1} & \cdots & \alpha_{3,12} \end{pmatrix}.$$

By splitting the extended internal state EIS into 52 8-bit chunks $x_0, \dots, x_{51}$, we can define the conversion from EIS to α by $\alpha_{i,j} = x_{i+4\times j}$ and this mapping has a natural inverse. Before each iteration, the first column of α is overwritten with the incoming message block. To conclude the description, the permutation P is defined as

$$P(\alpha) = \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes} \circ \text{AddConstant}(\alpha).$$

MixColumns: This transformation is defined as in the AES specifications, that is a linear mixing operation which operates on each column of the state independently, combining the four bytes in each column in order to provide diffusion.

ShiftRows: As for AES, this transformation cyclically shifts bytes a number of positions along each row, but here the i-th row is rotated by ρ_i positions to the right, with $\rho_0 = 1$, $\rho_1 = 2$, $\rho_2 = 4$, and $\rho_3 = 10$.

SubBytes: The only non-linear part of the permutation. This substitution replaces each byte of the state by its corresponding byte in the AES Sbox lookup table.

AddConstant: Because we are in the hash function setting, no key is available and the *AddRoundKey* function from AES has to be changed. Therefore, it is replaced by the function AddConstant which is simply defined by $\alpha_{3,12} \longleftarrow \alpha_{3,12} \oplus \texttt{01}$, where 01 is the byte-wise hexadecimal value of 1.

The 512-bit version of Grindahl is based on the same design principle as the 256-bit version, but the extended internal state is larger (8 rows instead of 4). The compression function mode for Grindahl-256 simply consists in hashing 40 4-byte message blocks for each compression function call.

3. First Observations

Before describing the whole collision attack, we begin this section with some remarks about Grindahl that will be useful in the following sections. The first observation allows us to build a differential path in a precomputation phase and the second one speeds up the final collision search.

3.1. *A potential attack and the truncated differences*

In the original Grindahl paper [29], a section explains a potential attack method, pointed out by an anonymous reviewer. This method seems quite natural: the attacker does not look at the actual values of differences inserted in the bytes of the internal state, but only checks if there is a difference or not (this greatly simplifies the analysis). Said in other words, he only forces the zero difference to some bytes of the state, while allowing any difference for the remaining bytes. We call this kind of zero or non-zero differences *truncated differences* in reference to the very similar truncated differences used by Knudsen in [28]. Then, a chain of truncated differences in which in every round the number of actives bytes (bytes with a non-zero truncated difference) is low must be found. In this differential path, the truncated differences can only be erased during two stages of an iteration: during a MixColumns transformation or during the truncation at the end of the iteration. This implies that the number of truncated differences in a column can be reduced and their row position changed by a clever use of the MixColumns transformation, even if one can never erase all the truncated differences of a column at a time. Otherwise, a truncated difference is deleted if it goes to the first column of α at the end of the iteration, due to the truncation $trunc_{384}(\cdot)$. Since at this stage of the attack, the whole differential path is already settled, one cannot force anything for the truncation but one can play with the message blocks inserted at each iteration, in order to force a correct behavior in the MixColumns processes (see Sect. 3.2). In fact, the message bytes act as *control bytes* in the sense that new input bytes do not affect some parts of the internal state for a limited number of rounds (see Sect. 3.3).

To summarize, a truncated difference can be removed during the differential path construction by the truncation or during the collision search by a clever use of control bytes. The feasibility of this method was left as an open problem: one of its main points is that the attacker has to always keep as few active bytes as possible in the differential path, but we will see later that the designers of Grindahl prevented this kind of low weight path during the conception. We argue in Section 4.1 that there exists a better technique to find collisions for Grindahl.

3.2. *Differences Transitions in the MixColumns Function*

The MixColumns transformation matrix used in Grindahl is the same as in the specifications of AES [10], and its Maximum Distance Separable (MDS) property ensures maximal difference propagation. More precisely, the sum of the number of active bytes of the input and the output is always greater than or equal to 5. In other words, the number of non-zero truncated differences of the input and the output of MixColumns is always greater than or equal to 5 (or obviously equal to zero if there is no difference at all).

More formally, let $V = (A, B, C, D)$ be an input vector of four bytes A, B, C, and D; and let $W = (A', B', C', D')$ be an output vector of four bytes A', B', C', and D'. We denote the function MixColumns by $MC : V \longmapsto W$ or $MC : (A, B, C, D) \longmapsto (A', B', C', D')$. We also denote by $D_i(V_1, V_2)$ the function returning 1 if the i-th byte of the 4-byte vectors V_1 and V_2 are different, and 0 otherwise. Finally, $ND(V_1, V_2)$ returns the number of such differences, i.e., $ND(V_1, V_2) = \#\{i \mid D_i(V_1, V_2) = 1\}$. We thus have that if $W_1 = MC(V_1)$ and $W_2 = MC(V_2)$ with $V_1 \neq V_2$, then

$$ND(V_1, V_2) + ND(W_1, W_2) \geq 5.$$

Another interesting property is that any input byte of MixColumns defines a permutation for any output byte. Therefore, with $W_1 = MC(V_1)$, $W_2 = MC(V_2)$, and $V_1 \neq V_2$ drawn uniformly and randomly in $\{0, 1\}^{4\times 8}$, we have for any $1 \leq i \leq 4$:

$$\begin{aligned} P_D &= P[D_i(W_1, W_2) = 0] = \frac{256^3 - 1}{256^4 - 1} \simeq 2^{-8}, \\ \overline{P_D} &= P[D_i(W_1, W_2) = 1] = 1 - P_D \simeq 1 - 2^{-8}. \end{aligned}$$

Our goal is to compute the probability that a fixed mask of input- truncated differences maps to a fixed mask of output-truncated differences (later this will be often utilized in order to compute the probability of success of the differential path). For example, we want to compute the probability that two input words V_1 and V_2 distinct in their 2 first bytes result in two output words different in their 3 first bytes through MixColumns (note that this is slightly different from the event that any 2-byte difference input maps to any 3-byte difference output). We can compute those probabilities in two ways, formally or empirically by testing exhaustively all the input values: since MixColumns are linear, dealing with differences or values is the same (during the test, instead of looking for differences or non-differences, we checked for zero values or non-zero values). We give in Table 1 an approximation of the probability P that two 4-byte input words with $D_I \neq 0$ different bytes in predefined positions map to two 4-byte output words with $D_O \neq 0$ different bytes in predefined positions through MixColumns: $P \simeq 2^{-8\times(4-D_O)}$ if $D_I + D_O \geq 5$, and $P = 0$ otherwise.

Table 1. Approximate probability that two 4-byte input words with $D_I \neq 0$ different bytes in predefined positions map to two 4-byte output words with $D_O \neq 0$ different bytes on predefined positions through MixColumns. The values are base 2 logarithms.

D_O D_I	0	1	2	3	4
0	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
1	$-\infty$	$-\infty$	$-\infty$	$-\infty$	0
2	$-\infty$	$-\infty$	$-\infty$	-8	0
3	$-\infty$	$-\infty$	-16	-8	0
4	$-\infty$	-24	-16	-8	0

Table 2. Influences on the columns of the extended internal states for a modification of a byte of the message block $M_k = (A_k, B_k, C_k, D_k)$ incoming at iteration k in Grindahl. We denote by ✓ if the column is affected (or active) and void if not. The first table shows influences on IS_{k-1}, the second on IS_k and the third on IS_{k+1}.

	0	1	2	3	4	5	6	7	8	9	10	11	12
A_k		✓											
B_k			✓										
C_k					✓								
D_k											✓		

	0	1	2	3	4	5	6	7	8	9	10	11	12
A_k			✓	✓		✓						✓	
B_k				✓	✓		✓						✓
C_k		✓				✓	✓		✓				
D_k		✓						✓				✓	✓

	0	1	2	3	4	5	6	7	8	9	10	11	12
A_k	✓		✓	✓	✓	✓	✓	✓	✓	✓			✓
B_k	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓		
C_k			✓	✓		✓	✓	✓	✓	✓	✓	✓	✓
D_k	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓

3.3. *The Control Bytes*

Modifying some message bytes will obviously modify quite quickly the internal state, but not immediately. For each modified byte of the message M_k, we give in Table 2 the columns of s (in its matrix representation α) affected by this modification after 1, 2, and 3 iterations. For more than 3 iterations, the Grindahl diffusion is such that any message byte affects the complete internal state. This diffusion feature will allow us to attack different columns of different iterations independently: we will be able to control independently the behavior of some MixColumns transitions. Those *control bytes* will be useful during the collision search phase of the attack and will later help us to speed up the search for a message pair that follows exactly our predefined differential path.

4. High-Level View of the Attack

In this section, we study possible ways of finding a good differential path for the 256-bit version of Grindahl: we look for a path of k iterations starting from IS_0 and so that with two different messages M and M' we have the same hash output, i.e., $trunc_{256}(EIS_{m+8}) = trunc_{256}(EIS_{m'+8})$. Finding a differential path leading to a collision and including the blank rounds seems hard since no message block is inserted during this last operation and so we have very little control on this part. However, the problem looks much easier when trying to find an internal collision: a differential path excluding the blank rounds, i.e., $EIS_m = EIS_{m'}$. One can easily see that by avoiding to add any difference right after, an internal collision will directly provide a full collision after the blank rounds. Here, we explain how to find such a differential path and give techniques that decrease the overall complexity of the attack.

4.1. *A Counterintuitive Strategy*

We now have all the necessary tools to build a truncated differential path, evaluate its probability of success, and speed up the collision search. But how to actually find a good truncated differential path? The natural intuition one would have (as the anonymous reviewer suggested) is to always maintain a low number of truncated differences along the path in order to increase its probability of success, though finding one such path seems really difficult as one can convince oneself with Property 1 from the original Grindahl paper [29]:

Property 1. An internal collision for Grindahl-256 requires at least 5 iterations. Moreover, any differential path starting or ending in the extended state with no difference contains at least one round where at least half of the extended state bytes (excluding the first column) are active.

This property can be verified with a meet-in-the-middle exhaustive search, as explained in the original Grindahl paper. Besides, with a small speed improvement of this algorithm, one can check that an internal collision for Grindahl-256 requires in fact at least 6 iterations. Another observation is that by introducing differences in the state and after a few iterations we quickly come to an *all-active* pair of extended states. This all-active pair of extended states is almost stable: the probability that an all-active pair of columns remains an all-active pair of columns through MixColumns is approximatively $P_A = (1 - 2^{-8})^4$, so for the twelve columns of the extended state (except the first column) we have a probability of $P_A^{12} \simeq 2^{-0.27}$. Thus, our first idea is to not search for a path starting from a zero difference but from an all-active pair of extended states, which is very easy to get. The overwhelming probability P_A^{12} allows us to start with as much valid starting states as we want (each valid starting state can be generated with an average complexity of $2^{0.27}$ computations).

This concept of letting all the differences spread is really counterintuitive for a cryptographer, especially when dealing with hash functions where an attacker always tries to keep control of the difference spreading during the differential path. Nevertheless, this idea makes sense here because it looks like the designers built their hash function with the major security argument being that controlling the difference spreading should be hard, as illustrated by Property 1 from their original paper. Thus, we will let it be totally out of our control (but in fact completely under control in terms of truncated differential path) and right after try to force it to a collision. Furthermore, this method is facilitated by the fact that unconstrained fresh message words are arriving at each iteration and then the two different parts of the attack (first arrive to an all difference state and second make it a collision) can be done independently.

Overall, this method allows us to greatly simplify the truncated differential path search (just like we simplified the analysis with the truncated differences). Even if we might not obtain the best possible path with this technique, we will get very good ones, which will be sufficient for a collision attack.

4.2. *How to Build a Truncated Differential Path*

Searching for a differential path starting from an all-active pair of extended internal states and ending in a collision is quite easy. One method is to search backward almost exhaustively since in Grindahl the truncated differences propagate in the forward direction as quickly as in the backward direction. More precisely, if we look for a collision at the end of iteration k, we try all the possible truncated difference masks for the message blocks inserted at iterations k, $k-1$, etc., and all the possible backward transitions of truncated differences through MixColumns (same as for the onward direction), until we come to an all-active pair of extended states. This algorithm can be greatly improved with an early-abort strategy: we compute a lower bound on the complexity cost of the current path we are building (taking in account the control provided by the active/passive bytes, see Section 5) and we stop the search branch if the complexity of the attack is already greater than or equal to 2^{128} operations. We also stop the search if we went too far in terms of number of iterations: in some particular cases, the overall complexity of a differential path can remain stable even if its number of iterations increases, for example, when the number of MixColumns transitions imposed is lower than or equal to the number of control bytes inserted.

Obviously, always adding truncated differences to all the message blocks inserted is the fastest way to reach this goal. However, we will later use the message bytes inserted as control bytes to attack some parts of the differential path independently and, therefore, increase the probability of success of the path. Then, it may be better not to go too fast on adding truncated differences in order to have more iterations during the differential path. Doing so increases the total number of message blocks inserted and, therefore, provides more control bytes. This can be regarded as some kind of dilution of the MixColumns constraints to be imposed by stretching the differential path. For example, we can find a path starting from an all-active pair of extended internal states and requiring only 4 iterations to get a collision, with a probability of success of approximatively 2^{-312}. Still, another path requiring 8 iterations to get a collision with a probability of success of approximatively 2^{-440} may be better. Indeed, in the latter case, even if the probability of success has been divided by a factor 2^{128}, we have inserted 8 message word pairs instead of only 4 in the former case. Consequently, we get roughly 256 additional degrees of freedom compared to the former case (4 pairs of message of 4 bytes each) and those can be used to attack some parts independently, potentially decreasing the complexity by more than a factor 2^{128}. Naturally, limits exist: at some point, adding more iterations does not improve things anymore. Also, control bytes cannot always be used in an ideal way and we only come to a lower bound on the complexity cost of the path. Once potentially good paths have been found, a case-by-case analysis is required. This analysis can be automated and is explained in the next section.

5. A Collision Attack for Grindahl-256

In this section, we use the previous observations to present a complete collision attack for the 256-bit version of Grindahl. Other attacks might be possible, depending on which differential path we use, but we explain here the details for the collision attack corresponding to the best path found according to our technique.

5.1. *Our Truncated Differential Path*

Before describing our attack, we give in Fig. 1 the truncated differential path used, which has been generated with a program implementing the previously explained method (see Sect. 4.2). It starts from an all-active pair of states and collides after 9 iterations (for space reasons, the first iteration is not represented in Fig. 1, but it simply maps an all-active truncated difference to itself). A cell stands for a byte and each group of cells represents a 52-byte extended internal state. A dark cell means that we have a non-zero difference for this byte, and a light cell stands for no difference. Each row of extended internal states represents one iteration. The first column gives the differences in the state just after its update with the 4-byte message word, and the second column gives the same state after application of the ShiftRows transformation. Finally, the third column represents the internal state just after application of the MixColumns function. In this third column, the dark active cells marked with a light-gray circle in the middle represent the cells located in a column for which the differential transition through the MixColumns is not free (the cells located in the first column of the state are filled with dark gray to depict that we do not care about the differences in these cells since they will be erased by the truncation). Note that because the AddConstant and SubBytes functions have no effect on the differential path, they are omitted here. Each first 4-byte column of the first column states represents the message words inserted at each iteration, that will later be used as control bytes. The first 4-byte column of the state after every MixColumns transition can have whatever difference mask since those bytes will be immediately truncated.

This differential path is the best found among other possible candidates leading to the same complexity. We denote by k the number of the last iteration of our differential path, i.e., the last row of Fig. 1. First, one can check that all the MixColumns transitions are valid, i.e., verify the MDS property. This differential path has a probability of success of approximatively $2^{-55\times 8} = 2^{-440}$ (55 MixColumns constraints forced in total), which seems very low at first sight. However, in this path, we also have a lot of message blocks inserted that one can use during the collision search to force some MixColumns constraints independently.

Our aim is to find a pair of messages following the expected differential path. For this, we do not handle each iteration one by one, but we deal with each of the 4-byte message

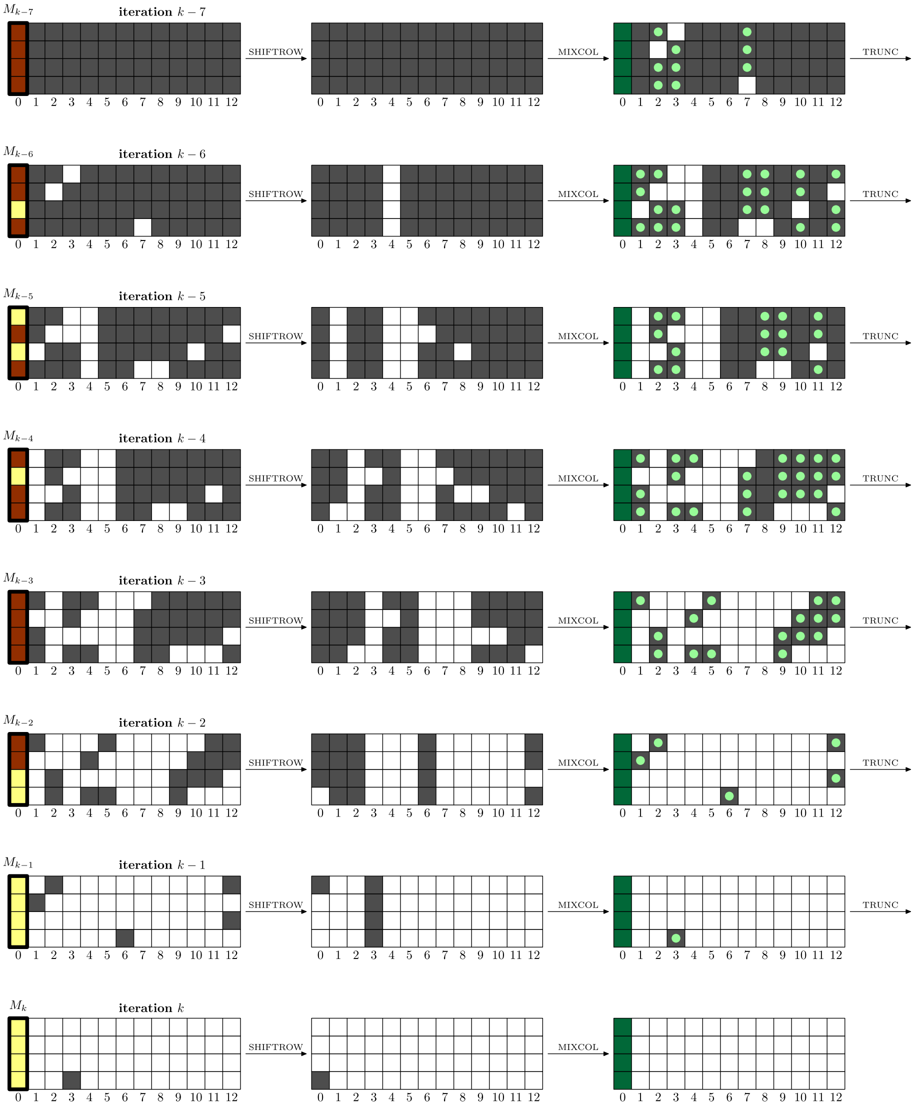

Fig. 1. A possible truncated differential path, starting from an all-active internal state and potentially providing a collision after 8 iterations.

words inserted one by one. Said in other words, we will fix the four bytes of a message word pair and check that the newly imposed MixColumns differential transitions are the ones expected in our truncated differential path. If so, we continue to the next message word pair until we get a collision.

In Table 3, we give all the dependencies of the MixColumns transitions with the message blocks inserted, used as control bytes during the collision search, following the differential path from Fig. 1. The cost of all the transitions is given (see Sect. 3.2) along with the number of control bytes inserted at each iteration (see Sect. 3.3). The second column of the table gives the position of the columns of the state in which we force a truncated differential transition during a MixColumns transformation, and the first column indicates in which iteration of this event occurs. For each transition, we give in

Table 3. Dependencies of the message blocks used as control bytes and inserted during the truncated differential path from Figure 1, for a collision at the end of iteration k.

it	col	cost	message blocks inserted																											
			$k-8$				$k-7$				$k-6$				$k-5$				$k-4$				$k-3$				$k-2$			
			A	B	C	D	A	B	C	D	A	B	c	D	a	B	c	D	A	b	C	D	A	B	C	D	A	B	c	d
k-7	2	1	–					×																						
	3	1	×	×																										
	7	1				×																								
k-6	1	1		–		–			–	–	×																			
	2	1	–		–	–	–					×																		
	3	2	–	–	–	–	×	×																						
	7	1	–	–	–					×																				
	8	1	–	–	–	–			×																					
	10	1		–	–									×																
	12	1	–		–	–		×		×																				
k-5	2	1	–	–	–	–	–		–	–	–					×														
	3	1	–	–	–	–	–	–	–	–	×	×																		
	8	1	–	–	–	–	–	–	–	–			×																	
	9	1	–	–	–	–	×	×	×	×																				
	11	1	–	–	–	–			–	–	×			×																
k-4	1	1	–	–	–	–	–	–	–	–		–		–			–	–	×											
	3	1	–	–	–	–	–	–	–	–	–	–	–	–	×	×														
	4	2	–	–	–	–	–	–	–	–	–	–		–		–					×									
	7	1	–	–	–	–	–	–	–	–	–	–	–					×												
	9	1	–	–	–	–	–	–	–	–	×	×	×	×																
	10	1	–	–	–	–	–	–	–	–		–	–									×								
	11	1	–	–	–	–	–	–	–	–			–	–	×			×												
	12	1	–	–	–	–	–	–	–	–	–		–	–		×		×												
k-3	1	3	–	–	–	–	–	–	–	–	–	–	–	–		–		–			–	–	×							
	2	2	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–	–					×						
	4	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–		–					×					
	5	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	×		×									
	9	2	–	–	–	–	–	–	–	–	–	–	–	–	×	×	×	×												
	10	2	–	–	–	–	–	–	–	–	–	–	–	–		–	–									×				
	11	1	–	–	–	–	–	–	–	–	–	–	–	–			–	–	×			×								
	12	2	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–		×		×								
k-2	1	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–		–			–	–	×			
	2	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–	–					×		
	6	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–			×	×					
	12	2	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–		–	–		×		×				
k-1	3	3	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	×	×		
COST			**0**				**0**				**0**				**1**				**2**				**6**				**5**			

the third column its cost in terms of number of bytes (i.e., for a cost c, the transition has a probability of $2^{-c\times 8}$). Then, each of the seven other columns of the table represents a pair of message words that will be used as control bytes (the letters a or A, b or B, c or C, and d or D represent, respectively, the first, second, third, and fourth byte of the 4-byte message inserted). Capital letters mean that we have 2 control bytes (a difference is inserted for this message block and we can make independently both messages of the pair vary) and small letters mean that we only have 1 control byte (no difference inserted for this message block). In the core of the table, a dash or a cross represents the fact that the MixColumns transition indicated by the corresponding row is affected by the control byte indicated by the corresponding column. We divided those dependencies for the sake of simplicity, the crosses are the dependencies that may be used for the attack: chronologically they represent for each MixColumns transition the dependencies of the last involved message word during the attack. The last row gives the cost of each message word insertion during the collision search in terms of number of bytes. The sum for all the message blocks gives the total complexity of the attack.

By looking at Table 3, since there can be several crosses in a single column, one may have the impression that some bytes of the message are used several times during the attack. However, we recall that in Table 3 the crosses do not represent the dependencies that we will use, but the ones that we **may** use during the attack. We ensured that no freedom degree is used twice during the message bytes fixing procedure.

Finally, from Table 3, one can check that we need to test $2^{14\times 8} = 2^{112}$ all-active pairs of internal state in order to have a good probability of obtaining a collision, as 14 MixColumns constraints cannot be forced independently during the collision search. We explain the whole process in more details in the next section.

5.2. *The Collision Search*

Our final collision search is based on three steps. The first one generates a sufficient number of all-active extended internal states and the second one checks for each candidate if a collision can be found by using the control bytes. Once a pair of message blocks following the differential path is found, the third step ensures the validity of our new internal collision by forcing the last truncation.

5.2.1. *First Step:*

start with the predefined initial value of Grindahl and compute a few iterations with lots of truncated differences in the incoming message blocks in order to quickly come to an all-active pair of states denoted $\mathcal{A}$ after a few iterations. From this pair of states $\mathcal{A}$, generate $2^{14\times 8} = 2^{112}$ new all-active pairs of states $\mathcal{A}_1, \ldots, \mathcal{A}_{2^{112}}$, for example, by choosing randomly a new input pair of message blocks. When all the possible pairs of message blocks for $\mathcal{A}$ are exhausted, replace $\mathcal{A}$ by another all-active pair of states found during the process. This part requires $2^{112} \times 2^{0.27} = 2^{112.27}$ iterations.

5.2.2. *Second Step:*

in this step, for each pair of message words (M_{k-i}, M'_{k-i}) inserted, their bytes are used in order to adjust the behavior of the MixColumns transitions where crosses appear at column M_{k-i} in Table 3. Then, we continue the attack by fixing the control bytes iteration per iteration: for the message blocks inserted at the beginning of iterations $k-8$, $k-7$, $k-6$ of our truncated differential path from Table 3, there are more control bytes incoming than necessary. Indeed, we have 8, 8, and 7 control bytes available for the messages inserted at iterations $k-8$, $k-7$, and $k-6$, respectively, whereas we only require 2, 7, and 7 bytes of degrees of freedom. Note that since in Table 3 the crosses represent the bytes of the last message word involved in a transition, the previous dependencies (represented by a dash) are already fixed at this point. For each step, the total cost is equal to the sum of the costs of all the MixColumns transitions involved minus the number of control bytes available from M_{k-i}, provided they can all be utilized properly (which is often the case). Consequently, at this point of the attack, we maintain 2^{112} pairs of messages and states following the differential path. For the message words inserted at iteration $k-5$, we have 6 control bytes for 7 bytes of conditions, thus we only keep 1 out of 2^8 message pairs and we go to the $(k-4)$-th message word with 2^{104} valid pairs. We continue in the same way for the three remaining message words $k-4$, $k-3$, and $k-2$, having 7, 8, and 4 control bytes, respectively, and requiring 9, 14, and 9 bytes of conditions. Concerning the $k-2$ case, we only have 4 control bytes and not 6 as indicated in Table 3, because c and d are not involved in any MixColumns transition and so they cannot be considered as control bytes. Finally, we expect to have one pair of messages following the differential path with a good probability by starting with $2^{14\times 8} = 2^{112}$ all-active pairs of states.

5.2.3. *Third Step:*

add a $(k+1)$-th message block without truncated difference in order to force a truncation after the last iteration k of the differential path (remember that there is no truncation of the extended internal state before the blank rounds).

5.3. *Discussion on the Attack*

The distinction between crosses and dash dependencies is a restriction for the attacker but allows a simpler description of the attack. When dealing with crosses, the attacker knows that all the previous dependencies are already set and it makes things much simpler even if better attacks may exist by using a more complicated message byte fixing schedule. The very same remark applies for the fact that we fix the message bytes word by word: much simpler to describe but may be not the best technique.

To the contrary, dealing with crosses does not mean that the control bytes can be used carelessly. There may exist situations where the attacker cannot use all of its control bytes: we took care of the dependencies word wise with the crosses/dash distinction, but the byte-wise dependencies remain. Still, those situations occur relatively rarely and do not happen in our presented differential path.

For the sake of clarity, we explain more precisely how to deal with the control bytes by giving an example. Let us set ourselves when the attacker has to fix the message pair incoming at step $k-5$ (seventh column in Table 3). The previous message words have already been fixed during the attack, thus we only have to deal with the crosses in Table 3. Some MixColumns differential transitions have to behave as required by the truncated differential path, and this has a cost. For example, at the second column of the $(k-5)$-th iteration, we need a 4 non-zero truncated differences to 3 non-zero truncated differences transition and this will happen with probability 2^{-8}, therefore, with a cost of 1 byte. However, in order to make this event occur, we can use the second byte of the message word inserted at iteration $k-5$ in order to randomize the instantiation of the transition. There are several ways of doing this step, and this is discussed below. We actually have a good probability to find 2^8 valid pairs of message bytes for this transition: two control bytes for one byte of condition. We repeat the process for the seventh column transition of iteration $k-4$ with the fourth byte of the message word: again two control bytes for one byte of condition. Next, we identify the subset of the cross product of the two sets of 2^8 byte pairs such that the twelfth column transitions of iteration $k-4$ are verified (depending only on the two previously fixed pairs of message bytes), which costs one byte of condition. So, we maintain 2^8 valid possibilities. Then, we fix the first byte of the message word to deal with the third column transition of iteration $k-4$: since this costs one control byte for one byte of condition, we still maintain 2^8 valid possibilities. Finally, with the remaining byte of the message word (the third), we look for a good transition for the ninth column of iteration $k-3$: this costs one control byte for two bytes of conditions but we had maintained 2^8 valid possibilities before, so that in the end we have a good probability to find a valid message word for all the transitions cited. Yet we did not take care of the eleventh column of iteration $k-4$, which costs us one byte of condition. To summarize, this whole step will cost us 2^8 tries because we had a total of six control bytes for a total of seven bytes of conditions. Repeating this reasoning for all the message words inserted at each iteration of the differential path explains the 2^{112} tries cost for the whole collision attack.

For simplicity we described an attack requiring 2^{112} memory but a simple version without memory is also possible with the same computational complexity. During the first step, instead of memorizing all the elements $\mathcal{A}_1, \ldots, \mathcal{A}_{2^{112}}$ before launching the second step, one can directly run the second step for each element $\mathcal{A}_i$, without keeping track of them.

One may argue that even if the attacker needs to try 2^{112} all-active pairs of states, the basic operation may be costly when playing with the control bytes. Indeed, with the previous example, some steps require to pass through 2^8 or 2^{16} values of message words, each requiring only a SubBytes computation on a whole column, or one or two iteration processes (depending on the column in which the state the transition occurs). Even if it is still an attack, the complexity would be slightly higher. This argument is true if the attacker uses a naive search method. However, inexpensive precomputations allow to reduce the computational cost of the search table lookups. For example, with as few as 2^{32} precomputation time and memory, one can generate all the information needed to quickly execute the search needed during the third step of the collision search. Only a few table lookups would then be required. One might also wonder why we did not count the complexity of the few 4 non-zero truncated differences to 4 non-zero truncated differences transitions. Such transitions always have a high probability to happen $P_A = (1-2^{-8})^4 \simeq 2^{-0.02}$, and therefore, they have very little effect on the complexity of the attack. Finally, the compression function mode performs 40 iterations for one compression call. Thus, our attack actually runs in less than 2^{112} hash computations, all the complexity coming from the generation of 2^{112} all-active pairs of states.

We checked that this method also works with a complexity of at most 2^{120} hash computations for all the rotation constants providing the best diffusion, which seems to indicate that the internal state of Grindahl is not large enough to ensure a proper collision resistance.

6. Possible Patches

Most of the difficulty of the presented attack is to actually find a good differential path, and this is possible by the analysis simplification induced by letting the differences totally spread and start from an all-active pair of states. Besides, even if better differential paths may be found by maintaining a low weight of differences (which seems hard to find) instead of going through an all-active pair of states, we believe that the complexity would not drastically decrease compared to our attack. In fact, the complexity cost grows quickly due to the last iterations of the differential path where very few control bytes are available, and these steps will remain very costly whatever the differential path used. Said in other words, we can compute a lower bound on the complexity of an attack using any truncated differential path and control bytes. For example, a short program tells us that a similar truncated differential attack for the 256-bit version of Grindahl requires at least 2^{104} operations (whatever the truncated differential path and the message words byte fixing schedule) even if this does not mean that such an attack exists.

Thus, it would be very interesting to think of a new version of Grindahl, with a comparable efficiency, that resists not only the presented attack but also any attack dealing with truncated differences and control bytes (and also the independent flaw identified in [18]). However, one needs some assumption on the power of the control bytes. A plausible assumption could be that a control byte can only correct a MixColumns transition located at one, two, or three iterations after its introduction. This seems a relatively weak assumption since after three iterations, a message byte affects the entire internal state of the compression function. Using no assumption at all would lead to nonsense as in this case using a path with a huge number of iterations would theoretically provide enough control bytes for the entire differential path, reducing the lower bound to 1. It is a bidimensional problem since apart from the lower bound value, with a too weak assumption the program may go through too many leaves in the search tree (whose size is reduced with an early-abort programming) and never output anything. Moreover, a too strong assumption may not model well the attacker in practice.

Finally, one wants the lower bound on the complexity of an attack using truncated differential path and control bytes to be at least 2^{128} operations, and even larger for a good security margin. If this is possible, an attacker who wants to find a collision would have to first find a differential path and then to deal with the actual values of differences in order to lower the complexity. The SubBytes transformation would, therefore, discourage this kind of attack and we would obtain a hash function with a strong security argument. A new Grindahl version with such a property and a reasonable efficiency could be, therefore, designed by adding some more columns in the states for example.

The question of the number of columns to be added or other possible patches is left open for future research. However, it is to be noted that the SHA-3 semi-finalist FUGUE [19], a direct successor of Grindahl, was built with the aim of resisting to this type of attack [39]. In particular, the designers increased the internal state size and proposed a much better diffusion layer, finally succeeding in formally proving resistance of FUGUE against the methods we presented in this article. Moreover, the SHA-3 hash function candidates AURORA [22] and LUX [35] also proposed arguments with regard to our attacks as their proposals share similarities with Grindahl. However, some other vulnerabilities were later discovered for these two candidates [11,14,15,40].

7. Conclusion and Future Work

In this article, we described a collision attack on the AES-based Grindahl hash function using a byte-wise truncated differential path. The rather small internal state size can be exploited by a counterintuitive technique in which the attacker first lets all the differences spread. Then, by a sharp message bytes fixing schedule along with the backward construction of a colliding-truncated differential path starting from an all-active internal state, one can compute a collision with not more than 2^{112} round computations for the full 256-bit version of Grindahl.

Attacking the second-preimage resistance of Grindahl using the same method seems difficult since the attacker has access to much less freedom degrees (only the difference in the message bytes can be randomized, not the values anymore) and the security claim from the designers is the same as for collision resistance, i.e., $2^{n/2}$.

Acknowledgements

The author would like to thank the designers of Grindahl (Lars Knudsen, Christian Rechberger, Søren Thomsen), Henri Gilbert, Olivier Billet, and Yannick Seurin for their valuable remarks on the attack and discussions on the Grindahl design, as well as the anonymous referees for their helpful comments. This article is the extended and updated version of an article published at ASIACRYPT 2007 [39]. The author is supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

[1] D. Augot, M. Finiasz, N. Sendrier. A family of fast syndrome based cryptographic hash functions, in E. Dawson, S. Vaudenay, editors, *Progress in Cryptology—Mycrypt 2005*. Lecture Notes in Computer Science, vol. 3715 (Springer-Verlag, Berlin, 2005), pp. 64–83

[2] M. Bellare, T. Ristenpart, Multi-property-preserving hash domain extension and the EMD transform, in X. Lai, K. Chen, editors, *Advances in Cryptology—ASIACRYPT 2006*, Lecture Notes in Computer Science, vol. 4284, (Springer-Verlag, Berlin, 2006), pp. 299–314

[3] K. Bentahar, D. Page, M-J.O. Saarinen, J.H. Silverman, N.P. Smart, LASH, in Proceedings of *Second NIST Cryptographic Hash Workshop, 2006*. Available from: www.csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm

[4] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, The Keccak SHA-3 submission, in Submission to NIST (Round 3), 2011. Available from: http://keccak.noekeon.org/Keccak-submission-3.pdf

[5] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. RadioGatun, a Belt-and-Mill hash function, in Proceedings of *Second NIST Cryptographic Hash Workshop, 2006*. Available from: www.csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm

[6] E. Biham, O. Dunkelman, A framework for iterative hash functions: HAIFA, in Proceedings of *Second NIST Cryptographic Hash Workshop, 2006*. Available from: www.csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm

[7] O. Billet, M.J.B. Robshaw, T. Peyrin, On building hash functions from multivariate quadratic equations, in J. Pieprzyk, H. Ghodosi, E. Dawson, editors, *Information Security and Privacy—ACISP 2007*, Lecture Notes in Computer Science, vol. 4586 (Springer-Verlag, Berlin, 2007), pp. 82–95

[8] J. Black, P. Rogaway, T. Shrimpton, Black-box analysis of the block-cipher-based hash-function constructions from PGV, in M. Yung, editor, *Advances in Cryptology—CRYPTO 2002*, Lecture Notes in Computer Science, vol. 2442 (Springer-Verlag, Berlin, 2002) pp. 320–335

[9] S. Contini, A.K. Lenstra, R. Steinfeld, VSH, an efficient and provable collision-resistant hash function, in S. Vaudenay, editor, *Advances in Cryptology—EUROCRYPT 2006*, Lecture Notes in Computer Science, vol. 4004 (Springer-Verlag, Berlin, 2006) pp. 165–182

[10] J. Daemen, V. Rijmen, The design of rijndael. (Springer-Verlag, Berlin, 2002)

[11] W. Dai. OFFICIAL COMMENT: LUX. NIST mailing list (local link), 2008. Available from: http://ehash.iaik.tugraz.at/uploads/e/ec/Lux_dai.txt

[12] I. Damgård, A design principle for hash functions, in G. Brassard, editor, *Advances in Cryptology—CRYPTO'89*, Lecture Notes in Computer Science, vol. 435 (Springer-Verlag, Berlin, 1989), pp. 416–427

[13] R.D. Dean, Formal aspects of mobile code security, PhD thesis. (Princeton University, Princeton, 1999)

[14] N. Ferguson, RE:OFFICIAL COMMENT:LUX. NIST mailing list (local link), 2009. Available from: http://ehash.iaik.tugraz.at/uploads/2/21/Lux_niels.txt

[15] N. Ferguson, S. Lucks, Attacks on AURORA-512 and the double-mix Merkle-Damgaard transform. Cryptology ePrint Archive, Report 2009/113, 2009

[16] T. Fuhr, T. Peyrin. Cryptanalysis of radioGatún, in O. Dunkelman, editor, *Fast Software Encryption—FSE 2009*, Lecture Notes in Computer Science, vol. 5665 (Springer-Verlag, Berlin, 2009), pp. 122–138

[17] H. Gilbert, T. Peyrin, Super-sbox cryptanalysis: improved attacks for AES-like permutations, in S. Hong, T. Iwata, editors, *Fast Software Encryption—FSE 2010*, Lecture Notes in Computer Science, vol. 6147 (Springer-Verlag, Berlin, 2010), pp. 365–383

[18] M. Gorski, S. Lucks, T. Peyrin, Slide attacks on a class of hash functions, in J. Pieprzyk, editor, *Advances in Cryptology—ASIACRYPT 2008*, Lecture Notes in Computer Science, vol. 5350 (Springer-Verlag, Berlin, 2008), pp. 143–160

[19] S. Halevi, W.E. Hall, C.S. Jutla, The hash function fugue. Submission to NIST (updated), 2009

[20] S. Hirose, Some plausible constructions of double-block-length hash functions, in M.J.B. Robshaw, editor, *Fast Software Encryption—FSE 2006*, Lecture Notes in Computer Science, vol. 4047 (Springer-Verlag, Berlin, 2006), pp. 210–225

[21] D. Hong, D. Chang, J. Sung, S. Lee, S. Hong, J. Lee, D. Moon, S. Chee, A new dedicated 256-bit hash function: FORK-256, in M.J.B. Robshaw, editor, *Fast Software Encryption—FSE 2006*, Lecture Notes in Computer Science, vol. 4047 (Springer-Verlag, 2006), pp. 195–209

[22] T. Iwata, K. Shibutani, T. Shirai, S. Moriai, T. Akishita, AURORA: A Cryptographic Hash Algorithm Family. Submission to NIST, 2008. Available from: http://ehash.iaik.tugraz.at/uploads/b/ba/AURORA.pdf

[23] J-P. Aumasson, W. Meier, R.C.-W. Phan, The hash function family LAKE, in M.J.B. Robshaw, editor, *Fast Software Encryption—FSE 2008*, Lecture Notes in Computer Science, vol. 5086, (Springer-Verlag, Berlin, 2008), pp. 36–53

[24] A. Joux, Multi-collisions in iterated hash functions, Application to cascaded Constructions, in M. Franklin, editor, Advances in cryptology—CRYPTO 2004, Lecture Notes in Computer Science, vol. 3152 (Springer-Verlag, Berlin, 2004), pp. 306–316

[25] J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor, Advances in Cryptology—EUROCRYPT 2006, Lecture Notes in Computer Science, vol. 4004 (Springer-Verlag, Berlin, 2006), pp. 183–200

[26] J. Kelsey and B. Schneier, Second preimages on n-bit hash functions for much less than 2^n work, in R. Cramer, editor, *Advances in Cryptology—EUROCRYPT 2005*, Lecture Notes in Computer Science, vol. 3494, (Springer-Verlag, Berlin, 2005), pp. 474–490

[27] D. Khovratovich, Cryptanalysis of hash functions with structures, in M.J. Jacobson Jr., V. Rijmen, R. Safavi-Naini, editors, *Selected Areas in Cryptography—SAC 2009*, Lecture Notes in Computer Science, vol. 5867, (Springer-Verlag, Berlin, 2009), pp. 108–125

[28] L.R. Knudsen, Truncated and higher order differentials, in B. Preneel, editor, *Fast Software Encryption—FSE 1994*, Lecture Notes in Computer Science, vol. 1008 (Springer-Verlag, 1995), pp. 196–211

[29] L.R. Knudsen, C. Rechberger, S.S. Thomsen, Grindahl—a family of hash functions, in A. Biryukov, editor, *Fast Software Encryption—FSE 2007*, Lecture Notes in Computer Science, vol. 4593 (Springer-Verlag, Berlin, 2007), pp. 39–57

[30] M. Lamberger, F. Mendel, C. Rechberger, V. Rijmen, M. Schläffer, Rebound distinguishers: results on the full whirlpool compression function, in M. Matsui, editor, *Advances in Cryptology—ASIACRYPT 2009*, Lecture Notes in Computer Science, vol. 5912 (Springer-Verlag, Berlin, 2009), pp. 126–143

[31] F. Mendel, T. Peyrin, C. Rechberger, M. Schläffer, Improved cryptanalysis of the reduced grøstl compression function, ECHO permutation and AES block cipher, in M.J. Jacobson Jr., V. Rijmen, R. Safavi-Naini, editors, *Selected Areas in Cryptography—SAC 2009*, Lecture Notes in Computer Science, vol. 5867 (Springer-Verlag, Berlin, 2009), pp. 16–35

[32] F. Mendel, C. Rechberger, M. Schläffer, S.S. Thomsen, The rebound attack: cryptanalysis of reduced whirlpool and grøstl, in O. Dunkelman, editor, *Fast Software Encryption—FSE 2009*, Lecture Notes in Computer Science, vol. 5665 (Springer-Verlag, Berlin, 2009), pp. 260–276

[33] A.J. Menezes, S.A. Vanstone, P.C. Van Oorschot, *Handbook of applied cryptography*, (CRC Press Inc, Boca Raton, FL, 1996)

[34] R.C. Merkle, One way hash functions and DES, in G. Brassard, editor, Advances in Cryptology—CRYPTO'89, Lecture Notes in Computer Science, vol. 435 (Springer-Verlag, Berlin, 1989), pp. 428–446

[35] I. Nikolić, A. Biryukov, D. Khovratovich, Hash family LUX—Algorithm Specifications and Supporting Documentation. Submission to NIST, 2008. Available from: http://ehash.iaik.tugraz.at/uploads/f/f3/LUX.pdf

[36] NIST. National institute of standards and technology: Advanced hash standard. Available from: www.csrc.nist.gov/pki/HashWorkshop/index.html

[37] NIST. National institute of standards and technology: FIPS 180–2: Secure hash standard, August 2002. Available from: www.csrc.nist.gov

[38] NIST. National institute of standards and technology: FIPS 197 advanced encryption standard, November 2001. Available from: www.csrc.nist.gov

[39] T. Peyrin, Cryptanalysis of grindahl, in K. Kurosawa, editor, *Advances in Cryptology—ASIACRYPT 2007*, Lecture Notes in Computer Science, vol. 4833 (Springer-Verlag, Berlin, 2007), pp. 551–567

[40] T. Peyrin, Slide attacks on LUX. NIST mailing list (local link), 2008. Available from: http://ehash.iaik.tugraz.at/uploads/6/62/Lux_peyrin.txt

[41] T. Peyrin, H. Gilbert, F. Muller, M.J.B. Robshaw, Combining compression functions and block cipher-based hash functions, in X. Lai, K. Chen, editors, *Advances in Cryptology—ASIACRYPT 2006*, Lecture Notes in Computer Science, vol. 4284 (Springer-Verlag, Berlin, 2006), pp. 315–331

[42] B. Preneel, Analysis and design of cryptographic hash functions, PhD thesis, Katholieke Universiteit Leuven, 1993

[43] B. Preneel, R. Govaerts, J. Vandewalle, Hash functions based on block ciphers: a synthetic approach, in D.R. Stinson, editor, *Advances in Cryptology—CRYPTO '93*, Lecture Notes in Computer Science, vol. 773 (Springer-Verlag, Berlin, 1993), pp. 368–378

[44] R. L. Rivest, RFC 1321: The MD5 message-digest algorithm, April 1992. Available from: http://www.ietf.org/rfc/rfc1321.txt.

[45] Y. Seurin, T. Peyrin, Security analysis of constructions combining FIL random oracles, in A. Biryukov, editor, *Fast Software Encryption—FSE 2007*, Lecture Notes in Computer Science, vol. 4593 (Springer-Verlag, Berlin, 2007), pp. 119–136

[46] X. Wang, X. Lai, D. Feng, H. Chen, X. Yu, Cryptanalysis of the hash functions MD4 and RIPEMD, in R. Cramer, editor, *Advances in Cryptology—EUROCRYPT 2005*, Lecture Notes in Computer Science, vol. 3494 (Springer-Verlag, Berlin, 2005), pp. 1–18

[47] X. Wang, Y.L. Yin, H. Yu, Finding collisions in the full SHA-1, in V. Shoup, editor, *Advances in Cryptology—CRYPTO 2005*, Lecture Notes in Computer Science, vol. 3621 (Springer-Verlag, Berlin, 2005), pp. 17–36

[48] X. Wang, H. Yu, How to break MD5 and other hash functions, in R. Cramer, editor, *Advances in Cryptology—EUROCRYPT 2005*, Lecture Notes in Computer Science, vol. 3494 (Springer-Verlag, Berlin, 2005), pp. 19–35

[49] X. Wang, H. Yu, Y.L. Yin, Efficient collision search attacks on SHA-0, in V. Shoup, editor, *Advances in Cryptology—CRYPTO 2005*, Lecture Notes in Computer Science, vol. 3621 (Springer-Verlag, Berlin, 2005), pp. 1–16