Speeding up Collision Search for Byte-Oriented Hash Functions

Dmitry Khovratovich, Alex Biryukov, and Ivica Nikolic

University of Luxembourg
{dmitry.khovratovich,alex.biryukov,ivica.nikolic}@uni.lu

Abstract. We describe a new tool for the search of collisions for hash functions. The tool is applicable when an attack is based on a differential trail, whose probability determines the complexity of the attack. Using the linear algebra methods we show how to organize the search so that many (in some cases — all) trail conditions are always satisfied thus significantly reducing the number of trials and the overall complexity.

The method is illustrated with the collision and second preimage attacks on the compression functions based on Rijndael. We show that slow diffusion in the Rijndael (and AES) key schedule allows to run an attack on a version with a 13-round compression function, and the S-boxes do not prevent the attack. We finally propose how to modify the key schedule to resist the attack and provide lower bounds on the complexity of the generic differential attacks for our modification.

1 Introduction

Bit-oriented hash functions like MD5 [17] and SHA [10] have drawn much attention since the early 1990s being the de-facto standard in the industry. Recent cryptanalytic efforts and the appearance of real collisions for underlying compression functions [19,18,4,13] motivated researchers to develop new fast primitives. Several designs were presented last years (LASH [2], Grindahl [12], RadioGatun [3], LAKE [1]) but many have already been broken [14,15,6].

While many hash functions are designed mostly from scratch one can easily obtain a compression function from a block cipher. Several reliable constructions, such as so-called Davies-Meyer and Miyaguchi-Preneel modes, were described in a paper by Preneel et al. [16].

In this paper we present a method for speeding up collision search for byte-oriented hash functions. The method is relatively generic, which is an advantage in the view that most collision attacks exploit specific features of the internal structure of the hash function (see, e.g., the attack on Grindahl [15]) and can be hardly carried on to other primitives.

Most of the collision attacks are differential in nature. They consider a pair of messages with the difference specified by the attacker and study propagation of this difference through the compression function — a *differential trail*. The goal of a cryptanalyst is to find a pair of messages that follows the trail (a conforming pair). Our idea is to deal with fixed values of internal variables as sufficient conditions for the trail to be followed. We express all internal transformations as equations and rearrange them such that one can quickly construct a pair of executions that fits the trail.

We illustrate the method with a cryptanalysis of AES [7] in the Davies-Meyer mode. The AES block cipher attracted much attention and was occasionally considered as a basis for a compression function (mostly unofficially, though some modes were proposed [5] and AES-related hash designs were also investigated [12,11]).

This paper is organized as follows. In the next section we give an idea and formal description of the algorithm. Then in Section 3 we show how to find collisions, semi-free-start collisions and second preimages for the compression functions based on the versions of Rijndael. Section 4 is devoted to the properties of the Rijndael internal transformations, which are weaknesses in the hash function environment. Finally, we propose a modification to the original key schedule, which prevents our attack, and provide some lower bounds for attacks based on the differential techniques. The resulting hash function, which we call Cheetah, is formally introduced in Appendix.

2 Idea in Theory

State of the art. Most of the recent attacks on compression functions deal with a *differential trail*. Informally, a trail is a sequence of pairs of internal states with a restriction on the contents. An adversary looks for the pair of messages that produces such states in each round.

More formally, suppose that the compression function takes the initial value IV and the message M as an input and outputs the hash value H. The whole transformation is usually defined as a sequence of smaller transformations — rounds. Then the execution of a k-round compression function looks as follows:

$$IV \xrightarrow{f(M_1,\cdot)} S_1 \xrightarrow{f(M_2,\cdot)} S_2 \cdots S_{k-1} \xrightarrow{f(M_k,\cdot)} H,$$

where f is a round function of two arguments: the *message block* M_i and the current *internal state* S_{i-1}. The message blocks are a result of the message schedule — a transformation of the original message. A *collision* is a pair of messages (M, M') producing the same hash value H given the initial value IV.

In the collision search the exact contents of internal states are not important; the only conditions are the fixed IV and the coincidence of resulting hash values. Thus an adversary considers a pair of executions where the intermediate values are not specified. However, a naive collision search for a pair of colliding messages would have complexity $2^{n/2}$ queries (with the help of birthday paradox) where n is the bit length of the hash value. Thus the search should be optimized.

In the differential approach an adversary specifies the difference in message blocks M_i and internal states S_i. A pair of executions can be considered as the execution that deals with the differences: it starts and ends with a zero difference, and some internal differences are also specified.

A *differential trail* is a set of conditions on the pair of executions. We assume that this set of conditions is final, i.e. the attacker use them as is and tries message pairs one by one till all the conditions are satisfied. A trail may include non-differential conditions, ex. constraints on specific bits, in order to maximize the chances for the trail to be fulfilled. The complexity of the attack with a trail is then defined by the probability that a message pair produce an execution that fits the trail. This probability is determined by the nonlinear components (e.g. S-boxes) that affect the propagation of the difference.

First idea. As we pointed out before, an adversary may try all the possible pairs and check every condition. He may also strengthen a condition by *fixing* the input value (or the output) of a non-linear function. Then in each trail he checks whether the value is as specified. Thus to find a collision it would be enough to build an execution such that the specified values follow the trail, and the second execution will be derived by adding the differences to the first one. Our algorithm (given below) deals with this type of trails.

Our improvement. Our goal is to carry out the message trials so that many conditions are always satisfied. In such case the complexity of the attack is determined by the conditions that we do not cover. Before we explain how our algorithm works we introduce the notion of free variables.

First, we express all the transformations as equations, which link the internal variables. Variables refer to bits or bytes/words depending on the trail. Secondly, notice that the IV and the message fully define all the other variables and thus the full execution. We call *free variables*[1] a set of variables that completely and computationally fast define the execution. If some variables are pre-fixed the number of free variables decreases.

The idea of our method is to build a set of free variables provided that some variables are already fixed. The size of such a set depends on how many variables are fixed. The latter value also defines the applicability of our method. The heart of our tool is an algorithm for the search of free variables. It may vary and be combined with some heuristics depending on the compression function that it is applied for, but the main idea can be illustrated on the following example.

Example 1. Assume we have 7 byte variables s, t, u, v, x, y, and z which are involved in the following equations:

$$\begin{aligned}
&F(x \oplus s) \oplus v = 0;\\
&G(x \oplus u) \oplus s \oplus L(y \oplus z) = 0;\\
&v \oplus G(u \oplus s) = 0;\\
&H(z \oplus s \oplus v) \oplus t = 0;\\
&u \oplus H(t \oplus x) = 0.
\end{aligned}$$

[1] Recall the Gaussian elimination process. After a linear system has been transformed to the row echelon form all the variables are divided into 2 groups: bound variables and free variables. Free variables are to be assigned with arbitrary values; and bound variables are derived from the values of free variables.

where F, G, H, and L are bijective functions. Note that y is involved in only one equation so it can be assigned after all the other variables have been defined. Thus we temporarily exclude the second equation from the system. Then note that z is involved in only one equation among the remaining ones, so we again exclude the equation from the system. This Gaussian-like elimination process leads to the following system:

$$\begin{array}{rl}
F(y \oplus z) \oplus L(u \oplus x) \oplus s &= 0; \\
z \oplus H^{-1}(t) \oplus v \oplus s &= 0; \\
t \oplus H^{-1}(u) \oplus x &= 0; \\
u \oplus G^{-1}(v) \oplus s &= 0; \\
v \oplus F(x \oplus s) &= 0.
\end{array}$$

Evidently, x and s can be assigned randomly and fully define the other three variables. Thus x and s are free variables. Varying them we easily get other solutions.

Now assume that the variable u is pre-fixed to a value a. Then the system is transformed in a different way:

$$\begin{array}{rl}
F(y \oplus z) \oplus L(x \oplus a) \oplus s &= 0; \\
z \oplus H^{-1}(t) \oplus v \oplus s &= 0; \\
t \oplus x \oplus H^{-1}(a) &= 0; \\
x \oplus F^{-1}(v) \oplus s &= 0; \\
G^{-1}(v) \oplus a \oplus s &= 0.
\end{array}$$

Here only one variable — s — is free.

Now we provide a more formal description of the algorithm.

1. Build a system of equations based on the compression function. The values defined by the trail are fixed to constants.
2. Mark all the variables and all the equations as non-processed.
3. Find the variable involved in only one non-processed equation. Mark the variable and the equation as processed. If there is no such variable — exit.
4. If there exist non-processed equations go to Step 3.
5. Mark all non-processed variables as free.
6. Assign random values to free variables and derive variables of processed variables.

Depending on the structure of the equations, some heuristics can be applied at step 3. For example, if there are many linear equations, real Gaussian elimination can be applied. If there are terms of degree 2, one variable can be fixed to 0, and so on.

When the algorithm can be applied. If there is no restriction on the internal variables, the algorithm always succeeds: the message variables can be taken as free. As soon as we fix some internal variables we have fewer options for choosing free variables. In terms of block cipher based compression functions, we say that the more active S-boxes we have the fewer free variables exist.

The main property of the compression function that affects the performance of the algorithm is diffusion. The slower diffusion is, the more rounds can be processed by the algorithm. As we show in Section 3, a slow diffusion in the message schedule can be enough to maintain the attack.

This algorithm is not as universal as other algorithms dealing with non-linear equations: SAT-solver based and Gröbner basis based. However, if it works a cryptanalyst can generate a number of solutions in polynomial time while generic algorithms have exponential complexity. This is a real benefit, since we can use the algorithm at the top or at the bottom part of the trail, thus increasing probability of a solution.

Equation properties. There is a desired property of equations: each variable should be uniquely determined by the other ones. If this is not the case (fixing all but one variable may not give a bijection) then the last step of the algorithm becomes probabilistic (some values of free variables do not lead to the solution) or, on the contrary, some variables can be assigned by one of a few values. We can also emphasize not a single variable but a group of variables if it is fully determined by the other variables involved in the equation.

We conclude that under these assumptions the exact functions that link variables do not matter. The only requirement is that they can be easily inverted, which is typically true for the internal functions of a block cipher or a hash function. If no heuristics which mix rows are applied then the algorithm does not need the information about the non-linear functions, only the variables that are involved in. Thus we consider not a system of equations but a *matrix of dependencies* where rows correspond to equations, and columns to variables. The following matrix represents the system from Example 1:

$$\text{Before triangulation:} \begin{pmatrix} \begin{array}{ccccccc} s & t & u & v & x & y & z \\ \hline 1&0&0&1&1&0&0 \\ 1&0&1&0&1&1&1 \\ 1&0&1&1&0&0&0 \\ 1&1&0&1&0&0&1 \\ 0&1&1&0&1&0&0 \end{array} \end{pmatrix}. \qquad \text{After:} \begin{pmatrix} \begin{array}{ccccc|cc} y & z & t & u & v & x & s \\ \hline 1&1&0&1&0&1&1 \\ 0&1&1&0&1&0&1 \\ 0&0&1&1&0&1&0 \\ 0&0&0&1&1&0&1 \\ 0&0&0&0&1&1&1 \\ &&&&& \multicolumn{2}{c}{\substack{\textit{free} \\ \textit{variables}}} \end{array} \end{pmatrix}.$$

Comparison to other methods. Several tools that reduce the search cost by eliminating some conditions were recently proposed. They are often referred to as *message modification* (a notion introduced by Wang in attacks on SHA) though there is often no direct "modification". The idea is to satisfy conditions by restricting internal variables to pre-fixed values and trying to carry out those restrictions from internal variables to message bits, which are controlled.

Compared to message modification and similar methods, our algorithm may give a solution even if the restrictions can not be carried out to message bits directly. If a system of equations is solved, the algorithm produces one or many solutions that satisfy the restrictions. Even if all the restrictions can not be processed then one may try to solve the most expensive part. Thus we expect our method to work in a more general and automated way compared to the dedicated methods designed before.

3 Idea in Practice

We illustrate our approach with the cryptanalysis of a hash function based on Rijndael [7] in the Davies-Meyer mode. The security of Rijndael as a compression function has been frequently discussed in both official and non-official talks though no clear answer was provided in favour of or against such a construction. Additionally, the exact parameters of Rijndael as a hash function are a subject of a discussion.

The Davies-Meyer mode has been chosen due to its message length/block length ratio, which is crucial for the performance. Assume we want to construct a compression function with performance comparable to SHA-1. The AES block length of 128 bits is too small against birthday attack so 160 bits would be the minimal admissible value. The size of a message block to be hashed should be also increased in order to achieve better performance. However, there should be a tradeoff between the message length and the number of rounds. A simple solution is to take the message block equal to 2 internal blocks (320 bits). The 14-round Rijndael-based construction gives us performance comparable to SHA-1. We will concentrate on this set of parameters though other ones will be also pointed out.

3.1 Properties of Rijndael Transformations. How to Build a Trail

Rijndael is surprisingly suitable for the analysis with our method due to simplicity of its operations and properties of its S-boxes. A differential trail provides a set of active S-boxes. Due to the special differential properties of Rijndael S-boxes (2^{-6} maximal differential probability) the number of possible input/output values is limited to not more than 4 possibilities. We take one of the few values of an S-box input that provides the propagation of differences as a sufficient condition. We found a 12-round trail (Figure 4) which has 50 active S-boxes (44 in the `SubBytes` transformations and 6 in the `KeySchedule`). However, most of the active S-boxes are in the upper part of the trail, which allows us to use the algorithm.

The crucial weakness of the Rijndael key schedule, which is exactly the message schedule procedure in the considered compression function, is the XOR operation that produces columns of the next subkey. It provides a good diffusion as a key schedule, which was the goal of the Rijndael design, but is not adapted for the use in a compression function, where all the internal variables are known to an adversary.

The KeySchedule transformation for a key of size 256 bits and more is given by the following expressions:

$$\begin{aligned}
k_{i,0} &\leftarrow S(k_{i+1,NK-1}) \oplus C_r, && 0 \le i \le 3;\\
k_{i,j} &\leftarrow k_{i,j-1} \oplus k_{i,j}, && 0 \le i \le 3,\ 1 \le j \le 3;\\
k_{i,4} &\leftarrow S(k_{i,3}) \oplus k_{i,4}, && 0 \le i \le 3;\\
k_{i,j} &\leftarrow k_{i,j-1} \oplus k_{i,j}, && 0 \le i \le 3,\ 5 \le j \le NK-1,
\end{aligned} \tag{1}$$

where $S()$ stands for the SubBytes transformation, and C_r — for the round-dependant constant. It is easy to check that the first byte in a row affects all the other bytes in the row, so that any difference will propagate through all the xor operations. NK is a parameter equal to 8 for a 256-bit key. However, the KeySchedule transformation is invertible, and its inversion has a slow diffusion. This is the fact that we exploit. More precisely, the formulas for the inversion are as following:

$$\begin{aligned}
k_{i,j} &\leftarrow k_{i,j-1} \oplus k_{i,j}, && 0 \le i \le 3,\ NK-1 \ge j \ge 5,\\
k_{i,4} &\leftarrow S(k_{i,3}) \oplus k_{i,4}, && 0 \le i \le 3;\\
k_{i,j} &\leftarrow k_{i,j-1} \oplus k_{i,j}, && 0 \le i \le 3,\ 1 \le j \le 3;\\
k_{i,0} &\leftarrow S^{-1}(k_{i+1,NK-1} \oplus C_r), && 0 \le i \le 3.
\end{aligned} \tag{2}$$

We build two trails: for 12 rounds and for 7 rounds. We use a local collision illustrated in Figure 1 as a base for both of them. There, a one byte difference is injected by the AddRoundKey transformation and spread to 4-byte difference after MixColumns. The 4-byte difference is canceled out by the next AddRoundKey. Due to a long message block both differences can be arranged into different columns. The 4-byte difference is fully determined by the contents of the one active S-box. We mark this value by a in Figure 1. It is a *sufficient* condition for the local collision.

If we start with this pattern and go down all the bytes of the message block will likely have the difference. However, the backward propagation is much different. We can build a 7-round trail with only 9 active S-boxes (Figure 5). In order to build a longer trail we swap the left and the right halves of the message block and use some ad-hoc tricks in the first rounds. As a result, we obtain 12-round trail with 50 active S-boxes (Figure 4).

3.2 Collisions, Second Preimages and the Matrix of Dependencies for the Rijndael-Based Hash

Matrix of Dependencies. First we explain in details our usage of variables and equations. We consider *byte* variables: the IV (4*NB variables[2]), the output (4*NB), the message (4*NK per message schedule round), the internal states. We deal with two internal states per round: after the SubBytes transformations and after the MixColumns transformation. Thus we obtain 8*NB variables per round. The equations are derived from the following transformations:

[2] NB is the number of columns in the internal state. NB is equal to 8 in Rijndael-256.

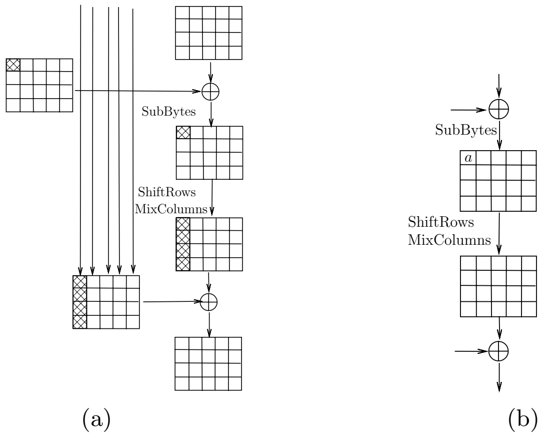

Fig. 1. Local collision: non-zero differences (a) and fixed values (b)

I `SubBytes`∘`AddRoundKey` transformations: 4*NB equations in each internal round;
II `MixColumns`∘`ShiftRows` transformations: NB equations in each internal round;
III `KeySchedule` transformations: 4*NK equations in each schedule round.

The `MixColumns` transformation is actually a set of 4 linear transformations. In the latter ones any 4 of 5 variables uniquely determine the other one. For the `MixColumns` transformation as a whole a more complicated property holds: any 4 of 8 variables are determined by the other ones.

The variables that are predefined by a trail are substituted into the equations and are not considered in the matrix.

320/160. 5 rounds. The simplest challenge is to build a 5-round collision for the Rijndael-based hash with 320-bit message block and 160-bit internal state. The trail is derived by removing the first two rounds from the 7-round trail (Figure 5). There are 5 active S-boxes, which fix 5 of the 320 internal variables. There are 225 equations. The resulting matrix of dependencies is presented in Figure 2. The non-zero elements are color pixels with green ones representing the `MixColumns` transformation.

The value of the one-byte difference is chosen randomly as well as that of the active S-box. Let us denote the one-byte difference by δ and the input to the active S-box by a. Then the 4-byte difference is the `MixColumns` matrix M multiplied by $(S(a) + S(a + \delta), 0, 0, 0)$.

The matrix is easily triangulated (Figure 3). We obtain 55 free variables. Any assignment of those variables and 5 fixed S-box inputs fully determine the IV and the message.

320/160. 7 rounds. The trail in Figure 5 is a 7-round collision trail with 9 active S-boxes. Although the triangulation algorithm can not be directly applied

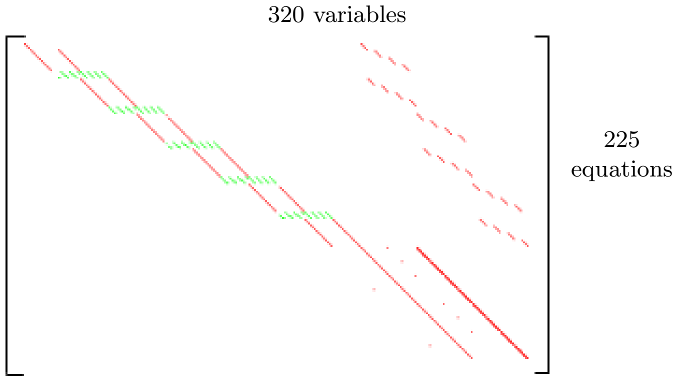

Fig. 2. The matrix of dependencies for the 5-round trail before the triangulation

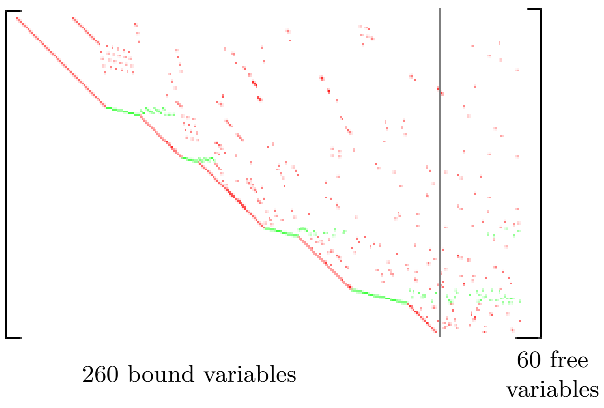

Fig. 3. The matrix of dependencies after the triangulation

to the resulting matrix, we can run it for the first 5 rounds, which contain 8 of 9 active S-boxes. Since we have several free variables we are able to generate many colliding pairs. About one of 2^7 pairs satisfy the condition on the one remaining active S-box so we repeat the last step of the algorithm 2^7 times and obtain a 7-round collision. The exact colliding messages are presented in Table 1.

320/160. 12 rounds. A 12-round trail is presented in Figure 4. We swap the left and the right halves of the message block and use some ad-hoc tricks in the first rounds. As a result, we obtain a 12-round trail with 50 active S-boxes with only 6 of them in `KeySchedule` transformations.[3]

[3] The most of non-zero columns in the differences between message blocks are of the form $(a, 0, 0, 0)$ or (b, c, d, e) where a, b, c, d and e are the same in all message blocks. Those values are that are used for obtaining a local collision (Figure 1). They are marked as grey cells in Figures 5 and 4. If they interleave some other values are produced. The latter ones are marked as olive cells.

Table 1. 7-round collision for the Rijndael-based compression function. Message bytes with different values are emphasized.

IV	b8 29 68 d1 f8	Message 1	**77 e6 a7 1e** e3 **40 e6 ef 56** 26
	5b d0 01 bd 17		7e 1b aa 2b fa **44 70 88 66** 0c
	05 83 8a 43 4b		04 2b 7b e1 d6 **df 4d 09 52** 5c
	40 40 9e 0c c5		4a 81 31 98 b6 **df 67 79 c6** ab
Output	83 e5 06 a4 46	Message 2	**76 e7 a6 1f** e3 **42 e4 ed 54** 26
	5f e7 7c ba 49		7e 1b aa 2b fa **45 71 89 67** 0c
	8e 7d 1e bd 96		04 2b 7b e1 d6 **de 4c 08 53** 5c
	b8 d4 e3 e9 a0		4a 81 31 98 b6 **dc 64 7a c5** ab

The trail is too long to be processed by the triangulation algorithm directly. Instead we fix not all S-box inputs. More precisely, we fix the 6 variables that enter the active S-boxes in the first three `KeySchedule` transformations (actually all active S-boxes in the message scheduling) and the 35 variables that are the outputs of active S-boxes in first 4 rounds. There are 9 active S-boxes left unfixed. We have 11 free variables and generate $2^{7*9} = 2^{63}$ colliding pairs so that one of them pass through those 9 S-boxes and gives the 12-round collision. The resulting complexity is 2^{63} compression function calls.

Fixed IV. So far we considered that the IV is constant but can be freely chosen, mainly because we do not attack an already existing standard or a particular proposal. Nevertheless, compression functions with a similar structure, which may be designed later, would require an attack with the fixed IV.

The algorithm described before may be easily adapted to this case. We just mark all the input variables in the trail as pre-fixed, which is equivalent to just the removal of the corresponding columns from the matrix of dependencies. The number of equations is not changed so the probability of successful triangulation can only decrease, not increase. This is the case: now we are not able to reduce the matrix for the 5-round trail, but for the 3-round one we can still do this. This fact does not imply that the 3-round collisions is the maximum achieved level. Actually we just bypass the next two rounds with some probability. If the number of active S-boxes in the trail after these 3 rounds is not large, this probability may still be reasonable.

For example, we can use the trail for 7-rounds collision and process by the algorithm only first three rounds. Then we have to bypass through 3 active S-boxes, which requires about 2^{21} evaluations of the compression function and can be done in real time.

512/256. If we just increase the hash length keeping the message/hash ratio we actually get a much weaker compression function.

For example, a differential trail for 13 rounds with no active S-boxes in the message scheduling can be easily built from the trail in Figure 5. The matrix triangulation algorithm works for 7 rounds, and a 13-round collision can be found after 2^{35} computations of the compression function, which is substantially faster than the birthday attack.

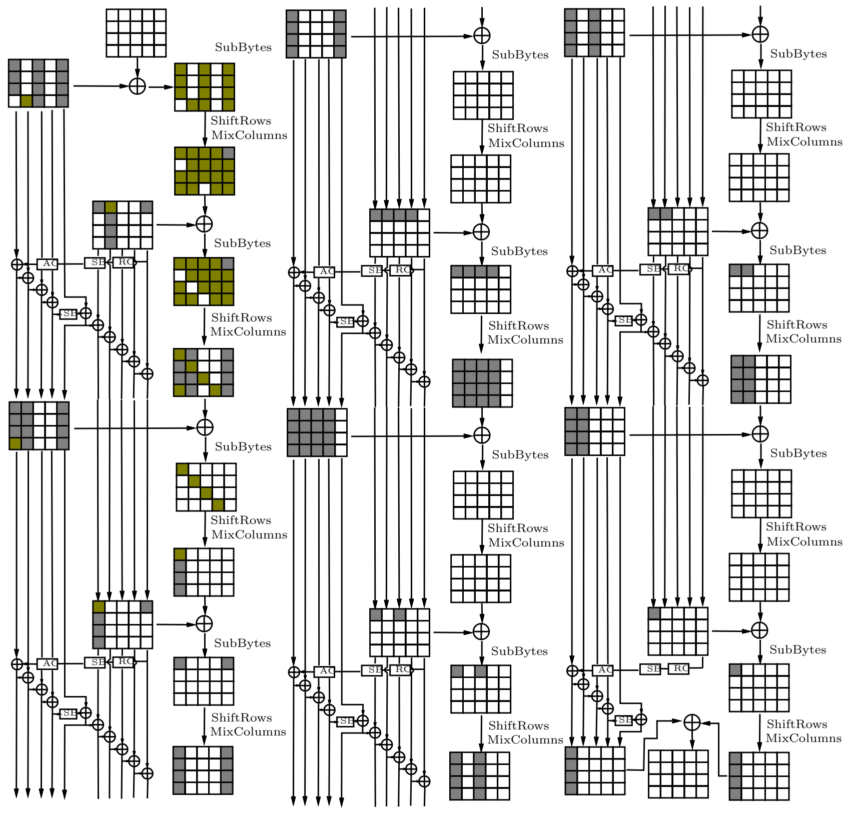

Fig. 4. 12-round differential trail for the RIJNDAEL-based compression function with the 320-bit message block and the 160-bit internal state (RIJNDAEL-hash 320/160)

3.3 Second Preimage Attack

320/160. Here we assume that the message is fixed, the IV is constant but not fixed and we have to find a message such that it produces the same hash value as the first one. Our goal is to obtain a second preimage faster than for 2^{160} calls. We just take any trail such that the conditions on the message variables do not confuse with the pre-fixed values. For example, the 7-round trail (Figure 5) do not impose such restrictions on message variables.

We mark all the message variables as fixed and run the triangulation algorithm on first three rounds. We obtain $60 - 40 - 6 = 14$ variables that can be assigned randomly. We generate 2^{21} pairs (IV, second message) so that one of them passes the three other active S-boxes in rounds 4-7. The resulting complexity of the second-preimage search is about 2^{21} compression function calls.

Although we have a longer collision trail (Figure 4), it can not be used because the number of active S-boxes is bigger than the number of the degrees of freedom.

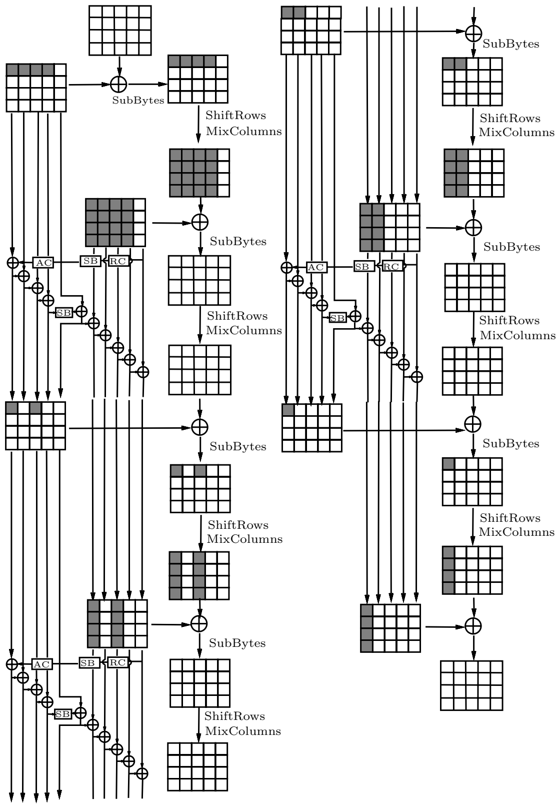

Fig. 5. 7-round differential trail for Rijndael-hash 320/160

4 Rijndael Properties That May Lead to Weaknesses in Compression Functions

Here we summarize the properties of Rijndael that allow us to attack Rijndael-based compression functions.

Let us first look at the Rijndael key schedule ((1), (2)). The weakness that we exploited is a non-symmetric diffusion. More precisely, one byte in block i affects only two or three bytes in block $i-1$. Furthermore, one can build a trail in a key schedule without active S-boxes for NK$-$4 schedule rounds. Full diffusion in key schedule may take up to 4*NK schedule rounds if we consider one-byte difference in a corner byte and proceed backwards.

Even active S-boxes in a trail give some additional power to an adversary. Due to the differential properties of the Rijndael S-box non-zero difference Δa can be converted to any of about 127 differences Δb; only half of differences can not be reached. The exact value of the output difference is guaranteed by the value of the S-box input variable.

The derivation of unknown internal variables from the known ones is also easier than that in the bit-oriented hash functions such as the SHA-family. In Rijndael if we know two of three variables in the bitwise addition or four of eight in `MixColumns` then we can uniquely determine the other variables. We would not be able to do this if we had equations of type $x_1x_2 = x_3$. So we claim that equations in Rijndael are actually pseudo-linear rather that non-linear. Though we do not know how to exploit this fact in attacks on Rijndael as a block cipher, it is valuable if we consider a Rijndael-based compression function.

Finally we note that our attacks are only weakly dependant on some Rijndael parameters such as actual S-box tables, the `MixColumns` matrix coefficients and `ShiftRows` rotation values. For example, we only need a row that is not rotated by the `ShiftRows` but it does not matter matter where it is exactly located.

5 Modification to the Message Schedule — Our Proposal

In this section we propose an improvement to the Rijndael message schedule, which prevents the low-weight trails that were shown above.

The key idea is to use primitives providing good diffusion. The Rijndael round function was designed so that no low-weight trails can be built for the full cipher. We propose to use a modified version of this round function in the message schedule for future Rijndael-based hash functions.

First, we significantly extend the size of the message block that is processed by one call of the compression function. Secondly, 256 bits is going to be the main digest size for SHA-3 [9]. A 1024-bit message block combined with a 256-bit internal state give a good security/performance tradeoff, which is justified below.

The message block is treated as a 8×16 byte square and passes through 3 iterations of a round function. Like that of Rijndael, the round function we propose is a composition of the `SubBytes`, the `ShiftRows`, and the `MixColumns` transformations. While S-boxes remain the same, the `ShiftRows` and `MixColumns` operations are modified in order to get a maximal possible diffusion. We propose to use the following offset table and the `MixColumns` matrix:

The matrix A, which is modified version of the matrix used in Grindahl [12], is an MDS-matrix.

Table 2. `ShiftRows` and `MixColumns` parameters for a new message schedule proposal

Index	Offset	Index	Offset
0	0	4	5
1	1	5	6
2	2	6	7
3	3	7	8

$$A = \begin{pmatrix} 02 & 0c & 06 & 08 & 01 & 04 & 01 & 01 \\ 01 & 02 & 0c & 06 & 08 & 01 & 04 & 01 \\ 01 & 01 & 02 & 0c & 06 & 08 & 01 & 04 \\ 04 & 01 & 01 & 02 & 0c & 06 & 08 & 01 \\ 01 & 04 & 01 & 01 & 02 & 0c & 06 & 08 \\ 08 & 01 & 04 & 01 & 01 & 02 & 0c & 06 \\ 06 & 08 & 01 & 04 & 01 & 01 & 02 & 0c \\ 0c & 06 & 08 & 01 & 04 & 01 & 01 & 02 \end{pmatrix}$$

This round function provides full diffusion after 3 rounds thus giving 4096 bits to inject to the internal state. Since the internal state is 16 times smaller, we propose to increase the number of rounds to 16. The resulting compression function is more formally introduced in Appendix.

Resistance to attacks. We did not manage to find a good trail for the resulting compression function so we can not apply our attack. Furthermore, now we give arguments supporting that low-weight trails are impossible in the resulting design: we give a lower bound on the number of active S-boxes in such a trail.

We consider a 16-round trail which starts and ends with a zero-difference state. Let us denote the number of non-zero differences in the internal state before the `SubBytes` transformation by s_i, $1 \leq i \leq 16$. The last s_i is equal to zero. Let us also denote by c_i the number of non-zero differences in the internal after the internal `MixColumns` transformation. The last c_i is equal to 0 as well. Finally, we denote by m_i the number of non-zero differences in the round message block that is xored to the internal state. These differences either cancel non-zero differences in the internal state or create them. Thus the following condition holds

$$s_i + c_{i-1} \geq m_i \tag{3}$$

Due to the branch number of the internal `MixColumns` transformation c_i is upper bounded: $c_i \leq 4s_i$. Thus we obtain the following:

$$s_i + 4s_{i-1} \geq m_i \;\Rightarrow\; \sum_i s_i + \sum_i 4s_{i-1} \geq \sum_i m_i \;\Rightarrow\; S \geq \frac{M}{5},$$

where S is the number of active S-boxes in the internal state of the compression function, and M is the number of non-zero byte differences in the expanded message.

Now we estimate the minimum number of non-zero byte differences in the message scheduling only. First we note that this number is equal to the number of the active S-boxes in the message scheduling extended to 4 round. Such a 4-round transformation is actually a Rijndael-like block cipher, which can be investigated using the theory of the wide trail design by Daemen and Rijmen [8].

Daemen and Rijmen estimated the minimum number of active S-boxes in 4 rounds of a Rijndael-like block cipher (Theorem 3, [8]). The sufficient condition to apply their theorem is that the ShiftRows should be diffusion optimal: bytes from a single column should be distributed to different columns, which is the case. Thus the number of active S-boxes can be estimated as the square of the branch number of the 8×8 `MixColumns` matrix, which is equal to 9. As a result, any pair of different message blocks has difference in at least $M = 81$ bytes of ExpandedBlock. This implies the lower bound 17 for S. Thus we obtain the following proposition.

Proposition 1. *Any collision trail has at least 17 active S-boxes in the internal state.*

Thus any attack using such minimal trail as is would be only slightly faster than the birthday attack. However, we expect that the values of M even close to minimal do not give collision trails due to the following reasons:

– Small number of active S-boxes in the internal state implicitly assumes many local collisions;
– The distribution of non-zero differences in the message scheduling is not suitable for local collisions due to high diffusion;
– The `MixColumns` matrix in the message scheduling differs from that of the internal transformation so, e.g., 4-byte difference collapse to 1-byte difference via only one of two transformations.

6 Conclusions and Future Directions

We proposed the triangulation algorithm for the efficient search of the message pairs that fit a differential trail with fixed internal variables. We illustrated the work of the algorithm by applying it to Rijndael in the Davies-Meyer mode with different parameters. Although the trails that we built contain many active S-boxes, the task of the search for a message pair becomes much easier with our algorithm. It allows to build message pairs that satisfy subtrails of an original trails. Such subtrail can be chosen in order to minimize the number of active S-boxes in the other part of the trail.

In Table 3 we summarize our efforts on building collisions and preimages for Rijndael-based compression functions.

Table 3. Summary of attacks

Hash length	Message length	Rounds	Compl.	Type of a collision
160	320	7	2^7	Full collision
160	320	12	2^{63}	Full collision
160	320	7	2^{21}	Second preimage
256	512	13	2^{35}	Full collision

We also investigated why Rijndael as a compression function is vulnerable to collision attacks. We showed how the non-symmetric diffusion in the message schedule allows to build long differential trails.

As a countermeasure, we propose a new version of the message schedule for the Rijndael-based compression functions and provide lower bounds for the probability of differential trails for the resulting function.

Acknowledgements

The authors thank anonymous reviewers for their valuable and helpful comments. Dmitry Khovratovich is supported by PRP "Security & Trust" grant of the University of Luxembourg. Ivica Nikolić is supported by the BFR grant of the FNR.

References

1. Aumasson, J.-P., Meier, W., Phan, R.C.-W.: The hash function family LAKE. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 36–53. Springer, Heidelberg (2008)
2. Bentahar, K., Page, D., Saarinen, M.-J.O., Silverman, J.H., Smart, N.: LASH, Tech. report, NIST Cryptographic Hash Workshop (2006)
3. Bertoni, G., Daemen, J., Peeters, M., van Assche, G.: Radiogatun, a belt-and-mill hash function (2006), `http://radiogatun.noekeon.org/`
4. De Cannière, C., Rechberger, C.: Finding SHA-1 characteristics: General results and applications. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 1–20. Springer, Heidelberg (2006)
5. Cohen, B.: AES-hash, International Organization for Standardization (2001)
6. Contini, S., Matusiewicz, K., Pieprzyk, J., Steinfeld, R., Jian, G., San, L., Wang, H.: Cryptanalysis of LASH. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 207–223. Springer, Heidelberg (2008)
7. Daemen, J., Rijmen, V.: AES proposal: Rijndael, Tech. report (1999), `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`
8. Daemen, J., Rijmen, V.: The wide trail design strategy. In: IMA Int. Conf., pp. 222–238 (2001)
9. Cryptographic hash project, `http://csrc.nist.gov/groups/ST/hash/index.html`
10. FIPS 180-2. secure hash standard (2002), `http://csrc.nist.gov/publications/`
11. International Organization for Standardization, The Whirlpool hash function. iso/iec 10118-3:2004 (2004)
12. Knudsen, L.R., Rechberger, C., Thomsen, S.S.: The grindahl hash functions. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 39–57. Springer, Heidelberg (2007)
13. Manuel, S., Peyrin, T.: Collisions on SHA-0 in one hour. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 16–35. Springer, Heidelberg (2008)
14. Matusiewicz, K., Peyrin, T., Billet, O., Contini, S., Pieprzyk, J.: Cryptanalysis of FORK-256. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 19–38. Springer, Heidelberg (2007)
15. Peyrin, T.: Cryptanalysis of Grindahl. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 551–567. Springer, Heidelberg (2007)
16. Preneel, B., Govaerts, R., Vandewalle, J.: Hash functions based on block ciphers: A synthetic approach. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 368–378. Springer, Heidelberg (1994)
17. Rivest, R.L.: The MD5 message-digest algorithm, request for comments (RFC 1320), Internet Activities Board, Internet Privacy Task Force (1992)
18. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
19. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)

A Hash Function Cheetah-256

Now we formally introduce a hash function proposal, which is based on the ideas discussed in Section 5. Due to space restrictions we limit ourselves to the design of the compression function.

The Cheetah compression function is an iterative transformation based on the Rijndael block cipher. The 128-byte message block is expanded to a 512-byte block by the message schedule. The internal state is of size 32 bytes and is iterated for 16 rounds. The output hash value is 32 bytes (256 bits).

The message block is expanded by the means of the message schedule. The resulting block is divided into 16 vectors, which are xored to the internal state before every round. The Cheetah compression function is then defined by the following pseudo-code:

```
CheetahCompression(IntermediateHashValue, MessageBlock) {
    InternalState = IntermediateHashValue;
    ExpandedBlock = MessageExpansion(MessageBlock);
    for(i=1; i<= 16; i++)
    {
        InternalState +=RoundBlock(ExpandedBlock,i);
        InternalState = InternalRound(InternalState);
    }
    return InternalState;
}
```

The procedures *MessageExpansion*, *RoundBlock*, and *InternalRound* are determined below.

Message Schedule. The *MessageExpansion* procedure is a Rijndael-like transformation, which is defined in pseudocode as follows:

```
MessageExpansion(byte MessageBlock[128]) {
    byte ExpandedBlock[512];
    ExpandedBlock[0..127] = MessageBlock;
    for(i=1; i<=3; i++)
    {
        SubBytes(MessageBlock);
        ShiftRows8(MessageBlock);
        MixColumn8(MessageBlock);
        AddRoundConstant(MessageBlock,i);
        ExpandedBlock[128*i..128*i+127] = MessageBlock;
    }
}
```

The *SubBytes* transformation is the byte-wise SubBytes transformation used in Rijndael. The `ShiftRows8` and the `MixColumns8` operation parameters were given in Table 2.

The *AddRoundConstant* operation adds a 32-bit constant to the message block. The constant is a function of the round index r:

$$m_{i,0} = S[4 * r + i],\ 0 \leq i \leq 3,$$

where S stands for the S-box.

The *RoundBlock* operation selects a 32-byte block from ExpandedBlock = (E_0, E_1, E_2, E_3). Define the round index r as $r = 4l + m$, $0 \leq l, m \leq 3$. Then the selected block is the 4×8 byte array M_r, that is defined as follows:

$$M_r = (m_{i,j})^{4\times 8},\ E_l = (e_{i,j})^{8\times 16};$$
$$m_{i,j} = e_{4*(m\%2)+i,4*(m/2)+j}.$$

The selected block is bytewise xored to the InternalState: $a_{i,j}^{\text{new}} \leftarrow a_{i,j} + m_{i,j}$.

Internal round. The *InternalRound* transformation is actually the Rijndael round as it would be used with 32-byte block. It consists of three operations: `SubBytes`, `ShiftRows`, and `MixColumns`.

```
InternalRound(byte InternalState[256]) {
    SubBytes(InternalState);
    ShiftRows4(InternalState);
    MixColumn4(InternalState);
}
```

The `SubBytes` operation has already been defined above. Both the `ShiftRows` and `MixColumns` operations treat the InternalState as a byte array of size 4×8, with 4 rows and 8 columns.

Parameters of the `ShiftRows` and the `MixColumns` transformations are the same as that of Rijndael-256 (Table 4).

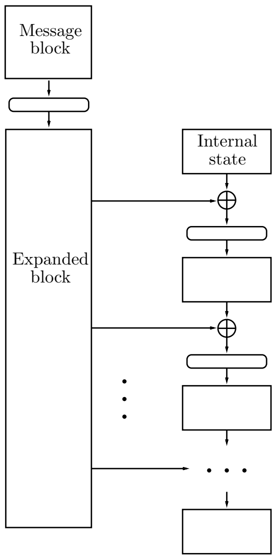

Fig. 6. The outline of the compression function

Table 4. `ShiftRows` and `MixColumns` parameters for the internal round function

Row index i	Offset c_i
0	0
1	1
2	3
3	4

$$B = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}.$$