Collisions for the Compression Function of LAKE*

Jian Guo[1], Krystian Matusiewicz[2], Praveen Gauravaram[2], San Ling[1], Josef Pieprzyk[3], and Huaxiong Wang[1]

[1] School of Physical and Mathematical Sciences,
Nanyang Technological University, Singapore
{guojian,lingsan,hxwang}@ntu.edu.sg
[2] Department of Mathematics,
Technical University of Denmark, Denmark
{K.Matusiewicz,P.Gauravaram}@mat.dtu.dk
[3] Centre for Advanced Computing - Algorithms and Cryptography,
Macquaire University, Australia
josef@ics.mq.edu.au

Abstract. In this paper, we analyse the security of the compression function of the cryptographic hash function LAKE-256 proposed at FSE 2008 by Aumasson, Meier and Phan. We present a near collision attack on the compression function with complexity equivalent to around 2^{30} calls to the compression function and practical memory requirements. We show an example of nearly colliding 256-bit outputs of the compression function of LAKE-256 where only 16 bits differ. Using this method, we present a collision attack on the compression function in around 2^{42} evaluations of the compression function. An interesting feature of this attack is that it is independent of the number of rounds used by the compression function.

1 Introduction

The wave of cryptanalytical results on the cryptographic hash functions following the attacks on MD5 and SHA-1 by Wang et al. [19, 18, 17] has seriously undermined the confidence in many currently deployed hash functions. Around the same time, new generic attacks such as multicollision attack [9], long message second preimage attack [11] and herding attack [10], exposed some undesirable properties and weaknesses in the popular Merkle-Damgård (MD) construction [14, 7]. These recent developments have renewed the interest in the design of hash functions. Subsequent announcement by NIST of the Advanced Hash Standard (AHS) competition, aiming at augmenting the FIPS 180-2 [15] standard with a

* A preliminary version of this paper will appear in FSE2009, this is the full version.

new cryptographic hash function, has further stimulated the interest in the design and analysis of hash functions.

The hash function family LAKE [2], presented at FSE 2008, is one of the new designs. The LAKE hash function family follows the design principles of the HAIFA framework [3, 4] – a strenghtened alternative to the MD construction. The LAKE iteration follows the HAIFA structure. As the additional inputs to the compression function, LAKE uses a random value (also called salt) and an index value, which counts the number of bits/blocks in the input message processed so far.

The designer of LAKE conjecture the ideal security levels against collision and (second) preimage attacks. They also claim that it is hard to find pseudo collisions or near collisions for the members of the LAKE family. So far, the only published cryptanalytical result has been a collision attack on a reduced version of LAKE-256. The attack published by Mendel and Schläffer [13] has complexity 2^{109} if LAKE is applied to 4 rounds (instead of 8).

Our contributions In this work, we analyse the collision resistance of the compression function of LAKE-256. We present a practical near collision attack against the full compression function of LAKE. We also show an example of two distinct input pairs (salt, chaining variable) that, for the same message block, produce digests that differ on 16 out 256 bits. The complexity of our near collision attack is 2^{30} evaluations of the LAKE compression function and requires a manageable amount of memory. An interesting feature of our attack is that it is independent of the number of rounds used by the compression function. Thus, increasing the number of rounds does not increase the security of LAKE. We show how to extend this attack to find full collisions for the compression function with estimated complexity of around 2^{42}.

Our collision attack on the compression function does not threaten the hash function itself directly, but it demonstrates that the arguments put forward in the discussion about the collision resistance of LAKE are no longer valid. We expect that a modification of our attack is also applicable to LAKE-512 but its complexity would be higher because solving an appropriate system of constrains for longer words is going to be more complicated.

The rest of the paper is organized as follows. After introducing the notation in Section 1.1, we briefly describe LAKE-256 in Section 2. Some important properties of the internal function f are discussed in Section 3. In Section 4, we introduce the techniques used for finding the differentials that are in our attack. Finally, in Section 5 we discuss the algorithm for solving the system of conditions induced by the differentials and give the complexity analysis. Section 6 compares our attack with some other attacks. Section 7 concludes the paper.

1.1 Notation

Throughout the paper, we assume that every addition and subtraction is performed modulo 2^n unless otherwise specified, where $n = 32$ for LAKE-256. We use the notation -1 for a word with all bits set to one, i.e. $(1, \ldots, 1)$. Moreover, we use the following notation

- x_i : the i-th bit of x, where $i \in \{0, \ldots, n-1\}$ and x_0 is the least significant bit of x.
- $\overline{x}$: the bitwise complement of x, e.g. $\overline{11001110} = 00110001$
- XOR difference is $x^{\oplus} \stackrel{\text{def}}{=} x \oplus x'$, the modular difference is $\Delta x \stackrel{\text{def}}{=} x' - x$. When we say difference in x, it refers to Δx.
- $x \ggg k$: circular rotation of x to the right by k bits.
- $s = [x^L_k | x^R_k]$, where x^L_k is the most significant $n-k$ bits and x^R_k is the least significant k bits of x, i.e. $s = x^L_k 2^k + x^R_k$.
- $\mathbf{1}[expr]$ is the characteristic function of the expression $expr$, $\mathbf{1}[true] = 1$, $\mathbf{1}[false] = 0$.

2 Description of LAKE

In this section, we provide a brief description of LAKE compression function. Since our analysis does not require all the details of the hashing process, we skip those that are not relevant to our attack. A full description of LAKE can be found in [2].

Basic functions – LAKE uses two functions f and g defined as follows:

$$\begin{aligned} f(a, b, c, d) &= (a + (b \vee C_0)) + ((c + (a \wedge C_1)) \ggg 7) + \\ &\qquad ((b + (c \oplus d)) \ggg 13) \quad , \\ g(a, b, c, d) &= ((a + b) \ggg 1) \oplus (c + d) \quad , \end{aligned}$$

where each variable is a 32-bit word and C_0, C_1 are constants.

The compression function of LAKE has three integral components: SaltState, ProcessMessage and FeedForward. The functionality of these components are described in Algorithms 1, 2 and 3, respectively. The whole compression function of LAKE is described in Algorithm 4. Our attack does not depend on the constants C_i for $i = 0, \dots, 15$ and hence we do not provide their actual values here.

SaltState – This function takes as its input 256-bit initial value H, 128-bit salt S and a 64-bit block index $t_0 \| t_1$. The SaltState expands the combined state size from $256+128+64$ bits to 512 bits. The indices i for S_i, H_i and F_i are reduced modulo 4, 8 and 16, respectively.

```
Input: H = H_0||...||H_7, S = S_0||...||S_3, t = t_0||t_1
Output: F = F_0||...||F_15
for i = 0,...,7 do
 |  F_i = H_i;
end
F_8 = g(H_0, S_0 ⊕ t_0, C_8, 0);
F_9 = g(H_1, S_1 ⊕ t_1, C_9, 0);
for i = 10,...,15 do
 |  F_i = g(H_i, S_i, C_i, 0);
end
```

Algorithm 1: LAKE's SaltState

ProcessMessage This function processes a 512-bit message block by mixing it with the 512-bit input state to produce a 512-bit output state. ProcessMessage uses two non-linear functions f and g, each iterated 16 times as shown in Algorithm 2. The order in which message words are processed is defined by the permutation σ. The indices i for F_i, M_i and W_i are reduced modulo 16.

```
Input: F = F_0||...||F_15, M = M_0||...||M_15, σ
Output: W = W_0||...||W_15
for i = 0,...,15 do
 |  L_i = f(L_{i-1}, F_i, M_{σ(i)}, C_i);
end
W_0 = g(L_15, L_0, F_0, L_1);
L_0 = W_0;
for i = 1,...,15 do
 |  W_i = g(W_{i-1}, L_i, F_i, L_{i+1});
end
```

Algorithm 2: LAKE's ProcessMessage

FeedForward The FeedForward function of LAKE mixes 512-bit output of ProcessMessage with the 256-bit initial value H, 128-bit salt and 64-bit block index to yield an output of 256 bits. The index i for S_i is reduced modulo 4.

Input: $W = W_0 \| \ldots \| W_{15}$, $H = H_0 \| \ldots \| H_7$, $S = S_0 \| \ldots \| S_3$, $t = t_0 \| t_1$
Output: $H = H_0 \| \ldots \| H_7$
$H_0 = f(W_0, W_8, S_0 \oplus t_0, H_0)$;
$H_1 = f(W_1, W_9, S_1 \oplus t_1, H_1)$;
for $i = 2, \ldots, 7$ **do**
| $H_i = f(W_i, W_{i+8}, S_i, H_i)$;
end

Algorithm 3: LAKE's FeedForward

CompressionFunction The description of the r-round compression function of LAKE is presented in Algorithm 4. The LAKE-256 compression function calls ProcessMessage eight times ($r = 8$).

Input: $H = H_0 \| \ldots \| H_7$, $M = M_0 \| \ldots \| M_{15}$, $S = S_0 \| \ldots \| S_3$,
$t = t_0 \| t_1$
Output: $H = H_0 \| \ldots \| H_7$
$F = \mathsf{SaltState}(H, S, t)$;
for $i = 0, \ldots, r-1$ **do**
| $F = \mathsf{ProcessMessage}(F, M, \sigma_i)$;
end
$H = \mathsf{FeedForward}(F, H, S, t)$;

Algorithm 4: LAKE's CompressionFunction

3 Properties of the function f

We start with presenting some properties of the function f that are important for our analysis. The following observation of the rotation effect on the modular addition allows us to simplify the analysis of the behavior of f.

Lemma 1 ([8]) *$(a + b) \ggg k = (a \ggg k) + (b \ggg k) + \alpha - \beta 2^{n-k}$, where $\alpha = \mathbf{1}[a_k^R + b_k^R \geq 2^k]$ and $\beta = \mathbf{1}[a_k^L + b_k^L + \alpha \geq 2^{n-k}]$.*

From the definition, f can be written as

$$f(a, b, c, d) = a + b \vee C_0 + (c \ggg 7) + ((a \wedge C_1) \ggg 7) + (b \ggg 13) + ((c \oplus d) \ggg 13) + \alpha_1 + \alpha_2 - \beta_1 2^{25} - \beta_2 2^{19}, \quad (1)$$

where

$$\alpha_1 = \mathbf{1}[c_7^L + (a \wedge C_1)_7^L \geq 2^7], \qquad \beta_1 = \mathbf{1}[c_7^R + (a \wedge C_1)_7^R + \alpha_1 \geq 2^{25}],$$
$$\alpha_2 = \mathbf{1}[b_{13}^L + (c \oplus d)_{13}^L \geq 2^{13}], \qquad \beta_2 = \mathbf{1}[b_{13}^R + (c \oplus d)_{13}^R + \alpha_2 \geq 2^{19}].$$

Note that α_2 and β_2 are independent of a. Consider now the difference of the outputs of f induced by the difference in the variable a, i.e.

$$\begin{aligned}
\Delta f &= f(a', b, c, d) - f(a, b, c, d) \\
&= [a' + (a' \wedge C_1) + \alpha_1' - \beta_1' 2^{25}] - [a + (a \wedge C_1) + \alpha_1 - \beta_1 2^{25}] \\
&= a' + ((a' \wedge C_1) \ggg 7) - [a + ((a \wedge C_1) \ggg 7)] + (\alpha_1' - \alpha_1) - (\beta_1' - \beta_1) 2^{25} \\
&= f_a(a') - f_a(a) + (\alpha_1' - \alpha_1) - (\beta_1' - \beta_1) 2^{25},
\end{aligned}$$

where

$$f_a(a) \stackrel{\text{def}}{=} a + ((a \wedge C_1) \ggg 7) \ .$$

A detailed analysis (cf. Lemma 4) shows that given random a, a' and c, $P(\alpha_1 = \alpha_1', \beta_1 = \beta_1') = \frac{4}{9}$, so with probability $\frac{4}{9}$, a collision of f_a is also a collision of f when input difference is in a only. Let us call this a *carry effect*. However, if we have control over the variable c, we can adjust the values of $\alpha_1, \alpha_1', \beta_1, \beta_1'$ and always satisfy this condition. From here we can see that $(a + b) \ggg k$ is not a good mixing function when we are considering modular differences.

This reasoning can be repeated for differences in the variable b and similarly for differences in a pair of the variables c, d. It is easy to see that also for those cases, with a high probability, collisions in f happen when the following functions collide

$$f_b(b) \stackrel{\text{def}}{=} b \vee C_0 + (b \ggg 13) \ ,$$
$$f_{cd}(c, d) \stackrel{\text{def}}{=} (c \ggg 7) + ((c \oplus d) \ggg 13) \ .$$

So, when we follow differences in only one or two variables, we can consider only those variables without the side effects from other variables. we summarize these in the following statement.

Observation 1 *Collisions or output differences of f for input differences in one variable can be made independent from the values of other variables.*

We denote the set of solutions for f_a and f_b with respect to input pairs and modular differences as

$$\begin{aligned}
S_{fa} &\stackrel{\text{def}}{=} \{(x, x') | f_a(x) = f_a(x')\} \ , \\
S_{fa}^{A} &\stackrel{\text{def}}{=} \{x - x' | f_a(x) = f_a(x')\} \ , \\
S_{fb} &\stackrel{\text{def}}{=} \{(x, x') | f_b(x) = f_b(x')\} \ , \\
S_{fb}^{A} &\stackrel{\text{def}}{=} \{x - x' | f_b(x) = f_b(x')\} \ .
\end{aligned}$$

Choose the odd elements from S_{fb}^A and define them to be $S_{fb_{odd}}^A$. Note that we can easily precompute all the above solution sets using 2^{32} evaluations of the appropriate functions and 2^{32} words of memory (or some more computations with proportionally less memory).

4 Finding high-level differentials

The starting idea of our analysis is to inject differences in the input chaining values and salt and then cancel them within the first iteration of ProcessMessage. Consequently, no difference appears throughout the compression function until the FeedForward step. If the differences in the chaining values and salt variables are selected appropriately, we can hope they cancel each other, so we get no difference at the output of the compression function.

To find a suitable differential for the attack, an approach similar to the one employed to analyse FORK-256 [12, Section 6] can be used. We model each of the registers a, b, c, d, as a single binary value δa, δb, δc, δd that denotes whether there is a difference in the register or not. Moreover, we assume that we are able to make any two differences cancel each other to obtain a model that can be expressed in terms of arithmetics over $\mathbb{F}_2$. We model the differential behavior of function g simply as $\delta g(\delta a, \delta b, \delta c, \delta d) = \delta a \oplus \delta b \oplus \delta c \oplus \delta d$, where $\delta a, \delta b, \delta c, \delta d \in \mathbb{F}_2$, as this description seems to be functionally closest to the original. For example, it is impossible to get collisions for g when only one variable has differences and such a model ensures that we always have two differences to cancel each other if we need no output difference of g. When deciding how to model $f(a, b, c, d)$, we have more options. First, note that when looking for pure pseudo-collisions, there are no differences in message words and the last parameter of f is a constant, so we need to deal with differences in only two input variables a and b. Since we can find collisions for f when differences are only in a single variable (either a or b), we can model f not only as

$\delta f(\delta a, \delta b) = \delta a \oplus \delta b$ but more generally as $\delta f(\delta a, \delta b) = \gamma_0(\delta a) \oplus \gamma_1(\delta b)$, where $\gamma_0, \gamma_1 \in \mathbb{F}_2$ are fixed parameters. Let us call the pair (γ_0, γ_1) a γ-configuration of δf and denote it by $\delta f_{[\gamma_0,\gamma_1]}$, As an example, $\delta f_{[1,0]}$ corresponds to $\delta f(\delta a, \delta b) = \delta a$, which means that whenever a difference appears in register b, we need to use the properties of f to find collisions in the coordinate b. For functions f appearing in $\mathsf{FeedForward}$, we use the model $\delta f = \delta a \oplus \delta b \oplus \delta c \oplus \delta d$.

With these assumptions, it is easy to see that such a model of the whole compression function is linear over $\mathbb{F}_2$ and finding the set of input differences (in chaining variables $H_0, \ldots, H_7$ and salt registers $S_0, \ldots, S_3$) is just a matter of finding the kernel of a linear map. Since we want to find only simple differentials, we are interested in those that use as few registers as possible. To find them, we can think of all possible states of the linear model as a set of codewords of a linear code over $\mathbb{F}_2$. That way, finding differentials affecting only few registers corresponds to finding low-weight codewords. So instead of an enumeration of all 2^{12} possible states of of $H_0, \ldots, H_7, S_0, \ldots, S_3$ for each γ-configuration of f functions, this can be done more efficiently by using tools like MAGMA [6].

We implemented this method in MAGMA and performed such a search for all possible γ-configurations of the 16 functions f appearing in the first $\mathsf{ProcessMessage}$. We used the following search criteria: (a) as few active f functions as possible; (b) as few active g functions as possible; (c) non-zero differences appear only in the first few steps using function g as it is harder to adjust the values for later steps due to lack of variables we control; (d) we prefer γ-configurations $[1, 0]$ and $[0, 1]$ over $[1, 1]$ because it seems easier to deal with differences in one register than in two registers simultaneously.

The optimal differential for this set of criteria contains differences in registers $H_0, H_1, H_4, H_5, S_0, S_1$ with the following γ-configurations of the first seven f functions in $\mathsf{ProcessMessage}$: $[0, 1]$, $[1, 1]$, $[0, 1]$, $[\cdot, \cdot]$, $[0, 1]$, $[1, 1]$, $[0, 1]$ (Note a simpler configuration (H_0, H_4, S_0) is not possible here). Unfortunately, the system of constraints resulting from that differential has no solutions, so we introduced a small modification of it, adding differences in registers H_2, H_6, S_2, ref. Figure 1. After introducing these additional differences, we gain more freedom at the expense of dealing with more active functions and we can find solutions for the system of constraints. The labels for all constraints are defined Figure 1, we will refer to them throughout the text.

SALTSTATE
input: $H_0, \ldots, H_7$, $S_0, \ldots, S_3$, t_0, t_1
$\mathbf{\Delta} F_0 \leftarrow \mathbf{\Delta} H_0$
$\mathbf{\Delta} F_1 \leftarrow \mathbf{\Delta} H_1$
$\mathbf{\Delta} F_2 \leftarrow \mathbf{\Delta} H_2$
$F_3 \leftarrow H_3$
$\mathbf{\Delta} F_4 \leftarrow \mathbf{\Delta} H_4$
$\mathbf{\Delta} F_5 \leftarrow \mathbf{\Delta} H_5$
$\mathbf{\Delta} F_6 \leftarrow \mathbf{\Delta} H_6$
$F_7 \leftarrow H_7$
$F_8 \leftarrow g(\mathbf{\Delta} H_0, \mathbf{\Delta} S_0 \oplus t_0, C_8, 0)$ {s1}
$F_9 \leftarrow g(\mathbf{\Delta} H_1, \mathbf{\Delta} S_1 \oplus t_1, C_9, 0)$ {s2}
$F_{10} \leftarrow g(\mathbf{\Delta} H_2, \mathbf{\Delta} S_2, C_{10}, 0)$ {s3}
$F_{11} \leftarrow g(H_3, S_3, C_{11}, 0)$
$F_{12} \leftarrow g(\mathbf{\Delta} H_4, \mathbf{\Delta} S_0, C_{12}, 0)$ {s4}
$F_{13} \leftarrow g(\mathbf{\Delta} H_5, \mathbf{\Delta} S_1, C_{13}, 0)$ {s5}
$F_{14} \leftarrow g(\mathbf{\Delta} H_6, \mathbf{\Delta} S_2, C_{14}, 0)$ {s6}
$F_{15} \leftarrow g(H_7, S_3, C_{15}, 0)$
output: $F_0, \ldots, F_{15}$

FEEDFORWARD
input: $R_0, \ldots, R_{15}$, $H_0, \ldots, H_7$,
$S_0, \ldots, S_3$, t_0, t_1
$H_0 \leftarrow f(R_0, R_8, \mathbf{\Delta} S_0 \oplus t_0, \mathbf{\Delta} H_0)$ {f1}
$H_1 \leftarrow f(R_1, R_9, \mathbf{\Delta} S_1 \oplus t_1, \mathbf{\Delta} H_1)$ {f2}
$H_2 \leftarrow f(R_2, R_{10}, \mathbf{\Delta} S_2, \mathbf{\Delta} H_2)$ {f3}
$H_3 \leftarrow f(R_3, R_{11}, S_3, H_3)$
$H_4 \leftarrow f(R_4, R_{12}, \mathbf{\Delta} S_0, \mathbf{\Delta} H_4)$ {f4}
$H_5 \leftarrow f(R_5, R_{13}, \mathbf{\Delta} S_1, \mathbf{\Delta} H_5)$ {f5}
$H_6 \leftarrow f(R_6, R_{14}, \mathbf{\Delta} S_2, \mathbf{\Delta} H_6)$ {f6}
$H_7 \leftarrow f(R_7, R_{15}, S_3, H_7)$
output: $H_0, \ldots, H_7$

PROCESSMESSAGE
input: $F_0, \ldots, F_{15}$, $M_0, \ldots, M_{15}$, σ
$L_0 \leftarrow f(F_{15}, \mathbf{\Delta} F_0, M_{\sigma(0)}, C_0)$ {p1}
$\mathbf{\Delta} L_1 \leftarrow f(L_0, \mathbf{\Delta} F_1, M_{\sigma(1)}, C_1)$ {p2}
$\mathbf{\Delta} L_2 \leftarrow f(\mathbf{\Delta} L_1, \mathbf{\Delta} F_2, M_{\sigma(2)}, C_2)$ {p3}
$L_3 \leftarrow f(\mathbf{\Delta} L_2, F_3, M_{\sigma(3)}, C_3)$ {p4}
$L_4 \leftarrow f(L_3, \mathbf{\Delta} F_4, M_{\sigma(4)}, C_4)$ {p5}
$\mathbf{\Delta} L_5 \leftarrow f(L_4, \mathbf{\Delta} F_5, M_{\sigma(5)}, C_5)$ {p6}
$\mathbf{\Delta} L_6 \leftarrow f(\mathbf{\Delta} L_5, \mathbf{\Delta} F_6, M_{\sigma(6)}, C_6)$ {p7}
$L_7 \leftarrow f(\mathbf{\Delta} L_6, F_7, M_{\sigma(7)}, C_7)$ {p8}
$L_8 \leftarrow f(L_7, F_8, M_{\sigma(8)}, C_8)$
$\vdots$
$L_{15} \leftarrow f(L_{14}, F_{15}, M_{\sigma(15)}, C_{15})$

$W_0 \leftarrow g(L_{15}, L_0, \mathbf{\Delta} F_0, \mathbf{\Delta} L_1)$ {p9}
$W_1 \leftarrow g(W_0, \mathbf{\Delta} L_1, \mathbf{\Delta} F_1, \mathbf{\Delta} L_2)$ {p10}
$W_2 \leftarrow g(W_1, \mathbf{\Delta} L_2, \mathbf{\Delta} F_2, L_3)$ {p11}
$W_3 \leftarrow g(W_2, L_3, F_3, L_4)$
$W_4 \leftarrow g(W_3, L_4, \mathbf{\Delta} F_4, \mathbf{\Delta} L_5)$ {p12}
$W_5 \leftarrow g(W_4, \mathbf{\Delta} L_5, \mathbf{\Delta} F_5, \mathbf{\Delta} L_6)$ {p13}
$W_6 \leftarrow g(W_5, \mathbf{\Delta} L_6, \mathbf{\Delta} F_6, L_7)$ {p14}
$W_7 \leftarrow g(W_6, L_7, F_7, L_8)$
$\vdots$
$W_{15} \leftarrow g(W_{14}, L_{15}, F_{15}, W_0)$
output: $W_0, \ldots, W_{15}$

Fig. 1. High-level differential used to look for collisions

5 Algorithm and Analysis

The process of finding the actual pair of inputs following the differential can be split into two phases. The first one is to solve the constraints from ProcessMessage to get the required Fs (same as Hs used in SaltState). Then, in the second phase, we look back at the SaltState to find appropriate salts to have constraints in FeedForward satisfied. We can do this because the output from ProcessMessage has only a small effect on the solutions for FeedForward.

5.1 Solving the ProcessMessage

An important feature of our differentials in ProcessMessage is that it can be separated into two disjoint groups, i.e. $(F_0, F_1, F_2, L_1, L_2)$ and $(F_4, F_5, F_6, L_5, L_6)$. Differentials for these two groups have exactly the same structure. Thanks to that, if we can find values for the differences in the first group, we can reuse them for the second group by making corresponding registers in the second group equal to the ones from the first group. Following Observation 1 we can safely say that the second group also follows the differential path with a high probability. Algorithm 5 gives the details of solving the constrains in the first group of ProcessMessage.

1: Randomly pick $(L_2, L'_2) \in S_{fa}$
2: **repeat**
3:     Randomly pick F_1, compute $F'_1 = -1 - \Delta L_2 - F_1$
4: **until** $f_b(F_1) - f_b(F'_1) \in S^A_{fb_{odd}}$
5: **repeat**
6:     Randomly pick L_1, F_2
7:     Compute $L'_1 = f_b(F'_1) - f_b(F_1) + L_1$
8:     Compute F'_2 so that $f_b(F'_2) = \Delta L_2 + f_a(L_1) - f_a(L'_1) + f_b(F_2)$
9: **until** $p11$ is fulfilled
10: Pick $(F_0, F'_0) \in S_{fb}$ so that $\Delta F_0 + \Delta L_1 = 0$

Algorithm 5: Find solutions for the first group of differences of ProcessMessage

Correctness We show that after the execution of Algorithm 5, it indeed finds values conforming to the differential. In other words, we show that constraints $p1 - p4$ and $p9 - p11$ hold. Referring to Algorithm 5:

Line 1: (L_2, L'_2) is chosen in such a way that $p4$ is satisfied.
Line 3: F'_1 is computed in such a way that $(F_1 + L_2) \oplus (F'_1 + L'_2) = -1$
Line 4: $\Delta L_1 = \Delta f_b(F_1)$ is odd together with $(F_1 + L_2) \oplus (F'_1 + L'_2) = -1$. This implies that $p10$ could hold, which will be discussed later in Lemma 2. The fact that $\Delta L_1 \in S^A_{fb_{odd}}$ makes it possible that $p1$ and $p9$ hold.
Line 7: L'_1 is computed in such a way that $p2$ holds.
Line 8: F'_2 is computed in such a way that $p3$ holds.
Line 9: after exiting the loop $p11$ holds.
Line 10: (F_0, F'_0) is chosen in such a way that $p1, p9$ hold.

Probability and Complexity Analysis Let us consider the probability for exiting the loops in Algorithm 5. We require $f_a(F_1) - f_a(F_1') \in S^A_{fb_{odd}}$ and the constraint $p11$ to hold. The size of the set $S^A_{fb_{odd}}$ is around 2^{11}. By assuming that $f_a(F_1) - f_a(F_1')$ is random, the probability to have it in $S^A_{fb_{odd}}$ is 2^{-21}. This needs to be done only once. Now we show that the constraint $p11$ is satisfied with the probability 2^{-24}. We have sufficiently many choices, i.e. 2^{64}, for (L_1, F_2) to have $p11$ satisfied. The constraint $p11$ requires that $[(W_1+L_2) \ggg 1] \oplus (F_2+L_3) = [(W_1+L_2']) \ggg 1] \oplus (F_2'+L_3)$, which is equivalent to $[(W_1+L_2) \oplus (W_1+L_2')] \ggg 1 = (F_2+L_3) \oplus (F_2'+L_3)$, where $W_1, L_2, L_2', F_2, F_2'$ are given from previous steps. We have choices for L_3 by choosing an appropriate $M_{\sigma(3)}$. The problem could be rephrased as follows: *given random A and D, what is the probability to have at least one x such that $x \oplus (x+D) = A$?*

To answer this question, let us note first that $x \oplus y = (1, \ldots, 1)$ iff $x + y = -1$. This is clear as $y = \overline{x}$ and always $(x \oplus \overline{x}) + 1 = 0$. Now we can show the following result.

Lemma 2 *For any odd integer d, there exist exactly two x such that $x \oplus (x+d) = (1, \ldots, 1)$. They are given by $x = (-1-d)/2$ and $x = (-1-d)/2 + 2^{n-1}$.*

Proof. $x \oplus (x+d) = -1$ implies that $x + x + d = -1 + k2^n$ for an integer k, so $x = \frac{-1-d+k2^n}{2}$. Only when d is odd, $x = \frac{-1-d}{2} + k2^{n-1}$ an integer and a solution exists. As we are working in modulo 2^n, $k = 0, 1$ are the only solutions. $\square$

Following the lemma, given an odd ΔL_1 and $(F_1+L_2) \oplus (F_1'+L_2') = -1$, we can always find two W_0 such that $(W_0 + L_1) \oplus (W_0 + L_1') = -1$, then $p10$ follows. Such W_0 could be found by choosing an appropriate L_{15} which could be adjusted by choosing $M_{\sigma(15)}$ (if such $M_{\sigma(15)}$ does not exist, although the chance is low, we can adjust L_{14} by choosing $M_{\sigma(14)}$).

Coming back to the original question, consider A as "0"s and blocks of "1"s. Following the lemma above, for $A_i = 0$, we need $D_i = 0$ (except "0" as MSB followed by a "1"); for a block of "1"s, say $A_k = A_{k+1} = \cdots = A_{k+l} = 1$, the condition that needs to be imposed on D is $D_k = 1$. By counting the number of "0"s and the number of blocks of "1"s, we can get number of conditions needed. For an n-bit A, the number is $\frac{3n}{4}$ on average (cf. Appendix Lemma 3).

For LAKE-256, it is 24, so the probability for $p11$ to hold is 2^{-24}. We will need to find the appropriate L_3 so that $p11$ holds. Note we have control over L_3 by choosing the appropriate $M_{\sigma(3)}$. For each differential path found, we need to find message words fulfilling the path. The probability to find a correct message is $1 - \frac{1}{e}$ for the first path by assuming f_c is random (because for a random function from n bits to n bits the probability that a point from the range has a preimage is $1 - \frac{1}{e}$), and $\frac{4}{9}$ for second path because of the carry effect. For example, given L_0, F_{15}, F_0, C_0, the probability to have $M_{\sigma(0)}$ so that $L_0 = f(F_{15}, F_0, M_{\sigma(0)}, C_0)$ is $1 - \frac{1}{e}$. The same $M_{\sigma(0)}$ satisfies $L'_0 = f(F'_{15}, F'_0, M_{\sigma(0)}, C_0)$ (note for this case $F'_{15} = F_{15}$ and $L_0 = L'_0$) is $\frac{4}{9}$. So for each message word, the probability for it to fulfill the differential path is 2^{-2}. We have such restrictions on $M_{\sigma(0)} - M_{\sigma(2)}, M_{\sigma(4)} - M_{\sigma(6)}$ (we don't have such restriction on $M_{\sigma(3)}$ and $M_{\sigma(7)}$ because we still have control over F_3 and F_7), so overall complexity for solving ProcessMessage is $5 \cdot 2^{36}$ in terms of calls to f_a or f_b. The compression function of LAKE-256 calls functions f and g 136 times each and f_a, f_b contain less than half of the operations used in f. So the complexity for this part of the attack is 2^{30} in terms of the number of calls to the compression function.

Solving the second group of ProcessMessage After we are done with the first group, we can have the second group of differential path for free by assigning $F_{i+4} = F_i$, $F'_{i+4} = F'_i$ for $i = 0, 1, 2$ and $L_{i+4} = L_i, L'_{i+4} = L'_i$ for $i = 1, 2$. In this way, we can have $p5 - p8$ and $p12$ automatically satisfied. Similarly, for constraint $p13$ and $p14$, we will need appropriate W_4 and L_7. We have control over W_4 by choosing F_3 and L_4 (note we need to keep L_3 stable to have $p11$ satisfied, this can be achieved by choosing appropriate $M_{\sigma(3)}$). We also have control over L_7 by choosing $M_{\sigma(7)}$.

That way we can force the difference to vanish within the first ProcessMessage. Table 1 shows an example of a set of solutions we found on a standard PC (Core2 Duo 2.33GHz with 4GB memory) using this method.

5.2 Near collisions

In this section we explain how to get a near collision directly from collisions of ProcessMessage. Refer to SaltState and FeedForward in Fig. 1. Note that the function $g(a, b, c, d)$ with differences at positions (a, b) means $\Delta a + \Delta b = 0$, then constraints $(s1 - s6)$ in SaltState can be simplified to:

$$s1 : \Delta H_0 + \Delta S_0 = 0 \tag{2}$$
$$s2 : \Delta H_1 + \Delta S_1 = 0 \tag{3}$$
$$s3 : \Delta H_2 + \Delta S_2 = 0 \tag{4}$$

Table 1. Example of a pair of chaining values F, F' and a message block M that yield a collision in ProcessMessage

F	1E802CB8	799491C5	1FE58A14	07069BED	1E802CB8	799491C5	1FE58A14	74B26C5B
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
F'	C0030007	B767CE5E	30485AE7	07069BED	C0030007	B767CE5E	30485AE7	74B26C5B
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
M	683E64F1	9B0FC4D9	0E36999A	A9423F09	27C2895E	1B76972D	BEF24B1C	78F25F25
	00000000	00000000	00000000	00000000	00000000	00000000	657C34F5	3A992294
L	D0F3077A	31A06494	395A0001	10E105FC	82026885	31A06494	395A0001	10E105FC
	ECF7389A	2F4D466F	9FFC71E1	54BAFAE6	FCDDBCDB	E635FFB7	5D302719	CD102144
L'	D0F3077A	901D9145	95A99FDB	10E105FC	82026885	901D9145	95A99FDB	10E105FC
	ECF7389A	2F4D466F	9FFC71E1	54BAFAE6	FCDDBCDB	E635FFB7	5D302719	CD102144
$L^{\oplus}$	00000000	A1BDF5D1	ACF39FDA	00000000	00000000	A1BDF5D1	ACF39FDA	00000000
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
W	1F210513	1A8E2515	1932829B	1C00C039	1F210513	1A8E2515	1932829B	F4A060BE
	5F868AC3	D8959978	E8F3FF4A	E20AC1C3	8941C0F8	EA8BC74E	6ECDD677	82CFFECE
W'	1F210513	1A8E2515	1932829B	1C00C039	1F210513	1A8E2515	1932829B	F4A060BE
	5F868AC3	D8959978	E8F3FF4A	E20AC1C3	8941C0F8	EA8BC74E	6ECDD677	82CFFECE
$W^{\oplus}$	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000

Note that $H_{i+4} = H_i, H'_{i+4} = H'_i$ for $i = 0, 1, 2$ as required by ProcessMessage, Let $t_0 = t_1 = 0$, then conditions $s4 - s6$ follow $s1 - s3$. Conditions in FeedForward could be simplified to:

$$f1 : f_{cd}(S_0, H_0) = f_{cd}(S'_0, H'_0) \tag{5}$$

$$f2 : f_{cd}(S_1, H_1) = f_{cd}(S'_1, H'_1) \tag{6}$$

$$f3 : f_{cd}(S_2, H_2) = f_{cd}(S'_2, H'_2) \tag{7}$$

and $f4 - f6$ follow $f1 - f3$. This set of constraints can be grouped into three independent sets (si, fi) for $i = 0, 1, 2$ each one of the same type, i.e. $\Delta H + \Delta S = 0$ and $f_{cd}(S, H) = f_{cd}(S', H')$.

To find near collisions, we proceed as follows. First we choose those S_i with $S'_i = S_i - \Delta H_i$ so that the Hamming weight of $f_{cd}(S'_i, H'_i) - f_{cd}(S_i, H_i)$ is small for $i = 0, 1, 2$. Thanks to that, only small differences are expected in the final output of the compression function, due to the fact that inputs from a, b of function f have only carry effect to the final difference of f when inputs differ in c, d only. We choose values of S_i without going through the compression function, so the number of rounds of the compression function does not affect our algorithm. Further, the complexity for finding values of S_i is much smaller than that of ProcessMessage, so it does not increase the 2^{30} complexity. Experiments show that, based on the collision in ProcessMessage, we can have near collisions with very little additional effort. Table 2 shows a sample result with 16-bit of differences out of 256 bits of output.

5.3 Extending the attack to full collisions

It is clear that finding full collisions is equivalent to solving equations (5)-(7). The complexity to solve a single equation is around 2^{12} (cf. Lemma 5 in Appendix). Looking at Algorithm 5, $(s1, f1)$ can be checked when F_1 and F_1' are chosen, so it does not affect the overall complexity. The pair $(s0, f0)$ can be checked immediately after (L_1, L_1') is given as show in Line 7 of Algorithm 5. Similarly, $(s2, f2)$ can be checked after (F_2, F_2') is chosen in Line 8. So the overall complexity for our algorithm to get a collision for the full compression function is 2^{54}.

Table 2. Example of a pair of chaining values F, F', salts S, S' and a message block M that yield near collision in CompressionFunction with 16 bits differences out of 256 bits output. Hs are final output.

F	7B2000C4	23E79FBD	73D102C3	88E0E02B	7B2000C4	23E79FBD	73D102C3	00000000
F'	801FF801	18C0005E	846FD480	88E0E02B	801FF801	18C0005E	846FD480	00000000
S	00010081	23043423	03C5B03E	D44CFD2C				
S'	FB010944	2E2BD382	F326DE81	D44CFD2C				
M	00000012	64B31375	CFA0A77E	8F7BE61F	1E30C9D3	6A9FB0DA	290E506E	3AAE159C
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	1B89AA75
H	261B50AA	3873E2BE	BDD7EC4D	7CE4BFF8	007BB4D4	869473FF	833D9EFA	9DABEDDA
H'	361150AA	387BE23E	FDD6E84D	7CE4BFF8	1071B4D4	869C737F	C33C9AFA	9DABEDDA
$H^{\oplus}$	100A0000	00080080	40010400	00000000	100A0000	00080080	40010400	00000000

5.4 Reducing the Complexity

In this subsection, we show a better way (rather than randomly) to choose (L_2, L_2') so that the probability for the constraint $p11$ to hold increases, which reduces the complexity for collision finding to 2^{42}.

Note the constraint $p11$ is as follows: given W_1, L_2, L_2', what is the probability to have L_3 and (F_2, F_2') so that $((W_1 + L_2) \oplus (W_1 + L_2')) \ggg 1 = (F_2 + L_3) \oplus (F_2' + L_3)$. We calculate the probability by counting the number of 0s and block of 1s in $((W_1 + L_2) \oplus (W_1 + L_2')) \ggg 1$ (let's denote it as $\alpha = \#(((W_1 + L_2) \oplus (W_1 + L_2')) \ggg 1))$. Now we show that the number α can be reduced within the first loop of the algorithm, i.e. given only (L_2, L_2') and (F_1, F_1'), we are able to get the count α and hence, by repeating the loop sufficiently many times, we can reduce the count α to a certain number less than 24 (we don't fix it here, but will give it later).

Note that to find α, we still need W_1 besides (L_2, L_2'). Now we show W_1 can be computed from (L_2, L_2') and (F_1, F_1') only. $W_1 \stackrel{\text{def}}{=} ((W_0 + L_1) \ggg$

$1) \oplus (F_1 + L_2)$, where we restrict $(W_0 + L_1) \oplus (W_0 + L'_1) = -1$. Denote $S = (W_0 + L_1)$, then the equation can be derived to $S \oplus (S + \Delta L_1) = -1$, where $\Delta L_1 \stackrel{\text{def}}{=} f_b(F'_1) - f_b(F_1)$.

So let's make 2^y more effort in the first loop so that α is reduced by y. The probability for first loop to exit becomes 2^{-33-y} and for the second loop, the probability becomes 2^{-60+y}. Choosing the optimal value $y = 13$ (y must be an integer), the probabilities are 2^{-46} and 2^{-47}, respectively. Hence this gives final complexity 2^{42} for collision searching.

6 Comparing with other attacks

Besides the (H, S)-type (differences fall in chaining value H and salt S) attack here, Biryukov *et al.* [5] gives (H, t)-type collision attack and (H)-type near collision attack; both attacks are focused on the compression function of LAKE with complexities of 2^{40} and 2^{105}, respectively.

We note that the (H, t)-type collision attack of [5] on the compression function of LAKE would never extend to the hash function LAKE unless other types of collisions for compression function are found that could extend to the hash function LAKE. When we try to extend the (H, t)-type collision attack on the compression function to the hash function, the colliding block must be the last block for each message. Since a collision on the hash function could have been spanned at least one message block, the block next to the "colliding block" will introduce difference in the chaining value due to the fact that block indices 't' are different. However, in the (H, t)-type collision attack of [5], the triplet (H, M, S) are same after the "colliding block"(H contains no difference, this does not satisfy configuration of the attack, hence introduces differences in output H unless other types of collision attack is found). This means the lengths of the two colliding messages for the LAKE hash function are different. Note that this length is encoded into the last block of the message as part of the padding rule, which means that the last block of the padded message must differ. This contradicts the assumption of the attack that the colliding messages have no difference.

We note that our (H, S)-type collision attack on the compression function of LAKE is not limited by the above restriction to extend it to the hash function. While salt values are controlled by the user in the (H, S)-type collision attack, they are not encoded into the message during padding. To summaries, though there is no guarantee that our (H, S)-type collision attack on the LAKE compression function extends to its hash function, this extension is certainly not ruled out as in the (H, t)-type collision attack of [5].

7 Conclusions and future work

In this paper we showed how to find near collisions in practice and full collisions with complexity 2^{42} for the compression function of the cryptographic hash function LAKE-256.

The presented work can be extended in several directions. It is possible that the same method of looking for high level differentials could be also used to look for ones suitable to generate collisions for the complete hash function.

Combining our results with results presented in [13] may lead to a more efficient hybrid attack which may be worth investigating.

We believe that the methods presented here and used to analyse LAKE-256 can be useful to the analyse of some of the candidates selected for first round of NIST SHA-3 competition. Our collision attack on LAKE-256 compression function does not extend to its successor BLAKE [1], as the internal function used in BLAKE is bijective with respective to each chaining variable, so internal collisions do not exist.

Acknowledgement

The work in this paper is supported in part by Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03, the Singapore Ministry of Education under Research Grant T206B2204, and the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. Praveen Gauravaram is supported by the Danish Research Council for Technology and Production Sciences grant number 274-08-0052. Krystian Matusiewicz is supported by the Danish Research Council for Technology and Production Sciences grant number 274-07-0246. Josef Pieprzyk is supported by Australian Research Council grants DP0663452 and DP0987734. We thank Wei Lei for his helpful discussion and anonymous reviewers of FSE2009 for their useful comments.

References

1. J.-P. Aumasson, L. Henzen, W. Meier, and R. C.-W. Phan. Sha-3 proposal blake. Candidate to the NIST Hash Competition 2008. Available at http://131002.net/blake/.

2. J.-P. Aumasson, W. Meier, and R. Phan. The hash function family LAKE. In *Fast Software Encryption – FSE 2008*, volume 5086 of *LNCS*, pages 36–53. Springer, 2008.
3. E. Biham and O. Dunkelman. A framework for iterative hash functions – HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. Available at http://eprint.iacr.org/2007/278.
4. E. Biham, O. Dunkelman, C. Bouillaguet, and P.-A. Fouque. Re-visiting HAIFA and why you should visit too. ECRYPT workshop "Hash functions in cryptology: theory and practice", June 2008. The slides of this presentation are available at http://www.lorentzcenter.nl/lc/web/2008/309/presentations/Dunkelman.pdf.
5. A. Biryukov, P. Gauravaram, J. Guo, D. Khovratovich, S. Ling, K. Matusiewicz, I. Nikolic, J. Pieprzyk, and H. Wang. Crypanalysis of lake hash family. Accepted by FSE2009, to appear.
6. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3–4):235–265, 1997. http://magma.maths.usyd.edu.au/.
7. I. B. Damgård. A design principle for hash functions. In G. Brassard, editor, *Advances in Cryptology – CRYPTO '89*, volume 435 of *LNCS*, pages 416–427. Springer-Verlag, 1989.
8. M. Daum. *Cryptanalysis of Hash Functions of the MD4-Family.* PhD thesis, Ruhr-Universität Bochum, May 2005.
9. A. Joux. Multicollisions in iterated hash functions. application to cascaded constructions. In *Advances in Cryptology – CRYPTO 2004*, volume 3152 of *LNCS*, pages 306–316. Springer, 2004.
10. J. Kelsey and T. Kohno. Herding Hash Functions and the Nostradamus Attack. In S. Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006, Proceedings*, volume 4004 of *LNCS*, pages 183–200. Springer, 2006.
11. 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*, volume 3494 of *LNCS*, pages 474–490. Springer-Verlag, 2005.
12. K. Matusiewicz, T. Peyrin, O. Billet, S. Contini, and J. Pieprzyk. Cryptanalysis of FORK-256. In *Fast Software Encryption – FSE'07*, volume 4593 of *LNCS*, pages 19–38. Springer, 2007.
13. F. Mendel and M. Schläffer. Collisions for round-reduced LAKE. In *Information Security and Privacy – ACISP 2008*, volume 5107 of *LNCS*, pages 267–281. Springer, 2008.
14. R. C. Merkle. One way hash functions and DES. In G. Brassard, editor, *Advances in Cryptology - CRYPTO '89*, volume 435 of *LNCS*, pages 428 – 446. Springer-Verlag, 1989.
15. National Institute of Standards and Technology. Secure hash standard (SHS). FIPS 180-2, August 2002.
16. S. Paul and B. Preneel. Solving systems of differential equations of addition. In C. Boyd and J. M. G. Nieto, editors, *ACISP*, volume 3574 of *Lecture Notes in Computer Science*, pages 75–88. Springer, 2005.
17. X. Wang, Y. L. Yin, and H. Yu. Collision search attacks on SHA-1. http://theory.csail.mit.edu/ yiqun/shanote.pdf, 13 Feb 2005.
18. X. Wang, Y. L. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology – CRYPTO'05*, volume 3621 of *LNCS*, pages 17–36. Springer, 2005.
19. X. Wang and H. Yu. How to break MD5 and other hash functions. In R. Cramer, editor, *Advances in Cryptology – EUROCRYPT'05*, volume 3494 of *LNCS*, pages 19–35. Springer-Verlag, 2005.

A Lemmas and proofs

Lemma 3 *Given random x of length n, then the average number of "0"s and block of "1"s, excluding the case "0" as MSB followed by "1", is $\frac{3n}{4}$.*

Proof. Denote C_n as the sum of the counts for "0"s and blocks of "1"s for all x of length n, denote such x as x_n. Similarly we define P_n as the sum of the counts for all x of length n with MSB "0" (let's denote such x as x_n^0); and Q_n for the sum of the counts for all x of length n with MSB "1" (denote such x as x_n^1). It is clearly that

$$C_n = P_n + Q_n \tag{8}$$

Note that there are 2^{n-1} many x with length $n-1$, half of them with MSB "0", which contribute to P_{n-1} and the other half with MSB "1", which contribute to Q_{n-1}. Now we construct x_n of length n from x_{n-1} of length $n-1$ in the following way:

- Append "0" with each x_{n-1}^1, this "0" contribute to C_n once for each x_{n-1}^1 and there are 2^{n-2} many such x_{n-1}^1.
- Append "1" with each x_{n-1}^1, this "1" does not contribute to C_n
- Append "0" with each x_{n-1}^0, this contributes 2^{n-2} to C_n
- Append "1" with each x_{n-1}^0, this contributes 2^{n-2} to C_n

So overall we have $C_n = P_{n-1}+P_{n-1}+2^{n-2}+Q_{n-1}+2^{n-2}+Q_{n-1}+2^{n-2} = 3\cdot 2^{n-2}+2C_{n-1}$. Note $C_1 = 2$, solving the recursion, we get $C_n = \frac{3n+1}{4}\cdot 2^n$. Exclude the exceptional case, we have final result $\frac{3n}{4}$ on average.

Lemma 4 *Given random $a, a', x \in \mathbb{Z}_{2^n}$ and $k \in [0, n)$, $\alpha \stackrel{\text{def}}{=} \mathbf{1}[a_k^L + x_k^L \geq 2^k], \alpha' \stackrel{\text{def}}{=} \mathbf{1}[a_k'^L + x_k^L \geq 2^k], \beta \stackrel{\text{def}}{=} \mathbf{1}[a_k^R + x_k^R + \alpha \geq 2^{n-k}], \beta' \stackrel{\text{def}}{=} \mathbf{1}[a_k'^R + x_k^R + \alpha \geq 2^{n-k}]$ as defined in Lemma 1, then $P(\alpha = \alpha', \beta = \beta') = \frac{4}{9}$.*

Proof. Consider α and α' first,

$$\begin{aligned}
P(\alpha = \alpha' = 1) &= P(a_k^L + x_k^L \geq 2^k, a_k'^L + x_k^L \geq 2^k) \\
&= P(x_k^L \geq (2^k - min\{a_k^L, a_k'^L\})) \\
&= P(a_k^L \geq a_k'^L)P(x_k^L \geq 2^k - a_k'^L) + P(a_k'^L > a_k^L)P(x_k^L \geq 2^k - a_k^L) \\
&= \frac{1}{2}\cdot\frac{1}{3} + \frac{1}{2}\cdot\frac{1}{3} \\
&= \frac{1}{3}
\end{aligned}$$

Similarly we can prove $P(\alpha = \alpha' = 0) = \frac{1}{3}$, so $P(\alpha = \alpha') = \frac{2}{3}$. Note the definitions of β and β' contain α and α', but $\alpha, \alpha' \in \{0,1\}$, which is generally much smaller than 2^{n-k}, so the effect of α to β is negligible. We can roughly say $P(\beta = \beta') = \frac{2}{3}$. So $P(\alpha = \alpha', \beta = \beta') = P(\alpha = \alpha')P(\beta = \beta') = \frac{4}{9}$.

Lemma 5 *Given random H, H', then the probability to find S, S' with $\Delta S + \Delta H = 0$ and $f_{cd}(S,H) = f_{cd}(S',H')$ is 2^{-12}.*

Proof. Let's expand the expression $f_{cd}(S,H) = f_{cd}(S',H')$:

$$
\begin{aligned}
& f_{cd}(S,H) = f_{cd}(S',H') \\
\iff\ & S \ggg 7 + (S \oplus H) \ggg 13 = S' \ggg 7 + (S' \oplus H') \ggg 13 \\
\iff\ & S \ggg 7 - S' \ggg 7 = (S' \oplus H') \ggg 13 - (S \oplus H) \ggg 13 \\
\overset{p=0.242}{\iff}\ & -\Delta S \ggg 7 = (S' \oplus H' - S \oplus H) \ggg 13 \\
\iff\ & -\Delta S \lll 6 = S' \oplus H' - S \oplus H \\
\iff\ & S' \oplus H' = S \oplus H - \Delta S \lll 6 \\
\iff\ & (S + \Delta S) \oplus H' = S \oplus H - \Delta S \lll 6 \\
\iff\ & (S - \Delta H) \oplus H' - \Delta H \lll 6 = S \oplus H
\end{aligned}
$$

Given H, H' and ΔH, we are to solve S for the above. This family of problems are solved by Paul and Preneel [16]. Experiments show that the probability for the above to have solution is 2^{-12}.