Scrutinizing rebound attacks: new algorithms for improving the complexities

María Naya-Plasencia*

FHNW, Windisch, Switzerland

Abstract. Rebound attacks are a state-of-the-art analysis method for hash functions. These cryptanalysis methods are based on a well chosen differential path and have been applied to several hash functions from the SHA-3 competition, providing the best known analysis in these cases. In this paper we study rebound attacks in detail and find for a great number of cases, that complexities of existing attacks can be improved. This is done by determining problems that adapt optimally to the cryptanalytic situation, and by using better algorithms to follow the differential path. These improvements are essentially based on merging big lists in a more efficient way, as well as on new ideas on how to reduce the complexities. As a result, we introduce general purpose new algorithms for enabling further rebound analysis to be as performant as possible. We illustrate our new algorithms for real hash functions and demonstrate how to reduce the complexities of the best known analysis on five hash functions: JH, Grøstl, ECHO, Luffa and Lane (the first four are round two SHA-3 candidates).

Keywords: hash functions, SHA-3 competition, rebound attacks, algorithms

1 Introduction

The rebound attack is a recent technique introduced in [11] by Mendel et al. It was conceived to analyze AES-like hash functions (like Grøstl [5] in [12, 6, 13], Echo [2] in [12, 6, 15], Whirlpool [1] in [9]). The rebound attack is composed of two parts: the inbound phase and the outbound phase. The inbound phase finds with a low cost a big number of pairs of values that satisfy a part of a differential path that would be very expensive to satisfy in a probabilistic way. The outbound phase uses these values to perform an attack. This technique has been used in other algorithms that use permutations that are less AES-like. For example JH [17](reduced to 22 rounds) [14] and Luffa [3](reduced to 7 rounds) [8], that use Sboxes of size 4×4 and have a linear part in which mixing is done in a very different way than in the AES; or LANE [7], that includes several AES states treated by the AES round transformation but where the mixing between these states is a different transformation, and was analysed in [10, 18]. In these cryptanalysis results, the rebound attack needs to be refined and adapted to each case, but all of them are based first, on a differential path, and second, on an algorithm that finds solutions that verify this differential path. Something common to all those algorithms is that a merge of big lists is needed. In this paper we introduce some new algorithms specific for the rebound attack that allow a lower complexity for finding pairs that verify the differential path by introducing some ideas that improve the previous procedures and/or by making a better merging of lists than the ones done so far. This will allow us to reduce the complexities of the attacks, as shown on Table 1. In this table we can see how we have considered the best existing attacks against five hash functions (4 of them are second round candidates of the SHA-3 competition) and we have been able to improve their complexity by scrutinizing the original attack and finding an algorithm more efficient for finding the wanted solutions for the differential path, which most of the times involves a better merging of the lists, and sometimes it is due to stating more adequate conditions in the general algorithm. The problem of merging the lists can be described as one generalized problem, and, depending on which case we are in, one or more algorithms will be considered. In most of the cases, the main idea is to do a sieving (in general like the one done

*Supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center of the Swiss National Science Foundation under grant number 5005-67322

in [4]) so that we do not have to try all the elements in one list with all the elements in the other(s).

The generalized problem: Let $L_1, \ldots, L_{2N}$ be $2N$ lists with $2^{l_1}, \ldots, 2^{l_{2N}}$ elements respectively. Each element in a list consists of x groups of values and y groups of differences. Each group contains a specific value (or difference) stored with $|s|$ bits. We want to merge these $2N$ lists into a new one that consists of the $2N-$tuples formed by elements of $L_1, \ldots, L_{2N}$ so that the elements of each $2N$-tuple verify a certain relation t.

The relation t can have many different forms, as we will see later. Let us remark here that the aim is to find all the $2N$-tuples that verify the relation t (the case where we want to find just some of them for a linear t is treated in [16]). Depending on the values of N, x, y and in particular on the type of relation t, we will propose different algorithms for merging these lists in an efficient way. Finding which is the reduced problem (lists, parameters, transformation) that best adapts to our situation and that will give us the best complexity is an important task that must be done by the cryptanalyst in each particular case. We are going to give here different algorithms for several very common cases and for different parametters: when t is linear, when t is non-linear, when t can be applied group-wise. We will also show in some particular cases how to correctly identify the more adequate problem (or problems) to solve can reduce the complexity.

We will give some real examples of application:

Table 1. Improvements on best known attacks.

Hash function	Best Known analysis	Rounds	Previous			This paper	
			Time	Memory	Ref.	Time	memory
JH	semi-free-start coll.	16	2^{190}	2^{104}	[14]	$\mathbf{2^{96.12}}$	$\mathbf{2^{96.12}}$
JH	semi-free-start near coll.	19,22	2^{168}	$2^{143.70}$	[14]	$\mathbf{2^{95.63}}$	$\mathbf{2^{95.63}}$
Grøstl-256	compresion function prop.	full(10)	2^{192}	2^{64}	[13]	$\mathbf{2^{182}}$	2^{64}
Grøstl-256	internal permutation dist.	full(10)	2^{192}	2^{64}	[13]	$\mathbf{2^{175}}$	2^{64}
Grøstl-512	compresion function prop.	11	2^{640}	2^{64}	[13]	$\mathbf{2^{630}}$	2^{64}
ECHO-256	internal permutation dist.	full(8)	2^{182}	2^{37}	[15]	$\mathbf{2^{151}}$	2^{67}
Luffa	semi-free-start coll.	7	2^{132}	2^{132}	[8]	$\mathbf{(2^{104})}$	$\mathbf{(2^{102})}$
LANE-256	semi-free-start coll.	full(6+3)	2^{96}	2^{88}	[10]	$\mathbf{2^{80}}$	$\mathbf{2^{66}}$
LANE-512	semi-free-start coll.	full(8+4)	2^{224}	2^{128}	[10]	2^{224}	$\mathbf{2^{64}}$

- JH: For $d = 4$ (d is the dimension of a block of bits) and $d = 8$ some reduced round attacks were given in [14]. When one-inbound phase is used (the 16 rounds analysis), we can instantiate a problem where $N = 1$, the relation t applies independently to each group, and is derived from a linear transformation (where the elements of both lists are each half of the input of the linear transformation, and we want half of the output to have no differences). We propose an algorithm that improves the complexities. In the case of 19 and 22 rounds where three-inbounds are used, we can improve the complexities by using the same algorithm as for the one-inbound attack, combined with a different order of solving the inbound phases, as we will see in Section 5.1.
- Grøstl: In [13] a non-random property of the Grøstl-256 compression function is given using a technique called SuperSbox with a complexity of 2^{192} in time and 2^{64} in memory. The same complexity is given for a distinguisher on the internal permutation. Here we have identified an instance of the problem that allows us to perform the same attacks with time complexity of 2^{182} and 2^{175} respectively and the same memory.

- ECHO-256: In [15] a distinguisher on the internal permutation is given with complexity 2^{182} in time and 2^{37} in memory. We propose here an algorithm that solves the problem for a particular case of $N = 1$, $x = 0$ and t cannot be applied independently to each group, and that allows to perform the same attack with a complexity of 2^{151} in time and 2^{67} in memory, which is a previously unknown trade off that improves the time complexity.
- Luffa: We will have the case where $N = 1$, $x = 0$ and relation t can be applied independently to each group and establishes a relation between an element of the input and an element of the output. In [8] an algorithm (parallel matching) was introduced that could reduce the complexity (from 2^{132} to 2^{102}) for solving this problem. Here we generalize this method.
- Lane-256: Here several problems are identified and several algorithms are applied for reducing the total complexity of the attack given in [10] from 2^{96} in time and 2^{88} in memory to 2^{80} in time 2^{58} in memory + 2^{64} in time and 2^{66} in memory (which are the complexities of two different steps). The first algorithm solves the problem in the case $N = 2$ and where t is the identity for the y groups of differences and is a linear relation between the x elements of the lists L_1, L_2, L_3 and L_4. The other algorithm solves the problem for $N = 1$, $x = 0$ and t cannot be applied independently to each group.
- Lane-512: Here we use three algorithms for reducing the complexity of the attack given in [10]. The first one is applied in the case $N = 2$ and t is the identity for the y groups of differences and a linear relation between the elements of the lists L_1, L_2, L_3 and L_4 independent for each group, so it is the same one as we used for Lane-256. The second one is applied when $N = 2$, $y = 0$ and t is a linear relation between the elements of the lists L_1, L_2, L_3 and L_4. The last algorithm is applied when $N = 3$, $y = 0$ and t cannot be applied independently to each group.

Besides these results, the aim and main interest of this paper is to show some general lines for improving the rebound attacks. In particular, the introduction of several new algorithms that improve considerably the overall effectiveness when big lists are needed to be merged in different rebound scenarios and that we will be able to apply in a quite automated way once we have identified the problem. First we will introduce the basic algorithms classified by the relation t and choosing some parameters for simplifying the explanation that can be adapted in each particular case. All these algorithms are quite related and can allow some trade-offs, depending on the actual parameters. We will next show how to identify and decompose a specific problem for applying these algorithms with some examples on real attacks, improving their complexity. Identifying the most adequate problem is a fundamental task.

1.1 Notations

- $|s|$ is the size of a group. In general the size of the $|s| \times |s|$ Sbox involved.
- S will represent the Sbox involved in each particular case.
- d_j^i is the ith group out of y from the list L_j. It represents a difference of a group.
- v_j^i is the ith group out of x from the list L_j. It represents a value of a group.
- ℓ_i is a linear permutation.
- 2^{-p_o} is the probability of a property in o to be verified.
- AES state: is a state of size 128 bits that can be seen as a 4x4 matrix of bytes.
- ECHO state: is a state of size 2048 bits that can be seen as a 4x4 matrix of AES states.
- Grøstl state: is a state of size 512 bits that can be seen as a 8x8 matrix of bytes.
- Lane state: is two (for 256) or four (for 512) AES states in parallel.
- JH state: a state of size 2^{d+2} that can be seen as 2^d words of 4 bits.
- Luffa state: is a state of size 256 that can be seen as 64 4-bit words.
- SB: The AES-like transformation SubBytes.
- SR: The AES-like transformation ShiftRows.
- MC: The AES-like transformation MixColumn.

– SC: The Lane Transformation SwapColumns, that mixes two or four AES states by interchanging it's columns.
– BigSR, BigMC, BigSB: the three operations defined in ECHO for treating the AES states (instead of bytes) that are similar to the AES ones.
– SuperSbox: is an Sbox defined by SR ∘ SB ∘ MC ∘ SR ∘ SB. Applied on an AES state, it can be seen as a 32x32 Sbox. Applied on a Grøstl state, as a 64x64 Sbox.
– SuperSbox set: each one of the 4 (in the AES state) or 8 (in the Grøstl state) sets that act as input and output of the SuperSbox.
– BigSuperSbox: is an SuperSbox defined by BigSR ∘ BigSB ∘ BigSC ∘ BigSR ∘ BigSB. Applied to ECHO it defines sets of size 4 AES-states.

2 Algorithms for t linear and group-wise

These algorithms are the simplest ones. Still, they have not been aplied to some of the previously mentioned attacks though they can reduce considerably the complexity. This is mainly due to the overall complexity of the attacks when considering all the parts together, and that is also why it is useful to identify and isolate the problem and the algorithm to solve it, as we have done here.

2.1 $N = 2$, $x \neq 0$, $y \neq 0$

Here we treat the case when t is a very simple relation that can be decomposed into smaller ones, which modelizes quite well the message insertion relations:

$$d_1^i = d_2^i = d_3^i = d_4^i \text{ and}$$

$$\ell_1(v_1^j, v_2^j) = \ell_2(v_3^j, v_4^j),$$

for $i \in [1, y]$ and $j \in [1, x]$. Without loss of generality, we consider that $l_1 + l_2 \leq l_3 + l_4$. Merging these four lists for keeping the values that verify the defined relation can be done efficiently by making a sieve regarding the differences. We will first order the four lists by their groups of differences. Next, for each of the $2^{|s|y}$ values for these differences, we consider the $2^{l_1-|s|y}$ and the $2^{l_2-|s|y}$ elements related to lists L_1 and L_2 and we compute $\ell_1(v_1^j, v_2^j)$ for $j \in [1, x]$ with each possible pair. The cost of this step will be $2^{l_1+l_2-2|s|y}$, and we generate a list L_{12}. We can do the same with lists L_3 and L_4 generating the new list L_{34}. Now we want to merge these two lists. We will do it by checking, for each element of L_{12} if it appears in L_{34}. This is done with complexity also $2^{l_3+l_4-2|s|y}$. Let us remark here that list L_{34} does not need to be stored but we can check each element as soon as we generate it. We repeat this for each of the possible values of the y groups of differences. The total complexity will be $2^{l_1+l_2+l_3+l_4-3|s|y-|s|x} + 2^{l_1+l_2-|s|y} + 2^{l_3+l_4-|s|y}$ in time and $2^{l_1} + 2^{l_2} + 2^{l_3} + 2^{l_4} + 2^{l_1+l_2-2|s|y}$ in memory. The number of solutions on the merged list will be $2^{l_1+l_2+l_3+l_4-3|s|y-|s|x}$

2.2 $N = 2$, $x \neq 0$, $y = 0$

This case can be treated as a special case of the previous one. The proposed algorithm will be useful in cases where some other step is the time bottle-neck and we want to reduce the memory requirements. It allows us to make a time-memory trade-off (that reduces memory) for the particular problem. We write t like the relation on the values that we had before

$$\ell_1(v_1^j, v_2^j) = \ell_2(v_3^j, v_4^j),\ j \in [1, x].$$

Now, we do not have differences to make a sieve. If we applied the same algorithm as before, we would need a memory of $\max(2^{l_1+l_2}, 2^{l_3+l_4})$. If we want to reduce this memory requirements,

because this part of the attack is far from the bottle neck in time but is the bottleneck in memory, we can reduce considerably the memory requirements (it will be $2^{l_1} + 2^{l_2} + 2^{l_3} + 2^{l_4}$) by increasing the time complexity: if we can re-write t as

$$\ell'(v_1^j, v_2^j, v_3^j) = v_4^j.$$

So with a complexity of $2^{l_1+l_2+l_3} + 2^{l_1+l_2+l_3+l_4-x|s|}$ (instead of $2^{l_1+l_2} + 2^{l_1+l_2+l_3+l_4-x|s|}$) we can try all the combinations of these 3 lists and for each, we compute $\ell'(v_1^j, v_2^j, v_3^j)$ and check if this value is on the list L_4 (which we had previously ordered). We will obtain $2^{l_1+l_2+l_3+l_4-x|s|}$ matches with a time complexity of $2^{l_1+l_2+l_3}$ and no additional memory requirements, appart from the already given four lists, if each time we obtain a match we use it instead of stocking it (as it is the case in the example that of Section 5.6).

2.3 $N = 1$, $x \neq 0$, $y \neq 0$ and the elements on both lists are inputs to t

In this case we can write the relation t as

$$\ell(d_1^i, d_2^i) = o^i,$$

where o^i must have some special property. Let us remark here that the size of the input of ℓ is $2^{2|s|}$. We can precompute a table of all it's possible inputs and all it's related outputs with a cost of $2|s|$ in both time and memory. We will obtain the wanted property in o_i with a probability of $2^{-p_{o_i}}$ once the first input is fixed. Now, for merging lists L_1 and L_2 we will order them by the y groups of differences. We next just have to go through all the $min(2^{|s|y} - 1, 2^{l_1})$ possible values for these differences in L_1 and for each one check which values of the precomputed table can be separately associated to $d_1^0 \dots d_1^y$. We will obtain $2^{(|s|-p_{o_i})y}$ possible matchs for all of the y involved groups. Next we check if those values of differences can be found in the L_2 list. In total, we will find $2^{(|s|-p_{o_i})y} \min(2^{|s|y}, 2^{l_2})$ matches on the y groups of differences between the two lists with a complexity of $2^{(|s|-p_{o_i})y} \min(2^{|s|y} - 1, 2^{l_1})$ in time and $2^{l_1} + 2^{l_2}$ in memory. We can see that this scenario would be the same if $x = 0$, as we do not use the x values and we just keep a difference match. In real attacks the values will be used in further steps. For example, if another relation is established between the remaining groups, for each match of the y groups, we can try the $2^{l_1-|s|y}$ elements of L_1 associated with the corresponding $2^{l_2-|s|y}$ elements of $|L_2|$ to see if they verify the remaining relations. The total complexity will be $[2^{(|s|-p_{o_i})y} \min(2^{|s|y} - 1, 2^{l_1}) 2^{l_1+l_2-2|s|y}] + 2^{l_1+l_2-yp_{o_i}}$ in time and $2^{l_1} + 2^{l_2} + 2^{l_1+l_2-yp_{o_i}}$ in memory. As in the previous section, if each time we obtain a match we can use it and don't need to store it, this last memory term can be reduced to $2^{l_1} + 2^{l_2}$.

3 Algorithm for t non-linear and group-wise: Parallel matching

In this section we will only discuss the case when $x = 0$. Otherwise, the groups of differences that have an associated value will be instantly matched to their corresponding value (if possible). The scenario for the case with $x = 0$ is a very interesting one, where we want to find a solution of differences for a part of a differential path. In [8] a new technique is introduced to find solutions for a differential path where they first find all the possible differences that could verify it, and next, the associated values. In this case we have to merge big lists of differences before and after an Sbox. Also in [8], a technique was introduced for reducing the cost of merging these lists: the parallel matching. Here we generalize this technique. When talking about differences, $(2^{|s|} - 1)$ is all the possible differences in the input of each one of the y groups, as we exclude 0. The relation t here is

$$S(a) \oplus S(a \oplus d_1^i) = d_2^i,$$

where a exists but we do not want to recover it here and $i \in [1, y]$. Lets say that a difference in the input of the Sbox can be associated to a difference in the output with a probability of $2^{-p_{d_2^i}}$. Then, we expect $2^{l_1+l_2-yp_{d_2^i}}$ matches.

We order the lists $L_1 = \{d_1^1, \ldots, d_1^y\}$ and $L_2 = \{d_2^1, \ldots, d_2^y\}$ by the differences in the first n groups $\{d_i^1, \ldots, d_i^n\}$. We will supose for simplicity that $2^{l_i} > 2^{n|s|}$ (which is a more usual case to consider). This will define $2^{n|s|}$ sets in each list with $2^{l_1-n|s|}$ and $2^{l_2-n|s|}$ elements associated to each set and to L_1 and L_2 respectively. We can build a new list L_n of matches of size $2^{2n|s|-np_{d_2^i}}$ formed by elements of the form $\{d_1^1, \ldots, d_1^n, d_2^1, \ldots, d_2^n\}$. We repeat the process with the m next groups, $\{d_i^{n+1}, \ldots, d_i^{n+m}\}$, creating the list L_m of size $2^{2m|s|-mp_{d_2^i}}$ formed by elements of the form $\{d_1^{n+1}, \ldots, d_1^{n+m}, d_2^{n+1}, \ldots, d_2^{n+m}\}$. Next, this list L_m is used to build a new one, L'_m in the following way:
Each element $\{d_1^{n+1}, \ldots, d_1^{n+m}, d_2^{n+1}, \ldots, d_2^{n+m}\}$ from the list L_m is associated to $2^{l_2-m|s|}$ elements $\{d_2^1, \ldots, d_2^n, d_2^{n+m+1}, \ldots, d_2^y\}$ by list L_2. With this, we can build the list L'_m of size $2^{l_2+m(|s|-p_{d_2^i})}$ and of elements of the form $\{d_1^{n+1}, \ldots, d_1^{n+m}, d_2^1, \ldots, d_2^y\}$ which are ordered by the first $(n+m)$ groups. Now, for each one of the $2^{2n|s|-np}$ matchs obtained from the list L_n, and for each of the $2^{l_1-n|s|}$ elements $\{d_1^{n+1}, \ldots, d_1^y\}$ from L_1 associated, we check if the value their for the differences $d_1^{n+1}, \ldots, d_1^{n+m}, d_2^1, \ldots, d_2^n$ appears in the list L'_m. This can be done with cost one for each of the $2^{l_1+n|s|-n_{d_2^i}}$ tries. When the value appears of L'_m, we know that this pair of differences can satisfy the first $(n+m)$ groups (we will find $2^{l_1+l_2-(n+m)p_{d_2^i}}$), so we have to check if it verifies the remaining $(y-n-m)$ groups. In total we will find the $2^{l_1+l_2-yp_{d_2^i}}$ matches that exist, with a complexity in time $(2^{l_1+l_2-(n+m)p_{d_2^i}} + 2^{l_1+n|s|-np_{d_2^i}})$ and $2^{l_2+m(|s|-p_{d_2^i})}$ in memory.

4 Algorithm for t non-linear and non-group-wise

We are going to explain these algorithms in the particular framework where they have been applied, giving also the general expressions and characteristics that will allow them to be used in other cases.

4.1 $N = 1$, $x = 0$ and t can be split into two independent relations

We are going to study here a particular case that can be easily applied when mixing two or more AES states. We suposse that t can be separated in two relations (a and b), involving each half of the elements. So t can be represented this way:

$$F_a(c_a, d_1^a) = d_2^a \text{ and } F_b(c_b, d_1^b) = d_2^b, \text{ with}$$

$$F_a(c_a, d_1^a) = f(c_a) \oplus f(c_a \oplus d_1^a)$$
$$F_b(c_b, d_1^b) = f(c_b) \oplus f(c_b \oplus d_1^b)$$

where $c_a = c_a^1 || c_a^2$ and $c_b = c_b^1 || c_b^2$ are the values associated to each match of differences that we also want to recover and $d_1 = d_1^a || d_1^b$ and $d_2 = d_2^a || d_2^b$ are the elements from the two lists. We are going to consider the case where we can decompose:

$$F_a(c_a, d_1^a) = F_a^3(c_a^{2'}, F_a^2(F_a^1(c_a^1, d_1^a))),$$
$$F_b(c_b, d_1^b) = F_b^3(c_b^{2'}, F_b^2(F_b^1(c_b^1, d_1^b))),$$

and re-write t as

$$F_a^2(F_a^1(c_a^1, d_1^a)) = F_a^{3^{-1}}(c_a^2, d_2^a),$$
$$F_b^2(F_b^1(c_b^1, d_1^b)) = F_b^{3^{-1}}(c_b^2, d_2^b).$$

It is easier to understand the utility of these expressions with an example that defines F_a^i and F_b^i for $i \in [1,3]$. As in the LANE-256 attack from [10] we can use this algorithm for improving the complexity we will explain this particular case here, which will simplify the explanation. In the LANE-256 hash function, each lane has a size of two AES states. If we refer to the differential path used in [10], who's two first inbounds are represented in Figure 1, we can observe that the state that intervenes in the first inbound (from #1 to #8) can be separated in two parts so that they do not interact with each other during this first inbound and additionally are associated each to one half of the state intervening in the second inbound (from #9 to #16) so that they do not interact with each other either. Using the notations from [10], if we consider

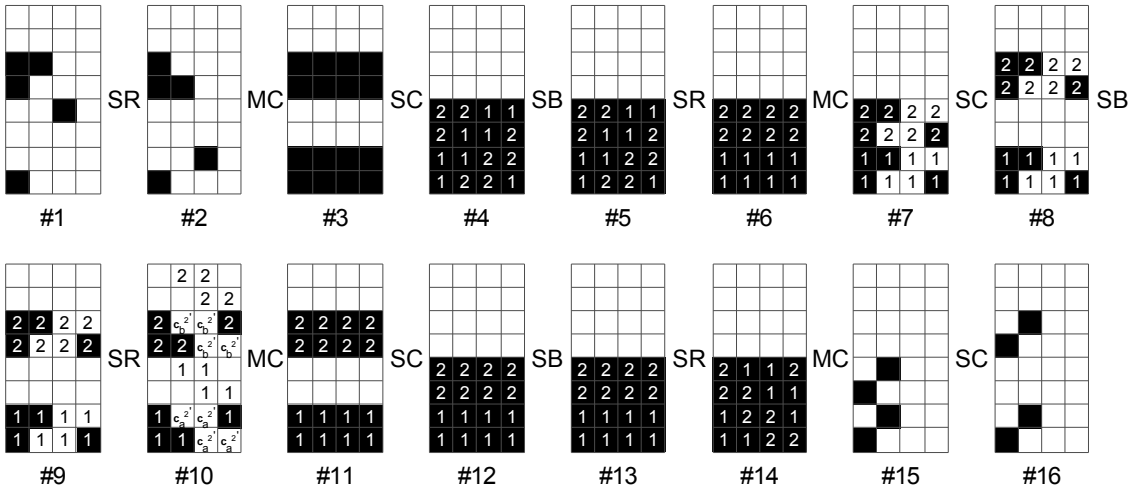

Fig. 1. Differential path associated to the first improvement on the LANE analysis.

the active AES state at instant 4, we can separate it in the two sets that we have just talked about: one formed by the bytes [(0, 2),(0, 3),(1, 1),(1, 2), (2, 0),(2, 1),(3, 0),(3, 3)] and the other by [(0, 0),(0, 1),(1, 0),(1, 3),(2, 2),(2, 3), (3, 1),(3, 2)]. The first set will be associated to the following bytes of the state 14: [(0, 1),(0, 2),(1, 2),(1, 3),(2, 0),(2, 3), (3, 0),(3, 1)] and the second set will be associated to [(0, 0),(0, 3),(1, 0),(1, 1), (2, 1),(2, 2),(3, 2),(3, 3)]. It is important to notice that the list L_1 will be the list of possible differences at state 4 and L_2 will be the list of possible differences at the state 14. So, as we have just seen, the differences d_1 in L_1 can be separated conforming this two sets in d_1^a and d_1^b, as well as the differences in L_2 can be separated in d_2^a and d_2^b. We can now forget about the ShiftRow opperations, as we will see each of the sets as one block, already including implicitely this operation, and define the relation t using the previous notations (we will define it for F_a, representing the first set, being similar for the second set, F_b). F_a^1 takes 8 bytes of values and differences (16 bytes of information in total) of input and outputs 8 bytes of values with 4 bytes of differences (so 12 bytes of information in total, corresponding to half of the active state in 7, but afterwards we will only use the values and differences associated to the active bytes: $c_a^{1'}$ and $d_1^{a'}$); F_a^2 takes as input $c_a^{1'}$ and $d_1^{a'}$ and outputs another 4 active bytes with it's values and differences (8 bytes in total); F_a^3 takes as input the values and differences of the 4 active bytes of the output of F_a^2, and also a 4 bytes value $c_a^{2'}$ that completes the 8 byte half state in the instant 10 and outputs 8 bytes of values c_a^2 and differences d_2^a (16 in total) of state 14:

$$F_a^1(c_a^1, d_1^a) = \text{MixColumns}(\text{SubBytes}(c_a^1)) \oplus \text{MixColumns}(\text{SubBytes}(c_a^1 \oplus d_1^a))$$

$$F_a^2(c_a^{1'}, d_1^{a'}) = \text{SubBytes}(c_a^{1'}) \oplus \text{SubBytes}(c_a^{1'} \oplus d_1^{a'})$$

$$F_a^{3^{-1}}(c_a^2, d_2^a) = \text{MixColumns}(\text{SubBytes}(c_a^2)) \oplus \text{MixColumns}(\text{SubBytes}(c_a^2 \oplus d_2^a)).$$

As already said, here we want to find a match between the two lists of differences as well as finding the values that make it possible. We will choose 2^n values for the differences in the active bytes of $F_a^{3^{-1}}(c_a^2, d_2^a)$ and we will try tp match them with the 2^{l_2} differences d_2^a in L_2 (the probability for finding a match will be 2^{-p} and it will produce 2^p possible values associated). This will define 2^p values c_a^2 for each one of the 2^{n+l_2-p} matchs of differences that we will obtain. With this we build a list L_a of differences and values of size 2^{n+l_2}. We repeat the same with the other half of the differences, building the list L_b, of the same size than L_a.
Now, for each one of the 2^{l_1} differences in L_1, we do the following: first, we choose 2^m values for the differences in $F_a^1(c_a^1, d_1^a)$ and we will try to match them with the correspondings d_1^a (the probability for finding a match will be 2^{-p_1} and it will produce 2^{p_1} possible values associated). This will leave 2^{m-p_1} matchs with 2^{p_1} possibles c_a^1 each; so 2^m possible values for pairs (d_1^a, c_a^1). For each of this 2^m values and differences, we compute $F_a^1(c_a^1, d_1^a)$. The probability of finding a match between the input and the output of F_a^2 is 2^{-p_2}. The number of matches that we will find between this values and the precomputed list L_a will be $2^{n+l_2+m-p_2}$. We repeat all of this with the differences in $F_b^1(c_b^1, d_1^b)$ and L_b obtaining also $2^{n+l_2+m-p_2}$ matches. Now, for each of the $2^{n+l_2+m-p_2}$ d_2^a, we check if it's corresponding half d_2^b associated by L_2 belongs to one of the $2^{n+l_2+m-p_2}$ obtained with L_b. The time complexity will be $2^{l_1+m+1} + 2^{l_1+m+l_2+n-p_2} + 2^{l_2+n+1}$ and the memory complexity $2^{l_1} + 2^{l_2} + 2^{l_2+n+1}$ for obtaining

$$2^{l_1+2(l_2+n+m-p_2)-l_2} \text{ solutions.}$$

Depending the number of solutions needed and the disponibility, we will choose n and m. We will see in Section 5.5 how this algorithm allows to considerably reduce the complexity of the LANE-256 semi-free-start collision presented in [10]; and in Section 5.3 how an related algorithm can be applied for improving the time complexity of the ECHO distinguisher.

4.2 $N = 3$, $y = 0$ and we have some additional constraints

In this case, t can not be separated into smaller relations, but for it to be verified we can identify some constraints that will facilitate the task of finding a match between both lists. We will study the particular case in which t can be represented this way:

$$F_1(v_1, v_2, v_3) = F_2(v_4, v_5, v_6), \text{ with}$$

$$F_1(v_1, v_2, v_3) = f(k_1, v_1, v_2, v_3) \oplus f(k_2, v_1, v_2, v_3)$$
$$F_2(v_4, v_5, v_6) = f(k_3, v_4, v_5, v_6) \oplus f(k_4, v_4, v_5, v_6),$$

where k_1, k_2, k_3 and k_4 are constants and v_1, v_2, v_3, v_4, v_5 and v_6 are the elements from the six lists. In particular, we are looking to reduce the memory needs. A basic algorithm would build a table of size $2^{l_1+l_2+l_3}$ with all the possible combinations of the elements of the three first lists, and for each combination, would compute and store in order the value $F_1(v_1, v_2, v_3)$. Next would go through all the possible combination of the lists L_4, L_5 and L_6, for each compute $F_2(v_4, v_5, v_6)$ and check if it is included on the previously computed list. The complexity will be of $2^{l_1+l_2+l_3} + 2^{l_4+l_5+l_6}$ in time and $2^{l_1+l_2+l_3}$ in memory. We are going to see how, when $F_2(v_4, v_5, v_6)$ (and $F_1(v_1, v_2, v_3)$ then) can only take a reduced number of values in an output of size 2^n, bigger than $2^{l_4+l_5+l_6}$, this will allow us to reduce the memory needs of solving this problem if this values can be known and follow a certain structure that allow us to identify them easily. As done in the previous section, we will identify the previous expressions with a concrete case, which will help understanding the framework. For instance, we are going to apply the following algorithm in a part of the LANE-512 attack for reducing the memory needs. In this case t is non-linear and non-group-wise, $N = 3$ and $y = 0$. The elements from the lists L_1

to L_6 correspond to v_1 to v_6. The constants k_1 to k_4 are determined by the previous inbounds steps of the attack. The previous expression f will be:

$$f : \text{SB} \circ \text{SC} \circ \text{MC} \circ \text{SR} \circ \text{SB} \circ \text{SC} \circ \text{MC} \circ \text{SR} \circ \text{SB} \circ \text{SC} \circ \text{MC} \circ \text{SR} \circ \text{SB} \circ \text{SC}$$

So we divide the output of F_2 in four equal parts, each one corresponding to a column, $F_2(v_4, v_5, v_6) = c_1||c_2||c_3||c_4$, and we observe that each part c_i takes values among $2^m = 2^{(l_4+l_5+l_6)/4}$ instead of $2^{n/4}$, where $2^{n/4-(l_4+l_5+l_6)/4}$ is the number of values that will never be taken by one of the c_i parts of the output of F_2. In these case we can store the 4 lists of $2^{n/4-(l_4+l_5+l_6)/4}$ values that will never be taken by each part of the output of F_2. Next, for each one of the $2^{l_1+l_2+l_3}$ possible combinations of lists L_1, L_2 and L_3 we compute $F_1(v_1, v_2, v_3) = c'_1||c'_2||c'_3||c'_4$. When one of the four c'_i parts appears to be in the corresponding precomputed list of imposible values for F_2, we know that this combination of (v_1, v_2, v_3) won't make a match, so we do not store it. This way, when building the list formed by all the possible good combinations of lists L_1, L_2 and L_3, it's size will be smaller than before, as it will be $2^{l_1+l_2+l_3-(n-(l_4+l_5+l_6))}$, as those are the only ones we are keeping. The rest of the algorithm will work the same way as the basic one: we go through all the possible combination of the lists L_4, L_5 and L_6, for each compute $F_2(v_4, v_5, v_6)$ and check if it is included on the previously computed list. The complexity will now be of $2^{l_1+l_2+l_3} + 2^{l_4+l_5+l_6}$ in time and $2^{n-(l_4+l_5+l_6)} + 2^{l_1+l_2+l_3-(n/4-(l_4+l_5+l_6)/4)}$ in memory.

As we said, this case simulates very well the AES differentials generated by fewer differences, and we can directly apply it to the Lane-512 case described in [10]. In the associated differential path, in the state before the last MixColumn of f, the four active bytes have already a value and a difference assigned, because of the third inbound, but the remaining three bytes of each column are not determined yet. That means that if we separate the differences of the final state in the four active columns defined before the last ShiftRows (as the last operations are linear, we can omit them for finding the collision), the difference in each column will only take 2^{24} values out of 2^{32}. It is easy to compute, then, which are the possible values that the difference of each column can take. If we do this for one of the two lanes, we can try all the possibilities for the other lane, and only store the ones that can be a match, i.e., the ones that have the four differences in the four columns that can correspond to the first lane. This way, we will only store a list of $2^{32\times3}2^{-8\times4} = 2^{64}$ elements instead of 2^{96}. Next, we just have to try all the possibilities for the first lane and try to find a match. We will see in Section 5.6 a more detailed analysis of the improvements on the analysis of Lane-512.

5 Identifying the problems: examples

In this section we are going to show how, from the best known attacks on the hash functions JH, Grøstl, ECHO, Luffa, Lane-256 and Lane-512 we can apply some improvements based on the proposed algorithms and on the correct identification of the problem to solve, that will reduce the overall complexity of the attacks. For a detailed description of the hash functions, we refer to their SHA-3 submission documents. As those attacks are quite complex, we will not explain here all the details, but we will just give the information needed for identifying the problem, refering in each case to the corresponding attack.

5.1 JH

For simplicity, we consider here the attack on JH with ($d = 4$) for 8 rounds when using the three-inbound attack given in [14] with a complexity of $2^{33.09}$ in time and $2^{24.18}$ in memory. We shall see here how, when we apply one of the previously introduced algorithms, this complexity can be significantly improved. For $d = 8$ the improvement is performed the same way for the three-inbound attack on 19 and 22 rounds, and it is simpler in the case of one-inbound for 16

rounds, where we just have to apply the algorithm from Section 2.3. The three-inbound attack for ($d = 4$) uses the differential path represented in Figure 2, where #0 represents the initial internal state and #8 the final one. The colored parts are the parts with a difference. Each small square represents the 4x4 Sbox for rows from 0 to 15, and each rectangle represents the linear permutation on 8 bits. Each wire contains 4 bits.

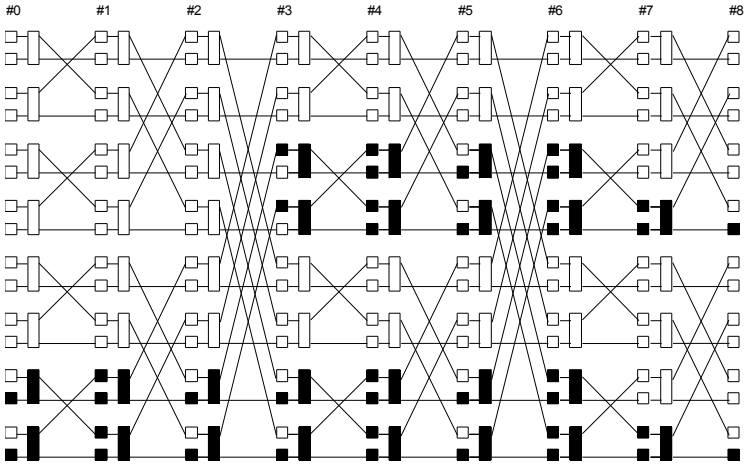

Fig. 2. Differential path for $d = 4$ of JH of the three-inbound attack.

To improve this attack, we use the algorithm from Section 2.3 together with a reordering of the steps. We start the attack as in [14] by finding the possible solutions for the first inbound (from round #0 to the beginning of round #2), storing a list L_A of $2^{11.36}$ solutions with a cost of 2^{16}. We will consider a different third inbound: from round #5 to the beginning of round #7. In this part, we will have two sets, each one associated to a list: $L_{0,1,8,9}$ and $L_{2,3,10,11}$. Using the algorithm from Section 2.3 we can generate this two lists of values and differences of size 2^{16} that satisfy the corresponding parts of the differential path between rounds #5 and #7 with a cost of 2^{16}: at the end of round #5, four outputs of the linear permutations are active and have been generated by a 4-bit difference (each one will define a set). For each one of these four sets, we can guess the values of the 8 bits and of the 4-bit difference and compute the output of round #5. This way we build 4 lists $(L^5_{0,1}, L^5_{2,3}, L^5_{8,9}, L^5_{10,11})$ of $2^{11.95}$ elements in each. So each list $L^5_{j,j+1}$ will have $y = 2$ difference groups $d^1_{j,j+1}$ and $d^2_{j,j+1}$ (differences at the end of round #5). We are now merging the lists $L^5_{0,1}, L^5_{8,9}$ into $L^5_{0,1,8,9}$ using the algorithm from Section 2.3, where ℓ is the linear permutation. Here the special property of $o^{i,j}$ is that only its 4 rightmost bits are active and $p_{o^{i,j}} = 4.09$ (so we will have 8.18 bit conditions per merge):

$$\ell(d^1_{0,1}, d^2_{8,9}) = o_{0,9}$$

$$\ell(d^1_{8,9}, d^2_{0,1}) = o_{8,1}.$$

We will obtain $L^5_{0,1,8,9}$ of size 2^{16} with a cost of 2^{16}. Merging the lists $L^5_{2,3}, L^5_{10,11}$ into $L^5_{2,3,10,11}$ will be done the same way. Next, for rounds #3 and #4 we will repeat the same procedure at the same cost for obtaining two sets of solutions for these two rounds: $L^4_{0,1,2,3}$ and $L^4_{8,9,10,11}$ of size also 2^{16}. We will now merge these two lists and the list L_A. Merging $L^4_{0,1,2,3}$ and $L^4_{8,9,10,11}$ determines 3.91×2 bit conditions, and for merging both of them with L_A 16 bit conditions need to be verified (from the two actives 4-bit words where they collide). Applying a close variant of the algorithm from Section 2.3 we obtain a new list L_B, of size $2^{19.54}$, of solutions for the

rounds #0 to #5 with a cost of $2^{19.54}$. Next, in a similar way we will merge L_B with $L^5_{0,1,8,9}$ and $L^5_{2,3,10,11}$ (there are here 32 bit conditions to verify), and we will obtain $2^{19.54}$ solutions that verify the merge (and so rounds from #0 to #7) with a cost of $2^{19.54}$. For each solution, we check if it also verifies round #8 (3.91×2 bit conditions), obtaining $2^{11.72}$ solutions (as in [14], before taking the symmetries into account). The complexity of the attack using our algorithm will then be $2^{19.54}$ in time and $2^{19.54}$ in memory, improving the previous complexity of $2^{33.09}$ in time and $2^{24.18}$ in memory. Similarly as we have shown for $d = 4$ and 8 rounds, we can identify the same problem and apply the algorithm of Section 2.3 to the attack on 19 and 22 rounds of [14] that uses three-inbound attacks and has a complexity of $2^{168.02}$ in time and $2^{143.70}$ in memory, so that it can also be improved using the same algorithm, and having a final complexity of $2^{95.63}$ in time and memory. The 16 rounds with one-inbound attack of [14], can also be improved to $2^{96.12}$ in time and memory, while it's complexity was 2^{190} in time and 2^{104} in memory.

5.2 *Grøstl*

Here, we consider the results on Grøstl-256 presented in [13], where, in particular, distinguishers are given for the full compression function as well as for the internal permutation. We can improve by a factor of 2^{10} or 2^{17} (depending on the differential path considered) their time complexities. In this case, instead of finding a new algorithm we have identified a better problem to solve: the lists L_1 and L_2 of differences in the input and in the output of the SuperSbox phase respectively, that in [13] are built with all the possible differences, can be smaller and be just built with the differences that we know for sure might also satisfy the outbound phase. The factor that we are going to gain will depend on the number of active columns in the input (N_i) and the number of active columns in the output (N_o). So instead of merging two lists of size $2^{l_1} = 2^{64N_i}$ and $2^{l_2} = 2^{64N_o}$, we want to merge one list of size $2^{l_1} = 2^{63N_i}$ and one list of size $2^{l_2} = 2^{56N_o}$. The algorithm applied to merge these lists is the same one as in [13], obtaining a complexity in time of $2^{63N_i+56N_o}$ instead of $2^{64(N_i+N_o)}$. This is possible because in this attack the one byte differences introduced by the constants additions have a fixed value, implying that the number of possible differences at the input and ouput of the SuperSbox will be smaller. In the 10-round compression function analysis this improves from 2^{192} to 2^{182} and in the permutation distinguisher from 2^{192} to 2^{175}. In the case of Grøstl-512 we can improve time complexity of the analysis on 11 rounds of the compression function from 2^{640} to 2^{630}.

5.3 ECHO

In [15] an analysis of the whole ECHO-256 permutation is provided which has complexity 2^{182} in time and 2^{37} in memory. By studying in detail this analysis we have been able to provide some trade-offs that were previously unknown and that allow to improve the time complexity. For example, we can perform the same attack with a complexity 2^{151} in time and 2^{67} in memory. For this, we will use an algorithm closely related to the one introduced in Section 4.1, that we explain here in detail. We consider the differential path given in [15]. In Figure 3, the inbound part is represented. We need to find 2^{86} solutions of this part in order to satisfy also the outbound part. In the figure, the BigSB are decomposed into the AES operations (2 rounds, where we omit operations that do not influence the differential path) and we can see how two BigSB can be seen as a BigSuperSbox (from #α to #D), where the sets formed by it have the form of the highlighted sets of four AES states. Inside this BigSuperSbox, we can identify two separated SuperSboxes, each one formed by one of the BigSB, and where the sets formed by them have the form of the grey bytes. We are going to explain our method for finding solutions for this inbound phase at a lower cost. In [15], 2^{32} solutions are found with a cost of 2^{128}. Here we will find 2^{64} solutions with the same cost. We explain first how to find solutions for one BigSuperSbox set, as the remaining three can be done the same way (one ECHO state is

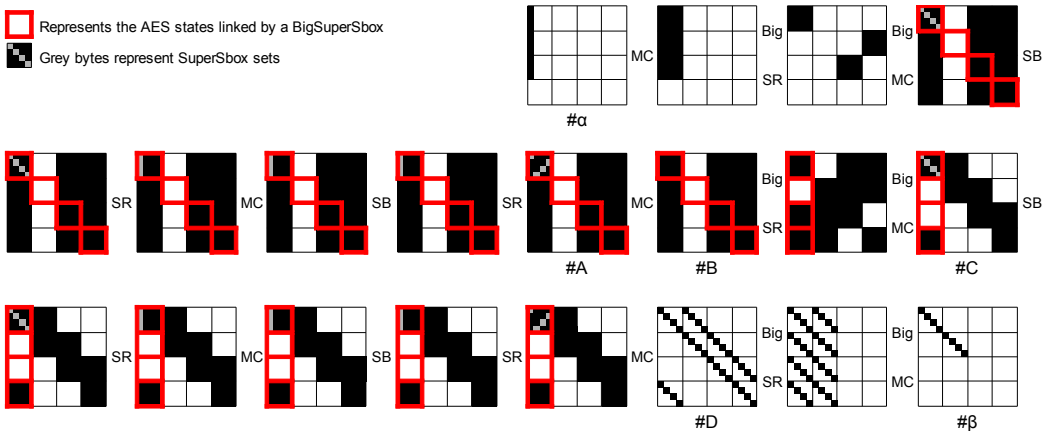

Fig. 3. Inbound part of the differential path on ECHO. A number of 2^{86} solutions needs to be found for satisfying the outbound part.

composed by 4 BigSuperSbox sets). Once we have 2^{64} solutions for each BigSuperSbox set, each one will be associated to only one on each of the other sets by the differences in $\#\beta$ (the last state represented in Figure 3), giving in total 2^{64} solutions for the whole inbound phase. We will describe the procedure for the BigSuperSbox set defined in Figure 3, being similar for the remaining three.

First, we will fix the differences in $\#\alpha$. Next, we compute and store the 2^{32} values and differences for the 12 associated SuperSbox sets in #A, building the lists L^i_A for $i \in [1, 12]$.

Then, for each of the 2^{32} possible differences in one AES state from #D (AES state (0,0) or AES state (3,0) for the first column), we will compute and store the 2^{32} values and differences for the 4 associated SuperSbox sets in #C. This will build the (4×2^{32}) lists $L^i_{\Delta_{(0,0)}}$, each one containing the 2^{32} possible values and differences for the diagonal i of the AES state (0,0) in #C; as well as the (4×2^{32}) lists $L^i_{\Delta_{(3,0)}}$, associated to AES state (3,0) in #C.

We will go through all the 2^{64} possible differences in the two first diagonals of the AES state (0,0) in #B (and not the 2^{128} possible differences of the AES state as done in [15]). These two diagonals will determine the differences in the same two diagonals of the AES states (2,2) and (3,3) in #B (because of the linear conditions imposed by the two AES states without differences in #C), and also the differences in the same two diagonals of the AES states (0,0) and (3,0) in #C (the two diagonals of differences of the active states in #B are already determined, and with the BigSR and BigMC transformations we can compute the difference in the same diagonals in #C). So for each one of these 2^{64} differences:

- From the lists $L^i_{\Delta_{(0,0)}}$ and $L^i_{\Delta_{(3,0)}}$ for $i \in [1, 2]$, we will create a new list L_o. Recall that the differences in the first and second diagonal of the AES state (0,0) of #C are fixed. If we want to find the elements from $L^1_{\Delta_{(0,0)}}$ and $L^2_{\Delta_{(0,0)}}$ that satisfy these differences, we will find one element per list and per $\Delta_{(0,0)}$, i.e., for each one of the 2^{32} $\Delta_{(0,0)}$ we will find one value from the first list and one value from the second that will generate the desired difference. The same will happen with the $L^i_{\Delta_{(3,0)}}$ lists. If we combine both results, we can generate the list L_o of size 2^{64} where we will store all the possible values $(v^1_{(0,0)}, v^2_{(0,0)})$ for the diagonals 1 and 2 of (0,0) (in #C) combined with all the possible values $(v^1_{(3,0)}, v^2_{(3,0)})$ for the same diagonals of (3,0) (in #C). We are going to apply a transformation to the values that we have stored: we will store, for each combination of $\Delta_{(0,0)}$ with $\Delta_{(3,0)}$ (and so, for each quadruple of values), the following two 32 bits values: $v^1_{(0,0)} \oplus (3 \times v^1_{(3,0)})$ and $v^2_{(0,0)} \oplus (3 \times v^2_{(3,0)})$. Let us call $b^i_{(x,y)}$ the values from diagonal i of AES state (x,y) of #B. From the BigMC transformation,

we know that:

$$v^1_{(0,0)} \oplus (3 \times v^1_{(3,0)}) = (2 \times b^1_{(0,0)}) \oplus (b^1_{(2,2)}) \oplus (b^1_{(3,3)}) \oplus (9 \times b^1_{(0,0)}) \oplus (3 \times b^1_{(2,2)}) \oplus (6 \times b^1_{(3,3)}),$$

$$v^2_{(0,0)} \oplus (3 \times v^2_{(3,0)}) = (2 \times b^2_{(0,0)}) \oplus (b^2_{(2,2)}) \oplus (b^2_{(3,3)}) \oplus (9 \times b^2_{(0,0)}) \oplus (3 \times b^2_{(2,2)}) \oplus (6 \times b^2_{(3,3)}).$$

We will order list L_o by these two values that we had just computed: $v^1_{(0,0)} \oplus (3 \times v^1_{(3,0)}), v^2_{(0,0)} \oplus (3 \times v^2_{(3,0)})$. The cost of this step is about 2^{64}.

– Now, we try the 2^{64} possible remaining values for the differences in the AES state (0,0) in #B. For each one of these 2^{64} possible differences:
  - This will determine the remaining differences in #B and #C and will also allow us to compute all the differences in #A. Once the total difference in #A is fixed, it will have only one associated value by the 12 lists L^i_A for the three active states. This is due to the fact that each difference on the 32 bit sets will appear only once on the lists of size 2^{32}, having then the fixed difference just one associated pair of values (determined by the lists L^i_A) for the active part of the state. With these values, we can compute
  $$(2 \times b^1_{(0,0)}) \oplus (b^1_{(2,2)}) \oplus (b^1_{(3,3)}) \oplus (9 \times b^1_{(0,0)}) \oplus (3 \times b^1_{(2,2)}) \oplus (6 \times b^1_{(3,3)})$$
  and
  $$(2 \times b^2_{(0,0)}) \oplus (b^2_{(2,2)}) \oplus (b^2_{(3,3)}) \oplus (9 \times b^2_{(0,0)}) \oplus (3 \times b^2_{(2,2)}) \oplus (6 \times b^2_{(3,3)}),$$
  and check if we obtain a match with one of the $v^1_{(0,0)} \oplus (3 \times v^1_{(3,0)})$ and $v^2_{(0,0)} \oplus (3 \times v^2_{(3,0)})$, so we will check if what we have obtained appears in the list L_o, which is ordered conforming to these values, and so the cost of this will be about one. The probability of finding these values is 2^{-64} times the size of the list, which is 2^{64}, so we will find one match in the list.
  - Once the match is found, $\Delta_{(0,0)}$ and $\Delta_{(3,0)}$ will be fixed by the value matched in L_o, and as the differences for the whole BigSuperSbox set in #C are also fixed, we will obtain from $L^3_{\Delta_{(0,0)}}$, $L^4_{\Delta_{(0,0)}}$, $L^3_{\Delta_{(3,0)}}$ and $L^4_{\Delta_{(3,0)}}$ the asigned values for the diagonals 3 and 4. From the relations of BigMC they will need to satisfy 64 bit conditions, so it will be a valid solution with a probability of 2^{-64}.

In the end, we will have obtained $2^{64}2^{64}2^{-64} = 2^{64}$ solutions per BigSuperSbox, and as we explained at the beginning of this section, this will mean 2^{64} solutions for the inbound part of the differential path, with a complexity of 2^{129} in time and 2^{67} in memory. Recall that we need 2^{86} solutions for this part for satisfying the outbound part, so we will have to repeat this 2^{22} times (with different differences fixed in $\#\alpha$), having a total compelxity of 2^{151} in time and 2^{67} in memory.

5.4 *Luffa*

In [8], a way of finding a semi-free-start collision is provided for 7 rounds out of 8. In this case, the problem is very easy to identify: we have two lists of differences, one of differences of the inputs of 52 active Sboxes, the other one of the outputs of these 52 active Sboxes. Then, t is non-linear and group-wise. We can apply the algorithm from Section 3, where $y = 52$, $n = m = 13$, $l_1 = 65.6$ and $l_2 = 67$. The complexity was reduced from 2^{132} to 2^{102} (in time and in memory).

5.5 Lane-256

The analysis in [10] provides a way of finding a semi-free start collision for the complete compression function of Lane-256 with a complexity of 2^{96} in time and 2^{88} in memory. In this section we are going to identify 2 concrete problems extracted from this attack, and by applying two of the previously described algorithms, we are able to reduce the total complexity of the attack to 2^{80} in time and 2^{66} in memory, or more precisely, to 2^{80} in time and 2^{58} in memory + 2^{64} in time and 2^{66} in memory. We are not going to describe in detail here the analysis from [10], but we give the information needed for identifying and defining the problem to be treated by the corresponding algorithm.

Identifying the first problem: In this attack, the three first steps have as aim to find 2^{56} solutions for two inbounds in 4 independent lanes. Each one of the four lanes represents an independent and similar problem. Instead of looking at it as three steps, we are going to unify it in just one, and we will use the differential path from Figure 1. We can now build the list of possible differences in the input: from five active bytes, we obtain 2^{40} possible differences in the input, before the first SB considered. This will form the list L_1, where $x = 0$. We can do the same with the possible difference in the output: out of a totally full active AES state, we want to reach a possition with only 4 active bytes. The list L_2 will be formed by all the 2^{32} possible differences in the output after the last Sbox considered (in the two inbounds). We want to merge these two lists keeping the differences that can verify the whole path defined by the two inbounds as well as recover the associated values that enable to obtain a total of 2^{56} values and differences as solutions. It is quite obvious that the relation t between the differences in both lists is not linear and cannot be applied group-wise. We will directly apply the algorithm from Section 4.1 with the following parameters: $m = 24$, $n = 32$, $p = p_1 = 8$, $p_2 = 64$. The cost of this step was 2^{96} in time and 2^{88} in memory (it was the bottleneck in both). Now, we can perform these two inbound phases with a complexity of 2^{66} in time complexity and 2^{65} in memory. As this step is not anymore the time or memory bottleneck of the attack (it would have been if we had just improved the other steps), we can try to reduce the rest of the complexities now.

Identifying the second problem: Once we have finished the newly defined first step (the previous one) we have obtained 2^{56} solutions for the first two inbounds, for each lane (four lists of values and differences). They need to be merged so that they verify the message expansion. In [10] this is done in steps 4 and 5 with a complexity of 2^{80} in time and memory. This memory complexity can be reduced to 2^{48} by directly applying the algorithm in Section 2.1, where $y = 4$ and $x = 8$, obtaining 2^{64} solutions for this step and giving the new bottleneck of the time complexity of the attack: 2^{80}. The last part of the attack is the same as in [10], and is the bottleneck in memory: $2^{64} \times 4$.

5.6 Lane-512

A semi-free-start collision attack is given in [10] for the whole compression function of Lane-512 with a complexity of 2^{224} in time and 2^{128} in memory. Applying three of the previously described algorithms we can reduce this memory complexity from 2^{128} to 2^{66}. For being able to do this we have to identify 3 problems.

Identifying the first problem: The first step of the attack on Lane-512 obtains 2^{56} solutions for a first inbound. Steps 2 and 3 merge these 4 lists for finding one solution that verifies also the message expansion. We can apply, as we did before, the algorithm from Section 2.1 where $y = 4$, $x = 16$ and we obtain one solution with complexity 2^{56} in memory and 2^{50} in time.

Identifying the second problem: In the attack on Lane-512, in the Starting Points phase, four lists of values are built, of size 2^{64}. For doing the Merge Lanes and Message Expansion phases, they need a complexity of 2^{128} in time and 2^{128} in memory. We can instead apply the algorithm of Section 2.2 where $l_i = 64$, $|s| = 8$, $x = 8$, and with a complexity of 2^{192} in time and $2^{64} \times 4$ in memory we can obtain the 2^{128} starting points needed for repeating the rest of the attack enough times to find one solution for the whole path (and so a semi-free-start collision). We do not need to store these 2^{128} starting points, because we can perform the rest of the attack as soon as we find one. This way, the memory complexity does not go beyond 2^{64}, and the time complexity, though higher, won't be the bottleneck.

Identifying the third problem: In [10], the second merge of inbound phases (that finds a collision between two lanes) needs a memory of $2^{96} \times 4$. With the previous improvements, the memory

needed is 2^{64}, so we want to reduce the memory needs of this last phase to 2^{64}. We have three lists of 2^{32} elements for each of the two lanes of the same branch (6 in total). Instead of merging the three lists into a new one of size 2^{96}, as done in [10], we can apply the algorithm from Section 4.2 and obtain the wanted memory complexity. This way we will only store a list of 2^{64} elements. Next, we just have to try all the possibilities for the first lane and try to find a match. So, applying the algorithm from Section 4.2 we will have the same time complexity as before but a memory complexity of 2^{64} instead of 2^{96}.

Acknowledgements

The author would like to thank Willi Meier, Thomas Peyrin and Andrea Röck for many helpful comments and discussions.

References

1. Barreto, P.S.L.M., Rijmen, V.: The Whirlpool Hashing Function, revised in 2003.
2. Benadjila, R., Billet, O., Gilbert, H., Macario-Rat, G., Peyrin, T., Robshaw, M., Seurin, Y.: Sha-3 proposal: ECHO. Submission to NIST (updated) (2009)
3. Canniere, C.D., Sato, H., Watanabe, D.: Hash Function Luffa: Specification. Submission to NIST (Round 2) (2009)
4. Canteaut, A., Naya-Plasencia, M.: Internal collision attack on Maraca. In: Seminar 09031, Symmetric Cryptography. Dagstuhl Seminar Proceedings (2009)
5. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl – a SHA-3 candidate. Submitted to the SHA-3 competition, NIST (2008), http://www.groestl.info
6. Gilbert, H., Peyrin, T.: Super-Sbox Cryptanalysis: Improved Attacks for AES-like permutations. In: FSE. LNCS (2010), to appear
7. Indesteege, S.: The Lane hash function. Submitted to the SHA-3 competition, NIST (2008), http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf
8. Khovratovich, D., Naya-Plasencia, M., Røck, A., Schläffer, M.: Cryptanalysis of Luffa v2 components. In: SAC. Lecture Notes in Computer Science (2010), to appear
9. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In: ASIACRYPT. LNCS, vol. 5912, pp. 126–143. Springer (2009)
10. Matusiewicz, K., Naya-Plasencia, M., Nikolic, I., Sasaki, Y., Schläffer, M.: Rebound Attack on the Full Lane Compression Function. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 5912, pp. 106–125. Springer (2009)
11. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In: Fast Software Encryption - FSE 2009. Lecture Notes in Computer Science, vol. 1008. Springer (5665)
12. Mendel, F., Peyrin, T., Rechberger, C., Schläffer, M.: Improved cryptanalysis of the reduced Grøstl compression function, ECHO permutation and AES block cipher. In: Jacobson, Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) Selected Areas in Cryptography. LNCS, vol. 5867, pp. 16–35. Springer (2009)
13. Peyrin, T.: Improved Differential Attacks for ECHO and Grøstl. In: Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6223, pp. 370–392. Springer (2010)
14. Rijmen, V., Toz, D., Varici, K.: Rebound Attack on Reduced-Round Versions of JH. In: FSE. Lecture Notes in Computer Science (2010), to appear
15. Sasaki, Y., Li, Y., Wang, L., Sakiyama, K., Ohta, K.: Non-Full-Active Super-Sbox Analysis Applications to ECHO and Grøstl. In: ASIACRYPT. Lecture Notes in Computer Science (2010), to appear
16. Wagner, D.: A generalized birthday problem. In: CRYPTO. Lecture Notes in Computer Science, vol. 2442, pp. 288–303. Springer (2002)
17. Wu, H.: The hash function JH. Submission to NIST (updated) (2009), http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh_round2.pdf
18. Wu, S., Feng, D., W.Wu: Cryptanalysis of the Lane hash fonction. In: SAC 2009 - Selected Areas in Cryptography. Lecture Notes in Computer Science, Springer (2009)