How to Improve Rebound Attacks

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 large number of cases that the complexities of existing attacks can be improved.
This is done by identifying problems that optimally adapt to the cryptanalytic situation, and by using better algorithms to find solutions for the differential path. Our improvements affect one particular operation that appears in most rebound attacks and which is often the bottleneck of the attacks. This operation, which varies depending on the attack, can be roughly described as *merging* large lists. As a result, we introduce new general purpose algorithms for enabling further rebound analysis to be as performant as possible. We illustrate our new algorithms on real hash functions.

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

1 Introduction

The rebound attack is a recent technique introduced in [13] by Mendel *et al.* It was conceived to analyze AES-like hash functions (like Grøstl [7] in [14, 8, 16], Echo [2] in [14, 8, 18], Whirlpool [1] in [11]). A rebound attack is composed of two parts: the inbound phase and the outbound phase. The aim of the inbound phase is to find, at a low cost, a large 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 then uses these values to perform an attack.

This technique has been applied to other algorithms with inner permutations which are not AES-like; for instance it has been applied to JH [21] (reduced to 22 rounds) in [17] and Luffa [4] (reduced to 7 rounds) in [10]; both of those hash functions use Sboxes of size 4×4 and have a linear part in which the mixing is done in a very different way than in the AES. The hash function LANE [9], which includes several AES states, each treated by the AES round transformation, and

*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.

a different transformation for mixing these states has also been analysed in [12, 22] using rebound attacks.

In these cryptanalysis results, the rebound attack technique needs to be refined and adapted to each case, but all of them follow the same scheme: first find a differential path, then find solutions verifying this differential path. This paper focuses on optimizing the latter part. In all the previously mentioned cryptanalysis, that part involves enumerating, from a very large set of possible candidates represented as a cross product of lists, all those that verify a given relation. We call this operation *"merging" the lists*. The merging problem can be described more formally as follows.

Merging problem with respect to t: Let t be a Boolean function taking N k-bit words as input, *i.e.* $t : (\{0,1\}^k)^N \to \{0,1\}$. Let $L_1, \ldots, L_N$ be N given lists of k-bit words drawn uniformly and independently at random from $\{0,1\}^k$. We assume that the probability over all N-tuples X in $L_1 \times \ldots \times L_N$ that $t(X) = 1$ is P_t. For any given function t and any given N-tuple of lists $(L_1, \ldots, L_N)$ the *merging problem* consists in finding the list $\mathcal{L}_{sol}$ of all $X \in L_1 \times \ldots \times L_N$ satisfying $t(X) = 1$. We call this operation *merging* the lists $L_1, \ldots, L_N$ to obtain $\mathcal{L}_{sol}$.

It is assumed that the image of a given input under t can be easily computed. In the following, the size of a list L is denoted by $|L|$. A brute force method for solving this problem therefore consists in enumerating all the $|L_1| \times \ldots \times |L_N|$ inputs, in computing t on all of them and in keeping the ones verifying $t = 1$. Note that, in the lack of any additional information on t, it is theoretically impossible to do better. However, in practice, the function t often has a set of properties which can be exploited to optimize this approach. We aim at reducing the number of candidates which have to be examined, in some cases by a preliminary sieving similar to the one used in [5]. This paper presents such optimization techniques, that, when applied to most of the rebound attacks published on the SHA-3 candidates, yield significant improvements in the overall time and/or memory complexities of the attack, as shown on Table 1. In this table we can see that we have considered the best existing attacks against four hash functions and the best rebound attack on a fifth (two of them are finalists and two are second-round candidates of the SHA-3 competition), where by best attack we denote the one on the highest number of rounds. We have been able to improve their complexities by scrutinizing the original attack and finding a more efficient algorithm for obtaining the solutions for the differential path. Most of the time the improvement relies on a better *merging* of the lists, and sometimes it is due to the use of more adequate conditions in the general algorithm. Let us recall here that the aim is to find *all* the N-tuples that verify $t = 1$ for a complex function t, which is significantly different from finding just *one* (or few) of them for a linear t such as in [20, 19, 6, 3]. As in the previous rebound analysis, we will work throughout the paper with average values in the probabilistic cases.

In Section 2, we define *Problem 1* that corresponds to functions t with a particular form, and we propose three generic algorithms to solve it. These 3 algorithms have different optimal scenarios. Some examples of applications are given. In Section 3 we define *Problem 2* and propose the *stop-in-the-middle algorithms* for solving it. We also present two concrete algorithms in this family applied to the scenarios of ECHO and Lane. In Section 4 we show briefly how applying these algorithms combined with an appropriate definition and decomposition of the problem in each case, allows us to improve the complexities of the best known rebound attacks on 5 SHA-3 candidates. Besides the results in

Table 1. Improvements on best known attacks. The highlighted values are the improved complexities. For Luffa we consider the best known rebound attack where the complexities presented in the second row have already been obtained in [10] by a dedicated algorithm similar to our general approach.

Hash function	SHA3 Round	Best Known Analysis	Rounds / total	Previous			This paper	
				Time	Memory	Ref.	Time	Memory
JH	Final	semi-free-start coll.	16 / 42	2^{190}	2^{104}	[17]	$\mathbf{2^{97}}$	$\mathbf{2^{97}}$
JH		semi-free-start near coll.	22 / 42	2^{168}	$2^{143.70}$	[17]	$\mathbf{2^{96}}$	$\mathbf{2^{96}}$
Grøstl-256	Final*	(compr. function property)	10 / 10	2^{192}	2^{64}	[16]	$\mathbf{2^{182}}$	2^{64}
Grøstl-256		(internal permutation dist.)	10 / 10	2^{192}	2^{64}	[16]	$\mathbf{2^{175}}$	2^{64}
Grøstl-512		(compr. function property)	11 / 14	2^{640}	2^{64}	[16]	$\mathbf{2^{630}}$	2^{64}
ECHO-256	2^{nd}	internal permutation dist.	8 / 8	2^{182}	2^{37}	[18]	$\mathbf{2^{151}}$	2^{67}
Luffa	2^{nd}	semi-free-start coll.	7 / 8	2^{132}	$2^{68.8}$	[10]	$\mathbf{2^{112.9}}$	$2^{68.8}$
							$\mathbf{(2^{104})}$	(2^{102})
Lane-256	1^{st}	semi-free-start coll.	6+3 / 6+3	2^{96}	2^{88}	[12]	$\mathbf{2^{80}}$	$\mathbf{2^{66}}$
Lane-512		semi-free-start coll.	8+4 / 8+4	2^{224}	2^{128}	[12]	2^{224}	$\mathbf{2^{66}}$

* The Grøstl analysis does not apply after the final round tweak.

Table 1, the main interest of this paper is to present a general framework for improving rebound attacks. We introduce several new algorithms that considerably improve the overall effectiveness when the attack needs to *merge* large lists. We provide a formal definition of the field of application of those algorithms, and describe them as a set of constraints on t, in hope that designers of rebound attacks will be able to easily *identify* scenarios where one of these algorithms, or variants, may be applied. This was motivated by our own research path, when we realized that a generalization of the techniques leveraged in specific cases allowed us to find similar improvements in almost all of the rebound attacks that we have studied so far.

2 When t is Group-Wise

In some cases we can considerably reduce the complexity of the *merging* problem by redefining it into a more concrete one. We consider here a very common case that will appear in many rebound scenarios, as we will later show with the examples. This case corresponds to a function t that can be decomposed in smaller functions. After introducing the general problem, we will illustrate it with an example. Though we preferred to state the problem in full generality for any possible N, in the concrete rebound examples that we studied, the number of lists N was either 2, 4 or 6. Also, the elements of each list can be decomposed in sets of small size s, where s is typically the size of the involved Sbox; and z is the number of such sets involved [1] in the function t.

Problem 1: Let $L_1, \ldots, L_N$ be N lists of size $2^{l_1}, \ldots, 2^{l_N}$ respectively, where the elements are drawn uniformly and independently at random from $\{0,1\}^k$.
Let t be a Boolean function, $t : \left(\{0,1\}^k\right)^N \to \{0,1\}$ for which there exists $N' < N$, an integer z and some triples of functions $t_j : \{0,1\}^{2s} \to \{0,1\}$, $f_j : (\{0,1\}^k)^{N'} \to \{0,1\}^s$ and $f'_j : (\{0,1\}^k)^{(N-N')} \to \{0,1\}^s$ for $j = 1, \ldots, z$ such that, $\forall\, (\boldsymbol{x_1}, \ldots, \boldsymbol{x_N}) \in L_1 \times \ldots \times L_N$:

$$t(\boldsymbol{x_1}, \ldots, \boldsymbol{x_N}) = 1 \quad \Leftrightarrow \quad \begin{array}{l} \forall j = 1, \ldots, z, \\ \begin{cases} t_j(v_j, v'_j) = 1 \\ \text{with } v_j = f_j(\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N'}}) \\ \text{and } \; v'_j = f'_j(\boldsymbol{x_{N'+1}}, \ldots, \boldsymbol{x_N}) \end{cases} \end{array}$$

Let P_t be the probability that $t = 1$ for a random input.
Problem 1 consists in *merging* these N lists to obtain the set $\mathcal{L}_{sol}$, of size $P_t 2^{\sum_{i=1}^{N} l_i}$, of all N-tuples of $(L_1 \times \ldots \times L_N)$ verifying $t = 1$.

Reduction from N to 2: For any $N \geq 2$ *Problem 1* can be reduced to an equivalent and simplified problem with $N = 2$, *i.e.* merging two lists L_A and L_B, which consist of elements in $\left(\{0,1\}^s\right)^z$ corresponding to $\boldsymbol{x_A} = \boldsymbol{v} = (v_1, \ldots, v_z)$ and $\boldsymbol{x_B} = \boldsymbol{v'} = (v'_1, \ldots, v'_z)$, with respect to the function $\boldsymbol{x_A}, \boldsymbol{x_B} \mapsto \Pi_{j=1}^{z} t_j(v_j, v'_j)$. The reduction is performed as follows:

1. Build a table T_A^* of size $2^{\sum_{i=1}^{N'} l_i}$ storing each element $\boldsymbol{e_A} = (\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N'}})$ of $L_1 \times \ldots \times L_{N'}$, indexed[2] by the value of $(f_1(\boldsymbol{e_A}), \ldots, f_z(\boldsymbol{e_A}))$, *i.e.* $(v_1, \ldots, v_z)$. Store the corresponding $(v_1, \ldots, v_z)$ in a list L_A. Note that several $\boldsymbol{e_A}$ may lead to the same value of $(v_1, \ldots, v_z)$.
2. Build a similar table T_B^* of size $2^{\sum_{i=N'+1}^{N} l_i}$ storing each element $\boldsymbol{e_B} = (\boldsymbol{x_{N'+1}}, \ldots, \boldsymbol{x_N})$ of $L_{N'+1} \times \ldots \times L_N$, indexed by $(f_1(\boldsymbol{e_B}), \ldots, f_z(\boldsymbol{e_B}))$, *i.e.* $(v'_1, \ldots, v'_z)$. Store $(v'_1, \ldots, v'_z)$ in a list L_B.
3. *Merge* L_A and L_B with respect to $\Pi_{j=1}^{z} t_j$ and obtain $\mathcal{L}_{sol}$.
4. Build $\mathcal{L}_{sol}^*$ by iterating over each pair $((v_1, \ldots, v_z), (v'_1, \ldots, v'_z))$ of $\mathcal{L}_{sol}$, and adding the set of all $(\boldsymbol{x_1}, \ldots, \boldsymbol{x_{N'}}, \boldsymbol{x_{N+1}}, \ldots, \boldsymbol{x_N}) \in T_A^*\left[(v_1, \ldots, v_z)\right] \times T_B^*\left[(v'_1, \ldots, v'_z)\right]$. $\mathcal{L}_{sol}^*$ is the solution to the original problem.

Let $2^{T_{\mathrm{merge}}}, 2^{M_{\mathrm{merge}}}$ be the time and memory complexities of step 3. The total time complexity of solving *Problem 1* is $\mathcal{O}(sz 2^{\sum_{i=1}^{N'} l_i} + sz 2^{\sum_{i=N'+1}^{N} l_i} + 2^{T_{\mathrm{merge}}} +$

[1] Sometimes, elements are only partially involved in t.

[2] Here and in the following sections we can use standard hash tables for storage and lookup in constant time, since the keys are integers.

$P_t 2^{\sum_{i=1}^{N} l_i}$) where the last term comes from the fact that only the N-tuples satisfying $t = 1$ are examined at step 4 because of the sieve applied at step 3. The proportion of such tuples is then P_t. The memory complexity [3] is $\mathcal{O}((zs + N'k)2^{\sum_{i=1}^{N'} l_i} + (zs + (N - N')k)2^{\sum_{i=N'+1}^{N} l_i} + 2^{M_{\text{merge}}} + P_t 2^{\sum_{i=1}^{N} l_i})$, where the last term appears only when the solutions must be stored.

Using the brute force approach, $2^{T_{\text{merge}}}$ would be $2^{l_A + l_B}$ where 2^{l_A} (respectively 2^{l_B}) denotes the size of L_A (L_B), and $2^{M_{\text{merge}}}$ would be negligible. We present in the following sections some algorithms for solving *Problem 1* considering $N = 2$ with L_A and L_B, that provide better complexities than the brute force approach. Note that the roles of L_A and L_B are assigned by choice to obtain the best overall complexity. Those algorithms can be applied for obtaining a smaller $2^{T_{\text{merge}}}$ when $N > 2$.

2.1 Basic Algorithm for Solving *Problem 1*: Instant Matching

As s is typically very small we can enumerate the solutions (v_j, v'_j) of $t_j(v_j, v'_j) = 1$ and store them in tables T_j of size $\leq 2^{2s}$, indexed by v'_j. This costs $\mathcal{O}(z \cdot 2^{2s})$ in time and memory. We propose in Fig. 1 a first algorithm for solving *Problem 1*, which has lower complexity than the brute-force approach. Although being the simplest algorithm presented in this paper, it has not been applied in critical steps of some of the previously mentioned attacks, though it could yield significant improvements.

Fig. 1 Instant matching algorithm.

Require: Two lists L_A, L_B and a Boolean function t as described in *Problem 1*.
Ensure: The returned list $\mathcal{L}_{sol}$ will contain all elements of $L_A \times L_B$ verifying t.

```
1: for j from 1 to z do
2:    for all (v_j, v'_j) in {0,1}^s × {0,1}^s do
3:       if t_j(v_j, v'_j) = 1, then add v_j to T_j[v'_j].
4: for each (v'_1, ..., v'_z) ∈ L_B do
5:    Empty L_aux.
6:    for j from 1 to z do
7:       if T_j[v'_j] is empty, then go to 4.
8:    Add all tuples (v_1, ..., v_z) verifying ∀j v_j ∈ T_j[v'_j] to L_aux.
9:    for each (v_1, ..., v_z) in L_aux do
10:      if (v_1, ..., v_z) ∈ L_A then
11:         Add (v_1, ..., v_z, v'_1, ..., v'_z) to L_sol.
12: Return L_sol.
```

Let 2^{-p_j} be the probability over all pairs (v_j, v'_j) that $t_j(v_j, v'_j) = 1$. The relationship between t and the $(t_j)_{1 \leq j \leq z}$ implies that $\sum_{j=1}^{z} p_j = -\log_2(P_t)$ where P_t is the probability that $t = 1$.

[3] The first two terms, corresponding to the storage of T_A^* and T_B^* could be avoided if they were the bottleneck by slightly increasing the time complexity by a factor of 2.

Let us determine the average size of L_{aux}. The average size of $T_j[v'_j]$ over all v'_j is 2^{s-p_j}. Then the average size of L_{aux} is $2^{zs-\sum_{j=1}^{z} p_j} = P_t 2^{zs}$. It follows that the time complexity of the algorithm is $\mathcal{O}(z2^s + zP_t2^{l_B+zs})$ and is proportional to the product of the size of L_B by the average size of [4] L_{aux}. The memory complexity is $\mathcal{O}(z2^s + 2^{l_A} + 2^{l_B} + P_t2^{l_A+l_B})$. In some cases, the last term can disappear, namely if we do not need to store the list $\mathcal{L}_{sol}$, but just use each solution as soon as it is obtained. The same way, the list L_B does not need to be stored, if it can be given on the fly.

We now describe a concrete example of application of the *instant-matching algorithm* in a case included in a particular rebound attack, improving its complexity. In the extended version of this paper [15] two more examples are provided in Appendix A, where it clearly appears that identifying and isolating the most appropriate problem (or problems) to solve is of major importance. These two last examples might help also to understand the role of f_j and f'_j.

Example 1: Application of the Instant Matching Algorithm We use here a case presented in the analysis of JH [17] which is the attack on 8 rounds using one inbound when the dimension of a block of bits denoted by d is 4. Here we improve step 3 of the attack, which is also the bottleneck in time complexity. Two lists are given, L_A and L_B of size $2^{24.18}$ elements each. The aim of step 3 is to *merge* those lists, *i.e.* find all pairs $(\boldsymbol{v}, \boldsymbol{v'}) \in L_A \times L_B$ verifying 10 conditions on groups of $s = 4$ bits of $(\boldsymbol{v}, \boldsymbol{v'})$.

In [17] this is solved by exhaustive search, *i.e.* all possible pairs are examined and only the ones that verify the 10 conditions are kept, which has cost $2^{48.36}$. We can improve this complexity by applying the instant-matching algorithm: first, we notice that 6 out of these 10 conditions can be written as

$$t_j(v_j, v'_j) = 1, \forall j \in \{1, \ldots, 6\},$$

where variables v_j and v'_j represent groups of differences of 4 bits. The functions t_j return 1 when the linear function of JH, L, applied to v_j and v'_j produces 4 bits out of 8 without difference in the wanted positions. Those functions t_j can be computed directly by using a precomputed table of size 2^8.

This is an instance of *Problem 1* with the parameters: $z = 6$ (corresponding to the number of relations $t_1, \ldots, t_6$), and $p_j = 3.91\ \forall j$. Hence $P_t 2^{zs} = 2^{0.09 \cdot 6} = 2^{0.54} \simeq 1.45$. The instant-matching algorithm allows us to find all pairs satisfying these 6 conditions with a complexity of $2^{27.8}$ in time and no additional memory. We then obtain $2^{24.9}$ pairs of elements that pass the first 6 conditions. To complete step 3 of the attack, we evaluate the 4 remaining conditions for each pair, for a global complexity of $2^{24.9}$.

To summarize, we were able to resolve step 3 of the attack with a time complexity of about $2^{27.8}$, improving significantly the complexity of $2^{48.36}$ given in [17].

[4] The cost of building and storing the lists $T_j[v'_j]$ is negligible.

2.2 Solving Problem 1 when $P_t 2^{zs} > 2^{l_A}$: Gradual Matching

In Fig. 2 we present an algorithm for solving *Problem 1* that is useful in cases where the average size of L_{aux} exceeds the size of L_A, *i.e.*[5] $P_t 2^{zs} > 2^{l_A}$. In this case the instant-matching algorithm has a higher complexity than the exhaustive search. This is why here, instead of directly matching the z groups that appear in relation t, we will first match the $z' < z$ ones, and next, the $z - z'$ remaining ones. We present here how to use one step of the *gradual-matching algorithm* for solving Problem 1. This algorithm reminds the method used in Example 1 where the problem is first solved with only 6 relations. But the difference is that the remaining $z - z'$ relations can also be written in the form needed for *Problem 1* and $P_t 2^{zs} > 2^{l_A}$. Let us suppose that we choose z' so that $z's < l_A$ (the best value for z' depends on the situation).

Fig. 2 Gradual matching algorithm.

Require: Two lists L_A and L_B and a function t as described in *Problem 1*.
Ensure: List $\mathcal{L}_{sol} \subset L_A \times L_B$ of all elements verifying t.

```
for j from 1 to z do
    for all (v_j, v'_j) in {0,1}^s × {0,1}^s do
        if t_j(v_j, v'_j) = 1, then add v_j to T_j[v'_j].
for each α = (α_1, ..., α_z') in ({0,1}^s)^z' do
    Empty L_aux.
    Consider the sublist L_B(α) of all elements in L_B with (v'_1, ..., v'_z') = α.
    for each (v_1, ..., v_z') in T_1[α_1] × ... × T_z'[α_z'] do
        add (v_1, ..., v_z') to L_aux.
    for each γ = (γ_1, ..., γ_z') in L_aux do
        Consider the sublist L_A(γ) of all elements of L_A with (v_1, ..., v_z') = γ.
        Merge L_A(γ) with L_B(α) with respect to t' = Π_{j=z'+1}^{z} t_j.
        Add the solutions to L_sol.
Return L_sol, containing about P_t 2^(l_A+l_B) elements.
```

Let 2^{merge} be the time complexity of *merging* once lists $L_B(\alpha)$ and $L_A(\gamma)$ as defined in Fig. 2. Since their respective average sizes are $2^{l_A - z's}$ and $2^{l_B - z's}$ the complexity of the brute force is $2^{l_A + l_B - 2z's}$. It can be improved by using one of the proposed algorithms from this section but it cannot be smaller than the size of the resulting merged list, *i.e.* $2^{l_A + l_B - 2z's - \sum_{j=z'+1}^{z} p_j}$. Now the average size of L_{aux} [6] is $\mathcal{S} = 2^{z's - \sum_{j=1}^{z'} p_j}$. Then, the time complexity of this algorithm is $\mathcal{O}(z2^s + 2^{z's}(z' + \mathcal{S}2^{\text{merge}}))$. It is worth noticing that this complexity corresponds to $z'2^{z's} + 2^{l_A + l_B - \sum_{j=1}^{z'} p_j}$ when the intermediate lists are merged by the

[5] When $P_t 2^{zs}$ is close to 2^{l_A} this algorithm might also outperform the instant-matching technique.

[6] Here and in the previous section, there is no need for storing L_{aux}, as each element can be treated as soon as it is obtained, but these auxiliary lists are very useful for describing the complexities.

brute force algorithm and to $z'2^{z's} + P_t 2^{l_A+l_B}$ if they are merged by an optimal algorithm. The memory complexity is $\mathcal{O}(z2^s + 2^{l_A} + 2^{l_B} + \mathcal{S} + P_t 2^{l_A+l_B})$. Again, in some cases, the last term can disappear, if we do not need to store the list $\mathcal{L}_{sol}$, but just use the solutions on the fly.

2.3 Time-Memory Trade-Offs when $P_t 2^{zs} > 2^{l_A}$: Parallel Matching

The *parallel-matching algorithm* improves the time complexity of the gradual-matching by a time-memory trade-off and can be applied in the same situations. It is a generalization of an algorithm proposed in [10]. As the gradual-matching algorithm this algorithm first finds elements that verify $t_j = 1$ for $j \in \{1, \ldots, z'\}$ and then, for each of them, it checks if the remaining $(z - z')$ relations are also verified. However, in this algorithm, the matching of the z' relations is done in parallel for n and m relations, so that $z' = m + n$. The motivation of choosing different variables for n and m is showing that there is no need for them to be the same when applying the algorithm. We choose n so that $n < z$, $ns < l_A$

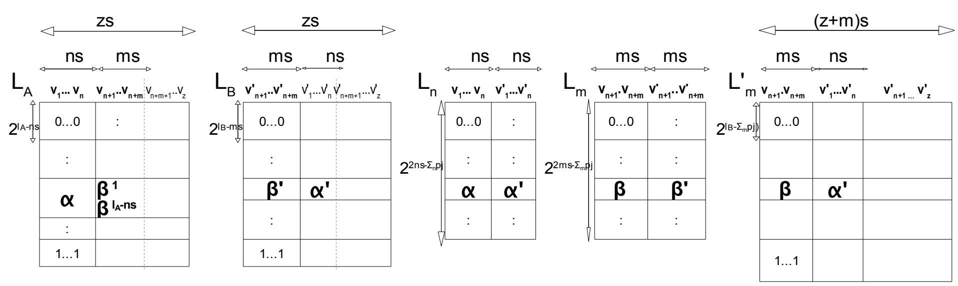

Fig. 3: Representation of the parallel-matching algorithm.

and $ns < l_B$, and in the same way, we choose m $(n + m = z' \leq z)$. This algorithm will be explained with ordered lists, as it is more graphical and helps the understanding. However, since we can perform it with hash tables indexed by the values we want to have ordered, we do not need to take into account the logarithmic terms for ordering and searching in the final complexity. First we build the lists that we will use and that are represented in Fig. 3:

- We order the list L_A by the first n groups $(v_1, \ldots, v_n)$. L_A has $2^{l_A - sn}$ elements in average corresponding to a given value of these n groups.
- We order the list L_B by the next m groups $(v'_{n+1}, \ldots, v'_{n+m})$. L_B has $2^{l_B - sm}$ elements in average corresponding to a given value of these m groups.
- We build the list L_n of size $2^{2ns - \sum_{j=1}^{n} p_j}$ formed by all $(v_1, \ldots, v_n, v'_1, \ldots, v'_n)$ with $v_j \in T_j[v'_j]$ for all $1 \leq j \leq n$. All the elements from this list satisfy $t_j(v_j, v'_j) = 1$ for $j \in [1, \ldots, n]$.

- We build the list L_m of size $2^{2ms-\sum_{j=n+1}^{n+m} p_j}$ formed by all $(v_{n+1}, \ldots, v_{n+m}, v'_{n+1}, \ldots, v'_{n+m})$ with $v_j \in T_j[v'_j]$ for all $(n+1) \leq j \leq (n+m)$. All the elements from this list satisfy $t_j(v_j, v'_j) = 1$ for $j \in [n+1, \ldots, n+m]$.
- From L_m and L_B we build L'_m as follows: for each (β, β') in L_m, we add to L'_m all elements $(\beta, v'_1, \ldots, v'_z)$ of L_B such that $(v'_{n+1}, \ldots, v'_{n+m}) = \beta'$ and we store them ordered by the values of $(\beta, v'_1, \ldots, v'_n)$. The average size of L'_m is $2^{l_B+sm-\sum_{j=n+1}^{n+m} p_j}$. Then we perform the algorithm given in Fig. 4.

Fig. 4 Parallel matching algorithm.

1: **for** each (α, α') in L_n **do**
2:   **for** each $(v_1, \ldots, v_z)$ in L_A with $(v_1, \ldots, v_n) = \alpha$ **do**
3:     **if** L'_m contains any element $(v_{n+1}, \ldots, v_{n+m}, v'_1, \ldots, v'_z)$ starting by $(v_{n+1}, \ldots, v_{n+m}, \alpha')$ **then**
4:       **if** $(v_1, \ldots, v_z, v'_1, \ldots, v'_z)$ satisfies the remaning $(z-n-m)$ conditions **then**
5:         Add $(v_1, \ldots, v_z, v'_1, \ldots, v'_z)$ to $\mathcal{L}_{sol}$.
6: Return $\mathcal{L}_{sol}$ containing about $P_t 2^{l_A+l_B}$ elements.

In total we find the $2^{l_A+l_B-\sum_{j=1}^{z} p_j}$ existing matches, with a complexity of $\mathcal{O}(2^{l_n} + 2^{l_m} + 2^{l_A+l_B-\sum_{j=1}^{n+m} p_j} + 2^{l_A+ns-\sum_{j=1}^{n} p_j} + 2^{l_B+ms-\sum_{j=n+1}^{m} p_j})$ in time and $\mathcal{O}(2^{l_n} + 2^{l_m} + 2^{l_B} + 2^{l_B+ms-\sum_{j=n+1}^{m} p_j} + 2^{l_A+l_B-\sum_{j=1}^{z} p_j})$ in memory, where the last term corresponds to the storage of all solutions, not always needed. In this case, the storage of L_A is not necessary.

2.4 Example 2: Gradual Matching vs Parallel Matching

We are going to apply both previous algorithms to the analysis of Luffa presented in [10]. We are given two lists L_A and L_B of size 2^{67} and $2^{65.6}$. These lists contain elements formed by $z = 52$ groups of differences of $s = 4$ bits. List L_A contains the possible differences for the input of 52 Sboxes. List L_B contains the possible differences for the output of the same 52 Sboxes. For the j-th Sbox, the probability that one input difference can be associated to one output difference is $2^{-p_j} = 2^{-1.23}$. The average size of L_{aux} if we apply the instant-matching algorithm is then $P_t 2^{zs} = 2^{144.04}$. In this case t can be decomposed in 52 t_j, one per Sbox. So $t_j(v_j, v'_j) = 1$ if there exists $x \in \{0,1\}^s$ such that

$$\text{Sbox}(x) \oplus \text{Sbox}(x \oplus v_j) = v'_j.$$

The brute force algorithm for solving this problem has complexity of $2^{65.6+67} = 2^{132.6}$ in time and of $2^{68.8}$ in memory. If we apply the gradual-matching algorithm with $z' = 16$ we have $\mathcal{S} = 2^{44.32}$, and we obtain the $2^{68.8}$ solutions with a time complexity of $2^{112.9}$ and the same memory as before as no additional memory is needed. If instead we apply the parallel-matching algorithm with $m = n = 13$, we can obtain the solutions with a time complexity of 2^{104} and a memory complexity of 2^{102}. Different choices of parameters allow many other time-memory trade-offs, but we just show here the one that provides the lowest time complexity, and so the highest memory needs, for contrast with the gradual matching algorithm.

3 Stop-in-the-Middle Algorithms

In this section we present another case that allows to reduce the complexity of solving the basic problem. It is described in *Problem 2*. Then, we define the main lines of the *stop-in-the-middle algorithms*, that we use for solving *Problem 2*. Next, we present such an algorithm that solves *Problem 2* in the scenario of Lane-256. Then a more complex variant of this algorithm is applied to an ECHO-256 scenario. We believe that, in particular, this kind of algorithms can be adapted and applied to functions that use several AES (like) states in parallel which are then merged at the end of each round.

In the following, we consider a permutation F from $\{0,1\}^{sk}$ to $\{0,1\}^{sk}$ and we assume that there exists a decomposition function ϕ (respectively ψ) of the input of F (respectively the output) in k elements of $\{0,1\}^s$. These two decompositions may be different. Then, instead of the original function F we will now focus on the function $f = \psi \circ\ F \circ \phi^{-1}$ which is a function over $(\{0,1\}^s)^k$ (see Fig. 5). In the following (u, w) denotes the word corresponding to the concatenation of the vectors u and w.

Problem 2: Let z_A and z_B be two integers less than or equal to k . Let L_A be a list of elements in $(\{0,1\}^s)^{z_A}$ and L_B be a list of elements on $(\{0,1\}^s)^{z_B}$. The *Problem 2* consists on finding all triples (a, b, c) with $a \in L_A$, $b \in L_B$ and $c \in L_C = (\{0,1\}^s)^k$ such that

$$f(c) \oplus f(c \oplus (a, 0^{s(k-z_A)})) = (b, 0^{s(k-z_B)}),$$

where there exists the function $F_1 : (\{0,1\}^s)^k \to (\{0,1\}^s)^k$ and some permutations of $\{0,1\}^s$, $g_1, \ldots, g_k$ and $h_1, \ldots, h_k$ over $\{0,1\}^s$ such that

$$f = H \circ F_1 \circ G$$

where

$$\begin{aligned} G : \quad & (\{0,1\}^s)^k \to (\{0,1\}^s)^k \\ & (x_1, \ldots, x_k) \to (g_1(x_1), \ldots, g_k(x_k)) \end{aligned}$$

and

$$\begin{aligned} H : \quad & (\{0,1\}^s)^k \to (\{0,1\}^s)^k \\ & (x_1, \ldots, x_k) \to (h_1(x_1), \ldots, h_k(x_k)) \end{aligned}$$

It is worth noting that we assume that both decompossitions ϕ and ψ have been chosen in an appropriate way such that the z_A words of a (respectively the

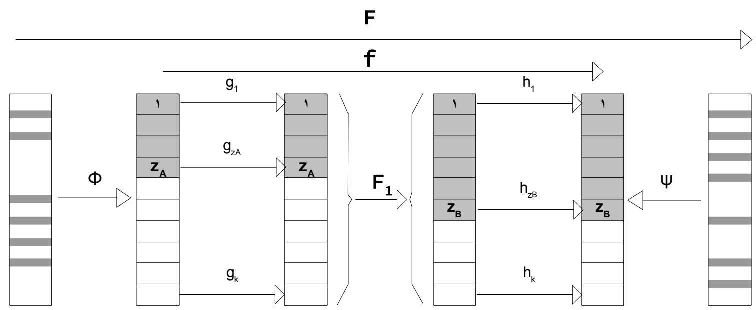

Fig. 5: Representation of F from *Problem 2.*

z_B words of b) correspond to the first words of the input state (respectively of the output state). We call *stop-in-the-middle algorithms* those that solve *Problem 2* following the main general scheme described in Fig. 6. The associated complexities depend on the particular form of F_1, as we show in the next sections.

Fig. 6 General scheme of stop-in-the-middle algorithms.

for each b in L_B **do**
  for each $j \in [1, \dots, z_B]$ **do**
    for each $y_j \in \{0,1\}^s$ **do**
      add $(h_j^{-1}(y_j), h_j^{-1}(y_j) \oplus h_j^{-1}(y_j \oplus b_j))$ to $L_{j,b}$.
for each a in L_A **do**
  for each $i \in [1, \dots, z_A]$ **do**
    for each x_i in $\{0,1\}^s$ **do**
      add $(g_i(x_i), g_i(x_i) \oplus g_i(x_i \oplus a_i))$ to L_i.
  Using the previous lists L_i and $L_{j,b}$, match in the middle using F_1, *i.e.* construct $L_{aux} = \{(x, b_1, \dots, b_{z_B}), x \in (\{0,1\}^s)^k\}$ such that $((F_1[g_1(x_1), \dots, g_{z_A}(x_{z_A}), x^*)], F_1[g_1(x_1 \oplus a_1), \dots, g_{z_A}(x_{z_A} \oplus a_{z_A}), x^*)]) = \big((h_1^{-1}(y_1), \dots, h_{z_B}^{-1}(y_{z_B}), y^*), (h_1^{-1}(y_1 \oplus b_1), \dots, h_{z_B}^{-1}(y_{z_B} \oplus b_{z_B}), y^*)\big)$ for some $x^* \in (\{0,1\}^s)^{k-z_A}$ and $y^* \in (\{0,1\}^s)^{k-z_B}$.
  for all $(x, b_1, \dots, b_{z_B})$ in L_{aux} **do**
    if $b = (b_1, \dots, b_{z_B}) \in L_B$ **then**
      add (a, b, x) to $\mathcal{L}_{sol}$.
Return $\mathcal{L}_{sol}$.

In the cases we have studied and that we detail below, the function f is formed by several AES transformations in parallel. We then expect $2^{l_A+l_B}$ solutions, as for each $a \in L_A$ and each $b \in L_B$ there exists one $c \in L_C$ so that the condition of *Problem 2* holds. The match-in-the-middle step is assumed to be simple due to the simple form of F_1 (typical functions F_1 are linear diffusion layers). For the same reason, L_{aux} can typically be written in a compact way, for example, in several independent lists.

3.1 Algorithm for Lane-256

Each lane of the internal state of Lane-256 is composed of two AES states. An AES state is a state of size 128 bits that can be seen as a 4×4 matrix of bytes. The AES transformations are noted: SB for SubBytes, SR for ShiftRows and MC for MixColumn. The transformation SC mixes the two AES states at the end of each round by interchanging their columns. We consider Fig. 7 that represents a part of the differential path used in [12]. In that attack it was treated as the merging of two inbounds and 2^{64} solutions were found with a complexity of 2^{96} in time and 2^{88} in memory. We consider the scheme represented in Fig. 7 where we have swapped lines and columns for a more easy intuitive understanding (so SR is applied to the columns and MC is applied to the lines).

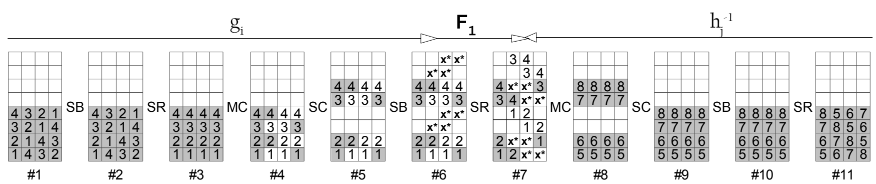

Fig. 7: Differential path associated to the first improvement on the Lane analysis.

Using the example from [12], $l_A = 32$ and $l_B = 32$ and L_C is the list of all possible input values and needs to be neither stored nor computed. We consider that the input state (respectively the output state) of the function f presented in Fig. 7 is decomposed into eight 32-bit words (*i.e.* $s = 32$ and $k = 8$). The input differences and output differences that we consider in L_A and L_B correspond to the first $z_A = z_B = 4$ 32-bit words of the state. In Fig. 7 each one of the $4+4 = 8$ 32-bits active word corresponds to the four active bytes with the same number written on them (1 to 4 for the four active input words and 5 to 8 for the 4 active output words).

With the algorithm described in Fig. 8 we find the 2^{64} solutions with a complexity of 2^{66} in time and 2^{65} in memory. The time complexity associated to the studied path is $z_B 2^{l_B+32} + 2^{l_A+32}$. This comes from the fact that each L_i has average size 2^{16}. Then, $L_{5,6}$ and $L_{7,8}$ have size 2^{l_B+32}. Then the size of both L^0_{aux} and L^1_{aux} is 2^{l_B} since in each we keep the pairs of elements that match on 4 active bytes, and this happens with a probability of 2^{-64} (32 values and 32 differences); and the number of possible pairs is $2^{16+16+l_B+32}$. The memory complexity is $2^{l_B+32+1} + 2^{32+1} + 2^{l_A+l_B}$ for obtaining $2^{l_A+l_B}$ solutions. In [15] a detailed explanation on how this algorithm allows to considerably reduce the

Fig. 8 Algorithm for solving two inbounds of Lane-256.

Require: Function f and lists L_A and L_B of differences in #1 and #11 respectively.
Ensure: List $\mathcal{L}_{sol} = \{(a, b, c)$ such that $f(c \oplus (a, 0^{s(k-z_A)})) \oplus f(c) = (b, 0^{s(k-z_B)})\}$.

for each b in L_B **do**
  for i from 5 to 8 **do**
    for each $y \in \{0,1\}^{32}$ **do**
      if $h_i^{-1}(y) \oplus h_i^{-1}(y \oplus b_i)$ has only the two bytes active (*see state #7*) **then**
        Store $(y, b_i, h_i^{-1}(y), h_i^{-1}(y \oplus b_i))$ in L_i, where the last two terms are truncated to the 2 active bytes.
  for each (y_5, b_5, u_5, w_5) from L_5 and (y_6, b_6, u_6, w_6) from L_6 **do**
    Add $(u_5, w_5, u_6, w_6, y_5, y_6, b_5, b_6)$ in $L_{5,6}$ indexed by u_5, w_5, u_6, w_6.
  for each (y_7, b_7, u_7, w_7) from L_7 and (y_8, b_8, u_8, w_8) from L_8 **do**
    Add $(u_7, w_7, u_8, w_8, y_7, y_8, b_7, b_8)$ in $L_{7,8}$ indexed by u_7, w_7, u_8, w_8.
  Empty L_5, L_6, L_7 and L_8.
for each a in L_A **do**
  for i from 1 to 4 **do**
    for each $x_i \in \{0,1\}^{32}$ **do**
      if $g_i(x_i) \oplus g_i(x_i \oplus a_i)$ has only the two bytes active (*see state #4*) **then**
        Store $(x_i, g_i(x_i), g_i(x_i \oplus a_i))$ in L_i, where the two last terms are truncated to the 2 active bytes.
  for i from 0 to 1 **do**
    for each $(x_{2i+1}, u_{2i+1}, w_{2i+1})$ in L_{2i+1} and $(x_{2i+2}, u_{2i+2}, w_{2i+2})$ in L_{2i+2} **do**
      if there exists an element in $L_{5+2i,6+2i}$ indexed by $(u_{2i+1}, w_{2i+1}, u_{2i+2}, w_{2i+2})$ **then**
        Add $(x_{2i+1}, x_{2i+2}, b_{5+2i}, b_{6+2i})$ to L^i_{aux} indexed by (b_{5+2i}, b_{6+2i}).
  for each (x_1, x_2, b_5, b_6) in L^0_{aux} **do**
    for each (b_7, b_8) such that $(b_5, b_6, b_7, b_8) \in L_B$ **do**
      if there exists an element in L^1_{aux} indexed by (b_7, b_8) **then**
        add $(a, (b_5, b_6, b_7, b_8), (x_1, x_2, x_3, x_4))$ to $\mathcal{L}_{sol}$.
Return $\mathcal{L}_{sol}$.

complexity of the Lane-256 semi-free-start collision presented in [12] is given, when applied jointly with other improvements concerning other steps of the attack.

3.2 Algorithm for ECHO-256

An ECHO-256 state is a state of size 2048 bits that can be seen as a 4×4 matrix of AES states. The ECHO operations BigSR, BigMC and BigSB are similar to the AES ones, but they operate on AES states instead of bytes. A SuperSbox is an Sbox defined by SR ∘ SB ∘ MC ∘ SR ∘ SB. Applied on an AES state, it can be seen as a 32×32 Sbox. We define a *SuperSbox set* as each one of the 4 (in the AES state) sets of bits that act as input and output of the SuperSbox. We define a *BigSuperSbox* as an Sbox defined by BigSR ∘ BigSB ∘ BigMC ∘ BigSR ∘ BigSB. Applied to ECHO it defines 4 sets of size 4 AES-states.

We consider Fig. 9, where each column represents the four AES states that form a BigSuperSbox at a certain state #i, for i from 1 to 13. Each possible differences in #1 in L_A consist of $z_A = 12$ 32-bit words and the possible differences in #13 consist of $z_B = 8$ 32-bit words, where L_B can be written as $L_B = L_{B_1} \times L_{B_2}$ with both L_{B_1} (associated to AES state B_1 in Fig. 9) and L_{B_2} (associated to AES state B_4) are subsets of $(\{0,1\}^{32})^4$ each of size 2^{32} (this is a particular case which has to be adapted in other cases). Finding solutions for this differential path with the previously mentioned conditions is a problem proposed in [18] and was solved in such a way that 2^{32} solutions could be found with a complexity of 2^{128} in time and 2^{37} in memory. We propose here a new algorithm that can solve it for obtaining 2^{64} solutions with the same time complexity and a memory of 2^{67}. Variants of our algorithm can be applied in several cases, like when the transition in #7 to #8 is from 2 active states to 3, or from 1 to 4 or from 4 to 1. Additionally we believe that it can improve the complexity of other future attacks on ECHO-256.

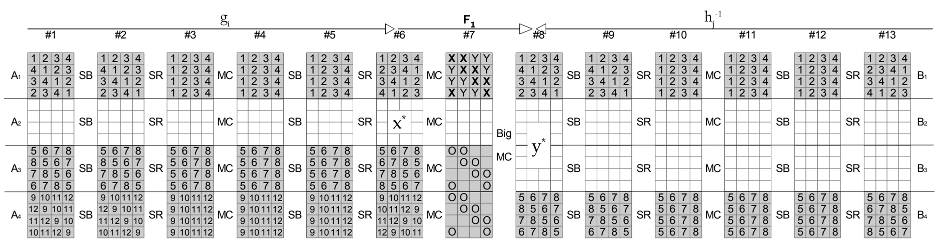

Fig. 9: Differential path on a BigSuperSbox of ECHO-256.

The list L_C contains all the possible values for the input state. This list needs to be neither computed nor stored. Here the aim is to find for each possible (a, b^1, b^2) in $L_A \times L_{B_1} \times L_{B_2}$ the associated c so that $f(c) \oplus f(c \oplus (a, 0^{s(k-z_A)}) = (b^1, b^2, 0^{s(k-z_B)})$. In Fig. 9 we can see how the function f can be written in the way requested by *Problem 2*. We omit the operation BigSR as it does not affect the states, as well as the round keys that are taken into account in the different g_i and h_j. For the sake of simplicity we consider in Fig. 9 the list L_B of possible differences before the last MC of the BigSuperSbox. This can be done by a simple transformation MC^{-1} of the differences in #B'. The grey bytes represent the bytes with differences. We can observe that, from #1 to #6 there are $z_A = 12$ independent active SuperSbox sets ($s = 32$), denoted in Fig. 9 by a number from 1 to 12. To each of these groups we can associate a difference from L_A and a value from L_C at state #1 and we can apply independently g_i, $i \in [1, \ldots, 12]$ to obtain the value and the difference of the group in #6. The same way, from #8 to #13 there are $z_B = 8$ independent active SuperSbox sets and the corresponding functions $h_i^{-1}, i \in [1, \ldots 8]$ that link state #13 with state #8. The function F_1 =MC∘BigMC takes a complete internal state in #6 and computes the corresponding state in #8. Let $f(x) = y$, and let $d_i^{\#7}$ the ith active diagonal in state #7. Without knowing the values of x^* nor of y^* represented in Fig. 9 we can still write the following equations that have to be verified, that are obtained from BigMC, and that are used in the algorithm:

$$2\times d_i^{\#7} \oplus d_{i+4}^{\#7} \oplus d_{i+8}^{\#7} \oplus 9\times d_i^{\#7} \oplus 3\times d_{i+4}^{\#7} \oplus 6\times d_{i+8}^{\#7} = h_i^{-1}(y_i) \oplus 3\times h_{i+4}^{-1}(y_{i+4}) \quad (1)$$

for $i \in 1, \ldots, 4$ where the multiplication corresponds to the one in the definition of MC applied independently to each byte of the diagonal.

We consider that the input state (respectively the output state) of the function presented in Fig. 9 is decomposed into sixteen 32-bit words (*i.e.* $s = 32$ and $k = 16$). The input differences (respectively output differences) that we consider in L_A (L_B) correspond to the first $z_A = 12$ ($z_B = 8$) 32-bit words of the state. In Fig. 9 each one of the 12 (respectively 8) 32-bits active word from the input (respectively the output) corresponds to the four active bytes with the same number written on them (1 to 12 for the twelve active input words and 1 to 8 for the eight active output words).

Let V_X (V_Y, V_O respectively) be the values at the positions in #7 marked with an X (Y, O) and Δ_X (Δ_Y, Δ_O) their differences. Let $\Delta_{j'}^{\#r}$ be an auxiliary variable denoting the difference for the SuperSbox set j' in state $\#r$. The algorithm is described in Fig. 10. So the time complexity is $\mathcal{O}(z_B 2^{l_{B_1}+s} + z_B 2^{l_{B_2}+s} + z_A 2^s + 2^{l_A+64}(2^{l_{B_1}} + 2^{l_{B_2}} + 2^{l_{B_1}} 2^{l_{B_2}} + z_A 2^{64}))$. The memory complexity is $\mathcal{O}(z_B 2^{l_{B_1}+s} + z_B 2^{l_{B_2}+s} + 2^{l_{B_1}+l_{B_2}} + |\mathcal{L}_{sol}|)$. In the case of $l_A = 0$, we will obtain a complexity of 2^{129} in time and 2^{66} in memory for obtaining 2^{64} solutions. This algorithm proposes several trade-offs when changing the values of $|\Delta_X|$, and can be adapted for other forms of L_B.

4 How to improve the best known attacks on five SHA-3 candidates

In this section we enumerate briefly the main algorithms or ideas that we use to improve the best known attacks on the hash functions JH, Grøstl, ECHO, Luffa and Lane as shown on Table 1. In the full version of the paper [15] more detailed descriptions are provided.

- **JH:** To improve the complexities over the ones in [17] we use the instant-matching (as in Section 2.1) and gradual-matching algorithms as well as the fact that we do not merge the lists until we really have to (to keep lists of smaller sizes, with a smaller complexity).
- **Grøstl:** Instead of the initial lists used in [16], we can define them so that we erase the elements that for sure won't verify the outbound part. Having lists of smaller size translates to a smaller complexity.
- **ECHO:** Using conviniently the algorithm from Section 3.2 we provide better trade-offs improving the time complexity from [18].
- **Luffa:** The parallel-matching algorithm is applied in [10], improving the time complexity over the brute force merging method by increasing the memory requirements. If we apply instead the gradual-matching algorithm, the time

Fig. 10 Algorithm for finding solutions for one ECHO BigSuperSbox.

Require: Function f, lists of differences L_A (in #1) and L_{B_1} and L_{B_2} (in #13).
Ensure: $\mathcal{L}_{sol} = \{(a, b^1, b^2, c),$ such that $f(c) \oplus f(c \oplus (a, 0^{s(k-z_A)})) = (b^1, b^2, 0^{s(k-z_B)})\}$

1: **for** j from 1 to 4 **do**
2:   **for** each $y_j \in \{0,1\}^{32}$ and **for** each b^1 from L_{B_1} **do**
3:     Store $(h_j^{-1}(y_j), h_j^{-1}(y_j) \oplus h_j^{-1}(y_j \oplus b_j^1))$ in $L^j_{\#8,b^1}$ (4×2^{32} *lists of size* 2^{32}).
4: **for** j from 5 to 8 **do**
5:   **for** each $y_j \in \{0,1\}^{32}$ and **for** each b^2 from L_{B_2} **do**
6:     Store $(h_j^{-1}(y_j), h_j^{-1}(y_j) \oplus h_j^{-1}(y_j \oplus b_j^2))$ in $L^j_{\#8,b^2}$ (4×2^{32} *lists of size* 2^{32}).
7: **for** each a in L_A **do**
8:   **for** i from 1 to 12 **do**
9:     **for** each $x_i \in \{0,1\}^{32}$ **do**
10:       Store $(g_i(x_i), g_i(x_i) \oplus g_i(x_i, a_i))$ in $L^i_{\#6}$.
11:   **for** Δ_X from 0 to $2^{64} - 1$ (*and not the 128 bits as done in [18]*) **do**
12:     Compute Δ_O and $\Delta^{\#8}_{j'}$ for $j' \in \{1,2,5,6\}$ (*with BigMC and linear condit.*).
13:     **for** each b^1 in L_{B_1} and for $j = [1,2]$ **do**
14:       Find an element in $L^j_{\#8,b^1}$ such that $h_j^{-1}(y_j) \oplus h_j^{-1}(y_j \oplus b_j^1) = \Delta_j^{\#8}$ and store $(h_1^{-1}(y_1), \Delta_1^{\#8}, h_2^{-1}(y_2), \Delta_2^{\#8}, b^1)$ in L_{aux_1}.
15:     **for** each b^2 in L_{B_2} and for $j = [5,6]$ **do**
16:       Find an element in $L^j_{\#8,b^2}$ such that $h_j^{-1}(y_j) \oplus h_j^{-1}(y_j \oplus b_j^2) = \Delta_j^{\#8}$ and store $(h_5^{-1}(y_5), \Delta_5^{\#8}, h_6^{-1}(y_6), \Delta_6^{\#8}, b^2)$ in L_{aux_2}.
17:     **for** each $(h_1^{-1}(y_1), \Delta_1^{\#8}, h_2^{-1}(y_2), \Delta_2^{\#8}, b^1)$ in L_{aux_1} and **for** each $(h_5^{-1}(y_5), \Delta_5^{\#8}, h_6^{-1}(y_6), \Delta_6^{\#8}, b^2)$ in L_{aux_2} **do**
18:       Compute $V_1' = h_1^{-1}(y_1) \oplus 3 \times h_5^{-1}(y_5)$ and $V_2' = h_2^{-1}(y_2) \oplus 3 \times h_6^{-1}(y_6)$, and store $((h_1^{-1}(y_1), \Delta_1^{\#8}, h_2^{-1}(y_2), \Delta_2^{\#8}, b^1), (h_5^{-1}(y_5), \Delta_5^{\#8}, h_6^{-1}(y_6), \Delta_6^{\#8}, b^2))$ in a hash table T indexed by these (V_1', V_2').
19:     **for** Δ_Y from 0 to $2^{64} - 1$ **do**
20:       Determine by BigMC $\Delta^{\#8}_{j'}$ for $j' = 3,4,7,8$; and $\Delta^{\#6}_j$ for $j \in [1, \ldots, 12]$.
21:       **for** i from 1 to 12 **do**
22:         Find the element from $L^i_{\#6}$ such that $g_i(x_i) \oplus g_i(x_i, a_i) = \Delta_i^{\#6}$.
23:       Compute with them by MC the values $d_i^{\#7}$ of the active diagonals in #7 and $V_j = 2 \times d_j^{\#7} \oplus d_{j+4}^{\#7} \oplus d_{j+8}^{\#7} \oplus 9 \times d_j^{\#7} \oplus 3 \times d_{j+4}^{\#7} \oplus 6 \times d_{j+8}^{\#7}$ for $j = 1, 2$.
24:       **if** there is an element such that $V_1' = V_1$ and $V_2' = V_2$ in T (*one on average, determines* b^1 *and* b^2) **then**
25:         Find $(h_{j'}^{-1}(y_{j'}), \Delta_{j'}^{\#8})$ from $L^{j'}_{\#8,b^1}$ for $j' = 3,4$. This implies y_3 and y_4.
26:         Find $(h_{j'}^{-1}(y_{j'}), \Delta_{j'}^{\#8})$ from $L^{j'}_{\#8,b^2}$ for $j' = 7,8$. This implies y_7 and y_8.
27:         **if** with these values of $(h_{j'}^{-1}(y_{j'}),\ j' = 3,4,7,8$ and the ones obtained in step 22 of $g_i(x_i)$ for $i = 3,4,7,8,11,12$, the equation (1) for $i = 3,4$ derived from F_1 can be verified (*happens with a probability of* 2^{-64}) **then**
28:           The value x_* is determined. Add $(x_1, \ldots, x_{z_A}, x^*, a, b^1, b^2)$ to $\mathcal{L}_{sol}$
29: Return $\mathcal{L}_{sol}$, containing about 2^{64+l_A} elements.

complexity can still be better than the brute force one while the memory needs are not increased.

– LANE**:** In the cases of LANE-256 and LANE-512 several improvements are applied at different steps of the attacks from [12]. They use the instant-matching algorithm, as well as some more appropriate ways to formulate the problem, and the algorithms from Section 3.1 and from [15, App.B].

5 Conclusion

The main contribution of this paper is to propose several algorithms for solving the problem which constitutes the bottleneck of most rebound attacks, leading to improvements of the previously known complexities. We also highlight the importance of identifying the situations that could help improving the complexity of this type of attacks. This is often a difficult task due to the high technicality of the attacks and algorithms.

Finally, the previous contributions lead to improvements of most of the best known rebound attacks applied to the SHA-3 candidates JH, Grøstl, Luffa, ECHO-256 and LANE. It is important to point out that we just tried to improve the complexities of existing attacks. However, the work presented in this paper can be very useful for future rebound attacks, in particular we believe that the attacks on JH and on the compression function of ECHO can be improved (extending the number of rounds attacked) by exploiting the algorithms and ideas presented here. Finally, we believe that some of these algorithms, specially those of Section 2, will be applicable in other contexts besides rebound attacks.

Acknowledgements

The author would like to thank Willi Meier, Simon Knellwolf, Marine Minier, Thomas Peyrin, Martin Schläffer, Joana Treger and Fabien Viger for many helpful comments and discussions. A special mention is needed for Anne Canteaut and Andrea Röck for all the help and suggestions to improve this paper.

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. Camion, P., Patarin, J.: The knapsack hash function proposed at Crypto'89 can be broken. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 547, pp. 39–53. Springer (1991)
4. Canniere, C.D., Sato, H., Watanabe, D.: Hash Function Luffa: Specification. Submission to NIST (Round 2) (2009)
5. Canteaut, A., Naya-Plasencia, M.: Structural weaknesses of permutations with low differential uniformity and generalized crooked functions. In: Finite Fields: Theory and Applications - Selected papers from the 9th International Conference Finite

Fields ans applications. Contemporary Mathematics, vol. 518, pp. 55–71. AMS (2010), http://www-rocq.inria.fr/secret/Maria.Naya_Plasencia/papers/canteaut-nayaplasencia.pdf
6. Chose, P., Joux, A., Mitton, M.: Fast correlation attacks: An algorithmic point of view. In: EUROCRYPT. Lecture Notes in Computer Science, vol. 2332, pp. 209–221. Springer (2002)
7. 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
8. Gilbert, H., Peyrin, T.: Super-Sbox Cryptanalysis: Improved Attacks for AES-like permutations. In: FSE. Lecture Notes in Computer Science (2010), to appear
9. Indesteege, S.: The Lane hash function. Submitted to the SHA-3 competition, NIST (2008), http://www.cosic.esat.kuleuven.be/publications/article-1181.pdf
10. Khovratovich, D., Naya-Plasencia, M., Røck, A., Schläffer, M.: Cryptanalysis of Luffa v2 components. In: SAC. Lecture Notes in Computer Science, vol. 6544, pp. 388–409 (2010)
11. Lamberger, M., Mendel, F., Rechberger, C., Rijmen, V., Schläffer, M.: Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In: ASIACRYPT. Lecture Notes in Computer Science, vol. 5912, pp. 126–143. Springer (2009)
12. 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)
13. 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)
14. 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. Lecture Notes in Computer Science, vol. 5867, pp. 16–35. Springer (2009)
15. Naya-Plasencia, M.: How to Improve Rebound Attacks. Cryptology ePrint Archive, Report 2010/607 (2010), http://eprint.iacr.org/2010/607.pdf, extended version.
16. 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)
17. Rijmen, V., Toz, D., Varici, K.: Rebound Attack on Reduced-Round Versions of JH. In: FSE. Lecture Notes in Computer Science, vol. 6147, pp. 286–303 (2010)
18. 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, vol. 6477, pp. 38–55 (2010), to appear
19. Schroeppel, R., Shamir, A.: A T=O($2^{n/2}$), S=O($2^{n/4}$) algorithm for certain NP-complete problems. SIAM J. Comput. 10(3), 456–464 (1981)
20. Wagner, D.: A generalized birthday problem. In: CRYPTO. Lecture Notes in Computer Science, vol. 2442, pp. 288–303. Springer (2002)
21. Wu, H.: The hash function JH. Submission to NIST (updated) (2009), http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh_round2.pdf
22. Wu, S., Feng, D., Wu, W.: Cryptanalysis of the Lane hash fonction. In: SAC 2009 - Selected Areas in Cryptography. Lecture Notes in Computer Science, Springer (2009)