Refinements of the k-tree Algorithm for the Generalized Birthday Problem

Ivica Nikolić[1] and Yu Sasaki[1,2]

[1] Nanyang Technological University, Singapore
[2] NTT Secure Platform Laboratories
inikolic@ntu.edu.sg sasaki.yu@lab.ntt.co.jp

Abstract. We study two open problems proposed by Wagner in his seminal work on the generalized birthday problem. First, with the use of multicollisions, we improve Wagner's 3-tree algorithm. The new 3-tree only slightly outperforms Wagner's 3-tree, however, in some applications this suffices, and as a proof of concept, we apply the new algorithm to slightly reduce the security of two CAESAR proposals. Next, with the use of multiple collisions based on Hellman's table, we give improvements to the best known time-memory tradeoffs for the k-tree. As a result, we obtain the a new tradeoff curve $T^2 \cdot M^{\lg k - 1} = k \cdot N$. For instance, when $k = 4$, the tradeoff has the form $T^2 M = 4 \cdot N$.

Keywords: Generalized birthday problem, k-list problem, k-tree algorithm, time-memory tradeoff

1 Introduction

Arguably, the most popular problem in private key cryptography is the collision search problem. It appears frequently not only in its classical usage, e.g. finding collisions for hash functions, but also as an intermediate subproblem of a wider cryptographic problem. The collision search has been widely studied and well understood. Besides this problem, and along with the search of multicollisions and multiple collisions, perhaps the next most popular is the generalized birthday problem (GBP).

The GBP is defined as follows: given k lists of random elements, choose a single element in each list, such that all the chosen elements sum up to a predefined value. Wagner is the first to investigate the GBP for all values of k and as an independent problem. In his seminal paper [34], he proposes an algorithm to solve GBP for all values of k and shows wide variety of applications ranging from blind signatures, to incremental hashing, low weight parity checks, and cryptanalysis of various hash functions.

Prior to Wagner, GBP problem has been mostly studied in the context of its application and only for a concrete number of lists (usually four lists, i.e. $k = 4$). Schroeppel and Shamir [31] find all solutions to the 4-list problem. Bernstein [4] uses similar algorithm to enumerate all solutions to a particular equation. Boneh, Joux and Nguyen [11] use Schroeppel and Shamir's algorithm for solving integer knapsacks as well as Bleichenbacher [9] in his attack on DSA. Chose, Joux, and Mitton [12] use it to speed up search for parity checks for stream cipher cryptanalysis. Joux and Lercier [22] use related ideas in point-counting algorithms for elliptic curves. Blum, Kalai, and Wasserman [10] apply it to find the first known subexponential algorithm for the learning parity with noise problem. Ajtai, Kumar, and Sivakumar findings [1] base on Blum, Kalai, and Wasserman's algorithm as a subroutine to speed up the shortest lattice vector problem.

To solve the k-list problem, Wagner proposes a so-called k-tree algorithm. In a nutshell, the k-tree is a divide and conquer approach and at each step it operates on only two lists. The step operations are based on a simple collision search. When the k lists are composed of n-bit words, Wagner's k-tree algorithm solves the GBP in $\mathcal{O}(k \cdot 2^{\frac{n}{\lfloor \lg k \rfloor + 1}})$ time and memory and requires lists of around $2^{\frac{n}{\lfloor \lg k \rfloor + 1}}$ elements [3].

Even though the GBP has been shown to be very important to many problems in cryptography, more than a decade after its publication neither significant improvement to the k-tree algorithm nor other dedicated algorithms have emerged. However, moderate improvements and refinements have been published. As one of the most important, we single out the extended k-tree algorithm by Minder and Sinclair [24] that provides solution to GBP when the lists have smaller sizes and the time-memory tradeoffs by Bernstein et al. [5, 6].

Our contribution. Wagner points out a few open problems of the GBP and of the k-tree algorithm. Two of these problems, namely, improving the efficiency of k-tree when k is not a power of two and memory reduction of the k-tree, are in fact the main research topics of our paper.

The k-tree algorithm discards part of the lists when k is not a power of two (note how the complexity of k-tree takes lower bound of $\lg k$). For instance, 7-tree works only with 4 lists, while the remaining 3 lists are not processed. Our improvement to the 3-tree is to work with the discarded list (we call it a *passive* list) by creating multicollisions from the list, i.e. set of values that coincide on certain l bits, where $l < n$. Then, we produce several solutions with the 3-tree from the other (active) lists, and for the same l bits. Finally, the remaining $n - l$ bits are absorbed by combining the multicollisions from the passive list, and the solutions from the active lists. The speed-up is sufficient to break the $\mathcal{O}(2^{\frac{n}{2}})$ complexity bound for the 3-list problem and to show that in applications this can matter. As an example, we show a security reduction of two authenticated encryption CAESAR [3] proposals, Deoxys [17] and KIASU [19], based on the latest results of Nandi [25]. He shows that a forgery attack for COPA based candidates can be reduced to the 3-list problem. We apply our improved 3-tree algorithm to this problem and reduce the security bound of the candidates by 2 bits. Note, our analysis does not apply to the updated version of Deoxys [20].

[3] Note, we use lg for $\log_2$.

Our second contribution are time-memory tradeoffs for the k-tree algorithm[4]. This research topic has been investigated by Bernstein et al. Their best tradeoffs are described with the curves $TM^{\lg k} = k \cdot N$ and $T^2 \cdot M^{\lg k - 2} = \frac{k^2}{4} \cdot N$, where M and T, are the memory and time complexity, respectively, and N is the size of the space of elements. To achieve a better tradeoff, we play around with the idea of producing multiple collisions in a memory constrained environment with the use of Hellman's tables[5]. It allows us to significantly reduce the memory complexity of the first level of the k-tree algorithm and to achieve better tradeoffs. As a result, we obtain the tradeoff $T^2M^{\lg k-1} = k \cdot N$. This translates to $T^2M = 4 \cdot N$ for $k = 4$, and $T^2M^2 = 8 \cdot N$ for $k = 8$ (cf. $TM^2 = 4 \cdot N$ and $TM^3 = 8 \cdot N$ curves of Bernstein et al.). As illustrated further in Fig. 5, for a given amount of memory, the new tradeoff always leads to a lower time complexity than the previous tradeoffs. The improvement of the tradeoff can be seen on the case of generalized birthday problem for the hash function SHA-160 and $k = 8$. Our new tradeoff requires around 2^{50} SHA-1 computations and 2^{30} memory on 8 cores (with the use of van Oorschot and Wiener's parallel collisions search [33]), while with the same memory, the old tradeoff needs around 2^{65} SHA-1 computations.

2 The Generalized Birthday Problem

Wagner introduced the generalized birthday problem (GBP) as multidimensional generalization of the birthday problem. GBP is also called a k-list problem, and is formalized as follows:

Problem 1 *Given k lists $L_1, \ldots, L_k$ of elements drawn uniformly and independently at random from $\{0,1\}^n$, find $x_1 \in L_1, \ldots, x_k \in L_k$ such that $x_1 \oplus x_2 \oplus \ldots \oplus x_k = 0$.*

Obviously, if $|L_1| \times |L_2| \times \ldots \times |L_k| \geq 2^n$, then with a high probability the solution to the problem exists. The real challenge, however, is to find it efficiently.

When all the lists have a minimal size, i.e. $|L_i| = 2^{\frac{n}{k}}$, efficient algorithms to the k-list problem are known only for the cases when $k = 2$, and $k \geq n$. The former is due to the collisions search algorithm, i.e. 2-list problem is equivalent to finding collisions thus it can be solved in $2^{n/2}$. The latter is due to the Bellare and Micciancio result [2] which states that such problem can be solved by Gaussian elimination in $O(n^3 + kn)$. A trivial algorithm is known for the k-list when $2 < k < n$. The algorithm first creates two larger lists $\overline{L_1}, \overline{L_2}$, where $\overline{L_1} = \{X | X = x_1 \oplus \ldots \oplus x_{k/2}, x_i \in L_i\}$, $\overline{L_2} = \{Y | Y = x_{k/2+1} \oplus \ldots \oplus x_k, x_i \in L_i\}$ and subsequently it finds a collision between the two lists. The size of the lists is $2^{\frac{n}{2}}$ thus the time complexity of the algorithm is $\mathcal{O}(2^{\frac{n}{2}})$.

Wagner proposed the k-tree algorithm that solves GBP (k-list problem) faster under the assumption that the list sizes are larger. Further we describe the case

[4] Similar tradeoffs were found by Biryukov and Khrovatovich, see [7].

[5] Joux and Lucks [23] use this technique to generate multiple collisions, which later lead to multicollisions.

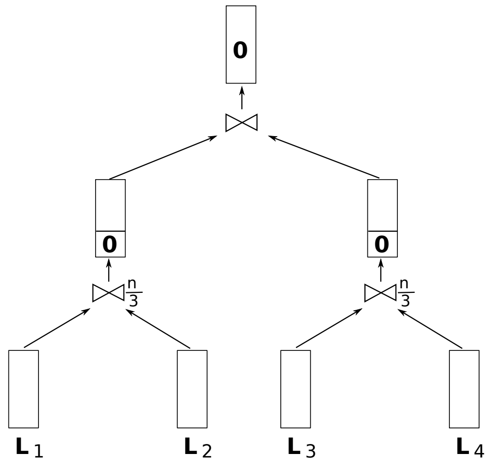

Fig. 1. Wagner's 4-tree algorithm.

when $k = 4$, refer to Fig. 1. Let us define $S \bowtie T$ as a list of elements common to both S and T, and let $low_l(x)$ stand for the l least significant bits of x. Furthermore, let us define $S \bowtie_l T$ as a set that contains all pairs from $S \times T$ that agree in their l least significant bits (the xor on the least significant bits is zero). Assume L_1, L_2, L_3, L_4 are four lists, each containing 2^l elements (l will be defined further). First we create a list L_{12} of values $x_1 \oplus x_2$, where $x_1 \in L_1, x_2 \in L_2$, such that $low_l(x_1 \oplus x_2) = 0$. Similarly, we create a list L_{34} of values $x_3 \oplus x_4$, where $x_3 \in L_3, x_4 \in L_4$, such that $low_l(x_3 \oplus x_4) = 0$. Finally, we search for a collisions between L_{12} and L_{34}. It is easy to see that such a collision reveals the required solution, i.e. $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = 0$.

The main advantage of the k-tree algorithm lies in the way the solution is found – at each of the two levels, only a simple collision search algorithm is used, and only a certain number of bits is made to fulfill the final goal (the xor is zero on all bits). At the first level, the lists L_{12}, L_{34} contain words that have zeros on the l least significant bits, thus xor of any two words from the lists must have zeros on these bits. At the second level, the xor of the words from the two lists will result in zeros on the remaining $n - l$ bits, if there are enough pairs. To get the sufficient number of pairs, the value of l is defined as $l = n/3$. Then each of L_{12}, L_{34} will have $2^{n/3} \cdot 2^{n/3}/2^{n/3} = 2^{n/3}$ words, and thus at the second level there will be $2^{n/3} \cdot 2^{n/3} = 2^{2n/3}$ possible xors, one of which will have zeros on the remaining $n - n/3 = 2n/3$ bits. It is important to note that l is chosen as to balance the complexity of the two levels. Obviously, the total memory and the time complexities of the 4-tree algorithm are $\mathcal{O}(2^{n/3})$ each.

The very same idea is used to tackle any k-list problem, where k is a power of two. The only difference is in the choice of l, and in the number of levels. In general, the number of levels equals $\lg k$, and at each level except the final, additional l bits are set to zero. At the final level, the remaining $2l$ bits are zeroed. Hence, $l \cdot \lg k + l = n$, and thus $l = n/(\lg k + 1)$. The algorithm works in $\mathcal{O}(k2^{\frac{n}{\lg k+1}})$ time and memory and requires lists of sizes $2^{\frac{n}{\lg k+1}}$. As an example, let us focus on 8-list problem, i.e. we have $L_1, \ldots, L_8$ lists, $\lg 8 = 3$ levels, and $l = n/4$. At the first level we build $L_{12}, L_{34}, L_{56}, L_{78}$, by combining two lists L_i, L_j, each with $2^l = 2^{n/4}$ elements that have zeros in the $n/4$ least significant bits. At the second, we build L_{1234} and L_{5678} that have again $2^{n/4}$ elements with zeros in the next $n/4$ bits. Finally, at the third level, we find the solution that will have zeros on the remaining $n/2$ bits.

Wagner's algorithm works similarly when k is not a power of two. The trick is to make some lists *passive*, i.e. to choose one element from each of the passive lists, and then continue with the algorithm as for the case of power of two and the remaining lists. For instance, to solve 6-list problem for lists $L_1, \ldots, L_6$, we take any element $v_5 \in L_5$ and $v_6 \in L_6$, and then solve the 4-list problem $x_1 \oplus x_2 \oplus x_3 \oplus x_4 = v_5 \oplus v_6$, for the lists $L_1, \ldots, L_4$. We can easily remove the non-zero condition $v_5 \oplus v_6$ in the right part, by adding this value to each element of the list L_1. Hence, the complexity of the k-list problem for the case when k is not a power of two equals the complexity to the closest (and smaller) power of two case. Thus, for any value of k, the k-tree algorithm works in $\mathcal{O}(k \cdot 2^{\frac{n}{\lfloor \lg k \rfloor + 1}})$ time and memory.

3 Improved Algorithm for the 3-list Problem

We focus on the 3-list problem and show how to improve the complexity of Wagner's 3-tree algorithm. Our improvement is based on the idea of multicollisions. The technique mimics the approach developed by Nikolić et al. [27] and further generalized by Dinur et al. [13]. We exploit the 3-tree algorithm, but we also work with the passive list and make it more active. Namely, instead of simply taking one element from the passive list, we derive from it partial multicollisions – a set of words that share the same value on particular bits. We then force the active lists on these bits to have a specific value (which is xor of all the values of the partial multicollisions), and at the final step, merge the results of the active and passive lists to obtain zero on the remaining bits. Let us take a closer look at this idea.

Definition 1 *The set of n-bit words $S = \{x_1, \ldots, x_p\}$ forms a p-partial multicollision on the s least significant bits, if $low_s(x_1) = low_s(x_2) = \ldots = low_s(x_p)$.*

This is to say that all p words are equal on the last s bits. Note, the choice to work with the least bits is not crucial but is introduced to simplify the presentation. Given an arbitrary set, we can create a p-partial multicollision from this set, i.e. we can find a subset that is p-partial multicollision. The maximal value of p depends on the size of the initial set and will be analyzed later in the section.

Let us assume that we are given a 3-list problem with lists L_1, L_2, L_3, each of size 2^l. If we apply the k-tree algorithm, then l should equal $n/2$, the lists L_1, L_2 will be active, while L_3 will be passive. Instead of marking L_3 as passive, let us

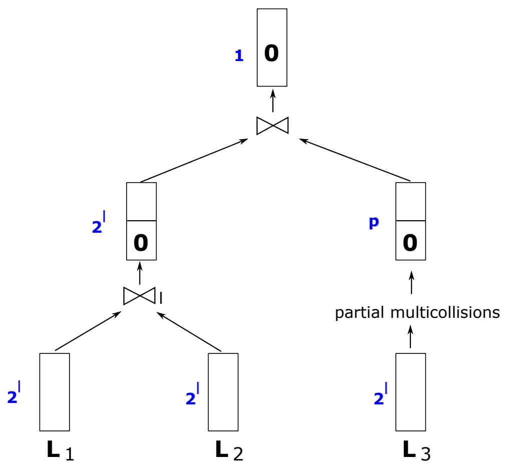

Fig. 2. Multicollision technique for k=3. The values in blue denote the size of the lists.

create a p-partial multicollision from L_3 on the l least significant bits (LSB) and denote this set as $\overline{L_3}$ (refer to Fig. 2). Without loss of generality we can assume that the colliding value of the l bits is zero (if not, we xor this value to all the elements of the list L_1). Furthermore, with the use of the join operator, from L_1, L_2 we create a list L_{12} of all values $x_1 \oplus x_2$, where $x_1 \in L_1, x_2 \in L_2$ and $low_l(x_1 \oplus x_2) = 0$; obviously $|L_{12}| \simeq 2^l$ with high probability. Finally, we use the join operator once again between L_{12} and $\overline{L_3}$, to find the required solution. As we have to cancel additional $n - l$ bits, the solution will exist with a high probability as long as $p|L_{12}| \geq 2^{n-l}$, that is, $p2^{2l-n} \geq 1$.

The complexity of our algorithm depends on the complexity of the two join operators and of producing multicollisions. The join operators (which are indeed simple collision searching algorithms) work in $\mathcal{O}(2^l)$ as in each of the cases, the sizes of the lists are not larger than 2^l. Furthermore, the partial multicollisions from $|L_3| = 2^l$ can be produced in $\mathcal{O}(2^l)$ time and memory[6]. Hence the multicollision technique solves the 3-list problem in $\mathcal{O}(2^l)$ time and memory.
Let us find the value of l. For this purpose we replace the inequality $p2^{2l-n} \geq 1$, with

$$p2^{2l-n} = 1, \tag{1}$$

and obtain

$$l = \frac{n}{2} - \frac{1}{2} \lg p. \tag{2}$$

[6] It is to initialize counters for each possible value of the colliding bits, then for each $x \in L_3$ increase the counter $low_l(x_3)$. After all elements have been processed, counter with the highest value corresponds to the largest multicollision set.

Therefore, the complexity of our algorithm is $\mathcal{O}(2^{\frac{n}{2}}/\sqrt{p})$, hence the speed-up factor is $\sqrt{p}$. Recall that p is the size of the multicollision set produced from the passive list L_3 – the larger the size, the greater the speed-up. Note, in the original Wagner's 3-tree algorithm, one element is chosen at random from L_3 and therefore the multicollision set consists of a single element. That is, for the classical 3-tree, $p = 1$ and the complexity is $\mathcal{O}(2^{\frac{n}{2}})$.

Let us examine the maximal possible value of p, i.e. the size of the p-partial multicollisions set on l bits produced from the set L_3 of size 2^l. Theorem 5 of [32] defines the number of elements in a set required to produce multicollision with a high probability, and by this theorem we obtain

$$(p!)^{1/p} 2^{\frac{p-1}{p} l} = 2^l. \tag{3}$$

A more straightforward way that we use to find the value of p is based on the so-called balls-into-bins problem: m balls are thrown into m bins (the bin for each ball is chosen uniformly at random), and the problem is to find the expected maximum load, i.e. the expected number of balls in a bin that contains the most balls. The solution to this problem is well known and the expected maximum load asymptotically is:

$$\Theta\left(\frac{\ln m}{\ln \ln m}\right). \tag{4}$$

Our multicollision problem is an instance of the balls-into-bins problem as the number of elements in the passive list L_3 (the number of balls), and the size of the multicollision space (the number of bins) are both 2^l. Therefore, the asymptotics of $p(l)$ can be evaluated as $\Theta(\frac{\ln 2^l}{\ln \ln 2^l}) = \Theta(\frac{l}{\ln l})$. Finally, as $l \approx \frac{n}{2}$, we obtain that the speed-up factor $\sqrt{p}$ of our improved 3-tree (over Wagner's 3-tree) is $\sqrt{\frac{n/2}{\ln n/2}}$, thus the complexity of our algorithm is

$$\mathcal{O}\left(2^{\frac{n}{2}}/\sqrt{\frac{n/2}{\ln n/2}}\right). \tag{5}$$

To find the actual speed-up for concrete values of n, we need to approximate the asymptotics of $p(l)$, i.e. need to find the approximate value of c in $p(l) = c\frac{l}{\ln l}$. For this purpose, we have run a series of experiments. For each value of $l = 10, \ldots, 28$, we have generated 2^l random values (of bit length l) and have checked the maximal number of multicollisions. For each l, the experiments have been repeated 20 times. The outcomes of the experiments are reported in Tbl. 1. Based on the experiments, the value of c can be approximated as $c \approx 1.3$. With such an assumption, we have computed the speed-up factor of our improved 3-tree for various values of n – we refer the reader to Tbl. 2.

The above strategy is in line with the multicollision approach by Nikolić et al. used in the analysis of the lightweight block cipher LED [15]. The advanced approach by Dinur et al., however, cannot be used for further improvements. One of their main ideas is to work simultaneously with a few multicollisions,

Table 1. Experimental search of number of multicollisions.

l	Average size	$\frac{l}{\ln l}$	c
10	5.80	4.34	1.34
11	5.85	4.59	1.27
12	6.10	4.83	1.26
13	6.45	5.07	1.27
14	7.00	5.30	1.32
15	7.15	5.54	1.29
16	7.55	5.77	1.31
17	7.90	6.00	1.32
18	8.15	6.23	1.31
19	8.50	6.45	1.32
20	8.70	6.68	1.30
21	9.05	6.89	1.31
22	9.50	7.12	1.33
23	9.65	7.34	1.31
24	10.30	7.55	1.36
25	10.40	7.77	1.34
26	10.60	7.98	1.33
27	11.05	8.19	1.35
28	11.15	8.40	1.33

Table 2. A comparison of the time complexities of Wagner's 3-tree with our new approach.

n	Speed-up ($\sqrt{p}$)	l
64	3.43	31
128	4.42	62
256	5.82	126
512	7.71	253

instead of only one. In the case of the k-tree algorithm, this would mean to produce from L_3 several p-partial multicollision sets. However, each such set will collide on s different value of the l LSBs, i.e. the elements of the first p-partial multicollision set will have the value v_1 on the l LSB, the elements of the second set will have the value v_2, etc. The different values will increase the complexity of the later stage of k-tree by a factor of s. When using the join operator on l bits of L_1 and L_2 there will be s targets (whereas previously we had only one), thus a simple collision search will have to be repeated s times. Therefore, in this particular case, Dinur et al. approach cannot be used.

Applications. The improvement of the 3-tree algorithm can be used for cryptanalysis of authenticated encryption schemes proposed to the ongoing CAESAR [3]. Some of these schemes, to process the final incomplete blocks of messages, use a construction called XLS proposed by Ristenpart and Rogaway [30]. Initially, XLS was proven to be secure, however Nandi [25] points out flaws in the security proof and shows a very simple attack that requires three queries to break the construction. However, the CAESAR candidates that rely on XLS, do not allow this trivial attack as the required decryption queries are not permitted by the schemes. To cope with this limitation, Nandi proposes another attack [26], that requires only encryption queries. He is able to reduce the design flaw of XLS to the 3-list problem. Therefore, Nandi is able to attack schemes that claim birthday bound *query* complexity because with only $2^{\frac{n}{3}}$ queries (equivalent to size of the lists in the 3-list problem), he can find a solution to the 3-list problem (in $2^{\frac{2n}{3}}$ time). However, Nandi cannot break the schemes that claim birthday bound *time* complexity, as he cannot solve the 3-list problem faster than $2^{\frac{n}{2}}$. Note, Nandi constructs the 3-list problem from only $2^{\frac{n}{3}}$ queries, rather than from $3 \cdot 2^{\frac{n}{3}}$, as the elements of all three lists depend on the same $2^{\frac{n}{3}}$ ciphertexts.

The CAESAR schemes based on XLS, such as Deoxys [17], Joltik [18], KIASU [19], Marble [14], SHELL [35], claim only birthday bound *time* complexity, thus Nandi's findings do not break the security claims of these candidates. However, our improved 3-tree algorithm goes below the birthday bound and thus can be used to show a slight weakness in some of these candidates.

Let us focus on the 128-bit CAESAR candidates Deoxys and KIASU. The 3-list problem for XLS in these candidates has the parameter $n = 128$. According to Tbl. 2, we can take $\sqrt{p} = 4.42$ and $l = 62$. Consequently, the complexity of a forgery based on the XLS weakness is $C \cdot 2^{62}$, where C is a constant factor. The value of C is 1 because: 1) as mentioned above, the 3 lists can be produced from the same 2^{62} ciphertexts, and 2) all of the operations required by the improved 3-tree algorithm are significantly less expensive than one encryption of the analyzed schemes. As a result, we obtain a forgery on the COPA modes of Deoxys and KIASU in 2^{62} encryptions and thus the security level of these schemes is reduced by 2 bits from the claimed 64 bits.

We note that our analysis does not apply to the updated versions of Deoxys [20] and Joltik [21], as these versions no longer use XLS.

4 Improved Time-Memory Tradeoffs

In applications, usually the elements of the lists L_i are in fact outputs of functions f_i, thus GBP is often formulated as:

Problem 2 *Given non-invertible functions* $f_1, \ldots, f_k : \{0,1\}^{n'} \to \{0,1\}^n$, $n' \geq n$, *find* $y_1, \ldots, y_k \in \{0,1\}^{n'}$ *such that* $f_1(y_1) \oplus f_2(y_2) \oplus \ldots \oplus f_k(y_k) = 0$.

In some applications, all the functions are identical, and the problem is to find distinct inputs:

Problem 3 *Given a non-invertible function* $f : \{0,1\}^{n'} \to \{0,1\}^n, n' \geq n$, *find distinct* $y_1, \ldots, y_k \in \{0,1\}^{n'}$ *such that* $f(y_1) \oplus f(y_2) \oplus \ldots \oplus f(y_k) = 0$.

Both of the above problems give rise to the possibility of time-memory tradeoffs, i.e., reducing the memory complexity of the k-tree algorithm at the expense of time. We will investigate time-memory tradeoffs for the GBP as defined in Problem 3. Recall that k-tree in its current form assumes that both time and memory are of equal magnitude, i.e. $T = M = O(k \cdot 2^{\frac{n}{\lg k + 1}})$.

Bernstein et al. [5, 6] investigate k-tree in memory restricted environments and propose a few tradeoffs. Their main approach is to solve the k-list problem on less than n bits. Assume $M = 2^m$, where $M < 2^{\frac{n}{\lg k + 1}}$. Then, a k-list problem on $\bar{n} = m(\lg k + 1)$ bits (instead of n bits) can easily be solved with the k-tree algorithm. The first tradeoff idea is to perform a precomputation (or prefiltration) such that all the entries in each list have the value of 0 in the $n - \bar{n}$ most significant bits.[7] For the remaining $\bar{n}$ least significant bits, they apply the k-tree algorithm and thus find a solution for all n bits. The time complexity is the sum of the cost for precomputation and for solving the k-tree algorithm, which is $k \cdot (2^{n-\bar{n}} \cdot 2^m + 2^m) \approx k \cdot 2^{n-m \lg k}$. The tradeoff is therefore defined as

$$T \cdot M^{\lg k} = k \cdot N. \tag{6}$$

Their second idea is similar but does not use precomputation. They apply the k-tree algorithm on $\bar{n} = m(\lg k + 1)$ bits until the value of the remaining $n - \bar{n}$ bits probabilistically becomes zero. Obviously, in total there will be $2^{n-\bar{n}}$ repetition of the k-tree, thus the time complexity becomes $T = k \cdot 2^m \cdot 2^{n-\bar{n}} = k \cdot 2^{n-m \lg k}$, which provides the same tradeoff as the previous one, i.e.,

$$T \cdot M^{\lg k} = k \cdot N. \tag{7}$$

The third idea also relies on reduction of n, but the technique is more advanced. Assume, $f_1 = f_2, f_3 = f_4, \ldots$, i.e. the functions are pairwise identical. The k-list problem is regarded as two separate $\frac{k}{2}$ problems, the first involving the functions $f_1, f_3, \ldots$, while the second $f_2, f_4, \ldots$. If the amount of available memory is 2^m, then it is possible to solve each of these $\frac{k}{2}$-list problems on up to $\bar{n} = m(\lg \frac{k}{2} + 1) = m \cdot \lg k$ bits. By elevating the two $\frac{k}{2}$-lists to k-list, the remaining $n - \bar{n}$ bits can be zeroed with the use of memoryless collision search algorithm. Therefore the time complexity is $T = \frac{k}{2} \cdot 2^{\frac{n-\bar{n}}{2}} \cdot 2^m = \frac{k}{2} \cdot 2^{\frac{n}{2} - m(\frac{\lg k}{2} - 1)}$ and their tradeoff curve is defined as

$$T \cdot M^{\frac{\lg k}{2} - 1} = \frac{k}{2} \cdot N^{\frac{1}{2}},$$

which is converted to

$$T^2 \cdot M^{\lg k - 2} = \frac{k^2}{4} \cdot N, \tag{8}$$

Because this method solves $\frac{k}{2}$-list problem, it is meaningful when $k > 4$. We note that when $M < 2^{\frac{n}{\lg k + 2}}$, then (8) provides better tradeoff while for $M > 2^{\frac{n}{\lg k + 2}}$, (6) is better.

The k-tree relies on producing multiple collisions. For instance, at the first level of 4-tree, $2^{\frac{n}{3}}$ colliding pairs on $\frac{n}{3}$ bits are produced. Producing these pairs is trivial when the amount of available memory is $2^{\frac{n}{3}}$. However, once the memory is reduced to $2^m, m < \frac{n}{3}$, the trivial collision search does not work.

The fact that k-tree requires multiple collisions, opens doors to the following technique based on Hellman's tables [16] [8].

[7] It is pointed out in [6] that $n - \bar{n}$ bits can have an arbitrary value as long as the sum of all lists is zero. The technique is called *clamping through precomputations*.

[8] Note, we could not exploit the more advanced Rivest's distinguished points and Oechslin's rainbow tables [28] to improve the analysis.

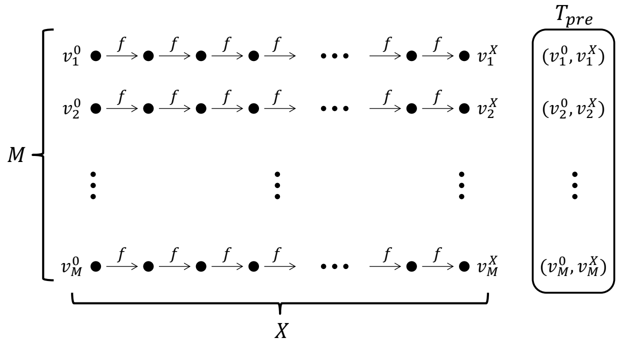

Fig. 3. Hellman's table T_{pre} when M memory is available.

Fact 1 (Hellman's table) *Let $f : \{0,1\}^* \rightarrow \{0,1\}^n$ be an arbitrary-size input and n-bit output function, $N = 2^n$, and let $M = 2^m$ be the amount of available memory. Once the precomputation equivalent to MX evaluations of f is performed, the cost of generating new collisions for f is $\frac{N}{MX}$ per collision.*

The technique works as follows. Choose M distinct values $v_i^0 \in \{0,1\}^n$, where $i = 1, 2, \ldots, M$. For each of them, compute chains of length X with the target function f, i.e. compute $v_i^j \leftarrow f(v_i^{j-1})$ for $i = 1, \ldots, M, j = 1, \ldots, X$, and store only the first and last values of each chain, i.e. (v_i^0, v_i^X), in a precomputation table T_{pre}. The construction of T_{pre} is depicted in Fig. 3. Note, even though MX values exist in all the chains, only $2M$ values are stored in T_{pre}. Once T_{pre} is constructed, to generate a collision, start with a random point and construct a chain of length $\frac{N}{MX}$. As there are N possible values, and MX are in T_{pre}, one point of the new chain will collide with one point of the chains created during the construction of the table. The match can be detected by further extending the new chain at most X times, as eventually it will reach one of v_i^X stored in T_{pre}. Then, the exact colliding values can be detected by recalculating chains from v_i^0 and the starting value of the new chain. Obviously T_{pre} can be reused to find not only one, but multiple collisions.

Joux and Lucks [23] use this technique to produce 3-collisions. They set $M = X = 2^{\frac{n}{3}}$ to generate $2^{\frac{n}{3}}$ ordinary collisions with time $T = 2^{\frac{2n}{3}}$ and memory $M = 2^{\frac{n}{3}}$. Then, they find another collision between $2^{\frac{n}{3}}$ ordinary collisions and $2^{\frac{2n}{3}}$ single values. When they generate $2^{\frac{n}{3}}$ ordinary multiple collisions, Hellman's table has an important role to keep the memory M rather than MX.

Further, we will use Hellman's table to produce multiple collisions for the first level of k-tree, but only on certain l bits (where $l < n$).

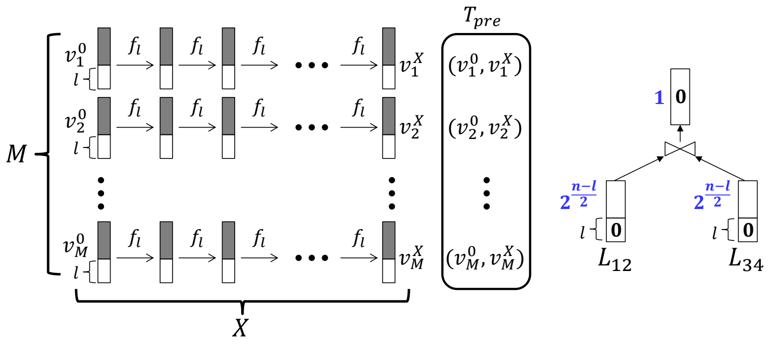

Fig. 4. Improved time-memory tradeoff for the 4-list problem with Hellman's table.

4.1 Improved Time-Memory Tradeoffs for the 4-list Problem

We present a more efficient time-memory tradeoff for GBP. Our tradeoff curve depends on the number of available lists, which is parameterized by k. For a better understanding, first we explain our algorithm for $k = 4$.

The original 4-tree algorithm consists of two-level collision searches (the parameter l used below will be determined later).

Level 1. Construct two lists, L_{12} and L_{34}, each containing $2^{\frac{n-l}{2}}$ partial collisions on l bits.

Level 2. Find a collision between the elements of L_{12} and L_{34} on the remaining $n - l$ bits.

Our new 4-tree algorithm works similarly with the exception of Level 1. At this level, we first construct Hellman's table, and then we use it to find $2^{\frac{n-l}{2}}$ collisions. As a result, our algorithm decomposes Level 1 into two parts. Its complexity depends on the available memory M which in turn determines the length of the chains X. The updated 4-tree is illustrated in Fig. 4 and is specified as follows.

Level 1a. Construct Hellman's table containing M chains, each of length of X.

Level 1b. With the use of Hellman's table, find $2 \cdot 2^{\frac{n-l}{2}}$ partial collisions on l bits. Store a half ($2^{\frac{n-l}{2}}$) of them in a list L_{12} and the other half in L_{34}.

Level 2. Find a collision between the elements of L_{12} and L_{34} on the remaining $n - l$ bits.

Construction of Hellman's table. For the Level 1a our algorithm first constructs Hellman's table which contains M chains of length X. However, unlike in [23], we have the following technical obstacle. The function f takes an n-bit input and produces an n-bit output and thus for such a function only full n-bit collisions can be identified. In other words, the classical Hellman's table cannot be used to find partial collisions.

To solve this problem, we define a reduction function $f_l : \{0,1\}^l \rightarrow \{0,1\}^l$ so that only the l bits are meaningful in the chain. For generating chains with f_l, $n-l$ bits of 0's are concatenated to the l-bit input, and this value is processed with $f : \{0,1\}^n \rightarrow \{0,1\}^n$. Finally, the n-bit output is truncated to l bits, and is used as the input to the next chain. That is, $f_l(x) = Trunc_l\big(f(0^{n-l}\|x)\big)$, where $Trunc_l(\cdot)$ truncates the n-bit input to the l least significant bits.

To summarize, we choose M distinct l-bit values v_i^0 for $i = 1, 2, \ldots, M$, for each of them generate a chain of length X by computing $v_i^{j+1} = f_l(v_i^j)$ where $j = 1, 2, \ldots, X$. In total, MX values are in all the chains and only the first and the last points of each chain are stored in T_{pre}. Thus Hellman's table requires around MX time and M memory.

Generation of l-bit collisions. According to Fact 1, once Hellman's table T_{pre} is constructed, the complexity for producing l-bit collisions is reduced significantly. Considering that the size of the values in the chains is l bits and the length of each chain is X, Fact 1 shows that the cost is $\frac{2^l}{MX}$ per collision.

To generate an l-bit collision, we choose a random l-bit value and with the function f_l from it compute a chain of length $\frac{2^l}{MX} + X$. On average, one collision will occur before we reach the $\frac{2^l}{MX}$ value of this new chain against the MX values stored in T_{pre}. The computation of additional X values in the chain ensures that the corresponding v_i^X will appear as one of the ending points of T_{pre}. The exact colliding pairs are detected by recomputing the chains from v_i^0 and the chosen l-bit value.

From the definition of f_l, the two inputs colliding on f always have the form $(0^{n-l}\|l_1, 0^{n-l}\|l_2)$, where 0^{n-l} is a sequence of $n-l$ zero bits and l_1 and l_2 are some l-bit values. A collision of the two chains means that $Trunc_l\big(f(0^{n-l}\|l_1)\big) = Trunc_l\big(f(0^{n-l}\|l_2)\big)$. Therefore, $f(0^{n-l}\|l_1)$ and $f(0^{n-l}\|l_2)$ only collide in the least significant l bits, while on the remaining $n-l$ bits behave randomly.

The collision generation process is iterated $2^{\frac{n-l}{2}}$ times and the input and output of each pair is stored in L_{12}. Similarly, the process is iterated additional $2^{\frac{n-l}{2}}$ times and the results are stored in L_{34}. Therefore the complexity of this step is around $2 \cdot 2^{\frac{n-l}{2}} \cdot \frac{2^l}{MX} = 2 \cdot \frac{N^{\frac{1}{2}} 2^{\frac{l}{2}}}{MX}$ time and $2 \cdot 2^{\frac{n-l}{2}} = 2 \cdot \frac{N^{\frac{1}{2}}}{2^{\frac{l}{2}}}$ memory.

Finding a solution to the 4-list problem. From the two lists L_{12} and L_{34} containing $2^{\frac{n-l}{2}}$ partial collisions on l bits, we find a collision on the remaining $n-l$ bits. This procedure is straightforward and it requires $2^{\frac{n-l}{2}} = \frac{N^{\frac{1}{2}}}{2^{\frac{l}{2}}}$ time and no additional memory.

Parameters and the tradeoff. The complexities for each step are as follows:

Level 1a.	Time $= MX$,	Memory $= M$
Level 1b.	Time $= 2 \cdot \dfrac{N^{\frac{1}{2}} 2^{\frac{l}{2}}}{MX}$,	Memory $= 2 \cdot \dfrac{N^{\frac{1}{2}}}{2^{\frac{l}{2}}}$
Level 2.	Time $= \dfrac{N^{\frac{1}{2}}}{2^{\frac{l}{2}}}$,	Memory = negligible

To balance the memory at Level 1a and Level 1b, M, N, l should satisfy the relation $M = 2 \cdot \frac{N^{\frac{1}{2}}}{2^{\frac{l}{2}}}$. From this relation, the time complexity of Level 2 becomes $\frac{M}{2}$, and thus is negligible compared to Level 1a when X is sufficiently large. To balance the time complexities of Level 1a and Level 1b, we need $MX = 2 \cdot \frac{N^{\frac{1}{2}} 2^{\frac{l}{2}}}{MX}$, which gives the relation $M^3 X^2 = 4 \cdot N$. Finally, as the time complexity T satisfies $T = MX$, we obtain the following tradeoff curve

$$T^2 \cdot M = 4 \cdot N. \tag{9}$$

For instance, when the available memory is $2^{\frac{n}{4}}$ (instead of $2^{\frac{n}{3}}$ as in the original 4-tree), then the updated 4-tree finds a solution in around $2^{\frac{3n}{8}}$ time. This is to be compared to Bernstein et al. tradeoffs given in (6) and (7) which require around $2^{\frac{n}{2}}$ time. Additional points of the tradeoff curve and comparison to previous results are given in Table 4.

During the analysis, we relied implicitly on several facts. First, we assumed that Hellman's table can contain an arbitrary number of points. In order to avoid collisions between the chains, however, the values of M and X cannot be arbitrary, but should depend on l. That is, during the construction of Hellman's table, the number of chains and their length is bounded by the value of l. Biryukov and Shamir in [8] call this a matrix stopping rule, and define it as $MX^2 \leq 2^l$. It is trivial to see that this inequality holds in our case as $MX^2 = M\frac{4N}{M^3} = \frac{4N}{M^2} = \frac{4N}{(2N^{\frac{1}{2}}/2^{\frac{l}{2}})^2} = 2^l$. For instance, when $M = 2^{\frac{n}{4}}$, then $l \approx \frac{n}{2}$, $T = 2^{\frac{3n}{8}}$, $X = 2^{\frac{n}{8}}$. Therefore, obviously $MX^2 = 2^{\frac{n}{2}} = 2^l$. We assumed as well that the tradeoff applies only to Problem 3. However, a close inspections of our algorithm reveals that it can be applied to the case of pairwise identical functions, i.e., $f_1 = f_2, f_3 = f_4$. That is, the area of application of the tradeoff is wider, and is similar to the area of the tradeoff given by Bernstein et al. in (8). To deal with the extended case, we have to create two Hellman's tables at the initial stage, one for each pair of functions. Thus the time and memory complexities will increase by a factor of two at Level 1a, and will stay the same at Levels 1b and 2.

4.2 Improved Time-Memory Tradeoff for the k-list Problem

In this section, we generalize the time-memory tradeoff for the k-tree algorithm, where $k = 2^d$. Overall, we replace the collision generation at Level 1 of the k-tree algorithm with a generation based on Hellman's table. Hereafter, we call the bits whose sum is fixed to zero *clamped bits*.

The ordinary k-tree algorithm initially starts from 2^d lists containing $M = 2^m$ elements. At Level 1, 2^{d-1} lists containing M elements are generated with m bits clamped. At Level i for $i = 2, 3, \ldots, d-1$, 2^{d-i} lists containing M elements are generated with im bits clamped. At the last Level d there are two lists containing M elements with $(d-1)m$ bits clamped. As no longer M collisions are required, but rather only one, the sum on up to $(d+1)m$ bits can be 0, by setting $(d+1)m = n$, and thus the k-tree algorithm will find the solution to the k-list problem. However, if the memory size is restricted, i.e. $m \ll \frac{n}{d+1}$, the k-tree algorithm can enforce the sum of only $(d+1)m$ bits to zero.

Our algorithm replaces Level 1 with Hellman's table collision generation and performs the same procedure as the k-tree algorithm from Level 2 to Level d. To find the required solution after Level d, however, at Level 1 we clamp more bits. Let the number of the clamped bits at Level 1 be l. After the first level we will have 2^{d-1} lists, each with $M = 2^m$ elements. Similarly, after Level i for $i = 2, 3, \ldots, d-1$, we will have 2^{d-i} lists containing M elements with $l+(i-1)m$ bits clamped. After the final Level d, we will have one element with $l + dm$ zero bits. Therefore, we set $l + dm = n$, i.e. $l = n - dm$, to get at least one solution on all n bits. In Table 3, we compare the number of clamped bits of the k-tree and our algorithm.

Table 3. A comparison of the number of clamping bits between the k-tree and our algorithm.

	#lists	#clamped bits	
		k-tree algorithm	Our algorithm
Level 1	2^{d-1}	m	l
Level i, $(i = 2, \ldots, d-1)$	2^{d-i}	im	$l + (i-1)m$
Level d	1	$(d+1)m$	$l + dm$

From the condition $l = n - dm$ and the parameters k and m, we can determine the reduction function f_l for Hellman's table. We create M chains of length X, and only store the first and last values of the chains in Hellman's table T_{pre}. Once T_{pre} is constructed, we can find an l-bit partial collision with a cost of $\frac{2^l}{MX}$ per a collision, which is equivalent to $\frac{N}{M^{d+1}X}$. At Level 1, we produce in total $(2^{d-1} \cdot M)$ l-bit collisions, and store them in 2^{d-1} lists each with M elements. The total cost for producing the partial collisions and thus the complexity of Level 1 is $2^{d-1} \cdot \frac{N}{M^d X}$.

Complexity evaluation and the tradeoff curve. The complexity to generate T_{pre} is MX time and M memory. As mentioned above, Level 1 requires $2^{d-1} \cdot \frac{N}{M^d X}$ time and $2^{d-1} \cdot M$ memory. The time and memory complexities of the remaining Levels 2 to d are all M, thus negligibly small compared to the generation of T_{pre}. We balance the time complexity of Hellman's table generation and of Level 1, which gives the relation $T = MX = 2^{d-1} \cdot \frac{N}{M^d X}$, and can further be reduced to $(MX)^2 = 2^{d-1} \cdot \frac{N}{M^{d-1}}$ and approximately results in a tradeoff curve

$$T^2 \cdot M^{\lg k - 1} = k \cdot N \tag{10}$$

Note, the tradeoff given in Section 4.1 can be obtained from the above tradeoff by setting $k = 4$. In Table 4, we compare the previous tradeoffs given in (6), (8) to our new tradeoff for $k = 4, 8$ and for two particular memory amounts. Obviously, the time complexity of our algorithm is significantly smaller for the same amount of available memory.

Table 4. Comparison of tradeoffs. For simplicity, the constant multiplication for N is ignored.

	Method	M	T	Other parameters
$k=4$	Bernstein et al. Eq.(6)	$2^{\frac{n}{4}}$	$2^{\frac{n}{2}}$	–
	$(T \cdot M^2 = N)$	$2^{\frac{n}{6}}$	$2^{\frac{2n}{3}}$	–
	Our	$2^{\frac{n}{4}}$	$2^{\frac{3n}{8}}$	$X = 2^{\frac{n}{8}}, l = \frac{n}{2}$
	$(T^2 \cdot M = N)$	$2^{\frac{n}{6}}$	$2^{\frac{5n}{12}}$	$X = 2^{\frac{n}{4}}, l = \frac{2n}{3}$
$k=8$	Bernstein et al. Eq.(6)	$2^{\frac{n}{5}}$	$2^{\frac{2n}{5}}$	–
	$(T \cdot M^3 = N)$	$2^{\frac{n}{6}}$	$2^{\frac{n}{2}}$	–
	Bernstein et al. Eq.(8)	$2^{\frac{n}{5}}$	$2^{\frac{2n}{5}}$	–
	$(T^2 \cdot M = N)$	$2^{\frac{n}{6}}$	$2^{\frac{5n}{12}}$	–
	Our	$2^{\frac{n}{5}}$	$2^{\frac{3n}{10}}$	$X = 2^{\frac{n}{10}}, l = \frac{2n}{5}$
	$(T^2 \cdot M^2 = N)$	$2^{\frac{n}{6}}$	$2^{\frac{n}{3}}$	$X = 2^{\frac{n}{6}}, l = \frac{n}{2}$

The tradeoff curves of these three methods are also depicted in Fig. 5. The vertical axis and horizontal axis represent the logarithm of the time complexity t and memory complexity m, respectively. Curves for $k = 8$ and $k = 16$ are drawn in Fig. 5 with red lines and black lines, respectively. For $k = 8$ with $m \geq \frac{n}{4}$, the ordinary k-tree algorithm with $t = \frac{n}{4}$ can be performed. Thus, the time-memory tradeoffs are meaningful only when the memory amount is limited to $m < \frac{n}{4}$, and Fig. 5 only describes the curves in this range. Similarly, for $k = 16$ only $m < \frac{n}{5}$ is shown in the figure.

The previous curve given in (6) achieves the same time complexity as the k-tree algorithm when sufficient memory is available, while the time complexity is about 2^n when the available amount of memory is very limited. The previous curve given in (8) cannot reach the time complexity of the k-tree algorithm even if sufficient memory is available, while the time complexity is at most $2^{\frac{n}{2}}$ for very limited amount of memory. It is easy to see that our tradeoff takes advantages

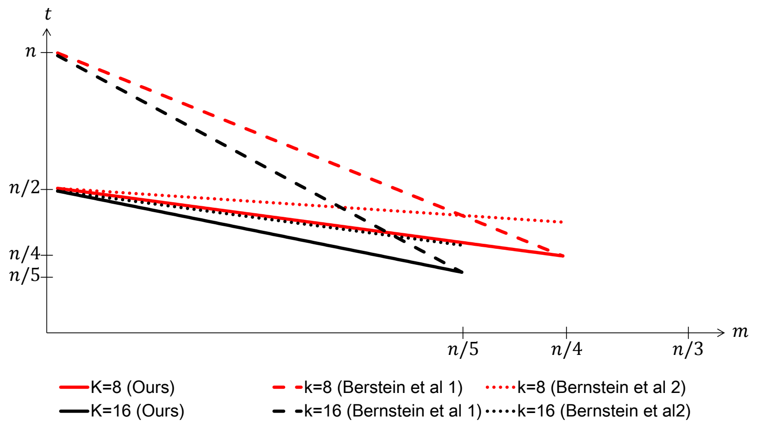

Fig. 5. Comparison of tradeoff curves. Our curve for $k = 8$ and Bernstein et al 2 for $k = 16$ are overlapped in the range of $m < n/5$.

of those two curves, i.e. it requires the same complexity as the k-tree algorithm when sufficient memory is available and requires only $2^{\frac{n}{2}}$ time complexity when the available amount of memory is limited. Therefore, our tradeoff always allows a lower time complexity than both of the previous tradeoffs. It improves the time complexity and simplifies the situation, as it is the best for any value of m (unlike the previous two tradeoffs that outperformed each other for different values of m).

5 Conclusion

We have shown improvements to Wagner's k-tree algorithm for the case when $k = 3$ and when the available memory is restricted. For the former case, our findings indicate that the passive list can be used to reduce the complexity of the 3-tree by a factor of $\sqrt{\frac{n/2}{\ln n/2}}$. Rather than discarding the passive list, we have produced multicollisions sets from it, and later, we have used the sets to decrease the size and thus the complexity of the 3-tree algorithm. In the case of a memory restricted k-list problem, we have provided a new time-memory tradeoff based on the idea of Hellman's table. The precomputed table has allowed us to efficiently produce a large number of collisions at the very first level of the k-tree algorithm, and thus to reduce the memory requirement of the whole algorithm. As a result, we have achieved an improved tradeoff that follows the curve $T^2 M^{\lg k - 1} = k \cdot N$.

We point out that we have run series of experiments to confirm parts of the analysis. In particular, we have verified that the predicted number of multicollisions and we have completely implemented the tradeoff for $k = 4, n = 60$ and various sizes of available memory, e.g, $m = 8, 10, 14$. The outcome of the experiments has confirmed the tradeoff.

The 3-list problem appears frequently in the literature and as our improved 3-tree algorithm is the first that solves this problem with below the birthday bound complexity, we expect future applications of the algorithm. However, although our improved 3-tree asymptotically outperforms Wagner's 3-tree algorithm, the speed up factor is lower for smaller values of n. Thus we urge careful analysis when applying the improved 3-tree.

Bernstein [5] argues that the large memory requirement of Wagner's k-tree algorithm makes it impractical. He assumes that the memory access is far more expansive, thus the actual cost of the algorithm is miscalculated. He introduces tradeoffs (discussed in Section 4) to reduce the memory requirement, and to obtain algorithms of lower complexity (measured by the new metric). We note that as our tradeoffs are more memory effective, by the new metric they lead to better algorithms for the k-tree problem with pairwise identical functions.

There are several future research directions. One is to consider restrictions on the amount of available data. The functions f_i in the k-list problem are often assumed to be public, i.e. the attacker can evaluate them offline. When f_i are not public, the data needs to be collected by making online queries. Thus developing new time-memory-data tradeoffs for this scenario is an interesting open problem. Another direction is to consider the weight of each function in the total cost of the algorithm, which leads to the case of an unbalanced GBP. This is based on the fact that in specific applications, it may occur that some of the functions are more costly to compute than other functions. The algorithm that solves an unbalanced GBP will be different than the one for the balanced GBP.

Acknowledgements. The authors would like to thank Pierre-Alain Fouque and Paul Kirchner for pointing out an error in the conference version of the paper and the anonymous reviewers of ASIACRYPT'15 for very helpful comments and suggestions (in particular, for pointing out the balls-into-bins problem). Ivica Nikolić is supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. M. Ajtai, R. Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In J. S. Vitter, P. G. Spirakis, and M. Yannakakis, editors, *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 601–610. ACM, 2001.
2. M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: Incrementality at reduced cost. In W. Fumy, editor, *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, volume 1233 of *Lecture Notes in Computer Science*, pages 163–192. Springer, 1997.
3. D. Bernstein. CAESAR Competition. `http://competitions.cr.yp.to/caesar.html`, 2013.
4. D. J. Bernstein. Enumerating solutions to p(a) + q(b) = r(c) + s(d). *Math. Comput.*, 70(233):389–394, 2001.

5. D. J. Bernstein. Better price-performance ratios for generalized birthday attacks. In *Workshop Record of SHARCS'07: Special-purpose Hardware for Attacking Cryptographic Systems*, 2007. `http://cr.yp.to/rumba20/genbday-20070719.pdf`.
6. D. J. Bernstein, T. Lange, R. Niederhagen, C. Peters, and P. Schwabe. Fsbday. In B. K. Roy and N. Sendrier, editors, *Progress in Cryptology - INDOCRYPT 2009, 10th International Conference on Cryptology in India, New Delhi, India, December 13-16, 2009. Proceedings*, volume 5922 of *Lecture Notes in Computer Science*, pages 18–38. Springer, 2009.
7. A. Biryukov and D. Khovratovich. Asymmetric proof-of-work based on the generalized birthday problem. *Proceedings of NDSS 2016*, page 13, 2016.
8. A. Biryukov and A. Shamir. Cryptanalytic time/memory/data tradeoffs for stream ciphers. In Okamoto [29], pages 1–13.
9. D. Bleichenbacher. On the generation of DSA one-time keys. In *The 6th Workshop on Elliptic Curve Cryptography (ECC 2002)*, 2002.
10. A. Blum, A. Kalai, and H. Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. In F. F. Yao and E. M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 435–440. ACM, 2000.
11. D. Boneh, A. Joux, and P. Q. Nguyen. Why textbook ElGamal and RSA encryption are insecure. In Okamoto [29], pages 30–43.
12. P. Chose, A. Joux, and M. Mitton. Fast correlation attacks: An algorithmic point of view. In L. R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 209–221. Springer, 2002.
13. I. Dinur, O. Dunkelman, N. Keller, and A. Shamir. Key recovery attacks on 3-round Even-Mansour, 8-step LED-128, and full AES2. In K. Sako and P. Sarkar, editors, *ASIACRYPT (1)*, volume 8269 of *Lecture Notes in Computer Science*, pages 337–356. Springer, 2013.
14. J. Guo. Marble v1. Submitted to CAESAR, 2014.
15. J. Guo, T. Peyrin, A. Poschmann, and M. J. B. Robshaw. The LED block cipher. In B. Preneel and T. Takagi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, volume 6917 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2011.
16. M. E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, 1980.
17. J. Jean, I. Nikolić, and T. Peyrin. Deoxys v1. Submitted to CAESAR, 2014.
18. J. Jean, I. Nikolić, and T. Peyrin. Joltik v1. Submitted to CAESAR, 2014.
19. J. Jean, I. Nikolić, and T. Peyrin. KIASU v1. Submitted to CAESAR, 2014.
20. J. Jean, I. Nikolić, and T. Peyrin. Deoxys v1. 3. *CAESAR Round*, 2, 2015.
21. J. Jean, I. Nikolić, and T. Peyrin. Joltik v1. 3. *CAESAR Round*, 2, 2015.
22. A. Joux and R. Lercier. "chinese & match", an alternative to Atkins "Match and Sort" method used in the sea algorithm. *Mathematics of computation*, 70(234):827–836, 2001.
23. A. Joux and S. Lucks. Improved generic algorithms for 3-collisions. In M. Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2009.

24. L. Minder and A. Sinclair. The extended k-tree algorithm. *J. Cryptology*, 25(2):349–382, 2012.
25. M. Nandi. XLS is not a strong pseudorandom permutation. In P. Sarkar and T. Iwata, editors, *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014. Proceedings, Part I*, volume 8873 of *Lecture Notes in Computer Science*, pages 478–490. Springer, 2014.
26. M. Nandi. Revisiting security claims of XLS and COPA. Cryptology ePrint Archive, Report 2015/444, 2015. `http://eprint.iacr.org/`.
27. I. Nikolić, L. Wang, and S. Wu. Cryptanalysis of round-reduced LED. In S. Moriai, editor, *FSE*, volume 8424 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2013.
28. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Advances in Cryptology-CRYPTO 2003*, pages 617–630. Springer, 2003.
29. T. Okamoto, editor. *Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Kyoto, Japan, December 3-7, 2000, Proceedings*, volume 1976 of *Lecture Notes in Computer Science*. Springer, 2000.
30. T. Ristenpart and P. Rogaway. How to enrich the message space of a cipher. In A. Biryukov, editor, *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2007.
31. R. Schroeppel and A. Shamir. A T=O(2ˆn/2), S=O(2ˆn/4) algorithm for certain NP-complete problems. *SIAM journal on Computing*, 10(3):456–464, 1981.
32. K. Suzuki, D. Tonien, K. Kurosawa, and K. Toyota. Birthday paradox for multi-collisions. In M. S. Rhee and B. Lee, editors, *ICISC*, volume 4296 of *Lecture Notes in Computer Science*, pages 29–40. Springer, 2006.
33. P. C. van Oorschot and M. J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptology*, 12(1):1–28, 1999.
34. D. Wagner. A generalized birthday problem. In M. Yung, editor, *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002.
35. L. Wang. SHELL v1. Submitted to CAESAR, 2014.