J Cryptol (2019) 32:1448–1490
https://doi.org/10.1007/s00145-018-9303-2

Efficient Dissection of Bicomposite Problems with Cryptanalytic Applications*

Itai Dinur
Computer Science Department, Ben-Gurion University, Beer-Sheva, Israel

Orr Dunkelman
Computer Science Department, University of Haifa, Haifa, Israel
orrd@cs.haifa.ac.il

Nathan Keller
Department of Mathematics, Bar-Ilan University, Ramat Gan, Israel

Adi Shamir
Computer Science Department, The Weizmann Institute, Rehovot, Israel

Communicated by Serge Vaudenay.

Received 26 November 2017 / Revised 9 July 2018
Online publication 21 August 2018

Abstract. In this paper, we show that a large class of diverse problems have a bicomposite structure which makes it possible to solve them with a new type of algorithm called *dissection*, which has much better time/memory tradeoffs than previously known algorithms. A typical example is the problem of finding the key of multiple encryption schemes with r independent n-bit keys. All the previous error-free attacks required time T and memory M satisfying $TM = 2^{rn}$, and even if "false negatives" are allowed, no attack could achieve $TM < 2^{3rn/4}$. Our new technique yields the first algorithm which never errs and finds all the possible keys with a smaller product of TM, such as $T = 2^{4n}$ time and $M = 2^n$ memory for breaking the sequential execution of r $= 7$ block ciphers. The improvement ratio we obtain increases in an unbounded way as r increases, and if we allow algorithms which can sometimes miss solutions, we can get even better tradeoffs by combining our dissection technique with parallel collision search. To demonstrate the generality of the new dissection technique, we show how to use it in a generic way in order to improve rebound attacks on hash functions and to solve with better time complexities (for small memory complexities) hard combinatorial search problems, such as the well-known knapsack problem.

Keywords. Bicomposite problems, Dissection algorithm, Time-memory tradeoff, Cryptanalysis, Multiple encryption, Knapsack problems.

*A shortened version of this paper [9] was presented at CRYPTO 2012.

1. Introduction

A composite problem is a problem that can be split into several simpler subproblems which can be solved independently of each other. To prevent attacks based on such decompositions, designers of cryptographic schemes usually try to entangle the various parts of the scheme by using a complex key schedule in block ciphers, or a strong message expansion in hash functions. While we can formally split such a structure into a top part that processes the input and a bottom part that produces the output, we cannot solve these subproblems independently of each other due to their strong interactions.

However, when we deal with higher level constructions which combine multiple primitives as black boxes, we often encounter unrelated keys or independently computed outputs which can provide exploitable decompositions. One of the best examples of such a situation was the surprising discovery by Joux [17] in 2004 that finding collisions in hash functions defined by the *parallel execution* of several independent hash functions is much easier than previously believed. In this paper, we show the dual result that finding the key of a multiple encryption scheme defined by the *sequential execution* of several independent cryptosystems is also easier than previously believed.

Since we can usually reduce the time complexity of cryptanalytic attacks by increasing their memory complexity, we will be interested in the full tradeoff curve between these two complexities rather than in a single point on it. We will be primarily interested in algorithms which use an exponential combination of $M = 2^{mn}$ memory and $T = 2^{tn}$ time for a small constant m and a larger constant t, when the key size n grows to infinity. While this setup may sound superficially similar to Hellman's time/memory tradeoff algorithms [14], it is important to notice that Hellman's preprocessing phase requires time which is equivalent to exhaustive search and memory which is at least the square root of the number of keys, and that in Hellman's online phase the product of time and memory is larger than the number of keys. In our model, we do not allow free preprocessing, we can use smaller amounts of memory, and the product of time and memory is strictly smaller than the number of keys.

The type of problems we can solve with our new techniques is characterized by the existence of two orthogonal ways in which we can decompose a given problem into (almost) independent parts. We call such problems *bicomposite*, and demonstrate this notion by considering the problem of cryptanalyzing the sequential execution of r block ciphers which use independent n-bit keys to process n-bit plaintexts, usually called *multiple encryption scheme* (see Fig. 1). In order to make the full rn-bit key of this scheme unique with a reasonable probability, the cryptanalyst needs r known plaintext/ciphertext pairs. The full encryption process can thus be described by an $(r + 1) \times r$ matrix whose columns correspond to the processing of the various plaintexts and whose rows

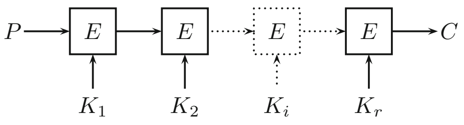

Fig. 1. Multiple encryption scheme with r independent keys.

correspond to the application of the various block ciphers, called in the sequel *execution matrix*. The attacker is given the r plaintexts at the top and the r ciphertexts at the bottom, and his goal is to find all the keys with a generic algorithm which does not assume the existence of any weaknesses in the underlying block ciphers. The reason we say that this problem is bicomposite is that the keys are independently chosen and the plaintexts are independently processed, and thus we can partition the execution matrix both horizontally and vertically into independent parts. In particular, if we know certain subsets of keys and certain subsets of intermediate values, we can independently verify their consistency with the given plaintexts or ciphertexts without knowing all the other values in the execution matrix. This should be contrasted with the standard constructions of iterated block ciphers, in which a partial guess of the key and a partial guess of some state bits in the middle of the encryption process usually cannot be independently verified by an efficient computation.

The security of multiple encryption schemes had been analyzed for more than 35 years, but most of the published papers had dealt with either double or triple encryption (which are widely used as DES-extensions, e.g., in the electronic payment industry). For example, Diffie and Hellman's original meet-in-the-middle attack [8], Lucks' improvement for Triple-DES [23], or Biham's work on triple modes of operation [6]. While the exact security of double and triple encryption are well understood and we can not push their analysis any further, our new techniques show that surprisingly efficient attacks can be applied already when we make the next step and consider quadruple encryption, and that additional improvements can be made when we consider even longer combinations.[1]

Standard meet-in-the-middle (MITM) attacks (introduced in [8]), which account for the best-known results against double and triple encryption, try to split such an execution matrix into a top part and a bottom part with a single horizontal partition line which crosses the whole matrix from left to right. Our new techniques use a more complicated way to split the matrix into independent parts by exploiting its two-dimensional structure. Consider, for example, the sequential execution of 7 independent block ciphers. We can find the full $7n$-bit key in just 2^{4n} expected time and 2^n expected memory by guessing two of the seven internal states after the application of the third block cipher and one of the seven internal states after the application of the fifth block cipher. We call such an irregular way to partition the execution matrix with partial guesses a *dissection*, since it mimics the way a surgeon operates on a patient by using multiple cuts of various lengths at various locations.

Our new techniques make almost no assumptions about the internal structure of the primitive operations, and in particular, they can be extended with just a slight loss of efficiency to primitive operations which are one-way functions rather than easily invertible permutations. This makes it possible to find improved attacks on message authentication codes (MACs) which are defined by the sequential execution of several keyed hash functions. Note that standard MITM attacks cannot be applied in this case, since we have to encrypt the inputs and decrypt the outputs in order to compare the results in the middle of the computation.

[1]These results are independent of the theoretical analysis proposed in [13] which discusses only query complexity (i.e., data complexity), disregarding any time or memory considerations.

To demonstrate the generality of our techniques, we show in this paper how to apply them to several types of combinatorial search problems. A main example is the knapsack problem: Given n generators $a_1, a_2, \ldots, a_n$ which are n-bit numbers, find a subset that sums modulo 2^n to S.[2] For 30 years, the best-known special purpose algorithm for this problem was the 1981 Schroeppel–Shamir algorithm [28], with complexity of $T = O(2^{n/2})$ and $M = O(2^{n/4})$. At Eurocrypt 2011, Becker et al. [4] presented several improved special purpose algorithms for different ranges of (T, M) (see Sect. 5). Our generic dissection technique provides better time complexities *for small memory complexities*.

To show the connection between knapsack problems and multiple encryption, describe the solution of the given knapsack problem as a two-dimensional $r \times r$ execution matrix, in which we partition the generators into r groups of n/r generators, and partition each number into r blocks of n/r consecutive bits. Each row in the matrix is defined by adding the appropriate subset of generators from the next group to the accumulated sum computed in the previous row. We start with an initial value of zero, and our problem is to find some execution that leads to a desired value S after the last row. This representation is bicomposite since the choices made in the various rows of this matrix are completely independent, and the computations made in the various columns of this matrix are almost independent as the only way they interact with each other is via the addition carries which do not tend to propagate very far into the next block. This makes it possible to guess and operate on partial states, and thus, we can apply almost the same dissection technique we used for multiple encryption schemes. Note that unlike the case of multiple encryption in which the value of r was specified as part of the given problem, here we can choose any desired value of r independently of the given value of n in order to optimize the time complexity for any available amount of memory. In particular, by choosing $r = 7$, we can reduce the best-known time complexity for hard knapsacks when we use $M = 2^{n/7} = 2^{0.1428n}$ memory from $2^{(3/4-1/7)n} = 2^{0.6071n}$ in [4] to $2^{4n/7} = 2^{0.5714n}$ with our new algorithm.

Previous algorithms for the knapsack problem [4,28] crucially depend on two facts: (1) addition is an associative and commutative operation on numbers, and (2) sets can be partitioned into the union of two subsets in an exponential number of ways. Our algorithms make no such assumptions, and thus, they can be applied under a much broader set of circumstances. For example, consider a non-commutative variant of the knapsack problem in which the generators a_i are permutations over $\{1, 2, \ldots, k\}$, and we have to find a product of length ℓ of these generators which is equal to some given permutation S (a special case of this variant is the problem of finding the fastest way to solve a given state of Rubik's cube by a sequence of face rotations, which was analyzed extensively in the literature). To show that this problem is bicomposite, we have to represent it by an execution matrix with independent rows and columns. Consider an $\ell \times k$ matrix in which the ith row represents the action of the ith permutation in the product, and the jth column represents the current location of element j from the set. Our goal is to start from the identity permutation at the top, and end with the desired

[2] We note that in the standard formulation of the knapsack problem, one searches for a subset that sums to S, without the modular reduction. However, as explained in [4], the modular formulation is computationally equivalent.

permutation S at the bottom. We can reduce this matrix to size $r \times r$ for various values of r by bunching together several permutations in the product and several elements from the set. The independence of the rows in this matrix follows from the fact that we can freely choose the next generators to apply to the current state, and the independence of the columns follows from the fact that we can know the new location of each element j if we know its previous location and which permutation was applied to the state, even when we know nothing about the locations of the other elements in the previous state. This makes it possible to guess partial states at intermediate stages and thus to apply the same dissection algorithms as in the knapsack problem with the same improved complexities.

We note that generic ideas similar to the basic dissection attacks were used before, in the context of several specific bicomposite problems. These include the aforementioned algorithms of Schroeppel and Shamir [28] and of Becker et al. [4] which analyzed the knapsack problem, the algorithm of van Oorschot and Wiener [27] which attacked double and triple encryption, and the results of Isobe [16] and of Dinur et al. [12] in the specific case of the block cipher GOST.[3] A common feature of all these algorithms is that none of them could beat the tradeoff curve $TM = N^{3/4}$, where N is the total number of keys. The algorithms of [12,16,27,28] matched this curve only for a single point, and the recent algorithm of Becker et al. [4] managed to match it for a significant portion of the tradeoff curve. Our new dissection algorithms not only allow to beat this curve, but actually allow to obtain the relation $TM < N^{3/4}$ for any amount of memory in the range $M \leq N^{1/4}$.

Follow-up Work Since the conference version of this paper has appeared, the dissection technique was studied further and applied in numerous papers, in a wide variety of contexts. To mention a few:

On the theoretical side, Austrin et al. [1] transformed the complexity analysis of the dissection technique from the average-case complexity setting considered here to the worst-case complexity setting and Wang [29] generalized the application to knapsacks to the k-SUM problem.

On the practical side, Canteaut et al. [7] used dissection in their 'sieve in the middle' generic technique for MITM attacks, Lallemand and Naya-Plasencia [20] used it in their semi-practical attack on the full stream cipher Sprout, Baek et al. [2] used it in an attack on a new white-box implementation of the AES, Kirchner and Fouque [18] used it to obtain improved algorithms for lattice enumeration, and Bar-On et al. [3] used it in attacks on reduced-round AES.

In a follow-up work [11], the authors applied the dissection technique to Feistel networks, obtaining generic attacks that outperform the best-known specialized attacks on a number of block ciphers, including the AES candidate DEAL. In addition, in [10], the authors elaborated on how to apply dissection to solve Rubik's cube with the smallest possible number of face rotations.

Paper Organization The paper is organized as follows: In Sect. 3, we introduce the dissection technique and present our best error-free attacks on multiple encryption.

[3] The basic idea of guessing internal values and "attacking" from them appeared in several prior works, most notably in Merkle and Hellman attack on 2K-3DES [25] and in Biham's work on triple modes of operation [6].

In Sect. 4, we consider the model when "false negatives" are allowed and show that the dissection algorithms can be combined with the parallel collision algorithm of van Oorschot and Wiener [27] to get an improved time-memory tradeoff curve. In Sect. 5, we apply our techniques to other cryptographic problems, such as solving the hardest instances of knapsack problems, and improving rebound attacks on hash functions. Finally, Sect. 6 concludes the paper and describes some open problems.

2. Notations and Conventions

In this paper, when we consider multiple encryption (mostly in Sects. 3 and 4), we denote the basic block cipher by E and assume that it uses n-bit blocks and n-bit keys (we can easily deal with other sizes, but it makes the notation and the discussion cumbersome). We denote by E^i the encryption process with key k_i, and denote by $E^{[1\dots r]}$ the multiple encryption scheme which uses r independent keys to encrypt the plaintext P and produce the ciphertext C via $C = E_{k_r}(E_{k_{r-1}}(\cdots E_{k_2}(E_{k_1}(P))\cdots))$. The intermediate value produced by the encryption of P_j under $E^{[1\dots i]}$ is denoted by X^i_j, and the decryption process of $E^{[1\dots r]}$ is denoted by $D^{[1\dots r]}$ (which applies the keys in reverse order). To attack $E^{[1\dots r]}$, we are usually given r plaintext/ciphertext pairs, which are expected to make the key unique (at intermediate stages, we may be given fewer than $j-i+1$ plaintext/ciphertext pairs for $E^{[i\dots j]}$, and then we are expected to produce all the compatible keys). In all our exponential complexity estimates, we consider expected rather than maximal possible values (under standard randomness assumptions, they differ by no more than a logarithmic factor), and ignore multiplicative polynomial factors in n and r.

When we consider execution matrices for bicomposite problems (mostly in Sect. 5), we denote the matrix by S, and the j'th element in its i'th row (which corresponds to the intermediate state X^i_j in multiple encryption) by $S_{i,j}$. In addition, we denote the 'actions' that can be performed on a state at the i'th row by a_i. The execution matrix notations are demonstrated in Fig. 2, which also emphasizes the fact that in a bicomposite execution matrix, if we know certain subsets of the actions and certain subsets of the intermediate values, we can independently verify their consistency without knowing all the other values in the execution matrix.

3. Dissecting the Multiple Encryption Problem

In this section, we develop our basic dissection algorithms that allow to solve efficiently the problem of multiple encryption. Given r-encryption with r independent keys, r n-bit plaintext/ciphertext pairs and 2^{mn} memory cells, the algorithms find all possible values of the keys which comply with the plaintext/ciphertext pairs, or prove that there are no such keys. The algorithms are deterministic, in the sense that they do not use random bits and they always succeed since they implicitly scan all possible solutions.

This section is organized as follows. In Sect. 3.1, we briefly describe the classical meet-in-the-middle attack which serves as a basis to our algorithms. In Sect. 3.2, we

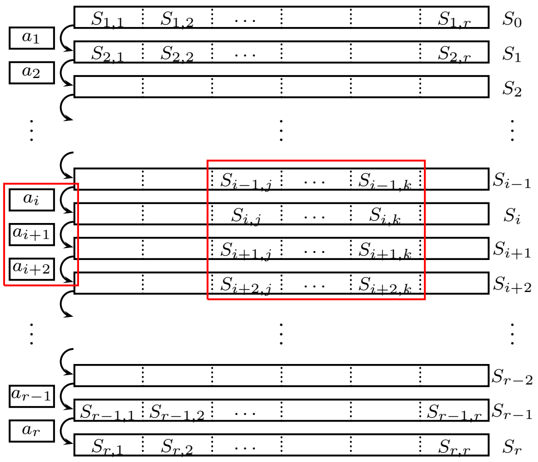

Fig. 2. An execution matrix of a bicomposite search problem.

present the most basic dissection algorithm, and apply it to 4-encryption. In Sect. 3.3, we discuss natural extensions of the basic dissection algorithm, which dissect the cipher in a symmetric way by splitting it into parts of equal size. In Sect. 3.4 we introduce *asymmetric* dissection algorithms (which split the cipher into parts of different sizes), and present a sequence of asymmetric dissection algorithms which are more efficient than the symmetric ones. In Sect. 3.5, we present a formal framework for dissection algorithms and show the optimality of our algorithms in this framework. While our basic algorithms and analysis apply only to the case where $M = 2^n$, in Sect. 3.6, we show that a small modification allows us to extend the dissection algorithms to any fixed amount of memory. We list the complexities of our most efficient deterministic dissection algorithms for all $r \leq 40$ and $m \leq 10$ in Table 1. Finally, in Sect. 3.7, we describe dissection algorithms in the case where instead of encryptions we are given a sequence of keyed one-way functions.

A reader which is interested mainly in the ideas of the dissection algorithms and not in details and generalizations, may concentrate on Sects. 3.2 and 3.4 and leave the other sections for later reading.

3.1. *Previous Work: The Meet-in-the-Middle Attack*

The trivial algorithm for recovering the key of an r-encryption scheme is exhaustive search over the 2^{rn} possible key values, whose time complexity is 2^{rn}, and whose memory requirement is negligible. In general, with no additional assumptions on the algorithm and on the subkeys, this is the best possible algorithm.

In [25] Merkle and Hellman observed that if the keys used in the encryption are independent, an adversary can trade time and memory complexities, using a MITM approach. In this attack, the adversary chooses a value u, $1 \leq u \leq \lfloor r/2 \rfloor$, and for each possible combination of the first u keys $(k_1, k_2, \ldots k_u)$ she computes the vector $(X_1^u, X_2^u, \ldots, X_r^u) = E^{[1\ldots u]}(P_1, P_2, \ldots, P_r)$ and stores it in a sorted table (along with the respective key candidate). Then, for each value of the last $r-u$ keys, the adversary computes the vector $D^{[u+1\dots r]}(C_1, C_2, \dots, C_r)$ and checks whether the value appears in the table (each such collision suggests a key candidate $(k_1, \dots, k_r)$). The right key is necessarily suggested by this approach, and in cases when other keys are suggested, additional plaintext/ciphertext pairs can be used to sieve the wrong key candidates.

The time complexity of this algorithm is $T = 2^{(r-u)n}$, whereas its memory complexity is $M = 2^{un}$. Hence, the algorithm allows to achieve the tradeoff curve $TM = 2^{rn}$ for any values T, M such that $M \leq 2^{\lfloor r/2 \rfloor n}$.[4] Note that the algorithm can be applied also if the number of available plaintext/ciphertext pairs is $r' < r$. In such a case, it outputs all the possible key candidates, whose expected number is $2^{(r-r')n}$ (since the plaintext/ciphertext pairs yield an $r'n$-bit condition on the 2^{rn} possible keys).

The MITM attack, designed for breaking double-encryption, is still the best-known generic attack on double-encryption schemes. It is also the best-known attack up to logarithmic factors[5] for triple encryption, which was studied very extensively due to its relevance to the former de-facto encryption standard Triple-DES.

3.2. *The Basic Dissection Algorithm: Attacking 4-Encryption*

In the following, we show that for $r \geq 4$, the basic MITM algorithm can be outperformed significantly, using a dissection technique. For the basic case $r = 4$, considered in this section, our algorithm runs in time $T = 2^{2n}$ with memory 2^n, thus allowing to reach $TM = 2^{3n}$, which is significantly better than the $TM = 2^{4n}$ curve suggested by the meet-in-the-middle attack.

The main idea behind the algorithm is to dissect the 4-encryption into two 2-encryption schemes and to apply the MITM attack to each of them separately. The partition is achieved by enumerating parts of the internal state at the dissection point. The basic algorithm, which we call $Dissect_2(4, 1)$ is given in Algorithm 1 and illustrated in Fig. 3. The notation $Dissect_2(4, 1)$ means "a dissection algorithm for 4-encryption, with $m = 1$ (i.e., with $2^{mn} = 2^n$ memory), and the division performed after round 2".

It is easy to see that once the right value for X_1^2 is considered, the right values of (k_1, k_2) are found in Step 2 and the right values of (k_3, k_4) are found in Step 6, and thus, the right value of the key is necessarily found. The time complexity of the algorithm is 2^{2n}. Indeed, Steps 2 and 6 are called 2^n times (for each value of X_1^2), and each of them runs the basic MITM attack on 2-encryption in expected time and memory of 2^n. Following the randomness of a block cipher (for a random block cipher, about 2^{-n} of the keys satisfy that the encryption of a given plaintext P is a given ciphertext C), the number

[4]We note that the algorithm, as described above, works only for $u \in \mathbb{N}$. However, it can be easily adapted to non-integer values of $u \leq \lfloor r/2 \rfloor$, preserving the tradeoff curve $TM = 2^{rn}$. This is done by trying 2^{un} key candidates for $(k_1, \dots, k_{\lceil u \rceil})$, storing the relevant partial encryptions, and testing the relevant $(k_{\lceil u \rceil+1}, \dots, k_r)$. If this procedure fails, the next 2^{un} key candidates for $(k_1, \dots, k_{\lceil u \rceil})$ are tested, and so forth.

[5]A logarithmic time complexity improvement can be achieved in these settings as suggested by Lucks [23]. The improvement relies on the variance in the number of keys encrypting a given plaintext to a given ciphertext. This logarithmic gain in time complexity comes at the expense of an exponential increase in the data complexity (a factor 8 gain in the time complexity when attacking triple-DES increases the data from 3 plaintext–ciphertext pairs to 2^{45} such pairs).

```
Input: Four plaintexts $(P_1, P_2, P_3, P_4)$ and their corresponding ciphertexts $(C_1, C_2, C_3, C_4)$
1: for all candidate values of $X_1^2 = E_{k_2}(E_{k_1}(P_1))$ do
2:   Run the standard MITM attack on 2-round encryption with $(P_1, X_1^2)$ as a single plaintext–ciphertext pair
3:   for all obtained $2^n$ values of $(k_1, k_2)$ of the previous step do
4:     Compute $X_2^2 = E_{k_2}(E_{k_1}(P_2))$ (i.e., partially encrypt $P_2$ under $k_1, k_2$)
5:     Store in a sorted table the corresponding values of $X_2^2$, along with the values of $(k_1, k_2)$
6:   Run the standard MITM attack on 2-round encryption with $(X_1^2, C_1)$ as a single plaintext–ciphertext pair
7:   for all obtained $2^n$ values of $(k_3, k_4)$ do
8:     Compute $X_2^2 = D_{k_3}(D_{k_4}(C_2))$ (i.e., partially decrypt $C_2$ using $(k_3, k_4)$)
9:     if the suggested value for $X_2^2$ appears in the table then
10:      Retrieve the corresponding $(k_1, k_2)$ from the table
11:      if $E_{k_4}(E_{k_3}(E_{k_2}(E_{k_1}(P_3, P_4)))) = (C_3, C_4)$ then
12:        return $(k_1, k_2, k_3, k_4)$
```

Algorithm 1: The $Dissect_2(4, 1)$ Algorithm

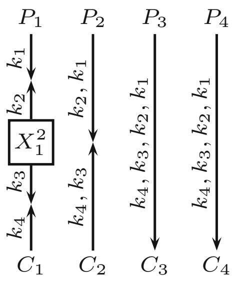

Fig. 3. Illustration of the $Dissect_2(4, 1)$ algorithm for 4-encryption.

of expected collisions in the table of X_2^2 is 2^n. Thus, the expected time complexity of the attack[6] is $2^n \cdot 2^n = 2^{2n}$.

The memory consumption of the 2-encryption MITM steps (Steps 2 and 6) is expected to be about 2^n. The size of the table computed in Step 5 is also 2^n, since each MITM step is expected to output 2^n key candidates. Hence, the expected memory complexity of the entire algorithm is 2^n.

[6]We remind the reader that we disregard factors which are polynomial in n and r.

3.3. *Natural Extensions of the Basic Dissection Algorithm*

We now consider the case $(r > 4, m = 1)$ and show that natural extensions of the $Dissect_2(4, 1)$ algorithm presented above, allow us to significantly increase the gain over the standard MITM attack for larger values of r.

It is clear that any algorithm for r'-encryption can be extended to attack r-encryption for any $r > r'$, by trying all possible $r - r'$ keys $(k_{r'+1}, \dots, k_r)$, and applying the basic algorithm to the remaining $E^{[1\dots r']}$. The time complexity is increased by a multiplicative factor of $2^{(r-r')n}$, and hence, the ratio $2^{rn}/TM$ is preserved. This leads to the following natural definition.

Definition 1. The gain of an algorithm A for r-encryption whose time and memory complexities are T and M, respectively, is defined as

$$Gain(A) = \log(2^{rn}/TM)/n = r - \log(TM)/n.$$

The maximal gain among all deterministic algorithms for r-encryption which use 2^{mn} memory, is denoted by $\text{Gain}_D(r, m)$ (where "D" stands for "deterministic").

By the trivial argument above, $\text{Gain}_D(r, 1)$ is monotone non-decreasing with r. The $Dissect_2(4, 1)$ algorithm shows that $\text{Gain}_D(r, 1) \geq 1$ for $r = 4$, and hence, for all $r \geq 4$. Below we suggest two natural extensions, which allow to increase the gain up to $\sqrt{r}$.

The LogLayer Algorithm: The first extension of the $Dissect_2(4, 1)$ is the recursive $LogLayer_r$ algorithm, applicable when r is a power of 2, which tries all the possible X_1^{2i} for $i = 1, 2, \dots, r/2 - 1$ and runs simple MITM attacks on each subcipher $E^{[2i+1,2i+2]}$ separately. As each such attack returns 2^n candidate keys (which can be stored in memory of $(r/2)\cdot 2^n$), the algorithm then groups 4 encryptions together, enumerates the values X_2^{4i} for $i = 1, 2, \dots, r/4 - 1$, and runs MITM attacks on each subcipher $E^{[4i+1\dots 4i+4]}$ separately (taking into account that there are only 2^n possibilities for the keys (k_{4i+1}, k_{4i+2}) and 2^n possibilities for the keys (k_{4i+3}, k_{4i+4})). The algorithm continues recursively (with $\log r$ layers in total), until a single key candidate is found. We illustrate $LogLayer$ for 8-encryption in Fig. 4.[7]

The memory complexity of $LogLayer_r$ is 2^n (as we need to store no more than r tables, each of size 2^n). As in the jth layer of the attack, $(r/2^j) - 1$ intermediate values are enumerated, and as each basic MITM attack has time complexity of 2^n, the overall time complexity of the attack is

[7] We note that one can slightly reduce the memory complexity of the attack by using only the pair (P_3, C_3) for checking the consistency between the candidates for the subkeys $(k_1, \dots, k_4)$ and the candidates for the subkeys $(k_5, \dots, k_8)$ and then using the remaining plaintext/ciphertext pairs sequentially to verify the correctness of the proposed full key $k_1, \dots, k_8$, as was done in the algorithm $Dissect_2(4, 1)$. As the improvement is only by a constant factor, for sake of simplicity we omit it and use all the remaining plaintext/ciphertext pairs in the consistency check between $(k_1, \dots, k_4)$ and the candidates for the subkeys $(k_5, \dots, k_8)$. We do the same in the following algorithms as well.

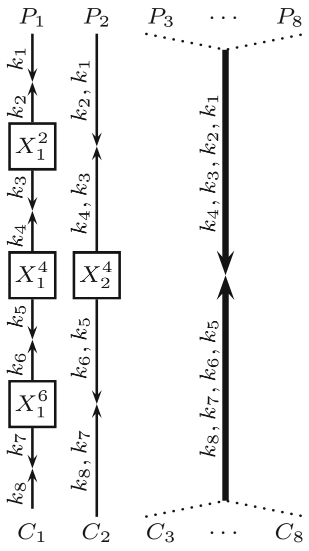

Fig. 4. Illustration of the *LogLayer* algorithm for 8-encryption.

$$\prod_{j=1}^{\log r} 2^{n((r/2^j)-1)} \cdot 2^n = 2^{n(r-\log r)}.$$

Therefore, $\text{Gain}(LogLayer_r) = \log r - 1$, which shows that $\text{Gain}_D(r, 1) \geq \lfloor \log r \rfloor - 1$.

The *Square*$_r$ Algorithm: This logarithmic gain of $LogLayer_r$ is significantly outperformed by the $Square_r$ algorithm, applicable when $r = (r')^2$ is a perfect square. The $Square_r$ algorithm starts by by enumerating all $X_1^{r'}, X_2^{r'}, \ldots, X_{r'-1}^{r'}, X_1^{2r'}, X_2^{2r'}, \ldots, X_{r'-1}^{2r'}, \ldots, X_1^{r'(r'-1)}, X_2^{r'(r'-1)}, \ldots, X_{r'-1}^{r'(r'-1)}$, i.e., $(r'-1)$ intermediate values every r rounds for $r'-1$ plaintexts. Given these values, the adversary can attack each of the r'-encryptions (e.g., $E^{[1\ldots r']}$), separately, and obtain 2^n "solutions" on average, which are stored in sorted tables. Then, the adversary can treat each r'-round encryption as a single encryption with 2^n possible keys, and apply an r'-encryption attack to recover the key. We illustrate $Square_9$ for 9-encryption in Fig. 5.

The time complexity of $Square_r$ is equivalent to repeating $2^{(r'-1)(r'-1)n}$ times a sequence of $r'+1$ attacks on r'-encryption. Hence, the time complexity is at most $2^{[(r'-1)(r'-1)+(r'-1)]\cdot n}$, and the memory complexity remains 2^n. Therefore, $\text{Gain}(Square_r) \geq \sqrt{r} - 1$, which shows that $\text{Gain}_D(r, 1) \geq \lfloor \sqrt{r} \rfloor - 1$.

Obviously, improving the time complexity of attacking r'-encryption with 2^n memory reduces the time complexity of $Square_r$ as well. However, as the best attacks of this kind known to us yield a gain of $O(\sqrt{r'}) = O(r^{1/4})$, the improvement to the overall gain of $Square_r$ is asymptotically negligible.

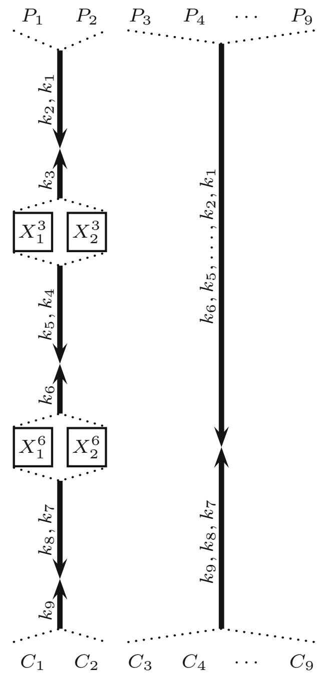
P_1 P_2 P_3 P_4 $\cdots$ P_9
k_2, k_1
k_3
X_1^3 X_2^3
k_5, k_4
k_6
X_1^6 X_2^6
k_8, k_7
k_9
$k_6, k_5, \ldots, k_2, k_1$
k_9, k_8, k_7
C_1 C_2 C_3 C_4 $\cdots$ C_9

Fig. 5. Illustration of the $Square_9$ algorithm for 9-encryption.

3.4. *Asymmetric Dissections: 7-Encryption and Beyond*

A common feature shared by the $LogLayer_r$ and the $Square_r$ algorithms is their symmetry. In both algorithms, every dissection partitions the composition into parts of the same size. In this section, we show that a better gain can be achieved by an asymmetric dissection.

We observe that the basic dissection attack on 4-encryption is asymmetric in its nature. Indeed, after the two separate MITM attacks are performed, the suggestions from the upper part are stored in a table, while the suggestions from the lower part are checked against the table values. As a result, the number of suggestions in the upper part is bounded from above by the size of the memory (which is now assumed to be 2^n and kept in sorted order), while the number of suggestions from the lower part can be arbitrarily large and generated on-the-fly in an arbitrary order. This suggests that an asymmetric dissection in which the lower part contains more rounds than the upper part may result in a better algorithm. This is indeed the case, as illustrated by the $Dissect_3(7, 1)$ algorithm given in Algorithm 2 and depicted in Fig. 6.

The memory complexity of the algorithm is 2^n, as both the basic MITM attack on triple encryption and the algorithm $Dissect_2(4, 1)$ require 2^n memory, and the size of the table computed in Step 5 is also 2^n.

The time complexity is 2^{4n}. Indeed, two n-bit intermediate encryption values are enumerated, both the basic MITM attack on triple encryption and the algorithm $Dissect_2(4, 1)$ require 2^{2n} time, and the remaining 2^{2n} possible values of (k_4, k_5, k_6, k_7)

```
Input: Seven plaintexts $(P_1, P_2, \ldots, P_7)$ and their corresponding ciphertexts $(C_1, C_2, \ldots, C_7)$
1: for all candidate values of $X_1^3, X_2^3$ do
2:   Apply the basic MITM algorithm to $E^{[1\ldots3]}$ with $(P_1, X_1^3)$ and $(P_2, X_2^3)$ as the plaintext–ciphertext pairs
3:   for all obtained $2^n$ values of $(k_1, k_2, k_3)$ of the previous step do
4:     Compute $X_3^3, X_4^3, \ldots, X_7^3 = E_{k_1,k_2,k_3}^{[1..3]}(P_3, P_4, \ldots, P_7)$
5:     Store in a sorted table the corresponding values of $X_3^3, X_4^3, \ldots, X_7^3$, along with the values of $(k_1, k_2, k_3)$
6:   Apply $Dissect_2(4, 1)$ to $E^{[4\ldots7]}$ with $(X_1^3, C_1)$ and $(X_2^3, C_2)$ as the plaintext–ciphertext pairs
7:   for all obtained $2^{2n}$ values of $(k_4, k_5, k_6, k_7)$ do
8:     Compute $X_3^3, X_4^3, \ldots, X_7^3 = D_{k_4,k_5,k_6,k_7}^{[4\ldots7]}(C_3, C_4, \ldots, C_7)$
9:     if the suggested value for $X_3^3, X_4^3, \ldots, X_7^3$ appears in the table then
10:      return $(k_1, k_2, \ldots, k_7)$
```

Algorithm 2: The $Dissect_3(7, 1)$ Algorithm

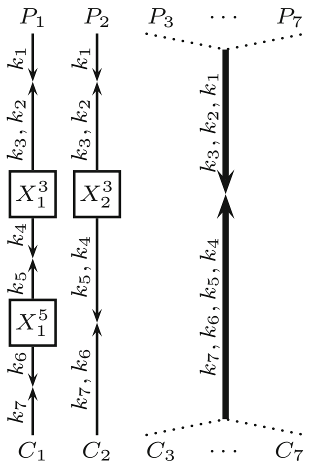

Fig. 6. Illustration of the $Dissect_3(7, 1)$ algorithm for 7-encryption.

are checked instantly. This leads to a time complexity of $2^{2n} \cdot 2^{2n} = 2^{4n}$. This shows that $\mathrm{Gain}(Dissect_3(7, 1)) = 2$, which is better than the algorithms $LogLayer_r$ and $Square_r$, for which the gain is only 1.

The algorithm $Dissect_3(7, 1)$ can be extended recursively to larger values of r, to yield a better asymptotical gain compared to the symmetric algorithms we presented. Given the algorithm $Dissect_j(r', 1)$ such that $\mathrm{Gain}(Dissect_j(r', 1)) = \ell - 1$, we define the algorithm $Dissect_{NEXT}^1 = Dissect_{\ell+1}(r' + \ell + 1, 1)$ for r-encryption, where $r = r' + \ell + 1$, using Algorithm 3 depicted in Fig. 7.

```
Input: $r$ plaintexts $(P_1, P_2, \ldots, P_r)$ and their corresponding ciphertexts $(C_1, C_2, \ldots, C_r)$
1: for all candidate values of $X_1^{\ell+1}, X_2^{\ell+1}, \ldots, X_\ell^{\ell+1}$ do
2:   Apply the basic MITM algorithm to $E^{[1\ldots(\ell+1)]}$ with $(P_1, X_1^{\ell+1})$, $(P_2, X_2^{\ell+1})$, ..., $(P_\ell, X_\ell^{\ell+1})$ as the plaintext–ciphertext pairs
3:   for all obtained $2^n$ values of $(k_1, k_2, \ldots, k_{\ell+1})$ of the previous step do
4:     Partially encrypt $P_{\ell+1}, P_{\ell+2}, \ldots, P_r$ using $(k_1, k_2, \ldots, k_{\ell+1})$
5:     Store in a sorted table the corresponding values of $X_{\ell+1}^{\ell+1}, X_{\ell+2}^{\ell+1}, \ldots, X_r^{\ell+1}$, along with the values of $(k_1, k_2, \ldots, k_{\ell+1})$
6:   Apply $Dissect_j(r', 1)$ to $E^{[(\ell+2)\ldots r]}$ with $(X_1^{\ell+1}, C_1)$, $(X_2^{\ell+1}, C_2)$, ..., $(X_\ell^{\ell+1}, C_\ell)$ as the plaintext–ciphertext pairs
7:   for all obtained $2^{(r'-\ell)n}$ values of $(k_{\ell+2}, k_{\ell+3}, \ldots, k_r)$ do
8:     Partially decrypt $C_{\ell+1}, C_{\ell+2}, \ldots, C_r$ using $(k_{\ell+2}, k_{\ell+3}, \ldots, k_r)$
9:     if the suggested value for $X_{\ell+1}^{\ell+1}, X_{\ell+2}^{\ell+1}, \ldots, X_r^{\ell+1}$ appears in the table then
10:      return $(k_1, k_2, \ldots, k_r)$
```

Algorithm 3: The $Dissect_{\ell+1}(r' + \ell + 1, 1)$ Algorithm

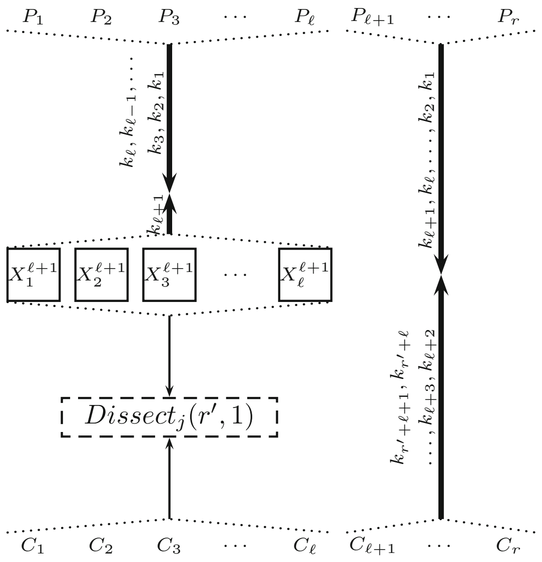

Fig. 7. Illustration of the $Dissect_{\ell+1}(r' + \ell + 1, 1)$ algorithm for $(r' + \ell + 1)$-encryption.

A similar argument to the one used for $Dissect_3(7, 1)$ shows that the time and memory complexities of $Dissect_{\ell+1}(r, 1)$ are $2^{r'n}$ and 2^n, respectively, which implies that $\text{Gain}(Dissect_{\ell+1}(r, 1)) = \ell$. In fact, $Dissect_3(7, 1)$ can be obtained from $Dissect_2(4, 1)$ by the recursive construction just described.

The recursion leads to a sequence of asymmetric dissection attacks with memory $M = 2^n$, such that the gain increases by 1 with each step of the sequence. Let r_ℓ be the smallest number of rounds at with a gain of ℓ is achieved, then by the construction, the sequence satisfies the recursion

$$r_\ell = r_{\ell-1} + \ell + 1,$$

which (together with $r_0 = 2$ which follows from the basic MITM attack) leads to the formula:

$$r_\ell = \frac{(\ell+1)(\ell+2)}{2} + 1.$$

The asymptotic gain of this sequence is obtained by representing ℓ as a function of r, and is equal to $(\sqrt{8r-7}-3)/2 \approx \sqrt{2r}$, which is bigger than the $\sqrt{r}$ gain of the $Square_r$ algorithm.

The analysis presented in Sect. 3.5 shows that the algorithms obtained by the recursive sequence described above are the optimal among all dissection algorithms that split the r rounds into two (not necessarily equal) parts and attacks each part recursively, using any dissection algorithm.

We conclude that as far as only dissection attacks are concerned, the *magic sequence* of the minimal numbers of rounds for which the gains are $\ell = 0, 1, 2, 3, \ldots$, called in the sequel *magic numbers*, is:

$$Magic_1 = \{2, 4, 7, 11, 16, 22, 29, 37, 46, 56, \ldots\}.$$

This "magic sequence" (also known as the Lazy Caterer's sequence) will appear several more times in the sequel.

3.5. *The $Dissect_u(r, 1)$ Algorithm*

In this section, we present a formal treatment of the dissection algorithm and show the optimality of the sequence $Magic_1$ presented above for algorithms which use the following framework: In the outer loop of the algorithm, the adversary dissects $E^{[1\ldots r]}$ into two parts, $E^{[1\ldots u]}$ and $E^{[u+1\ldots r]}$, and guesses a few of the X_i^u values. Then, she finds candidates for the keys $k_1, k_2, \ldots, k_u$ by attacking $E^{[1\ldots u]}$, and stores their values in a sorted table, along with some additional X_j^u values. At this point, the adversary attacks $E^{[u+1\ldots r]}$, deduces the candidate values for $k_{u+1}, k_{u+2}, \ldots, k_r$, computes the corresponding X_j^u values, and looks on-the-fly for matches in the table (each suggesting a value for the entire key of $E^{[1\ldots r]}$). Obviously, the attacks on $E^{[1\ldots u]}$ and $E^{[u+1\ldots r]}$ themselves can be performed using dissection algorithms.

As explained in Sect. 3.4, the number of X_i^u values the adversary has to guess is dictated by the number of key suggestions for $k_1, k_2, \ldots, k_u$ yielded by the attack on $E^{[1\ldots u]}$, since only these suggestions have to be stored in memory. As the amount of memory allowed in $Dissect_u(r, 1)$ is 2^n, we assume that the adversary guesses $u - 1$

```
Input: $r$ plaintexts $(P_1, P_2, \dots, P_r)$ and their corresponding ciphertexts $(C_1, C_2, \dots, C_r)$
1: for all candidate values of $X_1^u, X_2^u, \dots, X_{u-1}^u$ do
2:   Apply the basic MITM algorithm to $E^{[1\dots u]}$ with $(P_1, X_1^u), (P_2, X_2^u), \dots, (P_{u-1}, X_{u-1}^u)$ as the plaintext–ciphertext pairs
3:   for all obtained $2^n$ values of $(k_1, k_2, \dots, k_u)$ of the previous step do
4:     Partially encrypt $P_u, P_{u+1}, \dots, P_r$ using $(k_1, k_2, \dots, k_u)$
5:     Store in a sorted table the corresponding values of $X_u^u, X_{u+1}^u, \dots, X_r^u$, along with the values of $(k_1, k_2, \dots, k_u)$
6:   Attack $E^{[(u+1)\dots r]}$ (possibly using dissection) with $(X_1^u, C_1), (X_2^u, C_2), \dots, (X_{u-1}^u, C_u)$ as the plaintext–ciphertext pairs
7:   for all obtained $2^{[r-u-(u-1)]n}$ values[a] of $(k_{u+1}, k_{u+2}, \dots, k_r)$ do
8:     Partially decrypt $C_u, C_{u+1}, \dots, C_r$ using $(k_{u+1}, k_{u+2}, \dots, k_r)$
9:     if the suggested value for $X_u^u, X_{u+1}^u, \dots, X_r^u$ appears in the table then
10:      return $(k_1, k_2, \dots, k_r)$
```

[a] We recall that when attacking $E^{[u+1\dots n]}$, we expect that out of the $2^{(r-u)n}$ possible keys, only one in $2^{(u-1)n}$ is consistent with the given $u-1$ "plaintext"-ciphertext pairs.

Algorithm 4: The $Dissect_u(r, 1)$ Algorithm

values of the form X_i^u, as this makes the number of suggestions 2^n (under standard randomness assumptions).

Therefore, the $Dissect_u(r, 1)$ attack on r-encryption can be defined by Algorithm 4. Of course, the algorithms $Dissect_3(7, 1)$ and $Dissect_{\ell+1}(r' + \ell + 1, 1)$ presented in Sect. 3.4 are special cases of $Dissect_u(r, 1)$.

Complexity Analysis of $\boldsymbol{Dissect_u(r, 1)}$: It is easy to see that since the memory complexity of the attacks on $E^{[1\dots u]}$ and $E^{[u+1\dots r]}$ is at most 2^n, then the memory complexity of the whole attack is 2^n (recall that we expect 2^n candidates for $k_1, k_2, \dots, k_u$). Moreover, for the right guess of the $X_1^u, X_2^u, \dots X_{u-1}^u$ values, we are assured that the right key value is suggested by both Steps 2 and 6, and that the combination of the right key values leads to a match in the table in Step 9. Hence, the attack indeed returns the right key (perhaps with a few other candidates).

The time complexity of the attack is $2^{(u-1)n}$ times the complexity of Steps 2 and 6. The running time of Step 2 (attacking $u \le \lfloor r/2 \rfloor$ rounds[8]) does not affect the proof of optimality of our algorithms, and thus we ignore it.

Using our framework, we assume that the most efficient way to implement Step 6 is by calling $Dissect_{u^*}(r-u, 1)$ algorithm for some $u^* < r-u$. Thus, the running time of Step 6 is at least the time complexity of the $Dissect_{u^*}(r-u, 1)$ algorithm. In addition, one needs to note that the number of solutions suggested by this part of the attack is $2^{(r-2u+1)n}$, i.e., we expect another $2^{(r-2u+1)n}$ accesses to the table as part of Step 7).

[8] For a good block cipher E, the problem of attacking the encryption and the decryption direction is expected to be the same. Hence, we assume without lose of generality that indeed $u \le \lfloor r/2 \rfloor$.

For convenience of notation, let

$$f_1(r) \triangleq \min_{1 \le u \le r-1} \{r - 1 - Gain(Dissect_u(r, 1))\},$$

so that the lowest possible time complexity of an algorithm in our framework on r-encryption with 2^n memory is $2^{f_1(r)n}$.

Using this notation, the time complexity of $Dissect_u(r, 1)$ is at least

$$2^{(u-1)n} \cdot \max\left\{2^{f_1(u)n}, 2^{f_1(r-u)n}, 2^{(r-2u+1)n}\right\}.$$

Therefore, for a given value of r, the optimal value of u is the one that minimizes the above expression. In other words:

$$f_1(r) = \min_{1 \le u \le r} \{u - 1 + \max\{f_1(u), f_1(r-u), r - 2u + 1\}\}. \tag{1}$$

We can simplify this expression by plugging in $u = \lfloor r/2 \rfloor$, and obtaining $f_1(r) \le \lfloor r/2 \rfloor + 1 + f_1(\lceil r/2 \rceil)$. Thus, since $f_1(r)$ is non-decreasing, the minimum cannot be obtained for $u > \lfloor r/2 \rfloor$, which implies:

$$f_1(r) = \min_{1 \le u \le \lfloor r/2 \rfloor} \{u - 1 + \max\{f_1(r-u), r - 2u + 1\}\}. \tag{2}$$

Using Eq. 2 and the known values of $f_1(1) = f_1(2) = 1$ (which follow from the standard MITM algorithm on 2-encryption), it is easy to show by induction on r that the minimal complexities are achieved by the sequence of algorithms presented in Sect. 3.4.

Remark 1. One may be concerned by the fact that when the algorithm is run recursively there are no additional "plaintext" values that allow constructing the table in Step 2 (as we require at least one additional "plaintext" to filter some of the key candidates found in Step 6). This issue is solved by the fact that once the top part (being $E^{[1\ldots r^*]}$ for some r^*) has at most 2^n possible keys, we can partially encrypt as many plaintexts as we have to generate the required data.

3.6. *Deterministic Dissection Algorithms for $m > 1$*

While the algorithms presented above seem tailored to the case $m = 1$, it turns out that a small tweak in the recursive sequence of the dissection algorithms presented in Sect. 3.4 is sufficient to obtain optimal algorithms in our framework for any integer $m > 1$.

First, we define the general family of dissection algorithms $Dissect_u(r, m)$, which correspond to a memory complexity of 2^{mn}, in Algorithm 5.

As in the case $m = 1$, we denote $f_m(r) = r - m - \mathrm{Gain}_D(r, m)$, so that the lowest possible time complexity of a deterministic dissection attack on r-encryption with 2^{mn} memory is $2^{f_m(r)n}$. In these notations, exactly the same analysis as in the case $m = 1$ shows that

```
Input: $r$ plaintexts $(P_1, P_2, \ldots, P_r)$ and their corresponding ciphertexts $(C_1, C_2, \ldots, C_r)$
1: for all candidate values of $X_1^u, X_2^u, \ldots, X_{u-m}^u$ do
2:   Apply the basic MITM algorithm to $E^{[1\ldots u]}$ with $(P_1, X_1^u)$, $(P_2, X_2^u)$, ..., $(P_{u-m}, X_{u-m}^u)$ as the plaintext–ciphertext pairs
3:   for all obtained $2^{mn}$ values of $(k_1, k_2, \ldots, k_u)$ of the previous step do
4:     Partially encrypt $P_{u-m+1}, P_{u-m+2}, \ldots, P_r$ using $(k_1, k_2, \ldots, k_u)$
5:     Store in a sorted table the corresponding values of $X_u^u, X_{u+1}^u, \ldots, X_r^u$, along with the values of $(k_1, k_2, \ldots, k_u)$
6:   Attack $E^{[(u+1)\ldots r]}$ (possibly using dissection) with $(X_1^u, C_1)$, $(X_2^u, C_2)$, ..., $(X_{u-1}^u, C_u)$ as the plaintext–ciphertext pairs
7:   for all obtained $2^{[r-u-(u-1)]n}$ values of $(k_{u+1}, k_{u+2}, \ldots, k_r)$ do
8:     Partially decrypt $C_u, C_{u+1}, \ldots, C_r$ using $(k_{u+1}, k_{u+2}, \ldots, k_r)$
9:     if the suggested value for $X_u^u, X_{u+1}^u, \ldots, X_r^u$ appears in the table then
10:      return $(k_1, k_2, \ldots, k_r)$
```

Algorithm 5: The $Dissect_u(r, m)$ Algorithm

$$f_m(r) = \min_{1 \le u \le \lfloor r/2 \rfloor} \{u - m + \max\{f_m(r-u), r - 2u + m\}\}. \qquad (3)$$

The optimal choice of the "cut points" u for each value of r is obtained by a recursive sequence of algorithms, which is a simple generalization of the sequence that we obtained for $m = 1$.

First, note that if an algorithm $Dissect_j(r', m)$ satisfies $\mathrm{Gain}(Dissect_j(r', m)) = \ell - m$, then the algorithm $Dissect_{\ell+m}(r' + \ell + m, m)$ (which uses $Dissect_j(r', m)$ as a subroutine in the lower part) satisfies $\mathrm{Gain}(Dissect_{\ell+m}(r' + \ell + m, m)) = \ell$. This allows us to construct a recursive sequence of algorithms, in which the gain is increased by m at every step, thus generalizing the case of $m = 1$.

As the starting points for the sequence, we start with the m algorithms $Dissect_{m+i}(2m+2i, m)$ for $i = 0, 1, \ldots, m-1$. In the algorithm $Dissect_{m+i}(2m+2i, m)$, the adversary enumerates i intermediate values after $m + i$ rounds, applies simple MITM attacks on each part separately, and then applies a MITM attack between the 2^{mn} key suggestions from the two parts. The time and memory complexities of the algorithm are $2^{(m+i)n}$ and 2^{mn}, respectively, and hence, its gain is i.

Using these m starting points and the recursive step, the entire sequence can be computed easily. It turns out that the "magic sequence" of the numbers of rounds at which the gain $\ell = 0, 1, 2, \ldots$ obtained is

$$Magic_m = \{2m, 2m+2, 2m+4, \ldots, 4m, 4m+3, 4m+6, \ldots, \\ 7m, 7m+4, 7m+8, \ldots, 11m, \ldots\},$$

and the asymptotic gain is approximately $\sqrt{2mr}$.

Using Eq. (3), it is easy to show by induction on r that the optimal complexities among $Dissect_u(r, m)$ algorithms are achieved by the sequence of algorithms presented above.[9]

We present the time and memory complexities of the optimal $Dissect_u(r, m)$ algorithms for all $r \leq 40$ and $m \leq 10$ in Table 1.

3.7. *Dissection Algorithms for a Composition of Keyed One-Way Functions*

In this section, we consider compositions of keyed one-way functions (OWFs), which appear in the context of layered Message Authentication Codes, such as NMAC [5]. It turns out that the deterministic dissection algorithms presented above can be easily modified to yield efficient dissection algorithms for this scenario.

In the scenario of composition of OWFs, the goal is to retrieve $k_1, k_2, \ldots, k_r$ used in

$$F^{[1\ldots r]}(P) = F_{k_r}(F_{k_{r-1}}(\cdots F_{k_2}(F_{k_1}(P))\cdots)),$$

where $F_k(\cdot)$ is a keyed one-way function from an n-bit input and an n-bit key into an n-bit output. We shall assume that the keyed one-way function F is a family of 2^k random functions (where each function corresponds to a different key).

Given 2^{2n} memory, one can simply store a table of $(F_k(X), k, X)$ sorted by the values of $F_k(X)$ and k. Thus, given $F_k(X)$ and k, one can find all possible X values (there is one such value on average) that are "encrypted" into $F_k(X)$ under the key k. As this actually generates the "decryption" functionality by one memory access per each possible X (and thus amortized complexity of one memory access per each $(F_k(X), k)$ tuple), we can repeat the same $Dissect_u(r, m)$ algorithms designed for multiple encryption, if $m \geq 2$.

In case we only have 2^n memory, we use a slightly different approach in the dissection of $F^{[1\ldots r]}$ into two parts. In the lower part, instead of fixing intermediate values and checking which keys "decrypt" the known outputs[10] into these intermediate values, we go over all tuples of $(intermediate\ values, key)$ and store in a table those tuples which comply with the known outputs.[11] Then, we obtain candidates for the keys in the upper part and check them against the table. This procedure is somewhat less efficient than the original $Dissect_u(r, m)$, but obtains similar asymptotical results.

In the simplest case of $r = 3$, the algorithm, which we call $DissectOWF_2(3)$, is defined as in Algorithm 6.[12]

[9]Our algorithms only work for integer values of m, since $u - m$ (the number of intermediate values that we guess in Step 1 of Algorithm 5) has to be an integer. These algorithms can be extended to fractional values of m as well. However, as the extended algorithms are more cumbersome, we decided to not present them in this paper.

[10]As we consider one-way functions, we use the terms inputs and outputs rather than plaintexts and ciphertexts.

[11]Different from the case of multiple encryption, in the case of one-way functions, we build a table for the lower part of the cipher.

[12]Recall that we discuss only the case in which 2^n memory is given, as for 2^{2n} memory the previous *Dissect* algorithms are still applicable. Hence, we omit the amount of memory from the notation.

Table 1. $f_m(r)$ values.

r	$m=1$	$m=2$	$m=3$	$m=4$	$m=5$	$m=6$	$m=7$	$m=8$	$m=9$	$m=10$
1	1	1	1	1	1	1	1	1	1	1
2	**1**	1	1	1	1	1	1	1	1	1
3	2	2	2	2	2	2	2	2	2	2
4	**2**	**2**	2	2	2	2	2	2	2	2
5	3	3	3	3	3	3	3	3	3	3
6	4	**3**	**3**	3	3	3	3	3	3	3
7	**4**	4	4	4	4	4	4	4	4	4
8	5	**4**	**4**	**4**	4	4	4	4	4	4
9	6	5	5	5	5	5	5	5	5	5
10	7	6	**5**	**5**	**5**	5	5	5	5	5
11	**7**	**6**	6	6	6	6	6	6	6	6
12	8	7	**6**	**6**	**6**	**6**	6	6	6	6
13	9	8	7	7	7	7	7	7	7	7
14	10	**8**	8	**7**	**7**	**7**	**7**	7	7	7
15	11	9	**8**	8	8	8	8	8	8	8
16	**11**	10	9	**8**	**8**	**8**	**8**	**8**	8	8
17	12	11	10	9	9	9	9	9	9	9
18	13	**11**	**10**	10	**9**	**9**	**9**	**9**	**9**	9
19	14	12	11	**10**	10	10	10	10	10	10
20	15	13	12	11	**10**	**10**	**10**	**10**	**10**	**10**
21	16	14	**12**	12	11	11	11	11	11	11
22	**16**	**14**	13	**12**	12	**11**	**11**	**11**	**11**	**11**
23	17	15	14	13	**12**	12	12	12	12	12
24	18	16	15	14	13	**12**	**12**	**12**	**12**	**12**
25	19	17	**15**	**14**	14	13	13	13	13	13
26	20	18	16	15	**14**	14	**13**	**13**	**13**	**13**
27	21	**18**	17	16	15	**14**	14	14	14	14
28	22	19	18	**16**	16	15	**14**	**14**	**14**	**14**
29	**22**	20	**18**	17	**16**	16	15	15	15	15
30	23	21	19	18	17	**16**	16	**15**	**15**	**15**
31	24	22	20	19	18	17	**16**	16	16	16
32	25	**22**	21	**19**	**18**	18	17	**16**	**16**	**16**
33	26	23	**21**	20	19	**18**	18	17	17	17
34	27	24	22	21	20	19	**18**	18	**17**	**17**
35	28	25	23	22	**20**	20	19	**18**	18	18
36	29	26	24	**22**	21	**20**	20	19	**18**	**18**
37	**29**	27	25	23	22	21	**20**	20	19	19
38	30	**27**	**25**	24	23	22	21	**20**	20	**19**
39	31	28	26	25	**23**	**22**	22	21	**20**	20
40	32	29	27	**25**	24	23	**22**	22	21	**20**

Items marked in bold are magic numbers

Steps 1 and 2 go over 2^{2n} values, with an n-bit filtering condition (in Step 3). Hence, their running time is 2^{2n} and the expected memory consumption[13] is 2^n. Step 5 iterates over all 2^{2n} values of (k_1, k_2), and for each such key pair, we expect on average one

[13] A given output may have several inputs (even when the key is fixed), thus the number of "solutions" to the equation $F_{k_3}(X_1^2) = C_1$ may be larger than 2^n. However, assuming $F(\cdot)$ is a "good" one-way function, the number of solutions is not expected to be significantly larger than 2^n.

```
Input: Three inputs $(P_1, P_2, P_3)$ and their corresponding outputs $(C_1, C_2, C_3)$
1: for all candidate values of $X_1^2$ do
2:     for all $k_3$ do
3:         if $F_{k_3}(X_1^2) = C_1$ then
4:             Store $(X_1^2, k_3)$ in a sorted table
5: for all candidate values of $k_1, k_2$ do
6:     Compute $X_1^2 = F_{k_2}(F_{k_1}(P_1))$
7:     if $X_1^2$ appears in the table then
8:         Obtain $k_3$ from the table
9:         if $F^{[1\ldots3]}(P_2, P_3) = (C_2, C_3)$ then
10:            return $(k_1, k_2, k_3)$
```

Algorithm 6: The $DissectOWF_2(3)$ Algorithm

value of X_1^2 in the table. This value suggests on average a single value of k_3, from which we obtain a suggestion for the complete key. We conclude that the time complexity of $DissectOWF_2(3)$ is 2^{2n} and its memory complexity is 2^n.

We note that (similarly to the case of multiple encryption) a trivial extension of $DissectOWF_2(3)$ allows us to retrieve the key of a composition of r OWFs for any $r \geq 3$, in time $2^{(r-1)n}$ and memory 2^n. Moreover, if we are given only $r-1$ input/output pairs, the same algorithm allows us to retrieve the 2^n keys which comply with these pairs. This algorithm will be used in the recursive step below.

Starting with $DissectOWF_2(3)$, we can recursively construct a sequence of dissection algorithms. The construction is similar to the $Dissect_{NEXT}$ construction presented in Sect. 3.4, with a few differences which follow from the special structure of OWFs.

Given $\ell \geq 2$ and an algorithm $DissectOWF_j(r')$ whose gain is $\ell - 1$, i.e., $\text{Gain}(DissectOWF_j(r')) = \ell - 1$, we define the algorithm $DissectOWF_{NEXT} = DissectOWF_{r'}(r' + \ell + 1)$ for r-encryption, where $r = r' + \ell + 1$, by Algorithm 7.

An analysis similar to that of the $Dissect^1_{NEXT}$ algorithm presented in Sect. 3.4 shows that the time and memory complexities of $DissectOWF_{NEXT}$ are $T = 2^{r'n}$ and $M = 2^n$. Indeed, the only essential difference between $DissectOWF_{NEXT}$ and $Dissect^1_{NEXT}$ is the second part of Step 1 (i.e., computing the preimages), but the time complexity of this part is 2^{2n}, which is less than the complexity of the other steps of the algorithm, since $\ell \geq 2$.

This shows that the sequence of $DissectOWF_j(r)$ algorithms satisfies the same recursion relation as the sequence $Dissect_j(r, 1)$. Hence, if we denote by $\tilde{r}_\ell$ the ℓ'th element of the sequence (i.e., the number of rounds for which the gain is ℓ), then

$$\tilde{r}_\ell = \tilde{r}_{\ell-1} + \ell + 1.$$

```
Input: $r$ inputs $(P_1, P_2, \ldots, P_r$ and their corresponding outputs $(C_1, C_2, \ldots, C_r)$
1: for all candidate values of $X_1^{r'}, X_2^{r'}, \ldots, X_\ell^{r'}$ do
2:     for all $k_{r'+1}, k_{r'+2}, \ldots, k_r$ do
3:         if $F^{[r'+1\ldots r]}(X_1^{r'}, X_2^{r'}, \ldots, X_\ell^{r'}) = (C_1, C_2, \ldots, C_\ell)$ then
4:             Store $(X_1^{r'}, X_2^{r'}, \ldots, X_\ell^{r'}, k_{r'+1}, k_{r'+2}, \ldots, k_r)$ in a sorted table
5: Apply $DissectOWF_j(r')$ to $E^{[1\ldots r']}$ with $(P_1, X_1^{r'}), (P_2, X_2^{r'}), \ldots, (P_\ell, X_\ell^{r'})$ as the input–output pairs
6: for all obtained $2^{(r'-\ell)n}$ values of $(k_1, k_2, \ldots, k_{r'})$ of the previous step do
7:     Compute $X_1^{r'}, X_2^{r'}, \ldots, X_\ell^{r'} = F^{[1\ldots r']}(P_1, P_2, \ldots, P_\ell)$
8:     if $X_1^{r'}, X_2^{r'}, \ldots, X_\ell^{r'}$ appears in the table then
9:         Obtain $k_{r'+1}, k_{r'+2}, \ldots, k_r$ from the table
10:        if $F^{[1\ldots 3]}(P_{\ell+1}, P_{\ell+2}, \ldots, P_r) = (C_{\ell+1}, C_{\ell+2}, \ldots, C_r)$ then
11:            return $(k_1, k_2, \ldots, k_r)$
```

Algorithm 7: The $DissectOWF_{r'}(r)$ Algorithm (for $r = r' + \ell + 1$)

Since $\tilde{r}_0 = 3$, we get the formula

$$\tilde{r}_\ell = r_\ell + 1 = \frac{(\ell+1)(\ell+2)}{2} + 2.$$

Therefore, the "magic sequence" corresponding to a composition of OWFs is

$$Magic_1^{\text{OWF}} = \{3, 5, 8, 12, 17, 23, \ldots\},$$

and the asymptotic gain is approximately $\sqrt{2r}$, as in the basic r-encryption case.

4. Parallel Collision Search via Dissection

In Sect. 3, we only considered deterministic algorithms for r-encryption schemes which never err, that is, algorithms which find all the possible values of the keys which comply with the plaintext–ciphertext pairs, or prove that there are no such keys. For this type of algorithm, we improved the previously best-known generic tradeoff curve (obtained by MITM attacks) from $TM = 2^{rn}$ to $TM = 2^{(r-\sqrt{2r})n}$ using our dissection algorithms.

We now consider non-deterministic algorithms which find the right keys with some probability $p < 1$, which can be made arbitrarily close to one (i.e., Monte Carlo algorithms). In this case, an improved tradeoff curve of $T^2M = 2^{(3/2)rn}$ can be obtained by the *parallel collision search* algorithm of van Oorschot and Wiener [27]. In this section, we combine the dissection algorithms presented in Sect. 3 with the parallel collision search algorithm to obtain an even better tradeoff curve with a multiplicative advantage of at least $2^{(\sqrt{2r}/8)n}$ over the curve of [27].

This section is organized as follows: In Sect. 4.1, we give a brief description of the Parallel Collision Search algorithm. Our new algorithm, which we call "Dissect & Collide", is presented in Sect. 4.2. In Sect. 4.3, we present several extensions of the basic Dissect & Collide algorithm and analyze its gain compared to the PCS algorithm. Finally, we present a comparison between the performances of the dissection, PCS, and Dissect & Collide algorithms for selected values of r and m in Table 2.

4.1. *Brief Description of the Parallel Collision Search Algorithm*

We start with a brief description of the Parallel Collision Search (PCS) algorithm of van Oorschot and Wiener [27]. For more information on the algorithm and its applications, the reader is referred to the original paper [27].

The Memoryless Algorithm The simplest way to present the PCS algorithm is to consider "memoryless" (i.e., constant memory) attacks on r-encryption. As mentioned in Sect. 3, the time complexity of exhaustive search is 2^{rn}, and the MITM attack does not perform better given constant memory. Van Oorschot and Wiener showed that the time complexity can be reduced to $2^{(3/4)rn}$, using the PCS algorithm.

The basic observation behind the algorithm is that given constant memory, one can efficiently find key candidates which comply with half of the plaintext–ciphertext pairs.

Assume, for sake of simplicity, than r is even and the adversary is given r plaintext–ciphertext pairs $(P_1, C_1), \ldots, (P_r, C_r)$. The first step of the PCS algorithm consists of finding candidates for the keys $(k_1, \ldots, k_r)$, which comply with the pairs $(P_1, C_1), \ldots, (P_{r/2}, C_{r/2})$. In order to find them, the adversary constructs the two step functions

$$F^{\text{upper}} : (k_1, \ldots, k_{r/2}) \mapsto (X_1^{r/2}, \ldots, X_{r/2}^{r/2}), \quad \text{and}$$
$$F^{\text{lower}} : (k_{r/2+1}, \ldots, k_r) \mapsto (X_1^{r/2}, \ldots, X_{r/2}^{r/2}),$$

and uses a variant of Floyd's cycle finding algorithm [19] to find a collision between them. Thus, the adversary obtains a value of $(k_1, \ldots, k_{r/2}, k_{r/2+1}, \ldots, k_r)$ which complies with $(P_1, C_1), \ldots, (P_{r/2}, C_{r/2})$. As both functions are from $(r/2)n$ bits to $(r/2)n$ bits, Floyd's algorithm is expected to find a collision in time $2^{(r/4)n}$, with constant memory. In the sequel, we call such collisions *partial collisions*.

In the second step of the algorithm, the adversary checks whether the found key candidate also complies with the pairs $(P_{r/2+1}, C_{r/2+1}), \ldots, (P_r, C_r)$. By standard randomness assumptions, this occurs with probability $2^{-(r/2)n}$, and hence the adversary is expected to find the key after testing $2^{(r/2)n}$ candidates. The total time complexity of the algorithm is thus $2^{(3/4)rn}$.

We note that one may be tempted to use the naive approach to find a collision between F^{upper} and F^{lower}, by trying to construct a self-colliding chain of values, generated by alternating the applications of the two functions on the current value. However, this approach does not work, and in order to efficiently obtain the desired collision, one has to embed pseudo-randomness into the generation of the chain in order to decide at each stage which of the functions to apply next (for more details, refer to [27]). Moreover, the adversary has to use different flavors of the step functions F^{upper} and F^{lower} in order to produce the $2^{(r/2)n}$ *distinct* partial collisions required for the second step of the algorithm. Thus, the algorithm is probabilistic also in the sense that its success probability depends on the (randomly chosen) starting points of Floyd's algorithm and the different flavors.

Time/Memory Tradeoff in Parallel Collision Search If more memory is available, then the algorithm described above can be combined with the techniques used in the classical Hellman's time-memory tradeoff attack [14] to obtain the tradeoff curve $T^2M = 2^{(3/2)rn}$. The reduction in the time complexity is achieved by obtaining many partial collisions simultaneously, with a lower amortized time per collision.

Assume that the available memory is $M = 2^{mn}$. The adversary chooses M random starting points V_i, and for each of them she computes a chain of values starting from V_i. Each chain is terminated once a value with $(r/4 - m/2)n$ zero LSBs is obtained, and this "distinguished point" is stored in a table, along with V_i and its total length. For each reached distinguished point, the adversary checks whether it already appears in the table. If it indeed appears, then the corresponding chains give a collision between F^{lower} and F^{upper} with high probability. This collision can be easily found using the two starting points of the chains, and their lengths.

The expected length of each of the 2^{mn} paths is $2^{(r/4-m/2)n}$, and thus, the structure covers a total of about $2^{(r/4+m/2)n}$ values. Using the birthday paradox, since every path contains about $2^{(r/4-m/2)n}$ values, we expect it to collide with at least one of the $2^{(r/4+m/2)n}$ covered points with high probability. Thus, we expect to find a total of about 2^{mn} partial collisions using this algorithm. Generating the structure requires a total of about $2^{(r/4+m/2)n}$ operations, and we can obtain each of the 2^{mn} partial collisions in about $2^{(r/4-m/2)n}$ time (the expected length of a chain in the structure). Thus, the total time complexity of obtaining the collisions is about $2^{(r/4+m/2)n}$, which implies that the time complexity of the algorithm is also about $2^{(r/4+m/2)n}$.

In total, the algorithm requires about $2^{(r/4+m/2)n}$ operations in order to find 2^{mn} partial collisions. Since $2^{rn/2}$ partial collisions are needed, the overall time complexity is

$$T = 2^{(r/4+m/2)n} \cdot 2^{(r/2-m)n} = 2^{(3r/4-m/2)n},$$

which leads to the tradeoff curve $T^2M = 2^{(3/2)rn}$.

4.2. *The Dissect & Collide Algorithm*

In this section, we present the Dissect & Collide (DC) algorithm, which uses dissection to enhance the PCS algorithm.

The basic idea behind the DC algorithm is that it is possible to fix several intermediate values after $r/2$ rounds, $(X_1^{r/2}, \ldots, X_u^{r/2})$, and construct complex step functions $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$ in such a way that all the keys they suggest partially encrypt P_i to $X_i^{r/2}$ and partially decrypt C_i to $X_i^{r/2}$, for all $i \leq u$. This is achieved by incorporating an attack on $E^{[1\ldots r/2]}$ with $(P_1, X_1^{r/2}), \ldots, (P_u, X_u^{r/2})$ as the plaintext–ciphertext pairs into the function F^{upper}, and similarly with $E^{[r/2+1\ldots r]}$ and F^{lower}. As a result, a partial collision which complies with the pairs $(P_1, C_1), \ldots, (P_{r/2}, C_{r/2})$ can be found at a smaller "cost" than in the PCS algorithm. It should be noted that this gain could potentially be diminished by the "cost" of the new step functions $\tilde{F}$, which is higher than the "cost" of the simple step functions F. However, we show that when the efficient dissection algorithms presented in Sect. 3 are used to attack the subciphers $E^{[1\ldots r/2]}$ and $E^{[r/2+1\ldots r]}$, the gain is bigger than the loss, and the resulting DC algorithm is faster than the PCS algorithm (given the same amount of memory).

A Basic Example: Applying *DC* to 8-encryption As the idea of the DC algorithm is somewhat involved, we illustrate it by considering the simple case ($r = 8, m = 1$). In the case of 8-encryption, the goal of the first step in the PCS algorithm is to find partial collisions which comply with the pairs $(P_1, C_1), \ldots, (P_4, C_4)$. Given memory of 2^n, the average time in which PCS finds each such collision is $2^{1.5n}$. The DC algorithm allows us to achieve the same goal in 2^n average time.

In the DC algorithm, we fix three intermediate values: (X_1^4, X_2^4, X_3^4) and want to attack the subciphers $E^{[1\ldots4]}$ and $E^{[5\ldots8]}$. Recall that the algorithm $Dissect_2(4, 1)$ presented in Sect. 3 retrieves all 2^n values of (k_1, k_2, k_3, k_4) which comply with the pairs $(P_1, X_1^4), (P_2, X_2^4), (P_3, X_3^4)$ in time 2^{2n} and memory 2^n. Furthermore, given a fixed value X_1^2, there is a single value of (k_1, k_2, k_3, k_4) (on average) which complies with the three plaintext–ciphertext pairs and the X_1^2 value, and this value can be found in time 2^n (since the $Dissect_2(4, 1)$ algorithm starts with guessing the value X_1^2 and then performs only 2^n operations for each guess).

Given plaintexts (P_1, P_2, P_3, P_4), their corresponding ciphertexts (C_1, C_2, C_3, C_4), and a guess for (X_1^4, X_2^4, X_3^4):

Define the step functions $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$ by:

$$\tilde{F}^{\text{upper}} : X_1^2 \mapsto X_4^4 \qquad \text{and} \qquad \tilde{F}^{\text{lower}} : X_1^6 \mapsto X_4^4.$$

In order to compute the step function $\tilde{F}^{\text{upper}}$, apply the $Dissect_2(4, 1)$ algorithm to $E^{[1\ldots4]}$ with the plaintext–ciphertext pairs $(P_1, X_1^4), (P_2, X_2^4), (P_3, X_3^4)$ and the intermediate value X_1^2 to obtain a unique value of the keys (k_1, k_2, k_3, k_4). Then, partially encrypt P_4 through $E^{[1\ldots4]}$ with these keys to obtain $\tilde{F}^{\text{upper}}(X_2^1) = X_4^4$. The function $\tilde{F}^{\text{lower}}$ is computed similarly. The resulting algorithm is given in Algorithm 8 and depicted in Fig. 8.

By the construction of the step functions, each suggested key $(k_1, \ldots, k_4)$ (or $(k_5, \ldots, k_8)$) encrypts (P_1, P_2, P_3) to (X_1^4, X_2^4, X_3^4) (or decrypts (C_1, C_2, C_3) to (X_1^4, X_2^4, X_3^4), respectively), and hence, each collision between $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$ yields a suggestion of $(k_1, \ldots, k_4, k_5, \ldots, k_8)$ which complies with the pairs $(P_1, C_1), \ldots, (P_4, C_4)$. Since we find 2^n collisions in step 1(a), we expect a collision for each possible value of X_4^4. Thus, once we iterate over the correct value of (X_1^4, X_2^4, X_3^4), we expect a collision on the correct value of X_4^4, which will suggest the correct key with high probability.

Since the step functions are from n bits to n bits, we can find the required 2^n partial collisions with 2^n invocations using 2^n memory in Step 2. By the properties of the algorithm $Dissect_2(4, 1)$ mentioned above, the invocation of the step functions $\tilde{F}$ can be performed in 2^n time and memory. Thus, Step 2 requires a total of 2^{2n} time, and the total running time of the algorithm is $2^{4n-n} \cdot 2^{2n} = 2^{5n}$.

```
Input: Eight plaintexts $(P_1, P_2, \ldots, P_8)$ and their corresponding ciphertexts $(C_1, C_2, \ldots, C_8)$
1: for all candidate values of $X_1^4, X_2^4, X_3^4$ do
2:     Use PCS to obtain $2^n$ collisions between $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$
3:     for all obtained collisions do
4:         if $E^{[1..8]}_{k_1,\ldots,k_4,k_5,\ldots,k_8}(P_5, P_6, P_7, P_8) = (C_5, C_6, C_7, C_8)$ then
5:             return $k_1, \ldots, k_4, k_5, \ldots, k_8$
```

Algorithm 8: The DC(8, 1) Algorithm

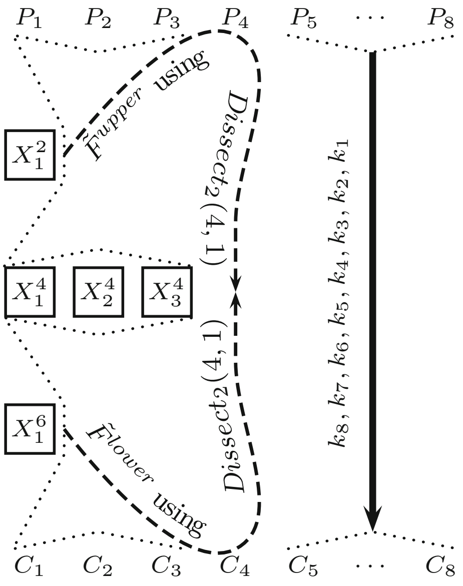
P_1 P_2 P_3 P_4 P_5 $\cdots$ P_8
X_1^2
$\tilde{F}^{upper}$ using
$Dissect_2(4,1)$
X_1^4 X_2^4 X_3^4
X_1^6
$\tilde{F}^{lower}$ using
$Dissect_2(4,1)$
$k_8, k_7, k_6, k_5, k_4, k_3, k_2, k_1$
C_1 C_2 C_3 C_4 C_5 $\cdots$ C_8

Fig. 8. Illustration of the DC(8, 1) algorithm for 8-encryption.

We note that while our DC algorithm outperforms the respective PCS algorithm (whose time complexity is $2^{5.5n}$), it has the same performance as the $Dissect_4(8, 1)$ algorithm presented in Sect. 3. However, as we show in the sequel, for larger values of r, the DC algorithms outperform the $Dissect$ algorithms significantly.

The General $DC(r, m)$ Algorithms We are now ready to give a formal definition of the DC(r, m) class of algorithms, applicable to r-encryption (for an even r),[14] given memory of 2^{mn}. An algorithm $A \in$ DC(r, m) is specified by a number u, $1 \leq u \leq r/2$,

[14] We note that for sake of simplicity, we discuss in this section only even values of r. An easy (but probably non-optimal) way to use these algorithms for an odd value of r is to guess the value of the key k_r, and for each guess, to apply the algorithms described in this section to $E^{[1\ldots r-1]}$.

```
Input: $r$ plaintexts $(P_1, P_2, \ldots, P_r)$ and their corresponding ciphertexts $(C_1, C_2, \ldots, C_r)$
1: for all candidate values of $X_1^{r/2}, X_2^{r/2}, \ldots, X_u^{r/2}$ do
2:     Use PCS to obtain $2^m n$ collisions between $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$
3:     for all obtained collisions do
4:         if $E^{[1..r]}_{k_1,\ldots,k_{r/2},k_{r/2+1},\ldots,k_r}(P_{r/2+1}, P_{r/2+2}, \ldots, P_r) = (C_{r/2+1}, C_{r/2+2}, \ldots, C_r)$ then
5:             return $k_1, \ldots, k_{r/2}, k_{r/2+1}, \ldots, k_r$
```

Algorithm 9: The DC(r, m) Algorithm

and two sets I^{upper} and I^{lower} of intermediate locations in the subciphers $E^{[1\ldots r/2]}$ and $E^{[r/2+1\ldots r]}$, respectively, such that $|I^{\text{upper}}| = |I^{\text{lower}}| = r/2 - u$.

To apply the algorithm, u intermediate values $(X_1^{r/2}, \ldots, X_u^{r/2})$ are fixed, and the step functions $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$ are defined by:

$$\tilde{F}^{\text{upper}} : V^{\text{upper}} \mapsto (X_{u+1}^{r/2}, \ldots, X_{r/2}^{r/2}), \quad \text{and}$$
$$\tilde{F}^{\text{lower}} : V^{\text{lower}} \mapsto (X_{u+1}^{r/2}, \ldots, X_{r/2}^{r/2}),$$

where V^{upper} and V^{lower} denote intermediate values at the locations I^{upper} and I^{lower}, respectively.

The step function $\tilde{F}^{\text{upper}}$ is computed by applying a dissection attack to $E^{[1\ldots r/2]}$ with the plaintext–ciphertext pairs $(P_1, X_1^{r/2}), \ldots, (P_u, X_u^{r/2})$, where the intermediate values at locations I^{upper} are fixed to the values V^{upper}, to retrieve a unique value of the keys $(k_1, \ldots, k_{r/2})$, and then partially encrypting $(P_{u+1}, \ldots, P_{r/2})$ with these keys to obtain $(X_{u+1}^{r/2}, \ldots, X_{r/2}^{r/2})$. The step function $\tilde{F}^{\text{lower}}$ is computed in a similar way, with respect to $E^{[r/2+1\ldots r]}$ and the set I^{lower}. The adversary finds 2^{mn} collisions between $\tilde{F}^{\text{upper}}$ and $\tilde{F}^{\text{lower}}$ using a Hellman-like data structure, where each collision gives a suggestion of $(k_1, \ldots, k_{r/2}, k_{r/2+1}, \ldots, k_r)$, complying with the plaintext–ciphertext pairs (P_1, C_1), $\ldots$, $(P_{r/2}, C_{r/2})$. The resulting algorithm is described in Algorithm 9.

Denote the time complexity of each application of $\tilde{F}$ by $S = 2^{sn}$. An easy computation shows that the overall time complexity of the algorithm DC(r, m) is:

$$2^{(r/2)n} \cdot 2^{((r/2-u-m)/2)n} \cdot 2^{sn} = 2^{((3/4)r-(u+m-2s)/2)n}. \tag{4}$$

As the time complexity of the PCS algorithm with memory 2^{mn} is $2^{((3/4)r-m/2)n}$, the multiplicative advantage of the DC algorithm is $2^{(u/2-s)n}$. In particular, for the specific DC$(8, 1)$ algorithm described above for 8-encryption, we have $s = 1$, and thus, the gain is indeed $2^{(3/2-1)n} = 2^{n/2}$. In the sequel, we denote the parameters I^{upper}, I^{lower}, u, s which specify a DC(r, m) algorithm A and determine its time complexity by $I^{\text{upper}}(A)$, $I^{\text{lower}}(A)$, $u(A)$, and $s(A)$, respectively.

Flavors in the Step Function of the Algorithm *DC* We conclude this section by pointing out a difficulty in the implementation of the DC algorithm (which does not exist in the PCS algorithm) and presenting a way to resolve it.

Recall that in the PCS algorithm, for each value of $(k_1, \ldots, k_{r/2})$, there is exactly one corresponding value of $\tilde{F}^{\text{upper}}(k_1, \ldots, k_{r/2}) = (X_1^{r/2}, \ldots, X_{r/2}^{r/2})$. On the other hand, in DC, for some values of V^{upper}, there are no corresponding values of $\tilde{F}^{\text{upper}}(V^{\text{upper}}) = (X_{u+1}^{r/2}, \ldots, X_{r/2}^{r/2})$ at all, while for other values of V^{upper}, there are several possible outputs of $\tilde{F}^{\text{upper}}$. This happens since the number of keys $(k_1, \ldots, k_{r/2})$ which comply with the u plaintext–ciphertext values $(P_1, X_1^{r/2}), \ldots, (P_u, X_u^{r/2})$ and the $r/2 - u$ fixed intermediate values at the locations I^{upper} is not always 1, but is distributed according to a Poisson distribution with mean 1. While this feature does not influence the expected performance of the *Dissect* attacks, its effect on the DC attack is crucial: in an e^{-1} fraction of the cases, the step function $\tilde{F}^{\text{upper}}$ returns no value, and thus, the expected length of its generated chains is constant!

In order to resolve this difficulty, we introduce *flavors* into the definition of the step function. Formally, for each value of V^{upper}, $\tilde{F}^{\text{upper}}(V^{\text{upper}})$ is a (possibly empty) multiset. Based on this, we define:

$$\bar{F}^{\text{upper}}(V^{\text{upper}}) = \min(\{\tilde{F}^{\text{upper}}(V^{\text{upper}} \oplus i_0)\}),$$

where $i_0 \in \{0, 1\}^{(r/2-u)n}$ is minimal such that the set $\{\tilde{F}^{\text{upper}}(V^{\text{upper}} \oplus i_0)\}$ is non-empty, and the minimums are taken with respect to the lexicographic order. In other words, if the set $\tilde{F}^{\text{upper}}(V^{\text{upper}})$ is empty, then we replace V^{upper} by $V^{\text{upper}} \oplus (0, 0, \ldots, 0, 1)$ and compute $\tilde{F}^{\text{upper}}$ again. We continue until we reach a value of i_0 for which the set of outputs of $\tilde{F}^{\text{upper}}$ is non-empty, and the single output $\bar{F}^{\text{upper}}(V^{\text{upper}})$ is chosen from this set according to the lexicographical order. The same modification is applied also to $\tilde{F}^{\text{lower}}$.

Using this modification, the step function becomes uniquely defined as in the case of PCS, and the computation overhead required by the new definition is small, since the output of $\tilde{F}^{\text{upper}}$ is non-empty for an $1 - 1/e$ fraction of the inputs. On the other hand, the introduction of flavors gives rise to another difficulty which does not arise in the original PCS algorithm, namely, the possibility that in an execution of the algorithm with the correct values of $I^{\text{upper}}/I^{\text{lower}}$, the correct key will not be found since the set of outputs of $\tilde{F}^{\text{upper}}$ contains more than one solution. Since the algorithm is forced to pick only one solution, it may miss the correct key. To deal with this possibility, if we do not find the key in an execution of the algorithm, we change the ordering algorithm of the solutions and run the algorithm again. Since we expect no more than a few solutions in each execution of the step functions, we expect to repeat the algorithm only a small number of times. After these modifications, which result in a small (constant) penalty in its time complexity, we can claim that the success probability of the DC algorithm can be made arbitrarily close to 1 (similarly to the PCS algorithm).

4.3. *The Gain of the Dissect & Collide Algorithm Over the PCS Algorithm*

In this section, we consider several natural extensions of the basic DC(8, 1) algorithm presented in Sect. 4.2. We use these extensions to show that the gain of the DC algorithms over the PCS algorithm is monotone non-decreasing with r and is lower bounded by $\lfloor\sqrt{2r}\rfloor/8$ for any $r \geq 8$.

Before we present the extensions of the basic DC algorithm, we formally define the notion of *gain* in the non-deterministic setting. As the best previously known algorithm in this setting is the PCS algorithm, whose time complexity given 2^{mn} memory is $2^{((3/4)r-m/2)n}$, we define the gain with respect to it.

Definition 2. The gain of a probabilistic algorithm A for r-encryption whose time and memory complexities are T and $M = 2^{mn}$, respectively, is defined as

$$\mathrm{Gain}_{\mathrm{ND}}(A) = (3/4)r - m/2 - (\log T)/n.$$

The maximal gain among all probabilistic DC algorithms for r-encryption which require 2^{mn} memory, is denoted by $\mathrm{Gain}_{\mathrm{ND}}(r, m)$.

Note that it follows from Eq. (4) that if $A \in \mathrm{DC}(r, m)$, then

$$\mathrm{Gain}_{\mathrm{ND}}(A) = u(A)/2 - s(A). \tag{5}$$

Monotonicity of the gain The simplest extension of the basic DC algorithm preserves the gain when additional "rounds" are added. While in the deterministic case, such an extension can be obtained trivially by guessing several keys and applying the previous algorithm, in our setting this approach leads to a decrease of the gain by $1/2$ for each two added rounds (as the complexity of the PCS algorithm is increased by a factor of $2^{3n/2}$ when r is increased by 2). However, the gain can be preserved in another way, as shown in the following lemma.

Lemma 1. *Assume that an algorithm $A \in DC(r', m)$ has gain ℓ. Then, there exists an algorithm $B \in DC(r' + 2, m)$ whose gain is also equal to ℓ.*

Proof. Recall that the sets of intermediate locations fixed in the algorithm A are denoted by $I^{\mathrm{upper}}(A)$ and $I^{\mathrm{lower}}(A)$. We describe the algorithm B by fixing the sets of intermediate values at locations: $I^{\mathrm{upper}}(B) = I^{\mathrm{upper}}(A) \cup \{X_1^{r'/2}\}$, and similarly for $I^{\mathrm{lower}}(B)$. Note that as the DC algorithms for r-encryption satisfy $u + |I^{\mathrm{upper}}| = r/2$, our choice of $I^{\mathrm{upper}}(B)$ and $I^{\mathrm{lower}}(B)$ ensures that $u(B) = u(A)$. Hence, Eq. (4) implies that in order to show that $\mathrm{Gain}_{\mathrm{ND}}(B) = \mathrm{Gain}_{\mathrm{ND}}(A)$, it is sufficient to show that $s(B) = s(A)$. Let $Step(A)$ be an algorithm which allows us to compute the function $\tilde{F}^{\mathrm{upper}}(A) : V^{\mathrm{upper}}(A) \mapsto (X_{u+1}^{r'/2}, \ldots, X_{r'/2}^{r'/2})$ in time 2^{sn} (where $u = u(A)$ and $s = s(A)$), given the plaintext–ciphertext pairs $(P_1, X_1^{r'/2}), \ldots, (P_u, X_u^{r'/2})$ and the intermediate values at locations $I^{\mathrm{upper}}(A)$. We have to find an algorithm $Step(B)$ which computes the function $\tilde{F}^{\mathrm{upper}}(B) : V^{\mathrm{upper}}(B) \mapsto (X_{u+1}^{(r'+2)/2}, \ldots, X_{(r'+2)/2}^{(r'+2)/2})$ in time 2^{sn}

given the plaintext–ciphertext pairs $(P_1, X_1^{(r'+2)/2}), \dots, (P_u, X_u^{(r'+2)/2})$ and the intermediate values at locations $I^{\text{upper}}(B)$. We define the algorithm $Step(B)$ as follows:

1. Use the values $X_1^{r'/2}$ and $X_1^{(r'+2)/2}$ to compute a unique value of the key $k_{(r'+2)/2}$ which complies with them.
2. Use the value of $k_{(r'+2)/2}$ to partially decrypt the vector $(X_1^{(r'+2)/2}, \dots, X_u^{(r'+2)/2})$ through $E^{(r'+2)/2}$ to obtain the vector $(X_1^{r'/2}, \dots, X_u^{r'/2})$.
3. Use the algorithm $Step(A)$ to deduce the vector $(X_{u+1}^{r'/2}, \dots, X_{r'/2}^{r'/2})$ from the plaintext–ciphertext pairs $(P_1, X_1^{r'/2}), \dots, (P_u, X_u^{r'/2})$ and the intermediate values at locations $I^{\text{upper}}(A)$.
4. Partially, encrypt the vector $(P_1, X_1^{r'/2}), \dots, (P_u, X_u^{r'/2})$ through $E^{(r'+2)/2}$ to obtain the desired vector $\tilde{F}^{\text{upper}}(B)(V^{\text{upper}}(B)) = (X_{u+1}^{(r'+2)/2}, \dots, X_{(r'+2)/2}^{(r'+2)/2})$.

It is clear that Step 1 of the algorithm requires at most 2^n operations, Steps 2 and 4 require at most r operations each, and Step 3 requires 2^{sn} operations. Hence, if $s \geq 1$ (which is the case for all DC algorithms), the time complexity of $Step(B)$ is indeed equal to that of $Step(A)$. The same argument applies also to the function $\tilde{F}_{\text{lower}}$. Finally, it is clear that the memory requirement of B is equal to the memory requirement of A, which completes the proof. □

Lemma 1 implies that the gain of the DC algorithms is monotone non-decreasing with r, and in particular, that $\text{Gain}_{\text{ND}}(r, 1) \geq 1/2$, for any even $r \geq 8$.

An analogue of the *LogLayer* algorithm The next natural extension of the basic DC algorithm is an analogue of the *LogLayer* algorithm presented in Sect. 3.3. Recall that the $LogLayer_r$ algorithm, applicable when r is a power of 2, consists of guessing the set of intermediate values:

$$I_0 = \left\{X_1^2, X_1^4, \dots, X_1^{r-2}, X_2^4, X_2^8, \dots, X_2^{r-4}, X_3^8, \dots, X_3^{r-8}, \dots, X_{\log r-1}^{r/2}\right\},$$

and applying a recursive sequence of MITM attacks on 2-encryption. Using this algorithm, we can define the algorithm $LL_r \in \text{DC}(2r, 1)$, by specifying $I^{\text{upper}}(LL_r) = I_0$, and $I^{\text{lower}}(LL_r)$ in a similar way. Since $|I_0| = r - \log r - 1$, we have $u(LL_r) = r - (r - \log r - 1) = \log r + 1$. It follows from the structure of the $LogLayer_r$ algorithm that given the values in I_0, it can compute the keys $(k_1, \dots, k_r)$ in time and memory of 2^n. Hence, we have $s(LL_r) = 1$. By Eq. (5), it follows that $\text{Gain}(LL_r) = (\log r + 1)/2 - 1 = (\log r - 1)/2$.

The basic algorithm for 8-encryption is the special case of this algorithm LL_4. The next two values of r also yield interesting algorithms: LL_8 yields gain of 1 for $(r = 16, m = 1)$, which amounts to an attack on 16-encryption with $(T = 2^{10.5n}, M = 2^n)$, and LL_{16} yields gain of 1.5 for $(r = 32, m = 1)$, which amounts to an attack on 32-encryption with $(T = 2^{22n}, M = 2^n)$. Both attacks outperform the *Dissect* attacks and are the best-known attacks on 16-encryption and on 32-encryption, respectively.

An analogue of the $\boldsymbol{Square_r}$ algorithm: The logarithmic asymptotic gain of the LL sequence can be significantly outperformed by an analogue of the $Square_r$ algorithm, presented in Sect. 3.3. Recall that the $Square_r$ algorithm, applicable when $r = (r')^2$ is a perfect square, starts by guessing the set of $(r' - 1)^2$ intermediate encryption values:

$$I_1 = \{X_1^{r'}, X_2^{r'}, \dots, X_{r'-1}^{r'}, X_1^{2r'}, X_2^{2r'}, \dots, X_{r'-1}^{2r'}, \dots, \\ X_1^{r'(r'-1)}, X_2^{r'(r'-1)}, \dots, X_{r'-1}^{r'(r'-1)}\},$$

and then performs a two-layer attack, which amounts to $r' + 1$ separate attacks on r'-encryption. Using this algorithm, we can define the algorithm $Sq_r \in \mathrm{DC}(2r, 1)$, by specifying $I^{\text{upper}}(Sq_r) = I_1$, and $I^{\text{lower}}(Sq_r)$ in a similar way. Since $|I_1| = (r'-1)^2$, we have $u(Sq_r) = r-(r'-1)^2 = 2r'-1$. The step complexity $s(Sq_r)$ is the time complexity required for attacking r'-encryption without fixed intermediate values. Hence, by Eq. (5),

$$Gain(Sq_r) = r' - 1/2 - f_1(r'),$$

where $2^{f_1(r)n}$ is the time complexity of our best attack on r-encryption with 2^n memory.

The basic algorithm for 8-encryption is the special case Sq_2 of this algorithm. Since for small values of r', the best-known attacks on r'-encryption are obtained by the dissection attacks presented in Sect. 3.4, the next elements of the sequence Sq_r which increase the gain, correspond to the next elements of the sequence $Magic_1 = \{1, 2, 4, 7, 11, 16, \dots\}$ described in Sect. 3.4. They lead to gains of 1.5, 2.5, and 3.5 for $r = 32, 98$, and 242, respectively. For large values of r, the PCS algorithm outperforms the $Dissect$ algorithms, and using it we obtain:

$$Gain(Sq_r) \geq r' - 1/2 - ((3/4)r' - 1/2) = r'/4 = \sqrt{2r}/8.$$

This shows that the asymptotic gain of the DC algorithms is at least $\sqrt{2r}/8$.

We note that as for $r' \geq 16$, the DC algorithm outperforms both the $Dissect$ and the PCS algorithms, we can use it instead of PCS in the attacks on r'-encryption in order to increase the gain for large values of r. However, as the gain of DC over PCS for r'-encryption is only of order $O(\sqrt{r'}) = O(r^{1/4})$, the addition to the overall gain of Sq_r is negligible.

Two-layer *DC* algorithms A natural extension of the Sq_r algorithm is the class of *two-layer* DC algorithms. Assume that $r = 2r_1 \cdot r_2$, and that there exist algorithms A_1, A_2 for r_1-encryption and for r_2-encryption, respectively, both of which perform in time 2^{sn} and memory 2^n given sets of intermediate values I_1^{upper} and I_2^{upper}, respectively.

Then we can define an algorithm $A \in \mathrm{DC}(r, 1)$ whose step function is computed by a two-layer algorithm: First, $E^{[1\dots r/2]}$ is divided into r_2 subciphers of r_1 rounds each, and algorithm A_1 is used to attack each of them separately and compute 2^n possible suggestions for each set of r_1 consecutive keys. Then, each r_1-round encryption is considered as a single encryption with 2^n possible keys, and algorithm A_2 is used to attack the resulting r_2-encryption. The set $I^{\text{upper}}(A)$ is chosen such that both A_1 and A_2 algorithms perform in time 2^{sn}. Formally, if we denote $u_1 = |I_1^{\text{upper}}|$, then the set $I^{\text{upper}}(A)$ consists of r_2 "copies" of the set I_1^{upper}, $r_1 - 1 - u_1$ intermediate values after each r_1 rounds, and one copy of the set I_2^{upper}. The set $I^{\text{lower}}(A)$ is defined similarly. Hence,

$$u(A) = r/2 - |I^{\text{upper}}(A)| = r/2 - (r_2 \cdot u_1 + (r_2 - 1)(r_1 - 1 - u_1) + u_2)$$
$$= r_2 + r_1 - u_1 - u_2 - 1.$$

As $s(A) = s$, we have $\text{Gain}_{\text{ND}}(A) = (r_2 + r_1 - u_1 - u_2 - 1)/2 - s$.

Note that the algorithm Sq_r is actually a two-layer DC algorithm, with $r_1 = r_2 = r'$ and $I_1^{\text{upper}} = I_2^{\text{upper}} = \emptyset$. It turns out that for all $8 \le r \le 128$, our maximal gains are obtained by two-layer DC algorithms where r_1, r_2 are chosen from the sequence $Magic_1$ presented in Sect. 3.4, and A_1, A_2 are the respective $Dissect$ algorithms. The cases of $r = 8, 16, 32$ presented above are obtained with $r_1 = 4$ and $r_2 = 1, 2, 4$ (respectively), and the next numbers of rounds in which the gain increases are $r = 56, 88, 128$, obtained for $r_1 = 4$ and $r_2 = 7, 11, 16$, respectively. The continuation of the "non-deterministic magic sequence" is, however, more complicated. For example, the two-layer algorithm for $r = 176$ with $(r_1 = 4, r_2 = 22)$ has the same gain as the algorithm with $(r_1 = 4, r_2 = 16)$, and the next increase of the gain occurs only for $r = 224$, and is obtained by a two-layer algorithm with $(r_1 = 7, r_2 = 16)$. For larger values of r, more complex algorithms, such as a three-layer algorithm with $r_1 = r_2 = r_3 = 7$ for 686-encryption, outperform the two-layer algorithms. We leave the analysis of the whole magic sequence as an open problem, and conclude that using the two-layer algorithms, the minimal numbers of rounds for which the gain equals 0.5, 1, 1.5, 2, ... are:

$$Magic_1^{\text{ND}} = \{8, 16, 32, 56, 88, 128, \ldots\}.$$

Finally, we note that a similar analysis to that presented in Sect. 3.6 shows that two-layer DC algorithms can be applied also for $m > 1$, and can be used to show that the first numbers of rounds for which $\text{Gain}_{\text{ND}}(r, m) = 0.5, 1, 1.5, 2, \ldots$ are:

$$Magic_m^{\text{ND}} = \{8m, 8m + 8, 8m + 16, \ldots, 16m, 16m + 16, 16m + 32, \ldots,$$
$$32m, 32m + 24, 32m + 48, \ldots, 56m, \ldots\}.$$

We give in Table 2 a comparison between the time complexities of the $Dissect$, PCS, and DC algorithms. We also give in Fig. 9 a comparison of the gains obtained by all the algorithms presented given 2^n memory.

5. Applications

In this section, we apply our new dissection algorithms to the knapsack problem in Sect. 5.1, and show how dissection can be used to reduce the memory complexity of *rebound* attacks on hash functions in Sect. 5.2.

The application to knapsacks consists of two stages: first, we find a way to represent the problem as a *bicomposite* problem, and then we choose the best appropriate dissection algorithm to solve it. In the case of the knapsack problem, the bicomposite representation is simple, and moreover, we have the freedom to choose the value of r, in order to optimize the complexities. By using different choices of r and m, we obtain a complete tradeoff curve between the time and memory complexities, which is better than the

Table 2. Comparison of the time complexities of Dissect, PCS, and Dissect & Collide.

r	$m = 1$			$m = 2$			$m = 3$		
	Dissect	PCS	DC	Dissect	PCS	DC	Dissect	PCS	DC
2	1	1	1	1	–	–	1	–	–
4	2	2.5	2.5	2	2	2	2	–	–
6	4	4	4	3	3.5	3.5	3	3	3
8	5	5.5	**5**	4	5	5	4	4.5	4.5
10	7	7	6.5	6	6.5	6.5	5	6	6
12	8	8.5	8	7	8	8	6	7.5	7.5
14	10	10	9.5	8	9.5	9.5	8	9	9
16	11	11.5	**10.5**	10	11	**10.5**	9	10.5	10.5
18	13	13	12	11	12.5	12	10	12	12
20	15	14.5	13.5	13	14	13.5	12	13.5	13.5
22	16	16	15	14	15.5	15	13	15	15
24	18	17.5	16.5	16	17	**16**	15	16.5	**16**
26	20	19	18	18	18.5	17.5	16	18	17.5
28	22	20.5	19.5	20	20	19	18	19.5	19
30	23	22	21	21	21.5	20.5	19	21	20.5
32	25	23.5	**22**	22	23	**21.5**	21	22.5	**21.5**
34	27	25	23.5	24	24.5	23	22	24	23
36	29	26.5	25	26	26	24.5	24	25.5	24.5
38	30	28	26.5	27	27.5	26	25	27	26
40	32	29.5	28	29	29	27.5	27	28.5	**27**

Items marked in bold are magic numbers

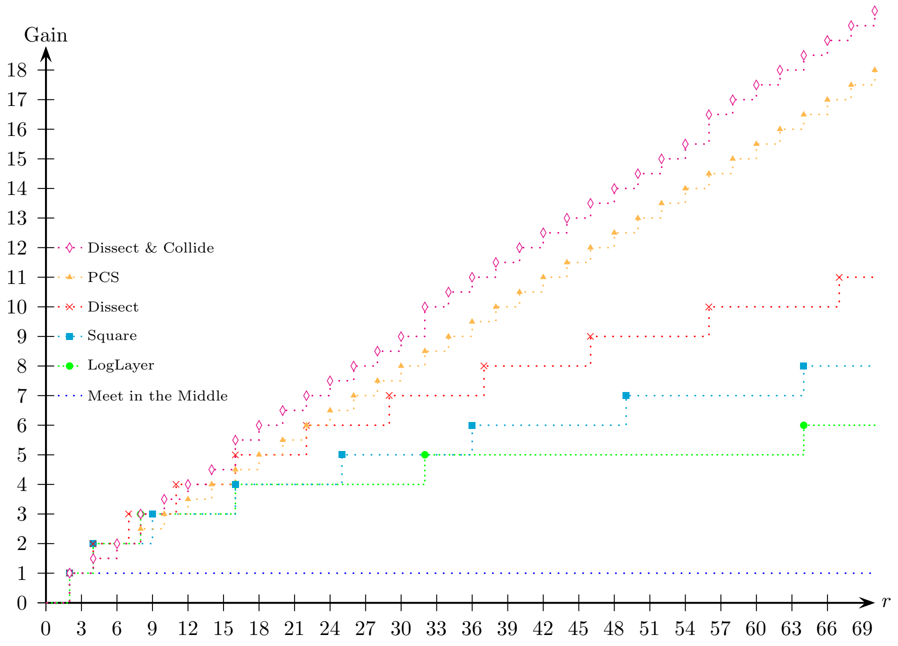

Fig. 9. Comparison of the gain of all algorithms for $M = 2^n$.

best previously known curve for all $2^{n/100} \leq M \leq 2^{n/6}$. We note that the analyses of our algorithms assume a uniform distribution of solutions. In the case of knapsacks, our deterministic algorithms can be adapted (with only a small memory overhead) to deal with cases where the distribution of solutions is far from uniform, as shown in the follow-up paper [1].

5.1. *Application to Knapsacks*

The knapsack problem is defined as follows: given a list of n positive integers $x_1, x_2, \ldots, x_n$ of n bits and an additional n-bit positive integer S, find a vector $\epsilon = (\epsilon_1, \epsilon_2, \ldots, \epsilon_n) \in \{0, 1\}^n$, such that $S = \sum_{i=1}^{n} \epsilon_i \cdot x_i \pmod{2^n}$.[15]

Knapsack is a well-known problem that has been studied for many years. For more than 30 years, the best-known algorithm for knapsacks was the Schroeppel–Shamir algorithm [28], which requires $2^{n/2}$ time and $2^{n/4}$ memory. In 2010, Howgrave-Graham and Joux [15] showed how to solve the knapsack problem in time $2^{0.337n}$ and memory $2^{0.256n}$, using a specialized algorithm (which does not apply if the addition is replaced by a non-commutative operation, such as the rotations in the Rubik's cube problem). The specialized algorithm of [15] was further improved by Becker, Coron and Joux [4] into an algorithm with time and memory complexities of $2^{0.291n}$. In addition, Becker et al. presented a specialized memoryless attack which requires only $2^{0.72n}$ time. All these attacks are heuristic in the sense that they may fail to find a solution even when it exists, and thus they cannot be used in order to prove the nonexistence of solutions. In addition to these heuristic algorithms, Becker, Coron and Joux [4] also considered deterministic algorithms that never err, and obtained a straight-line time-memory tradeoff curve of $TM = 2^{3n/4}$, for all $2^{n/16} \leq M \leq 2^{n/4}$.

It is worth noting that a pseudo-polynomial algorithm for the knapsack problem is suggested in [22] which uses polynomial storage. However, this algorithm's running time is about 2^n in the cryptographic settings (in which S is roughly 2^n).

In this section, we show how to use our generic dissection techniques in order to find deterministic algorithms for the knapsack problem which are better than the deterministic tradeoff curve described in [4] over the whole range of $2^{n/16} < M < 2^{n/4}$. In addition, we can expand our tradeoff curve in a continuous way for any smaller value of $M \leq 2^{n/4}$. By combining our generic deterministic and non-deterministic algorithms, we obtain a new curve which is better than the best knapsack-specific algorithms described in [15] and [4] in the interval $2^{n/100} < M < 2^{n/6}$. Note that the algorithms of [4,15] outperform our algorithms for $M > 2^{n/6}$ and in the memoryless setting.

We note that all the results presented in this section can easily be adapted to the closely related partition problem, in which we are given a set of n integers, $U = \{x_1, x_2, \ldots, x_n\}$, and our goal is to partition U into two complementary subsets U_1, U_2 whose elements sum up to the same value.

Representing Knapsack as a Bicomposite Problem First, we represent knapsack as a composite problem. We treat the problem of choosing the vector $\epsilon = (\epsilon_1, \ldots, \epsilon_n)$ as

[15] We work with modular knapsacks which are in general computationally equivalent to arbitrary knapsacks [4,15].

a sequence of n atomic decisions, where the i'th decision is whether to assign $\epsilon_i = 0$ or $\epsilon_i = 1$. We introduce a counter C which is initially set to zero, and then at the i'th step, if the choice is $\epsilon_i = 1$ then C is replaced by $C + x_i (\text{mod} 2^n)$, and if the choice is $\epsilon_i = 0$, then C is left unchanged. Note that the value of C after the n'th step is $\sum_i = \epsilon_i x_i (\text{mod} 2^n)$, and hence, the sequence of choices leads to the desired solution if and only if the final value of C is S.

In this representation, the partition problem has all the elements of a composite problem: an initial state ($C_{initial} = 0$), a final state ($C_{final} = S$), and a sequence of n steps, such that in each step, we have to choose one of two possible atomic and invertible actions. Our goal is to find a sequence of choices which leads from the initial state to the final state. In terms of the execution matrix, we define S_i to be the value of C after the i'th step (which is an n-bit binary number), and a_i to be the action transforming S_{i-1} to S_i, whose possible values are either $C \leftarrow C + x_i (\text{mod} 2^n)$ or $C \leftarrow C$.

The second step is to represent the problem as a bicomposite problem. The main observation we use here is the fact that for any two integers a, b, the m'th least significant bit of $a + b(\text{mod} 2^n)$ depends only on the m least significant bits of a and b (and not on their other bits). Hence, if we know the m LSBs of S_{i-1} and the action a_i, we can compute the m LSBs of S_i.

Using this observation, we define $S_{i,j}$ to be the j'th most significant bit of S_i. This leads to an n-by-n execution matrix $S_{i,j}$ for $i, j \in 1, 2, \ldots, n$ with the property that if we choose any rectangle within the execution matrix *which includes the rightmost column of the matrix*, knowledge of the substates $S_{i-1,j}, S_{i-1,j+1}, \ldots, S_{i-1,n}$ along its top edge and knowledge of the actions $a_i, a_{i+1}, \ldots, a_\ell$ to its left suffices in order to compute the substates $S_{\ell,j}, S_{\ell,j+1}, \ldots, S_{\ell,n}$ along its bottom edge.

In order to handle general rectangles, we ensure that all enumerations of intermediate states in the algorithm are done from right to left, i.e., in each state, the LSB is guessed first, then the second least significant bit is guessed, etc. As a result, when we deal with the processing of the state $S_{i-1,j}$ to $S_{\ell,j}$ via the actions $a_i, a_{i+1}, \ldots, a_\ell$, we already know the states $S_{i-1,k}$ for all $k > j$, and hence, we can keep track of the carry bits into the jth bit in the addition operations. Thus, our representation effectively satisfies the conditions of a bicomposite representation.

So far, we have represented knapsack as a bicomposite problem with an n-by-n execution matrix. In order to make the representation similar to the multiple encryption problem considered in Sects. 3 and 4, we note that for any $r \ll n$, the problem has a bicomposite representation with an r-by-r execution matrix.

Indeed, in the representation as a composite problem above, we can group sequences of n/r consecutive decisions,[16] such that we have r atomic decisions, where in the ith atomic decision we choose $(\epsilon_{(i-1)n/r+1}, \epsilon_{(i-1)n/r+2}, \ldots, \epsilon_{in/r}) \in \{0, 1\}^{n/r}$, and the operation is

$$C \leftarrow C + \sum_{j=(i-1)n/r+1}^{in/r} \epsilon_j x_j \pmod{2^n}.$$

[16]Since we are mostly interested in asymptotic analysis, and since $r \ll n$, we assume for sake of simplicity that n is divisible by r.

Similarly, in the bicomposite representation, we define $S_{i,j}$ as an n/r-bit value which consists of bits $(j-1)n/r+1, (j-1)n/r+2, \ldots, jn/r$ of S_i, where the counting order starts with the MSB indexed by 1.

In the knapsack problem we have the freedom to choose r such that the time complexity of the algorithm will be minimized, for a given amount of memory.

Formally, for any $r \ll n$, we apply one of the algorithms for multiple encryption described in Sects. 3 and 4 to an r-encryption scheme with a block size of $n^* = n/r$ bits and a memory parameter of m^*. As the memory complexity of the resulting algorithm is $2^{m^*n^*} = 2^{m^*n/r}$, it follows that if we want the memory complexity to be 2^{mn}, we should consider $m^* = rm$.

We denote by $f(r, n^*, m^*)$ the running time of our optimal dissection algorithm (among the algorithms presented in this paper) for r-encryption with a block size of n^* bits and $M^* = 2^{m^*n^*}$ available memory. In these notations, the problem of finding the optimal dissection algorithm for n-bit knapsack is reduced to finding r that minimizes $f(r, n^*, m^*) = f(r, n/r, mr)$. We call such a value of r an optimal value.

We note that the deterministic algorithms applied in [15] and [4] for $2^{n/16} \leq M \leq 2^{n/4}$ implicitly perform a reduction to multiple encryption with the fixed parameters $r = 4$ and $r = 16$. In fact, these algorithms are closely related to our $Square_r$ algorithms described in Sect. 3.3. However, as we show below, we can get a better tradeoff curve by using other choices of r.

Dissecting the Knapsack Problem - a Summary In this section, we use the dissection algorithms presented in Sects. 3 and 4 to obtain a new time-memory tradeoff curve of algorithms for the knapsack problem. We aim at obtaining a complete curve, which yields for any fixed memory complexity 2^{mn}, the time complexity 2^{tn} of our optimal dissection algorithm.

We consider deterministic and general algorithms separately. In the deterministic case, we show below that the curve is piece-wise linear, with "cut" points at all values of the form $m = 1/b_j$ where b_j is the jth element of the sequence $Magic_1 = \{2, 4, 7, 11, 16, 22, 29, \ldots\}$ presented in Sect. 3. For each such $m = 1/b_j$, an optimal algorithm is obtained by choosing $r = b_j$ and $m^* = rm = 1$. The tradeoff in the deterministic case is presented in Fig. 10.

In the non-deterministic case, the situation is similar, with the sequence $Magic_1$ replaced by the corresponding magic sequence presented in Sect. 4, i.e., $Magic_1^{\mathrm{ND}} = \{8, 16, 32, 56, 88, 128, \ldots\}$. It turns out that non-deterministic dissection algorithms outperform the deterministic ones for $m < 9/104$. A comparison between our general tradeoff curve and the previous results of [4,15] *for small memory complexities* is presented in Fig. 11.[17]

Optimal Choice of *r* for Deterministic Dissection algorithms Recall that given r and m, we apply a deterministic dissection algorithm with an effective block size of $n^* = n/r$ and an effective memory unit of $m^* = mr$. We would like to find, for a fixed m, the value of r that minimizes the running time of our algorithm $2^{tn} = f(r, n^*, m^*) = f(r, n/r, mr)$.

[17] In Fig. 11 we do not include the curve of $TM = 2^{3n/4}$ of [4] and the PCS curve of [27], since the curve obtained using our algorithms is strictly better for the whole range of $0 < m < 1/4$.

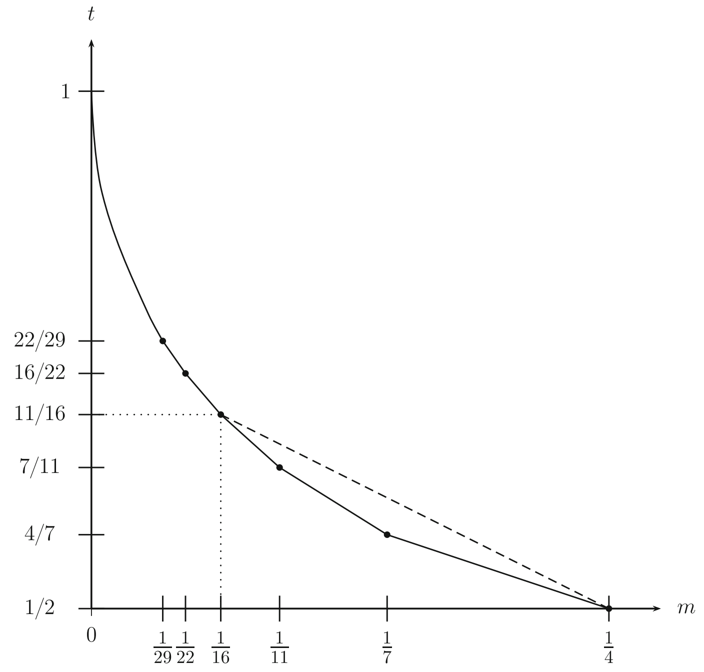

Fig. 10. Time-memory tradeoff curves for knapsack for deterministic algorithms. A comparison between our time-memory tradeoff curve and the curve obtained in [4] (shown as a dashed line) for deterministic algorithms. Our curve (defined for $m \leq 1/4$) is strictly better than the curve obtained in [4] (defined only for $1/16 \leq m \leq 1/4$) for any $1/16 < m < 1/4$.

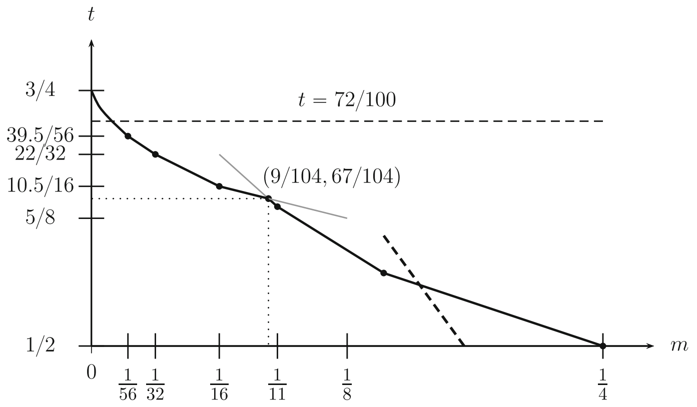

Fig. 11. Time-memory tradeoff curves for knapsack for general algorithms. A comparison in the range $0 \leq m \leq 1/4$ between our general time-memory tradeoff curve, the curve obtained by extending the ($m = 0.211, t = 0.421$) attack given in [15] (shown as a bold dashed line), and the memoryless attack with $t = 0.72$ obtained in [4] (shown as a light dashed horizontal line). Our general time-memory tradeoff curve is better than the attacks of [4] and [15] in the interval of (approximately) $1/100 \leq m < 1/6$.

First, we note that for any natural number z, it is easy to reduce our dissection algorithm with parameters $(z \cdot r, n^*/z, z \cdot m^*)$, to an algorithm with parameters (r, n^*, m^*) that runs with the same time complexity. As described in Sect. 3.4, the reduction aggregates every sequence of z blocks into a single block, and every sequence of z plaintext–ciphertext pairs into a single pair. The reduction implies that for any integer value of z, $f(z \cdot r, n^*/z, z \cdot m^*) \leq f(r, n^*, m^*)$. This, in turn, implies that given n and m, in order to find an optimal value of r, it is sufficient to consider all values of r which are multiples of some integer. In particular, it is sufficient to consider only values of r for which $m^* = rm$ is an integer (i.e., multiples of the denominator of m, assuming w.l.o.g. that m is rational).

Second, we note that if for some m^* and r, the number r appears in the magic sequence $Magic_{m^*}$, then for any $z \in \mathbb{N}$, we have $f(z \cdot r, n^*/z, z \cdot m^*) = f(r, n^*, m^*)$.

Indeed, by the structure of the sequence $Magic_{m^*}$, any of its elements can be written as $r = b_{j-1}m^* + ji$ for some i, j with $0 \leq i < r$ (where b_j denotes the j'th element of the sequence $Magic_1$, starting with $j = 0$, as above). Hence, if we consider $r' = zr$ for some $z \in \mathbb{N}$ and denote $n' = n^*/z$ and $m' = z \cdot m^*$, we have

$$r' = zr = b_{j-1}zm^* + jzi = b_{j-1}m' + j(zi),$$

for $0 \leq iz < r'$, which means that r' appears in the sequence $Magic_{m'}$. Then, as shown in Sect. 3.6, the improvement factor of the r'-round algorithm over exhaustive search is

$$2^{((j-1)m'+zi)n'} = 2^{((j-1)zm^*+zi)m^*/z} = 2^{((j-1)m^*+i)n^*} = 2^{((j-1)m+i/r)n},$$

which is indeed independent of z.

The arguments presented above imply that for $m = 1/b_j$ for some j, all choices $r = 1/m, 2/m, 3/m, \ldots$ are equivalent, and thus, the optimal gain is given by $r = 1/m$, $m^* = rm = 1$. In this case, the time complexity is $2^{(r-j)n^*} = 2^{n(1-jm)}$.

We would like to show now that for any rational[18] m which satisfies $1/b_j < m < 1/b_{j-1}$, the optimal time complexity parameter t is given by a linear extrapolation of the complexities at $m = 1/b_j$ and $m = 1/b_{j-1}$ (see Fig. 10). The proof consists of three simple propositions:

Proposition 1. *Let $m \in \mathbb{Q}_+$. There exists $r \in \mathbb{N}$ such that $rm \in \mathbb{N}$ and $r \in Magic_{rm}$.*

Proof. Let $m = p/q$ where $p, q \in \mathbb{N}$, and let $j \in \mathbb{N}$ be such that $b_{j-1} \leq 1/m < b_j$ (or equivalently, $b_{j-1} \leq 1/m < b_{j-1} + j$). We claim that $r = qj$ satisfies the condition of the proposition. First, $rm = (qj) \cdot (p/q) = pj \in \mathbb{N}$. Second, we have $r - b_{j-1}mr = qj - b_{j-1}pj = j(q - b_{j-1}p)$. Note that the sequence $Magic_{rm}$ contains elements of the form $b_{j-1}rm + ji$ for all $0 \leq i < rm$. Hence, if we show that $0 \leq i \triangleq q - b_{j-1}p < mr$, this would imply $r \in Magic_{rm}$.

This indeed holds, since by assumption, $b_{j-1} \leq q/p < b_{j-1} + j$, and thus, $b_{j-1}p \leq q < b_{j-1}p + jp$, which implies $0 \leq q - b_{j-1}p < jp = mr$, as asserted. $\square$

[18] There is clearly no loss of generality in assuming that m is rational as any number can be approximated by rational numbers up to any precision.

Proposition 2. *Let $m \in \mathbb{Q}_+$ and $r \in \mathbb{N}$ be such that $rm \in \mathbb{N}$ and $r \in Magic_{rm}$. Let $j \in \mathbb{N}$ be such that $b_{j-1} \leq 1/m < b_j$, and let $0 \leq s < 1$ be such that $1/m = sb_{j-1} + (1-s)b_j$. Then*

$$f(r, n/r, mr) = 2^{n(s(1-jm)+(1-s)(1-(j+1)m))}.$$

In other words, the exponent t is a linear extrapolation of the exponents corresponding to $m = 1/b_{j-1}$ and $m = 1/b_j$.

Proof. The claim follows immediately from the structure of the sequence $Magic_{rm}$. Indeed, since $b_{j-1} \leq 1/m < b_j$ and $r \in Magic_{rm}$, r is of the form $r = b_{j-1}rm + ij$ for some $0 \leq i < rm$. For each such i, the gain is $jrm + i$. Since by the assumption, $1/m = sb_{j-1} + (1-s)b_j$, we have $i = s/rm$. Hence, the gain which corresponds to $f(r, n/r, rm)$ is

$$jrm + (1-s)/rm = s \cdot jrm + (1-s) \cdot (j+1)rm.$$

The claim follows by substituting $f(r, n/r, mr) = 2^{n^* - rm - Gain(A)}$ where A is the corresponding dissection algorithm. □

Proposition 3. *Let $m \in \mathbb{Q}_+$ and $r \in \mathbb{N}$ be such that $rm \in \mathbb{N}$ and $r \notin Magic_{rm}$. Let $j \in \mathbb{N}$ be such that $b_{j-1} \leq 1/m < b_j$, and let $0 \leq s < 1$ be such that $1/m = sb_{j-1} + (1-s)b_j$. Then*

$$f(r, n/r, mr) > 2^{n(s(1-jm)+(1-s)(1-(j+1)m))}.$$

In other words, the exponent t is larger than in the linear extrapolation of the exponents corresponding to $m = 1/b_{j-1}$ and $m = 1/b_j$.

Proof. This claim also follows immediately from the structure of the sequence $Magic_{rm}$. Indeed, since $b_{j-1} \leq 1/m < b_j$ and $r \notin Magic_{rm}$, there exist $0 \leq i < rm$ and $1 \leq \ell < j$ such that r is of the form $r = b_{j-1}rm + ij + \ell$. Since by the assumption, $1/m = sb_{j-1} + (1-s)b_j$, we can write $i = s'/rm$ for some $s' > s$. By the structure of the sequence $Magic_{rm}$, we have

$$Gain(A) = jrm + (1-s')/rm < s \cdot jrm + (1-s) \cdot (j+1)rm$$

(since the gain at $r = b_{j-1}rm + i(s'/rm) + \ell$ is equal to the gain at $\tilde{r} = b_{j-1}rm + i(s'/rm)$). The claim follows by substituting $f(r, n/r, mr) = 2^{n^* - rm - Gain(A)}$. □

Combination of the three propositions above yields the complete tradeoff described in Fig. 10. Indeed:

- For each m such that $b_{j-1} \leq 1/m < b_j$, there exists r such that $r \in Magic_{rm}$ (by Proposition 1),
- The time complexity obtained for this r is optimal (by combination of Propositions 2 and 3), and

– The corresponding value of t is the linear extrapolation of the values at $m = 1/b_{j-1}$ and $m = 1/b_j$ (by Proposition 2).

Optimal Choice of *r* for Non-Deterministic Dissection Algorithms The arguments presented above hold with only slight changes for non-deterministic algorithms, with the $Magic$ sequences replaced by the corresponding $Magic^{\mathrm{ND}}$ sequences presented in Sect. 4. The resulting tradeoff curve is given, along with the tradeoff curves corresponding to deterministic dissection algorithms and to previous results, in Fig. 11.

5.2. *Using Dissection to Improve Rebound Attacks on Hash Functions*

Another application of our dissection technique can significantly reduce the memory complexity of rebound attacks [21,24] on hash functions. An important procedure in such attacks is to match input/output differences through an S-box layer (or a generalized S-box layer). More precisely, the adversary is given a list L_A of input differences and a list L_B of output differences, and has to find all the input/output difference pairs that can be combined through the S-box layer. A series of matching algorithms were presented by Naya-Plasencia [26] at CRYPTO 2011, optimizing and improving various rebound attacks.

Our dissection algorithms can be applied to this problem as well, replacing the *gradual matching* or *parallel matching* presented in [26]. As an example, we can improve the rebound attack on the SHA-3 candidate Luffa using a variant of our $Dissect_2(4, 1)$ algorithm.

In this rebound attack, the adversary is given 2^{67} possible input differences ($|L_A| = 2^{67}$) and $2^{65.6}$ output differences ($L_B = 2^{65.6}$) for 52 active 4-bit to 4-bit S-boxes. The adversary has to find an input difference in L_A that can become an output difference in L_B through the Luffa's S-box. We note that for this S-box, for any given input difference there are about 7 possible output differences (and vice versa). We refer the reader to [26] for the description of the previously best algorithm which takes time of 2^{104} and 2^{102} memory.

One can consider this problem to be a bicomposite problem and dissect it. The resulting attack algorithm follows the $Dissect_2(4, 1)$ algorithm. We start by (arbitrarily) dividing the 52 active S-boxes into four sets of 13 S-boxes each S_1, S_2, S_3, S_4. We then go over all the possible input differences $\delta \in L_A$ in the S-boxes of S_1, and for each such input difference, store all the possible *output* differences in S_2. We then go over all possible output differences $\delta' \in L_B$ in the S-boxes of S_1, and check for each of them whether the proposed difference in S_2 is in the table. If so, we check whether the output difference in S_1 is compatible with the input difference associated with the output difference in S_2 that was stored in the table. Each such match is further analyzed to determine whether the combination is feasible.

The resulting algorithm is depicted in Algorithm 10.

An analysis of this algorithm shows that its time complexity is 2^{104} operations, but its memory complexity is only 2^{52}. We note that the true memory complexity of this attack is actually 2^{66}, as one needs to store at least one out of L_A or L_B. Still, our results significantly improve those of [26]. These ideas were later used in [7].

```
Input: A list $L_A$ of input differences and a list $L_B$ of output differences
1: for all input differences $\delta \in L_A$ restricted to the S-boxes of $S_1$ do
2:   for all $7^{13}$ output differences which are possible in the S-boxes of $S_2$ do
3:     Store in the table the output differences along with the input difference in $S_1$.
4:   for all $7^{13}$ output differences that may be caused by $\delta$ in the S-boxes of $S_1$ do
5:     if there is a difference $\delta' \in L_B$ which agrees with $\delta$ in $S_1$ then
6:       Check that the difference of $\delta'$ in the S-boxes of $S_2$ is in the table
7:       if a match is found then
8:         Analyze the differences in the S-boxes of $S_3$ and $S_4$
9:         if $\delta$ and $\delta'$ match then
10:          return $\delta$ and $\delta'$
```

Algorithm 10: The $Dissect_2(4, 1)$ Rebound Attack

We note that this algorithm, besides improving other rebound attacks, can also be used when the problem is composed of layers which are relations (rather than functions or permutations) which allow multiple outputs for a single input.

6. Summary and Open Problems

In this paper, we introduced the new dissection technique which can be applied to a broad class of problems which have a bicomposite structure. We used this technique to obtain improved complexities for several well studied problems such as the cryptanalysis of multiple encryption schemes and the solution of hard knapsacks. The main open problem in this area is to either improve our techniques or to prove their optimality. In particular, we conjecture (but cannot prove) that any attack on multiple encryption schemes should have a time complexity which is at least the square root of the total number of possible keys. Another interesting problem for further research is whether the dissection technique can be combined with the list-merging algorithms presented in [26], in order to obtain the benefits of both approaches simultaneously.

Acknowledgements

The authors like to thank the anonymous reviewers of this paper for their useful comments and suggestions. The first author was supported in part by the Israeli Science Foundation through Grant No. 573/16. The second author was supported in part by the Israeli Science Foundation through Grant No. 827/12 and by the Commission of the European Communities through the Horizon 2020 program under Project Number 645622 PQCRYPTO. The third author was supported by the European Research Council under the ERC starting Grant Agreement Number 757731 (LightCrypt) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

References

[1] P. Austrin, P. Kaski, M. Koivisto, J. Määttä, Space-time tradeoffs for subset sum: an improved worst case algorithm, in F.V. Fomin, R. Freivalds, M.Z. Kwiatkowska, D. Peleg, (eds.) *ICALP (1)*. Lecture Notes in Computer Science, vol. 7965 (Springer, 2013), pp. 45–56

[2] C.H. Baek, J.H. Cheon, H. Hong, White-box AES implementation revisited. *J. Commun. Netw.* **18**(3), 273–287 (2016)

[3] A. Bar-On, O. Dunkelman, N. Keller, E. Ronen, A. Shamir, Improved key recovery attacks on AES with practical data and memory complexities, in *Accepted to CRYPTO 2018*, to appear in Lecture Notes in Computer Science (2018)

[4] A. Becker, J.S. Coron, A. Joux, Improved generic algorithms for hard knapsacks, in K.G. Paterson, (ed.) *EUROCRYPT*. Lecture Notes in Computer Science, vol. 6632 (Springer, 2011), pp. 364–385

[5] M. Bellare, R. Canetti, H. Krawczyk, Keying Hash functions for message authentication, in *Koblitz*, pp. 1–15

[6] E. Biham, Cryptanalysis of triple modes of operation. *J. Cryptol.* **12**(3), 161–184 (1999), https://doi.org/10.1007/s001459900050

[7] A. Canteaut, M. Naya-Plasencia, B. Vayssière, Sieve-in-the-middle: improved MITM attacks. In R. Canetti, J.A. Garay, (eds.) *Advances in Cryptology—CRYPTO 2013—33rd Annual Cryptology Conference, Santa Barbara, CA, USA*, August 18–22, 2013. Proceedings, Part I. Lecture Notes in Computer Science, vol. 8042 (Springer, 2013), pp. 222–240

[8] W. Diffie, M.E. Hellman, Special feature exhaustive cryptanalysis of the NBS data encryption standard. *IEEE Comput.* **10**(6), 74–84 (1977), https://doi.org/10.1109/C-M.1977.217750

[9] I. Dinur, O. Dunkelman, N. Keller, A. Shamir, Efficient dissection of composite problems, with applications to cryptanalysis, knapsacks, and combinatorial search problems, in R. Safavi-Naini, R. Canetti, (eds.) *CRYPTO*. Lecture Notes in Computer Science, vol. 7417 (Springer, 2012), pp. 719–740

[10] I. Dinur, O. Dunkelman, N. Keller, A. Shamir, Dissection: a new paradigm for solving bicomposite search problems. *Commun. ACM* **57**(10), 98–105 (2014), https://doi.org/10.1145/2661434

[11] I. Dinur, O. Dunkelman, N. Keller, A. Shamir, New attacks on Feistel structures with improved memory complexities, in *Gennaro and Robshaw*, pp. 433–454

[12] I. Dinur, O. Dunkelman, A. Shamir, Improved attacks on full gost, in A. Canteaut, (ed.) *FSE*. Lecture Notes in Computer Science, vol. 7549 (Springer, 2012), pp. 9–28

[13] S. Even, O. Goldreich, On the power of cascade ciphers, in D. Chaum, (ed.) *Advances in Cryptology, Proceedings of CRYPTO '83, Santa Barbara, California, USA*, August 21–24, 1983 (Plenum Press, New York, 1983), pp. 43–50

[14] M.E. Hellman, A cryptanalytic time-memory trade-off. *IEEE Trans. Inf. Theory* **26**(4), 401–406 (1980)

[15] N. Howgrave-Graham, A. Joux, New generic algorithms for hard knapsacks, In H. Gilbert, (ed.) *EUROCRYPT*. Lecture Notes in Computer Science, vol. 6110 (Springer, 2010), pp. 235–256

[16] T. Isobe, A single-key attack on the full GOST block cipher, in A. Joux, (ed.) *Fast Software Encryption—18th International Workshop, FSE 2011, Lyngby, Denmark*, February 13–16, 2011, *Revised Selected Papers*. Lecture Notes in Computer Science, vol. 6733 (Springer, 2011), pp. 290–305

[17] A. Joux, Multicollisions in iterated Hash functions. Application to cascaded constructions, in M.K. Franklin, (ed.) *CRYPTO*. Lecture Notes in Computer Science, vol. 3152 (Springer, 2004), pp. 306–316

[18] P.Kirchner, P. Fouque, Time-memory trade-off for lattice enumeration in a ball, in *IACR Cryptology ePrint Archive 2016* 222 (2016)

[19] D.E. Knuth, *The Art of Computer Programming, Volume II: Seminumerical Algorithms*, 2nd Edition. (Addison-Wesley, Reading, 1981)

[20] V. Lallemand, M. Naya-Plasencia, Cryptanalysis of Full Sprout, in *Gennaro and Robshaw*, pp. 663–682

[21] M. Lamberger, F. Mendel, M. Schläffer, C. Rechberger, V. Rijmen, The rebound attack and subspace distinguishers: application to whirlpool. *J. Cryptol.* **28**(2), 257–296 (2015)

[22] D. Lokshtanov, J. Nederlof, Saving space by algebraization, in Schulman, L.J. (ed.) *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, MA, USA*, 5–8 June 2010 (ACM, 2010), pp. 321–330, https://doi.org/10.1145/1806689.1806735

[23] S. Lucks, Attacking triple encryption, in S. Vaudenay, (ed.) *FSE*. Lecture Notes in Computer Science, vol. 1372 (Springer, 1998), pp. 239–253

[24] F. Mendel, C. Rechberger, M. Schläffer, S.S. Thomsen, The rebound attack: cryptanalysis of reduced Whirlpool and Grøstl, in O. Dunkelman, (ed.) *Fast Software Encryption, 16th International Workshop, FSE 2009, Leuven, Belgium*, February 22–25, 2009, *Revised Selected Papers*. Lecture Notes in Computer Science, vol. 5665 (Springer, 2009), pp. 260–276

[25] R.C. Merkle, M.E. Hellman, On the security of multiple encryption. *Commun. ACM* **24**(7), 465–467 (1981)

[26] M. Naya-Plasencia, How to improve rebound attacks, in P. Rogaway, (ed.) *Advances in Cryptology—CRYPTO 2011—31st Annual Cryptology Conference, Santa Barbara, CA, USA*, August 14–18, 2011. *Proceedings*. Lecture Notes in Computer Science, vol. 6841 (Springer, 2011), pp. 188–205

[27] P.C. van Oorschot, M.J. Wiener, Improving implementable meet-in-the-middle attacks by orders of magnitude, in *Koblitz*, pp. 229–236

[28] R. Schroeppel, A. Shamir, $\mathrm{AT} = \mathrm{O}(2^{\mathrm{n}/2})$, $\mathrm{S}=\mathrm{O}(2^{\mathrm{n}/4})$ algorithm for certain NP-complete problems. *SIAM J. Comput.* **10**(3), 456–464 (1981)

[29] J.R. Wang, Space-efficient randomized algorithms for K-SUM, in A.S. Schulz, D. Wagner, (eds.) *Algorithms—ESA 2014—22th Annual European Symposium, Wroclaw, Poland*, September 8–10, 2014. *Proceedings*. Lecture Notes in Computer Science, vol. 8737 (Springer, 2014), pp. 810–829