Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems

Itai Dinur[1], Orr Dunkelman[1,2],
Nathan Keller[1,3], and Adi Shamir[1]

[1] Computer Science department, The Weizmann Institute, Rehovot, Israel
[2] Computer Science Department, University of Haifa, Israel
[3] Department of Mathematics, Bar-Ilan University, Israel

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 attack hash functions with a rebound attack, to solve hard knapsack problems, and to find the shortest solution to a generalized version of Rubik's cube with better time complexities (for small memory complexities) than the best previously known algorithms.

Keywords: Cryptanalysis, TM-tradeoff, multi-encryption, knapsacks, bicomposite, dissection, rebound

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 [9] in 2004 that finding collisions in hash functions defined by the *parallel execution* of several independent iterated 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, 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. 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 $r \times r$ matrix whose columns corresponds to the processing of the various plaintexts and whose rows correspond to the application of the various block ciphers. 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 cannot be independently verified by an efficient computation.

The security of multiple-encryption schemes had been analyzed for more than 30 years, but most of the published papers had dealt with either double or triple encryption (which is widely used as a DES-extension in the banking industry). 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.

Standard meet-in-the-middle (MITM) attacks, 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} time and 2^n 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.

To demonstrate the generality of our techniques, we show in this paper how to apply them to several types of combinatorial search problems (some of which have nothing to do with cryptography), such as 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. The best known special purpose algorithm for this problem was published at Eurocrypt 2011 by Becker et al. [1], but 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 since 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 [1] to $2^{4n/7} = 2^{0.5714n}$ with our new algorithm.

The algorithm of Becker et al. [1] crucially depends on the fact that addition is an associative and commutative operation on numbers, and that 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 [11] 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 i-th row represents the action of the i-th permutation in the product, and the j-th 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 permutation S at the bottom. We can reduce this matrix to size $r \times r$ for any desired 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 algorithms of Schroeppel and Shamir [17] and of Becker et al. [1] which analyzed the knapsack problem, the algorithm of van Oorschot and Wiener [18] which attacked double and triple encryption, and the results of Dinur et al. [3] in the specific case of the block cipher GOST. 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 [3, 17, 18] matched this curve only for a single point, and the recent algorithm of Becker et al. [1] 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 $M \leq N^{1/4}$.

The paper is organized as follows: In Section 3 we introduce the dissection technique and present our best error-free attacks on multiple encryption. In Section 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 [18] to get an improved time-memory tradeoff curve. Finally, in Section 5 we apply our results to various problems, including knapsacks, rebound attacks on hash functions and search problems in databases.

2 Notations and Conventions

In this paper 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 cumbersome). We denote by E^i the encryption process with key k_i, and denote by $E^{[1\cdots r]}$ 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 under $E^{[1\cdots i]}$ is denoted by X^i, and the decryption process of $E^{[1\cdots r]}$ is denoted by $D^{[1\cdots r]}$ (which applies the keys in the reverse order). To attack $E^{[1\cdots r]}$, 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 $E^{[i\cdots j]}$, 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.

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 need random bits and they always succeed since they implicitly scan all possible solutions.

Here, we treat the case of general r and $m = 1$. The generalization of the algorithms to other integer values of m is given in the extended version of this paper [4]. The algorithms can be extended also to fractional values of m and to

compositions of one-way functions, which appear in the context of layered Message Authentication Codes, such as NMAC [2].[1] The first non-integer extension is presented in the full version of the paper, whereas the extension to one-way functions is presented in the extended version of this paper [4].

3.1 Previous Work — The Meet in the Middle Attack

The trivial algorithm for recovering the key of r-encryption 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 [14] Merkle and Hellman observed that if the keys used in the encryption are independent, an adversary can trade time and memory complexities, using a meet in the middle 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 table (along with the respective key candidate). Then, for each value of the last $r-u$ keys, the adversary computes the vector $D^{[u+1 \ldots r]}(C_1, C_2, \ldots, C_r)$ and checks whether the value appears in the table (each such collision suggests a key candidate $(k_1, \ldots, k_r)$). 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 $M \leq 2^{\lfloor r/2 \rfloor n}$.[2] Note that the algorithm can be applied also if the number of available plaintext/ciphertext pairs is $r' < r$. In such 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 meet in the middle attack, designed for breaking double-encryption, is still the best known generic attack on double encryption schemes. It is also the best known attack for triple encryption upto logarithmic factors,[3] which was studied very extensively due to its relevance to the former de-facto encryption standard Triple-DES.

---

[1] Given a keyed hash-function F, and a key $k = (k_1, k_2)$, the MAC function $NMAC(x)$ which works on inputs x of arbitrary length, is defined as $NMAC_k(x) = F_{k_1}(F_{k_2}(x))$.

[2] 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 $TM = 2^{rn}$.

[3] We note that a logarithmic time complexity improvement can be achieved in these settings as suggested by Lucks [12]. 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 hand in hand with 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).

3.2 The Basic Dissection Algorithm: Attacking 4-Encryption

In the followings we show that for $r \geq 4$, the basic meet in the middle 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 meet in the middle 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 $Dissect_2(4, 1)$ for reasons which will become apparent later, is as follows:

1. Given four known plaintexts (P_1, P_2, P_3, P_4) and their corresponding ciphertexts (C_1, C_2, C_3, C_4), for each candidate value of $X_1^2 = E_{k_2}(E_{k_1}(P_1))$:
2. (a) Run the standard meet in the middle attack on 2-round encryption with (P_1, X_1^2) as a single plaintext-ciphertext pair. For each of the 2^n values of (k_1, k_2) output by the attack, partially encrypt P_2 using (k_1, k_2), and store in a table the corresponding values of X_2^2, along with the values of (k_1, k_2).
   (b) Run the standard meet in the middle attack on 2-round encryption with (X_1^2, C_1) as a single plaintext-ciphertext pair. For each of the 2^n values of (k_3, k_4), partially decrypt C_2 using (k_3, k_4) and check whether the suggested value for X_2^2 appears in the table. If so, check whether the key (k_1, k_2, k_3, k_4) suggested by the table and the current (k_3, k_4) candidate encrypts P_3 and P_4 into C_3 and C_4, respectively.

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(a) and the right values of (k_3, k_4) are found in Step 2(b), and thus, the right value of the key is necessarily found. The time complexity of the algorithm is 2^{2n}. Indeed, Steps 2(a) and 2(b) are called 2^n times (for each value of X_1^2), and each of them runs the basic meet in the middle attack on 2-encryption in expected time and memory of 2^n. The number of expected collisions in the table of X_2^2 is 2^n. Thus, the expected time complexity of the attack[4] is $2^n \cdot 2^n = 2^{2n}$.

The memory consumption of the 2-encryption meet in the middle steps is expected to be about 2^n. The size of the table "passed" between Steps 2(a) and 2(b) is also 2^n, since each meet in the middle step is expected to output 2^n key candidates. Hence, the expected memory complexity of the entire algorithm is 2^n.

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 to increase the gain over the standard meet in the middle attack significantly for larger values of r.

[4] We remind the reader that we disregard factors which are polynomial in n and 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 $(k_{r'+1}, \ldots, k_r)$, and applying the basic algorithm to the remaining $E^{[1 \ldots r']}$. 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. *Given an algorithm A for r-encryption whose time and memory complexities are T and M, respectively, we define $Gain(A) = \log(2^{rn}/TM)/n = r - \log(TM)/n$. The maximal gain amongst all deterministic algorithms for r-encryption which use 2^{mn} memory, is denoted by $Gain_D(r, m)$.*

By the trivial argument above, $Gain_D(r, 1)$ is monotone non-decreasing with r. The $Dissect_2(4, 1)$ algorithm shows that $Gain_D(r, 1) \geq 1$ 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 $i = 1, 2, \ldots, r/2 - 1$ and runs simple meet in the middle attacks on each subcipher $E^{[2i+1 \ldots 2i+2]}$ 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 $i = 1, 2, \ldots, r/4-1$, and runs meet in the middle attacks on each subcipher $E^{[4i+1 \ldots 4i+4]}$ 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.

The memory complexity of $LogLayer_r$ is 2^n (as we need to store a number linear in r of lists of 2^n keys each). As in the j-th layer of the attack, $(r/2^j) - 1$ intermediate values are enumerated, and as each basic meet in the middle attack has time complexity of 2^n, the overall time complexity of the attack is

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

The gain is thus $Gain(LogLayer_r) = \log r - 1$, which shows that $Gain_D(r, 1) \geq \lfloor \log r \rfloor - 1$.

The $Square_r$ Algorithm: This logarithmic gain of *LogLayer* can be significantly outperformed by the $Square_r$ algorithm, applicable when $r = (r')^2$ is a perfect square. The $Square_r$ algorithm starts by trying all the possible values of $r' - 1$ intermediate values every r' rounds, a total of $(r' - 1)^2$ intermediate encryption values. Namely, the algorithm starts by enumerating all $X_1^{r'}, X_2^{r'}, \ldots, X_{r'-1}^{r'}, X_1^{2r'}, X_2^{2r'}, \ldots, X_{r'-1}^{2r'}, \ldots, X_1^{r'(r'-1)}, X_2^{r'(r'-1)}, \ldots, X_{r'-1}^{r'(r'-1)}$. 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. 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.

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 is kept at 2^n. Therefore, $Gain(Square_r) \geq \sqrt{r} - 1$, which shows that $Gain_D(r, 1) \geq \lfloor \sqrt{r} \rfloor - 1$.

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 known attacks of this kind yields a gain of $O(\sqrt{r'}) = O(r^{1/4})$, the addition to the overall gain of $Square_r$ is negligible.

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, and present the optimal dissection algorithms of this type for $m = 1$ and any number r of encryptions.

We observe that the basic dissection attack is asymmetric in its nature. Indeed, after the two separate meet in the middle 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 is bigger than the upper part, may result in a better algorithm. This is indeed the case, as illustrated by the following $Dissect_3(7, 1)$ algorithm:

1. Given 7 plaintext-ciphertext pairs $(P_1, C_1), (P_2, C_2), \ldots (P_7, C_7)$, for each possible value of X_1^3, X_2^3, perform:
   (a) 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, and obtain 2^n candidates for the keys (k_1, k_2, k_3). For each such candidate, partially encrypt the rest of the plaintexts using (k_1, k_2, k_3) and store the values $(X_4^3, \ldots, X_7^3)$ in a table, along with the corresponding key candidate (k_1, k_2, k_3).
   (b) 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. Note that since only two pairs are given, algorithm $Dissect_2(4, 1)$ produces 2^{2n} possible values of the keys (k_4, k_5, k_6, k_7). However, these values are produced sequentially, and can be checked on-the-fly by partially decrypting C_4, C_5, C_6, C_7, and checking whether the corresponding vector $(X_4^3, \ldots, X_7^3)$ appears in the table.

The memory complexity of the algorithm is 2^n, as both the basic meet in the middle attack on triple encryption and the algorithm $Dissect_2(4, 1)$ require 2^n memory, and the size of the table "passed" between Steps 2(a) and 2(b) is also 2^n.

The time complexity is 2^{4n}. Indeed, two n-bit values are enumerated in the middle, both the basic meet in the middle 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) are checked instantly. This leads to time complexity of $2^{2n} \cdot 2^{2n} = 2^{4n}$.

This shows that $Gain(Dissect_3(7,1)) = 2$, which is clearly better than the algorithms $LogLayer_r$ and $Square_r$, which reach gain of 2 for the first time only at $r = 8$ and at $r = 9$, respectively.

Furthermore, the algorithm $Dissect_3(7,1)$ can be extended recursively to larger values of r, to yield better asymptotic for the gain function. Given the algorithm $Dissect_j(r',1)$ such that $Gain(Dissect_j(r',1)) = \ell - 1$, we define the algorithm $Dissect^1_{NEXT} = Dissect_{\ell+1}(r'+\ell+1,1)$ for r-encryption, where $r = r' + \ell + 1$, as follows:

1. Given r plaintext-ciphertext pairs $(P_1, C_1), (P_2, C_2), \ldots (P_r, C_r)$, for each possible value of $X_1^{\ell+1}, \ldots, X_\ell^{\ell+1}$, perform:
   (a) Apply the basic MITM attack to $E^{[1\ldots\ell+1]}$ with $(P_1, X_1^{\ell+1}), \ldots, (P_\ell, X_\ell^{\ell+1})$ as the plaintext/ciphertext pairs, and obtain 2^n candidates for the keys $(k_1, \ldots, k_{\ell+1})$. For each such candidate, partially encrypt the rest of the plaintexts using $(k_1, \ldots, k_{\ell+1})$ and store the values $(X_{\ell+1}^{\ell+1}, \ldots, X_r^{\ell+1})$ in a table, along with the corresponding key candidate $(k_1, \ldots, k_{\ell+1})$.
   (b) Apply $Dissect_j(r',1)$ to $E^{[\ell+2\ldots r]}$ with $(X_1^{\ell+1}, C_1), \ldots, (X_\ell^{\ell+1}, C_\ell)$ as the plaintext/ciphertext pairs. Check each of the $2^{(r'-\ell)n}$ suggestions for the keys $(k_{\ell+2}, \ldots, k_r)$ on-the-fly by partially decrypting $C_{\ell+1}, \ldots, C_r$, and checking whether the corresponding vector $(X_{\ell+1}^{\ell+1}, \ldots, X_r^{\ell+1})$ appears in the table.

An exactly similar argument as the one used for $Dissect_3(7,1)$ shows that the time and memory complexities of $Dissect_{\ell+1}(r)$ are $2^{r'n}$ and 2^n, respectively, which implies that $Gain(Dissect_{\ell+1}(r)) = \ell$. 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. If we denote the number r of "rounds" in the ℓ's element of the sequence (i.e., the element for which the gain equals to ℓ) by r_ℓ, then by the construction, the sequence satisfies the recursion

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

which (together with $r_2 = 4$ which follows from $Dissect_2(4,1)$) leads to the formula:

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

The asymptotic gain of this sequence is obtained by representing ℓ as a function of r, and is equal to $(\sqrt{8r-7}-1)/2 \approx \sqrt{2r}$, which is bigger than the $\sqrt{r}$ gain of the $Square_r$ algorithm.

A thorough analysis, presented in the extended version of this paper [4], shows that the algorithms obtained by the recursive sequence described above are the optimal amongst all dissection algorithms that split the r rounds into two (not necessarily equal) parts, and attacks each part recursively, using an optimal 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 $\ell = 0, 1, 2, \ldots$, is:

$$Magic_1 = \{1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, \ldots\}.$$

This "magic sequence" will appear several more times in the sequel.

4 Parallel Collision Search via Dissection

In Section 3, we considered the scenario of deterministic algorithms which never err for r-encryption, 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. In this scenario, the best previously known generic attack is the meet in the middle attack, which obtains the tradeoff curve $TM = 2^{rn}$, where T and M are the time and memory complexities of the algorithm, respectively. In this model, we presented several dissection algorithms which allow to achieve the curve $TM = 2^{(r-\sqrt{2r})n}$.

In this section, we consider the scenario in which non-deterministic algorithms, which find the right keys with some probability $p < 1$, are allowed. 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 [18]. We now show how to combine the dissection algorithms presented in Section 3 with the parallel collision search algorithm to obtain an even better tradeoff curve with a multiplicative gain of $2^{(\sqrt{2r}/8)n}$ over the curve of [18].

4.1 Brief Description of the Parallel Collision Search Algorithm

We start with a very brief description of the PCS algorithm suggested in [18].

The algorithm consists of two steps:

1. Find *partial collisions*, which are key suggestions which comply with half of the plaintext/ciphertext pairs.
2. For each partial collision, check whether it complies with the second half of the plaintext/ciphertext pairs.

The first step is performed by constructing two step functions:[5]

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

[5] The idea of constructing two step functions was first proposed in [8].

which can be computed easily given the pairs $(P_1, C_1), \ldots, (P_{r/2}, C_{r/2})$, and using Floyd's cycle finding algorithm [10] (or another cycle finding algorithm which requires little memory, such as [16]) to find a collision between them. In the case of constant memory, Floyd's algorithm finds such a collision (which produces a key suggestion which complies with the pairs $(P_1, C_1), \ldots, (P_{r/2}, C_{r/2})$) in $2^{(r/4)n}$ time. If $M = 2^{mn}$ memory is given, this step can be speeded up by incorporating Hellman's time-memory tradeoff techniques [5], that allow to find 2^{mn} collisions simultaneously in $2^{(r/4+m/2)n}$ time. In both cases, after $2^{(r/2)n}$ partial collisions are found, it is expected that one of them passes the condition of the second step, which means that it is the desired key suggestion. The time complexity of the algorithm 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, that is, $(X_1^{r/2}, \ldots, X_u^{r/2})$, and construct step functions $\tilde{F}^{upper}$ and $\tilde{F}^{lower}$ 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 the smaller "cost" of finding a collision which complies only with $(P_{u+1}, C_{u+1}), \ldots, (P_{r/2}, C_{r/2})$. It should be noted that this gain could be diminished by the "cost" of the new step function $\tilde{F}$, that is higher than the "cost" of the simpler step function F. However, we show that if the efficient dissection algorithms presented in Section 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 (for 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 PCS requires for finding each such collision is $2^{1.5n}$. The DC algorithm allows to achieve the same goal in 2^n 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 $Dissect_2(4, 1)$ presented in Section 3 allows to retrieve 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).

The algorithm works as follows:

1. Given the plaintexts (P_1, P_2, P_3, P_4) and their corresponding ciphertexts (C_1, C_2, C_3, C_4), for each guess of (X_1^4, X_2^4, X_3^4):
2. (a) Define the step functions $\tilde{F}^{upper}$ and $\tilde{F}^{lower}$ by:

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

   In order to compute the step function $\tilde{F}^{upper}$, apply $Dissect_2(4,1)$ 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 $\tilde{F}^{upper}(X_2^1) = X_4^4$. The function $\tilde{F}^{lower}$ is computed similarly.

   (b) Find a collision between the functions $\tilde{F}^{upper}$ and $\tilde{F}^{lower}$ using a variant of Floyd's cycle finding algorithm which exploits the $M = 2^n$ available amount of memory.

   (c) Check whether the keys $(k_1, \ldots, k_4, k_5, \ldots, k_8)$ suggested by the partial collision, encrypt $(P_5, \ldots, P_8)$ to $(C_5, \ldots, C_8)$. If not, return to Step 2(a). After 2^n partial collisions are examined and discarded, return to Step 1, and pick a different guess for (X_1^4, X_2^4, X_3^4).

By the properties of the algorithm $Dissect_2(4,1)$ mentioned above, each step of the functions $\tilde{F}$ can be performed in 2^n time and memory. 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 $\tilde{F}^{upper}$ and $\tilde{F}^{lower}$ yields a suggestion of $(k_1, \ldots, k_4, k_5, \ldots, k_8)$ which complies with the pairs $(P_1, C_1), \ldots, (P_4, C_4)$. Finally, since the step functions are from n bits to n bits, collision between them can be found instantly given 2^n memory. Therefore, the time required for finding a partial collision is 2^n, and thus, the total running time of the algorithm is $2^{4n} \cdot 2^n = 2^{5n}$. 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 Section 3. However, as we will show in the sequel, for larger values of r, the DC algorithms outperform the $Dissect$ algorithms significantly.

The general algorithms $DC(r,m)$ Now we are ready to give a formal definition of the class $DC(r,m)$ of algorithms, applicable to r-encryption (for an even r)[6], given memory of 2^{mn}. An algorithm $A \in DC(r,m)$ is specified

[6] 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 $E^{[1\ldots r-1]}$.

by a number u, $1 \leq u \leq r/2$, 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 $|I^{upper}| = |I^{lower}| = r/2 - u$.

In the algorithm, the adversary fixes u intermediate values $(X_1^{r/2}, \ldots, X_u^{r/2})$. Then, she defines the step functions $\tilde{F}^{upper}$ and $\tilde{F}^{lower}$ by:

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

The step function $\tilde{F}^{upper}$ is computed by applying a dissection attack to $E^{[1\ldots r/2]}$ with the plaintext/ciphertext pairs $(P_1, X_1^{r/2}), \ldots, (P_u, X_u^{r/2})$ and the intermediate values contained in I^{upper} to retrieve a unique value of the keys $(k_1, \ldots, k_{r/2})$, and then partially encrypting $(P_{u+1}, \ldots, P_r)$ to obtain $(X_{u+1}^{r/2}, \ldots, X_{r/2}^{r/2})$. The step function $\tilde{F}^{lower}$ is computed in a similar way, with respect to $E^{[r/2+1\ldots r]}$ and the set I^{lower}. Then, a variant of Floyd's cycle finding algorithm which exploits the 2^{mn} amount of available memory is used to find a collision between $\tilde{F}^{upper}$ and $\tilde{F}^{lower}$, which yields a suggestion of $(k_1, \ldots, k_{r/2}, k_{r/2+1}, \ldots, k_r)$ which complies with the plaintext/ciphertext pairs $(P_1, C_1), \ldots, (P_{r/2}, C_{r/2})$.

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:

$$2^{(r/2)n} \cdot 2^{((r/2-u-m)/2)n} \cdot 2^{sn} = 2^{((3/4)r-(u+m-2s)/2)n}. \tag{1}$$

As the time complexity of the PCS algorithm with memory 2^{mn} is $2^{((3/4)r-m/2)n}$, the multiplicative gain 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 advantage is indeed $2^{(3/2-1)n} = 2^{n/2}$ (i.e., the gain is $1/2$), as mentioned above. 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 $I^{upper}(A), I^{lower}(A), u(A)$, and $s(A)$, respectively.

We conclude this section by mentioning a difficulty in the implementation of the DC algorithm. Unlike the PCS algorithm where the output of the step functions F is always uniquely defined, in DC the functions $\tilde{F}$ return no output for some of the inputs. 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 contained in I^{upper}, is distributed according to the distribution $Poisson(1)$, and in particular, equals to zero for an $1/e$ fraction of the inputs. This difficulty can be resolved by introducing *flavors* into the step function $\tilde{F}$, which alter the function in a deterministic way when it fails to produce output. The exact modification is described in the extended version of this paper [4].

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 Section 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 $2^{(\lfloor\sqrt{2r}\rfloor/8)n}$ for any $r \geq 8$.

Before we present the extensions of the basic DC algorithm, we would like to define formally 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*

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

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

Note that it follows from Equation (1) that if $A \in DC(r, m)$, then

$$Gain_{ND}(A) = u(A)/2 - s(A). \tag{2}$$

Monotonicity of the gain The most basic extension of the basic DC algorithm is to preserve 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.*

Due to space restrictions, the proof of the lemma is presented in the extended version of this paper [4]. Here we only note that the algorithm B is constructed from A by choosing $I^{upper}(B) = I^{upper}(A) \cup \{X_1^{r'/2}\}$, and similarly for $I^{lower}(B)$.

Lemma 1 implies that the gain of the DC algorithms is monotone non-decreasing with r, and in particular, that $Gain_{ND}(r, 1) \geq 1/2$, 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 Section 3.3. Recall that the $LogLayer_r$ algorithm, applicable when r is a power of 2, consists of guessing the set of intermediate values:

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

and applying a recursive sequence of meet in the middle attacks on 2-encryption. Using this algorithm, we can define the algorithm $LL_r \in DC(2r, 1)$, by specifying

$I^{upper}(LL_r) = I_0$, and $I^{lower}(LL_r)$ 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 $(k_1, \ldots, k_r)$ in time and memory of 2^n. Hence, we have $s(LL_r) = 1$. By Equation (2), it follows that $Gain(LL_r) = (\log r + 1)/2 - 1 = (\log r - 1)/2$.

The basic algorithm for 8-encryption is the special case LL_4 of this algorithm. 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 $Square_r$ algorithm: The logarithmic asymptotic gain of the LL sequence can be significantly outperformed by an analogue of the $Square_r$ algorithm, presented in Section 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:

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

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 $Sq_r \in DC(2r, 1)$, by specifying $I^{upper}(Sq_r) = I_0$, and $I^{lower}(Sq_r)$ in a similar way. Since $|I_0| = (r' - 1)^2$, 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 Equation (2),

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

where $2^{f_1(r)n}$ is the time complexity of the best possible 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 Section 3.4, the next elements of the sequence Sq_r which increase the gain, correspond to the next elements of the sequence $Magic_1 = \{1, 2, 4, 7, 11, 16, \ldots\}$ described in Section 3.4. They lead to gains of $1.5, 2.5$, and 3.5 for $r = 32, 98$, and $r = 242$, 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, 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 $A \in DC(r, 1)$ whose step function is computed by a two-layer algorithm: First, $E^{[1\ldots r/2]}$ is divided into r_2 subciphers of r_1 rounds each, and the 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 the algorithm A_2 is used to attack the resulting r_2-encryption. The set $I^{upper}(A)$ is chosen such that both A_1 and A_2 algorithms perform in time 2^s. Formally, if we denote $u_1 = |I_1^{upper}|$, then the set $I^{upper}(A)$ 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 $I^{lower}(A)$ is defined similarly. Hence,

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

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

Note that the algorithm Sq_r is actually a two-layer DC algorithm, with $r_1 = r_2 = r'$ and $I_1^{upper} = I_2^{upper} = \emptyset$. It turns out that for all $8 \le r \le 128$, the maximal gains are obtained by two-layer DC algorithms where r_1, r_2 are chosen from the sequence $Magic_1$ presented in Section 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 to the full version of the paper, and conclude that the minimal numbers of rounds for which the gain equals $0.5, 1, 1.5, \ldots$ are:

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

Finally, we note 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 $Gain_{ND}(r, m) = 0.5, 1, 1.5, 2, \ldots$ are:

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

The full analysis of the case $m > 1$ will appear in the full version of the paper.

5 Applications

In this section, we apply our new dissection algorithms to several well known bicomposite search problems. As described in the introduction, we can represent such a problem as an $r \times r$ execution matrix which is treated as a multiple-encryption scheme with r rounds. In the case of knapsacks, we are allowed to choose any constant value of r when n grows to infinity in order to optimize the value of t for any given m. In other cases, r is restricted to a specific value or to a set of values. For example, in the special case of Rubik's cube, we know that a 20-move solution exists for any reachable state, and thus it does not make sense to choose $r > 20$. Such constraints can limit the choice of parameters for our algorithms, and thus we may not be able to exploit the available memory as efficiently as in the case of knapsacks.

Since the analysis of our multiple encryption algorithms assumes the randomness of the underlying block ciphers, we have to justify this assumption for each reduction we consider. For example, in the case of knapsacks with n generators, strong randomness assumptions are common practice whenever n is sufficiently large (e.g., see [1, 7]), and we can use the same assumptions when we consider subproblems with n/r generators for any constant r.

5.1 Applications to Knapsacks

The knapsack problem 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 [17], which requires $2^{n/2}$ time and $2^{n/4}$ memory. Surprisingly, in 2010, Howgrave-Graham and Joux [7] showed how to solve the knapsack problem in time much better than $2^{n/2}$, by using the associativity and commutativity properties of addition. This result was further improved by Becker, Coron and Joux in [1]. In addition to their basic results, Howgrave-Graham and Joux [7] also described reduced-memory algorithms, and in particular [1] described a memoryless attack which requires only $2^{0.72n}$ time. All these new attacks are heuristic in a 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 [1] also considered deterministic algorithms that never err, and described a straight-line time-memory tradeoff curve, but this curve was only valid in the range $1/16 \leq m \leq 1/4$.

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 [1] over the whole range of $1/16 < m < 1/4$. In addition, we can expand our tradeoff curve in a continuous way for any smaller value of $m \leq 1/4$. By combining our generic deterministic and non-deterministic algorithms, we can get a new curve which is better than

the best knapsack-specific algorithms described in [7] and [1] for the large interval of (approximately) $1/100 \leq m < 1/6$.[7]

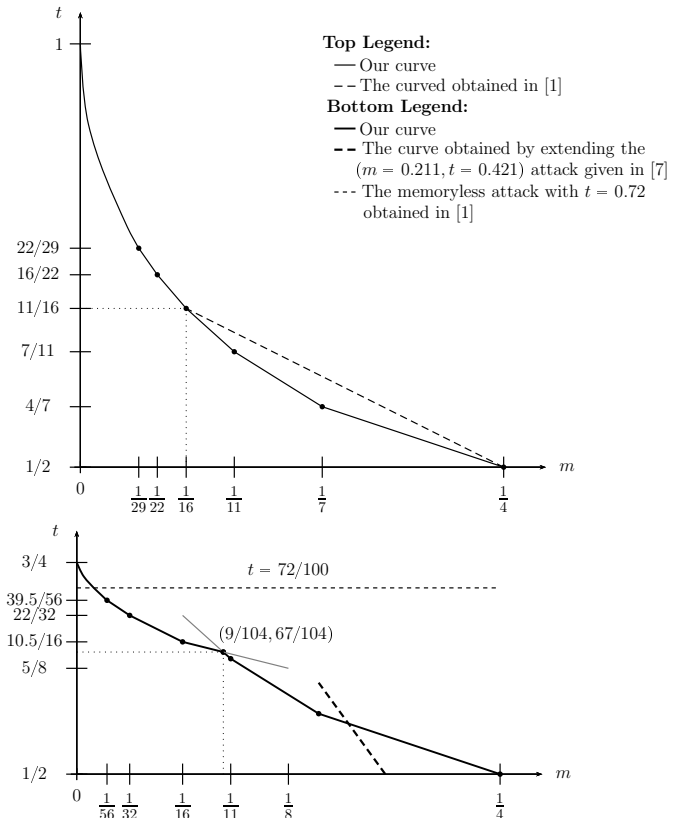
The formal reduction of the knapsack problem to r-round encryption (for any r) is given in the extended version of this paper [4], but it is not required in order to understand the rest of this paper. Given $M = 2^{mn}$ memory, our goal is to solve the knapsack problem by applying the reduction with a value of r which optimizes the time complexity of our multiple encryption algorithm. Formally, for any r, we apply the multiple encryption algorithm with an effective block size reduced by a factor of r, i.e., $n^* = n/r$. By equating $M = 2^{mn}$ with $M = 2^{n(m^*r)}$, we can see that the effective memory unit increases by the same ratio, i.e, $m^* = mr$. We denote by $f(r, n^*, m^*)$ the running time of our multiple encryption algorithm on an r-round block cipher with a block size of n^* bits and $M^* = 2^{m^*n^*}$ available memory, given r plaintext-ciphertext pairs. Using this notation, we would like to find 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 [7] and [1] for $1/16 \leq m \leq 1/4$ implicitly perform a reduction to multiple encryption with the fixed parameters $r = 4$ and $r = 16$. In fact, for the case of knapsacks and these choices of r, these algorithms are closely related to our square algorithms (described in Section 3.3). However, as we now show, we can get a better tradeoff curve by using other choices of r.

Time-Memory Tradeoff Curves for Knapsacks Using our multiple encryption algorithms, we construct time-memory tradeoff curves for knapsacks: we start with deterministic algorithms and consider first the case of $1/m \in \{1, 2, 4, 7, 11, 16, ...\}$, which is the "magic sequence" constructed in Section 3.4. In order to simplify our notation, we denote the j'th element of this sequence by b_j, starting from $j = 0$. In the case of $1/m = b_j$ for $j \geq 2$, we simply choose $r = 1/m = b_j$ and run the algorithm with $n^* = n/m$ and $m^* = m/m = 1$. For example, in case $m = 1/4$, we run the 4-round multiple encryption with $m^* = 1$, for which the time complexity is $T = 2^{2n^*} = 2^{2(n/4)} = 2^{n/2}$, or $t = 1/2$. In case $m = 1/7$, we run the 7-round dissection algorithm with $m^* = 1$, for which the time complexity is $T = 2^{4n^*} = 2^{4n/7}$, or $t = 4/7$. In the extended version of this paper [4], we show that in the case of $1/m \in \{4, 7, 11, 16, ...\}$, our choice of r is indeed optimal. Thus, we obtain a sequence of optimal points on the deterministic time-memory tradeoff curve. In order to obtain an optimal continuous curve for $0 < m \leq 1/4$, we need to use our algorithms for integral $m^* \geq 2$. As described in the extended version of this paper [4], these algorithms enable us to connect in a straight line any two consecutive time-memory tradeoff points for $1/m \in \{4, 7, 11, 16, ...\}$, and obtain a continuous curve (as shown on the left diagram of Figure 1).

[7] We note that since our algorithms do not efficiently exploit more than $2^{n/4}$ memory, our tradeoff curves are only defined for $m \leq 0.25$. This should be contrasted with the non-deterministic algorithms of [7] and [1], whose main results solve the problem in time less than $2^{n/3}$ with about $2^{n/3}$ memory.

For non-deterministic algorithms, we use the same approach, and consider first the "magic sequence" constructed in Section 4 for $1/m \in \{16, 32, 56, ...\}$. We choose $r = 1/m$ and the corresponding values of t (1/10.5, 1/22, 1/39.5,...). Similarly to the deterministic case, we can use our non-deterministic algorithms for integral $m^* \geq 2$ in order to obtain a continuous curve (as shown on the right diagram of Figure 1). The full details of how to connect consecutive points on the curve will be given in the full version of this paper.



On the top: A comparison between time-memory tradeoff curves obtained with deterministic algorithms. Our curve (defined for $m \leq 1/4$) is strictly better than the curve obtained in [1] (defined only for $1/16 \leq m \leq 1/4$) for any $1/16 < m < 1/4$.
On the bottom: A comparison between general time-memory tradeoff curves. Our general time-memory tradeoff curve is better than the attacks of [1] and [7] in the interval of (approximately) $1/100 \leq m < 1/6$.

Fig. 1. Time-Memory Tradeoff Curves for Knapsack

5.2 Improving Rebound Attacks On Hash Functions

Another application of our new techniques can significantly improve rebound attacks [13] 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 happen through the S-box layer. A series of matching algorithms were recently developed, optimizing and improving various rebound attacks [15].

Our dissection algorithms can be applied for this problem as well, replacing the gradual matching or parallel matching presented at Crypto 2011 by [15]. For example, we can improve the rebound attack on Luffa using a variant of our $Dissect_2(4,1)$ algorithm. As described in the extended version of this paper [4], we can reduce the memory complexity of the matching algorithm from 2^{102} to only 2^{66} without affecting the time complexity of the attack (which remains at 2^{104}).

5.3 Applications to Relational Databases

As a final example of the versatility of our algorithm, we note that in some cases, it may be possible to use the dissection technique to speed up the processing of queries in relational databases. The problem of composing block ciphers can be viewed as the problem of computing the *join* of several databases, where each database contains all the possible plaintext/ciphertext pairs, and the join operation equates the previous ciphertext with the next plaintext. When intermediate joined databases blow up in size but the final database is quite small, it may be better to use the dissection technique which guesses some middle values and splits the computation into smaller independent parts. More details about this potential application will be given in the full version of this paper.

6 Summary

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 can not 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.

References

1. Anja Becker, Jean-Sébastien Coron, and Antoine Joux, *Improved Generic Algorithms for Hard Knapsacks*, Advances in Cryptology, proceedings of EUROCRYPT 2011, Lecture Notes in Computer Science 6632, pp. 364–385, Springer-Verlag, 2011.

2. Mihir Bellare, Ran Canetti, and Hugo Krawczyk, *Keying Hash Functions for Message Authentication*, Advances in Cryptology, proceedings of CRYPTO 1996, Lecture Notes in Computer Science 1109, pp. 1–15, Springer-Verlag, 1996.
3. Itai Dinur, Orr Dunkelman, and Adi Shamir, *Improved Attacks on Full GOST*, Fast Software Encryption 2012, to appear in Lecture Notes in Computer Science. Available as IACR ePrint report 2011/558.
4. Itai Dinur, Orr Dunkelman, Nathan Keller and Adi Shamir, *Efficient Dissection of Composite Problems, with Applications to Cryptanalysis, Knapsacks, and Combinatorial Search Problems*, Cryptology ePrint Archive, Report 2012/217.
5. Martin E. Hellman, *A Cryptanalytic Time-Memory Tradeoff*, IEEE Transactions on Information Theory, Vol. 26, No. 4, pp. 401–406, 1980.
6. Amos Fiat, Shahar Moses, Adi Shamir, Ilan Shimshoni, and Gábor Tardos, *Planning and Learning in Permutation Groups*, Foundations of Computer Science 1989, pp. 274–279, IEEE Computer Society, 1989.
7. Nick Howgrave-Graham and Antoine Joux, *New Generic Algorithms for Hard Knapsacks*, Advances in Cryptology, proceedings of EUROCRYPT 2010, Lecture Notes in Computer Science 6110, pp. 235–256, Springer-Verlag, 2010.
8. Jean-Jacques Quisquater and Jean-Paul Delescaille *How Easy is Collision Search. New Results and Applications to DES*, Advances in Cryptology, proceedings of CRYPTO 1989, Lecture Notes in Computer Science 435, pp. 408–413, Springer-Verlag, 1990.
9. Antoine Joux, *Multicollisions in Iterated Hash Functions*, Advances in Cryptology, proceedings of CRYPTO 2004, Lecture Notes in Computer Science 3152, pp. 306–316, Springer-Verlag, 2004.
10. Donald Knuth, *The Art of Computer Programming*, 2nd Edition, Vol. 2, pp. 7, Addison-Wesley, 1981.
11. Richard E. Korf, *Finding Optimal Solutions to Rubik's Cube Using Pattern Databases*, Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, pp. 700–705, The MIT Press, 1997.
12. Stefan Lucks, *Attacking Triple Encryption*, proceedings of Fast Software Encryption 1998, Lecture Notes in Computer Science 1372, pp. 239–253, Springer-Verlag, 1998.
13. Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen, *The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl*, proceedings of Fast Software Encryption 2009, Lecture Notes in Computer Science 5665, pp. 260–276, Springer-Verlag, 2009.
14. Ralph C. Merkle and Martin E. Hellman, *On the Security of Multiple Encryption*, Commun. ACM vol. 24, no. 7, pp. 465–467, 1981.
15. María Naya-Plasencia, *How to Improve Rebound Attacks*, Advances in Cryptology, proceedings of CRYPTO 2011, Lecture Notes in Computer Science 6841, pp. 188–205, Springer-Verlag, 2011.
16. Gabriel Nivasch, *Cycle Detection Using a Stack*, Inf. Process. Lett. vol. 90, no. 3, pp. 135–140, 2004.
17. Richard Schroeppel and Adi Shamir, *A T=$O(2^{n/2})$, S=$O(2^{n/4})$ Algorithm for Certain NP-Complete Problems*, SIAM J. Comput. vol. 10, no. 3, pp. 456–464, 1981.
18. Paul C. van Oorschot and Michael J. Wiener, *Improving Implementable Meet-in-the-Middle Attacks by Orders of Magnitude*, Advances in Cryptology, proceedings of CRYPTO 1996, Lecture Notes in Computer Science 1109, pp. 229–236, Springer-Verlag, 1996.