New Technique for Solving Sparse Equation Systems

Håvard Raddum and Igor Semaev

Department of Informatics, University of Bergen, N-5020 Bergen, Norway

Abstract. Most of the recent cryptanalysis on symmetric key ciphers have focused on algebraic attacks. The cipher being attacked is represented as a non-linear equation system, and various techniques (Buchberger, F4/F5, XL, XSL) can be tried in order to solve the system, thus breaking the cipher. The success of these attacks has been limited so far. In this paper we take a different approach to the problem of solving non-linear equation systems, and propose a new method for solving them. Our method differs from the others in that the equations are not represented as multivariate polynomials, and that the core of the algorithm for finding the solution can be seen as message-passing on a graph. Bounds on the complexities for the main algorithms are presented and they compare favorably with the known bounds. The methods have also been tested on reduced-round versions of DES with good results. This paper was posted on ECRYPT's STVL website on January 16th 2006.

Keywords: sparse algebraic equations, block ciphers, algebraic attacks, DES.

1 Introduction

Most of the cryptanalysis done on symmetric key ciphers in the last few years has been focused on algebraic attacks. Much of this interest comes from the fact that the AES can be described as a system of quadratic equations, and that solving this system breaks the cipher [1].

One important feature algebraic attacks have is the fact that you only need very few known plaintexts in order to set up an equation system describing the cipher and determining the key uniquely. This makes these attacks more realistic and threatening than differential or linear attacks that typically require enormous amounts of known or chosen plaintexts.

Strategies for solving non-linear equation systems have been described [2–4] and some have been developed into cryptanalytic attacks [1, 5, 6]. Other work has described interesting algebraic properties found in the AES, but without any actual attacks [7, 8]. There has been some debate over the efficiency of the XSL-attack on the AES, and some papers have appeared [9, 10] indicating it is not as good as claimed by the authors.

One of the problems this field of research has encountered is the difficulty of producing good examples of algebraic attacks carried out in practice. The time and memory requirements of running some of the proposed attacks rapidly grows out of the bounds set by a typical workstation [11]. This means we lack firm proof of how the attacks work in practice, except for very small examples.

In this paper a new method for solving non-linear systems of equations is introduced. Our approach differs a lot from earlier work in that equations are not represented as polynomials. Equations are represented as lists of bit-strings, where each string is a value assignment of variables that satisfies the equation.

We have used reduced-round versions of DES to test our methods. The results show that our techniques easily solves the system coming from four rounds of DES, and that both five and six rounds of DES can be broken faster than exhaustive search. While breaking six rounds of DES is not very interesting in itself, this work gives an example of a successful algebraic attack against something that looks like a real-world cipher.

It will also become apparent that, on a normal workstation, the new method we are proposing is able to solve systems that are probably out of reach by known methods. For example, a system consisting of equations of degree five and 1080 variables describing four rounds of DES with 16 known plaintext/ciphertext blocks is solved in 25 minutes on a PC.

2 Representing equations and running the Agreeing algorithm

Throughout the paper we will only consider equations where the variables have values in $GF(2)$. Let X be a set of Boolean variables of size n. A system of m Boolean equations

$$f_1(X_1) = 0, \ldots, f_m(X_m) = 0 \tag{1}$$

is considered, where $f_i = f_i(X_i)$ is a Boolean function in variables X_i which is a subset of X of size k_i. We assume $k_i \leq k$ for some k.

Definition 1. *A* **configuration** *for an equation $f_i(X_i) = 0$ is an assignment of values to the variables X_i that satisfies $f_i = 0$.*

The variables in X_i are fixed in some order, and a configuration for f_i will be represented as a bit-string of length k_i. The bit in a specific position in the configuration has the value for the variable in the same position in X_i. We will not work with the equations as multivariate polynomials, but instead identify an equation $f_i(X_i) = 0$ with the ordered set of variables X_i, and a list L_i of all configurations for f_i. This object is the basic building block for our methods, so we give it a name.

Definition 2. *A* **symbol** *$S = (X, L)$ consists of an ordered set of variables $X = X(S)$ and a list $L = L(S)$ of configurations for X.*

We expect each f_i to have approximately 2^{k_i-1} configurations. Producing and storing the list L_i thus requires k to be relatively small, for each i we must be able to run through all 2^{k_i} bit-strings of length k_i to determine which are configurations and which are not. When we say our system should be *sparse* we mean that k is relatively small.

We now proceed to describe the core algorithm for solving the system of equations. Every solution of the system can be represented as an n-bit string, where each bit contains the value for one of the variables in X. Each equation in the system has to be satisfied with the solution, so for any symbol S, concatenating bits from the solution into a string for the variables $X(S)$ will produce a configuration found on $L(S)$.

Our idea is to delete configurations that can not be part of a solution from the configuration lists in the symbols. If we are able to remove all wrong configurations we can simply read the values of the remaining configurations in the various $L(S_i)$ to get a solution of the system, at least when it is unique.

Definition 3. *Let $Y \subseteq X$. A configuration for X is said to* **cover** *a configuration for Y if the two are equal for all variables in Y.*

Let two symbols S_i and S_j be given. Let $X_{ij} = X(S_i) \cap X(S_j)$ and L_{ij} be the set of all configurations for X_{ij} covered by at least one configuration on $L(S_i)$. Similarly, let L_{ji} be the set of all configurations for X_{ij} covered by some configuration on $L(S_j)$. We say that S_i and S_j agree if $L_{ij} = L_{ji}$. In case when they do not agree we apply a procedure called agreeing. That is, we delete from $L(S_i)$ and $L(S_j)$ all configurations that do not cover any configuration on $L_{ij} \cap L_{ji}$ and get lists L'_i and L'_j. None of the configurations deleted from $L(S_i)$ and $L(S_j)$ could be part of a full solution, since it would be impossible to satisfy both S_i and S_j with them. The symbols $S'_i = (X(S_i), L'_i)$ and $S'_j = (X(S_j), L'_j)$ agree now, and we replace S_i with S'_i and S_j with S'_j.

The Agreeing algorithm works by repeatedly finding two indices i and j such that S_i and S_j disagree, and apply the agreeing procedure to $L(S_i)$ and $L(S_j)$. When running the Agreeing algorithm we often run into situations where S_i and S_j agree, but S_j and S_k disagree. After deleting some configurations from S_j to make it agree with S_k, it may well be that S_i and S_j disagree. In other words, applying the agreeing procedure to one pair of symbols may cause disagreement in other pairs. If there is enough overlap among the various X_i we will get a chain-reaction of deletions of configurations that will actually remove all configurations except for those that are part of a full solution.

On the other hand, all pairs of symbols may agree before a solution can be seen. If the amount of readily available information in the system is below some critical mass, the chain-reaction of pairwise agreeings will die out before enough deletions have occurred. What typically happens for the reduced-round DES systems (that in all likelihood have unique solutions) is that deletions occur in the beginning of the Agreeing algorithm, but that the symbols are put into an agreeing state while there still are a lot of configurations left in the symbols, see Section 4 for a detailed description of the Agreeing algorithm's output and its complexity. The next section deals with strategies for overcoming this problem.

Running the Agreeing algorithm may be seen as message-passing on a graph. The nodes in the graph will be all the symbols $S_1, \ldots, S_m$ together with the symbols $S_{ij} = (X_{ij}, L_{ij} \cap L_{ji})$ that have a non-empty X_{ij}. There will be edges from each S_{ij} to S_i

and S_j. If a configuration c on L_i does not cover any configuration on $L(S_{ij})$, we can think of S_{ij} sending a message over the edge to S_i telling the symbol to delete c. The Agreeing algorithm will continue as long as there are messages to be sent in the graph.

3 Trying to force the solution to emerge

When the symbols are already in an agreeing state, the Agreeing algorithm will not do anything. In this case it is necessary to do something to make the Agreeing algorithm start again. We have examined two different strategies for getting out of an agreeing state.

3.1 Splitting

The first of the methods for re-starting the Agreeing algorithm we have called *Splitting*, and is quite simple. When the graph reaches an agreeing state with no visible solution what we do is the following: We focus on one symbol S, and split $L(S)$ in two parts, L_1 and L_2. We then replace $L(S)$ with L_1 or L_2 and start the Agreeing algorithm again. The correct configuration for S (assuming unique solution) is found on either L_1 or L_2, so what we are basically doing is guessing on which part that contains this configuration. If it becomes clear that the guess was wrong, we will run the Agreeing algorithm with the other list instead.

For the DES systems, guessing only once is not enough help for the Agreeing algorithm to solve the system. In general, the symbols will again come to an agreeing state with no apparent solution after the first guess. Then we need to guess again, run the Agreeing algorithm once more, and so on. Each time we guess, we are guessing one bit of information, since the correct configuration is found on either L_1 or L_2. One can see that guessing on the values of a set of variables for restarting the Agreeing algorithm is a particular case of splitting.

When will it become clear that a guess was wrong? When we make a wrong guess somewhere, we are deleting the correct configuration from a symbol's configuration list, making it impossible to find a solution. What happens, possibly after making more guesses, is that the Agreeing algorithm deletes all configurations in some $L(S)$. When this happens we will backtrack to the last guess made, and try the other possibility. If this also turns out to be a dead end, we will backtrack to the guess before that, and try the other possibility from that point, etc.

What we are doing is going through a binary search tree, looking for the solution of the system. When we follow the branch of the tree corresponding to the correct guesses we find a solution. The leaves of this tree will be the point where we either find a solution, or the points where some $L(S)$ becomes empty. The leaves will be at somewhat varying depths, and the complexity of this approach will be exponential in the average depth of the tree.

3.2 Gluing

The other main strategy for making the Agreeing algorithm start again is called *Gluing*. With this method we do not do any guessing, but instead merge two symbols into one, thereby bringing their joint information about the solution into the open.

Let symbols S_i and S_j be given. We put $Z = X(S_i) \cup X(S_j)$ and $X_{ij} = X(S_i) \cap X(S_j)$ and define a list L of configurations for Z. The list L consists of all configurations (a, b, c), where b is a configuration for X_{ij}, $(a, b) \in L(S_i)$, and $(b, c) \in L(S_j)$. In other words, L consists of all configurations that cover one configuration on $L(S_i)$ and one configuration on $L(S_j)$. The symbol $S = (Z, L) = S_i \circ S_j$ is the result of gluing the symbols S_i and S_j, and the configuration corresponding to the solution of the system is found on L.

When our symbols reach an agreeing state, we can glue together symbols to create new equations. When gluing S and T together we discard S and T, since all information in them are contained in $S \circ T$. When we have glued together several pairs of symbols the new set of symbols will in general not be in an agreeing state, so we can start the Agreeing algorithm again.

The price to pay when gluing together symbols is longer configuration lists. Assuming S and T agree, the number of configurations in $S \circ T$ will be at least as big as $\max\{|L(S)|, |L(T)|\}$, and may be as big as $|L(S)| \cdot |L(T)|$. In practice we have to set a threshold and only glue together symbols that produce configuration lists with size below this threshold. This means we may run into cases where we can not afford any symbols to be glued.

When computer constraints make it too costly to perform gluing, we have considered partial gluing. For two symbols (X_i, L_i) and (X_j, L_j), we choose $Y_i \subseteq X_i$ and $Y_j \subseteq X_j$. Next we make L'_i and L'_j by projecting all configurations on L_i onto Y_i and all configurations on L_j onto Y_j. In general, $|L'_i| < |L_i|$ and $|L'_j| < |L_j|$ since different configurations for X_i and X_j may be equal when projected onto Y_i and Y_j. If Y_i and Y_j were chosen carefully we may glue the symbols (Y_i, L'_i) and (Y_j, L'_j) and cause disagreement with some other symbols.

Two difficult issues concerning partial gluing is how to choose Y_i and Y_j in an optimal way, and how to ensure that the new symbol coming from partial gluing is actually bringing us closer to solving the system. One strategy for choosing Y-subsets has been tried, but so far we do not have any results to report about partial gluing. We only mention partial gluing as a topic for future research.

4 Complexity Issues

In this Section we will look at what the output of the Agreeing algorithm is, and what its running time is. We then compare the Agreeing-Gluing algorithm with known approaches for solving sparse Boolean systems of equations.

Given the set of symbols $S_i = (X_i, L_i)$, $1 \leq i \leq m$ related to the initial system of equations (1), we consider a set of subsymbols $(X_i, U_i) \subseteq (X_i, L_i)$ meaning that $U_i \subseteq L_i$ for all $1 \leq i \leq m$. The latter set of subsymbols is called a maximal agreed set of subsymbols if the symbols (X_i, U_i) pairwise agree and for any sets U_i'

$$U_i \subseteq U_i' \subseteq L_i,$$

with $U_i \subset U_i'$ for at least one i, the set of subsymbols (X_i, U_i'), $1 \leq i \leq m$ does not agree.

Proposition 1. *1. The maximal agreed set of subsymbols is unique.*
2. The output of the Agreeing algorithm is the maximal agreed set of subsymbols for the initial set of symbols.
3. The running time of the Agreeing algorithm is bounded by $O(k^2m^32^{2k})$ bit operations with memory requirements $O(m2^k)$ bits.

Proof: Assume there are two maximal agreed sets of subsymbols: (X_i, U_i), $1 \leq i \leq m$ and (X_i, U_i'), $1 \leq i \leq m$. Then one constructs a new set of subsymbols $(X_i, U_i \cup U_i')$, $1 \leq i \leq m$. The latter subsymbols pairwise agree. That is only possible when $U_i = U_i'$, $1 \leq i \leq m$. The first statement is proved.

To prove the second statement, let (X_i, U_i), $1 \leq i \leq m$ be the output of the Agreeing algorithm and

$$(X_i, U_i) \subseteq (X_i, U_i') \subseteq (X_i, L_i)$$

for some set of symbols (X_i, U_i'), $1 \leq i \leq m$ which pairwise agree. The intermediate stage of the Agreeing algorithm is a set of subsymbols $(X_i, L_i') \subseteq (X_i, L_i)$. Let this be the stage just before deleting any of the configurations in $U_i' \setminus U_i$, $1 \leq i \leq m$. Then $(X_i, U_i') \subseteq (X_i, L_i')$, $1 \leq i \leq m$. The Agreeing algorithm now deletes some $a \in U_i' \setminus U_i$. This means there is a symbol (X_j, L_j') such that L_j' does not contain any configuration that covers the projection of a on $X_{ij} = X_i \cap X_j$. Then U_j' does not contain any configuration that covers the projection of a on X_{ij} either, so (X_i, U_i') and (X_j, U_j') do not agree. This implies $U_i' = U_i$ for all $1 \leq i \leq m$ and proves the statement.

To prove the last statement one sees that

$$\sum_{i=1}^{m} |L_i| \leq m2^k.$$

To find a pair of symbols $S_i = (X_i, L_i')$ and $S_j = (X_j, L_j')$, which do not agree, one should try at most $m(m-1)/2$ pairs at any stage of the Agreeing algorithm. If such a pair exists $\sum_i |L_i|$ should be decreased by at least 1, otherwise the algorithm terminates. Finding the configurations to remove from L_i' and L_j' takes $O(k2^k)$ steps because common configurations in L_{ij}', L_{ji}' (see the definition of the agreeing procedure) may be found by a sorting algorithm. So on the whole the complexity is $O(km^32^{2k})$ operations with binary k-strings. The statement on the memory requirements is obvious. This finishes the proof of the Proposition.

Proposition 1 implies that for a fixed k the Agreeing algorithm has a polynomial behavior and its result does not depend on a particular way of pairwise agreeings. However, it generally fails to find a solution. Thus the Agreeing algorithm should be combined with some other techniques as splitting or gluing.

The gluing is by itself able to solve any system of sparse Boolean equations. Really, one applies gluing to construct the set of all solutions for the subsystem of equations

$$f_1(X_1) = 0, \ldots, f_t(X_t) = 0, \tag{2}$$

in variables $X(t) = X_1 \cup \ldots \cup X_t$ subsequently for $t = 1, \ldots, m$. We call this the Gluing algorithm. In another paper we prove

Theorem 1. *Let natural numbers m and $k_1, \ldots, k_m \leq k$ be given and the subsets of variables $X_1, \ldots, X_m \subset X$ and Boolean functions $f_1, \ldots, f_m$ be chosen uniformly and independently of each other. Then the mathematical expectation of the complexity of the Gluing algorithm is*

$$O((2e^{\gamma_0} + \epsilon)^n + poly(n)m) \tag{3}$$

bit operations as k is fixed and n tends to infinity. Here

$$\gamma_0 = -\frac{\ln 2}{k} - (2^{\frac{1}{k}} - 1)\ln(\frac{1-2^{-1}}{1-2^{-\frac{1}{k}}}),$$

ϵ is any fixed positive real number and $poly(n)$ is a polynomial in n.

We do not prove Theorem 1 here for lack of space. We will give a heuristic estimation on the complexity of the Gluing algorithm instead. We consider the symbols $(X(t), U_t), (X_{t+1}, L_{t+1}), \ldots, (X_m, L_m)$ after $t-1$ applications of gluing. Here U_t is a set of configurations for $X(t)$, solutions to the subsystem of equations (2) of the initial system. Let r_t be the size of $X(t)$. One sees that the size of the set U_t is about $2^{r_t - t}$ because adding a new independent equation reduces the number of solutions by one half on the average. One considers the number r_t as the number of filled boxes when particles are randomly allocated in n boxes by complexes of k particles. From [14], pp. 211-213 we know that the expected number of empty boxes is $n(1-k/n)^t$, so r_t is approximately $n - n(1-k/n)^t \approx n(1-e^{-tk/n})$ for a bounded k and n tending to infinity. It implies that U_t is of size approximately

$$2^{\delta(t/n)n},$$

where $\delta(h) = 1 - e^{-hk} - h$. One finds that $h = \ln k/k$ is the only extremum of this function in the interval $0 < h$ and $\delta(\ln k/k) = 1 - \ln k/k - 1/k$ is the maximum of $\delta(h)$ in this interval. So the running time of the algorithm is estimated at

$$O((2^{1-\ln k/k - 1/k} + \epsilon)^n + poly(n)m)$$

on the average. One can check that this heuristic bound is even lesser than (3) at least for k up to 20, but to be at the right side one should prefer (3). Let $k_1 = \ldots k_m = k$. Then we are able to prove that the probability that the complexity of the Gluing algorithm is bounded by $O(c^n)$, where $c < 2e^{\gamma_0}$, is tending to 0 as n tends to infinity. In this sense the bound (3) is optimal.

In the above formulation the Gluing algorithm requires as much memory as its running time, but there is a variant, called Gluing1, taking the same running time to terminate and only using $O(m2^k)$ bits of memory. Even a slightly faster way of gluing(Gluing2) is possible. To this end one applies the Gluing algorithm to find all solutions for the first t_0 equations in (1), and then one separately finds all solutions for the next t_0 equations for some parameter t_0. After that one glues these two symbols to get all solutions for the first $2t_0$ equations. Then one proceeds as in the basic Gluing algorithm. With the above heuristic argument t_0 should be $h_0 n$, where h_0 is the only solution to the equation $\delta(h) = \delta(2h)$. We always get a better bound than that of the basic Gluing algorithm and similarly the heuristic estimation is lesser that the mathematically proven one. We will not go further into details on the Gluing2 algorithm here.

The estimation (3) is also a bound on the complexity of the combined Agreeing-Gluing algorithm, where we run the Agreeing algorithm between some of the gluings. This is at least as efficient as the plain Gluing algorithm. The experiments described in the next section show that it is much more efficient, but its asymptotic running time remains unknown. For this reason we will use (3) as an upper bound in order to compare the Agreeing-Gluing algorithm with known approaches to solve systems of sparse Boolean equations.

Worst case estimations for (1) come from the k-SAT problem analysis. k-SAT is a problem to determine, given a conjunctive normal form F with n variables and such that each clause of F contains at most k literals, whether or not there is a satisfying assignment for F. These two problems are polynomial-time equivalent. Let

$$f(x_1, \dots, x_k) = 0 \tag{4}$$

be any Boolean equation in k Boolean variables. Let $(a_{11}, \dots, a_{1k}), \dots,$ and $(a_{s1}, \dots, a_{sk})$ be all binary vectors such that $f(a_{i1}, \dots, a_{ik}) = 1$. The vector $(b_1, \dots, b_k)$ is a solution to (4) if and only if it is a satisfying assignment for the conjunctive normal form

$$F_f = (x_1^{a_{11}} \vee \dots \vee x_k^{a_{1k}}) \wedge \dots \wedge (x_1^{a_{s1}} \vee \dots \vee x_k^{a_{sk}}),$$

where we denote

$$x^a = \begin{cases} x, & \text{if } a = 0, \\ \bar{x}, & \text{if } a = 1, \end{cases}$$

that is $x^a = 0$ if and only if $x = a$. Given the system of equations (1) one constructs a conjunctive normal form F which is a conjunction of F_{f_i}. One now sees that $(b_1, \dots, b_n)$ is a solution to (1) if and only if this vector is a satisfying assignment for F. Obviously, any k-SAT problem may be represented by a system of k-sparse Boolean equations.

k-SAT (for $k \geq 3$) is one of the classical NP-complete problems and there is a vast reference list on this problem. Recently there has been a large effort to design deterministic and randomized exact algorithms for k-SAT. These efforts resulted in a number of deep and powerful techniques for solving SAT efficiently. Nice examples of such techniques are the Davis-Putnam algorithm, Shöning Local Search and Random Walks. SAT on n variables can be trivially solved in time $O(2^n)$ by trying all possible assignments, but constructing an $O(c^n)$ algorithm for $c < 2$ is a long standing open problem. However, for small values of k there are much faster algorithms for k-SAT with $c = c_k < 2$, see the survey article [13]. Such bounds are worst case estimations to the problem of solving equations (1).

The estimations of the Gluing1 and Gluing2 algorithms compare favorably with the above worst case bounds at least for small k as one sees from the data tabulated below:

	the worst case	Gluing1, the average	Gluing2, the average
c_3	1.324	1.262	1.238
c_4	1.474	1.355	1.326
c_5	1.569	1.425	1.393
c_6	1.637	1.479	1.446
...	...	...	...

The constant c_k is given such that the algorithm runs in time $O(c_k^n)$.

In the case of n Boolean equations of algebraic degree d in n variables defining a so-called semi-regular system, the popular Gröbner Basis algorithm gets the complexity of $O(\binom{n}{\alpha_d n}^2) \sim 2^{2H(\alpha_d)n}$ bit operations, where $H(\alpha_d)$ denotes the binary entropy function. The numbers α_d are explicitly given in [12], e.g. $\alpha_2 \approx 0.09$, $\alpha_3 \approx 0.15$, and $\alpha_4 \approx 0.2$. For $d = 2$, one gets the bound $O(1.7^n)$ by guessing a number of variables before the Gröbner Basis algorithm (or XL) application, see [15]. But for $d \geq 3$ the value $2^{2H(\alpha_d)n}$ exceeds the cost of the brute force algorithm and guessing any number of variables does not help. It is clear that, although the algebraic degrees of random equations in (1) are bounded by k, the probability that a considerable part of them are of algebraic degree 2 or linear is very low. When $k \leq 4$ we are however able to replace the equation $f_i(X_i) = 0$ by an equation of algebraic degree ≤ 2, though with loosing some information on the final solution. Then we apply the Gröbner Basis algorithm. But the above estimations imply that our Agreeing-Gluing algorithm still gives a better bound in this case. So it looks plausible that the latter has generally a better behavior than the former.

5 Experiments

The ideas described have been tested on equation systems representing reduced-round versions of DES. The equation systems have been created using two parameters; r, the number of rounds and $ntxt$, the number of known plaintext/ciphertext pairs used. The number of equations and variables in a system, m and n, is then given by:

$$n = 56 + 32 \cdot ntxt \cdot (r - 2) \qquad\qquad m = 32 \cdot r \cdot ntxt.$$

The exact construction of the systems is given in Appendix A.

Some choices regarding the implementation of the splitting and gluing methods had to be made. The strategies implemented were as follows.

Splitting: Each time we need to split, we need to select one symbol whose configuration list should be cut in half. Some experimenting was done, and we found that always doing the splitting in the symbol that already has the smallest number of possible configurations, was most efficient. We could not see that it mattered much how the splitting of the chosen configuration list was done, so it was simply done by putting every other configuration on L_1 and the rest on L_2.

Gluing: A threshold t is initialized to 64 (chosen because of the specific nature of the DES systems). We then run the following pseudo-code:

```
while system not solved:
    while progress is made:
        - run Agreeing algorithm.
        - glue all pairs of symbols giving
        new symbols with #configurations≤ t.
    t := 2t.
```

Increasing the threshold t only when necessary is a dynamic way of finding how large configuration lists we must accept for the Agreeing-Gluing algorithm to work.

5.1 Three rounds

Not surprisingly, three rounds of DES is very easy to solve with the methods described. With $ntxt \geq 2$, the Agreeing algorithm is able to do the job alone (in about one second), no splitting or gluing is necessary.

With $ntxt = 1$, a little help from the splitting or gluing methods are needed. We tested the methods four times each, with a different plaintext/ciphertext pair each time. The average depth of the search tree using the splitting method was 8.2. The average size of the largest configuration list using the gluing method was 647.

5.2 Four rounds

Table 1 summarizes the results from testing our methods on systems coming from four rounds of DES. Each entry in the table is computed as an average of four different tests, each time with different sets of plaintext/ciphertext pairs.

We see that increasing *ntxt* reduces the complexities for the splitting and gluing methods. Here we also find the system referred to in the introduction; for $ntxt = 16$ and $r = 4$ we get an equation system with 1080 variables and 2048 non-linear equations.

ntxt	1	2	4	8	16
splitting - depth of tree	19.2	16.4	12.2	9.3	8.7
gluing - largest conf. list	$2^{22.8}$	$2^{20.9}$	2^{18}	$2^{15.3}$	2^{15}

Table 1. Complexities for solving systems from four rounds of DES.

5.3 Five and six rounds

For five and six rounds, we ran out of memory (1 GB) before the Agreeing-Gluing algorithm had found a solution, so we do not have any results using this method. All we can say is that when $ntxt = 8$, the size of the largest configuration list becomes larger than 2^{24}.

The splitting method is also very costly to do in practice, but here we can estimate the depth of the search tree without having to wait until the actual solution is found. Our implementation of the splitting method ran until 1024 leaves had been visited in the search tree, and computed the average depth of the tree based on this. We believe this gives a quite accurate result. The depth of the leaves does not vary a lot, and when monitoring the program traversing the tree, the different depths of leaves appear to be evenly distributed. Table 2 contains the results. Again, each entry is an average of four different tests using different plaintext/ciphertext pairs. For $r = 6$ and $ntxt \geq 4$ our program consumed too much memory to run.

ntxt	1	2	4	8
five rounds	27.6	26.8	25.8	23.7
six rounds	37.2	37.2	-	-

Table 2. Depth of search trees using splitting method.

6 Conclusions

In this paper we have presented some new techniques for attacking the problem of solving sparse systems of non-linear equations. The complexities for the Gluing algorithm compares very favorably to known approaches, and the experiments done indicate the methods in this paper are more efficient than classical methods using multivariate polynomials for representing the equations.

The techniques presented here are new, so there is still a lot of work to be done in this direction. One topic for future research is to find out whether partial gluing gives substantial improvements compared to regular gluing. Another is to find out if our methods can be combined with other techniques to make stronger algorithms. Of course, the Agreeing algorithm with its catalysts should also be tried on systems from other block or stream ciphers, especially the AES.

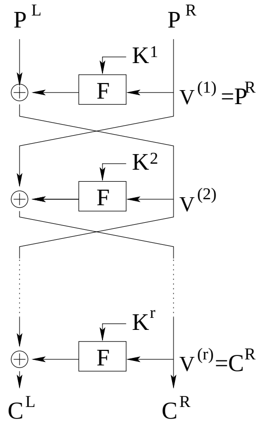
We hope the new methods for solving sparse systems of equations will be taken into the toolbox together with the others and developed further.

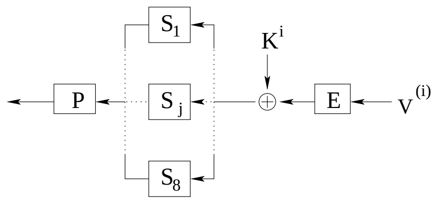
References

1. N. Courtois, J. Pieprzyk. *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations* ASIACRYPT 2002, LNCS 2501, pp. 267 – 287, 2002.
2. A. Shamir, J. Patarin, N. Courtois, A. Klimov. *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations*, EUROCRYPT 2000, LNCS 1807, pp. 392 – 407, 2000.
3. J.-C. Faugère. *A new efficient algorithm for computing Gröbner bases (F4)*, Journal of Pure and Applied Algebra, Volume 139, Issues 1 - 3, pp. 61 – 88, June 1999.
4. J.-C. Faugere. *A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)*, Proceedings of ISSAC '02, pp. 75 – 83, ACM Press, July 2002.
5. N. Courtois, W. Meier. *Algebraic Attacks on Stream Ciphers with Linear Feedback*, EUROCRYPT 2003, LNCS 2656, pp. 345 – 359, 2003.
6. N. Courtois. *The Security of Hidden Field Equations (HFE)*, CT-RSA 2001, LNCS 2020, pp. 266 – 281, 2001.
7. N. Ferguson, R. Schroeppel, D. Whiting. *A Simple Algebraic Representation of Rijndael*, Selected Areas in Cryptography 2001, LNCS 2259, pp. 103 – 111, 2001.
8. M. Robshaw, S. Murphy. *Essential Algebraic Structures within the AES*, CRYPTO 2002, LNCS 2442, pp. 1 – 16, 2002.
9. C. Cid. *Some Algebraic Aspects of the Advanced Encryption Standard*, 4th AES Conference, LNCS 3373, pp. 58 – 66, 2005.
10. C. Diem. *The XL-Algorithm and a Conjecture from Commutative Algebra*, ASIACRYPT 2004, LNCS 3329, pp. 323 – 337, 2004.
11. C. Cid, S. Murphy, M. Robshaw. *Small Scale Variants of the AES*, FSE 2005, LNCS 3557, pp. 145 – 162, 2005.
12. M.Bardet, J.-C.Faugére, and B.Salvy, *Complexity of Gröbner basis computation for semi-regular overdetermined sequences over F_2 with solutions in F_2,* Research report RR-5049, INRIA, 2003.
13. K. Iwama, *Worst-Case Upper Bounds for kSAT,* The Bulletin of the EATCS, (82), 2004, pp. 61 – 71.
14. F.Kolchin,A.Sevast'yanov, and V.Chistyakov, *Random allocations,* John Wiley & Sons, 1978.
15. B.-Y. Yang, J-M. Chen, and N.Courtois, *On asymptotic security estimates in XL and Gröbner bases-related algebraic cryptanalysis,* in ICICS 2004, LNCS 3269, Springer, pp. 401 – 413, 2004.

Appendix A - Constructing equation system from DES

Here we describe how we made the equation system representing the DES algorithm. All bits going into the round function in round i, except for the first and last rounds which have plaintext and ciphertext bits as input, are variables $V^{(i)}$. The rest of the variables are the 56 key bits. This is illustrated in the figure below, together with the round function.





The DES round function.

Each bit in the output of a DES S-box can be expressed as a function of its six inputs, and so defines an equation. The four equations coming from the same S-box share all variables input to the S-box, so it is natural to glue these equations together immediately. Doing this, we get one equation for each S-box in every round. The general form of the equation from S-box j in round i is

$$V_j^{(i-1)} \oplus V_j^{(i+1)} = S_j[V_j^{(i)} \oplus K_j^i],$$

where $V_j^{(i-1)}$ and $V_j^{(i+1)}$ are two four-bit strings and $V_j^{(i)}$ and K_j^i are two six-bit strings. K_j^i are the six bits of round key i going into S-box j, determined by the key schedule. The bits in $V_j^{(i-1)}$ and $V_j^{(i+1)}$ are taken from the input to the previous and the next round. Determined by the permutation P in the output of the round function, they represent the output of S-box j in round i. $V_j^{(i)}$ are the six bits of expanded input to round i going into S-box j.

When the equation comes from an S-box in one of the two first or the two last rounds, some of the $V^{(\cdot)}$-values will be constants from the plaintext or the ciphertext. Adding up the number of bits in the general equation we see that no equation contains more than 20 variables. In the second and the second to last round, the equations contain 16 variables each since $V_j^{(1)}$ and $V_j^{(r)}$ comes from the plaintext and ciphertext. In the first and the last round the equations contain only 10 variables each.

The general equation defines a four-bit condition to be satisfied. If an equation contains a variables, only 2^{a-4} of the 2^a configurations will satisfy the equation, so the largest configuration lists in the system will contain 2^{16} configurations.

The description above is using only one plaintext/ciphertext pair, but can easily be extended. To build a system using several plaintext/ciphertext pairs, the $V^{(i)}$-variables will have to be different for each plaintext/ciphertext pair used, but the key variables remain the same across all equations.