REPORTS IN INFORMATICS

ISSN 0333-3590

Solving Non-linear Sparse Equation Systems over $GF(2)$ Using Graphs

Håvard Raddum

REPORT NO 283 **October 2004**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL http://www.ii.uib.no/publikasjoner/texrap/ps/2004-283.ps

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

Solving Non-linear Sparse Equation Systems over $GF(2)$ Using Graphs

Håvard Raddum
haavardr@ii.uib.no

Department of Informatics, The University of Bergen, Norway

Abstract

Non-linear equation systems arise in many problems. It is well known that solving such systems is NP-complete in general. In this paper we present a method for solving sparse non-linear equation systems, using ideas from graph based iterative decoding techniques. The motivation for doing this comes from cryptanalysis, and we try to attack DES using our method. We show that it is easy to break three rounds of DES, and that four rounds also can be attacked by guessing some of the key bits.

1 Introduction

Linear and differential cryptanalysis were described a decade ago, and posed a serious threat to many block ciphers. However, it is understood how to design a block cipher to prevent these attacks, and most modern block ciphers are designed accordingly. This has encouraged the cryptographic community to look for other ways of doing cryptanalysis on block ciphers.

In the last few years there has been an increasing interest in algebraic attacks on block ciphers [1, 2, 3]. Several papers without attacks, but demonstrating algebraic properties of ciphers have also appeared [4, 5]. The attacks boil down to describing the cipher as a system of non-linear equations, and then try to solve the system. Since the systems that arise are structured and sparse, custom made versions of XL or Gröbner base algorithms [7, 1] may be much better suited to the specific system in question, than a general method for solving a system of non-linear equations. This gives hope for discovering good algebraic attacks, despite the fact that solving general systems of non-linear equations is NP-complete.

However, it is still unclear how well these attacks perform in practice. There is a need for good examples where an algebraic attack on a block cipher has been successfully implemented. This paper proposes a method for solving sparse systems of non-linear equations over $GF(2)$ using principles from LDPC-decoding. The methods used performs very well on random sparse systems. Inspired by this it has been implemented on DES with three (DES-3), and four (DES-4) rounds. The bottom line after trying various settings of parameters in the algorithm is that it is easy to break DES-3 with two known plaintexts, and that DES-4 can be broken using eight known plaintexts and a bit more work.

2 Building a graph from a system of equations

In this section we describe how to construct a graph from a system of equations.

Definition 2.1 *A vertex of the graph will be called a symbol, and it will be labelled by an ordered set of variables. The set of variables in symbol S will be denoted $v(S)$.*

Each equation in the system will have a corresponding symbol. The set of variables in one of these symbols will be all the variables the equation contains. There will also be many other symbols, and the variable set in one of these symbols will be contained in the variable set of a symbol connected to an equation. The symbols will be arranged in levels, such that two symbols on the same level will never have an edge between them (they will not be neighbours).

The idea behind the construction is that each equation has some information about the solution of the system, and the graph will serve to pass information among equations. Two equations that depend on some of the same variables will be able to share information on the common variables through other symbols.

2.1 Constructing the symbols

Assume our system has m equations in n variables. The set of variables that makes up equation i are grouped together into symbol S_i, $i = 1, \ldots, m$. These m symbols make up level 0 in the graph.

Now we recursively construct level l from level $l-1$ as follows ($l \geq 1$):

- If $v(S) \subset v(T)$ for two symbols S and T at level $l-1$, symbol S is moved down to level l.
- For each pair of symbols $\{S, T\}$ remaining at level $l-1$ we compute the set $v(S) \cap v(T)$. If this set is non-empty, we check if there already is a symbol at level l labelled by $v(S) \cap v(T)$. In this case we do nothing, otherwise a new symbol at level l is created, labelled with the set $v(S) \cap v(T)$ (and the variables in the set are fixed in some order).

Note that the symbol at level l with the largest set contains fewer variables than the symbol with the largest set at level $l-1$, so at some point a level will contain no symbols. This ensures that the above process stops, and that there will be a final level. The symbols created with this algorithm will be all the vertices in the graph.

2.2 Stretching edges between symbols

After the symbols have been created and separated into levels, the edges are added. We start by drawing the edges from level 1 to level 0, then from level 2 to level 1 and 0, then from level 3 to the above levels, and so on. Assuming the edges from level l and up has been drawn, the edges from level $l+1$ and up are constructed with the following procedure:

- For each symbol S at level $l+1$, and each symbol T at level l, we draw an edge between S and T if $v(S) \subset v(T)$.
- Next, we consider edges between level $l+1$ and level $l-1$. For each symbol S at level $l+1$, and each symbol T at level $l-1$, we draw the edge between S and T if $v(S) \subset v(T)$ **and** there does not already exist a path of length 2 from S to T.
- We continue in this fashion, considering the levels closest to level $l+1$ first. For each symbol S at level $l+1$, and each symbol T at level $l-i$, we draw the edge from S to T if $v(S) \subset v(T)$ and there does not already exist a path from S to T of length $i+1$, $i = 0, \ldots, l$.

Since a symbol U at level l has the label $v(S) \cap v(T)$ for two symbols S, T at some level above, we are guaranteed that a symbol not at the top level will have at least two neighbours at levels above. These neighbours will be able to send information to each other through U.

2.3 Example of graph construction

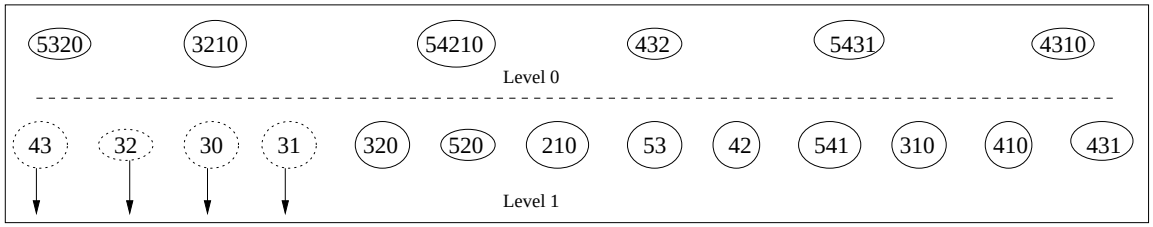
Here we will show an example of the graph construction described above. We are given the following system of 6 equations in 6 unknowns $x_0, \ldots, x_5$:

$$E_0 : x_5x_2x_0 + x_3x_2 + x_3x_0 + x_2x_0 + x_5 + x_3 + 1 = 0$$
$$E_1 : x_3x_2x_1x_0 + x_2x_1x_0 + x_3x_2 + x_2x_0 + x_2 + x_1 = 0$$

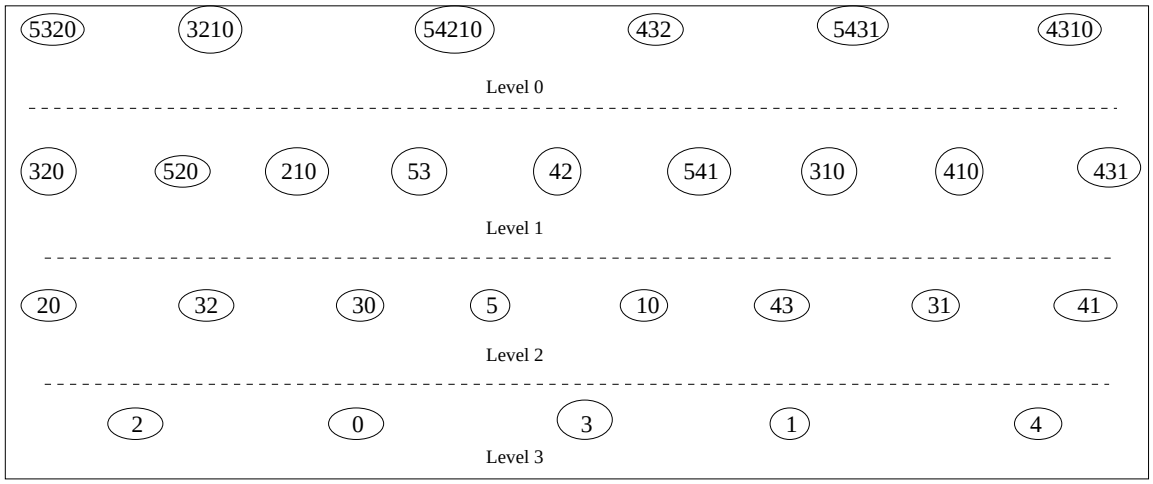
$$E_2 : x_5x_2x_1 + x_4x_1x_0 + x_2x_1x_0 + x_5x_1 + x_4x_0 + x_2x_1 + x_5 + 1 = 0$$
$$E_3 : x_4x_2 + x_3x_2 + x_3 = 0$$
$$E_4 : x_5x_3 + x_4x_3 + x_4 + x_1 + 1 = 0$$
$$E_5 : x_4x_1x_0 + x_4x_3 + x_3x_1 + x_3 + x_0 + 1 = 0$$

From this we see that equation E_0 contains the variables $\{x_5, x_3, x_2, x_0\}$, E_1 depends on $\{x_3, x_2, x_1, x_0\}$, etc. Level 0 of the graph will therefore look like the figure below, where we only label a symbol with the indices of the variables it contains.

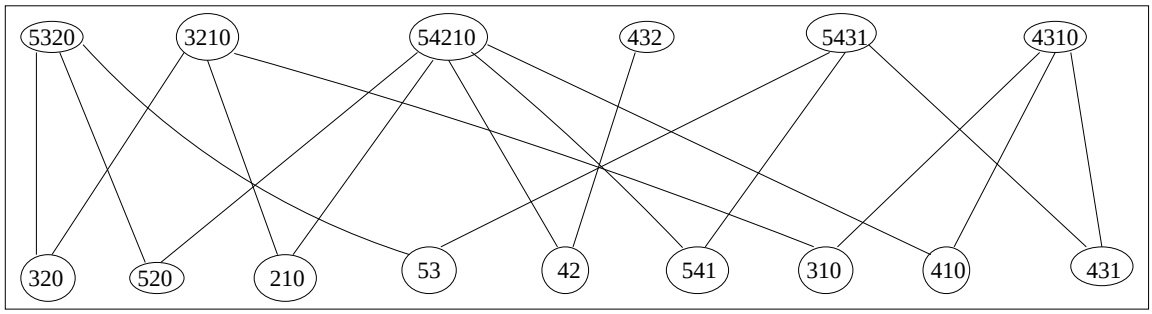
None of the symbols at level 0 has a label set contained in another symbol's set, so there will be no symbols moved down from level 0 to level 1. Next we take the intersection of label sets from each pair of symbols at level 0 to create new symbols at level 1. We get the following picture, where we also have indicated which of the new symbols have label sets contained in other symbols' sets at level 1, and will be moved down to level 2 in the next iteration of the construction.



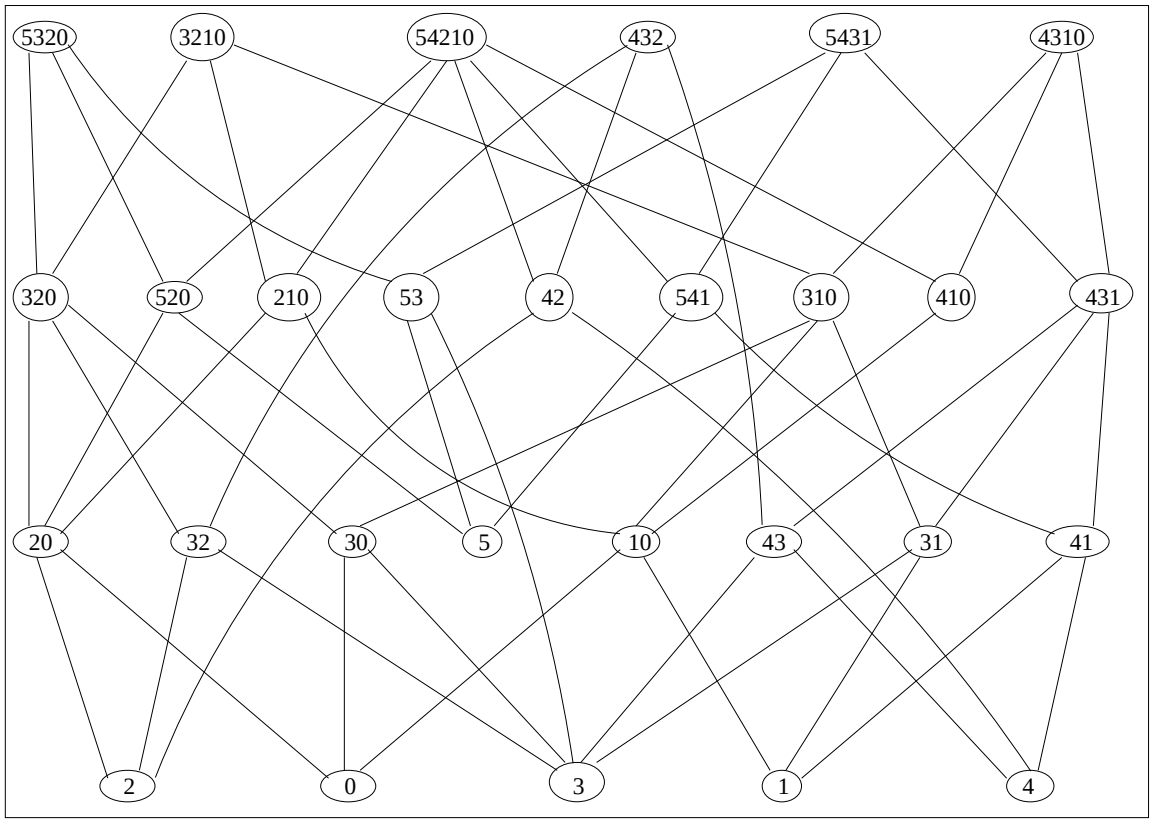
The construction algorithm creates four levels of symbols before it stops. The following picture contains all the symbols of the graph.



Next, we start drawing the edges by connecting symbols at level 0 and 1 according to the procedure above to get the following picture.



When we are finished making the edges, the graph construction is complete and we are left with the following graph.



Most of the edges connect symbols at neighbouring levels, but there are a few exceptions.

3 Message passing

Here we will describe how we will try to "decode" the solution of the system using the graph constructed in the previous section. This is inspired from LDPC-decoding, where messages are sent along the edges of the graph. The idea is that the information about the solution that every equation has will be spread over the whole graph. Eventually we hope a solution that satisfies all equations will emerge.

3.1 Configuration lists

We begin by defining what we mean by a configuration.

Definition 3.1 *A configuration for a symbol S is an assignment of values to the variables in the ordered set $v(S)$.*

In addition to the set of variables, a symbol S also contains a list of the possible configurations S can have. These values have the form of bit-strings of length $|v(S)|$. The symbols at level 0 are constructed from the equations in the system, so the list for one of these symbols will only contain configurations that satisfy the associated equation. For example, the list of configurations for the symbols (5320) and (432) in the example above will only contain the strings satisfying equations E_0 and E_3, respectively:

x_5	x_3	x_2	x_0
0	0	1	1
0	1	0	0
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	1
1	1	1	0

x_4	x_3	x_2
0	0	0
0	0	1
0	1	1
1	0	0

Since the equations defining the two symbols above are non-linear, the lists of configurations are "unbalanced". For example, symbol (5320) believes $x_5 = 1$ with probability 0.75. Symbol (432) knows that the symbol (32) can not have value 10, and believes it has value 00 with probability 0.5, 01 with probability 0.25 and 11 with probability 0.25. This unbalancedness in the equations are necessary for our algorithm to work. It is interesting to note that configuration lists from a linear system are completely balanced, and that the message-passing algorithm which we will describe is useless on a linear system.

Keeping these configuration lists requires a big system to be sparse, in the sense that any one equation in the system can not depend on too many variables. We must be able to run through all the $2^{|v(S)|}$ possible configurations for symbol S in order to find out which ones to put on the list and which ones to discard. The symbols at lower levels, not corresponding directly to an equation, also contain configuration lists. Initially these lists contain all configurations for the associated sequence of variables.

3.2 Combining messages

Definition 3.2 *Let S and T be connected by the edge e, and assume that S is found on a higher level than T. A message on e is a probability distribution over bit-strings of length $|v(T)|$.*

There will be two messages on e, one going to S and one going to T. Initially, all distributions in the graph will be uniform, corresponding to the fact that we do not know anything about the solution before any messages have been sent. We can combine two probability distributions of the same length using the point-product operator.

Definition 3.3 *Let $P = (p_0, \ldots, p_{2^r-1})$ and $Q = (q_0, \ldots, q_{2^r-1})$ be two probability distributions over bit-strings of length r. The point-product of P and Q, denoted by $P \times Q$ is defined as*

$$P \times Q = \frac{1}{N}(p_0 q_0, \ldots, p_{2^r-1} q_{2^r-1}),$$

where the normalizing constant $N = \sum_{i=0}^{2^r-1} p_i q_i$.

A message P coming in to S from a symbol at a higher level will be a distribution on configurations of $v(S)$. A message Q coming in to S from a neighbour T at a lower level will be a probability distribution on configurations of only a subset of $v(S)$, so these two distributions can not be combined directly using the point-product rule. We solve this by expanding Q into Q' as follows.

- For any bit-string i of length $|v(S)|$, take out the $|v(T)|$ bits in i found in positions corresponding to variables also in $v(T)$. Rearrange these bits according to the order of $v(T)$ into bit-string j.
- For $i = 0, \ldots, 2^{|v(S)|} - 1$, let $q'_i = q_j / 2^{|v(S)|-|v(T)|}$.

For example, if $Q = (0.1, 0.2, 0.3, 0.4)$ is a probability distribution for the symbol (30), the expanded Q' for the symbol (320) will be $Q' = (0.05, 0.1, 0.05, 0.1, 0.15, 0.2, 0.15, 0.2)$.

The expansion is chosen like this because it preserves the information in Q. If Q' is marginalized to the subset $v(T)$ the resulting distribution will be Q, so we have not lost any information. On the other hand, if Q' is marginalized to the set $v(S) \setminus v(T)$ the distribution will be uniform, so no information has been added.

A new message from T to S is now computed as follows. Let the degree of T, the number of edges incident to T, be d, and let $P_1, \ldots, P_d$ be the distributions coming in to symbol T, where we have expanded distributions coming from neighbours at lower levels. Let P_i be the distribution coming from S. A principle in LDPC-decoding is that a message from vertex v to vertex u should not depend on the message received from u. This is because it would make it easier for small errors in the messages from u to feed upon themselves and become big. We use the same principle, and compute the new message P from T to S as

$$P = P_1 \times \ldots \times P_{i-1} \times P_{i+1} \times \ldots \times P_d.$$

The configuration list of T shows all possible values the symbol can have. Before the message is sent, we therefore run through P and set the values in positions corresponding to configurations not in the list to zero, and re-normalize. After this, the message is ready to be sent to S.

The message from S to T is computed in the same way, combining all messages coming in to S except for the one from T, but this time the resulting distribution must be marginalized to the set $v(T)$.

The reason for choosing the point-product for combining messages is that the probabilities of the resulting distributions tend to be more extreme than the probabilities in the individual messages before combining. For example, if $P = (0.5, 0.5)$ and $Q = (0.3, 0.7)$ are two distributions for the symbol (0), then $P \times Q = (0.3, 0.7)$. The fact that P has complete uncertainty about the value of x_0 does not cause any doubt about the information Q has. If $P = (0.4, 0.6)$ and $Q = (0.3, 0.7)$, then $P \times Q = (0.222, 0.778)$, so when both P and Q agree on which value is more probable, the combined message will have even more certainty about the value.

This is a general trend for larger distributions too. So the more messages are made using the point-product rule, the less entropy they will have.

3.3 Deleting configurations solves the system

When a symbol receives a new message, it will run through its list of configurations. If the new message says a configuration on the list has probability 0, the configuration is deleted from the list.

As more and more messages are sent on all the edges in the graph, the configuration lists found in the symbols should become shorter, and hopefully they will eventually only contain one configuration (assuming the system has a unique solution). The remaining configuration in a symbol will tell us the values of the variables in the symbol, and collecting this information from all symbols gives us a solution of the system.

3.4 Strength of messages

As noted above, the messages tend to have less entropy as the algorithm advances. In an implementation of the method, two small numbers multiplied together may give the answer 0 because of the finite precision in the computer. If this causes the configuration that matches the solution to be deleted, the algorithm will fail.

It seems like the algorithm described so far tries to push too much information through the graph, too fast. Some messages lose their entropy faster than others, and these messages will dictate a partial solution that suits the symbols locally around the edges where they exist. This partial solution may fit very well in a subgraph, while not being part of the correct solution. When this happens, a symbol in another part of the graph will eventually delete all its configurations, killing all hope for a solution to be found.

What is needed is time to let the information in the messages flow through the whole graph before making any hard decisions.

We can help on the situation by adjusting the strength of the messages being sent.

Definition 3.4 *Let $P = (p_0, \ldots, p_{2^r-1})$ be a probability distribution. We define P^a, $a \geq 0$, as*

$$P^a = \frac{1}{N}(p_0^a, \ldots, p_{2^r-1}^a),$$

where $N = \sum_{i=0}^{2^r-1} p_i^a$, and with the convention that $0^0 = 0$. We say that P^a is P with strength a.

Note that $P^2 = P \times P$, so it is natural to use this definition of strength when the point-product rule is used to combine messages.

3.4.1 Strength 0

For $0 \leq a < 1$, the effect of giving P strength a is to bring the non-zero values in the distribution closer to a uniform distribution. In the extreme case P^0, the non-zero values will make a uniform distribution. If we set each message to strength 0 before sending it, we will not run any risk of deleting the correct configuration from a list. This is because all non-zero values in a distribution will be the same, so when combining two messages all multiplications of non-zero values will be the same. When a configuration is deleted from a list when using the strength zero strategy, it is because it has been shown to be impossible by some of the equations in the system, not because it was estimated to have too low probability.

Some systems may be solved using the strategy of letting each message have strength zero. When a symbol S is deleting some configurations from its list, there may be a neighbour of S that may use this information to also shorten its list, if it receives a message from S. This may give a chain reaction of deletions. When the degree of connection of the graph is above some critical level, we will never run out of symbols that can shorten its configuration list this way. Eventually only parts of the correct solution will remain on the lists.

The system given in the example from Section 2 can be solved using the strength zero strategy. Each equation in that system contains a large portion of the total number of variables, so each pair of equations has two or three common variables. This is enough to keep the chain reaction going. In the end, when no symbol can shorten its configuration list, each symbol has a list of one or two configurations, one for each of the two possible solutions $(x_5, x_4, x_3, x_2, x_1, x_0) = (1, 0, 1, 1, 0, 0)$ or $(x_5, x_4, x_3, x_2, x_1, x_0) = (1, 0, 0, 1, 1, 1)$.

For systems that are more sparse, the strength zero strategy does not work. If the graph is not connected well enough, the deletion process described above stops rather fast, after only relatively few configurations have been deleted. For these systems we may give a message strength a before sending it, for some $0 < a < 1$. If suitable a's are chosen, we may stall the loss of entropy in some messages for long enough to let information from other parts of the graph make an impact.

3.5 Message passing schedule

Now that we have defined how to combine messages and how to give them different strengths, we will look at ways to schedule the message passing. The algorithm may use a static schedule, where all the edges are updated according to a fixed order, or it may use a dynamic schedule, where the next edge to be updated is chosen according to the current state of the graph. Our experience is that dynamic scheduling is better than static, and this is what has been used on DES-3 and DES-4.

There are numerous ways to design a dynamic message passing schedule, and different lines of reasoning that suggest one or another. One may compare messages using euclidean distance or entropy. Euclidean distance has the advantage that two different messages will always have a positive distance, but entropy is an information theoretic measure that says more about the amount of information carried in a message, and thus is better suited for our use. When an edge is updated it may be quarantined for some number of iterations, in order to avoid small loops in the sequence of edges updated. Going in loops is bad for LDPC-decoding. We may have a bias towards updating edges whose message is less "advanced", either in the sense of high entropy or in the sense of low euclidean distance from the uniform distribution. The reasoning behind this is that it should cause the whole graph to be active, and that it will prevent any part of the graph to decide too quickly. Then there is the question of which strength to give a message once it has been chosen, and the question of how to compare probability distributions of different sizes.

A lot of trial and error has been done, and we will only describe the schedule that we found to give the best results. We choose the next edge to send a message in the following way:

- The measure used on a probability distribution is average bit-entropy. If P is a distribution on n bits, we compute $H(P)/n$ (H is the entropy function), and call this value the average bit-entropy of P.
- For each edge e containing the message P, we compute P', the message that will be sent on e if e is chosen as the next edge to be updated. We compute $(H(P) - H(P'))/n$, that is how much the average bit-entropy will drop in this potential new message, and store the value in a table.
- The edge that has the largest reduction in average bit-entropy is chosen as the next edge to be updated.

The reason for trying this scheduling is that information about the system is spread as quickly as possible. The edge containing the message that has the largest drop in average bit-entropy is the edge that gives the most new information. We divide by the number of variables the message says something about to get fair comparison of messages of different sizes.

Finally we handle the question about the strength of the messages being sent. We choose a lower limit l and an upper limit L. The strength is kept as a global variable in the system, and is initialized to 0. Each time an edge is updated, if the reduction in average bit-entropy is below l, the strength is increased and if it is above L it is decreased (unless it is 0). The limit l is there to make sure the algorithm is advancing, and the limit L is there to make sure no part of the graph tries to decide too quickly.

3.6 Random sparse systems

An early version of a message-passing algorithm has been tested on random sparse systems of equations that were a little overdefined. The results obtained were very promising and motivated the development of the methods.

The systems were generated using a parameter s, that controls how sparse the systems should be. When a system of equations in n variables is generated using s ($0 < s \leq 1$) to control the sparsity, it means that each equation in the system can contain at most sn variables. Before creating the equations, the solution the system should have, was chosen at random.

Each equation E in a system was generated as follows. First, the number of variables E should contain was picked at random from the interval $[2, sn]$. Let this number be n_E. Then, the number of configurations that should satisfy E was chosen at random from the interval $[2, 2^{n_E}]$. The configuration that matches the solution was put on the list first, and then the rest of the configurations were chosen at random and added to the list.

The number of equations generated for each system was $1.2n$, so the systems were slightly overdefined, and the constructed solution would be the only one with high probability.

	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5	0.6	0.7	0.8	0.9	1.0
10	-	-	-	-	-	-	199	-	197	199	198	199	196	193
15	-	-	-	-	199	-	200	-	200	199	196	187	182	176
20	-	-	-	-	199	-	195	-	196	193	184	177	-	-
25	-	-	195	-	197	-	199	-	195	185	183	-	-	-
30	-	-	195	-	199	-	195	-	189	169	-	-	-	-
40	-	192	194	197	198	198	190	188	-	-	-	-	-	-
50	190	197	197	199	199	191	-	-	-	-	-	-	-	-
60	195	196	199	198	193	-	-	-	-	-	-	-	-	-
70	185	198	198	197	-	-	-	-	-	-	-	-	-	-

Table 1: Number of times random systems were solved, out of 200 tries.

Several choices for the parameters s and n were chosen, and for each choice 200 systems were generated. The number of systems solved are summarised in Table 1. The rows are indexed with n, and the columns with s. When s and n are too low, many of the systems we get will be degenerate. When s and n are too high, it takes a long time to run the algorithm. For $n \leq 30$, s was increased in steps of 0.1. For larger n, this would give rather few tests for each n before the computations become too heavy, so for $n \geq 40$, s was increased in steps of 0.05. This explains why many of the cells in Table 1 only contain a dash.

As can be seen, almost all systems were successfully solved. One can also see that as the systems become less sparse, and contain more variables, the successrate drops.

4 Attacking DES

In the previous sections we have described how to construct a graph from a system of equations, and how to try to solve the system by using a message passing algorithm on the graph. Our motivation for studying sparse non-linear systems of equations comes from cryptanalysis, and the interest in algebraic attacks.

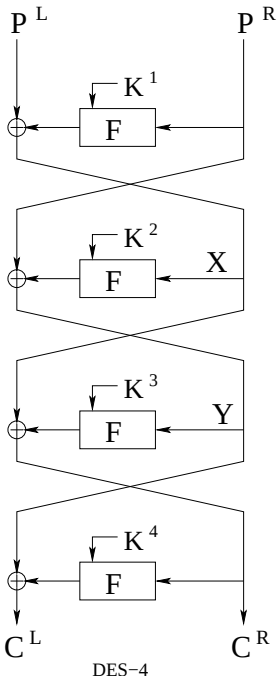
In this section we will first construct a sparse system of non-linear equations from the DES encryption algorithm, and then try our message passing algorithm to break the cipher by solving the system.

4.1 The DES equation system

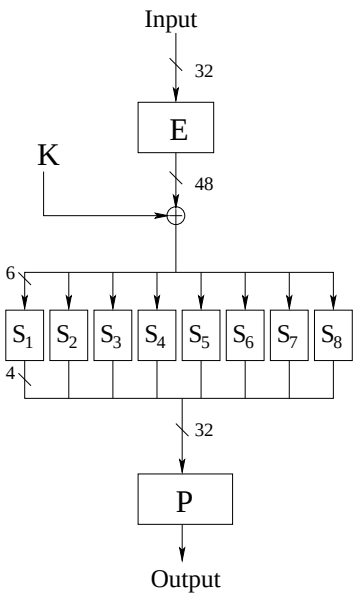
The DES is a Feistel cipher with 64-bit block and 56-bit key, details can be found in [6]. Here we will disregard the initial and final permutations since they do not have influence on the security. One important feature to note for our use is that each round key is only a permutation of 48 of the 56 user selected key bits, the user selected key bits are not mixed in the key schedule. To get a system of equations we must assume that we have some known plaintext/ciphertext pairs.

The round function of the cipher has four parts. First, the 32-bit input is expanded to 48 bits by repeating half of the input bits. Next, a 48-bit round key is xored onto the block. Then the block is split into eight 6-bit words, and each word is passed through an S-box producing a 4-bit output. The eight 4-bit outputs are joined to form a 32-bit word, and the bits in this word are permuted according to a fixed permutation P.

To get a system that is sparse enough, we give variable names to the 32 bits input to the round function in each round, except for the first and last rounds which have plaintext and ciphertext inputs. The setup for DES-4 is shown below, together with the details of the round function.



DES−4



Round function F

We can now describe the whole encryption algorithm as a system of equations. The straightforward way will be to let each output bit of each S-box be a function of the input bits. Since the four bits output from the same S-box share the same input, it is more efficient to group these four equations into one, and make one configuration list for the input and output variables of the S-box.

One equation in our system will thus have the form

$$B \oplus C = S_i[A \oplus K],$$

where B and C are two 4-bit strings, and A and K are two 6-bit strings. K represents the six bits of the round key input to S-box i, and A the six bits of the expanded input to the round that goes into S-box i. The strings B and C are both four bits from the variables (possibly plaintext or ciphertext) going into the rounds before and after the current round. The permutation P tells us which of these bits that xored together gives the output of S-box i.

For example, using the convention that bits are numbered from left to right starting with index 1, the equation for S-box 1 in round two will be

$$y_9 y_{17} y_{23} y_{31} \oplus p_{41} p_{49} p_{55} p_{63} = S_1[x_{32} x_1 x_2 x_3 x_4 x_5 \oplus k_{16} k_{19} k_{13} k_{26} k_3 k_7].$$

If we let $(p_{41} p_{49} p_{55} p_{63}) = (0001)$, the start of the configuration list for the symbol made from this equation will look like

k_{16}	k_{19}	k_{13}	k_{26}	k_3	k_7	x_{32}	x_1	x_2	x_3	x_4	x_5	y_9	y_{17}	y_{23}	y_{31}
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.

For each of the 2^{12} choices for the 12 input variables there is exactly one choice for the four y-variables that satisfies the equation, so the configuration list will have length 2^{12}. The general formula is that if an equation contains a variables, the configuration list in the corresponding symbol will have length 2^{a-4}.

Each equation that comes from the S-boxes in the first round of DES will have 10 variables: 6 key bits, 4 bits from the variables input to round two, the rest will be constants from the plaintext.

Likewise, the equations from the last round will also contain 10 variables. For DES-3, equations from the middle round will contain 12 variables: 6 bits from the input to the round, 6 key bits, the output of the S-box will be a constant xor of plaintext and ciphertext bits. For DES-r ($r \geq 4$), equations from S-boxes in the second round will contain 16 variables, 12 of them input to the S-box and four of them in the output, the rest of the bits in the output will come from the plaintext. Equations from round $r-1$ will also contain 16 variables. Equations from any of the rounds $3, \ldots, r-2$ will have 20 variables, all the strings A, B, C and K from the general equation will be variables.

This means that the largest configuration lists we will have to consider will have 2^{16} elements, and that the complexity of running a message-passing algorithm on the graph will only grow linearly with the number of rounds, after five rounds. This does not necessarily mean that the complexity of *solving* these systems only grows linearly after five rounds.

Building the system of equations as described above needs one known plaintext/ciphertext pair. If we have more than one pair, we can easily generalize the setup of the equation system. We build an equation system for each of the plaintext/ciphertext pairs we have. The variables representing the key bits should be the same among the systems, and the variables representing intermediate ciphertexts should be different. We can then build one graph from this larger system of equations, and run the message-passing algorithm on it.

4.2 Results for DES-3 and DES-4

The methods given in this paper have been implemented and tested on the equation systems from DES-3 and DES-4. Here we will present the results. The limits that were chosen for adjusting the strength of the messages are $l = 0.01$ and $L = 0.03$, and the strength was adjusted in steps of 0.001. It turns out that the performance of the algorithm depends on the plaintexts and the key. Some sets of plaintexts and keys give away the solution easier than others.

It is interesting to compare our results to other known attacks on DES-3 and DES-4. There is a meet-in-the-middle attack on DES-3 that requires two known plaintexts. This attack is a simple exhaustive search, where we are allowed to search for only 12 bits of the key at the time. A similar attack can not be extended to DES-4, but in [8], the authors write that a differential attack on DES-4 will succeed with 16 chosen plaintexts.

4.2.1 DES-3

There is a big difference in the hardness of solving the DES-3 systems using one plaintext versus the hardness using two plaintexts. One reason for this is that a lot of important symbols are made in a graph using two or more plaintexts, that are not made using only one. When using two plaintexts, the equation system above is constructed twice. The intermediate X-variables in each system should be different, but the key variables are shared. For each S-box in a round, there are two equations made using it, one for each plaintext. These equations are made into two symbols S and S'. In the graph construction, we will take the intersection of $v(S)$ and $v(S')$, and produce a new symbol from this set. This symbol will consist of all the six key variables that are going into this S-box. These symbols are made for each S-box in every round, and our results indicate they make a big difference.

When only one plaintext/ciphertext pair is used, the graph has 140 symbols and 522 edges. In this case the algorithm always fails. We may simplify the system by guessing on some of the key bits. The number of key bits that must be guessed in order to succeed with only one known plaintext is approximately ten. Running the algorithm ten times using ten different plaintexts, we only succeeded once when guessing on seven key bits. Running the algorithm ten times when guessing ten of the key bits resulted in eight successful attempts.

DES-3 can be broken with two known plaintexts. The graph constructed using two plaintext/ciphertext pairs has 303 symbols and 1280 edges. It is always possible to solve these systems by only sending messages with strength zero. They are completely solved, all messages have entropy 0, after about 1500 - 2000 messages have been sent, depending on the plaintexts and the key chosen. The running time on a normal PC with a 2.4 GHz processor is about one second.

	15	14	13	12	11	10	9
4	6	4	4	0	-	-	-
8	10	7	3	3	1	0	-
12	7	7	7	5	2	1	3
16	10	7	6	6	3	2	2

Table 2: Number of times the systems from DES-4 were solved, out of 10 tries. The columns indicate how many bits of the key was guessed, the rows indicate the number of known plaintext/ciphertext used.

4.2.2 DES-4

We have also succeded in solving systems from DES-4, when some key bits are guessed. Running some tests indicates there is a tradeoff between the number of bits guessed and the number of plaintexts used. The results are summarized in Table 2. The rows of the table specifies the number of plaintexts used and the columns are indexed by the number of key bits guessed. For each choice of number of plaintexts and number of bits guessed, we ran the algorithm ten times, on ten different sets of plaintexts. The numbers in the table show how many times the systems were solved.

As can be seen, when more plaintexts are used, the success rate drops more slowly when guessing on fewer key bits.

5 Conclusions and future work

The work presented in this paper is part of ongoing research, and is not finished. There are some natural questions that have not been answered in this paper. One of them is about using chosen plaintexts. If plaintexts (or ciphertexts) are chosen in the right way, not only key variables but also some variables from intermediate ciphertexts could be shared among subsystems coming from individual plaintexts. Another obvious question is how many key bits must be guessed in order to solve a system coming from five or more rounds of DES. During the tests, when we set some of the key bits it was always the first bits that were set to the correct values. There may be some sets of key bits that are more valuable to guess than $k_1, k_2, \ldots$ Finding the key bits that give away the most information about the solution of the whole system is also worth studying. There may also be other ways to combine messages and other ways to schedule the message passing which results in better algorithms.

These issues will be studied in the future. Meanwhile, we feel that the ideas and results presented in this paper already are interesting enough on their own. We are not aware of other methods of cryptanalysis based on iterative decoding algorithms, but the early results from this paper suggest it has some merit and should receive more study.

6 Acknowledgements

The author would like to thank Lars Knudsen, Thomas Jakobsen and Matthew Parker for fruitful discussions and helpfull hints during the development of the work in this paper.

References

[1] A. Shamir, J. Patarin, N. Courtois, A. Klimov. *Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations,* Eurocrypt 2000, LNCS 1807, Springer Verlag, pp. 392 - 407.

[2] N. Courtois, J. Pieprzyk. *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations,* ASIACRYPT 2002, LNCS 2501, Springer Verlag, pp. 267 - 287.

[3] N. Courtois, W. Meier. *Algebraic Attacks on Stream Ciphers with Linear Feedback,* EUROCRYPT 2003, LNCS 2656, Springer Verlag, pp. 345 - 359.

[4] E. Barkan, E. Biham. *In How Many Ways Can You Write Rijndael?,* ASIACRYPT 2002, LNCS 2501, Springer Verlag, pp. 160 - 175.

[5] S. Murphy, M. Robshaw. *Essential Algebraic Structure within the AES,* CRYPTO 2002, LNCS 2442, Springer Verlag, pp. 1 - 16.

[6] National Bureau of Standards. *Data Encryption Standard,* U.S. Department of Commerce, FIPS pub. 46, Jan. 1977. http://www.itl.nist.gov/fipspubs/fip46-2.htm

[7] J.C. Faugere,A. Joux. *Algebraic Cryptanalysis of Hidden Field Equation (HFE) Cryptosystems Using Gröbner Bases,* CRYPTO 2003, LNCS 2729, Springer Verlag, pp. 44 - 60.

[8] E. Biham, A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard,* Springer Verlag, 1993.