Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Evaluating Coppersmith's Criteria by way of SAT Solving

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Teresa Signati

Sessione I
Anno Accademico 2017/2018

To my family,
that believed in me
even when anyone didn't ...

Abstract

S-boxes are the non-linear part of DES cryptosystem. Along the years it has became clear that any kind of edit to the structure of DES S-boxes increases the probability of success of breaking the algorithm, which was very carefully designed. The reason why the S-boxes were built in this way was clarified by Coppersmith, years after the publication of the encryption algorithm.

The aim of this thesis is to investigate on Coppersmith's DES S-boxes design criteria and to evaluate them by way of SAT Solving, in order to analyze the performance of SAT-Solvers for different versions of DES algorithm, in which S-boxes respect only a sample of Coppersmith's design criteria. This aim is achieved thanks to the implementation of a Python tool: `DESBoxGen`.

The main challenge in the design of `DESBoxGen` is the one of finding a way to efficiently generating S-boxes satisfying certain criteria.

Introduction

Cryptography is an art, a science and a mathematical discipline, that studies techniques for making digital systems and information sharing secure. Its applications can be found everywhere.

One of the most widely known encryption schemes is the Data Encryption Standard, DES, which has been used worldwide for more than 20 years and for the same amount of time it has been studied by the whole world of cryptographers. Several attacks were attempted to break DES and most of them turned out not to be practicable due DES very careful design.

The motivations why attacks, like the differential cryptanalysis, are infeasible were clarified by Don Coppersmith [1], who explained the main design criteria respected in the implementation of DES S-boxes and P-box, years after the publication of the algorithm. Nowadays DES is not used anymore due its short key length but the well-designed structure of the algorithm, especially of the S-boxes, is undeniable.

In this thesis Coppersmith's criteria have been evaluated by way of SAT Solving through the implementation of an attack known as logical cryptanalysis. All these steps have been realized through the implementation of a Python tool: `DESBoxGen`.

The main challenge in the design of `DESBoxGen` is the efficient generation of S-boxes that respect only some of Coppersmith's criteria. Difficulties have been detected more in the generation phase than in the verification one, since naive algorithms almost never produce S-boxes in compliance with some criteria. To solve this problem, more complex implementations, like the

graph-based stochastic one, have been realized.

This thesis explains the path followed for generating S-boxes and `DESBoxGen`, mainly the encoding of DES variants into formulas and the evalutation of Coppersmith's design criteria. This work has the following structure:

- in the first chapter, SAT Solving is introduced, in order to explain its role in logical crypanalysis, the operations that have to be applied to formulas and the link between the search for a solution to a SAT problem and breaking an algorithm;
- in the second chapter, a brief overview on DES is given. More specifically, its architecture, the main attacks attempted on it and Coppersmith's explanation of the design choices behind implementation of the algorithm are described;
- in the third chapter, the generation of S-boxes is discussed with all the problems and the solutions detected;
- in the fourth chapter, all the implementation choices related to `DESBoxGen`'s structure are presented and the conversion of a cryptanalytic attack on DES into a SAT problem is explained;
- in the fifth chapter, the results of the tests previously generated through `DESBoxGen` are displayed in order to evaluate Coppersmith's criteria using three different SAT-Solvers: Picosat, CryptoMiniSat and Lingeling.

Contents

Chapter 1

SAT Solving in Cryptanalysis

The problem of deciding if a propositional logic formula is satisfiable, i.e. if there is an assignment of truth values to boolean variables such that the formula is true, is the prototypical NP-complete problem[1]. On the following, it will be referred to as SAT.

Despite the hardness of the problem, some of SAT instances can be solved in a reasonable amount of time[3] through SAT-Solvers, that given a propositional formula look for a variable assignment such that the formula evaluates to true. Many computational problems can be encoded as a SAT one, so finding a solution for the logical problem corresponds to solving the initial one.

In this thesis Logical Cryptanalysis [4] against DES and its variants has been implemented by implementing cryptanalysis as a SAT problem so that finding a model for this formula is equivalent to finding the encryption key in a cryptanalitic attack.

[1] NP-completeness of SAT, the satisfiability problem, was discovered by Stephen Cook and Leonid Levin in '70s; they (independently) noticed the relation between complexity of certain problems and that of the entire class. The demonstration of Cook-Levin theorem is reported in [2].

1.1 SAT-Solvers Input: DIMACS CNF

The majority of SAT-Solvers takes as input a propositional logic formula in a format called DIMACS CNF, where CNF stands, as usual, for conjunctive normal form.

1.1.1 Conjunctive and Disjunctive Normal Form

A CNF formula [5] is a conjunction of clauses,

$$F = \bigwedge_{i=1}^{len(F)} C_i$$

where each clause must be a disjunction of literals,

$$C_i = \bigvee_{j=1}^{len(C_i)} L_j$$

and a literal is either a boolean variable, or its negation,

$$L_j = A \mid \neg A$$

Please notice that the presence of other connectives different from $\wedge, \vee, \neg$ is not allowed. A similar form is the DNF, where DNF means disjunctive normal form, i.e. the formula is a disjunction of clauses, where each clause is a conjunction of literals. CNF formulas are the most common input to SAT-Solvers because they are easy to evaluate: only a literal for each clause has to be true so that the complete formula can be satisfied.

1.1.2 DIMACS CNF format details

The DIMACS CNF format is characterized by:

- some (optional) comments lines beginning with the character c
- a line indicating the format (CNF), followed by the number of variables appearing in the file, followed by the number of clauses contained in the file.

```
p cnf NUMBER_OF_VARS NUMBER_OF_CLAUSES
```

- the lines representing the clauses, in which each variable is represented by a number bigger than 0, if the variable is negated, it's preceded by the minus sign. 0 ends the clause.

For example, let

$$(\neg A \wedge D) \vee (A \wedge \neg B \wedge \neg C \wedge \neg D) \vee (B \wedge C \wedge \neg D)$$

be a propositional logic formula, its representation in DIMACS CNF format could be:

```
c Example of the previous formula in DIMACS CNF format
p cnf 4 3
-1 4 0
1 -2 -3 -4 0
2 3 -4 0
```

1.2 CNF Naive Conversion

To get the DIMACS file it's necessary to have a formula in conjunctive normal form. Each logic formula F can be transformed in CNF (or DNF) format, F^C, by applying standard logic rules [6]:

- Deletion of all operators other than $\vee, \wedge, \neg$ through the application of the following equivalences:
  - $(A \implies B) \equiv \neg A \vee B$
  - $(A \iff B) \equiv (A \implies B) \vee (B \implies A) \equiv (\neg A \vee B) \wedge (\neg B \vee A)$
  - $(A \oplus B) \equiv A \iff \neg B \equiv (\neg A \wedge B) \vee (\neg B \wedge A)$
- Pushing negation in front of single variables, so that only literals can be negated, applying the following equivalences:

  - Double negation law: $\neg\neg A \equiv A$
  - De Morgan's laws:
    $\neg\ (A \wedge B) \equiv \neg A \vee \neg B$
    $\neg\ (A \vee B) \equiv \neg A \wedge \neg B$

- Application of distributive laws such as Or distributive law in order to get conjunction of disjunctions:
  - Or distributive law: $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

In some cases (the worst one involving standard transformation is the conversion of a DNF formula to a CNF one), the application of standard rules can produce as side effect an exponential blowup in the size of the formula, that increases the hardness of the problem. The so-called Tseitin's encoding prevents this.

1.3 Tseitin's Encoding

Tseitin's transformation [7] takes as input a propositional logic formula and converts it into the CNF format, so it can be passed as input to a SAT-Solver. This transformation introduces several auxiliary variables, in order to have formula whose size has grown linearly relative to the input of the circuit, and not exponentially like applying the naive CNF conversion illustrated in Section 1.2. Even if Tseitin encoding introduces new variables, so the two formulas F and F^T are different, the conversion preserves their satisfiability, i.e. F is satisfiable if and only if F^T is satisfiable, thanks to the addition of a series of constraints on these auxiliary variables.

1.3.1 The Tseitin's Encoding, in Theory

Consider the following formula:

$$F = (A \wedge B) \vee (\neg C \wedge D)$$

F has four sub-formulas:

1. $\neg C$
2. $(\neg C \wedge D)$
3. $(A \wedge B)$
4. $(A \wedge B) \vee (\neg C \wedge D)$

Tseitin encoding introduces a new variable for each sub-formula:

1. $aux_1 \iff \neg C$
2. $aux_2 \iff (aux_1 \wedge D)$
3. $aux_3 \iff (A \wedge B)$
4. $aux_4 \iff aux_3 \vee aux_2$

To preserve the satisfiability of the formula, the conjunct of all the constraints is taken as the result formula:

$$F^T = aux_4 \wedge (aux_4 \iff aux_3 \vee aux_2) \wedge (aux_3 \iff (A \wedge B)) \wedge$$

$$\wedge (aux_2 \iff (aux_1 \wedge D)) \wedge (aux_1 \iff \neg C)$$

At this point it's sufficient to convert each constraint / substitution to CNF, applying the standard conversion algorithm (explained in 1.2), especially the Iff equivalence

$$A \iff B \equiv (\neg A \vee B) \wedge (\neg B \vee A)$$

In this way, the F^T of the example becomes:

$$F^T = aux_4 \wedge (\neg aux_4 \vee aux_3 \vee aux_2) \wedge (aux_4 \vee \neg(aux_3 \vee aux_2)) \wedge$$

$$\wedge (\neg aux_3 \vee (A \wedge B)) \wedge (aux_3 \vee \neg(A \wedge B)) \wedge (\neg aux_2 \vee (aux_1 \wedge D)) \wedge$$

$$\wedge (aux_2 \vee \neg(aux_1 \wedge D)) \wedge (\neg aux_1 \vee \neg C) \wedge (aux_1 \vee C)$$

1.3.2 Tseitin's Encoding in PyEDA and DESBoxGen

DESBoxGen makes use of PyEDA [8], a Python library for electronic design automation, that implements the representation of logic expressions, Tseitin's encoding, and the conversion of a CNF formula into DIMACS CNF format. PyEDA `tseitin()` method can be applied to expressions by specifying as optional parameter only the name that the auxiliary variables should have.

However this implementations doesn't allow (at the moment) an explicit substitution of an expression with a new variable, so it's possible to use this new variable in further computations to avoid that the formula has an exponentially large size when a big formula is converted to a CNF one. To work around this limitation, DESBoxGen implements a subroutine that realizes the replacement explicitly, applying PyEDA's `tseitin()` method and the appropriate logical equivalences so that the formula given as input is satisfiable if and only if the new variable is satisfiable.

The application of DESBoxGen `replace()` returns the new variable, that will replace the complex formula and will be used in all the further computations, and the binding, a CNF expression obtained through PyEDA `tseitin()` method, that will be used in the end to ensure the logic equivalence between the variable and the complex formula. The details of the use of the `replace()` function will be explained in the following chapters (see 4.4 for more details).

1.4 Truth table to CNF and DNF

In the implementation of DESBoxGen, it is necessary to recover the propositional logic formula associated to a circuit, the motivation of this will be explained in the forthcoming chapters. It's possible to derive the formula in CNF or DNF format directly from the truth table of the circuit.

```
def replace(formula, b_name, b_index, aux_name='aux'):
    """
    Implementation of "sharing":
    define a "binder" variable as the alias of a more
    complex formula,in order to use it
    in further operations.

    :param formula: the formula that must be replaced
    :param b_name: the name of the "binder" variable
    :param b_index: the index of the "binder" variable
    :param aux_name: the name of aux variables, to be used
    in the representation of the binding
    between formula and the "binder" variable
    :return: the "binder" variable and a formula that
    represents the "binding"
    """
    b = exprvar(b_name, b_index)
    binding = And(Or(Not(b), formula), Or(b, Not(formula)))
    binding = binding.tseitin(auxvarname=aux_name)
    return b, binding
```

Consider, for example, the following truth table associated to a circuit:

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Each row can be represented as the conjunction of the literals in the same row, in this case it is

$$\bigwedge_{i=0}^{numOfVars} L_j$$

The DNF representation of the formula will be obtained by the disjunction of the rows that satisfy F, because suffices an assignment of truth that satisfies the formula:

$$\bigvee_{j=0}^{2^{numOfVars}} row_j.(F_j = 1)$$

That in the example becomes:

$$F \equiv row_0 \vee row_3 \vee row_7$$

$$F \equiv (\neg A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge C) \vee (A \wedge B \wedge C)$$

Similarly it's possible to get a logically equivalent CNF representation of the formula by the conjunction of the negation of the rows that don't satisfy the formula:

$$\bigwedge_{j=0}^{2^{numOfVars}} \neg row_j.(F_j = 0)$$

In order to satisfy the formula, no one of the rows (corresponding to an assignment to the variables) that gives 0 as output can be chosen.

Going on with the example:

$$F \equiv \neg row_1 \wedge \neg row_2 \wedge \neg row_4 \wedge \neg row_5 \wedge \neg row_6$$

$$F \equiv \neg(\neg A \wedge \neg B \wedge C) \wedge \neg(\neg A \wedge B \wedge \neg C) \wedge \neg(A \wedge \neg B \wedge \neg C) \wedge$$

$$\wedge \neg(A \wedge \neg B \wedge C) \wedge \neg(A \wedge B \wedge \neg C)$$

$$F \equiv (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee B \vee C) \wedge (\neg A \vee B \vee \neg C) \wedge (\neg A \vee \neg B \vee C)$$

To get the formula of a logic circuit, the Circuit class of DESBoxGen implements `truth_table2formula()`, a function that takes as input the list of variables for the required encoded formula.

1.5 The Importance of Finding a Model

SAT Solving is used in DESBoxGen because it allows to find a model for a formula, where a model corresponds to a satisfying interpretation for the formula. In Logical Cryptanalysis, finding a model for a formula encoding a circuit like DES is equivalent to finding a key with a known-plaintext attack. These operations are simulated in DESBoxGen using SAT Solving to evaluate the strength of DES variants against SAT-Solvers and after it, it is possible to analyze the results in order to find differences in SAT-Solvers behaviour through the analysis of ciphers with different levels of complexity.

```
    def truth_table2formula(self, f_vars):
        """
        This method computes formulas representing each
        output bit on the basis of the input ones.

        :param f_vars: exprvars variables, one for each
        input bit
        :return: a list of formulas describing the truth
        value of each output bit
        """
        formula = [expr(0)] * self.m
        unconsidered_bit = [True] * 4
        for row in self.truth_table:
            for out_pos in range(self.m):
                if row[self.n + out_pos] == '1':
                    # compute the expression corresponding
                    to the row
                    for in_pos in range(self.n):
                        bit = f_vars[in_pos] if row[in_pos]
                         == '1' else Not(f_vars[in_pos])
                        if in_pos == 0:
                            ex = bit
                        else:
                            ex = And(ex, bit)
                    if unconsidered_bit[out_pos]:
                        formula[out_pos] = ex
                        unconsidered_bit[out_pos] = False
                    else:
                        formula[out_pos] = Or(formula[
                        out_pos], ex)
        return formula
```

Chapter 2

The Data Encryption Standard

The Data Encryption Standard, better known as DES, is a well-designed block cipher of great historical importance, which has been used worldwide for more than 20 years, even if now it is considered insecure because of its short key length. Before seeing DESBoxGen variants and their implementation for Logical Cryptanalysis, in this chapter there will be a brief overview on DES, its historical importance and its main components.

2.1 History

The DES symmetric-key algorithm was developed in the '70s by an IBM team working on cryptography (which involved also Horst Feistel) under the original name of Lucifer, and it was proposed to NBS (the National Bureau of Standards, now the National Institute of Standards and Technology (NIST)) that was looking for a standard cryptographic algorithm that should have been included in a program to protect computer and communication data [11]. This algorithm should have been cheaper and readily available and the compliance with all these features brought DES to success: despite the hardness of the algorithm, it used only simple logical operations that could be easily implemented in hardware.

NSA (National Security Agency) helped NBS in the evaluation of Lucifer's

security and suitability as standard and requested some substantial modifications to the original version of the algorithm, particularly a reduction of the key length from 128 bits to 56 bits. The reasons why NSA modified some components in the original algorithm became clear only in '90s but, despite the doubts, DES became a standard and the publication of its details allowed a software implementation, that led all the cryptographers to study the "secure" encryption algorithm.

Several details about DES history can be found in Federal Processing Standards Publications, for example in [13] there are all the reasons why DES should have been used and its suitability for Federal standards.
An encryption algorithm must satisfy the following requirements in order to be acceptable as a Federal standard:

1. It must provide a high level of security.
2. It must be completely specified and easy to understand.
3. The security provided by the algorithm must not be based upon the secrecy of the algorithm.
4. It must be available to all users and suppliers.
5. It must be adaptable for use in diverse applications.
6. It must be economical to implement in electronic devices and be efficient to use.
7. It must be amenable to validation.
8. It must be exportable.

The algorithm described in FIPS PUB 46 satisfies all these requirements.

In 1987, NSA noticed that the algorithm would have been soon broken but the diffused disappointment at NSA's announcement of not recertification of the standard and the absence of a valid alternative to DES encryption algorithm led to the reaffirmation of DES as a standard until 1992.

Between 1993 and 1994, at the dawn of the discovery of differential cryptanalysis, some motivations behind DES implementation choices became finally clear: the method of differential cryptanalysis, published by Biham and Shamir [9], was just discovered and it reported the first theoretical attack with less complexity than brute force but it required an unrealistic 2^{47} chosen plaintexts to succeed. Differential cryptanalysis was already known when DES was designed and its design criteria contributed in defeating this kind of attack. Despite its strength against differential cryptanalysis, the technological progress and advances in hardware blew down DES: its short key length was its main weakness and this allowed brute-force attacks to succeed on DES even in less than a day. DES was reconfirmed as standard under the form of Triple DES, until the publication of the Advanced Encryption Standard in 2001.

2.2 Structure

DES is a block cipher that works with 64-bit blocks, that implements a symmetric algorithm using a 16-round Feistel Network. As declared in [13]

> The DES algorithm is mathematically a one-to-one mapping of the 2" possible input blocks onto all 2^{64} possible output blocks. Since there are 2^{56} possible active keys, there are 2^{56} possible mappings. Selecting one key selects one of the mappings.

The plaintext is permuted through an IP, Initial Permutation, followed by the application of the key-based Feistel Network, and in the end it's applied a final 32-bit swap and a FP, Final Permutation, that is the inverse of the IP.

2.2.1 Symmetric Key Cipher

DES represents a symmetric encryption model that is characterized by

- an encryption algorithm, E, that performs various substitutions and transformations on the plaintext;
- a secret key given as input to E. Transformations performed by E on the plaintext depend on the key;
- a decryption algorithm, D, that is E run in reverse.

Symmetric key based communications security depends on the key, that is the only private element, kept by the sender and the receiver, that makes unintelligible encrypted messages readable. Given a message P and the private encryption key K, the encryption algorithm computes the ciphertext

$$C = E_K(P)$$

The receiver, who knows the secret key, can invert the transformation

$$P = D_K(C)$$

There are two different ways of processing the input: the stream cipher and the block cipher; DES represents the last one so it processes the input one block at time, producing an output block for each input block.

2.2.2 Feistel Network

A Feistel Network is a common pattern adopted in the construction of block ciphers, having the advantage that encryption and decryption are almost identical, they require only a reversed key schedule. As explained in [12]:

> A Feistel network thus gives a way to construct an invertible function from non-invertible components.

Its structure is composed of several rounds, which can inner use non invertible functions. In each round, a keyed round function is applied. If the block length of the cipher is l bits, the round function takes as input a $l/2$-bit string and a sub-key k_i and returns a $l/2$ bit string. Sub-keys are generated

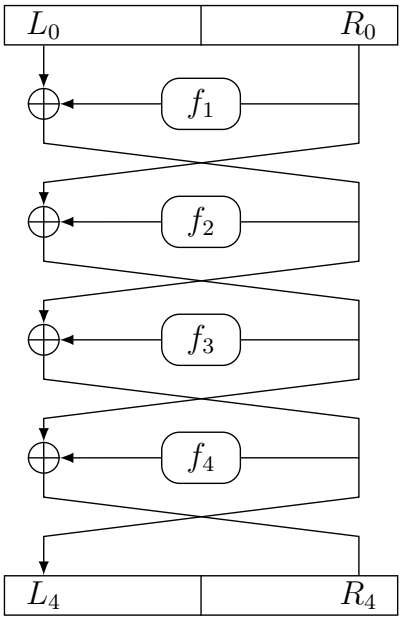


Figure 2.1: Example of a 4-round Feistel Network

starting from the master key. The input of the i-th round is divided in two halves, L_{i-1} and R_{i-1} of length $l/2$, and the respective output is computed as follows.

$$L_i = R_{i-1}$$

$$R_i = L_{i-1} \oplus f_i(R_{i-1})$$

A Feistel network is invertible regardless of the round functions. Given the output (L_i, R_i) the input can be computed as follows:

$$L_{i-1} = R_i \oplus f_i(R_{i-1})$$

$$R_{i-1} = L_i$$

without the inversion of the round function. As Katz and Lindell say in [12]:

> Let F be a keyed function defined by a Feistel network. Then regardless of the round functions $\{\hat{f}_i\}$ and of the number of rounds, F_k is an efficiently invertible permutation for all k.

2.2.3 Shannon's Confusion-Diffusion Paradigm

A block cipher must behave like a random permutation as described by Shannon's confusion-diffusion paradigm [10]. The application of the multiple rounds of the Feistel network ensures an avalanche effect, i.e. small changes in the input must affect all the yield. It's possible to make the behaviour of DES similar to a random permutation thanks to the confusion/diffusion steps corresponding to a round, so that a single input bit can potentially affect all the bits of the output. As Shannon said above confusion and diffusion:

> In the method of diffusion the statistical structure of M which leads to its redundancy is "dissipated" into long range statistics—i.e., into statistical structure involving long combinations of letters in the cryptogram. The effect here is that the enemy must intercept a tremendous amount of material to tie down this structure, since the structure is evident only in blocks of very small individual probability. Furthermore, even when he has sufficient material, the analytical work required is much greater since the redundancy has been diffused over a large number of individual statistics... The method of confusion is to make the relation between the simple statistics of E and the simple description of K a very complex and involved one.

Substitutions, permutations and other components of DES are an implementation of the confusion-diffusion paradigm.

2.2.4 DES Round Function

DES round function takes as input the sub-key and 32 bits corresponding to the right half of the input.

$$f(k_i, R_{i-1})$$

The input is expanded through an Expansion-box to a 48-bit value, that is XORed with the 48-bit sub-key, and the result is the input of the S-boxes.

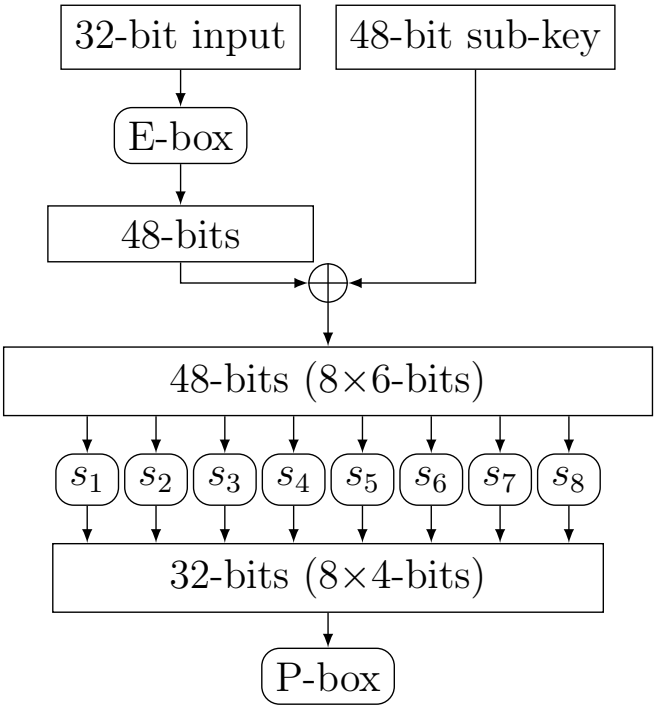


Figure 2.2: Structure of a round in DES

This value is divided into 8 parts, that become the input of the S-boxes, $S_1, ..., S_8$. An S-box takes as input 6 bits and returns 4 bits. The concatenation of the output of the S-boxes gives a 32 bit result. The application of the Permutation-box to the 32-bit result gives the final output of the round function. The Expansion-box and the Permutation-box are a linear component of DES round function and both of them perform diffusion [11], instead the S-boxes perform confusion.

S-boxes

The S-boxes are the only nonlinear component of DES round function. The substitution choices should not be chosen randomly but should be carefully designed in order to grant the avalanche effect in the application of multiple rounds of Feistel network according to Shannon's confusion and diffusion paradigm. Their official description is given as a lookup table so that given

the 6-bit input the 4-bit output is identified by selecting the row using the 2 outer bits of the input and the column using the 4 inner ones.

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

Table 2.1: DES S-box 1

In '70s the values associated to S-boxes created several suspicions about the presence of trapdoors in DES. The motivations behind the choices became clear only in '90s after the Biham and Shamir's discovery of differential cryptanalysis and Coppersmith's declaration about DES design criteria that revealed the careful S-box design criteria (see [1]).

2.2.5 Key Transformation

The application of the round function depends from R_i, the right part of the input text of the i-th round, and K_i, the round-sub-key. Sub-keys are generated starting from the 64-bit master-key, reduced to 56-bit key by ignoring eighth bits that can be used as parity bits. The 48-bit sub-keys are generated from the 56-bit key by splitting the input in two 28-bit halves that are left-shifted by one or two bits, depending on the round.

Key space and weak keys

The aim of an attacker is to discover the key so that if he knows the algorithm, all the encrypted messages can be decrypted. The probability to succeed depends also on the key space, that, in the case of DES, is near 2^{56}. In fact, as declared in [13], there are some weak and semiweak keys to avoid, because they make the same subkey to be generated in more than one round. The application of one of the 4 weak keys produces 16 identical subkeys , so

decryption is identical to encryption

$$E_K(E_K(P)) = P$$

The 4 weak keys in hex are:

`0x0101010101010101`

`0xFEFEFEFEFEFEFEFE`

`0xE0E0E0E0F1F1F1F1`

`0x1F1F1F1F0E0E0E0E`

Moreover DES has also semi-weak keys, such that for each key K there exists a key K' for which encryption with K is identical to decryption with K' and vice versa. These keys are called dual keys [13]

`0x011F011F010E010E`	`0x1F011F010E010E01`
`0x01E001E001F101F1`	`0xE001E001F101F101`
`0x01FE01FE01FE01FE`	`0xFE01FE01FE01FE01`
`0x1FE01FE00EF10EF1`	`0xE01FE01FF10EF10E`
`0x1FFE1FFE0EFE0EFE`	`0xFE1FFE1FFE0EFE0E`
`0xE0FEE0FEF1FEF1FE`	`0xFEE0FEE0FEF1FEF1`

By applying dual keys it results that

$$E_K(E_{K'}(P)) = P$$

$$D_K(D_{K'}(P)) = P$$

There are also 48 keys, the possibly weak keys listed in [14], that produce only four distinct subkeys (instead of 16) and should be avoided. In total the keys that should be avoided are 64 (4 weak keys, 6 pairs of semi-weak keys and 48 possibily weak keys) out of the 2^{56}.

2.3 Attacks on DES

Several attacks are possible against a cipher [12], which are listed below in order of increasing power of the attacker:

- **Ciphertext-only attack**: the adversary attempts to determine some information about the plaintext and the key observing only the ciphertext.
- **Known-plaintext attack**: the adversary is able to learn one or more $(plaintext, ciphertext)$ pairs generated using some key. The aim of the attacker is to recover the key in order to get further informations about the underlying plaintext of some ciphertext encrypted using the same key.
- **Chosen-plaintext attack**: the adversary can obtain plaintext/ciphertext pairs for plaintexts of its choice, in order to recover the key as in the previous case.
- **Chosen-ciphertext attack**: the adversary is able to obtain also informations about the decryption of ciphertexts of its choice, in order to recover the key for the same reasons of the previous cases.

The most practical attack on DES is still a brute-force attack, even if there are different theoretical cryptanalytic attacks, requiring an unrealistic number of couples $(plaintext, ciphertext)$.

2.3.1 Brute-force Attacks

A brute-force attack tries every key in key-space, and despite its naive approach, it can succeed because of the shortness of DES key length, that was reduced from 128 bits to 56 bits, after NSA involvement in DES implementation. DES vulnerability became definitely clear in 1990s. In 1997 a message encrypted with DES was broken for the first time and the following attacks required less and less time.

2.3.2 Differential Cryptanalysis

Differential cryptanalysis [15] is one of the theoretical attacks that can break DES with less complexity than a brute-force attack. This one is a chosen-plaintext attack, so the analysis of differences in the ciphertexts allows to formulate assumptions on the cipher key.

> The rationale behind differential cryptanalysis is to observe the behavior of pairs of text blocks evolving along each round of the cipher, instead of observing the evolution of a single text block.

Although the reduced complexity of this attack, it is only theoretical because it requires 2^{47} chosen plaintexts to succeed. After the rediscovery of differential cryptanalysis in 1991 by Eli Biham and Adi Shamir [9], Coppersmith revealed that it was known to both IBM and NSA, and DES design criteria were defined to increase the resistance against this attack, that's why it is not practicable (the details of design criteria will be explained in 2.4). Permutation and S-boxes modifications reflect the role of the need of strengthen DES against differential cryptanalysis as said in [15]

> Differential cryptanalysis of an eight-round LUCIFER algorithm requires only 256 chosen plaintexts, whereas an attack on an eight-round version of DES requires 2^{14} chosen plaintexts.

2.3.3 Linear Cryptanalysis

Another kind of attack attempted on DES is the linear cryptanalysis, discovered by Mitsuru Matsui in 1993, that uses 2^{43} known plaintexts, as well it is an infeasible attack on DES, even if DES should not be meant to be resistant to this attack. Variants of this attack with reduction in data complexity require from 2^{39} up to 2^{41} chosen-plaintexts.

2.4 Coppersmith's Criteria

> Differential cryptanalysis was well known, however, to the IBM team that designed DES, as early as 1974. Knowledge of this technique and the necessity to strengthen DES against this attack using it, played a large part in the design of the S-boxes and the permutation P.

The previous and the following are declarations by Don Coppersmith in [1], that's why a differential cryptanalysis attack against DES requires enormous amount of chosen plaintext.

> The IBM team knew about differential cryptanalysis but did not publish any reference to it.

In [1] Coppersmith, after an explanation of DES and differential cryptanalysis, listed the relevant criteria for the S-boxes and the permutation P, which were satisfied by the design of DES:

S-1 Each S-box has six bits of input and four bits of output. (This was the largest size that we could accommodate and still fit all of DES onto a single chip in 1974 technology.)

S-2 No output bit of an S-box should be too close to a linear function of the input bits. (That is, if we select any output bit position and any subset of the six input bit positions, the fraction of inputs for which this output bit equals the XOR of these input bits should not be close to 0 or 1, but rather should be near to 1/2)

S-3 If we fix the leftmost and the rightmost input bits of the S-box and vary the four middle bits, each possible 4-bit output is attained exactly once as the middle four input bits range over their 16 range possibilities.

S-4 If two inputs to an S-box differ in exactly one bit, the outputs must differ in at least two bits. (That is, if $|\Delta I_{i,j}| = 1$, then $|\Delta O_{i,j}| \geq 2$, where $|x|$ is the number of 1-bits in the quantity x.)

S-5 If two inputs to an S-box differ in the two middle bits exactly, the outputs must differ in at least two bits. (If $\Delta I_{i,j} = 001100$, then $|\Delta O_{i,j}| \geq 2$.)

S-6 If two inputs to an S-box differ in their first two bits and are identical in their last two bits, the two outputs must not be the same. (If $\Delta I_{i,j} = 11xy00$, where x and y are arbitrary bits, then $\Delta|O_{i,j}| \neq 0$)

S-7 For any nonzero 6-bit difference between inputs, $\Delta I_{i,j}$, no more than eight of the 32 pairs of inputs exhibiting $\Delta I_{i,j}$ may result in the same output difference $\Delta O_{i,j}$

S-8 Similar to (S-7), but with stronger restrictions in the case of $\Delta O_{i,j} = 0$, for the case of three active S-boxes on round i.

P-1 The four output bits from each S-box at round i are distributed so that two of them affect (provide input for) "middle bits" of S-boxes at round $i + 1$ (the two middle bits of input to an S-box, not shared with adjacent S-boxes), and the other two affect "end bits" (the two left-hand bits or the two right-hand bits, which are shared with adjacent S-boxes.)

P-2 The four output bits from each S-box affect six different S-boxes; no two affect the same S-box. (Remember that each "end bit" affects two adjacent S-boxes.)

P-3 For two S-boxes j, k, if an output bit from S_j affects a middle bit of S_k, then an output bit from S_k cannot affect a middle

bit of S_j. This implies that in the case $j = k$, an output bit from S_j must not affect a middle bit of S_j.

Chapter 3

Generating S-boxes

In order to analyze the behaviour of DES variants, it is necessary to build different S-boxes that will respect only some of Coppersmith's criteria. In this chapter the implementation of the construction and the verification for a sample of S-boxes design criteria will be discussed.

The generation has to be done in a random way in order to consider all the possible S-boxes in compliance with some criteria. At the moment DESBoxGen allows, for every i, the generation of S-boxes satisfying $s_1, ..., s_i$ until i equal to six.

Different stochastic generation algorithms will be discussed, since naive functions almost never satisfy some of Coppersmith's requirements.

3.1 S-box as Subclass of Circuit

The class `Sbox` is a subclass of the `Circuit` class that allows the representation of a logic circuit, characterized by a defined fixed number of input (`n_in`) and output bits (`m_out`) that are necessary for the construction of the truth table.

```
class Circuit:
    def __init__(self, n_in, m_out, io={}, others='z'):
```

The outputs of the circuit can be chosen either randomly or they can be specified during the creation of an instance of this class:

- The defined cases are realized through the `io` dictionary, where inputs are the keys and their outputs are the related values.
- In the undefined ones, i.e. if an input does not appear in the `io` dictionary, the output can be chosen in three different ways according to the value of the parameter `others`: it can be set to zero('`z`'), one ('`o`') or random('`r`').

```
class Sbox(Circuit):
    def __init__(self, n_in=6, m_out=4, lt=[], io={},
    others='r', min_dc=0):
```

`Sbox` attributes are:

`n`	integer the size of the input (inherited from the `Circuit`)
`m`	integer the size of the output (inherited from the `Circuit`)
`truth_table`	matrix (i.e. list of lists) the description of the S-box (inherited from the `Circuit`)
`dc`	dictionary its keys are the criteria, its output are a boolean (`True` if the criterion is complied with, `False` otherwise)

Objects of the class `Sbox` will be used in DES architecture in two different modes:

- the standard one, in which all the inputs (i.e. the key and the plaintext) are known;

- the one useful for cryptanalysis, in which one of the input can be unknown.

In the first case it's necessary to get the output for a certain known input: this feature is realized through the method `input2output` of the `Circuit` class.

```
def input2output(self, i):
    """
    This method returns the output for a required input
    :param i: a list representing a binary input of the
     circuit
    :return: the list corresponding to the binary
    output for the input given as parameter
    """
    return self.truth_table[int(''.join(i), 2)][self.n
    :]
```

In the second case the input value of the S-box could be unknown, so the function `truth_table2formula()` implemented by the `Circuit` class is used.

```
def truth_table2formula(self, f_vars):
    """
    This method computes formulas representing each
    output bit on the basis of the input ones.

    :param f_vars: exprvars variables, one for each
    input bit
    :return: a list of formulas describing the truth
    value of each output bit
    """
    formula = [expr(0)] * self.m
    unconsidered_bit = [True] * 4
    for row in self.truth_table:
        for out_pos in range(self.m):
            if row[self.n + out_pos] == '1':
```

```
                    # compute the expression corresponding
                    to the row
                    for in_pos in range(self.n):
                        bit = f_vars[in_pos] if row[in_pos]
                         == '1' else Not(f_vars[in_pos])
                        if in_pos == 0:
                            ex = bit
                        else:
                            ex = And(ex, bit)
                    if unconsidered_bit[out_pos]:
                        formula[out_pos] = ex
                        unconsidered_bit[out_pos] = False
                    else:
                        formula[out_pos] = Or(formula[
                        out_pos], ex)
        return formula
```

In brief, each output bit is the `Or` of all the inputs, that make it equal to `1`. Each input corresponds to the `And` of its bits encoded by formula: if the value in the truth table is true, the formula is considered as it is, it's negated otherwise.

`Sbox` extends the `Circuit` superclass with methods for the construction and the verification of S-boxes design criteria until (S-6). The results of the verification routine are stored into the dictionary `dc`. At the creation of the S-box the minimum design criterion (that has to be verified) can be specified. This is implemented through the generation of an `io` dictionary, that will be passed as input to the `Circuit`

Also DES standard S-boxes can be reused, by passing the lookup table `lt` of the specification as parameter, that will be converted into an `io` dictionary through the method `_sbox_std_dict()`, so its related `Circuit` can be generated.

```
    @staticmethod
    def _sbox_std_dict(s):
        """
        This function derives the output for each possible
        input of an S-box from its lookup table

        :param s: a matrix representing the lookup table of
         a standard S-box
        :return: the dictionary in which the keys are the
        input and the values are the corresponding output
        for the given S-box
        """
        d = {}
        for i in range(int('1' * 6, 2) + 1):
            bin_in = format(i, '06b')
            sr = int(bin_in[0] + bin_in[5], 2)
            sc = int(''.join(bin_in[1:5]), 2)
            d[bin_in] = format(s[sr][sc], '04b')
        return d
```

3.2 Analysis of S-1 and its Implementation

Remember, from Section 2.4, that criterion (S-1) asks that

> Each S-box has six bits of input and four bits of output.

Of course, this is a minimal requirement that can be easily verified and set.

Verification

(S-1) can be easily checked thanks to a boolean expression on the size of input and output.

```
self.dc['1'] = (self.n == 6 and self.m == 4)
```

Generation

The construction of (S-1) is easy, because it's sufficient to specify the input size equal to 6 and the output one equal to 4, that are set by the class `Sbox` by default, such that it results:

$$S : \{0,1\}^6 \to \{0,1\}^4$$

The other criteria are checked only if the first one is satisfied.

3.3 Analysis of S-2 and its Implementation

Remember from Section 2.4, that criterion (S-2) asks that

> No output bit of an S-box should be too close to a linear function of the input bits. (That is, if we select any output bit position and any subset of the six input bit positions, the fraction of inputs for which this output bit equals the XOR of these input bits should not be close to 0 or 1, but should rather be near to 1/2)

It's hard to define a construction method for (S-2) because of the hypothesis of non linearity, that implies a random behaviour of the S-box.

Verification

A verification method has been implemented, that computes the powerset of the six possible input positions, in order to get each possible subset of input indexes position. The method `_check_dc2()` checks if an output bit in a certain position equals to the `Xor` of a given subset of input bits. This check is done for each possible row of the truth table, output position and subset of input indexes. If the number of rows for which the output bit equals the `Xor` of the subset of input ones is lower than the 20% of the number of rows of the truth table, or it is greater than the 80%, the second criterion isn't respected.

Generation

DESBoxGen does not implement a construction of S-boxes that respect (S-2) that can be summarized as a random assignment of an output to a certain input. In order to get S-boxes that comply with (S-2), it will be necessary to generate completely random S-boxes, and *then* to verify whether (S-2) is respected, getting rid of those which do not.

The reason behind the absence of the implementation of (S-2) takes into account the generation of S-boxes that should respect criteria with a more complex implementation method, as the fifth one.

3.4 Analysis of S-3 and its Implementation

Remember, from Section 2.4, that criterion (S-3) asks that

> If we fix the leftmost and the rightmost input bits of the S-box and vary the four middle bits, each possible 4-bit output is attained exactly once as the middle four input bits range over their 16 range possibilities.

Starting from criterion (S-3) the idea of permutation is introduced into S-boxes, therefore some additional operations must be done in the generation

algorithm.

Verification

The verification method easily follows the specification: it generates the lists of the output nibbles[1] that have the same outer input bits and if a value is present more than once in the same list, the verification fails.

Generation

During the implementation phase it has been noticed that the random generation of an S-box that respects (S-1) and (S-2) almost never lead to the compliance with the third criterion. To solve this problem a randomized generation algorithm has been implemented into the `Sbox` class to grant the respect of (S-3). Going ahead with the criteria an increase of the difficult in the generation of S-boxes that respect only a sample of Coppersmith's design criteria shows up. Due to this issue, the implementation of ad-hoc generating functions is necessary in order to get a dictionary (`io`) satisfying a certain criterion.

The rest of this section is denoted to grant the respect of criterion (S-3). The S-box must behave like a permutation for fixed a and b

$$P_{a,b}(x) = S(a||x||b)$$

so 4 different permutations correspond to an S-box: $P_{0,0}$, $P_{0,1}$, $P_{1,0}$, $P_{1,1}$. To respect this constraint, it is sufficient to assign a possible nibble (chosen randomly from the list of the possible output nibbles for fixed outer input bits), to a certain input so that

$$\forall a, b \in \{0,1\}.\forall x, y.P_{a,b}(x) \neq P_{a,b}(y)$$

Once an output is selected from the list, it will be deleted to ensure the uniqueness of the output.

[1] a nibble is a 4-bit digit, namely an element of $\{0,1\}^4$

3.5 Graph-Based Stochastic Generation

An S-box that respects the third criterion almost never satisfies also the fourth or the fifth one, then some additional work with graphs is made through NetworkX [16], a Python language software package for the creation, manipulation, and study of the structure, dynamics, and function of complex networks. The use of graphs in this context has been inspired by the study [17], in which S-boxes that respected *all* the Coppersmith's criteria are generated, instead the aim of DESBoxGen is to satisfy only *a subset* of Coppersmith's criteria in order to analyze how SAT-Solvers react to these changes.

Anyway, graphs will be used in order to find permutations $P_{a,b}$ that respect the criterion for fixed a and b. The respect of the criterion for a defined permutation is a *necessary* but not *sufficient* condition, because input of different permutations have to satisfy the criteria as well. So graphs will be used to find many different permutations, and other subroutines will look for permutations that will respect (S-4).

3.6 Analysis of S-4 and its Implementation

Remember, from Section 2.4, that criterion (S-4) asks that

> If two inputs to an S-box differ in exactly one bit, the outputs must differ in at least two bits. (That is, if $|\Delta I_{i,j}| = 1$, then $|\Delta O_{i,j}| \geq 2$, where $|x|$ is the number of 1-bits in the quantity x.)

The verification method is simply implemented following the specifications, instead, for the generation one, some additional work with graphs is required, as declared in Section 3.5.

Verification

The routine for the verification of (S-4), `_check_dc4()`, computes for each input, the ones that differ in exactly one bit and for each possible couple with hamming distance equal to one compares their outputs: if their hamming distance is lower than 2, (S-4) is not verified.

Generation

Rationale

To create an S-box that respects both (S-4) and (S-3) and hardly ever (S-5), it's necessary to define previously the permutations $P_{0,0}$, $P_{0,1}$, $P_{1,1}$, $P_{1,0}$ corresponding to an S-box. Let G_1 be the graph of the possible input of a permutation $P_{a,b}$, in which the vertices are the possible nibbles of input and they are connected if and only if their hamming distance exactly 1.

$$(u, v) \in E \iff |\Delta V_{u,v}| = 1$$

If two input nibbles are connected by an edge in $G1$, their output should differ in at least two bits. The rationale behind the search for the output values of a permutation $P_{a,b}$ is to

- find a series of possible outputs for each "row" of $G1$ (as illustrated in Figure 3.1), i.e. for fixed outer bits of the permutations, through the construction of an intermediate graph $G2$.
- look for the output of the permutation $P_{a,b}$, such that outputs for inputs that differ in exactly one bit, will differ in at least two bits, through a graph $G3$.

Outputs for Fixed Outer Bits

The first step in the definition of a permutation

$$P : \{0, 1\}^4 \to \{0, 1\}^4$$

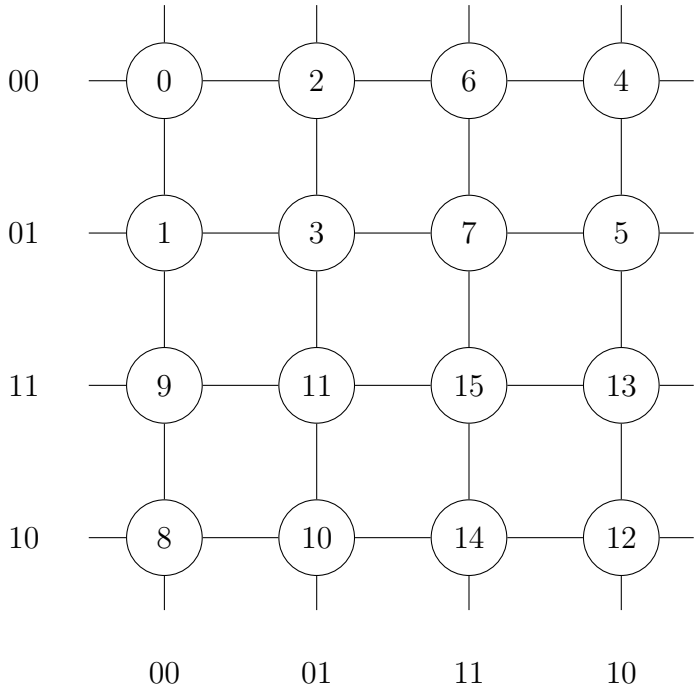


Figure 3.1: Graph G1 for the 4th criterion

is to fix its outer bits of input, in order to focus on outputs for inputs that differ only in the two middle bits. Let $C_{a,b}$ be the output of a permutation for fixed a and b

$$C_{a,b} : \{0,1\}^2 \rightarrow \{0,1\}^4$$

$$C_{a,b}(x) = P(a||x||b)$$

such that

$$\forall x, y \in \{0,1\}^2,\ |\Delta(x,y)| = 1,\ |\Delta(C_{a,b}(x), C_{a,b}(y))| \geq 2$$

Possible values for C can be found thanks to the generation of an intermediate graph, $G2$, whose vertices are all the possible nibbles from $G1$, but in this case are connected by an edge if and only if their hamming distance is at least 2. Notice that cycle in $G2$ corresponds to a possible output for a permutation with fixed outer bits $C_{a,b}$.

Output for a Permutation

In order to define a possible output for a permutation it's necessary to find 4 cycles in $G2$, for all the possible outer bits of the permutations: $C_{0,0}$, $C_{0,1}$, $C_{1,1}$, $C_{1,0}$. To ensure the compliance with the (S-4), if the outer bits of $G2$

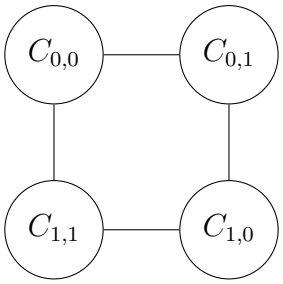


Figure 3.2: Cycle C

differ in exactly one bit, $C_{a,b}$ $C_{a',b'}$ have to respect the following condition for each index $i \in [0..4]$

$$\forall x_i \in C_{a,b}, \forall y_i \in C_{a',b'} \quad |\Delta(x_i, y_i)| \geq 2 \tag{3.1}$$

such that if two inputs differ in one bit, their outputs must differ in at least two bits. The search for some C that respect the condition and can be considered as valid permutations, can be seen like the problem of finding cycles in a graph $G3$, defined as follow.

Let $G3$ be the graph in which nodes are the cycles of $G2$ and they are connected if and only if the constraint 3.1 is respected. Since cycles are represented as tuples (x_1, x_2, x_3, x_4) and for simplicity two cycles are said to be compatible and adjacent if and only if vertices in the same position have an hamming distance greater or equal than 2, all the cycles should be considered in each possible order in the construction of $G3$ to avoid to be biased and to not consider some permutations in the complete generation.

The cycles in $G2$ are 2840, so considering their order there are 11360 cycles that should become the vertices of $G3$ (eight different representation for each cycle). Because of the large number of cycles of $G2$ that should

become vertices of $G3$, only a subgraph is considered: for each cycle C in $G2$ only one of its representations is selected by a random choice and will become a vertix of $G3$. Two vertices in $G3$ are connected if and only if the hamming distance for nibbles in the same position inside a node is at least 2.

A permutation that allows the compliance with criterion (S-4) corresponds to a cycle P in $G3$, such that

$$\forall i \in [0..15],\ \exists C \in P.\ i \in C$$

The search for permutations is stopped when a certain number is reached (by default this value is set to 3000) and it is repeated whenever an `Sbox` has to be generated so that all possible permutations can be considered.

Random Generation of an S-box

For the random generation of an S-box it is necessary to find 4 different permutations, $P_{0,0}$, $P_{0,1}$, $P_{1,1}$ and $P_{1,0}$, so that the design criterion can be satisfied. Once a new S-box has to be generated, permutations are regenerated to avoid to be biased and not to exclude some permutations in the generation of an S-box. In this phase permutations (represented as tuple of tuples) can undergo an inner and an outer permutation described below:

- The `inner_permutation()` in this case modifies the order inside the tuples, so that they continue to be cycles, i.e. the hamming distance for adjacent values in the tuple is at least 2.
- The `outer_permutation()` modifies the order of tuples, so that the tuples that form the permutation continue to represent a cycle of $G3$, i.e. the hamming distance for values in the same position in adjacent tuples is at least 2.

First of all, a permutation represented by a tuple of tuples is randomly chosen among those ones previously generated and its elements undergo an inner permutation and an outer one. These permutation will correspond to $P_{0,0}$

in the S-box, and its inner tuples will correspond respectively to $C_{0,0}$, $C_{0,1}$, $C_{1,1}$ and $C_{1,0}$. In a similar way $P_{0,1}$, $P_{1,1}$ and $P_{1,0}$ are found, but they require additional work. If we consider a permutation as it is, two different permutations can result incompatible even if they could be. For this reason it's necessary to consider all the different inner orders and the outer ones to check the compatibility between two permutations, i.e. if they can comply with criterion (S-4) if they are chosen as permutations for inputs that differ in just one of the outer bits. These steps are done through the application of the subroutine `get_comp()` that finds permutations compatible with the others given as input. Once $P_{0,0}$ is chosen, the build function looks for

- $P_{0,1}$ that has to match $P_{0,0}$
- $P_{1,0}$ that has to match $P_{0,0}$
- $P_{1,1}$ that has to match $P_{0,1}$ and $P_{1,0}$

All these steps lead to the creation of a dictionary that will correspond to an S-box in compliance with the 4th of Coppersmith's design criteria.

3.7 Analysis of S-5 and its Implementation

Remember, from Section 2.4, that criterion (S-5) asks that

> If two inputs to an S-box differ in the two middle bits exactly, the outputs must differ in at least two bits. (If $\Delta I_{i,j} = 001100$, then $|\Delta O_{i,j}| \geq 2$.)

As for (S-4), the verification method easily follows the specifications, instead the generation one is quite complex and requires some work with graphs, as discussed in 3.5,

Verification

The routine for the verification of (S-5), `_check_dc5()`, complements the two middle bits for each possible inputs so that for inputs that differ in the

two middle bits exactly it can check if the hamming distance between their outputs is at least 2. If it exists a couple of inputs that differ in their middle bits and the hamming distance between their outputs is lower than 2, the criterion isn't verified.

Generation

Rationale

To create an S-box that respects both (S-5), (S-4) and (S-3), like for checking the 4th criterion, it's necessary to define previously the permutations $P_{0,0}$, $P_{0,1}$, $P_{1,1}$, $P_{1,0}$ corresponding to an S-box. Let G_1 be the graph of the possible input of a permutation $P_{a,b}$ in which the vertices are the possible nibbles of input and they are connected by an edge if and only if their hamming distance exactly 1 or if they differ in their two middle bits exactly.

$$(u, v) \in E \iff |\Delta V_{u,v}| = 1 \vee \Delta V_{u,v} = 001100$$

As a result of the additional constraint for Coppersmith's 5th design criterion, in this case each row isn't a cycle but a 4-clique

If two input nibbles are connected by an edge in $G1$, their output should should differ in at least two bits. The rationale behind the search for the output values of a permutation $P_{a,b}$ is to

- find a series of possible output for each "row" of G1 (as illustrated in figure 3.3), i.e. for fixed outer bits of the permutations, through the construction of an intermediate graph $G2$ almost as for the 4th criterion
- look for the output of the permutation $P_{a,b}$, so that outputs for inputs that differ in exactly one bit or in their middle bits, will differ in at least two bits, through a graph $G3$

Outputs for fixed outer bits

The first step in finding the output of a permutation

$$P : \{0, 1\}^4 \to \{0, 1\}^4$$

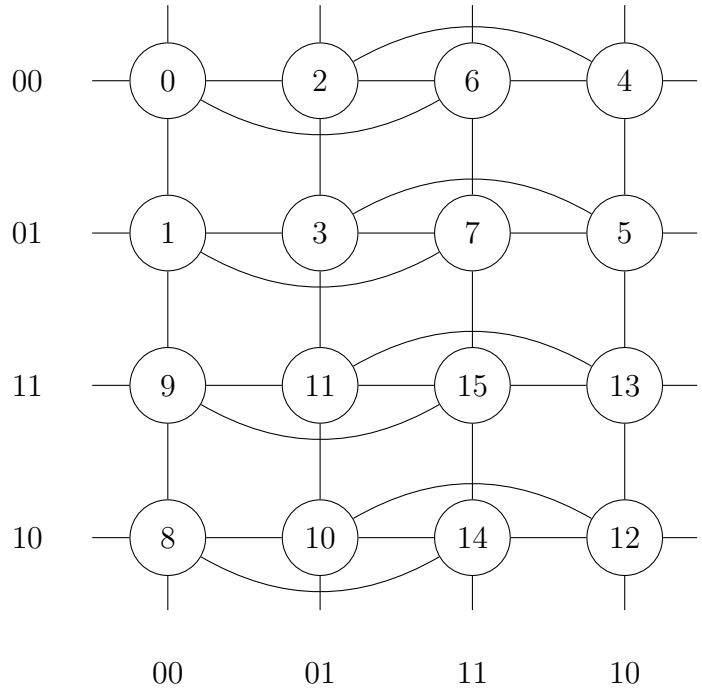


Figure 3.3: Graph G1 for the 5th criterion

is to fix its outer bits of input, in order to focus on outputs for inputs that differ only in the two middle bits. Let $C_{a,b}$ be the output of a permutation for fixed a and b

$$C_{a,b} : \{0,1\}^2 \to \{0,1\}^4$$
$$C_{a,b}(x) = P(a||x||b)$$

such that

$$\forall x, y \in \{0,1\}^2,\ x \neq y,\ |\Delta(C_{a,b}(x), C_{a,b}(y))| \geq 2$$

Similarly to the (S-4) case, possible values for C can be found thanks to the generation of an intermediate graph, $G2$, which vertices are all the possible nibbles like in $G1$ and they are connected by an edge if and only if the hamming distance is at least 2. In contrast to the 4th criterion in this case a possible output for a permutation with fixed outer bits $C_{a,b}$ isn't represented by a cycle but by a clique because each row of $G1$ corresponds to a 4-clique since both (S-4) and (S-5) have to be complied with.

Output for a permutation

In order to define a possible output for a permutation it's necessary to find 4-cliques in $G2$, for all the possible outer bits of the permutations: $C_{0,0}$, $C_{0,1}$, $C_{1,1}$, $C_{1,0}$. To ensure the compliance with the 5th criterion, if the outer bits

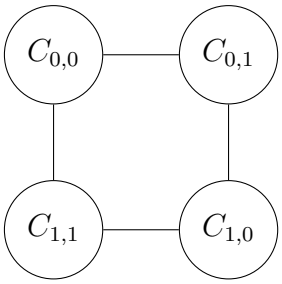


Figure 3.4: Cycle C

of vertices in $G2$ differ in exactly one bit or in their two middle bits, $C_{a,b}$ $C_{a',b'}$ have to respect the following condition for each index $i \in [0..4]$

$$\forall x_i \in C_{a,b}, \forall y_i \in C_{a',b'} \ |\Delta(x_i, y_i)| \geq 2 \tag{3.2}$$

such that if two inputs differ in one bit or in their middle bits, their outputs must differ in at least two bits. The search for some C that respect Condition 3.2 and can be considered as valid permutations, can be seen as the problem of finding cycles in a graph $G3$, defined as follow.

Let $G3$ be the graph in which nodes are the 4-cliques of $G2$ and they are connected if and only if Condition 3.2 is respected. Since cliques are represented as tuples (x_1, x_2, x_3, x_4) and for simplicity two cliques are said to be compatible and adjacent if and only if vertices in the same position have an hamming distance greater or equal to 2, all the 4-cliques should be considered in each possible order in the construction of $G3$ to avoid to be biased and to not consider some permutations in the complete generation.

In constrast to the 4th criterion, the number of 4-cliques is 228, so this allows a complete representation of $G3$, which vertices are 5472, because for each 4-clique we consider all its possible permutations. Two vertices in $G3$

are connected if and only if the hamming distance for nibbles in the same position inside a node is at least 2.

A permutation that allows the compliance with (S-5) corresponds to a cycle P in $G3$, so that

$$\forall i \in [0..15], \exists C \in P.i \in C$$

The search for permutations is stopped when a certain number is reached (by default this value is set to 3000).

Random generation of an S-box

For the random generation of an S-box it is necessary to find 4 different permutations, $P_{0,0}$, $P_{0,1}$, $P_{1,1}$ and $P_{1,0}$, so that the design criterion can be satisfied. Once a new S-box has to be generated, permutations are regenerated to avoid to be biased and not to exclude some permutations in the generation of an S-box. In this phase, permutations (represented as tuple of tuples) can undergo an inner and an outer permutation described below:

- The `inner_permutation()` in this case modifies the order inside the tuples, by considering each possible permutation of the values because all the vertices in the clique are connected by definition, so each permutation is still a clique, i.e. the hamming distance for adjacent values in the tuple is at least 2.

- The `outer_permutation()` modifies the order of tuples, so that the tuples that form the permutation continue to represent a cycle of $G3$, i.e. the hamming distance for values in the same position in adjacent tuples is at least 2 exactly like for (S-4).

First of all, a permutation represented by a tuple of tuples is randomly chosen among those ones previously generated and its elements undergo an inner permutation and an outer one. These permutations will correspond to $P_{0,0}$ in the S-box, and its inner tuples will correspond respectively to $C_{0,0}$, $C_{0,1}$,

$C_{1,1}$ and $C_{1,0}$. In a similar way $P_{0,1}$, $P_{1,1}$ and $P_{1,0}$ are found, but they require additional work as for (S-4). All the possible order for a permutation have to be considered in order to check compatibility through `get_comp()` like in the previous case with the subroutine `inner_permutation()` and `outer_permutation()` just described. Once $P_{0,0}$ is chosen, the build function looks for

- $P_{0,1}$ that has to match $P_{0,0}$
- $P_{1,0}$ that has to match $P_{0,0}$
- $P_{1,1}$ that has to match $P_{0,1}$ and $P_{1,0}$

All these steps lead to the creation of a dictionary that will correspond to an S-box in compliance with (S-5).

3.8 Analysis of S-6 and its Implementation

Remember, from Section 2.4, that criterion (S-6) asks that

> If two inputs to an S-box differ in their first two bits and are identical in their last two bits, the two outputs must not be the same. (If $\Delta I_{i,j} = 11xy00$, where x and y are arbitrary bits, then $\Delta|O_{i,j}| \neq 0$)

For criterion (S-6) only a verification method has been implemented and is explained below.

Verification

The routine for the verification of the 6th criterion, `_check_dc6()`, computes for each input the list of inputs that differ in their first two bits and have the same last two ones, through the complementary of the first ones and each possible variations of the middle ones. For each couple it checks if their outputs are different, i.e. if their hamming distance isn't zero. If a couple

of inputs that doesn't respect this constraint exists, then the criterion isn't verified.

Chapter 4

DESBoxGen's Architecture

DESBoxGen is a tool meant for logical cryptanalysis on DES and its variants. The variant of DES instance allows the specification of:

- a number of round, by default 16 as in DES standard algorithm
- the S-boxes that can replace DES standard ones and can be specified in three different ways:
  - through an identifier (an integer or a string) that identifies a series of S-boxes previously stored
  - through a list of objects that are instances of the class `Sbox()`
  - through the method `_sbox_generator(min_dc)`, that takes as input a minimum design criterion (`min_dc`) that must be satisfied through the following condition.

    ```
    all(s.dc[str(i)] is True for i in s.dc.keys() if
    int(i) <= min_dc)
    ```

    If `min_dc` isn't specified, the standard S-boxes are used.

```
class Des:
    def __init__(self, rounds=16, sboxes=None, min_dc=None)
```

Even if DESBoxGen is meant for logical cryptanalysis it allows the standard encryption expected from DES, so two main mode can be found in DESBoxGen:

- The standard one, in which all the bits of the plaintext and the key are known.
- The one meant for cryptanalysis that allows the presence of unknown bits both in plaintext and in the key used for the encryption.

```
def crypt(self, encrypt=True, key=None, plaintext=None,
key_dict=None, plaintext_dict=None, pair='0')
```

In the case of logical cryptanalysis with chosen plaintext attack, an identifier `pair` can be specified in order to distinguish the variables related to different couples $(plaintext, ciphertext)$.

4.1 Inputs to List

First of all the casting of the inputs, i.e. the key and the plaintext, has to be executed. If the input is completely known, the string could be in binary or not:

- If the string isn't a binary value, it will be converted into a list of bits (each char corresponds to 8 bits)
- If the string represents a binary value, a casting into list will be applied to the string

If the input isn't completely known (for example, `key=None`, equally for `plaintext`) a list of 64 variables is created: each variable corresponds to a certain bit of the unknown input. If only a part of the input is known (for example in the case of key `key is None` and `key_dict()` isn't empty) the value of these bits is set to 0 or 1 as specified into the dictionary of known bits received as input. All the following steps of the encryption algorithm will work on these list of length 64.

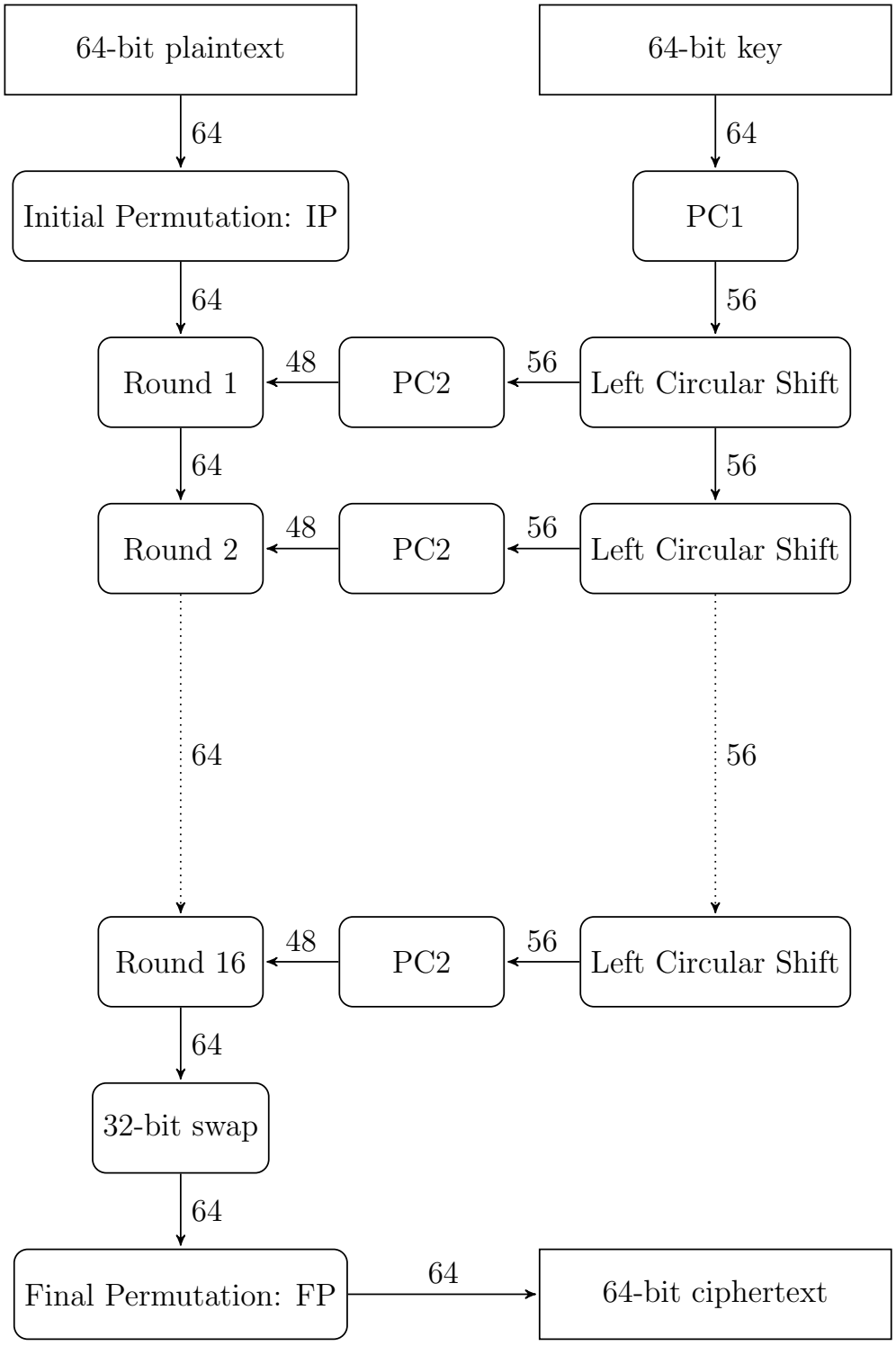


Figure 4.1: DES complete schema

4.2 Keys Generation

The generation of the round keys is the same if the key is known or unknown. First of all `PC1` is applied (for the explanation of permutations see 4.3), then the key is split into two halves, the left and the right part. For each of the two halves a shift is applied, the number of shifts is defined by DES specification. Once the shift is applied, the two halves are merged again and `PC2` will be applied on the resulting key.

4.3 Permutations and Similar Operations

A series of operations in DES involve only a permutation of values: bits have to change their previous position with another one according to a lookup table. In code it results:

```
def permutation(formula, table):
    return [formula[permuted_pos-1] for permuted_pos in
    table]
```

This edit is suitable for the following operations:

Initial Permutation `IP` the first permutation applied on the plaintext

Expansion-box `E` the expansion applied on the right half of the message

Permutation-box `P` the permutation applied on the right part after the application of the S-box

Final Permutation `FP` the permutation applied on the concatenation of the two halves of the message after all the rounds of the Feistel network, that give as output the complete ciphertext

4.4 Xor and S-box: operations with different behaviour depending on the mode

Unlike the operations discussed before in 4.3, the Xor and the Substitution performed by the S-box may vary depending on the execution mode, i.e. if there are unknown input bits that should be detected through logical crypt-analysis. The difference from 4.3 depends on the kind of operation that has to be executed: while in the previous case the permutations rely on the bits indexes and not on their value, in this case there is a major stress on the bit itself compared to its position.

4.4.1 Standard Encryption Mode

If the algorithm is executed in the easiest encryption mode, both Xor and S-box perform their standard execution.

```
[str(int(i) ^ int(j)) for i, j in zip(f1, f2)]
```

Listing 4.1: Xor in standard encryption

The `Xor` is performed element by element in the two lists `f1` and `f2`. This operation is applied in two moments:

- the `Xor` between the round key and the right part of the message after the application of the Expansion-Box
- the `Xor` between the right part of the message after the application of the Permutation-Box and the left half of the message

After the first one the S-box as to be applied. The 48-bit input of the S-box, represented by a list with length equal to 48, is divided in 8 lists of length 6 and the substitution is applied on each sublist.

The S-box that has to be applied corresponds to an instance of the class `Sbox` previously described in chapter 3, so it could be different from DES standard one. For each sublist the output is recovered through the method `input2output` of the class `Sbox` described in 3.1.

```
[s_out for i in range(8) for s_out in self.sboxes[i].
input2output(s_in[i])]
```

Listing 4.2: S-Box in standard encryption

4.4.2 Logical Cryptanalysis Mode

The second case, i.e. the presence of unknown bits in an input, is treated differently from the other one just described. In both case in the end the output will be composed by a binder and a binding obtained through the `replace()` routine (as explained in section 1.3.2).

- The binder will be the new variable that will correspond to a certain bit of the result.
- The binding will link the previous value of the bit(represented by a complex formula) to the new variable (i.e. the binder)

In the case of Xor, the values of the bit before the execution of the `replace()` are summarized by the following code:

```
tmp_xor = [And(Or(i, j), Or(Not(i), Not(j))) for i, j in
zip(f1, f2)]
```

Listing 4.3: Xor in encryption for logical cryptanalysis

instead in case of S-box, they will be:

```
s_tmp = [s_out for i in range(8) for s_out in self.sboxes[i
].truth_table2formula(s_in[i])]
```

Listing 4.4: S-box in encryption for logical cryptanalysis

In the end the list of complex formulas will contain just the new variables in order to simplify further operations. The functions (`xor()` and `substitution()`) will return two values:

- the list of binders that will reflect the edit on R_i
- the list of bindings that will be restored in the end

4.5 Encryption to Logical Cryptanalysis

The return values of the encryption (decryption) algorithm are:

`bin_out` the list of bits representing the ciphertext, if all the input bits were known, the list of the corresponding formulas otherwise

`str_out` the string corresponding to the result of the algorithm (`None` if some bits in the resulting ciphertext are unknown)

`binding` the list of all the bindings that have to be restored if there were unknown bits in the inputs

These return values are meant for logical cryptanalysis on DESBoxGen. The attack for which DESBoxGen is implemented for is logical cryptanalysis with chosen plaintext attacks.

First of all one or more couple of $(plaintext, ciphertext)$ have to be recovered. DESBoxGen easily implements the generation of these couples through the standard execution of the algorithm with completely known inputs (i.e. key and plaintext). Secondly, in order to emulate a known plaintext attack, DESBoxGen encrypts the plaintext with an unknown key, so the results of the encryption algorithm are a list representing the complete ciphertext and a list of bindings introduced during the computation. The value of the ciphertext isn't considered so far. The operation just described is iterated for each pair $(plaintext, ciphertext)$, and the values of the formulas, the ciphertexts and the bindings are stored into three different lists.

Once all the pairs have been parsed, `des_dimacs_cnf()` is executed, in order to generate the `DIMACS CNF` file that will be given as input to the SAT-Solver. Each pair corresponds to a formula that has to be recovered through the list of the 64 bits encoded in a certain formula, the final value of these bits (given by the known ciphertext) and the bindings introduced during the computation that link the plaintext to the bits representing the ciphertext.

The process that will restore the link between the list of formulas (representing the bits) and ciphertext is quite simple:

- the value of each bit is encoded in a formula but this value is known because it represents a bit in the ciphertext
  - if the value of that bit in the ciphertext that represents is `1`, the formula representing the bit is considered as it is (`f`)
  - otherwise if the value is `0`, the formula representing the bit is negated (`Not(f)`)
- all the formulas representing a bit are merged by `And`

In order to restore all the binders it's sufficient to apply the `And` between the formula just recovered and the bindings. By iterating this process for each formula and by doing the `And` between all the ones representing a certain pair, the final formula will be obtained.

At this point it's sufficient to apply `expr2dimacscnf()` (implemented by PyEDA) on the complete formula in order to get the `DIMACS CNF` corresponding to all the pairs encrypted with the same key.

Chapter 5

Evaluation of Coppersmith's Criteria

Thanks to DESBoxGen there are all the elements for the evaluation of Coppersmith's criteria by way of SAT Solving. In this phase three different SAT Solvers have been used:

Picosat A SAT-Solver written in C [18] that has several bindings in other languages, such as Pycosat in Python and PiGoSAT in Go.

CryptoMiniSat5 A SAT-Solver mostly written in C++ [19, 20] with interfaces for command-line, C++ library and Python.

Lingeling A SAT-Solver written in C [21, 22] that has taken part to several SAT Competitions.

Theoretically a bigger number of rounds, pairs or minimum design criterion respected by S-boxes should imply an increase in both the accuracy of the results and the time required by a SAT-Solver in recovering the key used during the encryption phase.

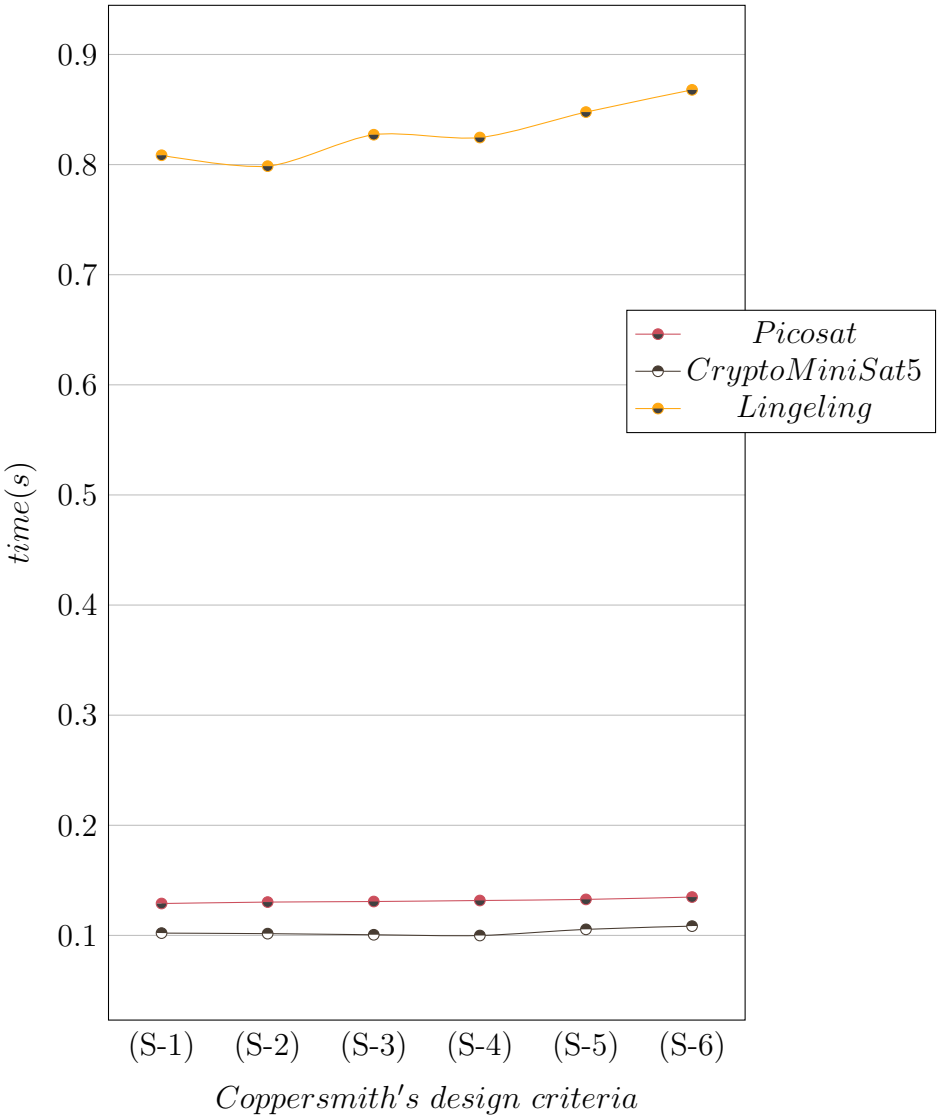
Twenty sets of S-boxes have been built for each criterion to compare the behaviour of SAT-Solvers applied on different version of DES. Therefore

in total there are 120 sets, i.e. 960 S-boxes different from the standard ones. Random keys and messages were used in order to produce several pairs (*plaintext*, *cipertext*) for the generation of `DIMACS CNF` through DESBoxGen. Below several tests are discussed with variations in number of rounds and pairs for each of Coppersmith's criteria until (S-6).

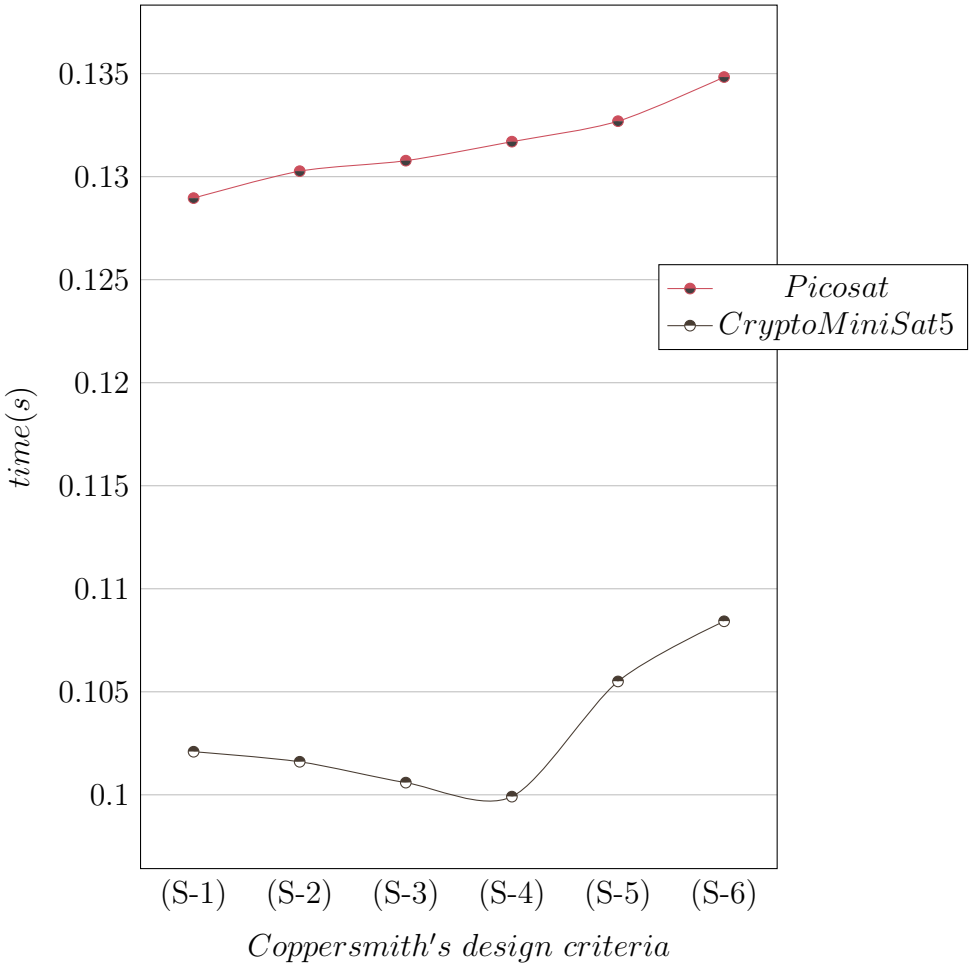
5.1 Results with 2 Rounds

Tests on the first round are not repeated here, because only half of the ciphertext is affected by the key therefore it should be too easy for a SAT-Solver to find a solution. The results of tests displayed below refer only to a number of pairs bigger than 4 due to their statistical relevance.

2 Rounds with 4 Pairs



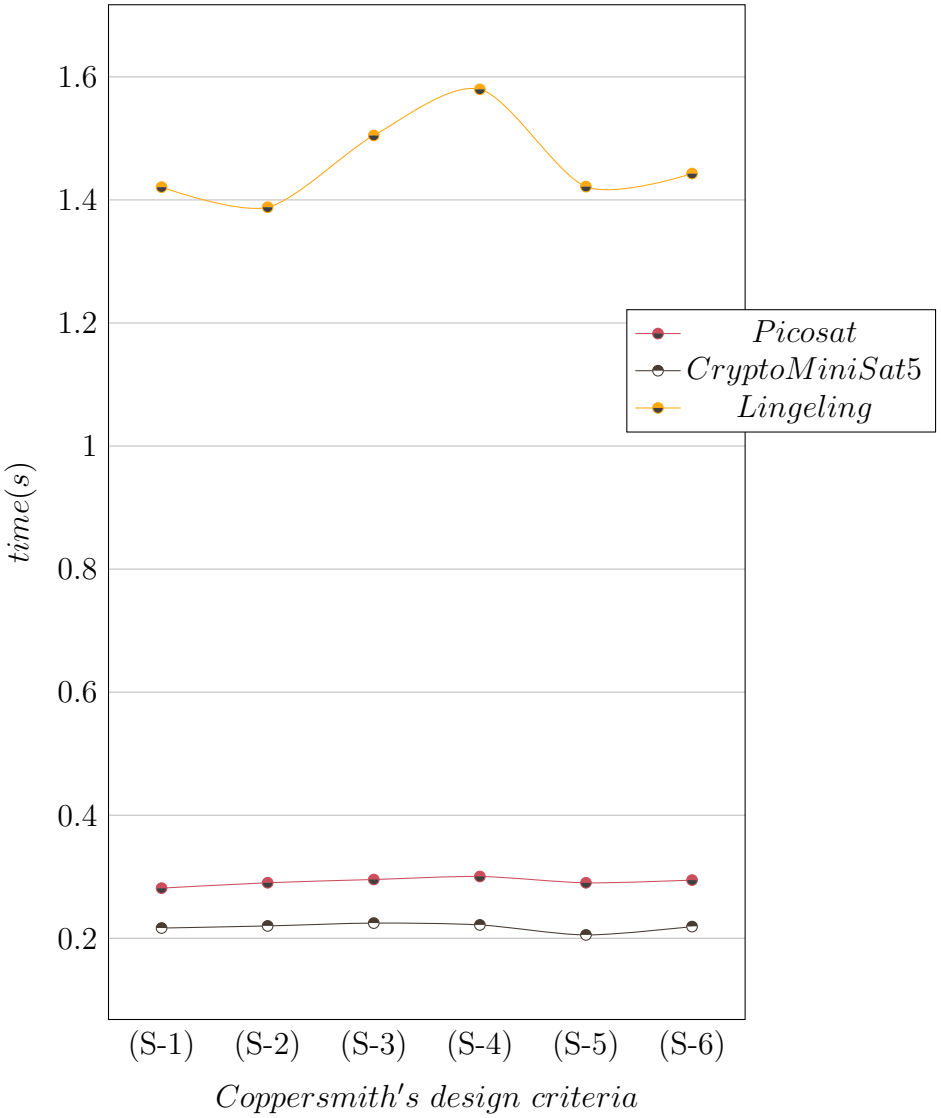
Lingeling seems to comply with Coppersmith's rationale about criteria even if it requires additional time compared to the others SAT-Solvers. The most unexpected behaviour of Lingeling is the rapid increase followed immediately by a little decrease in (S-4). After that point the behaviour of Lingeling seems to be in accordance with Coppersmith's revelation.



In Picosat instead there's a good compliance of Coppersmith's criteria: the line of plot is in continue growth. The behaviour of CryptoMiniSat with a close look appears in contrast with Coppersmith: there is a continuous decrease until (S-4). Only after (S-4) there is the expected increase up to (S-6).

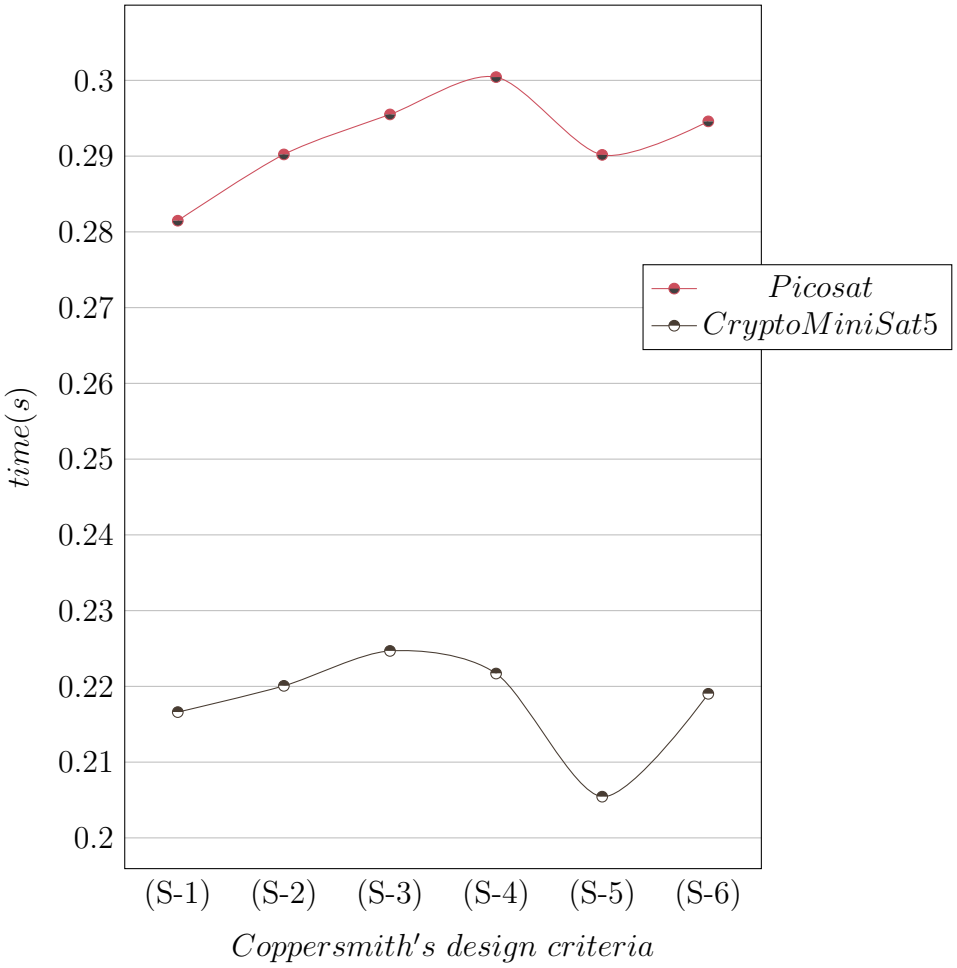
2 Rounds with 8 Pairs

In this case the results show a discrepancy from the previous ones and from Coppersmith's declaration only in the last steps. The point of disagreement is (S-5): in the passage from (S-4) to (S-5) there is a deep decrease of the complexity of the problem that the SAT-Solvers have to deal with.



This phenomenon is more evident in Lingeling, but also in CryptoMiniSat

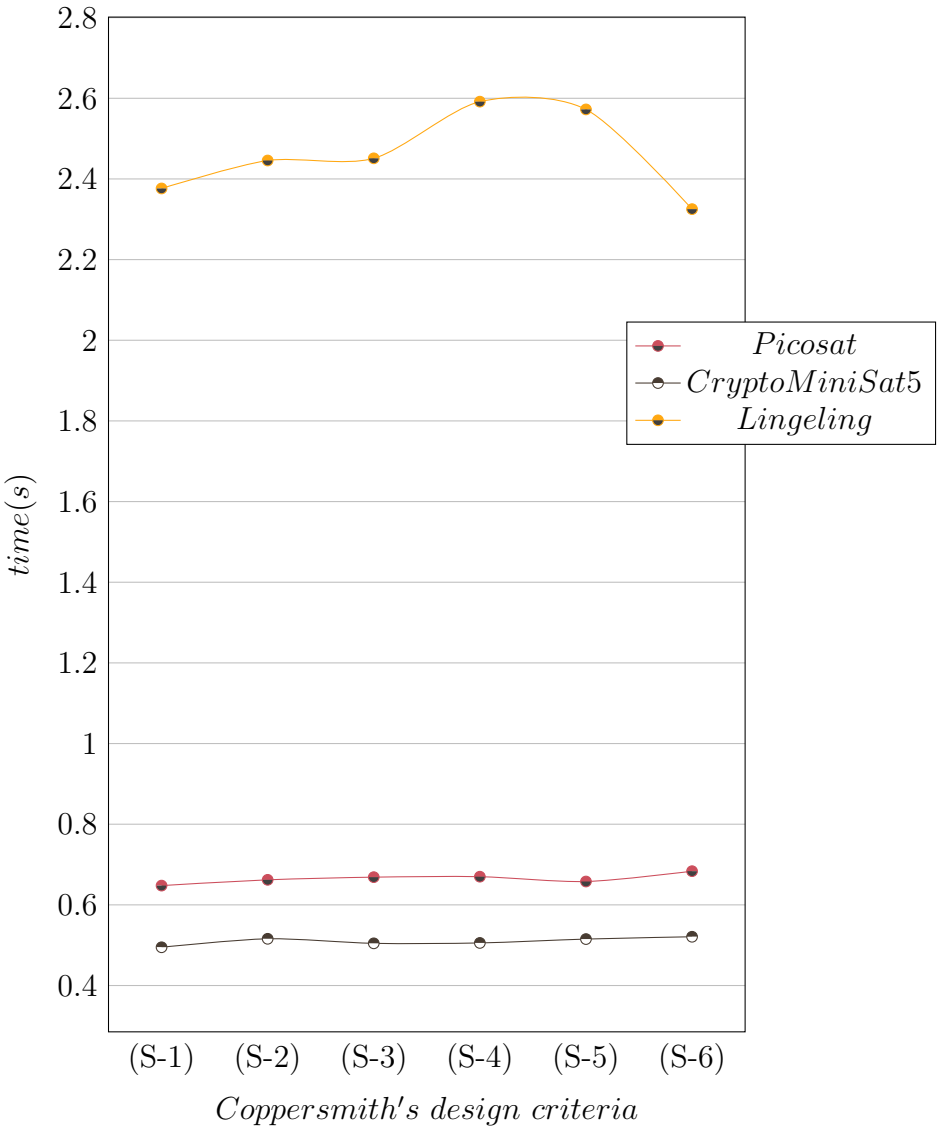
and Picosat that have a very similar shape. Lingeling continues to take more time than the others two in finding solutions and breaking DES variants.



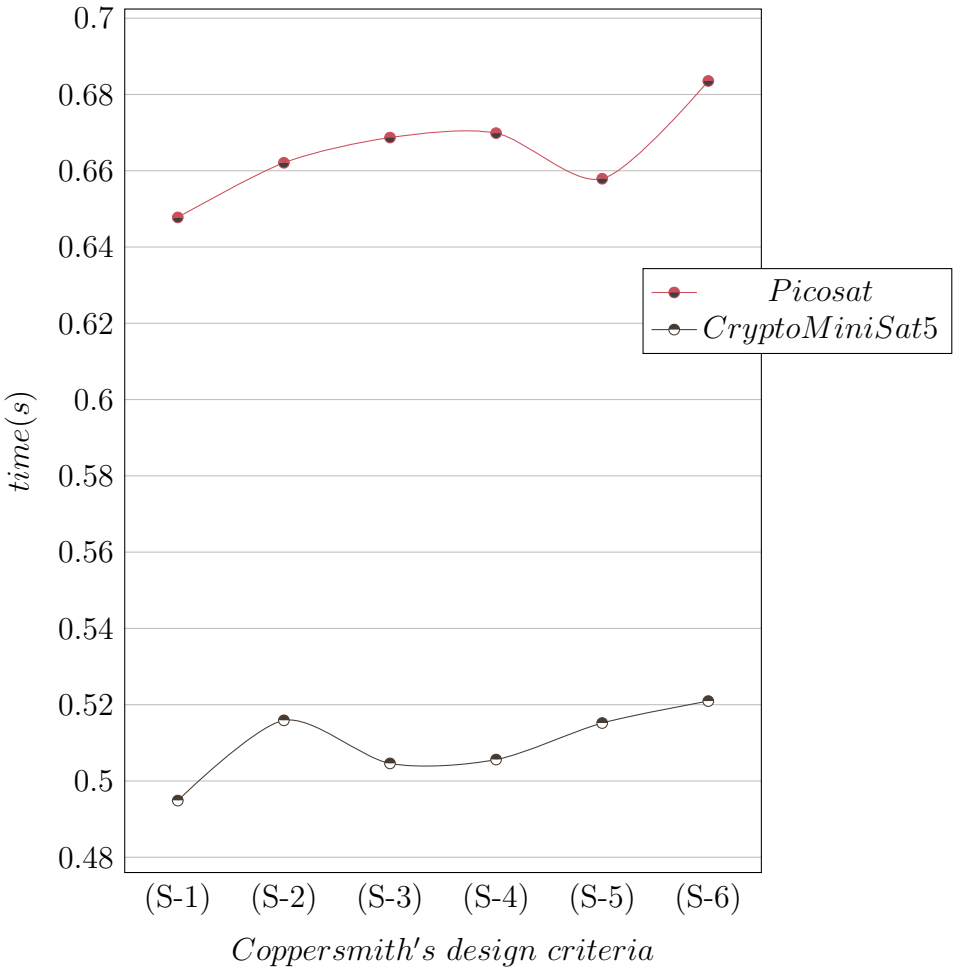
Also the time required by solving formulas involving (S-6) shows disagreement with Coppersmith: the time in (S-6) is lower than many others, for example notice in the plot that Lingeling takes less time to find a solution for formulas involving (S-6) than (S-3) on average.

2 Rounds with 16 Pairs

In these results there is a certain discrepancy. There are meaningful variations in the application of Lingeling that shows downfalls getting bigger from (S-4) to (S-6), opposed to expectations. Also the time required in solving formulas involving (S-3) seems to require less time than the ones involving (S-2).



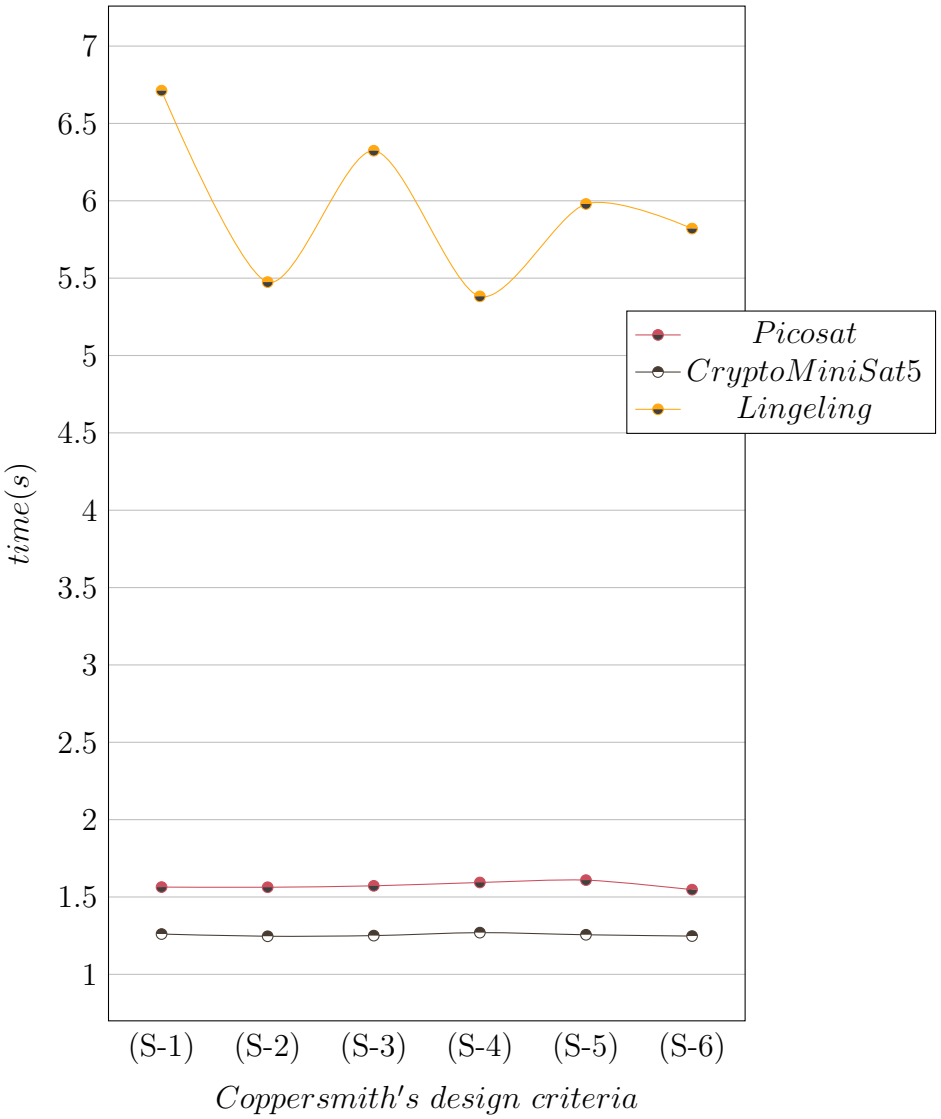
With a close look to Picosat an unexpected decrease at (S-5) can be noticed but unlike Lingeling the time required at (S-6) is compliant with Coppersmith.



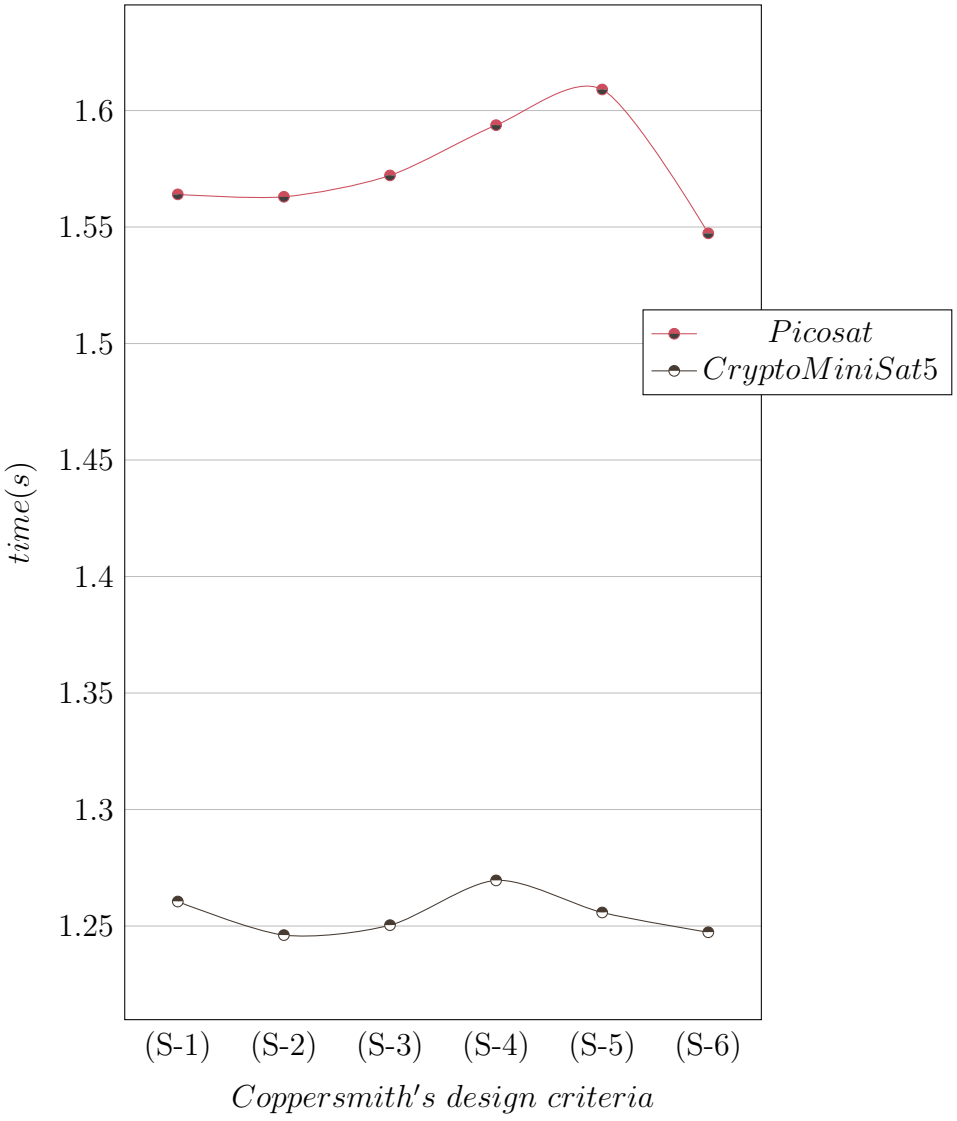
Instead in CryptoMiniSat there isn't a similar phenomenon at (S-5) but it can be found at (S-3). It could seem that the respect for only the property of non linearity, (S-2), is sufficient to increase the difficulty in breaking the algorithm in this case. In the remaining part of the plot of CryptoMiniSat it seems that there is a compliance with Coppersmith's rationale.

2 Rounds with 32 Pairs

Lingeling has an irregular behaviour by varying the criteria respected by the S-boxes. It seems that the difference in the criteria complied with isn't relevant for Lingeling in this case. The curve as several oscillations and downfalls that are in complete disagreement with Coppersmith.



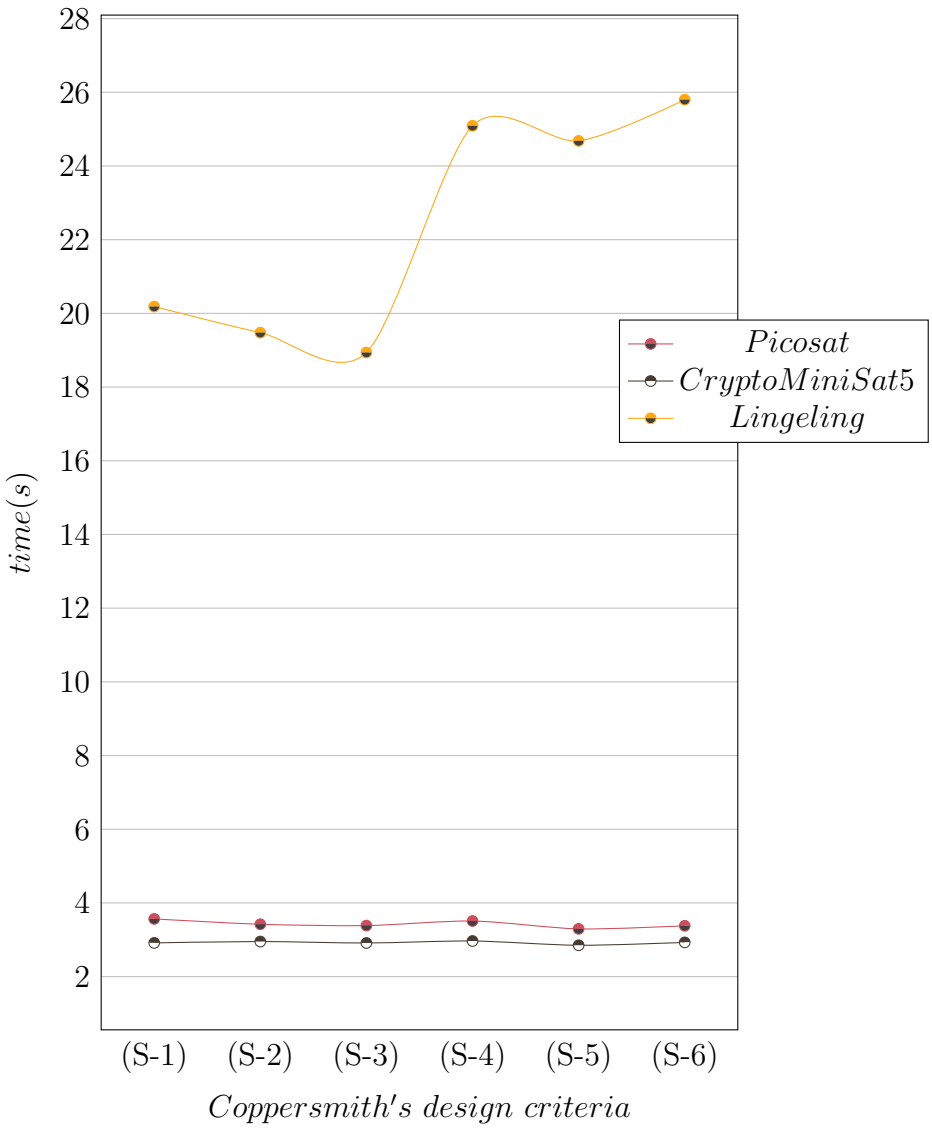
Instead, in Picosat there is a major compliance with Coppersmith until (S-5) but there is a significant downfall till (S-6).

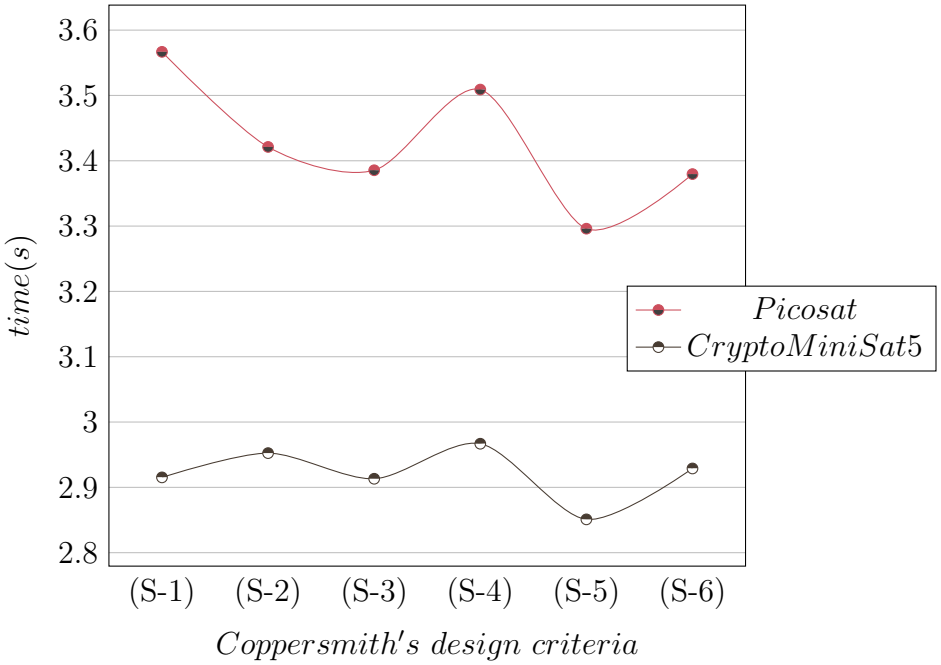


Also the behaviour of CryptoMiniSat doesn't show a total compliance with Coppersmith: it's sufficient to see the downfall in (S-5) and (S-6).

2 Rounds with 64 Pairs

The behaviour of Lingeling seems to respect Coppersmith's criteria more, as displayed by the increase of time required by CNF built in the respect of (S-4) compared to (S-3). Anyway there are decreases in the first three criteria and also in (S-5). The major compliance with the criteria may be due to the number of pairs given as input that should increase the difficulty of the problem.





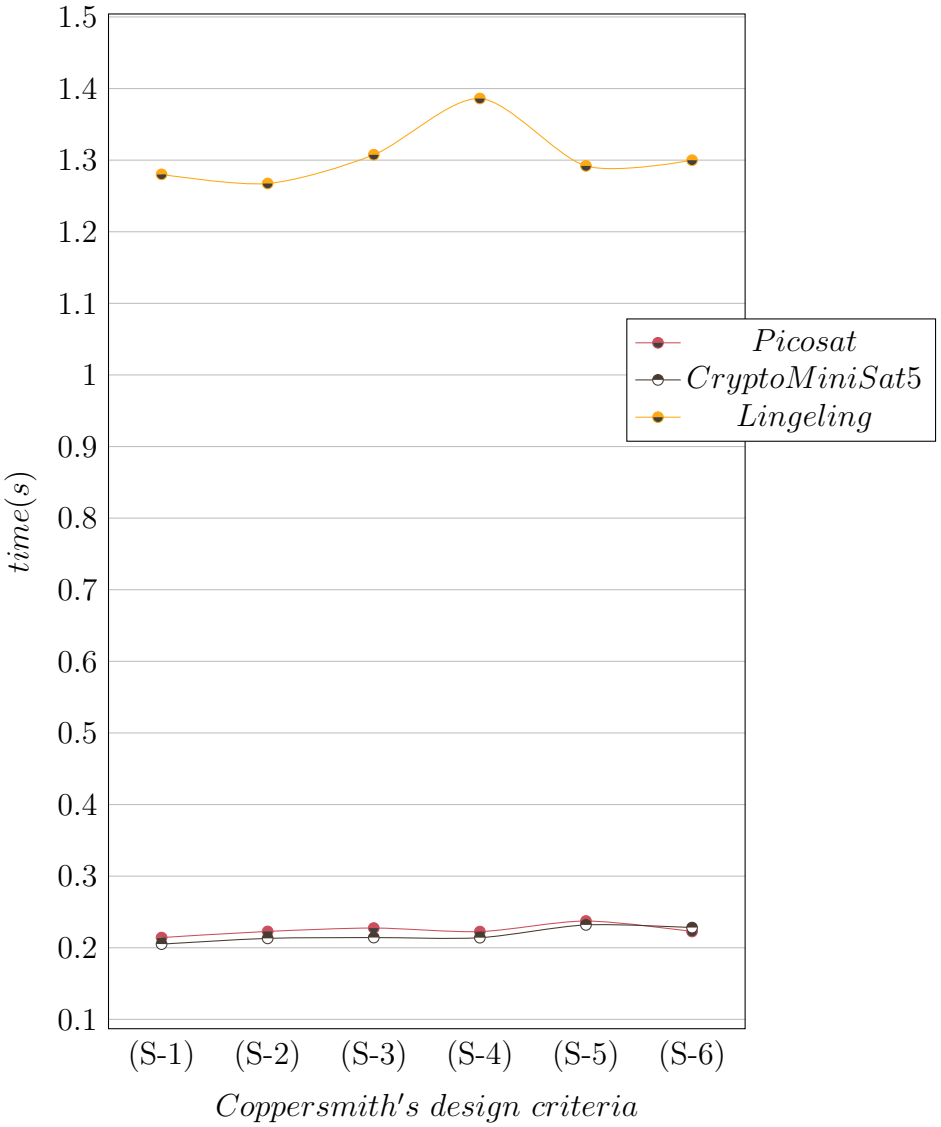
With a close look to Picosat and CryptoMiniSat several oscillations can be observed: for these two SAT-Solvers the difficulty introduced by the criteria doesn't seem relevant.

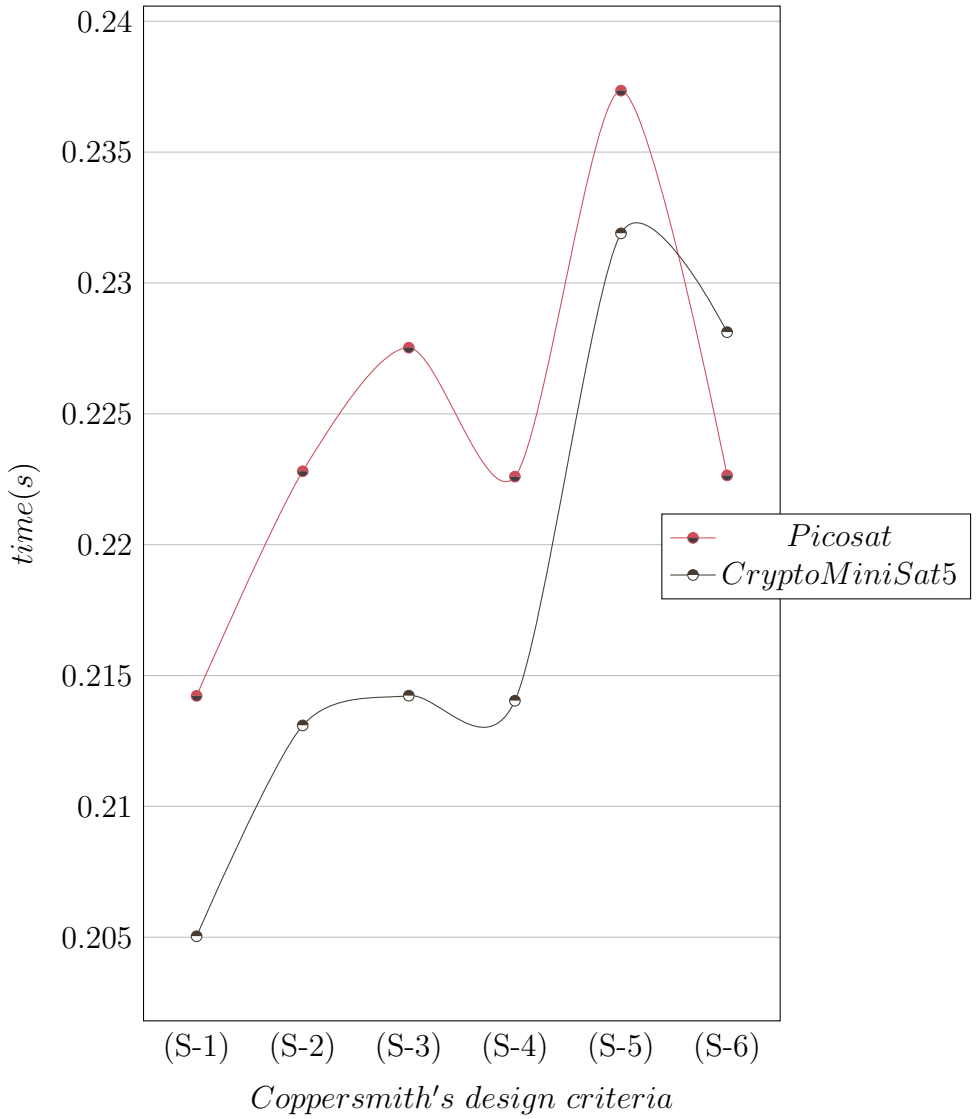
5.2 Results with 3 Rounds

The addition of another round increases the avalanche effect and the complexity of the problems that the SAT-Solvers have to deal with. Also in this case tests are done with different numbers of pairs $(plaintext, ciphertext)$ to observe more deeply some variations between the SAT-Solvers by changing the criteria complied with. As in the previous case only the results of tests that refer to a number of pairs lower than 4 aren't displayed.

3 Rounds with 4 Pairs

By the application of Lingeling is seems that Coppersmith's rationale isn't complied with. The most evident conflict with the criteria defined by Coppersmith can be found on `DIMACS` built through (S-5) and (S-6) that require less time than (S-4) ones.

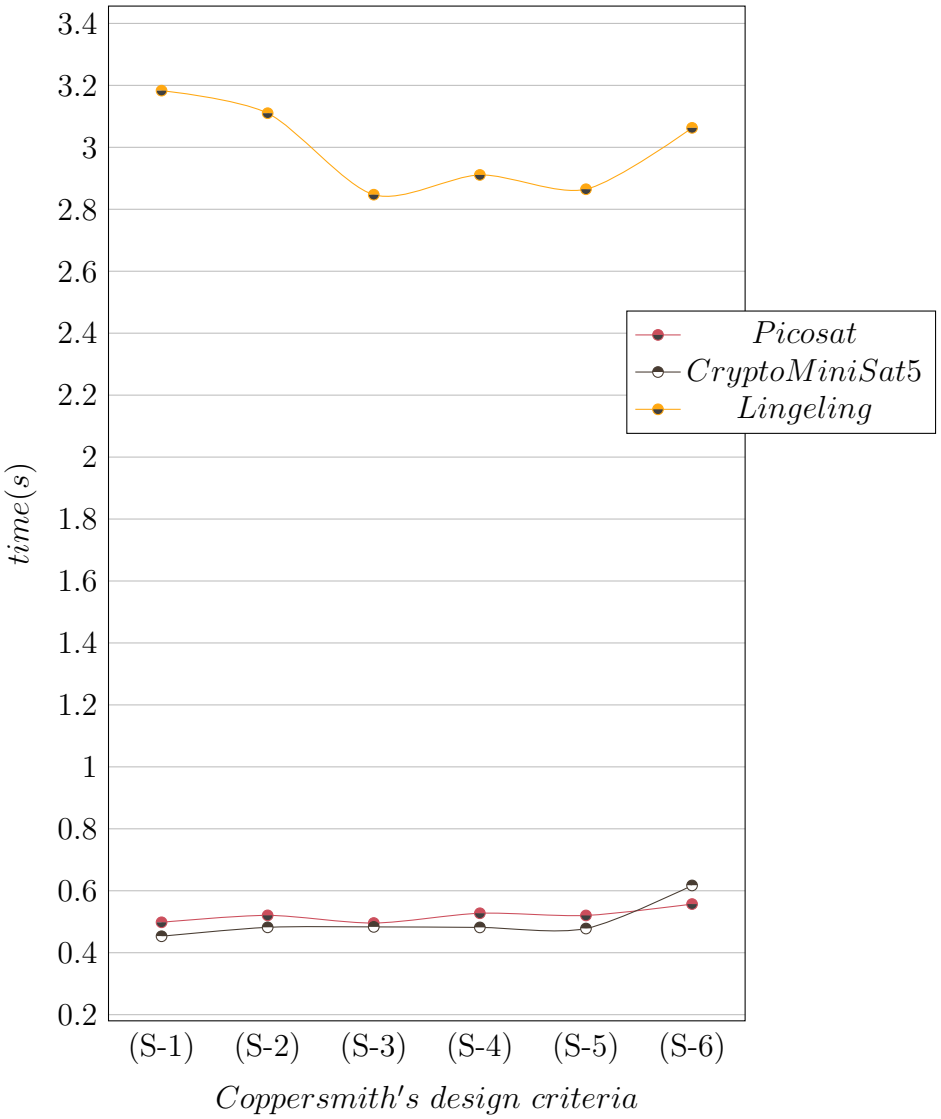


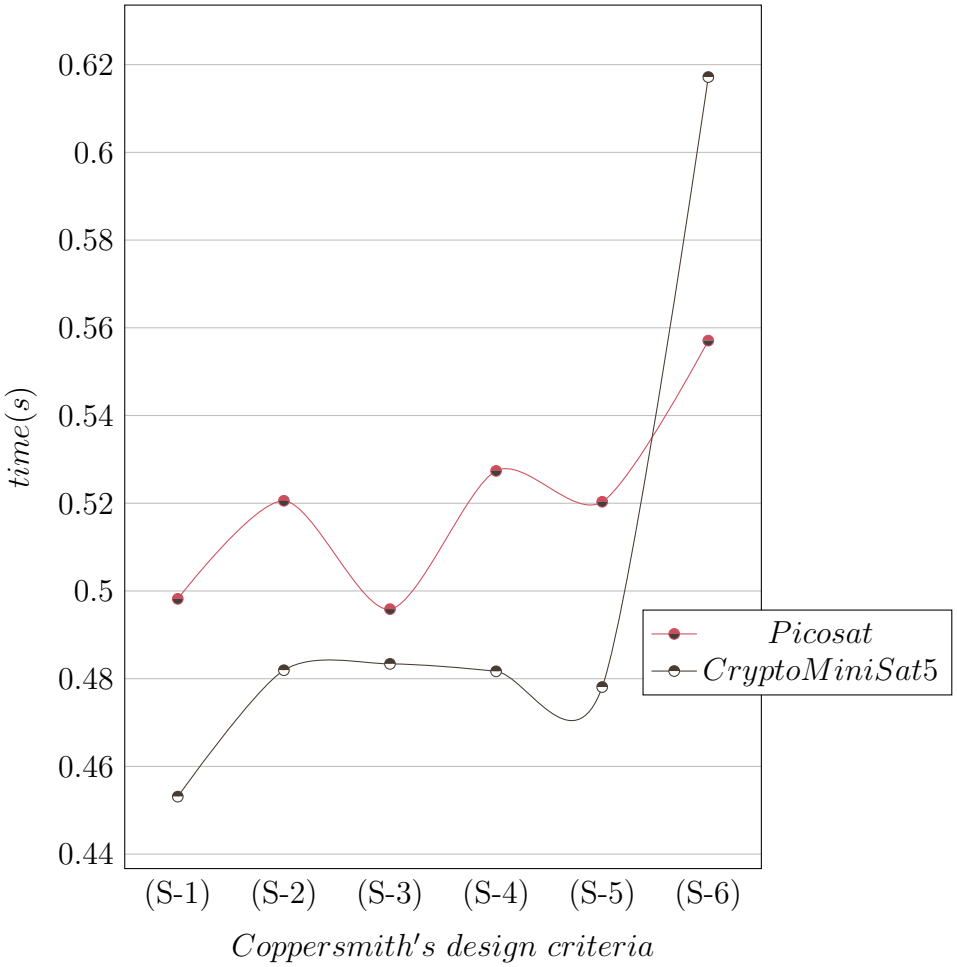


A significant unexpected downfall can be also seen in the plot of Picosat and CryptoMiniSat in correspondence of (S-4) that requires less time than (S-3) and (S-6) that requires less time than (S-5).

3 Rounds with 8 Pairs

The results of Lingeling as in the previous case are more unstable by changing the minimum design criteria complied with. The time required by the first criteria exceed the one required by the others.

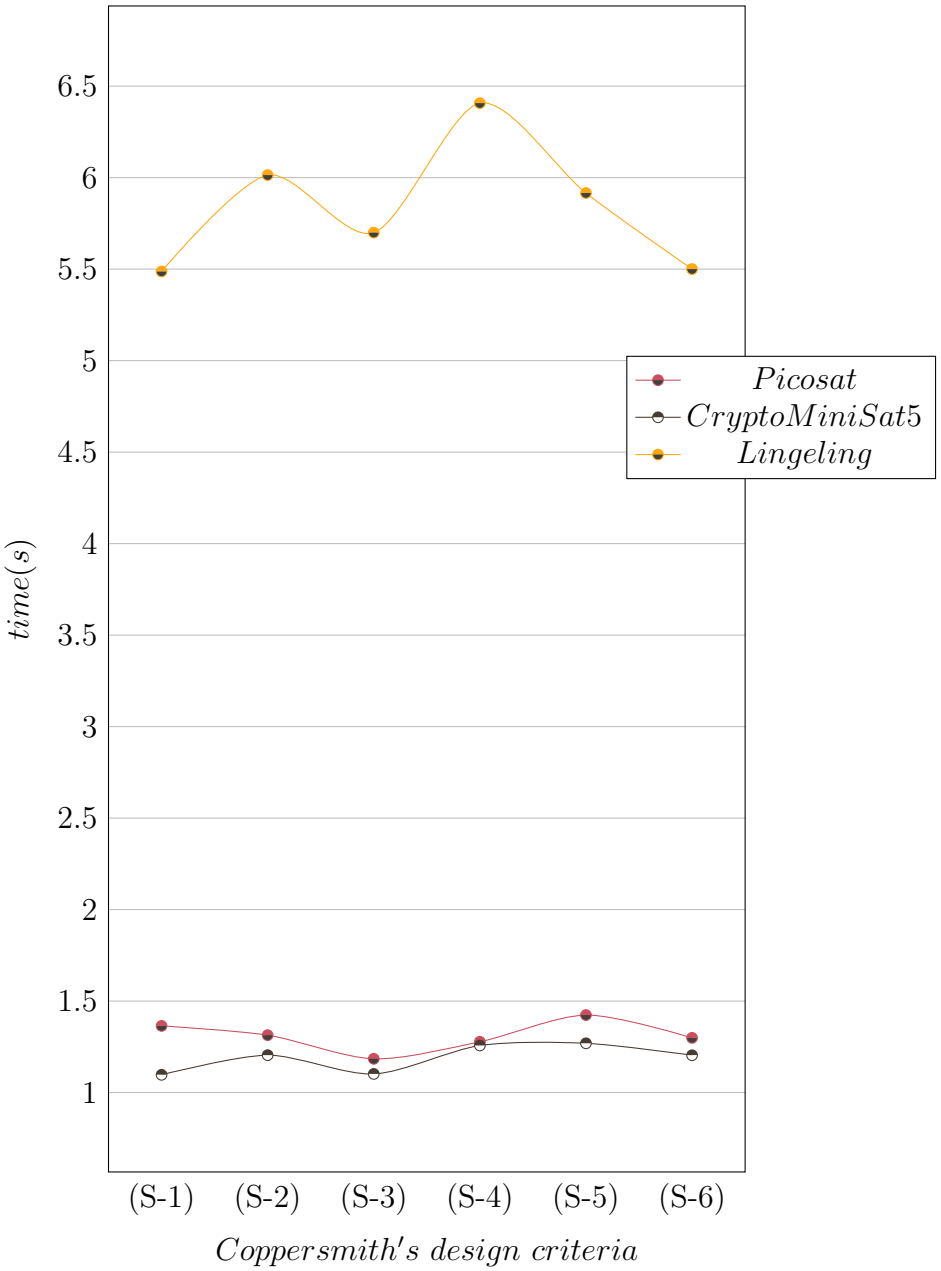




Picosat and CryptoMiniSat instead seem to have a bigger stability and a certain growth according to Coppersmith's rationale with the exception of Picosat in (S-3). Also the stationary points in the plot of CryptoMiniSat are unexpected and the slight decrease in correspondence of (S-5).

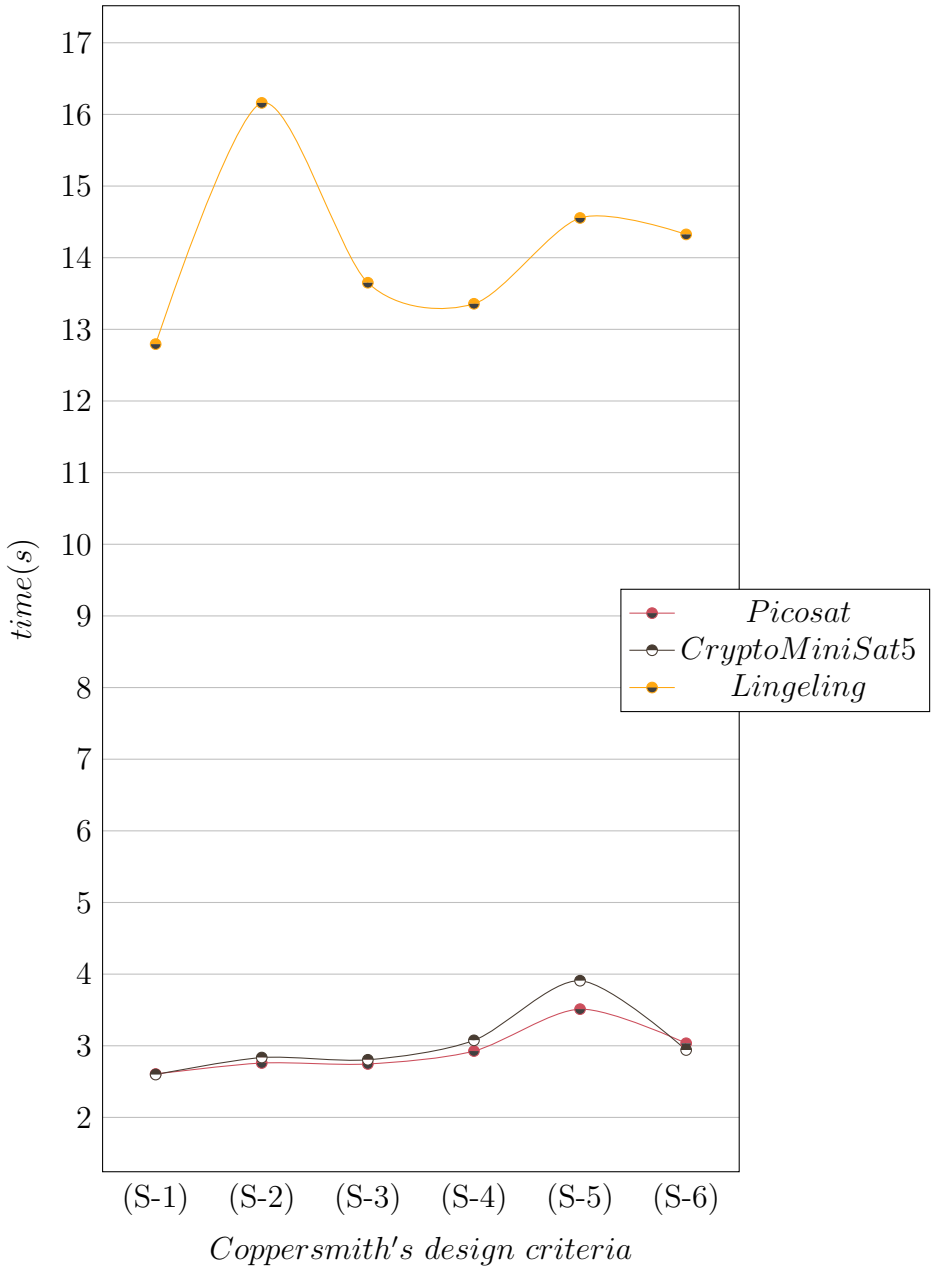
3 Rounds with 16 Pairs

The stability of both CryptoMiniSat and Lingeling of the previous case seems lost. Also in this case it appears that these results don't take into account Coppersmith's idea.



3 Rounds with 32 Pairs

In this case there are several unexpected behaviours. The most significant ones are the spike of Lingeling on (S-2) and the one of CryptoMiniSat and Picosat on (S-5).

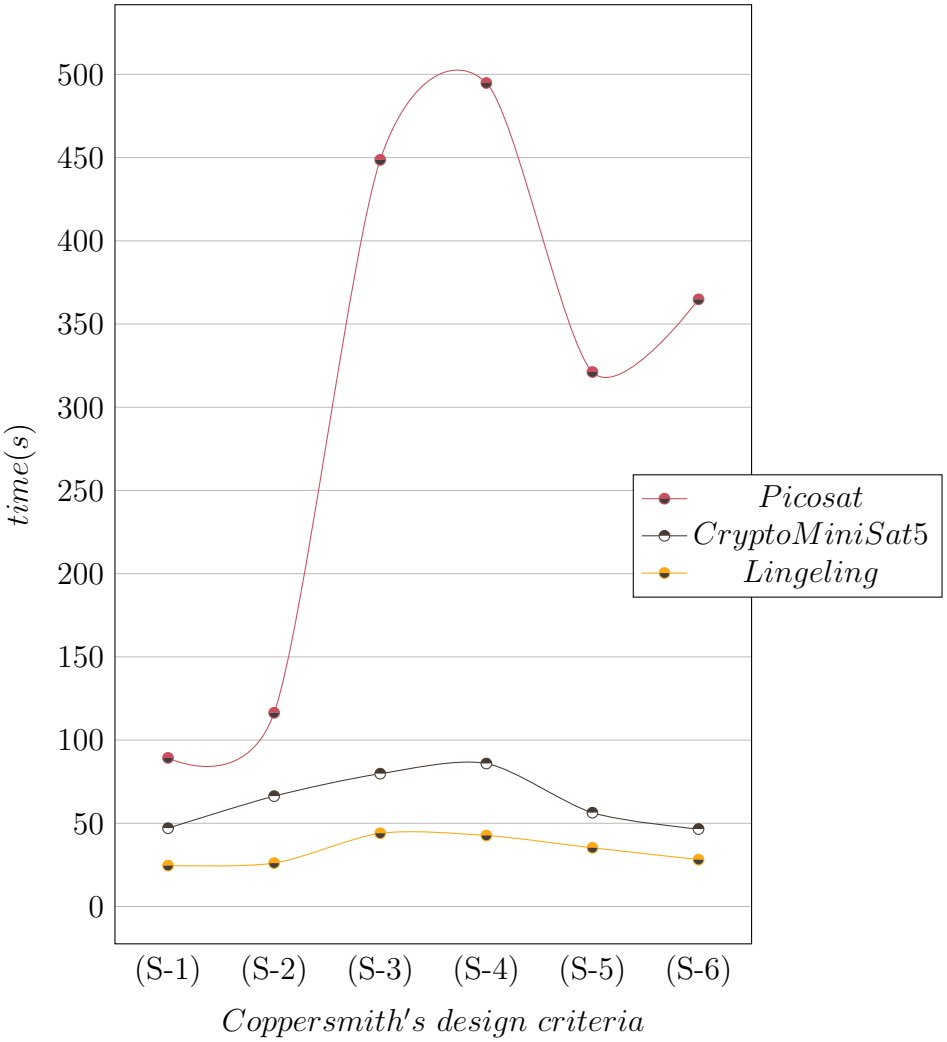


5.3 Results with 4 Rounds

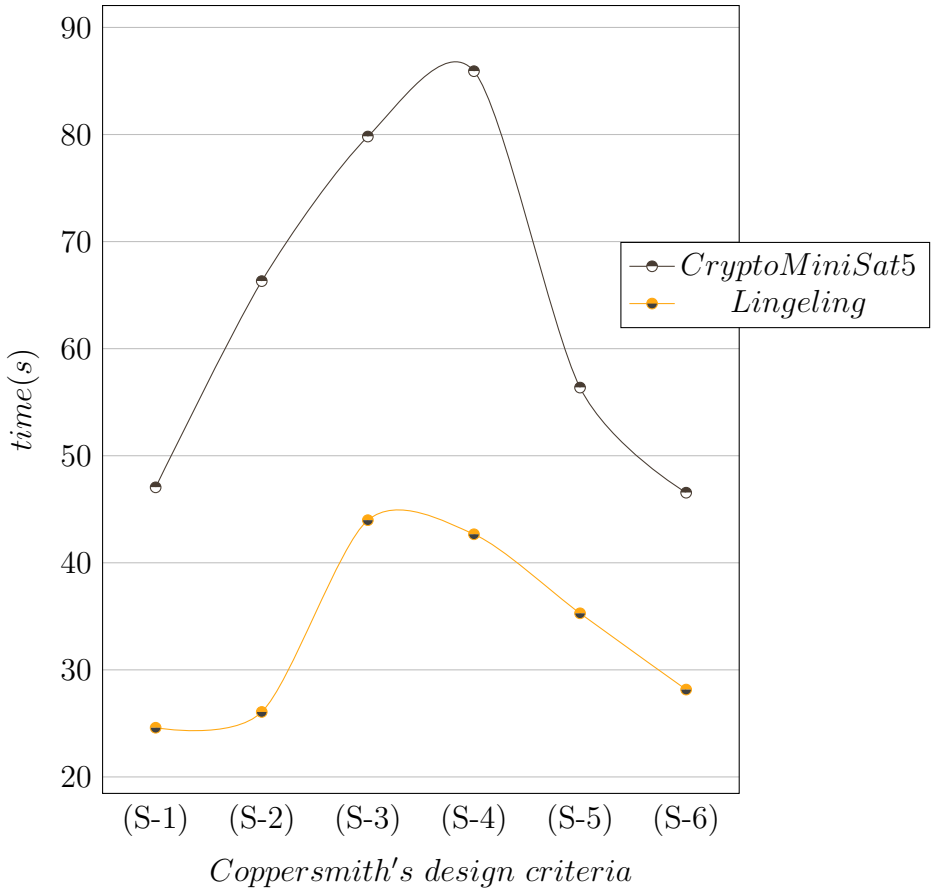
The addition of the fourth round, as in the passage from 2 to 3 ones, increases the avalanche effect, the complexity of the problems and by consequences the time required for solving them.

4 Rounds with 1 Pair

Now the overhead required by Lingeling (that previously made its performances worts than the others) seems to improve it in a significant way: it is the SAT-Solver that required less compared to the others.



In this case the worst performances are that ones of Picosat, that in certain cases required more than an hour and a quarter to find a solution for a formula (a problem in (S-4)) differently from CryptoMiniSat and Lingeling that took about one minute.



Despite the differences in time, the shape of the curve is quite similar for all the SAT-Solvers. The results until (S-4) seem to comply with Coppersmith's rationale. This accordance disappear in (S-5) that show a deep decrease in the required time in contrast with the results attained.

Conclusions and Outlooks

Through this thesis and DESBoxGen it has been demonstrated the possibility of creating S-boxes that respect only a sample of Coppersmith's criteria. As showed by the experimental results it has been noticed that the compliance with the criteria doesn't affect in a deep way the performance in time of the three SAT-Solvers chosen for these tests.

Often it seems that the time required in solving logical problems involving (S-5) is lower compared to the one involving some of the previous criteria. The S-boxes in compliance with this criterion are an interesting object of further study, as well as the ones respecting at least (S-4) that required a non indifferent solving time on average.

In order to get results with a greater statistical relevance and to express a better judgment about the relation between Coppersmith's design criteria and the difficulties in breaking DES algorithm and finding the key, other tests should be done with more resources, that lacked during the development of this thesis, to take the full advantage of logical cryptanalysis.

Acknowledgments

I want to thank Simone, my boyfriend, for standing by me during all the writing of the thesis. You have been perfect in (almost) never losing your temper when I fell apart.

I want to thank my supervisor, the professor Ugo Dal Lago, for his constant presence, his readiness, his patience and for guiding me through this work always carrying positive energy. I want to thank also all the professors of the course of study, especially the professor Renzo Davoli, for infusing the love for IT in me and making me fell in the right place.

The greatest thanks go to my family, mamy, daddy and Niky, for believing in me even in the worst periods. You are unique.

I thank Bologna and the Irnerio residence for giving me a second family: my brother Caso, my Bari, my companion of adventures JC and Sara. I'll never forget the nights of crazy and desperate study. Thanks to Luca, Paolo, Ciccio Boccuni, Hajar, Valeria, Fra, Ciccio Roccuzzo, Anna, Ivan, Annacarla, Aurel, Beppe, Fabio Proietti and Guaraldi, Jorgo, Sonia, Winston, Shou and all my house mates for their patience and laughs in three years of cohabitation.

Thanks to all my classmates, especially Maffo for encouraging me to improve and to be more self-confident.
Thanks to Serena, Tania, Amanda and Debora for always being there for me.

My last thanks go to Vittorio: partly it's his fault/merit if I decided to study IT in Bologna. My special thanks go to all of them.

Ringraziamenti

Grazie a Simone, il mio ragazzo, per avermi sopportata durante tutto il perido di scrittura della tesi. Sei stato perfetto a non sbroccare (quasi) mai nei miei momenti di crisi.

Grazie al mio relatore, il professore Ugo Dal Lago, per la presenza continua, la disponibilità, la pazienza e per l'avermi guidata in questo progetto sempre con energia positiva. Ci tengo a ringraziare inoltre i docenti del corso di studi, in particolare il professore Renzo Davoli, per avermi trasmesso l'amore per l'informatica e avermi fatta sentire nel posto giusto.

Il ringraziamento più grande va alla mia famiglia, mamy, daddy e Niky, per aver sempre creduto in me, anche nei momenti peggiori. Siete unici.

Ringrazio Bologna e lo studentato Irnerio per avermi dato una seconda famiglia: mio fratello Caso, la mia Bari, il mio compagno di avventure JC e Sara. Non dimenticherò mai le nottate di studio matto e disperato. Grazie a Luca, Paolo, Ciccio Boccuni, Hajar, Valeria, Fra, Ciccio Roccuzzo, Anna, Ivan, Annacarla, Aurel, Beppe, Fabio Proietti e Guaraldi, Jorgo, Sonia, Winston, Shou e tutti gli altri miei coinquilini per la pazienza e le risate in questi tre anni di convivenza.

Grazie a tutti i miei colleghi, soprattutto Maffo per avermi spronata sempre a migliorare e ad avere più fiducia in me stessa.
Grazie a Serena, Tania, Amanda e Debora per esserci da sempre.

L'ultimo ringraziamento va a Vittorio: è in parte colpa/merito suo se ho scelto di studiare Informatica a Bologna.

A tutti loro vanno i miei più sentiti ringraziamenti.

Bibliography

[1] D. Coppersmith, *The Data Encryption Standard (DES) and its strength against attacks*, IBM Journal of Research and Development, Vol. 38 No. 3 May 1994, pp. 243-250

[2] M. Sipser, *Introduction to the Theory of Computation*, Third Edition. pp. 299-311.

[3] K. Claessen, N. Een, M. Sheeran and N. Sörensson, *SAT-solving in practice*, Discrete Event Systems 2008. WODES 2008. 9th International Workshop on, pp. 61-67, 2008.

[4] F. Massacci and L.Marraro, *Logical Cryptanalysis as a SAT problem*, Journal of Automated Reasoning 24: pp. 165–203, 2000.

[5] D. Kroening and O. Strichman, *Decision Procedures An Algorithmic Point of View*, Springer; 2008 edition (July 7, 2008) pp. 1-23.

[6] A. Asperti and A. Ciabattoni, *Logica a informatica*, McGraw-Hill Education, 2003

[7] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*, Leningrad Seminar on Mathematical Logic, September 1966.

[8] C. Drake, *Python EDA Documentation*, Release 0.28.0, Aug 26, 2017

[9] E. Biham and A. Shamir, *Differential Cryptanalysis of DES-like Cryptosystems* The Weizmann Institute of Science Department of Applied Mathematics, July 19, 1990

[10] C. E. Shannon, *Communication Theory of Secrecy Systems*, Bell System Technical Journal, vol. 28-4, pp. 656–715, 1949

[11] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C 2nd Edition*, 1996

[12] J. Katz and Y. Lindell, *INTRODUCTION TO MODERN CRYPTOGRAPHY*, Chapman and Hall/CRC; 2 edition (November 6, 2014)

[13] NIST, *FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATION 1981 GUIDELINES FOR IMPLEMENTING AND USING THE NBS DATA ENCRYPTION STANDARD*, FIPS PUB 74 1981 April 1

[14] NIST, *Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*, NIST Special Publication (SP) 800-67 Revision 1 January 2012

[15] W. Stallings, *CRYPTOGRAPHY AND NETWORK SECURITY PRINCIPLES AND PRACTICE FIFTH EDITION*, Pearson College Div

[16] A. A. Hagberg, D. A. Schult and P. J. Swart, *Exploring network structure, dynamics, and function using NetworkX*, in Proceedings of the 7th Python in Science Conference (SciPy2008), Gäel Varoquaux, Travis Vaught, and Jarrod Millman (Eds), (Pasadena, CA USA), pp. 11–15, Aug 2008

[17] L. De Meyer, S. Vaudenay, *DES S-box generator*, Cryptologia, vol. 41, no. 2, pp. 153-171, 2017

[18] A. Biere. *PicoSAT Essentials.* Journal on Satisfiability, Boolean Modeling and Computation (JSAT), vol. 4, pp. 75-97, Delft University, 2008.

[19] Soos M., Nohl K., Castelluccia C. (2009) *Extending SAT Solvers to Cryptographic Problems.* In: Kullmann O. (eds) Theory and Applica-

tions of Satisfiability Testing - SAT 2009. SAT 2009. Lecture Notes in Computer Science, vol 5584. Springer, Berlin, Heidelberg

[20] CryptoMiniSat5, URL: https://www.msoos.org/cryptominisat5/

[21] Lingeling, URL: http://fmv.jku.at/lingeling/

[22] A. Biere. *CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. In Proceedings of SAT Competition 2017* - Solver and Benchmark Descriptions, Tomas Tomas, Marijn Heule, Matti Järvisalo (editors), vol. B-2017-1 of Department of Computer Science Series of Publications B, pp. 14-15, University of Helsinki, 2017.