

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

---

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI  
Corso di Laurea Triennale in Informatica

**CRITTOANALISI LOGICA  
DI DES**

Tesi di Laurea in Sicurezza e Crittografia

**Relatore:**  
Ill.mo Dott. Ugo Dal Lago

**Presentata da:**  
Daniele Raimondi

**III Sessione  
Anno Accademico 2009/2010**



*A Guido Vecchi*



## Introduzione

La crittografia ha sempre rivestito un ruolo primario nella storia del genere umano, dagli albori ai giorni nostri, e il periodo in cui viviamo non fa certo eccezione. Al giorno d'oggi, molti dei gesti che vengono compiuti anche solo come abitudine (operazioni bancarie, apertura automatica dell'auto, accedere a Facebook, ecc.), celano al loro interno la costante presenza di sofisticati sistemi crittografici. Proprio a causa di questo fatto, è importante che gli algoritmi utilizzati siano in qualche modo certificati come *ragionevolmente sicuri* e che la ricerca in questo campo proceda costantemente, sia dal punto di vista dei possibili nuovi *exploit* per forzare gli algoritmi usati, sia introducendo nuovi e sempre più complessi sistemi di sicurezza. In questa tesi viene proposto una possibile implementazione di un particolare tipo di attacco crittoanalitico, introdotto nel 2000 da due ricercatori dell'Università "La Sapienza" di Roma, e conosciuto come "Crittoanalisi Logica". L'algoritmo su cui è incentrato il lavoro è il Data Encryption Standard (DES), ostico standard crittografico caduto in disuso nel 1999 a causa delle dimensioni ridotte della chiave, seppur tuttora sia algebricamente inviolato.

Il testo è strutturato nel seguente modo:

- il primo capitolo è dedicato ad una breve descrizione di DES e della sua storia, introducendo i concetti fondamentali con cui si avrà a che fare per l'intera dissertazione
- nel secondo capitolo viene introdotta la Crittoanalisi Logica e viene fornita una definizione della stessa, accennando ai concetti matematici necessari alla comprensione dei capitoli seguenti.
- nel capitolo 3 viene presentato il primo dei due software sviluppati per rendere possibile l'attuazione di questo attacco crittoanalitico, una libreria per la rappresentazione e la manipolazione di formule logiche scritta in Java.
- il quarto ed ultimo capitolo descrive il programma che, utilizzando la libreria descritta nel capitolo 3, elabora in maniera automatica un insieme di proposizioni logiche semanticamente equivalenti a DES, la cui verifica di soddisfacibilità, effettuata tramite appositi tools (SAT solvers) equivale ad effettuare un attacco di tipo *known-plaintext* su tale algoritmo



# Indice

<b>1</b>	<b>Data Encryption Standard</b>	<b>9</b>
1.1	Cenni Storici . . . . .	9
1.2	Cifratura a Chiave Simmetrica . . . . .	11
1.3	La Struttura di DES . . . . .	14
1.3.1	Rete di Feistel . . . . .	14
1.3.2	S-Boxes . . . . .	17
1.3.3	Cipher Function . . . . .	19
1.3.4	Key Scheduling Algorithm . . . . .	20
1.4	Attacchi a DES . . . . .	20
1.4.1	Attacco Bruteforce . . . . .	21
1.4.2	Crittoanalisi Differenziale . . . . .	22
1.4.3	Crittoanalisi Lineare . . . . .	22
1.4.4	Attacchi su versioni ridotte di DES . . . . .	23
<b>2</b>	<b>Crittoanalisi Logica</b>	<b>25</b>
2.1	Definizione Generale . . . . .	26
2.2	Accenni di Logica Proposizionale . . . . .	28
2.2.1	Formule, Modelli e Interpretazioni . . . . .	28
2.2.2	Decidibilità della Logica Proposizionale . . . . .	29
2.2.3	Forma Normale Congiuntiva . . . . .	30
2.2.4	Il Formato DIMACS . . . . .	32
<b>3</b>	<b>Samael, una Libreria Logica in Java</b>	<b>34</b>
3.1	Rappresentazione delle Formule . . . . .	34
3.1.1	Parentele, liste e funzioni . . . . .	38
3.1.2	Alberi e Grafi . . . . .	39
3.1.3	Creazione del File DIMACS e risoluzione . . . . .	40
3.2	Rappresentazione di DES Mediante Logica Proposizionale . . . . .	40
<b>4</b>	<b>DeepThought, un Tool per la Crittoanalisi Logica di DES</b>	<b>42</b>
4.1	Tradurre DES in Logica Proposizionale . . . . .	42
4.1.1	Reverse <i>Logic</i> Engineering di DES: le Permutazioni . . . . .	44
4.1.2	Reverse <i>Logic</i> Engineering di DES: Cipher Function . . . . .	45
4.1.3	Reverse <i>Logic</i> Engineering: S-Boxes . . . . .	45

4.1.4	Reverse <i>Logic</i> Engineering di DES: XORing di Array di Formule . . . . .	47
4.1.5	Reverse <i>Logic</i> Engineering di DES: Key Scheduling Algorithm . . . . .	48
4.1.6	Reverse <i>Logic</i> Engineering di DES: la Rete di Feistel . . . . .	50
4.2	Elaborazione e Semplificazione delle Formule . . . . .	52
4.2.1	Semplificazioni Avanzate . . . . .	53
4.2.2	Il metodo di Tseitin . . . . .	55
4.2.3	Tseitin Iterativo con Clause Learning tramite HashTables . . . . .	59
4.3	Dati Sperimentali . . . . .	63
4.3.1	Risultati Sperimentali Crittoanalisi: 1 round . . . . .	64
4.3.2	Risultati Sperimentali Crittoanalisi: 2 rounds . . . . .	66
4.3.3	Risultati Sperimentali Crittoanalisi: 3 rounds . . . . .	69
4.3.4	Risultati Sperimentali Crittoanalisi: 4 rounds . . . . .	71
<b>5</b>	<b>Conclusioni e sviluppi futuri</b>	<b>73</b>



# 1 Data Encryption Standard

*There are some occasions in which a man must tell half his secret, in order to conceal the rest.*

– Philip Stanhope, *Letters to His Son*

## 1.1 Cenni Storici

L'algoritmo oggi conosciuto come DES, acronimo di Data Encryption Standard, è stato originariamente sviluppato negli anni '70 da Horst Feistel, ricercatore di origine tedesca di IBM, e dal suo gruppo di lavoro<sup>1</sup>, sotto il nome di Lucifer. In quel periodo la situazione riguardante l'utilizzo di strumenti crittografici era del tutto caotica: era risaputo che i militari e la NSA possedevano conoscenze non trascurabili, ma dal punto di vista commerciale esistevano solo prodotti opachi, incompatibili tra loro e la cui sicurezza non era formalmente provata in alcun modo. Nel 1973 l'NBS (National Bureau of Standards, oggi NIST) pubblicò una “call for proposal” con lo scopo di elevare a ruolo di standard un algoritmo di cifratura a chiave privata che fosse certificabile come *praticamente* sicuro e facilmente realizzabile in hardware, in modo da favorire la compatibilità delle comunicazioni cifrate. Alcuni dei *design criteria* specificati dal bando sono i seguenti, come riportato da [1, p. 266]:

1. The algorithm must provide a high level of security.
2. The algorithm must be completely specified and easy to understand.
3. The security of the algorithm must reside in the key; the security should not depend on the secrecy of the algorithm.
4. The algorithm must be available to all users.
5. The algorithm must be adaptable for use in diverse applications.
6. The algorithm must be economically implementable in electronic devices.
7. The algorithm must be efficient to use.
8. The algorithm must be able to be validated.

---

<sup>1</sup>Formato da Roy Adler, Don Coppersmith, Edna Grossman, Alan Konheim, Carl Meyer, Bill Notz, Lynn Smith, Walt Tuchman e Bryant Tuckerman.

## 9. The algorithm must be exportable.

Il primo bando non ebbe molto successo, in quanto nessuno degli algoritmi candidati era in grado di soddisfare i requisiti richiesti. Nel 1974 NBS replicò il bando e questa volta IBM propose uno degli algoritmi che i suoi ricercatori stavano sviluppando, un cifrario che aveva la caratteristica di introdurre un elemento del tutto nuovo nel suo design: la struttura iterativa oggi conosciuta come Rete di Feistel. Questo *block cipher*, detto Lucifer, apparve subito un candidato valido e la NSA (National Security Agency) instaurò delle trattative con IBM per ottenere principalmente due modifiche all'algoritmo.

La prima richiesta riguardava la riduzione della lunghezza della chiave da 128 a 48 bits, che si risolse con un compromesso per una dimensione di 56 bit. Molti ricercatori, tra cui anche Diffie e Hellman, obiettarono che tale riduzione esponeva l'algoritmo ad un attacco bruteforce. La seconda questione, più interessante, considerando i numerosi *rumors* che generò, riguardava invece una componente fondamentale di Lucifer, ovvero le S-Boxes. Alan Konheim, uno dei ricercatori IBM, commentò in seguito [1, p. 280]:

“We sent the S-boxes off to Washington. They came back and were all different. We ran our tests and they passed.”

Alla fine degli anni '80, quando i matematici israeliani Biham e Shamir scoprirono (ma soprattutto resero pubblica) la tecnica oggi conosciuta come crittoanalisi differenziale, fecero notare come DES fosse sorprendentemente resistente a questo tipo di attacco, facendo intuire che tale tecnica fosse già conosciuta negli anni '70 da NSA e che le modifiche espressamente richieste alle S-Boxes avessero avuto come scopo quello di irrobustire l'algoritmo.<sup>2</sup>

Nel 1977 DES fu certificato e reso pubblico, divenendo così il primo esempio di cifrario dichiaratamente robusto e aperto allo studio da parte dei ricercatori di tutto il mondo. L'egemonia di DES si concluse nel 1998, anno in cui si è definitivamente smesso di considerarlo abbastanza sicuro da poter essere uno standard internazionale. Tale obsolescenza non è tuttavia causata dalla scoperta di debolezze strutturali (il *brute-forcing* è tuttora il miglior attacco conosciuto) bensì dall'incremento della potenza di calcolo

---

<sup>2</sup>Se questo non bastasse, nel 1978 l'“U.S. Senate Committee on Intelligence” investigò sulla questione, giungendo alla conclusione che NSA fosse completamente estranea alle accuse e ai sospetti sollevati negli anni precedenti. D'altro canto, NSA avrebbe potuto voler “imporre” il contenuto delle S-Boxes per avere la certezza che non fosse IBM ad aver intenzionalmente inserito delle trapdoors.

dei processori e dalla diminuzione del costo dell'hardware specializzato necessario per effettuare un attacco esaustivo sull'insieme delle (circa)  $2^{56}$  chiavi possibili. Per ovviare a questo problema si è inizialmente pensato di triplicare la lunghezza della chiave, utilizzando una triplice cifratura (triplo-DES) effettuata usando due o tre chiavi (per un totale di 112 key-bits).

$$DES_{k_3}(DES_{k_2}^{-1}(DES_{k_1}(plaintext)))$$

Questa procedura, detta *Tuchman's triple encoding* [2, p. 171], ha sostituito (a partire dal 1999) il singolo DES a livello di utilizzo internazionale. Per quanto non sia stata tuttora messa in dubbio la sua robustezza<sup>3</sup>, il suo difetto principale risiede nella lentezza causata dalle tre applicazioni dello stesso algoritmo necessarie per cifrare ogni blocco di 8 bytes. Spinto da questo ed altri motivi, NIST (ex NBS) aveva nel frattempo indetto una competizione per selezionare un nuovo algoritmo di cifratura, destinato a diventare lo standard del nuovo millennio.

Dal bando che elesse DES erano trascorsi quasi 25 anni e nel frattempo la crittografia aveva definitivamente assunto il ruolo di scienza: mentre negli anni '70 queste "calls for proposal" finivano imbarazzantemente per essere ignorate, in questo caso furono candidati ben 15 algoritmi. Ciascuno di essi fu analizzato dai tecnici del NIST e dagli altri agguerriti gruppi in gara. Durante i primi due workshops, in cui gli algoritmi vennero analizzati e testati in cerca di ogni sorta di debolezze, si restrinse la scelta a 5 di essi. In seguito al terzo (e ultimo) workshop, NIST comunicò che tutti i candidati finalisti erano estremamente validi ed apparentemente esenti da vulnerabilità, ma per motivi di flessibilità ed efficienza era da ritenersi vincitore l'algoritmo "Rijndael" ideato da John Daemen e Vincent Rijmen.

## 1.2 Cifratura a Chiave Simmetrica

Partendo da un livello di astrazione piuttosto alto, si può definire un algoritmo di cifratura come una trasformazione che riceve in input due sequenze di bits e che ne restituisce una dotata di particolari proprietà. Formalmente quindi si distinguono tre vettori di bits: il testo in chiaro (plaintext)  $P$ , il testo cifrato (ciphertext)  $C$  e la chiave di cifratura  $K$ . Dato l'algoritmo di cifratura  $E$  (da *Encryption*), la procedura mediante la quale a partire da  $C$

---

<sup>3</sup>E' anzi stato dimostrato che DES non è un gruppo algebrico, come si vedrà in seguito.

e  $K$  si ottiene  $P$  è la seguente:

$$C = E_k(P)$$

La prima e fondamentale proprietà di questa trasformazione, condizione necessaria perché  $E$  possa essere ritenuto una cifratura robusta, è che deve essere estremamente difficile (se non impossibile) risalire a  $P$  conoscendo solo  $C$  (e quindi non la chiave  $K$ ).

L'operazione inversa  $E^{-1}$  (cioè la decifratura), permette di risalire a  $P$  conoscendo  $K$  e  $C$ . Se la chiave  $K$  usata per la cifratura è la stessa che viene usata per decifrare, si parla di algoritmo di cifratura *a chiave simmetrica*.

$$P = E_k^{-1}(C)$$

Altre proprietà di questi algoritmi sono molto interessanti, soprattutto in relazione al grande valore economico che informazioni riservate o “confidenziali” possono avere, ma d'altro canto risultano spesso molto difficili da analizzare. Chiunque usi uno strumento simile per la sicurezza dei propri dati vorrà sicuramente sapere se sono per esempio presenti delle *trapdoors*, come nel caso in cui esista una *universal key*, una sorta di passepartout capace di decifrare ogni messaggio. Per quanto riguarda DES, proprio in questa ottica si è già accennato al dibattito in merito alla scelta dei valori delle S-Boxes [1, Sez. 12.3].

Un'altra proprietà interessante, di carattere algebrico, è l'esistenza di *weak keys*, cioè di chiavi  $\bar{K}$  tali che

$$P = E_{\bar{K}}(E_{\bar{K}}(P))$$

Ovviamente utilizzare chiavi affette da questa “particolare” caratteristica sarebbe del tutto inutile dal punto di vista della sicurezza e, proprio per evitare che ne venga fatto un uso accidentale, esse andrebbero conosciute a priori. Per quanto riguarda DES, è risaputo che possiede quattro *weak keys* [1, p. 280]. Nello specifico, rappresentandole nel sistema di numerazione esadecimale, esse sono:

00000000000000

000000FFFFFFF

FFFFFFFF00000000

FFFFFFFFFFFFFFFF

Le chiavi patologiche di DES sono in tutto 64 e le restanti verranno trattate in seguito.

Un'altra caratteristica importante riguarda il fatto che l'algoritmo si comporti o meno come una *closed cipher* [2, p. 170]. Un algoritmo di cifratura si dice *chiuso* sse per ogni *plaintext*  $P$  e per ogni chiave  $K_1$  e  $K_2$  si può sempre individuare una terza  $K_3$  tale che

$$E_{K_2}(E_{K_1}(P)) = E_{K_3}(P)$$

Se si utilizza un algoritmo *chiuso*, non è quindi possibile incrementarne la robustezza applicandolo ripetutamente utilizzando chiavi diverse. Dimostrare che  $E$  è chiuso equivale a dimostrare che tale trasformazione forma un gruppo algebrico.<sup>4</sup> Per quanto riguarda DES, alla cui applicazione multipla si è già accennato prima (triplo DES), la dimostrazione che non è un gruppo è stata data Campbell e Weiner [3, pp. 512-520]. Estendendo il concetto di chiusura, un algoritmo di cifratura  $E$  si dice *puro* sse per ogni chiave  $K_1$ ,  $K_2$  e  $K_3$  esiste un  $K_4$  tale che

$$E_{K_3}(E_{K_2}(E_{K_1}(P))) = E_{K_4}(P)$$

Se  $E$  è chiuso, una sua doppia applicazione non ha quindi effetto dal punto di vista della sicurezza. Se  $E$  è puro, vale lo stesso discorso per una sua applicazione tripla. E' importante notare che mentre una *closed cipher* è necessariamente pura, non vale l'implicazione inversa, cioè una *pure cipher* non è necessariamente *closed*. Prima che venisse dimostrato che DES non è un gruppo (e che ovviamente non è puro, altrimenti procedure come il triplo DES perderebbero di significato) sono stati fatte numerosi tentativi sperimentali alla ricerca di una prova schiacciante in questo senso e pare che il team che sviluppò Lucifer ne fosse già a conoscenza [1, p. 283].

Per concludere questo breve excursus, una cifratura è da considerarsi

---

<sup>4</sup>Si dice *gruppo algebrico* è una struttura formata da un insieme su cui è definita una operazione binaria che rispetti la proprietà associativa, ammetta l'esistenza dell'elemento neutro e dell'inverso.

*faithful* sse

$$E_{K_1}(P) = E_{K_2}(P) \iff K_1 = K_2$$

In altre parole, utilizzando un algoritmo di cifratura che sia *faithful* non è possibile usare due chiavi diverse per generare lo stesso ciphertext  $C$  a partire dallo stesso plaintext  $P$ . Come è intuibile, tale proprietà ha sicuramente una grande rilevanza dal punto di vista probatorio, nell'informatica forense. Nel caso di DES, esistono 12 chiavi particolari per cui questo assunto non è verificato [1, p. 280]. Queste chiavi vengono dette *semiweak keys* e sono dovute alla procedura di *key scheduling* (descritta in seguito) con cui vengono generate le sottochiavi all'interno della rete di Feistel. Nel caso delle *weak keys* infatti tale algoritmo non produce 16 differenti subkeys, ma solo due, in maniera alternata. In base allo stesso principio, esistono anche 48 chiavi (*possibly weak keys*) per cui l'algoritmo di scheduling produce soltanto 4 differenti subkeys. Le chiavi di cui diffidare se si usa DES sono quindi 64 su un totale di  $2^{56}$ .

### 1.3 La Struttura di DES

DES è un algoritmo di cifratura che opera su blocchi di 64 bits utilizzando una chiave di 56 bits. I 64 bits del plaintext subiscono delle operazioni preliminari (permutazione iniziale) e vengono poi divisi in due parti da 32 bits ciascuna (left e right). Queste due metà sono quindi pronte per essere elaborate dalle 16 iterazioni previste dalla Rete di Feistel in cui sono applicate le tipiche trasformazioni *key-dependent*. Al risultato di queste operazioni viene applicato l'inverso della permutazione iniziale e i bits vengono restituiti come output. La rete di Feistel è la componente principale di DES, ed al suo interno viene utilizzata una *cipher-function* dipendente dalla chiave, a sua volta comprendente le S-Boxes e l'algoritmo di *key-scheduling*.

#### 1.3.1 Rete di Feistel

Una Rete di Feistel è una struttura (simmetrica o meno) usata nella progettazione di algoritmi di cifratura a blocco. Come indicato in [4, pp. 170-171], rispetto ad una *substitution-permutation network* la rete di Feistel elimina la necessità di utilizzare delle S-Boxes invertibili, incentivando quindi l'assenza di un preciso comportamento strutturato all'interno dell'algoritmo e favorendo

do in questo modo l'aspetto *casuale* dell'output tipico di una pseudorandom permutation.

A Feistel network is thus a way of constructing an invertible function from non-invertible components. [4, pp. 170-171]

Tale rete elabora i bits in input attraverso una serie di *rounds* in cui viene ripetutamente applicata una particolare funzione  $f$ , non necessariamente invertibile. Nel caso di DES, come già accennato, i 64 bits del plaintext vengono divisi in due metà,  $L$  e  $R$ , che costituiscono gli inputs del primo round. In generale, per ogni round  $i$  (con  $1 \leq i \leq 16$ ) si ha la corrispondente coppia formata da  $L_i$  e  $R_i$ , (sempre di 32 bits ciascuna). L'output del round  $i$  (quindi l'input del round  $i + 1$ ) sarà dato da:

$$L_{i+1} = R_i$$

$$R_{i+1} = L_i \oplus f(R_i, K_i)$$

Dove  $f$  viene generalmente detta *round function* ed è tipica dell'algoritmo che la utilizza. Nella fattispecie, per ogni algoritmo di cifratura a blocco  $\mathcal{A}$  con round function  $g$  che opera su blocchi di  $n$  bits,  $g$  riceverà come input  $\frac{n}{2}$  bits e ne restituirà come output la stessa quantità. Nonostante la restrizione per quanto riguarda la lunghezza di input e output, la funzione in questione non deve necessariamente essere iniettiva<sup>5</sup> e suriettiva<sup>6</sup>.

Come succede anche nelle *substitution-permutation networks*, per ogni round viene derivata dalla chiave principale  $K$  una sottochiave  $K_i$  che viene utilizzata nel corrispondente round  $i$ -esimo per introdurre la necessaria dipendenza del testo cifrato dalla chiave. Nel caso di DES, la round function prende come input i 32 bits di  $R$  e i 48 bits restituiti dalla funzione  $KS(i, K)$ , che, a partire dal numero del round corrente e dalla chiave principale di cifratura, elabora la *subkey* corrispondente. Tale algoritmo di scheduling delle chiavi verrà trattato in seguito.

Una caratteristica fondamentale delle Reti di Feistel è il fatto che esse sono invertibili a prescindere dalla *round function* utilizzata e dalle funzioni impiegate all'interno di essa. Per ogni coppia  $(L_i, R_i)$  è infatti sempre

---

<sup>5</sup>Una funzione si dice iniettiva se elementi distinti del dominio hanno immagini distinte nel codominio

<sup>6</sup>Una funzione si dice suriettiva se ogni elemento del codominio è immagine di almeno un elemento del dominio. Se una funzione è iniettiva e suriettiva, si dice biiettiva (biunivoca) e ammette una funzione inversa.

possibile risalire a  $(L_{i-1}, R_{i-1})$  semplicemente calcolando

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

$$L_{i-1} = R_i \oplus f(R_{i-1}, K_i)$$

senza bisogno di invertire  $f$  in alcun modo.

In una rete di Feistel, ogni singolo round svolge un ruolo di *confusion/diffusion step*, secondo i principi esposti nel 1949 da Claude Shannon in “*Communication Theory of Secrecy Systems*”, una delle pietre miliari per quanto riguarda la teoria dell’informazione. Per “confusione” si intende l’atto di rendere il più complesse e scorrelate possibili le (inevitabili, se non nel caso dell’*one-time-pad* [5]) relazioni che intercorrono tra chiave e testo cifrato. Citando l’autore stesso in [5],

Two methods (other than recourse to ideal systems) suggest themselves for frustrating a statistical analysis. These we may call the methods of diffusion and confusion. 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.

La ripetuta iterazione di ogni round della Rete di Feistel (e quindi la ripetuta applicazione dei principi sopra citati) fa in modo che, al termine della computazione, ogni piccola modifica dei dati in input produca un grande cambiamento sui bits in output. Questo comportamento è detto *avalanche effect* e si ottiene sse, dato un sufficiente numero di rounds<sup>7</sup>, le seguenti due

---

<sup>7</sup>Un buon numero di rounds può essere condizione necessaria ma non certamente sufficiente per ottenere un buon livello di sicurezza. Come è già stato detto, DES utilizza 16 rounds ed è una cifratura intrinsecamente robusta (la principale debolezza è la scarsa dimensione della chiave). GOST, un algoritmo “coetaneo” di DES ma sviluppato in Unione Sovietica, (è considerato robusto ma è certamente stato studiato meno del “rivale” americano) fa uso di una rete di Feistel che prevede di 32 rounds ma ha anche un “avalanche effect” che si presenta in maniera più lenta. In questo caso quindi questi fattori



condizioni sono verificate:

- cambiare un singolo bit nell'input delle S-Boxes provoca la modifica di almeno due bits nell'output delle stesse
- le permutazioni e gli *swapping* previsti in ogni round fanno in modo che i bits di output di una data S-Box saranno input di una S-Box differente nel round successivo.

L'avalanche effect di DES è decisamente marcato e inizia ad essere visibile a partire dal quarto round, come testimoniano anche le evidenze sperimentali rilevate da tecniche di crittoanalisi logica [2, p. 196]. Come mostrato in [1, p. 284], incrementando il numero di *rounds* si ottiene che

- dopo 5 rounds ogni bit del *ciphertext* è funzione di ogni bit del *plaintext* e della chiave,
- dopo 8 rounds  $C$  è il risultato di una funzione casuale di ogni bit di  $P$  e  $K$ .

La domanda che sorge spontanea a questo punto è: “Perché proprio 16 rounds? Perché 8 non bastavano?”. Con il passare degli anni, diverse varianti di DES che utilizzavano un numero ridotto di *rounds* sono state forzate. Il limite di tre è stato infranto nel 1982 e ben presto si è arrivati a 6 *rounds*. Con la scoperta della crittoanalisi differenziale, da parte di Biham e Shamir, è apparso evidente che qualsiasi numero di *rounds* minore di 16 avrebbe permesso di forzare DES con tale attacco in maniera più efficiente di quanto sia possibile fare con un bruteforcing, e forse questa è la spiegazione più logica della scelta di tale numero.

### 1.3.2 S-Boxes

Le Substitution Boxes sono una componente tipica degli algoritmi di cifratura a chiave simmetrica e, come il nome suggerisce, sono usate per operare delle sostituzioni sui dati che ricevono in input, secondo un particolare schema interno.<sup>8</sup> Tipicamente una S-Box prende come input una stringa di  $n$  bits e ne restituisce una lunga  $m$ , con  $m = n$  come condizione necessaria se la

---

si “equilibrano”. In generale viene usato un numero di rounds  $\geq 7$ .

<sup>8</sup>Schema che, in accordo con i principi di Auguste Kerckhoffs, può essere reso pubblico, così come è stato fatto per le specifiche di DES.

funzione applicata deve essere iniettiva e suriettiva.<sup>9</sup> A seconda dell'implementazione<sup>10</sup>, la funzione di mapping tra input e output può essere basata su tabelle statiche di  $2^n$  locazioni di dimensione  $m$  (come nel caso di DES) o può essere calcolata dinamicamente in funzione della chiave.

Come è intuibile da quanto accennato in precedenza, contrariamente a quanto potrebbe sembrare, il modo più affidabile per progettare delle S-Boxes non è quello di scegliere le sostituzioni da effettuare in maniera casuale. Il motivo più evidente di questo fatto è che, anche restringendo la casualità per essere certi di generare una funzione iniettiva e suriettiva, non si potrebbe avere la certezza che le condizioni per ottenere un *avalanche effect* (1.3.1) siano rispettate. Il secondo motivo, che come si è visto era già conosciuto da NSA durante il processo di standardizzazione di DES, consiste nel fatto che è possibile scegliere dei valori strutturati in modo tale da rendere praticamente inefficace la crittoanalisi differenziale e lineare. Le 8 S-Boxes utilizzate da DES all'interno della *cipher function* impiegata in ogni round sono state quindi accuratamente scelte per essere particolarmente refrattarie ad eventuali attacchi crittoanalitici. Alcune sperimentazioni hanno anche mostrato che una implementazione di DES che utilizzi S-Boxes generate casualmente risulta essere più vulnerabile dell'originale. Negli anni '70, sull'onda dei sospetti secondo cui NSA avrebbe nascosto una *trapdoor* all'interno delle S-Boxes modificate, esse furono analizzate a fondo, senza però che i risultati indicassero debolezze particolari. I riscontri ottenuti avevano tuttavia evidenziato la presenza di alcune "strutture" che le rendevano più simili a delle trasformazioni lineari piuttosto che al frutto di una scelta casuale di valori [1, p. 284]. Anni dopo, le linee guida usate nel design delle S-Boxes e delle P-Boxes vennero rese pubbliche. Di seguito ne sono riportate alcune (riportate da [1, pp. 293-294]):

- Each S-box has 6 input bits and 4 output bits. (This was the largest size that could be accommodated in a single chip with 1974 technology.)
- No output bit of an S-box should be too close to a linear function of the input bits.

---

<sup>9</sup>Se le S-Boxes sono utilizzate in una Rete di Feistel, come si è visto, la biunivocità (e quindi l'invertibilità) non è un requisito fondamentale.

<sup>10</sup>Le specifiche di DES sono descritte in [24].

- If you fix the left-most and right-most bits of an S-box and vary the 4 middle bits, each possible 4-bit output is attained exactly once.

- If two inputs to an S-box differ in exactly 1 bit, the output must differ in at least 2 bits.

- If two inputs to an S-box differ in the 2 middle bits exactly, the output must differ in at least 2 bits.

- If two inputs to an S-box differ in their first 2 bits and are identical in their last 2 bits, the two outputs must be not the same.

Ogni S-Box di DES è di tipo statico e riceve come input 6 bits restituendone 4. La rappresentazione tipica è data da una *lookup-table* di 4 righe e 16 colonne, con celle da 4 bits. I 6 bits di input forniscono sostanzialmente delle coordinate in due dimensioni per identificare la cella da restituire in output:

**1 1001 0**

In questo esempio, il primo e l'ultimo bit indicano la riga della tabella e i 4 bits centrali indicano la colonna: le coordinate ottenute sono quindi date dalla coppia (9, 2).

Le S-Boxes di DES ovviamente rispettano le proprietà necessarie per ottenere un *avalanche effect* crescente rispetto al numero di rounds. Nella fattispecie, [4, p. 175] riporta:

1. Each S-Box is a 4-to-1 function. (That is, exactly 4 inputs are mapped to each possible output.) This follows from the properties below.

2. Each row in the table contains each of the 16 possible 4-bit strings exactly once. (That is, each row is a *permutation* of the 16 possible 4-bit strings)

3. Changing *one bit* of the input always changes at least two bits of the output.

### 1.3.3 Cipher Function

La cipher function di DES è invocata in in ogni round come  $f(R_i, K_i)$  e contiene tutti gli elementi non invertibili finora citati (e anche alcuni che sono

stati omessi). Tale funzione applica prima di tutto una *expanding permutation* ai 32 bits di  $R$ , ottenendone 48. Questi 48 bits vengono quindi messi in XOR con la subkey  $K_i$  (generata dall'algoritmo di scheduling delle chiavi) e anch'essa di 48 bits. Il risultato di questa operazione diviene l'input per le 8 S-Boxes, che restituiscono i 32 bits "finali", che, dopo aver subito una ulteriore semplice permutazione, andranno a costituire l'output della cipher function  $f$ .

### 1.3.4 Key Scheduling Algorithm

L'algoritmo di *key scheduling*, tipico di strutture crittografiche simili alla rete di Feistel, si occupa di generare (a partire dalla chiave di cifratura principale) le 16 *subkeys* che verranno poi utilizzate nel corrispettivo round dell'iterazione principale. Alla *master key*  $K$  viene applicata una particolare permutazione (che ha come scopo l'eliminazione dei bit di parità) che restituisce 56 bits. Tali bits sono divisi in due metà,  $C$  e  $D$ . A partire da questi due gruppi di 28 bits ciascuno, a seconda dell'indice della *subkey* richiesta, si applicano una serie di *left shifts* (doppi o singoli). Data la *subkey* destinata al round  $i$ -esimo, è possibile calcolare quella corrispondente al round  $(i + 1)$ -esimo in maniera incrementale, semplicemente conservando le due metà  $C_i$  e  $D_i$  in modo da potervi applicare gli shifting necessari ad ottenere la sottochiave successiva. Il numero dei *left shifts* da applicare per ogni round è un dato fornito attraverso una tabella statica, così come lo è il comportamento della permutazione iniziale applicata alla chiave e di quella applicata dopo il concatenamento di  $C$  e  $D$ . Come visto in precedenza, la funzione di scheduling delle chiavi è "responsabile" di 60 delle 64 chiavi che manifestano un comportamento patologico.<sup>11</sup>

## 1.4 Attacchi a DES

Ogni attacco crittoanalitico può essere, indipendentemente dalla sua natura, classificato a priori in base ai requisiti di cui necessita per poter andare a buon fine. Le categorie di questa divisione, indicate da [4, p. 161], sono le seguenti:

---

<sup>11</sup>Ovvero le 12 semi-weak keys più le 48 possibly-weak keys.

- Ciphertext-only attacks, where the attacker is given only a series of outputs  $\{F_k(x_i)\}$  for some inputs  $\{x_i\}$  unknown to the attacker
- Known-plaintext attacks, where the attacker is given pairs of inputs and outputs  $\{(x_i, F_k(x_i))\}$
- Chosen-plaintext attacks, where the attacker is given  $\{(x_i, F_k(x_i))\}$  for a series of inputs  $\{x_i\}$  that are chosen by the attacker
- Chosen-ciphertext attacks, where the attacker is given  $\{(x_i, F_k(x_i))\}$  and  $\{(F_k^{-1}(y_i), y_i)\}$  for  $\{x_i\}, \{y_i\}$  chosen by the attacker.

#### 1.4.1 Attacco Bruteforce

La tipologia di attacco più semplice (dal punto di vista concettuale) è sicuramente quella che prevede la ricerca esaustiva della chiave corretta all'interno dell'insieme delle chiavi possibili. Tale attacco è solitamente di tipo *known-plaintext* ed è "l'ultima spiaggia" su cifrari considerati sicuri dal punto di vista delle altre tecniche, sicuramente più ingegnose. Il *bruteforcing* è per definizione applicabile ad ogni algoritmo e richiede un quantitativo minimo di dati: per essere effettuato su DES sono teoricamente necessari solo i 64 bits del plaintext e i corrispondenti 64 del ciphertext.<sup>12</sup> Per quanto riguarda la dimensione del problema, se la lunghezza della chiave è di  $n$  bits, la cardinalità dello spazio delle chiavi possibili è data dalle disposizioni con ripetizione di 2 elementi di classe  $k$ , cioè  $2^n$ . Mediamente, dopo  $2^{n-1}$  tentativi la chiave sarà stata già individuata con una probabilità di  $\frac{1}{2}$ . La sicurezza di un sistema crittografico è quindi quantificabile mettendo in relazione la massima probabilità di successo tollerata  $0 \leq \varepsilon \leq 1$  con uno specifico periodo di tempo  $t \geq 0$  durante il quale viene tentato l'attacco [4, p. 49]:

A [cryptographic] scheme is  $(t, \varepsilon)$ -secure if every adversary running for time at most  $t$  succeeds in breaking the scheme with probability at most  $\varepsilon$ .

In questo caso è conveniente misurare il tempo  $t$  in cicli di CPU e si può quindi dire che un algoritmo è sicuro (dal punto di vista del bruteforcing) se un attacco esaustivo sull'insieme delle possibili chiavi, utilizzando le migliori

---

<sup>12</sup>Ottenere una coppia (P, C) è più semplice di quanto si possa immaginare: la maggior parte delle trasmissioni cifrate hanno delle parti "standard" obbligatorie, per giunta spesso molto più lunghe di soli 8 bytes.

risorse computazionali disponibili, richiede un ammontare di tempo sufficientemente ampio da renderlo praticamente inutilizzabile. Alcuni approcci *sui generis* a questo attacco sono accennati da [1, p. 156].

#### 1.4.2 Crittoanalisi Differenziale

Questo metodo è stato introdotto da Eli Biham e Adi Shamir [6, pp. 3-72] nel 1990 ed è di tipo *chosen-plaintext*. Dato un certo numero di plaintexts costruiti in modo che possiedano delle particolari differenze, l'analisi delle discrepanze ottenute nei corrispondenti ciphertexts può portare a formulare delle ipotesi probabilistiche riguardo alla chiave usata. Maggiore è il numero delle coppie  $(P, C)$  più accurata sarà la previsione riguardo alla chiave. Questa tecnica non ha condotto ad attacchi effettivamente applicabili su DES [1, p. 289] perché richiede  $2^{37}$  applicazioni di DES durante l'analisi ed una enorme quantità di dati:  $2^{47}$  coppie  $(P, C)$  se applicata come *chosen-plaintexts* e  $2^{55}$  coppie se utilizzata come *known-plaintexts*. Come si è già detto, ci sono testimonianze [1, p. 290] che uno dei *design criteria* delle S-Boxes fosse proprio la capacità di resistere a questo attacco, che era già stato scoperto (ma secretato) da NSA e IBM.

#### 1.4.3 Crittoanalisi Lineare

La crittoanalisi lineare è stata inventata da Mitsuru Matsui nei primi anni '90 e, rispetto all'approccio differenziale, ha il pregio di essere pensata come *known-plaintexts attack*. Questo metodo considera, come il nome suggerisce, le relazioni lineari tra input e output [2, pp. 165-203]. Concettualmente, per identificare una buona approssimazione lineare di DES, è necessario riuscire ad approssimare ciascun round e combinarli insieme. L'attacco base usa  $2^{27}$  plaintexts e restituisce due bits della chiave. Una versione più sofisticata di questo metodo utilizza l'approssimazione lineare in 14 dei 16 rounds e quindi cerca di "indovinare" i valori che i bits assumono nel primo e nell'ultimo round, riuscendo a risalire a 26 dei 56 bits della chiave. L'attacco è quindi portato a termine da una ricerca esaustiva della chiave esatta tra le rimanenti. Per andare a buon fine su una istanza completa di DES, sono necessari circa  $2^{43}$  known-plaintexts. Tale attacco non era conosciuto durante lo sviluppo di Lucifer e quindi non sono stati previsti particolari accorgimenti per renderlo inefficace [1, p. 292].

#### 1.4.4 Attacchi su versioni ridotte di DES

Varianti di DES semplificate, con un numero limitato di rounds, sono soggette ad attacchi specifici (a scopo prevalentemente didattico). In questi casi DES non è una *pseudorandom function* perché prima del quarto round l'*avalanche effect* è ancora ben lungi dall'essere completo.

**DES con Round Singolo** Nel caso più semplice [4, pp. 176-177], quello in cui viene utilizzato un solo round, si ha un output costituito dalla coppia  $(L_1, R_1)$  dove

$$L_1 = R_0$$

$$R_1 = L_0 \oplus f_1(R_0, K_1)$$

da cui si ricava che

$$f_1(R_0, K_1) = L_0 \oplus R_1$$

in cui sia  $L_0$  che  $R_1$  sono noti. Applicando l'inverso della *expanding permutation* usata in  $f_1$ , si risale direttamente all'output delle S-Boxes, ciascuna rappresentata da un gruppo di 4 bits. Dato che le S-Boxes implementano una funzione 4 a 1, si ha che per ognuna di esse esistono quattro possibili gruppi di 6 bits che avrebbero potuto restituire il valore ottenuto. All'interno della *cipher function*, l'input delle S-Boxes è dato da  $E(R_0) \oplus K_0$  e, trattandosi di un attacco *known-plaintext*,  $R_0$  è conosciuto. Partendo da questo presupposto appare chiaro che ogni gruppo di 6 bits di  $K_1$  può assumere solo 4 valori distinti, restringendo quindi il numero delle possibili chiavi a  $2^{16}$ , che può facilmente essere oggetto di un attacco forza bruta.

**DES con Due Rounds** Nel caso in cui vengano utilizzati due soli rounds [4, pp. 176-177], l'output dell'algoritmo è dato dalla coppia  $(L_2, R_2)$  dove

$$L_1 = R_0$$

$$R_1 = L_0 \oplus f_1(R_0, K_1)$$

$$L_2 = R_1 = L_0 \oplus f_1(R_0, K_1)$$

$$R_2 = L_1 \oplus f_2(R_1, K_2)$$

In questo caso,  $R_0$ ,  $L_0$ ,  $L_2$  e  $R_2$  sono noti e anche  $L_1$  e  $R_1$  possono essere ricavati con estrema facilità. In tali condizioni, dato che l'input e l'output di entrambe le cipher function utilizzate è conosciuto, è possibile applicare per ciascuna di esse il metodo visto per il caso con il round singolo. La complessità dell'attacco è quindi di  $2 \cdot 2^{16}$  e può ulteriormente essere ottimizzata considerando il fatto che le sottochiavi  $K_1$  e  $K_2$  utilizzate non sono completamente distinte bensì condividono alcuni bits.

**DES con Tre Rounds** Il discorso inizia a farsi più complesso nel caso di DES con tre rounds: gli attacchi visti in precedenza non sono più di aiuto poiché non sono sempre conosciuti inputs e outputs delle S-Boxes e si è quindi costretti a cercare di "indovinare" il valore dei key bits. Dati  $L_1 = R_0$  e  $R_2 = L_3$ , rimane comunque incognito il valore assunto da  $R_1 = L_2$ . In tal caso [4, pp. 176-177] la complessità finale è inferiore a  $2^{30}$  e richiede uno spazio di  $2 \cdot 2^{12}$ .



## 2 Crittoanalisi Logica

MEFISTOFELE: *Figliuolo, fate buon uso del tempo, che, oimè, fugga sì rapido. Nondimeno chi ha ordine ha tempo; e perciò io vi consiglio innanzi tutto lo studio della logica. Per esso vi sarà ben indirizzato l'intelletto.*

– J. F. Goethe, *Faust*

La crittoanalisi logica è una tecnica introdotta da F. Massacci e L. Marzaro [2, pp. 174-198], ai cui si deve anche la prima applicazione sperimentale su DES. Questo spunto ha portato ad altri lavori [7, pp. 244-257] che si sono prevalentemente concentrati su algoritmi di cifratura caratterizzati da una bassa complessità, come per esempio alcune *stream ciphers* di uso comune. Questa tecnica, di cui in seguito verrà data una definizione più estesa, consiste sostanzialmente nel ridurre l'algoritmo  $\mathcal{A}$  oggetto dell'attacco ad un sistema di proposizioni logiche semanticamente equivalenti ad  $\mathcal{A}$ . Assumere che questa operazione sia sempre realizzabile è del tutto naturale: tali algoritmi sono nella maggior parte dei casi direttamente progettati per essere implementati in maniera efficiente tramite hardware, attraverso una loro rappresentazione circuitale che altro non è se non una formulazione rispondente alla logica proposizionale e all'algebra booleana.

In algebraic attacks, equations are constructed that express the output bits of a cipher in terms of its inputs, or its state. These equations are then solved and reasoned about with either dedicated equation solvers such as the F5 algorithm, or standard SAT solvers. [7, pp. 244-257]

Come sarà accennato in seguito, il sistema di proposizioni in questione non deve necessariamente appartenere alla logica proposizionale, sebbene tale forma sia la meglio gestibile dal punto di vista computazionale e da quello dei *tools* (SAT solvers e ragionatori) attualmente disponibili.

Una volta ottenuto  $S_{\mathcal{A}}$ , il sistema di proposizioni logiche semanticamente equivalente ad  $\mathcal{A}$ , intuitivamente si può operare su di esso assumendo (con i dovuti accorgimenti) che i risultati ottenuti siano validi anche per  $\mathcal{A}$ .

The intuition behind logical cryptanalysis is as simple as that. Once we have the formula describing the cipher, the cryptanalysis problems we have described can be easily formalized. [2, p. 175]

La caratteristica peculiare che differenzia la crittoanalisi logica dalle altre tecniche è che è possibile sfruttare la rappresentazione logica dell'algoritmo che si sta studiando anche per cercare di ottenere risultati che esulano dal semplice attacco. Trattandosi di formule logiche da “dare in pasto” a ragionatori e SAT solvers, è possibile usare il potere espressivo della matematica per formalizzare proposizioni da verificare in maniera automatica. Descrivendo in maniera appropriata il problema<sup>13</sup>, è teoricamente possibile verificare proprietà come quelle viste nella sezione 1.2.

I tools utilizzati nel caso specifico sono tre tipi diversi di SAT solvers. Questi programmi, date loro delle proposizioni logiche in CNF (Conjunctive Normal Form) rappresentate secondo un preciso formato (detto DIMACS CNF), si propongono di risolvere il problema della decidibilità della logica proposizionale, stabilendo se una formula ben formata  $F$  sia soddisfacibile o meno. Tale problema è (sfortunatamente) piuttosto arduo (NP-completo) e verrà brevemente esposto in seguito, insieme agli altri temi trattati in questo paragrafo.

## 2.1 Definizione Generale

La differenza principale tra la crittoanalisi logica e gli attacchi visti finora consiste nel fatto che, dato il plaintext  $P$ , la chiave  $K$  e il corrispondente ciphertext  $C = E(P, K)$ , essi non sono considerati più come *sequenze di bits* bensì come sequenze di variabili logiche, vere se il corrispondente bit è posto a 1 e false altrimenti. Partendo da questo presupposto, anche l'algoritmo di cifratura in analisi deve essere formalizzato attraverso la sua codifica in una formula logica (semanticamente equivalente)  $\mathcal{E}(P, K, C)$  tale che  $\mathcal{E}$  sia soddisfacibile sse  $C = E(P, K)$  è verificato, cioè

$$\mathcal{E}(P, K, C) \iff C = E(P, K)$$

Dal punto di vista pratico, occorre notare che se  $\mathcal{E}(P, K, C)$  è verificata, non necessariamente  $\mathcal{D}(C, K, P)$  lo è. Questo deriva dall'osservazione che, come descritto in [2, p. 175],  $\mathcal{E}(P, K, C)$  non costituisce un modello del fatto che decifrando  $C$  tramite  $K$  sia possibile risalire a  $P$ . Infatti, per quanto questi algoritmi siano definiti “simmetrici”, esistono sempre delle piccole differenze

---

<sup>13</sup>Nulla vieta di usare anche logiche più espressive di quella proposizionale, come per esempio quella del primo ordine (QBF, Quantified Boolean Formulae)

tra il processo di cifratura e il suo inverso, quello di decifratura: nel caso di DES tale variazione riguarda l'algoritmo di scheduling delle chiavi, ed è più che sufficiente a rendere  $\mathcal{D}(P, K, C)$  diversa da  $\mathcal{E}(P, K, C)$ .

A questo punto si può definire  $v_C$  come l'insieme ordinato dei valori di verità (**true/false**) assunti dal testo cifrato e  $v_P$  come quello dei valori assunti dal testo in chiaro. Verrà usata invece la notazione  $\mathcal{V}_K$  per indicare le 56 variabili booleane rappresentanti i bits della chiave. La ricerca della chiave nel caso di un attacco *ciphertext-only* si riduce quindi al trovare un modello<sup>14</sup> per la formula  $\mathcal{E}(P, K, v_C)$ , mentre nel caso si effettui un *known-plaintext* attack<sup>15</sup> è necessario verificare la soddisfacibilità di  $\mathcal{E}(v_P, K, v_C)$ . L'attacco trattato nel caso di studio (e quindi di cui si parlerà da qui in avanti) è del secondo tipo. Tale fatto permette incidentalmente anche di ottenere delle semplificazioni in grado di rendere più trattabile la dimensione delle formule da elaborare e risolvere. Indipendentemente dalle strategie implementative<sup>16</sup> attraverso cui si ottiene (e si elabora)  $\mathcal{E}$ , che come vedremo in seguito possono portare (per ragioni di convenienza) all'introduzione di nuove variabili ausiliarie, dal punto di vista concettuale le uniche variabili di controllo sono le 56  $\mathcal{V}_K \in K$ .

Al crescere del numero di round di DES, l'effetto "valanga" ha più tempo per propagarsi sui bits di  $C$  e, secondo lo stesso principio, anche la formula  $\mathcal{E}$  che lo modella si arricchisce di *constraints* che riducono il numero delle soluzioni possibili, tendendo verso una condizione di unicità delle chiavi. La formula  $\mathcal{E}_1(v_P, K, v_C)$ , semanticamente equivalente a DES con un solo round, ammette invece numerose soluzioni per ogni coppia  $(v_P, v_C)$  fissata proprio perché le propagazioni delle influenze sui bits di  $C$  da parte di  $P$

---

<sup>14</sup>Tale assegnamento di valori di verità alle variabili che compongono i vettori  $P$  e  $K$  è unico sse, fissato un certo  $C$ ,  $\exists$  (ed è unica) la coppia  $(P, K)$  tale che  $E(P, K) = C$ . Questo assunto non è verificato a causa della maggiore cardinalità del dominio rispetto al codominio, fattore che impedisce l'iniettività di  $E$ . Anche in queste condizioni però, aumentando il numero dei ciphertexts impiegati, è possibile far convergere il numero delle soluzioni accettabili a 1.

<sup>15</sup>In tal caso, (dal punto di vista teorico) le uniche variabili incognite sono quelle di  $K$ , che rappresentano i bits della chiave. Se  $v_C$  e  $v_P$  assumono valori consistenti ed  $\mathcal{E}$  rappresenta una istanza di DES completa (16 rounds), per la proprietà della *faithfulness* (rispettata da DES), si ha che  $K$  esiste ed è unico. In altre parole, se  $K$  esiste per i  $v_C$  e  $v_P$  calcolati su DES a 16 rounds, esso è unico.

<sup>16</sup>Massacci e Marraro [2] non disponevano di una generazione delle formule completamente automatizzata, per cui hanno preferito l'elaborazione *una tantum* di una  $\mathcal{E}$  generica (estremamente ottimizzata) in cui sostituire di volta in volta le variabili e i valori di verità necessari. Nel caso di studio specifico, si invece è ricorso a una creazione *ex novo* di una  $\mathcal{E}$  per ogni istanza, in maniera dipendente dalla scelta di  $P$  e dal numero di rounds.

e  $K$  sono ben lungi dall'essere complete. In linea teorica, per effettuare un attacco *known-plaintext* su DES completo, è necessario un solo blocco di *plaintext* (64 bits) con il corrispondente *ciphertext*. Per restringere il campo delle soluzioni accettabili su varianti di DES in cui l'*avalanche effect* non si è concluso, si rende invece necessario [2, p. 175] partire da un numero  $n$  più ampio di coppie  $(v_P, v_C)$  ottenute a partire dalla stessa chiave  $K$ :

$$\bigwedge_{i=1}^n \mathcal{E}(v_P^i, K, v_C^i)$$

Sperimentalmente si è verificato che al crescere del numero di rounds il numero dei modelli per  $\mathcal{E}$  diminuisce e che le istanze diventano via via più complicate da risolvere.

Nello specifico, è possibile verificare<sup>17</sup> che il modello che soddisfa  $\mathcal{E}$  è unico (scegliendo  $K$  in modo che non appartenga all'insieme delle *weak keys*) calcolando a priori<sup>18</sup> i valori  $(v_P, v_C)$  per la  $K$  scelta e quindi verificando (con l'aiuto di un SAT solver) che tale chiave è soluzione unica di  $\mathcal{E}(v_P, K, v_C)$ . Questa operazione può essere fatta (a condizione di conoscere la chiave usata) congiungendo ( $\forall i = 1, \dots, 56$ ) la  $\mathcal{E}$  ottenuta con  $\mathcal{V}_i$ , se il bit  $i$ -esimo di  $K$  è 1, e con  $\neg\mathcal{V}_i$  se esso è uguale a 0. Se  $K$  è unico, risolvendo la formula così ottenuta si otterrà come responso UNSAT.

## 2.2 Accenni di Logica Proposizionale

Questa sezione non ha la pretesa di essere esaustiva: verranno ripresi alcuni concetti necessari per spiegare il lavoro svolto, rimandando a [8] e [9] per una trattazione approfondita.

### 2.2.1 Formule, Modelli e Interpretazioni

Data una formula ben formata (FBF)  $\mathcal{F}$  [8, p. 6] e una interpretazione  $\mathcal{I}$ , definita in [8, p. 9] e in [9, p. 203], se  $\mathcal{I}(\mathcal{F}) = 1$ , allora si dice che  $\mathcal{I}$  è un modello per  $\mathcal{F}$ . Da questo deriva che  $\mathcal{F}$  è soddisfacibile se ha almeno un modello,

<sup>17</sup>Questo procedimento è attuabile sempre: l' $\mathcal{NP}$ -completezza di SAT implica che esso sia verificabile in tempo polinomiale se si conosce a priori la soluzione, in questo caso costituita dalla conoscenza della chiave.

<sup>18</sup>Una volta scelto il numero  $n$  di rounds e i valori di  $v_P$  e  $v_K$ , quelli di  $C$  possono essere ottenuti applicando DES mediante un tool esterno oppure, più comodamente, è possibile generare la formula  $\mathcal{E}_1(v_P, v_K, \mathcal{V}_C)$  e risolverla in funzione delle uniche variabili presenti, cioè le  $\mathcal{V}_C \in C$ .

cioè se esiste almeno una interpretazione che la soddisfa. In caso contrario, si dice che  $\mathcal{F}$  è contraddittoria o insoddisfacibile. Se ogni interpretazione  $\mathcal{I}$  è un modello per  $\mathcal{F}$  (detto in altri termini, se  $\neg\mathcal{F}$  è insoddisfacibile), si dice che  $\mathcal{F}$  è una tautologia.

Per verificare se  $\mathcal{F}$  sia soddisfacibile ( $\exists \mathcal{I}$  tale che  $\mathcal{I}(\mathcal{F}) = 1$ ) o contraddittoria ( $\mathcal{I}(\mathcal{F}) = 0 \forall \mathcal{I}$ ) è necessario verificare il comportamento della formula per tutti<sup>19</sup> gli assegnamenti possibili di valori di verità (0, 1) alle variabili contenute in  $\mathcal{F}$ . Se tali variabili sono  $n$ , appare chiaro che esistono  $2^n$  possibili interpretazioni, che solitamente vengono esaminate in maniera esaustiva [8, p. 13] tramite una tabella di verità<sup>20</sup>.

## 2.2.2 Decidibilità della Logica Proporzionale

Come si è visto, la verifica della soddisfacibilità tramite tabelle di verità ha un comportamento esponenziale rispetto alla dimensione della formula. Se fosse possibile realizzare un algoritmo che risolvesse questo problema in tempo polinomiale, la crittoanalisi logica diverrebbe probabilmente uno strumento estremamente potente al quale solo algoritmi del tutto particolari [2, p.200] (e non formalizzabili dal punto di vista logico) potrebbero resistere<sup>21</sup>. Come illustrato in [8, p. 14], non è tuttora stato dimostrato che SAT sia effettivamente esponenziale, ma è d'altro canto evidente che se fosse polinomialmente risolubile, anche una serie di importanti problemi di calcolo (per cui allo stesso modo non è conosciuta una soluzione efficiente), potrebbero essere risolti utilizzando lo stesso metodo.

La classe di problemi a cui si fa riferimento è quella dei problemi  $\mathcal{NP}$ -completi. Data la classe  $\mathcal{P}$  dei problemi risolvibili in tempo polinomiale,  $\mathcal{NP}$  indica invece la classe dei problemi risolvibili in tempo polinomiale da un algoritmo non deterministico<sup>22</sup>, capace cioè di compiere sempre (*indovinando*) la scelta più conveniente per quanto riguarda la soluzione del problema.

<sup>19</sup>In realtà, è necessario controllare tutti gli assegnamenti solo se si desidera verificare la *non* soddisfacibilità. Per verificare che  $\mathcal{F}$  sia soddisfacibile basta infatti verificare che esista almeno un modello.

<sup>20</sup>Sfortunatamente, al crescere di  $n$ , tale metodo diviene molto velocemente impraticabile.

<sup>21</sup>Questo potrebbe essere il caso di RSA, dato che è principalmente basato sulla teoria dei numeri invece che su concetti più vicini alla logica.

<sup>22</sup>Le soluzioni positive di un problema  $\mathcal{NP}$  possono essere *verificate* in tempo polinomiale, (come vedremo in seguito). Non è chiaro se questo implichi che esse possono essere anche *calcolate* con la stessa facilità.

Questa formulazione equivale a dire che tale algoritmo è in grado di “sdoppiare” il proprio flusso di esecuzione ogniqualvolta incontri un “bivio” costituito da una decisione da prendere. Al termine di questa computazione dal sapore quantistico, sarà stato visitato l’intero albero decisionale (formato da tutte le scelte che era possibile compiere) e verrà restituito il valore trovato da quel flusso di esecuzione che, dalla radice alla foglia finale, ha “azzeccato” tutti i bivi decisionali in maniera corretta. Partendo da questa definizione (più intuitiva che formale), appare chiaro come SAT sia a tutti gli effetti un problema  $\mathcal{NP}$ : gli assegnamenti dei valori di verità alle variabili possono essere considerati bivi decisionali e quindi essere “indovinati” dall’ipotetico algoritmo non deterministico capace di risolvere SAT polinomialmente.

La classe dei problemi  $\mathcal{NP}$ -completi comprende intuitivamente i problemi “più difficili” che appartengano anche a  $\mathcal{NP}$ , cioè quelli che meno probabilmente appartengono a  $\mathcal{P}$ .<sup>23</sup> I problemi  $\mathcal{NP}$ -completi hanno la particolarità di essere formalmente definiti in modo da essere polinomialmente riconducibili gli uni agli altri. Questo comporta che trovare un algoritmo che risolva uno di essi in tempo polinomiale permetterebbe per definizione di risolvere *tutti* i problemi  $\mathcal{NP}$ -completi nello stesso tempo. La  $\mathcal{NP}$ -completezza di SAT è stata dimostrata nel 1971 da Stephen Cook.

### 2.2.3 Forma Normale Congiuntiva

La Conjunctive Normal Form ha una grossa importanza per quanto concerne l’utilizzo di SAT solver, dato che essi generalmente richiedono che le formule da elaborare siano fornite in questa forma normale, codificate secondo il formato DIMACS.

Definiamo prima di tutto il concetto di *letterale* e *clausola* sfruttando le definizioni date in [9, p.204]: si dice *letterale* una *formula ben formata* che può essere o una formula atomica o  $\neg R$ , dove  $R$  è una proposizione atomica. Una *clausola* è invece una disgiunzione  $R_1 \vee R_2 \vee \dots \vee R_n$ , in cui ogni  $R_i$  è un letterale e in cui nessuna formula atomica compare più di una volta.

Una FBF  $\mathcal{F}$  si dice in forma normale congiuntiva (CNF) sse

$$\mathcal{F} = F_1 \wedge \dots \wedge F_n$$

---

<sup>23</sup>E’ infatti noto che  $\mathcal{P} \subset \mathcal{NP}$ , ma non è ancora chiaro se l’inclusione sia stretta o meno, cioè se sia le due classi coincidano.

con  $n \geq 1$ , e  $\forall i = 1, \dots, n$   $F_i$  è una disgiunzione di clausole o di letterali. È importante notare che per ogni formula  $\mathcal{F}$  esiste sempre una rappresentazione CNF e una DNF (Disjunctive Normal Form) equivalente a  $\mathcal{F}$ .

Data  $\mathcal{F}$ , il procedimento più intuitivo per ottenerne la rappresentazione CNF è il seguente [8, p. 24]:

- Eliminare da  $\mathcal{F}$  tutti i connettivi diversi da  $\wedge$ ,  $\vee$  e  $\neg$ , attraverso delle semplici equivalenze semantiche:

- (implicazione)  $A \rightarrow B \equiv (\neg A \vee B)$

- (se e solo se)  $A \leftrightarrow B \equiv ((A \rightarrow B) \wedge (B \rightarrow A)) \equiv ((\neg A \vee B) \wedge (\neg B \vee A))$

- (OR esclusivo)  $A \oplus B \equiv (A \leftrightarrow (\neg B)) \equiv ((\neg A \wedge B) \vee (\neg B \wedge A))$

- Utilizzare la legge della doppia negazione e le leggi di De Morgan per portare i simboli di negazione davanti ad ogni singolo letterale

- (doppia negazione)  $\neg\neg A \equiv A$

- (De Morgan)

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

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

- Utilizzare la distributività della disgiunzione (OR) rispetto alla congiunzione (AND) per ottenere come risultato finale una congiunzione di disgiunzioni, anche detta “prodotto di somme” o “prodotto di max-termini”.

- (distributività dell’OR)  $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

L’applicazione della proprietà distributiva attraverso il procedimento sopra citato è, a dire il vero, estremamente inefficiente. Nel caso più sfavorevole, quando la formula  $F$  su cui applicare tale proprietà è scritta in *Disjunctive Normal Form* (la forma normale speculare alla CNF, composta da una disgiunzione di congiunzioni) tale algoritmo effettua una sorta di “prodotto cartesiano” moltiplicando tra loro tutti i termini di ogni disgiunzione, provocando una “esplosione” esponenziale della complessità. Sfortunatamente, in queste condizioni, nessuna delle espressioni utili ai fini della crittoanalisi

logica risulterebbe lontanamente trattabile. Data l'importanza di questo problema, esso è stato oggetto di studio in molti lavori: in [11, 10] viene per esempio proposto un algoritmo che opera in tempo lineare utilizzando NICE DAGS. Generalmente, per eleganza e semplicità, le soluzioni tipiche [13] derivano dal lavoro del ricercatore russo G. S. Tseitin, che nel pluricitato (quanto ostico) articolo [15] (risalente agli anni '70) ha proposto un algoritmo per portare una espressione della logica proposizionale in CNF in tempo lineare, introducendo variabili ausiliarie. Tale algoritmo e la sua implementazione saranno discussi in seguito.

#### 2.2.4 Il Formato DIMACS

Il formato DIMACS CNF è un formato testuale che permette una rappresentazione semplice e intuitiva di formule logiche precedentemente portate in forma normale congiuntiva.

Le regole principali sono le seguenti:

- Le righe che iniziano con il carattere `c` sono considerate commento e quindi ignorate
- La prima riga di ogni file in questo formato dovrebbe<sup>24</sup> essere posta nella forma:

```
p cnf NUMERO_VARIABILI NUMERO_CLAUSOLE
```

- Ogni riga che non inizi con `p` e `c` rappresenta invece una clausola ed è costituita da una lista di variabili, separate da uno spazio, che termina obbligatoriamente con il carattere `0`.

Le variabili sono quindi rappresentate da numeri interi positivi non nulli (lo zero è il carattere speciale di terminazione). Ogni riga rappresenta una delle disgiunzioni di letterali di cui la formula CNF in questione è composta e tutte queste righe si considerano implicitamente messe in AND tra di loro. Se una variabile compare negata, tale negazione si indica facendola precedere da un segno `-`. Un esempio di formula della logica proposizionale scritta in tale formato è il seguente:

---

<sup>24</sup>Si è usato il condizionale perché dal punto di vista pratico, difficilmente un SAT solver “rifiuterà” un input solo a causa dell’omissione di questo piccolo *header*.



```
c Esempio di formato comprensibile dai SAT solvers.  
p cnf 5 3  
2 -1 4 0  
-2 1 3 4 0  
-3 -4 5 0
```

### 3 Samael, una Libreria Logica in Java

*"Forse tu non pensavi ch'io loico fossi!"*

– Dante, *Commedia*, XXVII v. 123

Si è già ampiamente parlato del fatto che la crittoanalisi logica richieda che l'algoritmo oggetto dell'attacco venga rappresentato (in maniera semanticamente equivalente) attraverso un insieme di formule logiche, in modo che possa essere elaborato da strumenti automatici per la risoluzione di questo tipo di formule. La forza dei computer risiede nella loro capacità di eseguire operazioni ripetitive in maniera estremamente veloce ed efficiente, quindi essi rappresentano lo strumento ideale per svolgere in modo automatico la noiosa operazione di districare il groviglio di porte logiche, permutazioni, shifting e sostituzioni di cui ogni algoritmo di cifratura è composto. Per poter programmare lo svolgimento di queste operazioni in maniera sufficientemente agevole è però necessario partire da un livello di astrazione piuttosto alto, che permetta di rappresentare i classici operatori logici in modo strutturato e che consenta di effettuare una serie di manipolazioni (prevalentemente sintattiche) sulle formule.

Con lo scopo di adempiere a questi (ed altri) compiti, è stata implementata Samael, una libreria per la manipolazione di formule logiche proposizionali scritta in Java. Tale linguaggio non è sicuramente tra i più efficienti, essendo compilato in un linguaggio intermedio e poi interpretato, ma offre API moderne e soprattutto una garanzia di portabilità su qualsiasi sistema su cui è stato installata l'apposita Java Virtual Machine.

#### 3.1 Rappresentazione delle Formule

La letteratura scientifica presenta numerosissimi esempi [10, 11, 12, 13] di strutture dati utilizzate per rappresentare formule logiche in maniera efficiente. Nella maggior parte dei casi queste strutture privilegiano un particolare scopo di utilizzo e supportano un set ristretto tra tutti gli operatori logici disponibili. Per esempio, la struttura AIG (And-Inverter Graph) sfrutta la proprietà detta "universalità del NAND" e rappresenta ogni formula utilizzando esclusivamente operatori AND e NOT. L'algoritmo per portare formule generiche in CNF presentato in [10, 11] è invece basato su NICE DAGs, acronimo che sta per *Negation, Ite, Conjunction and Equivalence DAG*. Oltre

ad una distinzione basata sul set degli operatori permessi, queste strutture dati possono differire anche per tipo e comportamento (si veda in seguito).

Il modo più intuitivo per rappresentare una FBF della logica proposizionale, composta da operatori, variabili e valori di verità è sicuramente quello di utilizzare le grammatiche generative proposte da Noam Chomsky negli anni '50 e, in particolare, le grammatiche libere da contesto. Per una trattazione più approfondita, ma comunque inerente allo specifico ambito informatico, si rimanda a [14, p. 30]. Tali strumenti permettono di definire grammatiche (in questo caso quella che descrive le FBF) e di derivarne tutte e sole le espressioni ammesse, rappresentandole tramite il cosiddetto “albero sintattico” o albero di parsing. In questo modo, gli operatori booleani sono rappresentati da nodi interni dell'albero e le variabili e i valori di verità compaiono esclusivamente come foglie<sup>25</sup>. Una visita in-order a tale albero permette di estrarre la formula in notazione infissa, una visita pre-order produce una rappresentazione prefissa e una post-order visualizza l'espressione in maniera postfissa.

La scelta implementativa riguardante Smael si è rivolta principalmente verso la flessibilità: si è cercato di ottenere una rappresentazione quanto più generica delle formule tramite una arietà  $n$ -aria per ogni operatore booleano (tranne NOT e IFF, con arietà rispettivamente pari a 1 e 2). La struttura dati più adatta a garantire naturalmente questi requisiti è parsa quella dell'albero generalizzato ( $n$ -ario), cioè un grafo connesso non orientato e aciclico. Sfruttando queste caratteristiche, si è creata una gerarchia di classi che supportasse intrinsecamente la ricorsione (stile di programmazione che si presta con naturalezza ad operare su alberi) mediante la definizione dell'interfaccia `Formula`, che deve essere implementata da tutte le classi che rappresentano simboli dell'albero sintattico. Tali classi si dividono in

- Simboli non terminali
  - operatori  $n$ -ari: `Or`, `And`, `Xor`
  - operatori binari e unari: `Iff`, `Not`
  
- Simboli terminali
  - valori di verità: `Truth`

---

<sup>25</sup>Questa notazione garantisce che la precedenza degli operatori sia intrinsecamente indicata dalla costruzione stessa della struttura dati.

– variabili: `Variable`

Questo accorgimento ha prodotto una sintassi per la definizione di oggetti di tipo `Formula` (che rappresentano espressioni logiche) che per certi versi ricorda l’aspetto di un linguaggio funzionale:

```
Formula f = new Or(new And(new Variable(1), new  
Variable(2)), new Xor(new Variable(3), new Variable(4)),  
new Truth(true));
```

definisce l’espressione  $(1 \wedge 2) \vee (3 \oplus 4) \vee \top$ . Un’altra similitudine con i linguaggi funzionali è data dal fatto che tutti gli oggetti `Formula` definiti da questa libreria rispettano due invarianti:

- Nessuna funzione o metodo (tranne `compact()`) modifica lo stato dell’oggetto su cui è chiamata. Esse restituiscono un nuovo oggetto di tipo `Formula` a cui sono state apportate le modifiche previste dall’esecuzione del metodo in questione.
- La struttura ad albero è rispettata anche dal punto di vista dei puntatori ad oggetti gestiti a basso livello da Java:
  - Assenza di oggetti duplicati: ogni albero rappresentato da un oggetto `Formula` è un insieme di oggetti distinti tra loro, cioè non può verificarsi il fatto che nodi diversi puntino allo stesso oggetto.<sup>26</sup>
  - Disgiunzione delle Formule: dati due alberi sintattici  $A$  e  $B$ , a prescindere da qualsiasi relazione intercorra tra gli insiemi di nodi di tali alberi ( $A \subseteq B$ ,  $B \subseteq A$ ,  $B \neq A$ , ecc), gli oggetti appartenenti a  $Formula_A$  e a  $Formula_B$ , che rappresentano rispettivamente  $A$  e  $B$ , sono costituiti da due insiemi disgiunti.

Queste caratteristiche sono volte a eliminare lo sharing tramite puntatori: ogni oggetto che implementa l’interfaccia `Formula` sarà puntato da un (e uno solo) riferimento per tutto il suo tempo di vita. In questo senso, tutti i metodi che si applicano direttamente ad una `Formula` sono da ritenersi

---

<sup>26</sup>Per esempio, due occorrenze della stessa variabile  $x$  all’interno dello stesso albero vengono rappresentate da due oggetti distinti, entrambi con “etichetta”  $x$ .

*safe*<sup>27</sup>: come è già stato detto, l'oggetto su cui il metodo viene chiamato non viene modificato in alcun modo. Le modifiche apportate dal metodo sono presenti solo sul nuovo oggetto restituito come valore di ritorno.

```
f = new Or(new Variable(1), new Variable(4), new And(new
Variable(2), new Variable(3)), new And(new Variable(5),
new Variable(6)));
Formula g = f.distributivityOr();
```

Questo esempio mostra come la funzione che applica la distributività lasci invariata **f** e crei un nuovo oggetto **g** contenente un clone di **f** a cui è stata applicata tale proprietà.

L'unica eccezione a questa regola è data dal metodo

```
public void compact();
```

che è comunque per definizione sempre *safe* e idempotente. Tale metodo controlla, per ogni operatore  $\mathfrak{D}$  appartenente alla **Formula** **f**, che non si verifichi una situazione in cui un nodo  $o_p$  di tipo  $\mathfrak{D}$  abbia un figlio  $o_f$  dello stesso tipo. In tal caso tale figlio viene rimosso e i figli di  $o_f$  vengono aggiunti alla lista dei figli di  $o_p$ . Appare evidente che anche se in questa operazione **f** può subire modifiche dal punto di vista sintattico, dal punto di vista semantico essa rimane sempre invariata.

Per garantire la *safeness* è stato fatto largo uso del metodo

```
public Formula sham();
```

il quale, chiamato su un oggetto **Formula**, restituisce un preciso “clone” creato visitando in maniera ricorsiva l'oggetto su cui è chiamato. Tale clone è utilizzabile in qualsiasi modo con la certezza che gli invarianti vengano rispettati.

Coerentemente con quanto mostrato finora, l'interfaccia **Formula** impone alle classi che la implementano di possedere a loro volta dei metodi che applichino in maniera ricorsiva alcune funzioni basilari di manipolazione di espressioni. Alcuni dei principali sono:

---

<sup>27</sup>Un metodo è da ritenersi *safe* sse applicandolo ad un oggetto che è coerente in base agli invarianti sopra definiti, non può lasciare tale oggetto in uno stato in cui gli invarianti non sono più rispettati.

- `public boolean equals(Formula f)`; Verifica in maniera ricorsiva se la formula corrente (`this`) è sintatticamente identica a quella passata come parametro.
- `public Formula simplify()`; Restituisce una `Formula` a cui sono state applicate ricorsivamente delle semplificazioni.
- `public Formula removeXor()`; Restituisce una `Formula` a in cui tutte le occorrenze dell'operatore XOR sono state rimpiazzate secondo le regole viste in 2.2.3.
- `public String visit()`; Restituisce in maniera ricorsiva una rappresentazione testuale (in notazione infissa) dell'espressione rappresentata dall'oggetto su cui viene chiamata.
- `public And asAnd()`; Se questo metodo è chiamato su un oggetto di tipo `And`, restituisce tale oggetto, in caso contrario restituisce `null`.<sup>28</sup>

### 3.1.1 Parentele, liste e funzioni

La struttura ad albero generalizzato ( $n$ -ario) è stata ottenuta nel seguente modo:

- ogni operatore che rappresenta un simbolo non terminale può trovarsi in posizione di nodo interno o radice.
- in base all'arietà specifica di tale operatore, l'oggetto in questione conterrà una lista<sup>29</sup> di oggetti `Formula`, rappresentanti i suoi operandi, da qui in poi indicati con il termine di figli.

La lista dei figli di ogni nodo è memorizzata usando la specifica classe `NodeList`, sottoclasse di `ArrayList<Formula>`, che implementa alcune funzioni utili per elaborare queste particolari *parentele*. I principali metodi forniti da questa classe sono:

- `public boolean opposti()`; Restituisce `true` se l'oggetto `NodeList` su cui è chiamato (`this`) contiene  $A$  e  $\neg A$ . Utilizzato nelle semplificazioni.

---

<sup>28</sup>Esiste un metodo analogo per ogni oggetto che implementi `Formula`.

<sup>29</sup>L'oggetto `Not` contiene un campo `Formula` singolo, l'`Iff` ne possiede due.

- `public boolean contains(Formula d)`; Restituisce `true` se `this` contiene occorrenze di `d`.
- `public void unique()`; Modifica l'oggetto `NodeList` su cui è chiamato in modo che non contenga più di una occorrenza di formule sintatticamente identiche.
- `public void removeAll(Formula a)`; Elimina ogni occorrenza di `a` dalla `NodeList` su cui è chiamato.
- `public void uniqueXor()`; Applica un particolare tipo di eliminazione di formule ripetute specifico per la semplificazione degli oggetti `Xor`.

### 3.1.2 Alberi e Grafi

La struttura usata da Samael per rappresentare le formule è, come si è visto, quella di un albero, ed è soggetta ad alcune restrizioni. Tali limitazioni ne semplificano l'implementazione, ma sono anche causa di una particolare inefficienza nel gestire le formule che ricorrono più volte. Se all'interno della Formula `f` il sottoalbero composto dalla formula `s` appare  $m$  volte, ogni occorrenza di `s` è sintatticamente identica ma formata da oggetti differenti, causando uno "spreco" di memoria che potrebbe essere ridotto di  $m - 1$  volte, se si evitasse di replicare ogni sottoformula ripetuta. Ciò può essere fatto inserendo un riferimento alla stessa `s` (istanziata una volta sola) ogni volta che essa compare nell'albero sintattico. Questo accorgimento viene detto *sharing* ed è utile per ridurre lo spazio di memoria occupato dagli oggetti di tipo `Formula`. Il risvolto negativo riguarda invece un aumento nella complessità del codice che lo implementa: per ogni oggetto, prima di poter effettuare modifiche e/o cancellazioni, è necessario provvedere al conteggio dei riferimenti entranti. Una struttura che rappresenti espressioni utilizzando questa tecnica perde le sembianze di un albero (cioè un grafo non orientato connesso aciclico) per diventare un DAG, un *direct acyclic graph*. Questo tipo di grafo è più generale rispetto alle restrizioni di un albero e può essere sfruttato per ottenere alcune ottimizzazioni basate sull'attento sfruttamento dello *sharing*. Come si vedrà in seguito, nonostante Samael non implementi in alcun modo tale tecnica, si è utilizzato un metodo alternativo, basato su *hash tables*, che si può dimostrare essere equivalente.

### 3.1.3 Creazione del File DIMACS e risoluzione

La libreria in questione permette l'istanziamento di un oggetto di tipo `DIMACSGenerator` a cui passare un insieme di oggetti `Formula` affinché venga creato un file di output, scritto secondo il formato DIMACS CNF (descritto in 2.2.4), che ne contenga una rappresentazione comprensibile ai SAT solvers. Impostando il path locale del SAT solver desiderato, è possibile lanciare la risoluzione del file da parte di tale software tramite una call simile alla famiglia di funzioni `exec` di C.

## 3.2 Rappresentazione di DES Mediante Logica Proporzionale

Come si è visto all'inizio di questa sezione, il primo passo per applicare la crittoanalisi logica ad un qualsiasi algoritmo consiste nell'ottenere una rappresentazione equivalente sotto forma di una o più formule logiche.

The generation of the formula  $\mathcal{E}(C, K, P)$  that describes the logical characteristics of DES has been a substantial operation of *reverse "logical" engineering*. [2, p. 178]

Nel caso di DES, che ha una struttura piuttosto complessa, è utile procedere per blocchi concettuali. L'articolo [2, p. 178] descrive a grandi linee le operazioni necessarie, fornendo anche numerosi spunti. L'idea di base è di analizzare l'algoritmo di DES, generando man mano le formule che corrispondono ad ogni operazione che viene eseguita, ma con alcune eccezioni. Per ottenere formule meglio gestibili, si può infatti evitare di riportare esplicitamente la rappresentazione logica di tutte le componenti che non lo richiedano strettamente. Questo accorgimento può essere attuato applicando operazioni di questo tipo direttamente alle formule che si stanno sviluppando, invece di codificarle anch'esse tramite nuove proposizioni. In DES, le permutazioni sono un caso tipico in cui si può attuare quanto descritto: invece di "codificare" tali operazioni mediante proposizioni logiche, esse sono eseguite direttamente, permutando gli oggetti stessi che costituiscono le formule.

Il *modus operandi* descritto in [2] differisce da quello di cui si parla in questa tesi principalmente per il fatto che il *reverse engineering* compiuto da Massacci non è completamente automatizzato. Esso prevede infatti delle fasi di minimizzazione *off-line* (tramite CAD tools) delle formule che



derivano, per esempio, dalla codifica delle S-Boxes. Questa caratteristica ha fatto sì che nel loro lavoro si sia preferito ottenere una formula generica  $\mathcal{E}(P, K, C)$  fortemente ottimizzata in cui sostituire gli specifici valori assunti dalle variabili a seconda dei diversi casi di utilizzo. Nel nostro caso, invece, l'intero algoritmo è stato codificato usando la libreria descritta nel capitolo 3, in grado di rappresentare formule logiche in maniera del tutto automatica. Sfruttando questa caratteristica, ogni formula generata è specifica dell'istanza dell'attacco che si sta eseguendo. Prima di specificare i dettagli della codifica di DES mediante un software che ne genera una descrizione semanticamente equivalente utilizzando la logica proposizionale, verrà illustrato a grandi linee il principio secondo cui si svolge l'attacco.

## 4 DeepThought, un Tool per la Crittoanalisi Logica di DES

*“Forty-two!” yelled Loonquawl. “Is that all you’ve got to show for seven and half million years’ work?”*

*“I checked it very thoroughly,” said the computer, “and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you’ve never actually known what the question is.”*

– Douglas Adams, *“The Hitchhiker’s Guide to the Galaxy”*

### 4.1 Tradurre DES in Logica Proporzionale

DeepThought<sup>30</sup> è un software che è stato sviluppato per ottenere in maniera automatica la “traduzione” dei circuiti di DES in un sistema di espressioni logiche, utilizzando la libreria vista nel capitolo 3. L’attacco di cui si vuole parlare è di tipo *known-plaintext*, e verrà brevemente spiegato qui di seguito.

Prima di tutto<sup>31</sup> è necessario scegliere i valori del plaintext  $P$ , della chiave  $K$  e il numero di rounds  $r$ . I valori di plaintext e chiave possono (per comodità) essere generati in maniera pseudocasuale o possono essere espressamente specificati, mediante la notazione esadecimale. A partire da  $P$  e  $K$ , viene calcolato il corrispondente  $C = E_r(P, K)$  utilizzando la libreria stessa: dato che le sole variabili incognite appartengono a  $C$ , è possibile generare la formula corrispondente al caso specifico  $\mathcal{E}_r(v_P, v_K, \mathcal{V}_C)$  e risolverla usando il metodo `simplify()`, che applica alcune semplificazioni e che sfrutta intelligentemente i valori di verità presenti in  $\mathcal{E}$  per decidere il valore binario delle incognite  $\mathcal{V}_C$ . L’efficienza di questo procedimento non è assolutamente paragonabile al calcolo di  $C$  utilizzando una implementazione specifica di DES, ma permette di effettuare tale operazione in maniera flessibile, con il numero di rounds desiderato e di sfruttare esclusivamente le potenzialità della libreria in questione.

---

<sup>30</sup>A causa del tempo di calcolo impiegato dalle prime implementazioni, è sembrato naturale, al momento di decidere il nome, fare riferimento all’interminabile calcolo narrato nella novella *“The Hitchhiker’s Guide to the Galaxy”* di Douglas Adams.

<sup>31</sup>E’ evidente che ciò vale per il caso sperimentale, in caso di utilizzo pratico  $P$  e  $C$  saranno valori “intercettati” nel flusso di dati su cui effettuare l’attacco.

Una volta che si è ottenuto  $C$ , si metterà  $K$  da parte e inizierà la vera e propria fase di attacco *known-plaintext*. Utilizzando i componenti software che “simulano” il comportamento dei blocchi concettuali di cui DES è composto, viene generato l’insieme di proposizioni logiche  $\mathcal{E}_r(v_P, \mathcal{V}_K, v_C)$ , la cui soluzione comporta ovviamente il risalire ai valori assunti dalle 56 incognite di cui è composto, cioè le variabili rappresentanti i bits della chiave. Verificare che  $\mathcal{E}$  sia soddisfacibile è quindi equivalente a compiere un attacco known-plaintext su un blocco 64 bits cifrato con  $r$  rounds di DES.

Finding a model ( $v_K$  or another assignment) is then equivalent to break the cipher with a known plaintext attack that uses only one or few plaintext/ciphertext pair. [2, p. 175]

Tale ricerca di un modello per  $\mathcal{E}$  viene quindi affidata ai migliori SAT solvers open source disponibili al momento in cui si scrive. Volendo prescindere dall’attacco in sè, si può notare come questo procedimento sia un modo tutto sommato semplice per ottenere un gran numero di *SAT problems* di difficoltà crescente, utilizzabili per scopi sperimentali nello studio di risolutori e ragionatori [2, p. 178]. Utilizzando  $P$ ,  $K$  ed il numero di rounds come variabili di controllo, sono potenzialmente generabili almeno  $2^{56} \times 2^{64}$  *solved instances* estremamente complesse, del tutto simili ai problemi pratici posti dal mondo della ricerca per fini commerciali. Il numero di *rounds* può fungere da ulteriore “moltiplicatore” per quanto riguarda la quantità di problemi generabili e da “potenziometro” per quel che concerne la complessità che si desidera ottenere. Per generare istanze di problemi non soddisfacibili è infine sufficiente, in base alla coppia  $(P, C)$  utilizzata, costringere il software che codifica DES a elaborare la formula relativa ad un attacco known-plaintext su  $(P, C^*)$ , con  $C \neq C^*$  (è sufficiente invertire dei bits in modo casuale).

Se l’esecuzione del SAT solver va a buon fine, verrà restituito un messaggio che indica che la formula in esame è soddisfacibile, alcune statistiche riguardanti l’esecuzione e un file DIMACS contenente l’assegnamento per le variabili incognite che è stato riscontrato essere modello di  $\mathcal{E}$ . Al crescere del numero di rounds  $r$ , il numero dei modelli possibili per ogni coppia  $(P, C)$  converge a 1 piuttosto rapidamente. Alternativamente è possibile ottenere questa convergenza utilizzando un numero maggiore di coppie (*plaintext*, *ciphertext*).

#### 4.1.1 Reverse *Logic Engineering* di DES: le Permutazioni

All'interno di DES vengono effettuate permutazioni su arrays di bits in diverse occasioni. La computazione inizia infatti con l'Initial Permutation (compiuta sui bits del testo in chiaro) e termina con il suo inverso, ma anche all'interno della *cipher function* e del *key scheduling algorithm* sono presenti delle permutazioni. Come avviene anche per altri attacchi, la codifica della permutazione iniziale e finale non è strettamente necessaria alla buona riuscita della generazione di una formula  $\mathcal{E}$  equivalente.

Dato che in questo tipo di attacco non si opera più con array contenenti bits bensì con variabili booleane e valori di verità, queste permutazioni sono state implementate tramite *lookup tables* contenenti i valori indicati dalle specifiche di DES, reperibili sul sito web del NIST [24]. Prendendo come esempio la permutazione iniziale, in base ai valori contenuti nell'array IP (che la rappresenta) e dato un array  $v$  di oggetti di tipo Formula<sup>32</sup>, gli elementi di  $v$  vengono scambiati di posizione, effettuando a tutti gli effetti una permutazione.

```
private final static int[] IP = { 58, 50, 42, 34, 26, 18,
10, 2, 60, 52, 44, 36, 28, 20, 12, 4, 62, 54, 46, 38, 30,
22, 14, 6, 64, 56, 48, 40, 32, 24, 16, 8, 57, 49, 41, 33,
25, 17, 9, 1, 59, 51, 43, 35, 27, 19, 11, 3, 61, 53, 45,
37, 29, 21, 13, 5, 63, 55, 47, 39, 31, 23, 15, 7 };

public static Formula[] shuffle(Formula[] v) {
    int i;
    Formula[] permVar = new Formula[64];
    for (i = 0; i < 64; i++)
        permVar[i] = v[IP[i]-1];
    return permVar;
}
```

Un procedimento del tutto analogo viene utilizzato per la permutazione finale  $IP^{-1}$ , per la permutazione applicata all'output delle S-Boxes all'interno della *cipher function* e in entrambe le *permuted choices* utilizzate durante lo *scheduling* delle chiavi.

---

<sup>32</sup>Questo espediente incentiva la genericità del codice facendo sì che essi possano contenere variabili, valori di verità o qualsiasi tipo di formula.

### 4.1.2 Reverse *Logic Engineering* di DES: Cipher Function

La cipher function invocata all'interno della Rete di Feistel è cruciale per ogni algoritmo di questo tipo, e anche DES non fa eccezione. Tale funzione riceve come input i 32 bits (in questo caso un array contenente 32 oggetti `Formula` chiamato `RIGHT`) rappresentanti la parte destra destra del ciphertext per il round corrente ( $R_i$ ) e un array contenente i 48 oggetti `Variable` (chiamato `KEY`) che rappresentano la *subkey* da utilizzare per il round corrente,  $K_i$ .

Per prima cosa, come da copione, a  $R_i$  deve essere applicata una espansione. Questa operazione viene effettuata invocando il metodo statico `expand(RIGHT)` della classe `ExpansionBox`, che implementa l'espansione in maniera del tutto simile a quanto descritto in 4.1.1.

```
Formula [] r = ExpansionBox.expand(RIGHT);
```

Una volta ottenuto il nuovo array espanso, creato a partire da `RIGHT` e contenente 48 elementi, è possibile usare il metodo `xoring(Formula a, Formula b)` della classe `ExclusiveOr`, che implementa l'applicazione dell'operatore XOR tra due array di tipo `Formula` che abbiano lo stesso numero di elementi (descritto in 4.1.4).

```
Formula [] xor = ExclusiveOr.xoring(r, KEY);
```

L'array `xor` contiene le formule in cui la `Formula`  $i$ -esima di  $r$  è messa in XOR con la `Formula`  $i$ -esima di `KEY`. Questo insieme di formule costituisce l'input delle S-Boxes, che però sono definite in modo da ricevere 6 bits ciascuna. Per gestire questa peculiarità, gli elementi contenuti nel vettore `xor` vengono trasferiti in un array bidimensionale composto da 8 arrays di 6 elementi ciascuno, rispettandone l'ordine. In questo modo, ogni S-Box avrà a disposizione il proprio input sotto forma di un array dedicato.

### 4.1.3 Reverse *Logic Engineering*: S-Boxes

All'interno della cipher function, si è già spiegato come venga "preparato" l'input per le S-Boxes, posizionando le formule che rappresentano i rispettivi bits all'interno di un array bidimensionale

```
Formula [][] sb = new Formula[8][6];
```

che viene elaborato dalle 8 S-Boxes in un ciclo `for`

```

Formula [] sbOut = new Formula[32];
Formula [] tmpOut = new Formula[4];
for (i = 0; i < 8; i++) {
    tmpOut = SBoxes.sbox2cnf(i, sb[i]);
    for (c = 0; c < 4; c++)
        sbOut[(i * 4) + c] = tmpOut[c].removeXor().simplify();
}

```

in cui ogni iterazione applica una simulazione del funzionamento di una di esse. L'array `tmpOut` contiene ogni volta 4 formule<sup>33</sup> che, se risolte in funzione delle incognite che vi compaiono (le variabili rappresentanti i bits della chiave), restituiscono un risultato booleano (`true/false`) semanticamente equivalente al valore dei 4 bits restituiti da una singola S-Box in una implementazione ordinaria di DES. L'ultimo ciclo `for` ha come scopo quello trasferire le 4 formule ottenute in un array di 32 elementi che verrà restituito come risultato della *cipher function*. Ad ogni formula di `tmpOut` viene applicata un metodo che rimuove gli XOR utilizzando le equivalenze semantiche viste in 2.2.3 e uno che cerca di generare formule quanto più ottimizzate possibile, effettuando alcune semplificazioni.

L'implementazione del metodo `sbox2cnf` all'interno della classe `SBoxes` è sfortunatamente più intricata dei blocchi concettuali visti finora. Per semplicità, verranno omessi alcuni dettagli implementativi non direttamente necessari alla comprensione della strategia usata. La prima operazione compiuta all'interno di questo metodo è

```
int[][] tt = getTruthTable(num);
```

che restituisce una matrice di 64 righe e 10 colonne contenente la tabella di verità che descrive l'S-Box numero `num`<sup>34</sup>. Questa matrice è così suddivisa:

- le prime 6 colonne rappresentano le possibili configurazioni dei 6 bits di input
- le ultime 4 colonne rappresentano i 4 bit di output corrispondenti ai 6 di ingresso presenti sulla stessa riga

---

<sup>33</sup>Già poste in CNF.

<sup>34</sup>Con  $0 \leq \text{num} \leq 7$

- le righe sono  $2^6$  e esauriscono quindi tutte le possibili configurazioni dei bits di input

I valori contenuti in `tt` sono utilizzati per ricavare la descrizione della S-Box in questione attraverso formule logiche, secondo il metodo classico, descritto in [8, p. 25]. Appare evidente che se gli output presenti nella tabella di verità sono 4 per ogni riga, la rappresentazione logica sarà composta da 4 formule separate, una per ogni bit dell'output<sup>35</sup>. Il vettore di 4 elementi di tipo `Formula` che descrive l'S-Box richiesta viene quindi restituito all'ambiente chiamante, che è in questo caso la *cipher function*. In essa come si è visto tali output sono ricomposti fino a formare i 32 bits necessari all'elaborazione nella Rete di Feistel.

La permutazione che viene effettuata al termine della cipher function non presenta particolarità degne di nota ed è stata implementata secondo i principi descritti in 4.1.1.

#### 4.1.4 Reverse *Logic Engineering* di DES: XORing di Array di Formule

Il funzionamento del metodo

```
public static Formula[] xoring(Formula [] A,
                               Formula [] B)
```

è piuttosto semplice, ma merita una breve descrizione. L'unico vincolo richiesto è che `A` e `B` abbiano la stessa dimensione, cioè che contengano lo stesso numero di elementi. La trasformazione applicata da questo metodo è semplice: dati gli inputs, esso restituisce un nuovo array di oggetti `Formula` `c` tale che  $\forall i \ c[i] = A[i] \oplus B[i]$ . Dal punto di vista implementativo:

---

<sup>35</sup>Per ottenere direttamente le formule in CNF è sufficiente, durante l'analisi della tabella, considerare le uscite con bit a 0 e prendere il complemento dei bit a 1 del corrispondente input, secondo il classico procedimento.

```

public static Formula[] xoring(Formula [] R, Formula []
K) {
    int i;
    Formula [] result = new Formula[R.length];
    for (i = 0; i < R.length; i++)
        result[i] = new Xor(R[i], K[i]);
    return result;
}

```

#### 4.1.5 Reverse *Logic* Engineering di DES: Key Scheduling Algorithm

L'ultimo dei "blocchi concettuali" che è necessario citare, prima di considerare quello che li comprende tutti, cioè la rete di Feistel, è l'algoritmo che genera le *subkeys* utilizzate in ogni round. Tale procedimento, già descritto in 1.3.4, non è particolarmente complesso e utilizza principalmente permutazioni e *left-shifts*. In DES, la chiave è composta da 56 bits, ma ha la caratteristica di essere rappresentata in modo da comprendere anche i bit di parità (indifferenti ai fini della computazione) per una lunghezza totale di 64 bits<sup>36</sup>. Per semplicità si è mantenuta questa consuetudine anche in DeepThought: la chiave è rappresentata da un array di 64 elementi di tipo `Variable`, che costituiscono le incognite della computazione. Le 8 variabili che si trovano in corrispondenza dei bits di parità sono totalmente ininfluenti ai fini del risultato, dal momento che la prima permutazione che viene loro applicata ha appunto lo scopo di eliminarle.

Dato il vettore `K` di 64 elementi, contenente oggetti di tipo `Variable`, la prima operazione che viene effettuata dalla classe `KeySchedule` prevede l'applicazione della *first permuted choice*

```
originalKey = PermutedChoice.shuffle(K, 0);
```

che restituisce un vettore di 56 elementi contenenti le sole variabili effettivamente utilizzate.

Tale vettore viene quindi diviso in due metà, nel più classico dei modi:

---

<sup>36</sup>Un bit di parità ogni 7 bits della chiave, per un totale di 8.



```

left = new Formula[28];
right = new Formula[28];
int i;
for (i = 0; i < 28; i++)
    left[i] = originalKey[i];
for ( ; i < 56; i++)
    right[i - 28] = originalKey[i];

```

Agli array `left` e `right` vengono applicati gli shifting necessari per elaborare la chiave per il round richiesto, secondo i valori memorizzati in una apposita *lookup-table*. Il *left shift* è stato codificato come metodo statico della apposita classe `LeftShift`. Quando è necessario effettuare uno spostamento di due posizioni, viene richiamata in maniera ricorsiva due volte la funzione che applica lo spostamento singolo.

```

private static int [] shiftXRound = { 1, 1, 2, 2, 2, 2,
2, 2, 1, 2, 2, 2, 2, 2, 2, 1};
public Formula[] nextKey() {
    currentRound++;
    left = LeftShift.shift(left, shiftXRound[currentRound]);
    right = LeftShift.shift(right, shiftXRound[currentRound]);
    Formula[] res = new Formula[56];
    int i;
    for (i = 0; i < 28; i++)
        res[i] = left[i].sham();
    for ( ; i < 56; i++)
        res[i] = right[i - 28].sham();
    return PermutedChoice.shuffle(res, 1);
}

```

Una volta inizializzato l'oggetto `KeySchedule`, ogni sottochiave viene generata in modo incrementale a partire dalla precedente: il metodo `nextKey()` conserva i vettori `left` e `right` più recenti, applica le operazioni necessarie a creare la *subkey* successiva e la restituisce come valore di ritorno.

#### 4.1.6 Reverse *Logic Engineering* di DES: la Rete di Feistel

Tutti i blocchi concettuali visti in precedenza (e non solo) sono utilizzati all'interno della Rete di Feistel, che è la struttura base dell'algoritmo e di cui si è già parlato in 1.3.1. La prima operazione da effettuare è quella di dividere il testo in chiaro in due metà, chiamate  $L$  e  $R$ . Nel caso in questione il *plaintext* è rappresentato da un array di 64 oggetti di tipo `Truth`<sup>37</sup>, che viene appunto diviso in due metà omonime. L'implementazione più semplice a cui si possa pensare è

```
for (i = 0; i < numRound; i++) {
    int j;
    Formula[] swp = new Formula[32];
    for (j = 0; j < 32; j++)
        swp[j] = L[j].simplify();
    for (j = 0; j < 32; j++)
    {
        R[j] = R[j].simplify();
        L[j] = R[j].sham();
    }
    R = ExclusiveOr.xoring(swp, CipherFunction.encipher(R,
ks.nextKey()));
}
```

che ha tuttavia il difetto di subire una crescita esponenziale della dimensione delle formule, causata da ogni esecuzione della *cipher function*. Durante il primo round, infatti,  $R_1$  è costituito da semplici valori di verità, ma, già a partire dal secondo, ciascuna delle 32 formule che compongono  $R_2$  sarà costituita dal valore di verità derivante da  $L_1$  messo in XOR con i risultati della precedente *cipher function*, che, a causa della presenza delle S-Boxes, saranno composti da formule piuttosto ingombranti. In queste condizioni l'output della cipher function, anche solo al termine del secondo round, subisce una esplosione della sua dimensione.

Per ovviare a questo inconveniente si è ricorso all'utilizzo di variabili ausiliarie in ogni round a partire dal secondo. L'algoritmo "originale" viene

---

<sup>37</sup>Si ricorda che l'attacco effettuato è known-plaintext, e quindi per definizione ogni bit è rappresentato da un valore di verità conosciuto.

modificato nel seguente modo:

- se il round in questione è il primo, vengono applicate le trasformazioni convenzionali:

$$L_1 = R_0$$

$$R_1 = L_0 \oplus f(R_0, K_1)$$

- per ciascuno dei seguenti rounds, vengono aggiunte 32 nuove variabili ausiliarie (indicate con *shaR*) in vece delle espressioni contenute in *R*, con lo scopo di mantenere costante la dimensione delle formule restituite dalla *cipher function* *f*. La computazione, dotata di questo accorgimento, concettualmente diventa:

$$L_{i+1} = shaR_i$$

$$R_{i+1} = L_i \oplus f(shaR_i, K_i)$$

In questo modo *f* restituisce in ogni round un vettore di formule di dimensione costante, indipendentemente da quante iterazioni sono state eseguite. Perchè la semantica della rete di Feistel sia conservata, per ogni vettore aggiunto di variabili “ombra”<sup>38</sup> di *R*, viene aggiunta la *constraint*

$$shaR_i \iff R_i$$

che rende effettiva la sostituzione. Il procedimento in questione produce quindi

- 64 formule (oggetti di tipo Formula), ciascuna di esse rappresentante (in funzione delle variabili incognite di *K*) il valore del corrispettivo bit del *ciphertext*
- $32 \cdot (r - 1)$  formule ausiliarie (dove *r* è il numero di rounds utilizzati), ciascuna contenente una *constraint* che lega il suo valore alla corrispondente formula di *R* che da essa è stata sostituita nel calcolo

per un totale di  $64 + 32 \cdot (r - 1)$  proposizioni logiche, che possono essere congiunte in un unico AND.

---

<sup>38</sup> *shaR* deriva da “shadows”, per indicare che si tratta delle variabili “ombra” di *R*.

Una volta generate le formule, il passo successivo consiste nell'applicare alcune ottimizzazioni e convertirle in formato DIMACS, in modo che un SAT solver possa portare a termine l'attacco, verificandone la soddisfacibilità.

## 4.2 Elaborazione e Semplificazione delle Formule

Le formule così ottenute hanno una dimensione al limite del trattabile, per cui è necessaria una certa accortezza nella loro manipolazione. Le semplificazioni convenzionali non sortiscono particolare effetto: un algoritmo di cifratura come DES è appositamente studiato per produrre un output privo di ridondanze o strutture riconoscibili e tale caratteristica si trasferisce anche alla sua rappresentazione logica. Sono comunque applicabili le tipiche equivalenze sintattiche descritte in [8, p. 21], quali idempotenza, assorbimento e doppia negazione. Una ulteriore semplificazione può essere effettuata sfruttando la propagazione dei valori di verità derivanti da *plaintext* e *ciphertext*. La funzione `simplify()` applica le seguenti risoluzioni, in maniera ricorsiva:

- se un oggetto `Or oi` contiene un figlio `f` di tipo `Truth` che valga `true`<sup>39</sup>, `oi` viene rimosso e viene rimpiazzato da un valore `true`. Se `oi` era l'operatore radice, la formula intera risulta vera.<sup>40</sup>
- se un `Or oi` contiene uno o più figli `false`, tutti questi figli vengono rimossi. Se `oi` contiene solo figli `false`, `oi` viene rimosso ed è sostituito da un `false`.
- se un `And ai` contiene uno o più oggetti `true`, tali oggetti vengono rimossi. Se `ai` rimane senza figli a causa di questa operazione, significa che conteneva solo `true` e quindi viene rimosso per essere sostituito da un `true`.
- se un `And ai` contiene un valore di verità `false`, `ai` viene rimosso e al suo posto viene inserito un oggetto `false`.
- se un `Or oi` contiene contemporaneamente una `Variable a` e `Not(a)`, `oi` viene rimosso e sostituito con un `true`.
- se un `And ai` contiene una `Variable p` e `Not(p)`, `ai` viene rimosso e sostituito con un `false`.

---

<sup>39</sup>Per semplicità, da qui in avanti questa notazione sarà considerata pleonastica ed evitata.

<sup>40</sup>Questo assunto è considerato ovvio da qui in poi.

Per dare una idea del procedimento usato, qui di seguito viene mostrato il codice che implementa la funzione `simplify()` per quanto riguarda gli oggetti di tipo `And`.

```
public Formula simplify() {
    this.compact();
    int i;
    And a = new And();
    for (i = 0; i < figli.size(); i++)
        a.getListaFigli().add(figli.get(i).simplify());
    if (a.getListaFigli().opposti() ||
        a.getListaFigli().contains(new Truth(false)))
        return new Truth(false);
    if (a.getListaFigli().contains(new Truth(true)))
        a.getListaFigli().removeAll(new Truth(true));
    a.getListaFigli().unique();
    if (a.getNumFigli() > 1)
        return a;
    else if (a.getNumFigli() == 1)
        return a.getListaFigli().get(0);
}
```

#### 4.2.1 Semplificazioni Avanzate

Come si è visto, le semplificazioni convenzionali riguardanti l'algebra booleana non sortiscono un effetto realmente degno di nota, nel caso di DES. Fortunatamente la letteratura scientifica contiene moltissimi spunti [18, 16, 19, 17, 9] per quanto concerne l'ottimizzazione di espressioni logiche. Sfortunatamente, invece, molti di questi procedimenti non sono sempre direttamente applicabili oppure il loro costo computazionale è molto elevato e non sempre il risultato ottenuto è tale da giustificare l'utilizzo. Alcune di queste ottimizzazioni, selezionate in base al rapporto tra semplicità ed efficacia, sono presentate qui di seguito.

**Regola di Eliminazione per Letterali Singoli** Questo metodo è presentato in [9, p. 211] ed è applicabile su formule in CNF. In Samael è implementato nella classe `SetBasedClauseOptimizations`, in cui è chiamato `singleClauseElimination()`, e il suo utilizzo è in grado di eliminare qualche centinaio di clauses e di letterali, a seconda dei parametri scelti. Le regole illustrate da Davis e Putnam sono in questo caso tre, ma la prima è stata omessa perché già trattata all’inizio di questa sottosezione. Le rimanenti due indicano che

- Se  $F$  è una formula in CNF e  $p$  è un letterale singolo (cioè  $\exists$  una clausola composta solo da  $p$ ), rimuovere da  $F$  tutte le clausole che contengono occorrenze positive di  $p$  ed eliminare ogni occorrenza di  $\neg p$  dalle altre clausole non ne pregiudica la soddisfacibilità.
- Se  $F$  è una formula in CNF e  $\neg p$  è un letterale singolo, rimuovere da  $F$  tutte le clausole che contengono  $\neg p$  ed eliminare ogni occorrenza di  $p$  all’interno delle rimanenti clausole, non pregiudica la soddisfacibilità di  $F$ .

**Affirmative-Negative Rule** Anche questo procedimento è presentato in [9, p. 211], ed è sostanzialmente una variante del precedente. Data una formula  $F$  in CNF, se il letterale  $p$  vi compare solo affermativamente o negativamente, tutte le clausole che contengono  $p$  possono essere rimosse, ottenendo la nuova formula  $\mathfrak{F}$ . Si può dimostrare che  $\mathfrak{F}$  è consistente sse anche  $F$  lo è. (Se  $\mathfrak{F}$  è vuota,  $F$  è consistente).

**Subsumption** Considerando la formula  $F$  in CNF come un insieme di clausole e considerando ogni clausola come un insieme di letterali, si dice che la clausola  $C_1$  include  $C_2$  sse  $C_2 \subseteq C_1$ . Se questa condizione è verificata,  $C_2$  è ridondante e può quindi essere rimossa da  $F$ . L’individuazione di *subsumed clauses* è stata implementata in Samael, ed è effettuata dal metodo `public void clauseSubsumption()`, riportato qui di seguito:

```

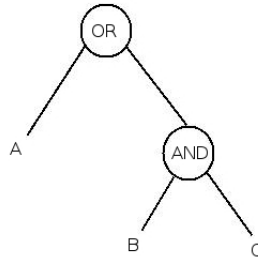
public void clauseSubsumption() {
    int i;
    for (i = 0; i < formula.getNumFigli(); i++) {
        if (formula.getFiglio(i).asOr() == null)
            continue;
        NodeList n = formula.getFiglio(i).getListaFigli();
        boolean ciclo = true;
        for (int j = 0; j < formula.getNumFigli()
            && ciclo; j++) {
            if (j == i || formula.getFiglio(j).asOr() == null)
                continue;
            else if (n.size() <= formula.getFiglio(j).getNumFigli()
                && n.isSubsumedby(formula.getFiglio(j).getListaFigli()))
            {
                formula.getListaFigli().remove(i);
                i--;
                ciclo = false;
                subsumption++;
            }
        }
    }
}

```

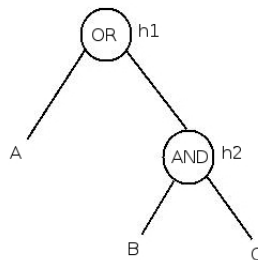
#### 4.2.2 Il metodo di Tseitin

In 2.2.3 si è accennato al problema della trascrizione di una formula  $F$  generica in una sua rappresentazione  $F^C$  in Conjunctive Normal Form. Si è anche visto come l'algoritmo classico, usato per esemplificare il concetto di distributività, sia estremamente inefficiente sia dal punto di vista temporale che da quello spaziale. Fortunatamente il problema della generazione di "buone" forme CNF ha avuto un ampio rilievo nella letteratura scientifica, e gli spunti sono numerosi. Il risultato più importante [15] risale agli anni '70 e consiste in un algoritmo (caratterizzato da una sorprendente eleganza e semplicità) che data una  $F$  generica, deriva in tempo e spazio lineare la rispettiva  $F^C$ , introducendo delle variabili ausiliarie e delle *constraints* su di esse.

Data l'espressione  $F = A \vee (B \wedge C)$ , essa è rappresentabile tramite il suo albero di parsing, che avrà la seguente forma:



L'algoritmo prevede prima di tutto che ogni nodo interno della formula venga etichettato con una variabile ausiliaria  $h_i$ , ottenendo un secondo albero che rappresenta  $F$ :



A questo punto, per ogni nodo etichettato con  $h_i$ , partendo da quello con l'indice  $i$  più basso (quindi con tutta probabilità il nodo radice<sup>41</sup>) è necessario aggiungere una constraint che definisca la nuova variabile  $h_i$ , utilizzando l'operatore Iff nella forma

$$(h_1 \leftrightarrow (A \vee h_2)) \wedge (h_2 \leftrightarrow (B \wedge C))$$

Per rendere completa questa trasformazione, l'ultimo passo da effettuare è quello riguardante l'*enforcement* del nodo radice attraverso la congiunzione

---

<sup>41</sup>Se il nodo radice è un NOT, si potrebbe evitare di "dedicargli" una specifica variabile ausiliaria, semplicemente negando la variabile  $h_i$  usata per rappresentare il suo unico figlio



della formula ottenuta con la variabile ausiliaria che lo rappresenta

$$TST = (h_1 \leftrightarrow (A \vee h_2)) \wedge (h_2 \leftrightarrow (B \wedge C)) \wedge h_1$$

A questo punto la nuova formula  $TST$  ottenuta è soddisfacibile sse anche  $F$  lo è.  $TST$  non contiene ancora, ovviamente, una formula in CNF, ma la sua nuova trascrizione permette di sfruttare le equivalenze semantiche viste in 2.2.3 in maniera tale da “favorire” l’ottenimento di tale forma normale.

Dato che la CNF ammette solo operatori di tipo AND, OR e NOT, il primo passo da effettuare è sostituire ogni occorrenza di operatori diversi da quelli elencati con una formula ad essi equivalente. Si nota facilmente che l’operatore Iff, pesantemente utilizzato dal metodo di Tseitin, ha la particolare caratteristica di avere una rappresentazione equivalente che è appunto in forma CNF

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

Utilizzando tale regola, è possibile derivare da  $TST$  la seguente formula

$$((\neg h_1 \vee (A \vee h_2)) \wedge (\neg(A \vee h_2) \vee h_1)) \wedge h_1 \wedge ((\neg h_2 \vee (B \wedge C)) \wedge (\neg(B \wedge C) \vee h_2))$$

A partire dall’espressione così ottenuta è necessario applicare prima di tutto la funzione `compact()` e `deMorgan()`, per eliminare parentesi inutili e per fare in modo che l’operatore NOT sia sempre riferito ad un letterale. La formula risultante di questo penultimo passaggio è

$$(\neg h_1 \vee A \vee h_2) \wedge ((\neg A \wedge \neg h_2) \vee h_1) \wedge h_1 \wedge (\neg h_2 \vee (B \wedge C)) \wedge (\neg B \vee \neg C \vee h_2)$$

a cui basta applicare, in maniera locale<sup>42</sup>, l’implementazione più “banale” della proprietà distributiva, per ottenere la forma CNF finale

$$(\neg h_1 \vee A \vee h_2) \wedge (\neg A \vee h_1) \wedge (\neg h_2 \vee h_1) \wedge h_1 \wedge (B \vee \neg h_2) \wedge (C \vee \neg h_2) \wedge (\neg B \vee \neg C \vee h_2)$$

La libreria `Samael` fornisce due diverse implementazioni di questo procedimento. La prima di esse è stata sviluppata in maniera ricorsiva ed è definita nei metodi previsti dall’interfaccia `Formula`. Per dare una idea riguardo al suo funzionamento, di seguito è riportato il codice relativo all’oggetto `And`,

---

<sup>42</sup>Con “applicazione locale” si intende dire che non è necessario elaborare l’intera formula, bensì è decisamente più conveniente agire sulle singole sottoformule.

ma da qui in avanti verrà principalmente trattata la seconda (e più sofisticata) implementazione.

```
public Formula tseitin(int hn, boolean root) {
    if (root)
        this.compact();
    Variable h = new Variable(hn);
    And o = new And();
    And a = new And();
    int i;
    for (i = 0; i < figli.size(); i++) {
        if (figli.get(i).asVar() != null ||
figli.get(i).asTruth() != null ||
            (figli.get(i).asNot() != null &&
            figli.get(i).getFiglio(0).getNumFigli() == 0) )
            o.getListaFigli().add(figli.safeGet(i));
        else {
            a.getListaFigli().add(figli.get(i).tseitin(++hn,
false));
            o.getListaFigli().add(new Variable(hn));
            hn += figli.get(i).getNumNodi()-1;
        }
    }
    Iff t = new Iff(h, o);
    a.getListaFigli().add(
        t.removeIff().deMorgan().simplify().distributivityOr());
    if (root) {
        a.getListaFigli().add(h);
        a.compact();
        return a.simplify();
    }
    return a;
}
```

### 4.2.3 Tseitin Iterativo con Clause Learning tramite HashTables

La prima implementazione del metodo di Tseitin, come era ovvio aspettarsi, ha segnato inizialmente un nettissimo miglioramento rispetto alle prestazioni fornite dall'implementazione più "classica" della proprietà distributiva. La seconda implementazione è invece stata frutto di una attenta ricerca nella letteratura scientifica di possibili nuove ottimizzazioni nella rappresentazione delle formule. Come si è già visto, i metodi algebrici di semplificazione più comuni non sortiscono un grande effetto sulle formule prodotte da DES, per cui si è optato per un metodo più radicale, traendo spunto dai lavori [11, 10, 13, 12]. L'idea di fondo comune a questi ricercatori riguarda l'utilizzo di grafi diretti aciclici per la rappresentazione delle espressioni, in modo da poter sfruttare a proprio vantaggio lo *sharing* dei sottoalberi ripetuti o ridondanti, evitandone la replicazione in memoria. Si è già discusso del fatto che la struttura utilizzata da Samael sia invece un albero vero e proprio, in cui tale accorgimento non è applicabile nativamente. Per ovviare a questo inconveniente è stata sviluppata una versione del metodo di Tseitin che agisce esternamente all'albero di parsing e che utilizza una *hash table* per tenere traccia dei sottoalberi  $s_i$  a cui è già stata assegnata una variabile ausiliaria  $h_i$ , in modo che non vengano utilizzate altre variabili  $h_j$  (con  $j \neq i$ ) nel caso si incontrassero nuovamente sottoalberi  $s_j$  sintatticamente equivalenti a  $s_i$ . Questo procedimento è in grado di produrre formule CNF ottimizzate almeno quanto quelle frutto dell'applicazione del metodo di Tseitin su formule rappresentate attraverso strutture che implementano lo *sharing*<sup>43</sup>.

L'implementazione di questo metodo è stata effettuata nella classe `HashedTseitin`, che contiene principalmente i due metodi

- `public Formula applyTseitin()`; applica una versione iterativa del metodo in questione, tenendo traccia esternamente di parentele e variabili ausiliarie utilizzate
- `public Formula applySmartTseitin()`; applica la versione iterativa del metodo di Tseitin in cui è stato introdotto un elementare meccanismo di *learning* (o meglio, di memorizzazione) dei nodi a cui è già stata assegnata una variabile ausiliaria, in modo che per ogni loro occorrenza sia impiegata sempre tale variabile.

---

<sup>43</sup>Si accennerà in seguito ad un caso di utilizzo in cui tale metodo risolve un problema che dei DAGs non potrebbero affrontare, se non con un accorgimento simile.

La *hash table* utilizzata è definita nella classe `FormulaHashTable` ed è sostanzialmente una tabella che risolve le collisioni attraverso il *chaining*, implementata come lista bidimensionale di oggetti `HashNode`.

```
private ArrayList<ArrayList<HashNode>> ht;
```

La classe `HashNode` istanzia semplicemente degli oggetti attraverso cui le formule sono indicizzate e memorizzate nella *hash table*: essa prevede infatti due campi, uno di tipo `int` che contiene il numero della variabile ausiliaria associata al sottoalbero memorizzato nell'altro campo, di tipo `Formula`.

La dimensione iniziale della tabella è calcolata in base al numero di nodi della formula da elaborare, ma non è necessariamente statica: il *load factor*  $\alpha$  è calcolato come  $\alpha = \frac{n}{m}$ , dove  $n$  è il numero di oggetti memorizzati nella *hash table* e  $m$  è il numero di celle totali. Solitamente, quando  $\alpha$  eccede un certo valore limite indicato da `threshold`<sup>44</sup>, è possibile invocare la funzione `rehash()` che genera una nuova tabella di dimensione  $2 \cdot m$ . La funzione di *hashing* tipica per questa tabella,

```
public int getHashCode();
```

è stata aggiunta ai metodi prescritti dall'interfaccia `Formula`, facendo in modo che possa essere calcolata ricorsivamente in base al valore di ogni elemento del *parsing tree*. La classe `FormulaHashTable` la utilizza previa mediazione del metodo

```
private int getHash(Formula f)
```

che evita alcuni errori banali.

Il procedimento è di per sé molto intuitivo: la funzione `applySmartTseitin()` utilizza una lista dinamica per tenere traccia dei nodi interni dell'albero a cui deve essere assegnata una variabile ausiliaria univoca. Questa lista è utilizzata per effettuare una visita (simile ad una BFS) dell'albero, accodando di volta in volta i nodi a cui potrebbe essere necessario associare una variabile  $h_i$ . Se una `Formula p` in questa lista ha le caratteristiche sufficienti per poter essere sostituita da una nuova variabile, cioè se non è un `Not` o una `Variable`, prima di tutto si verifica che nella *hash table* non sia già presente una definizione per `p`, attraverso l'ovvio metodo

```
public int get(Formula p)
```

---

<sup>44</sup>Solitamente tale valore è impostato a 0.75, ma è modificabile a piacere.

Tale metodo restituisce  $-1$  sse non è presente alcuna occorrenza di  $p$ . In caso contrario il valore di ritorno è un numero intero non nullo che indica il valore della variabile ausiliaria precedentemente usata per definire  $p$ . Ogni volta che si incontra un  $p$  per il quale il metodo `get` restituisce  $-1$ ,  $p$  viene inserito nella *hash table* attraverso il metodo

```
public void put(Formula f, int n)
```

che individua il *bucket* `ht[getHash(f)]` su cui effettuare l'*append* del nuovo elemento.

In questo modo, se la `Formula` in questione contiene sottoalberi ripetuti, ciascuno di essi sarà comunque identificato dalla stessa variabile. Tale accorgimento permette la creazione di una CNF in cui è stato introdotto il minimo numero di  $h_i$  necessarie, senza appesantirla inutilmente. La funzione di cui si è parlato è troppo lunga per poter essere riportata in questo documento.

L'ultimo punto trattato riguardo a questa implementazione è relativo ad un ulteriore accorgimento che permette di sfruttare al meglio la procedura mediante la quale si tiene traccia delle associazioni tra variabili ausiliarie e sottoalberi stabilite precedentemente.

Come si è visto, l'output di `DeepThought` è costituito da un insieme  $\mathfrak{F}_T$  di  $64 + 32 \cdot (r - 1)$  proposizioni logiche, di dimensione al limite del trattabile (se si utilizzano gli strumenti di cui si è parlato finora). Questo fatto implica che, per quanto all'interno del file DIMACS tutte le proposizioni appartenenti a  $\mathfrak{F}_T$  siano congiunte dagli AND impliciti previsti dalla notazione<sup>45</sup>, per motivi di efficienza, le semplificazioni post-generazione effettuate da `DeepThought` sono effettuate considerando una formula alla volta. L'effetto collaterale di questa scelta (forzata) implementativa è che essa restringe l'efficacia di numerose tecniche di ottimizzazione delle formule, come per esempio la *detection* di *subsumed clauses*. Dato che l'applicazione della versione *smart* dei metodi di Tseitin costituisce l'arma più efficace<sup>46</sup> che si è riuscita ad ottenere nel contesto descritto, si è, almeno per questo procedimento, ovviato al problema della frammentazione delle formule.

Ricapitolando, il metodo appena trattato viene applicato su una `Formula`  $f$  ed utilizza una *hash table* per memorizzare le coppie  $(s_i, h_i)$  formate da un

<sup>45</sup>Il file DIMACS può raggiungere la dimensione di svariati MegaBytes

<sup>46</sup>Tale procedimento è applicato con successo mediamente 10.000 volte, per un numero di rounds superiore a due. Il meccanismo di "learning" raddoppia questo valore.

sottoalbero e dalla relativa variabile ausiliaria che lo rappresenta. In questo modo, per ogni nuovo sottoalbero  $s_j$  a cui dovrebbe essere associata una nuova variabile ausiliaria  $h_j$ , è possibile controllare se esso non sia già stato incontrato in passato, in modo da rappresentarlo tramite la stessa variabile a cui era già stato associato allora, invece di definirne inutilmente una nuova. In questo scenario appare chiaro come sia presente una tabella di *hash* per ogni Formula elaborata. Il passo successivo è consistito nel far sì che ogni tabella generata ed utilizzata per elaborare una singola formula venisse conservata in un array di oggetti `FormulaHashTable`, in modo da costituire una sorta di memoria comune a cui tutte le future esecuzioni di `applySmartTseitin()` possano accedere, ottenendo quindi un effetto equivalente ad una applicazione di questo procedimento all'intero insieme  $\mathfrak{F}_T$ , considerandolo come una unica formula. La classe che implementa questo metodo è una variante di quella sopra descritta e si chiama `LearningHashedTseitin`. A parte i dettagli implementativi che permettono l'accesso sequenziale all'array che funge da "memoria storica", la funzione `applySmartTseitin()` fornita da tale classe è analoga a quella descritta precedentemente. L'unica differenza consiste nel fatto che se la prima verifica che il sottoalbero  $p$  non sia già stato precedentemente definito (attraverso il metodo `get(Formula p)`) restituisce un risultato negativo, non si assegna "così facilmente" la nuova variabile, bensì si procede con un controllo analogo sulle *hash tables* generate dalle applicazioni dello stesso metodo sulle formule precedenti, nella speranza che tale sottoalbero sia apparso in almeno una di esse. E' evidente che in questo modo, per ogni sottoalbero che non si è mai presentato in precedenza, si effettua un numero di ricerche senza successo (e quindi dal costo massimo) che cresce al crescere del numero di formule precedentemente trattate. Fortunatamente, l'utilizzo di memoria da parte di queste tabelle non è eccessivo, dato che le formule ivi contenute non sono replicate ma sono solo composte da puntatori all'istanza della formula contenuta nell'albero.

Per concludere, la ricerca non mostra sperimentalmente il degrado delle prestazioni che ci si potrebbe attendere, man mano che si procede con l'elaborazione di formule di indice più alto. Il tempo totale impiegato esclusivamente nella ricerca all'interno delle tabelle non supera solitamente i 20 secondi.

Come ultimo appunto, si possono aggiungere due evidenze. La prima, in difesa della scelta di implementare gli alberi sintattici utilizzati da Samael

sotto forma di albero vero e proprio e non di DAG, consiste nell’osservazione che, per quanto la non-gestione dello *sharing* causi un effettivo “spreco” di memoria (causato dai sottoalberi ripetuti e replicati ogni volta), nemmeno un utilizzo dei DAG avrebbe potuto ovviare al problema derivante dalla ridotta efficacia delle tecniche di ottimizzazione dovuta alla loro applicazione su formule singole invece che su  $\mathfrak{F}_T$ .

La seconda questione, invece, riguarda il procedimento mediante il quale vengono identificati i sottoalberi ripetuti all’interno delle versioni più sofisticate del metodo di Tseitin. La funzione che stabilisce se due oggetti `Formula a` e `b` sono identici, si basa praticamente in maniera esclusiva su una equivalenza di tipo sintattico, molto meno efficace di quella semantica. Applicare una vera equivalenza semantica avrebbe un costo computazionale insostenibile, per cui nel codice si è cercato di limitare questa mancanza con costrutti del seguente tipo:

```
if (a.simplify().equals(b.simplify())) {...}
```

che risolvono il problema per alcuni casi più semplici, come per esempio quello in cui  $\mathbf{a} = \neg\neg B \wedge C$  e  $\mathbf{b} = B \wedge C$ , che, per quanto sia banale, dal punto di vista sintattico costituisce già un problema.

### 4.3 Dati Sperimentali

Per le verifiche sperimentali del lavoro svolto, si sono utilizzati tre diversi SAT solver, due dei quali sono stati scelti in base ai risultati della competizione “SAT-Race 2010”<sup>47</sup>, tenutasi in luglio ad Edinburgo. Il terzo risolutore utilizzato è stato MiniSat (versione del 2005). Brevemente, il vincitore della prova principale del SAT-Race è stato CryptoMiniSat, sviluppato da Mate Soos e descritto in [20], mentre il secondo classificato è stato Lingelin, di cui si parla in [21]. Documentazione riguardante MiniSAT è invece reperibile in [22, 23].

Utilizzando questi strumenti, si sono effettuate delle prove per poter raccogliere alcune statistiche: per ogni round sono stati effettuati *known-plaintext attacks* che utilizzassero da un minimo di 1 ad un certo massimo di coppie  $(P, C)$ . Le statistiche sono state raccolte utilizzando i dati forniti in output dai SAT solver stessi e si dividono in tre categorie: utilizzo di memoria, tempo di risoluzione e un coefficiente  $L$  che rappresenta entrambi questi

<sup>47</sup>Il sito web si trova all’URL <http://baldur.iti.uka.de/sat-race-2010/>.

parametri, calcolato in maniera da privilegiare la comprensione del tipo di andamento asintotico che il SAT solver assume:

$$L = \log_{10}(\text{tempo} \cdot \text{memoria})$$

I test illustrati, se non è specificato diversamente, sono stati effettuati su una macchina AMD Athlon X2 Dual Core QL-60 con 4 Gb di memoria RAM, su cui è installato un *kernel* Linux 2.6.32-25-generic. Ogni SAT solver è stato eseguito in modalità *monothread*.

Questi *benchmarks* sono divisi, oltre che per numero di round, anche in base al metodo usato per generare le formule finali. Nelle didascalie, la notazione “HT” dopo il nome del SAT solver indica che tale risolutore è stato applicato su formule generate utilizzando le ottimizzazioni fornite dalla versione *smart* dell’algoritmo di Tseitin, che individua lo *sharing* dei sottoalberi tramite le *hash tables*. Se non è specificato niente, si intende che i SAT solvers hanno operato su formule elaborate dalla versione ricorsiva dell’algoritmo di Tseitin, senza l’eliminazione delle ridondanze. Per questioni di chiarezza, in seguito, si farà riferimento a “formule HT” o “non HT” per indicare il metodo usato per generarle.

#### 4.3.1 Risultati Sperimentali Crittoanalisi: 1 round

I DIMACS generati nel caso del singolo round non sono molto interessanti, dal punto di vista crittoanalitico o logico. L’*avalanche effect* è praticamente inesistente e tutte le formule che codificano bits appartenenti a  $L_1$  sono costituiti dagli stessi valori di verità della parte destra di  $P$  forniti in ingresso<sup>48</sup>.

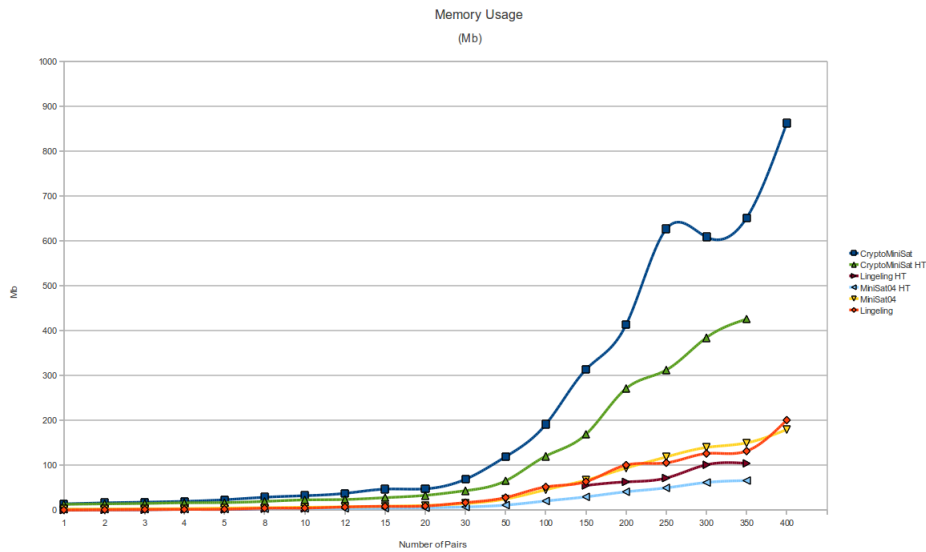
I file DIMACS hanno dimensioni contenute, vengono generati estremamente velocemente (la rete di Feistel non richiede l’utilizzo di variabili ausiliarie), ma non contengono un sufficiente quantitativo di constraints per ridurre a 1 il numero dei modelli.

L’andamento dell’utilizzo di memoria richiesto da ogni singolo SAT solver per effettuare il *known-plaintext attack* su una versione di DES limitata ad un solo round è indicato dalla seguente tabella:

---

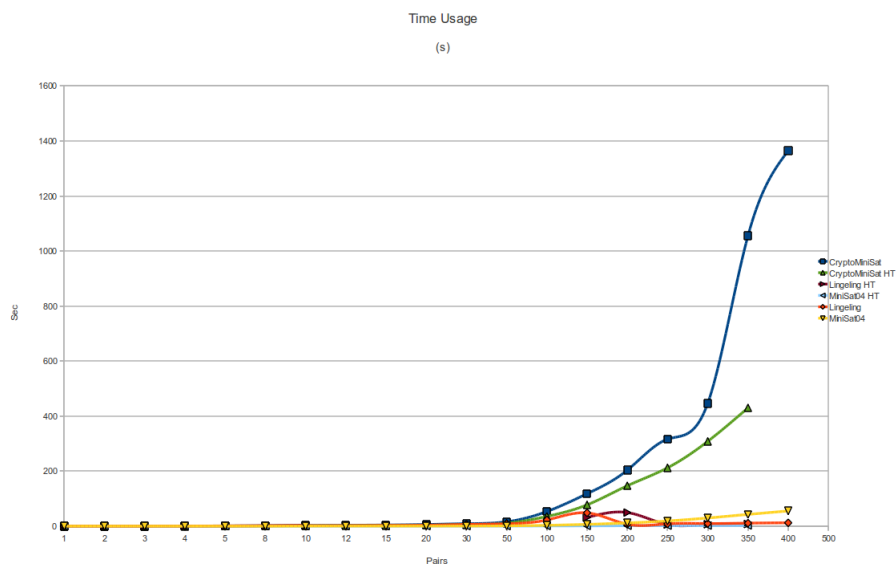
<sup>48</sup>Si ricorda che  $R_0 = L_1$ .





Si nota che CryptoMiniSat consuma il maggior quantitativo di memoria, mentre MiniSAT è il tool più “parsimonioso”. In generale, come era auspicabile, la risoluzione di formule generate tramite il metodo di Tseitin con *hash tables* richiede un minore utilizzo di memoria.

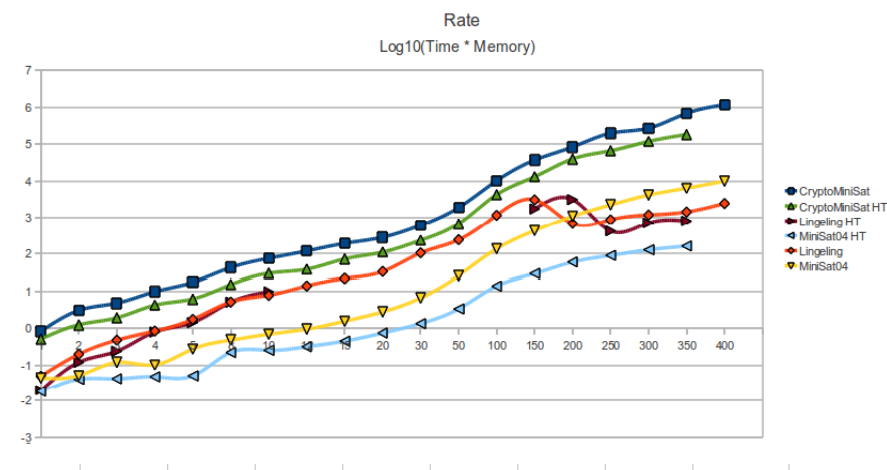
Il tempo impiegato dai SAT solver per riuscire a trovare un modello per i DIMACS è rappresentato dal seguente grafico:



A partire da 200 coppie  $(P, C)$ , CryptoMiniSat subisce una “esplosione” del tempo richiesto per la risoluzione, ma termina comunque il test. MiniSat

lavora egregiamente su formule ingombranti ma semplici come quelle in questione, ed ha un andamento lineare con un coefficiente angolare molto basso. Lingeling si dimostra in generale molto versatile, situandosi tra gli estremi.

Infine, il grafico seguente mostra l'andamento logaritmico:

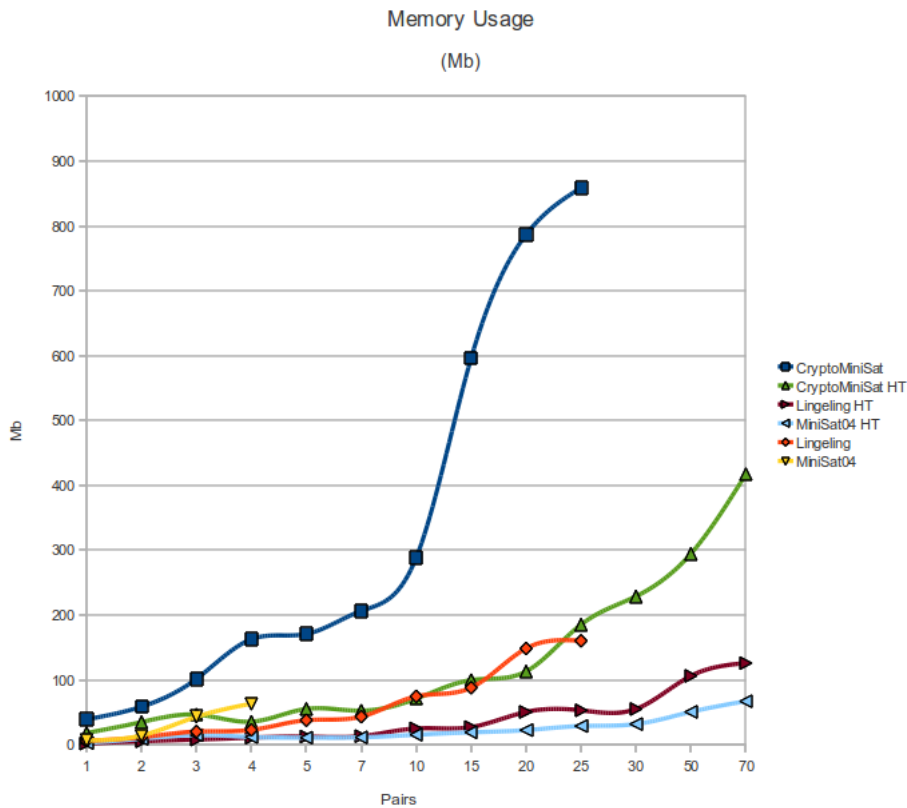


Ovviamente, la linea che nel grafico appare più in basso corrisponde ad un minor utilizzo delle risorse. Per quanto riguarda istanze di problemi *piccoli*, come nel caso di DES con un solo round, MiniSat è il risolutore che si comporta meglio. CryptoMiniSat utilizza generalmente un grande quantitativo di memoria, ma come si vedrà sarà l'unico SAT solver ad essere di aiuto al crescere dei rounds.

#### 4.3.2 Risultati Sperimentali Crittoanalisi: 2 rounds

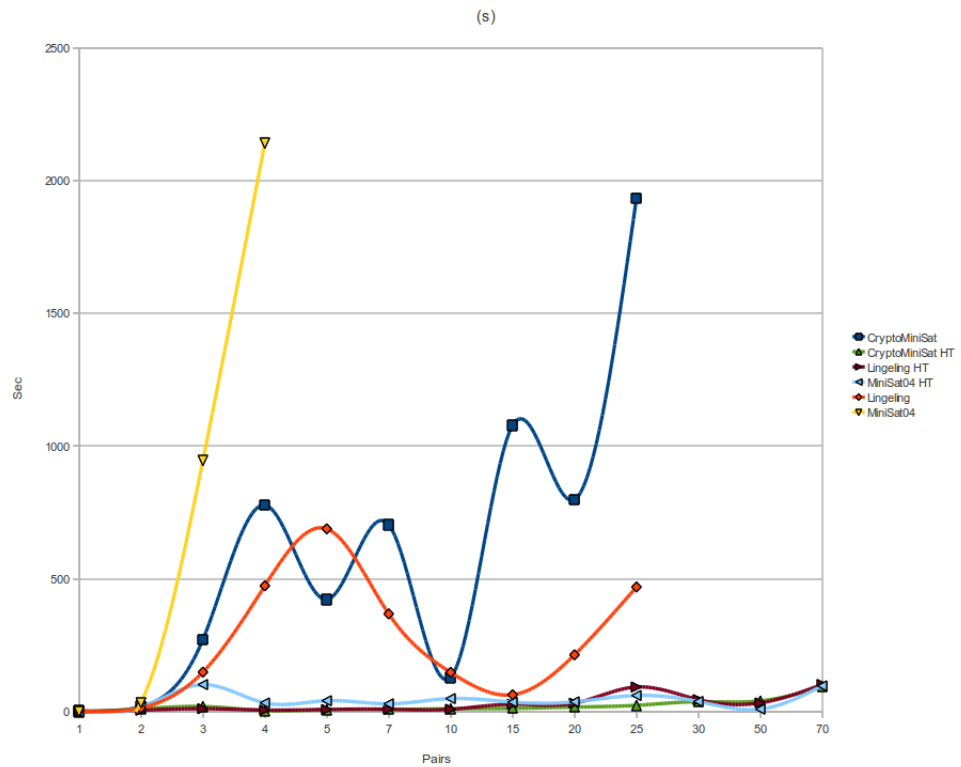
Sebbene l'*avalanche effect* sia incompleto, nel caso dei due rounds si presentano le prime formule impegnative, in cui cioè tutti i bits del testo cifrato sono rappresentati da una espressione non banale. Vengono generate anche le prime 32 variabili ausiliarie, utilizzate nella rete di Feistel per mantenere lineare la dimensione delle formule durante la loro elaborazione.

L'andamento dell'utilizzo di memoria è rappresentato dal primo grafico:



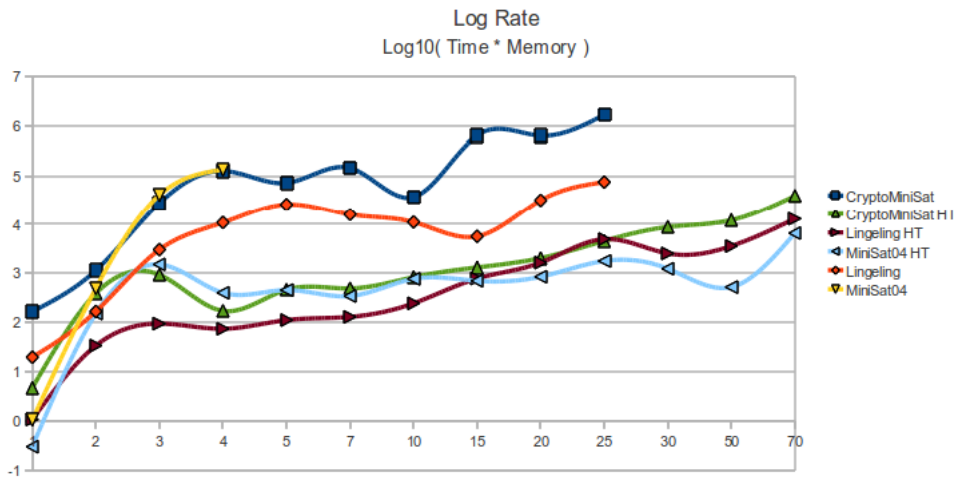
I problemi si fanno più difficili, sia dal punto di vista della dimensione che dal lato della complessità. Il primo a risentirne è MiniSat a cui sono applicate le formule generate dal metodo di Tseitin ricorsivo. L'esplosione computazionale è netta e il DIMACS contenente 4 coppie  $(P, C)$  si dimostra un problema insuperabile. A testimonianza del fatto che le formule in cui è stato gestito intelligentemente lo *sharing* (mediante `applySmartTseitin()`) sono decisamente più trattabili, si può notare come lo stesso programma, a cui però è chiesto di risolvere problemi HT, si comporti in maniera molto buona, riuscendo terminare egregiamente il test. In modo analogo, CryptoMiniSat applicato alle formule generate senza eliminare le ridondanze utilizza molta memoria, mentre l'andamento dello stesso programma nel caso in cui sia stato opportunamente trattato lo *sharing* ha un andamento quasi lineare. Lingeling si dimostra ottimo in tutti i casi.

Il grafico seguente rappresenta l'andamento temporale:



Come si era già visto nel grafico dell'utilizzo di memoria, MiniSat applicato su istanze non HT ha un andamento esponenziale, tipico del problema  $\mathcal{NP}$ -completo che sta cercando di risolvere. Dopo un certo *timeout*, la sua computazione è stata interrotta manualmente. CryptoMiniSat e Lingeling hanno un andamento piuttosto altalenante, probabilmente dovuto alle euristiche che entrano in gioco a seconda delle singole istanze dei problemi in esame. L'applicazione degli stessi programmi su problemi "HT" però linearizza il tempo di risoluzione di tutti i SAT solver, che terminano il test con ottimi risultati.

Il grafico logaritmico, che funge da "riepilogo" della situazione vista finora, è il seguente:

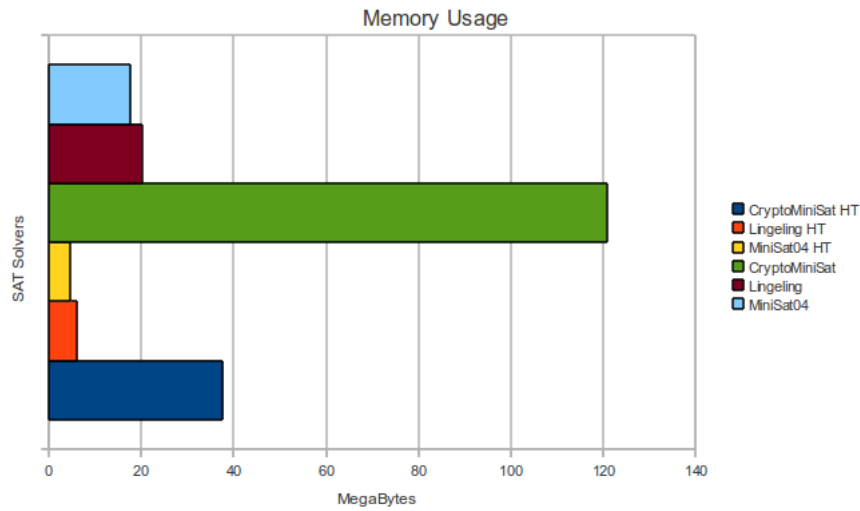


La crescita iniziale più repentina è quella di MiniSat applicato ad un problema non HT. Se non ne fosse stata interrotta l'esecuzione manualmente, al riguardo il grafico sarebbe ben più impietoso. CryptoMiniSat applicato a problemi non HT ha in generale l'andamento peggiore ma riesce comunque a operare in maniera sub-esponenziale. Aumentando il numero di rounds, aumenta anche il divario di prestazioni tra i solvers applicati ai problemi generati in maniera ricorsiva rispetto a quelli che sfruttano le ottimizzazioni dello sharing. Questo è soprattutto dovuto al fatto che al crescere della complessità delle formule, il meccanismo di individuazione di sottoalberi ripetuti diventa sempre più efficace, avendo più materiale da elaborare e confrontare.

### 4.3.3 Risultati Sperimentali Crittoanalisi: 3 rounds

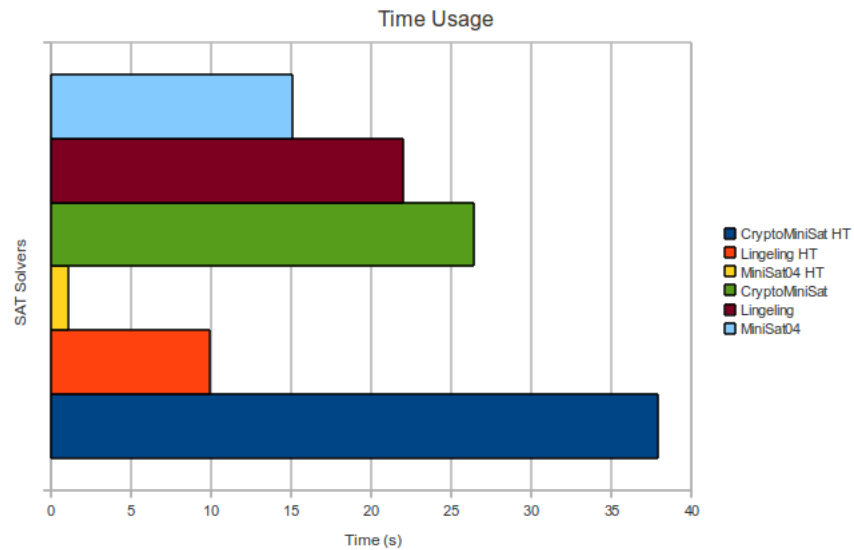
Il raggiungimento del terzo round ha delineato in maniera evidente le differenze tra i SAT solvers. A causa della difficoltà incontrata nel risolvere le formule ottenute, si è potuto ottenere dati riguardanti solo un attacco *known-plaintext* basato su una sola coppia  $(P, C)$ , di cui tutti i SAT solvers sono riusciti a trovare un modello. Portare il numero di coppie a 2 ha innalzato la difficoltà del problema al di là delle capacità dei risolutori: in 25 ore nessuno di essi è riuscito a verificarne la soddisfacibilità.

La prima tabella illustra l'utilizzo di memoria nell'unico caso che è stato trovato risolvibile:



In linea con quanto mostrato finora, CryptoMiniSat ha un grande consumo di memoria, i problemi HT sono di dimensione inferiore a quelli generati senza ottimizzazioni riguardanti lo *sharing* e MiniSat fa ancora una volta un uso parsimonioso della memoria che gli viene messa a disposizione.

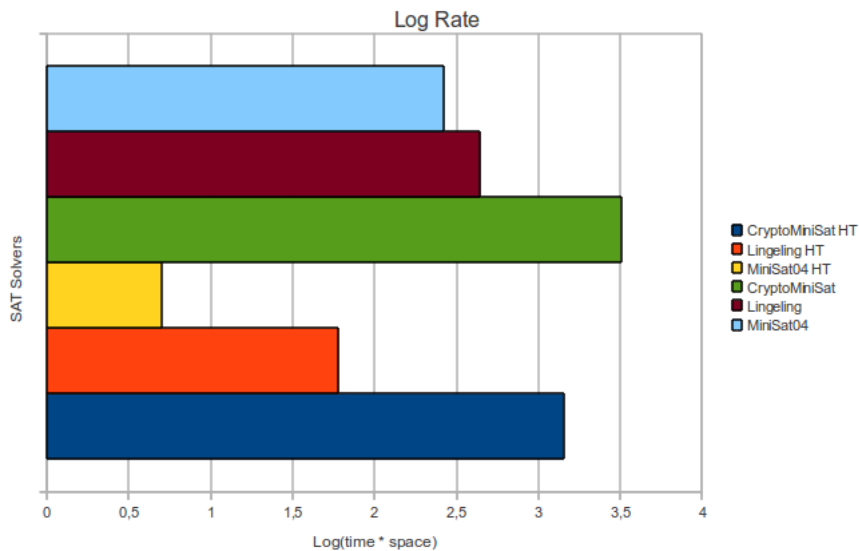
Per quanto riguarda il tempo impiegato nella risoluzione si ha che:



In questo caso, CryptoMiniSat HT utilizza solo (circa) un terzo della memoria usata dallo stesso programma a cui è chiesto di risolvere il problema senza le ottimizzazioni che riguardano lo *sharing*, ma utilizza circa un terzo

del tempo in più. MiniSat HT, oltre ad aver usato in assoluto meno memoria, detiene anche il primato del tempo impiegato. Lingeling come sempre si trova posizionato a metà strada dai valori estremi.

L'ultimo grafico è quello logaritmico:



Questa sorta di “riepilogo” indica che in questo test MiniSat si è comportato decisamente meglio di tutti gli altri SAT solvers. Sfortunatamente, aumentare il numero di coppie di *plaintexts* e *ciphertext* causa una sensibile impennata nella complessità del problema. Tale fatto non ha permesso di ottenere maggiori statistiche, con i mezzi computazionali a disposizione.

#### 4.3.4 Risultati Sperimentali Crittoanalisi: 4 rounds

Il passaggio tra i tre e i quattro rounds costituisce un punto cruciale per una sperimentazione di questo tipo. E' infatti dimostrato che tre rounds di DES non costituiscono ancora una *pseudorandom permutation*, mentre dai quattro in poi l'*avalanche effect* inizia ad essere quasi completo, raggiungendo un buon livello a 5 rounds.

Per questo tipo di attacco *known-plaintext* non sono disponibili statistiche di confronto tra SAT solvers: l'unico di essi che è riuscito a trovare un modello per un DIMACS estremamente ottimizzato è stato CryptoMiniSat, che è giunto al termine dell'elaborazione utilizzando 110,94 Mb di memoria in un tempo di 531,33 secondi. Sono stati tentati attacchi dello stesso tipo anche a

versioni di DES con 5 e 6 rounds, usando CryptoMiniSat, che sembra essere il SAT solver che complessivamente si adatta meglio a problemi complessi e/o di grosse dimensioni.

Per quanto riguarda i 5 rounds, dopo 722826.30 secondi di elaborazione, poco più di 8 giorni, (con un utilizzo di ben 2387.48 Mb di RAM) il SAT solver si è “arreso”, sentenziando che la ricerca era da ritenersi “INCONCLUSIVE”<sup>49</sup>. L’attacco su DES a 6 rounds è durato più di 30000 minuti, cioè più di 20 giorni, e non ha dato (finora) alcun responso.

---

<sup>49</sup>Nota bene: non UNSAT, cioè insoddisfacibile.



## 5 Conclusioni e sviluppi futuri

*“I spare not a single unit of thought on these cybernetic simpletons!” he boomed. “I speak of none but the computer that is to come after me!”*

*Fook was losing patience. He pushed his notebook aside and muttered, “I think this is getting needlessly messianic.”*

– Douglas Adams, *“The Hitchhiker’s Guide to the Galaxy”*

I concetti presentati in questa tesi sono frutto della personale elaborazione di una soluzione per applicare un attacco crittoanalitico di tipo logico attraverso un tool automatico. I risultati precedentemente ottenuti dagli ideatori di questa tecnica, Massacci e Marraro [2], sono stati uguagliati e superati, riuscendo a forzare anche il quarto round, in cui per la prima volta gli effetti dell'*avalanche effect* divengono tangibili. Le soluzioni e gli spunti a cui si è accennato nel corso della dissertazione sono solo un sottoinsieme delle possibili e innumerevoli vie che sono state percorse nella vasta letteratura scientifica che ha trattato argomenti affini. Ogni scelta implementativa che si compie, a partire dalla struttura con cui rappresentare le formule, offre nuove soluzioni per alcuni problemi e contemporaneamente limita l'applicazione di altri metodi che potrebbero risultare utili. Sicuramente esistono un gran numero di ottimizzazioni da applicare alle formule in seguito alla loro generazione, alcune delle quali sono descritte in [19, 18, 17, 16] e non hanno ancora trovato il meritato spazio all'interno di Samael. D'altro canto, il problema della soddisfacibilità logica rimane estremamente complesso, e, per quanto gli sviluppatori di SAT solver facciano del loro meglio per limare un margine sempre maggiore di efficienza, lo scoglio della  $\mathcal{NP}$ -completezza rimane pressoché insormontabile. In questo senso, la scelta di DES non facilita certo le cose: nonostante “l'età” rimane infatti un ingegnosissimo e tuttora matematicamente inviolato tempio perduto della crittografia. Fino a che la crittoanalisi logica non avrà degli strumenti veramente adeguati per competere con meccanismi come quelli visti finora, risultati migliori potranno essere ottenuti concentrandosi su sistemi di cifratura più deboli [7]. Una alternativa potrebbe invece consistere nell'associare questa tecnica ad altre più sperimentate, come la crittoanalisi differenziale o lineare.

Per correttezza, va anche ricordato che durante la fase di verifica sperimentale dei concetti esposti in questa tesi non è stato utilizzato alcun sistema che garantisca una adeguata capacità di calcolo: sono infatti stati usati dei semplici personal computers. In mancanza di mezzi migliori, la soluzione più alla portata può essere individuata nel calcolo distribuito, come per esempio nel caso di MPI (Message Passing Interface). Alcuni SAT solver, come PMSat, sono stati pensati principalmente per tale utilizzo, e del resto proprio questo procedimento potrebbe permettere di raggiungere la *massa critica* computazionale per poter seriamente cercare di trovare un modello per istanze di attacco riferite a 5 o 6 rounds di DES.

# Ringraziamenti

Alla mia ragazza Martina, che in questa avventura a Bologna mi ha quotidianamente *sopportato* e *supportato* in modo unico, dolce e insostituibile;  
a Mello, un instancabile *code monkey* con la passione del rock e dello *scrolling*,  
e a Mofo, una specie di *Begbie veneto* che si è assicurato che in questi anni io rispettassi la regola:

*“Qualunque cosa facciate nella vita, circondatevi di persone intelligenti che vi contestino.”*

Con loro, nel bene e nel male (ma soprattutto nel male!), ho condiviso tre lunghi anni di progetti, senza mai far mancare la giusta dose di *flash games*, stupidità e allegria;

al Dottor Ugo Dal Lago, che si è sempre mostrato professionale, gentile, paziente e che ha il pregio di saper sempre fornire spunti lungimiranti;  
alla mia famiglia, che mi ha finanziato e spronato in tutti questi anni;  
a tutte queste persone va il mio più sincero ringraziamento.

## Riferimenti bibliografici

- [1] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, Wiley, 1994.
- [2] F. Massacci e L. Marraro, *Logical Cryptanalysis as a SAT Problem*, in *Journal of Automated Reasoning* 24: 165–203, 2000.
- [3] K. Campbell e M. Weiner, *DES is not a group*, in *Proc. of Advances in Cryptography (CRYPTO-92)*, Lecture Notes in Comput. Sci., Springer-Verlag, 1992.
- [4] Jonathan Katz and Yehuda Lindell, *Introduction to modern Cryptography*, Chapman & Hall.
- [5] C. Shannon, *Communication Theory of Secrecy Systems*, Bell System Technical Journal 28, 1949.
- [6] E. Biham e A. Shamir, *Differential cryptanalysis of DES-like cryptosystems*, Technical report CS90-16, Weizmann Institute of Science CRYPTO'90 & Journal of Cryptology, 1991. Vol. 4, No. 1.
- [7] M. Soos, K. Nohl e C. Castelluccia, *Extending SAT Solvers to Cryptographic Problems*, in 12th International Conference on Theory and Applications of Satisfiability Testing (SAT), 2009, pp. 244-257.
- [8] A. Asperti e A. Ciabattoni, *Logica ad Informatica*, McGraw Hill, 1997
- [9] M. Davis e H. Putnam, *A Computing Procedure for Quantification Theory*, J. ACM 7(3) (1960), 201–215.
- [10] P. Manolios e D. Vroom, *Efficient Circuit to CNF Conversion*, SAT 2007, The Tenth International Conference on Theory and Applications of Satisfiability Testing, May 2007.
- [11] B. Chambers, P. Manolios e D. Vroom, *Faster SAT Solving with Better CNF Generation*, DATE 2009, Design Automation and Test in Europe, April 2009.
- [12] R. Brummayer e A. Biere, *Local Two-Level And-Inverter Graph Minimization without Blowup*, In Proc. 2nd Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'06), Mikulov, Czechia, October 2006.

- [13] P. Jackson e D. Sheridan, *Clause Form Conversions for Boolean Circuits*, (In: Prel. Proc. SAT'04)
- [14] M. Gabbrielli e S. Martini, *Linguaggi di Programmazione: principi e paradigmi*, McGraw-Hill Italia, 2006
- [15] G.S. Tseitin, *On the complexity of derivation in propositional calculus*, Zapiski nauchnykh seminarov LOMI, 8:234–259, 1968. English translation: Consultants Bureau, N.Y., 1970, pp. 115–125.
- [16] Niklas Eén e Armin Biere, *Effective Preprocessing in SAT Through Variable and Clause Elimination*, SAT 2005: 61-75
- [17] Matti Jarvisalo, Armin Biere e Marijn Heule, *Blocked Clause Elimination*, In J. Esparza and R. Majumdar (Eds.), TACAS '10, pp. 129–144. Lecture Notes in Computer Science 6015, Springer.
- [18] Sean Weaver, *A CNF Analogue to Strengthening*, Morehead Electronic Journal of Applicable Mathematics Issue 3 — CS-2002-02
- [19] Lintao Zhang, *On Subsumption Removal and On-the-Fly CNF Simplification*, In: 8th International Conference on the Theory and Applications of Satisfiability Testing (SAT 2005)
- [20] Mate Soos, *CryptoMiniSat 2.5.0*, URL: [http://baldur.iti.uka.de/sat-race-2010/descriptions/solver\\_13.pdf](http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_13.pdf)
- [21] Armin Biere, *Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010*, URL: [http://baldur.iti.uka.de/sat-race-2010/descriptions/solver\\_1+2+3+6.pdf](http://baldur.iti.uka.de/sat-race-2010/descriptions/solver_1+2+3+6.pdf)
- [22] Niklas Sörensson e Niklas Een, *MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization*
- [23] MiniSat Home Page, URL: <http://minisat.se/Papers.html>
- [24] FIPS PUB 46-3 , *DATA ENCRYPTION STANDARD (DES)*, U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, 1999 October 25, URL: <http://www.itl.nist.gov/fipspubs/fip46-2.htm>