UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica

Studio e analisi di cifrari utilizzati in ambito mobile

Relatore: Prof. Andrea VISCONTI

Alessandro RAVIZZOTTI
Matricola: 922845

Anno Accademico: 2020 - 2021

Indice

Prefazione **5**

1 A5/1 **6**
1.1 Introduzione 6
1.2 Descrizione 6
1.2.1 Registri 6
1.2.2 Ciclo dei Registri 7
1.2.3 Regola di Maggioranza 8
1.2.4 Le fasi 9
1.3 Attacchi 17

2 A5/2 **20**
2.1 Introduzione 20
2.2 Descrizione 20
2.2.1 Registri 21
2.2.2 Ciclo dei Registri 22
2.2.3 Calcolo bit di output 23
2.2.4 Le fasi 24
2.3 Attacchi 32

3 A5/3 **34**
3.1 Introduzione 34
3.2 Funzione KGCORE 34
3.3 Kasumi 36
3.3.1 FL 38
3.3.2 FO 39
3.3.3 FI 40
3.3.4 S-Box 41
3.4 Attacchi 42
3.5 Attacchi nella Storia dei tre cifrari 44

4 Attacco teorico a A5/1 - Attacco di Biham e Dunkelman **46**
4.1 Fase 1 . . . 46
4.2 Fase 2 . . . 48
4.3 Fase 3 . . . 49
4.4 Ritrovamento stato chiave . . . 50
4.4.1 100 *reverse clock* . . . 50
4.4.2 *Reverse* XOR frame . . . 51
4.5 Costo Totale . . . 52

5 Attacco pratico a A5/1 **54**
5.1 Struttura delle cartelle . . . 54
5.2 Il file di compilazione . . . 55
5.3 Le costanti . . . 56
5.4 Le funzioni di base di A5/1 . . . 56
5.5 L'albero degli stati . . . 58
5.6 L'attacco . . . 60
5.6.1 Fase 1 . . . 61
5.6.2 Fase 2 . . . 62
5.6.3 Fase 3 . . . 64
5.7 Il ritrovamento dello stato chiave . . . 67
5.7.1 100 *reverse clock* . . . 67
5.7.2 *Reverse* XOR frame . . . 72
5.8 Specifiche e durata . . . 73

Conclusioni **75**

Bibliografia **77**

Prefazione

Negli ultimi anni la crittografia, sopratutto in ambito mobile, ha acquisito un interesse sempre maggiore a causa della crescente informatizzazione della società e del continuo aumento dell'importanza della segretezza e dell'anonimato dei dati.

Sono sempre stato interessato alle problematiche relative alla sicurezza informatica, specie se riferite al mondo mobile e con questa tesi, sviluppata all'interno del Dipartimento di Informatica, ho avuto la possibilità di approfondire la tematica verificando in modo pratico la vulnerabilità dei cifrari analizzati.

Ho affrontato l'argomento partendo dallo studio dei cifrari del GSM per comprenderne il funzionamento e ho analizzato i vari attacchi che nel tempo sono stati ideati ed effettuati, con l'obiettivo di trovare quello più efficiente.

Individuato quest'ultimo nell'attacco a A5/1 ideato dai ricercatori Biham e Dunkelman, ho elaborato un personale approfondimento pratico, implementando, in linguaggio C, l'attacco al singolo *frame* fino a risalire allo stato chiave, sufficiente per riuscire a cifrare e decifrare ogni *frame* di un utente.

Ritengo di aver ottenuto risultati più che soddisfacenti, soprattutto in funzione di possibili sviluppi futuri riguardo all'esecuzione, in tempi ragionevoli, dell'attacco completo.

Capitolo 1

A5/1

1.1 Introduzione

A5/1 è un cifrario a flusso sviluppato nel 1987 e tenuto segreto fino al 1994 quando una serie di informazioni riguardanti l'algoritmo diventano di pubblico dominio a causa di "leak". Successivamente, nel 1999, Marc Briceno scopre l'esatto funzionamento dei cifrari A5/1 e A5/2 tramite "reverse engineering". L'algoritmo viene utilizzato per proteggere le comunicazioni tra i telefoni e le stazioni radio che utilizzano il GSM (*Global System for Mobile Communications*) in Europa e negli Stati Uniti. Con GSM le informazioni vengono trasmesse sotto forma di *frame* di 114 bit.

In questo e nei prossimi Capitoli il termine "*frame*" corrisponde convenzionalmente ai 114 bit di informazioni trasmesse tramite GSM, mentre "frame" è la chiave pubblica di 22 bit.

1.2 Descrizione

Il cifrario A5/1 utilizza una chiave a 64 bit, un frame pubblico di 22 bit e 3 LFSR (*Linear Feedback Shift Registers*) di lunghezze diverse per generare la *key stream* che serve per cifrare i dati tramite una semplice operazione di XOR.

1.2.1 Registri

I 3 registri a scorrimento sono indicizzati con LSB (*Least Significant Bit*) e presentano le seguenti caratteristiche.

Reg. LFSR	Lunghezza Reg. (bit)	Polinomio caratteristico	Bit di controllo	Bit calcolo XOR
R_1	19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	8	13, 16, 17, 18
R_2	22	$x^{22} + x^{21} + 1$	10	20, 21
R_3	23	$x^{23} + x^{22} + x^{21} + x^8 + 1$	10	7, 20, 21, 22

Tabella 1.1: *Informazioni dei registri a scorrimento a retroazione lineare*

1.2.2 Ciclo dei Registri

All'interno dell'algoritmo ogni registro deve ripetere un ciclo per molteplici volte. Il ciclo, che da ora in avanti sarà chiamato *clock*, è composto da due parti:

- Calcolo del valore da assegnare al bit 0
- Shift a destra di 1

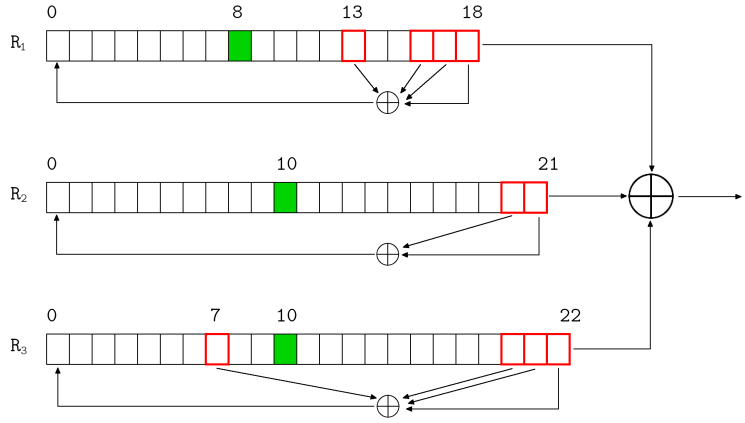


Figura 1.1: *Clock LFSR*
- Le caselle colorate in verde corrispondono ai bit utilizzati nella funzione di maggioranza
- Le caselle contornate di rosso sono quelle utilizzate per le operazioni di XOR

Calcolo bit 0

Per calcolare il bit 0 viene utilizzata l'operazione di XOR. Essa viene eseguita su bit differenti in base al registro e al corrispettivo polinomio caratteristico.

- In R_1 vengono utilizzati i bit: 13, 16, 17 e 18
- In R_2 vengono utilizzati i bit: 20 e 21
- In R_3 vengono utilizzati i bit: 7, 20, 21 e 22

In ogni registro il risultato dell'operazione viene scritto nel LSB dopo lo *shift*.

Shift

Il registro viene shiftato a destra di 1 bit. Prima di compiere questa operazione si calcola l'output del *round* XORando gli MSB dei singoli registri.

$$outputBit = R_1[18] \oplus R_2[21] \oplus R_3[22]$$

1.2.3 Regola di Maggioranza

La Regola di maggioranza utilizza 3 bit di controllo:

- $R_1[8]$
- $R_2[10]$
- $R_3[10]$

Essa determina il valore presente con maggiore frequenza. Se almeno due bit sono uguali ad 1, allora il bit di maggioranza sarà uguale ad 1. In caso contrario il bit di maggioranza sarà uguale a 0.

Quando in un *round* viene utilizzata la regola di maggioranza, i registri, R_1, R_2 e R_3 compiono un *clock* solamente se il relativo bit di controllo è uguale al bit di maggioranza, come nella tabella che segue.

Ogni registro ha quindi una probabilità di *clock* del 75%.

$R_1[8]$	$R_2[10]$	$R_3[10]$	Bit di Maggioranza	R_1	R_2	R_3
0	0	0	0	clock	clock	clock
0	0	1	0	clock	clock	still
0	1	0	0	clock	still	clock
0	1	1	1	still	clock	clock
1	0	0	0	still	clock	clock
1	0	1	1	clock	still	clock
1	1	0	1	clock	clock	still
1	1	1	1	clock	clock	clock

Tabella 1.2: *Regola di Maggioranza*

1.2.4 Le fasi

Il cifrario A5/1 è diviso in due macro fasi, la prima di inizializzazione e la seconda di cifratura dei dati inviati dal dispositivo mobile o dalla stazione radio.

Inizializzazione

L'inizializzazione si suddivide in 4 fasi.

Prima fase

Nella prima fase vengono impostati tutti i bit dei registri R_1, R_2 e R_3 uguali a 0.

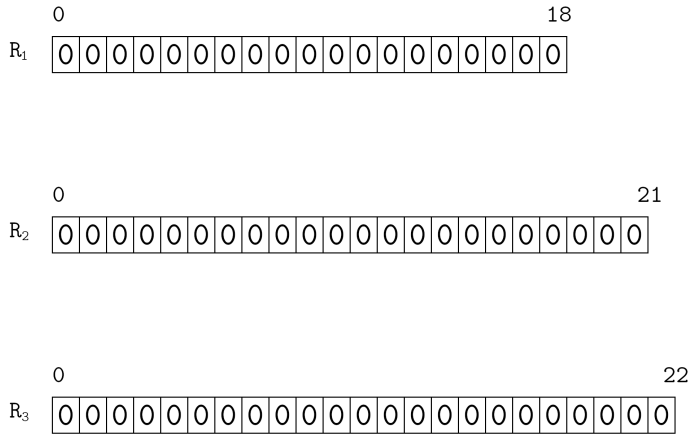


Figura 1.2: *Registri dopo la prima fase.*

Seconda fase

La seconda fase consiste nel combinare un bit della chiave segreta, tramite XOR, al bit meno significativo di ogni registro. Dopo queste operazioni i registri R_1, R_2 e R_3 eseguono un *clock* senza l'utilizzo della regola di maggioranza e scartando il bit di output prodotto. I passaggi descritti, compongono 1 *round* e vengono ripetuti per 64 volte. Ad ogni *round* il bit della chiave segreta utilizzato nei calcoli è quello nella posizione i-esima ($0 \leq i < 64$), dove i equivale al numero dell'iterazione.

for i:= 0 **to** 63 **do**:

$R_1[0] = R_1[0] \oplus Key[i]$

$R_2[0] = R_2[0] \oplus Key[i]$

$R_3[0] = R_3[0] \oplus Key[i]$

clock i 3 registri (senza l'utilizzo della regola di maggioranza)

Esempio del primo *round*

Le prime operazioni coinvolgono il bit meno significativo di ogni registro $R_1[0]$, $R_2[0]$, $R_3[0]$ e il bit della chiave $Key[0]$ in quanto al primo *round* $i = 0$. Si ricorda che, dalla prima fase, i bit dei registri sono tutti uguali a 0. I calcoli eseguiti sono i seguenti:

$R_1[0] = R_1[0] \oplus Key[0] = 0 \oplus 1 = 1$

$R_2[0] = R_2[0] \oplus Key[0] = 0 \oplus 1 = 1$

$R_3[0] = R_3[0] \oplus Key[0] = 0 \oplus 1 = 1$

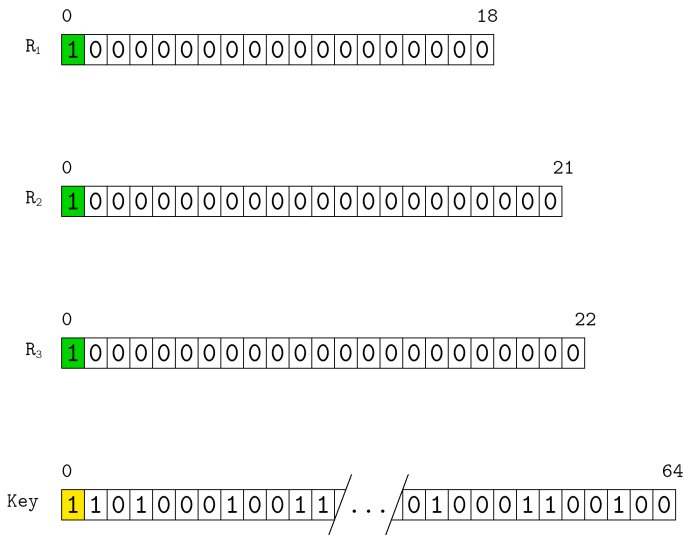


Figura 1.3: *Registri dopo l'operazione di XOR con la chiave.*
- Le caselle colorate in verde indicano i bit sovrascritti.
- La casella della chiave colorata in giallo indica l'i-esimo bit che viene utilizzato in questo round per le operazioni.

Dopo le operazioni con la chiave segreta i tre registri devono eseguire il *clock*. Nell'immagine che segue sono evidenziati tutti i bit coinvolti nelle operazioni del calcolo del bit 0. Si ricorda che il bit meno significativo viene sovrascritto solamente dopo l'operazione di *shift* e l'output derivato da $R_1[18] \oplus R_2[21] \oplus R_3[22]$ viene scartato.

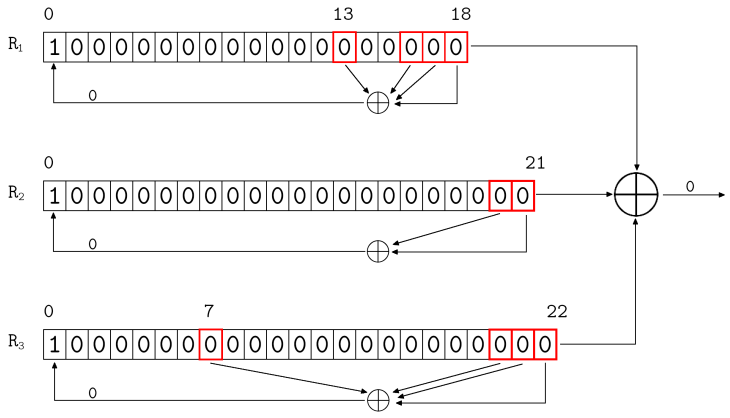


Figura 1.4: *Calcolo dei bit 0 e dei bit di output. I bit utilizzati per le operazioni sono quelli contornati in rosso. Il bit di output viene scartato.*

Nell'immagine seguente sono presenti i tre registri dopo l'esecuzione del *clock*.

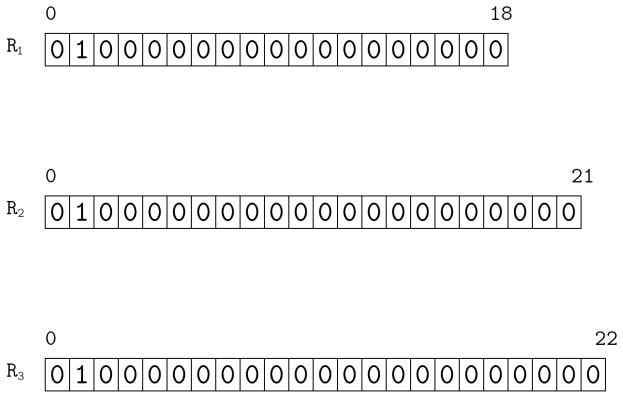


Figura 1.5: *Registri dopo l'esecuzione del clock, alla fine del primo round della seconda fase.*

A questo punto il *round* è finito e per completare la seconda fase occorre ripetere per 64 volte tutta la procedura, corrispondente ai passaggi illustrati nell'esempio.

Terza fase

La terza fase, molto simile alla seconda, consiste nel combinare un bit del frame pubblico, tramite XOR, al bit meno significativo di ogni registro. Dopo queste operazioni i registri R_1, R_2 e R_3 eseguono un *clock* senza l'utilizzo della regola di maggioranza e scartando il bit di output prodotto. I passaggi descritti, compongono 1 *round* e vengono ripetuti per 22 volte. Ad ogni *round* il bit del frame utilizzato nei calcoli è quello nella posizione i-esima ($0 \leq i < 22$), dove i equivale al numero dell'iterazione.

for i:=0 **to** 21 **do**:

$R_1[0] = R_1[0] \oplus Frame[i]$

$R_2[0] = R_2[0] \oplus Frame[i]$

$R_3[0] = R_3[0] \oplus Frame[i]$

clock i 3 registri (senza l'utilizzo della regola di maggioranza)

Esempio del primo *round*

Le prime operazioni coinvolgono il bit meno significativo di ogni registro $R_1[0]$, $R_2[0]$, $R_3[0]$ e il bit del frame $Frame[0]$ in quanto al primo *round* $i = 0$. Si ricorda che i valori di partenza dei registri di questa fase sono stati computati nella fase precedente. I calcoli eseguiti sono i seguenti:

$R_1[0] = R_1[0] \oplus Frame[0] = 1 \oplus 1 = 0$

$R_2[0] = R_2[0] \oplus Frame[0] = 0 \oplus 1 = 1$

$R_3[0] = R_3[0] \oplus Frame[0] = 1 \oplus 1 = 0$

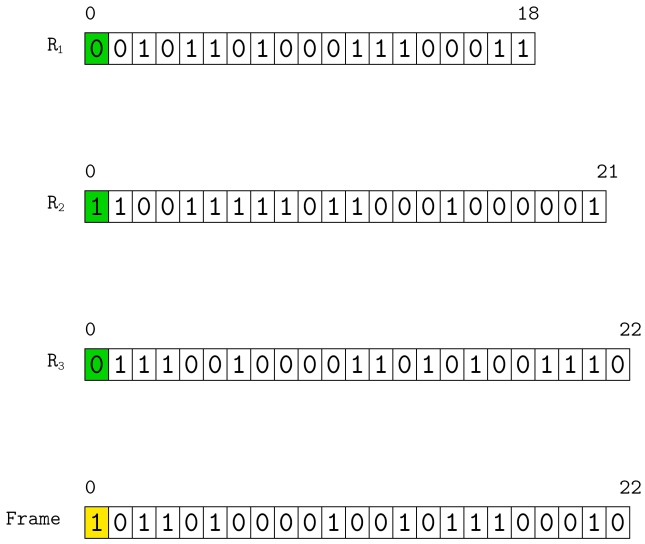


Figura 1.6: *Registri dopo l'operazione di XOR con il frame.*
- Le caselle colorate in verde indicano i bit sovrascritti.
- La casella del frame colorata in giallo indica l'i-esimo bit che viene utilizzato in questo round per le operazioni.

Dopo le operazioni con il frame pubblico i tre registri devono eseguire il *clock*. Nell'immagine che segue sono evidenziati tutti i bit coinvolti nelle operazioni del calcolo del bit 0. Si ricorda che il bit meno significativo viene sovrascritto solamente dopo l'operazione di *shift* e l'output derivato da $R_1[18] \oplus R_2[21] \oplus R_3[22]$ viene scartato.

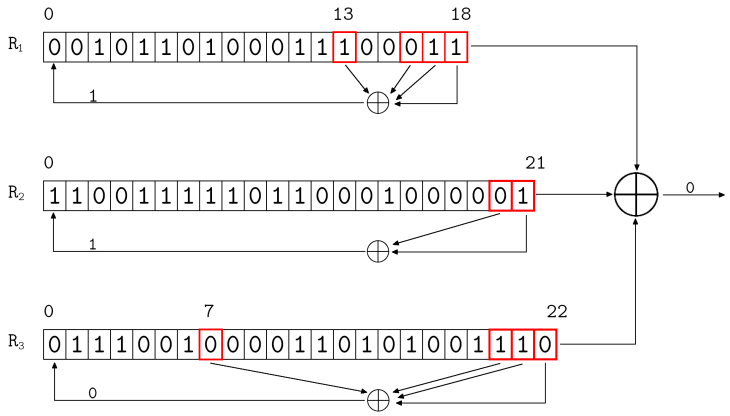


Figura 1.7: *Calcolo dei bit 0 e dei bit di output. I bit utilizzati per le operazioni sono quelli contornati in rosso. Il bit di output viene scartato.*

Nell'immagine seguente sono presenti i tre registri dopo l'esecuzione del *clock*.

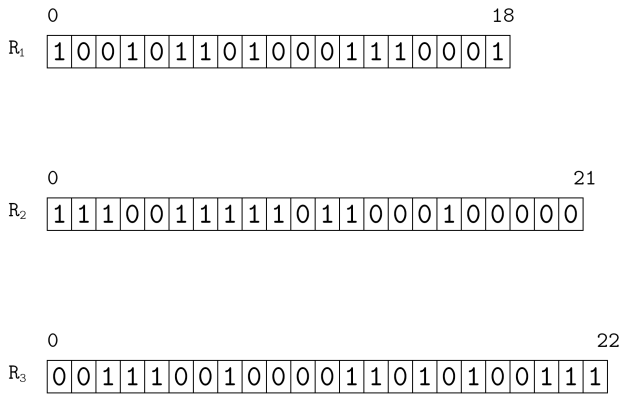


Figura 1.8: *Registri dopo l'esecuzione del clock, alla fine del primo round della terza fase.*

A questo punto il *round* è finito e per completare la terza fase occorre ripetere per 22 volte tutta la procedura, corrispondente ai passaggi illustrati nell'esempio.

Quarta fase
Nell'ultima fase i registri devono eseguire il *clock* mediante l'utilizzo della regola di maggioranza scartando l'output prodotto. Questo passaggio, detto *round*, viene ripetuto per 100 volte.

```
for i:=0 to 99 do:
    clock secondo la regola di maggioranza
```

Esempio del primo *round*
La prima operazione che viene effettuata è il calcolo del bit di maggioranza, dal quale si può determinare quali registri devono eseguire il *clock*.

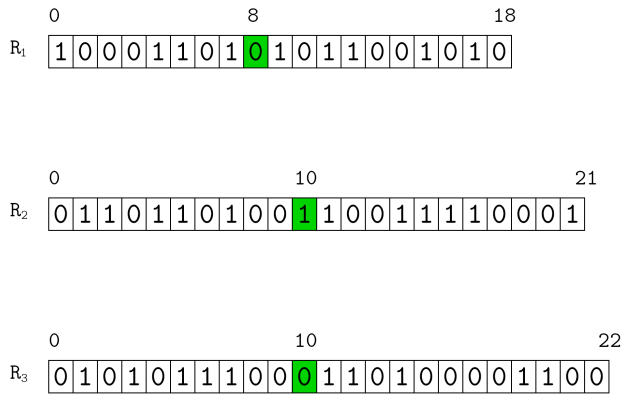


Figura 1.9: *Calcolo del bit di maggioranza.*
- Le caselle colorate in verde indicano quali bit sono stati utilizzati nell'operazione.

Il bit di maggioranza calcolato è uguale a 0, quindi solamente R_1 e R_3 eseguono il *clock*. Nell'immagine che segue sono evidenziati tutti i bit coinvolti nelle operazioni del calcolo del bit 0. Si ricorda che il bit meno significativo viene sovrascritto solamente dopo l'operazione di *shift* e l'output derivato da $R_1[18] \oplus R_2[21] \oplus R_3[22]$ viene scartato.

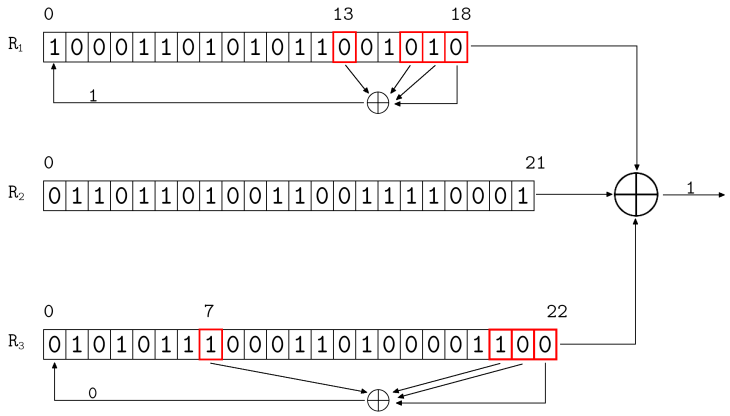


Figura 1.10: *Calcolo dei bit 0 e dei bit di output. I bit utilizzati per le operazioni sono quelli contornati in rosso. Il bit di output viene scartato.*

Nell'immagine seguente sono presenti i tre registri dopo l'esecuzione del *clock*.

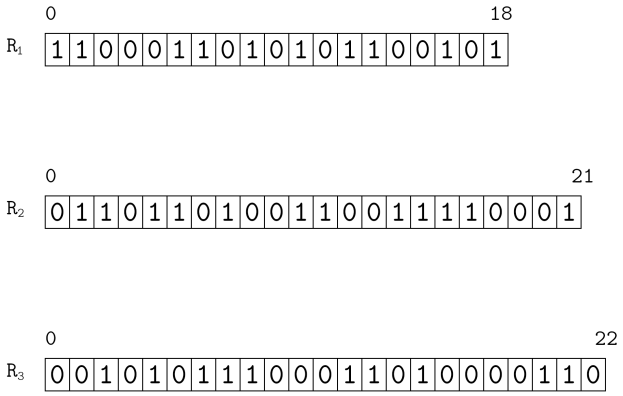


Figura 1.11: *Registri dopo l'esecuzione del clock, alla fine del primo round della quarta fase.*

A questo punto il *round* è finito e per completare la quarta fase occorre ripetere per 100 volte tutta la procedura, corrispondente ai passaggi illustrati nell'esempio.
Dopo le 100 iterazioni l'inizializzazione è terminata.

Cifratura

Dopo aver svolto tutte le quattro fasi dell'inizializzazione i registri sono pronti per cifrare i dati. Vengono eseguiti ulteriori 228 *round*, utilizzando la regola di maggioranza e ogni output prodotto serve per calcolare 1 bit della *key stream* di 228 bit, 114 bit servono per cifrare i dati dalla centrale fino al dispositivo mobile, mentre i restanti 114 bit vengono utilizzati per cifrare i dati dal telefono alla stazione.

for i:=0 **to** 227 **do**:

clock secondo la regola di maggioranza

$keyStream[i] = R_1[output] \oplus R_2[output] \oplus R_3[output]$

114 bit del testo in chiaro vengono cifrati con i 114 bit della *key stream*.

$$keyStream \oplus plainText = cipherText$$

Per cifrare il *frame* successivo si ripetono le operazioni dal primo punto dell'inizializzazione utilizzando la stessa chiave segreta ma il frame viene incrementato di 1.

1.3 Attacchi

Nonostante A5/1 utilizzi una chiave a 64 bit, gli ultimi 10 bit sono impostati a 0 e ciò comporta una notevole riduzione dei possibili stati della chiave da 2^{64} a 2^{54}. Questa criticità permette ad agenzie governative o altri enti in possesso di computer molto potenti di decodificare in tempi brevi le conversazioni mobile.

Nelle tabelle sotto riportate sono illustrati i principali attacchi effettuati nella storia al cifrario A5/1.

Anno	Autori	Tipologia	Complessità
1997	Javon Golic	Divide et impera	$2^{40.16}$
2000	Eli Biham e Orr Dunkelman	Divide et impera	$2^{39.91}$
2000	Alex Biryukov, Adi Shamir e David Wagner	Biased birthday attack	2^{42}
2000	Alex Biryukov, Adi Shamir e David Wagner	Random subgraph attack	2^{48}
2000	Ekdahl e Johannson	Correlation Attack	

Tabella 1.3: *Attacchi nella storia - Testo in chiaro noto*

Anno	Autori	Tipologia	Altro
2003	Elad Barkan, Eli Biham e Nathan Keller	Time-memory tradeoff	
2005	Elad Barkan e Eli Biham	Correlation Attack	
2007	Le università di Bochum e Kiel	Time-memory tradeoff	COPACOBANA
2008	The Hackers Choice	Time-memory tradeoff	
2009	Chris Paget e Karsten Nohl	Rainbow attack	

Tabella 1.4: *Attacchi nella storia - Testo cifrato noto*

Attacco di Golic [5]

Golic nel 1997 riuscì a ridurre la complessità dell'attacco a forza bruta da 2^{54} fino a $2^{40.16}$ tramite il suo attacco "Divide et impera" basato sulla risoluzioni di sistemi di equazioni lineari. Questo attacco di tipo known-plaintext prova a indovinare il contenuto dei due LFSR più corti.

Golic ha inoltre dimostrato che non tutti i possibili 2^{64} stati iniziali possono essere raggiunti ma solamente $2^{62.32}$.

Attacco di Biham e Dunkelman [3]

Questo attacco ha una complessità iniziale di circa 2^{47} ma, utilizzando alcune ottimizzazioni, il tempo richiesto cala drasticamente. La versione migliorata dell'attacco impiega un tempo totale pari a $2^{39.91}$ ma richiede la conoscenza di $2^{20.8}$ bit di testo in chiaro e una fase pre-computazionale di 2^{38} al fine di generare 32GB di dati.

Nel Capitolo 4 viene illustrato il funzionamento dell'attacco base e nel Capitolo 5 viene mostrata e descritta la relativa implementazione da me elaborata.

Attacchi di Biryukov, Shamir e Wagner [6]

I ricercatori Biryukov, Shamir e Wagner hanno eseguito due diversi attacchi al cifrario A5/1 basandosi sul compromesso tempo-memoria e sugli studi di Golic. In entrambi i casi è necessaria una fase di pre-computazione con complessità pari a 2^{42} nel "Biased birthday attack" e 2^{48} nel "Random subgraph attack", per generare rispettivamente 146GB e 292GB di dati. Dopo la fase pre-computazionale questi attacchi permettono di ricostruire la chiave:

- Nel "Biased birthday attack" in 1 secondo, avendo a disposizione 2 minuti di testo in chiaro (≈ 215 *frame*).
- Nel "Random subgraph attack" in pochi minuti, avendo a disposizione solamente 2 secondi di testo (≈ 29 *frame*).

I due attacchi hanno una complessità che cresce in modo esponenziale con la lunghezza dei LFSR, quindi l'aumento della dimensione dei registri renderebbe questi attacchi non completabili in tempi reali.

Attacco di Ekdahl e Johannson [7]

Ekdahl e Johannson sono due ricercatori che sono riusciti a sviluppare un "Correlation attack" con complessità non dipendente dalla lunghezza dei registri ma unicamente dal numero di cicli che essi devono eseguire prima dello stato iniziale e prima di produrre i bit di output. Con soli 2/5 minuti di conversazione richiesti, l'attacco viene compiuto in pochi minuti.

Attacco di Barkan, Biham e Keller [5]

L'attacco, convertito sulla base di uno precedentemente effettuato a A5/2, richiede un fase di pre-computazione molto lunga.

Attacco di Barkan e Biham [8]

Questo attacco, unicamente teorico, riesce ad estrarre, da SACCH (Slow Associated Control Channel), 1500-2000 *frame* di testo in chiaro, dati 3-4 minuti di testo cifrato. La più alta probabilità di successo è del 91% rispetto il 5% della prima versione. Non necessita di una fase di pre-computazione.

Attacco delle università di Bochum e Kiel [9]

Questo attacco è stato effettuato tramite l'utilizzo di CAPACOBANA, un hardware specifico basato su FPGA che ha la potenzialità di decifrare cifrari come A5/1, A5/2, DES e altri sistemi basati su curve ellittiche. Il tempo medio impiegato è di circa 6 ore, avendo come caso peggiore 12 ore di computazione. Per eseguire l'attacco non è presente nessuna fase pre-computazionale.

Attacco di The Hackers Choice [10]

Nel 2008 "The Hackers Choice" ha creato un progetto per attaccare il cifrario A5/1 in pochi minuti. Il problema principale è la complessità della fase pre-computazionale durante la quale occorre generare una tabella di associazioni di circa 3TB. Ad oggi questa impresa non si è rivelata possibile dal gruppo o da altri ricercatori anche se non si è a conoscenza di aziende governative aventi a disposizione super computer con potere computazionale maggiore.

Attacco di Paget and Nohl [11]

L'attacco consiste nel creare delle rainbow table che mappano lo stato interno di A5/1 con la sua *key stream*. L'azione è molto veloce ma la fase pre-computazionale è molto dispendiosa, infatti occorre generare circa 2TB di dati.

Capitolo 2

A5/2

2.1 Introduzione

A5/2 è un cifrario a flusso sviluppato nel 1993 per proteggere le comunicazioni GSM nei paesi orientali. É una versione volutamente più debole dell'algoritmo A5/1, utilizzato in Europa e negli Stati Uniti, a causa di restrizioni all'esportazione. Nel 1999 viene scoperto il suo funzionamento insieme al cifrario A5/1 da Marc Briceno tramite "reverse engineering".

Il A5/2 ha una struttura molto simile ad A5/1. Entrambi i cifrari hanno 3 LFSR con le medesime lunghezze e polinomi caratteristici. I due differiscono unicamente per la presenza del registro R_4 in A5/2 e nella fase di inizializzazione per l'impostazione di alcuni bit e per l'ultimo ciclo, che viene effettuato 99 volte contro le 100 di A5/1.

Attualmente il cifrario A5/2 è ritenuto troppo debole; infatti, contrariamente a A5/1, non viene più utilizzato.

Quando un dispositivo mobile non supporta le nuove tecnologie, al posto di usufruire di A5/2 si ricorre all'impiego del cifrario A5/0 che non apporta modifiche al testo in chiaro durante la trasmissione.

2.2 Descrizione

Il cifrario A5/2 utilizza una chiave a 64 bit, un frame pubblico di 22 bit come il cifrario A5/1. Diversamente dal quest'ultimo, presenta 4 LFSR (*Linear Feedback Shift Registers*) di lunghezze diverse per generare il flusso chiave che serve per cifrare i dati.

2.2.1 Registri

I 4 registri a scorrimento sono indicizzati con LSB (*Least Significant Bit*) e presentano le seguenti caratteristiche.

Reg. LFSR	Lunghezza Reg. (bit)	Polinomio caratteristico	Bit calcolo XOR
R_1	19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	13, 16, 17, 18
R_2	22	$x^{22} + x^{21} + 1$	20, 21
R_3	23	$x^{23} + x^{22} + x^{21} + x^8 + 1$	7, 20, 21, 22
R_4	17	$x^{17} + x^{12} + 1$	11, 16

Tabella 2.1: *Informazioni dei registri a scorrimento a retroazione lineare*

Il registro R_4

Il registro R_4 viene utilizzato unicamente per decidere quando i registri R_1, R_2 e R_3 devono eseguire il *clock*. I bit $R_4[3]$, $R_4[7]$ e $R_4[10]$ vanno in input alla *clock unit* che stabilisce il bit di maggioranza in base al valore con frequenza maggiore. I primi 3 registri effettuano il *clock* unicamente se il bit di maggioranza corrisponde ad un bit specifico di R_4:

- R_1 *clock* if $R_4[10] == bitMaggioranza$
- R_2 *clock* if $R_4[3] == bitMaggioranza$
- R_3 *clock* if $R_4[7] == bitMaggioranza$

La tabella che segue mostra le possibili combinazione tra i bit del registro R_4 e il *clock* dei registri R_1, R_2 e R_3. Ogni registro ha quindi una probabilità di *clock* del 75% come in A5/1.

$R_4[3]$	$R_4[7]$	$R_4[10]$	Bit di Maggioranza	R_1	R_2	R_3
0	0	0	0	clock	clock	clock
0	0	1	0	still	clock	clock
0	1	0	0	clock	clock	still
0	1	1	1	clock	still	clock
1	0	0	0	clock	still	clock
1	0	1	1	clock	clock	still
1	1	0	1	still	clock	clock
1	1	1	1	clock	clock	clock

Tabella 2.2: *Regola di Maggioranza*

2.2.2 Ciclo dei Registri

All'interno dell'algoritmo ogni registro deve ripetere il *clock* molteplici volte. Il ciclo viene eseguito prima dai registri R_1, R_2 e R_3 e solo dopo il loro completamento dal registro R_4.

Il *clock* è composto da quattro parti:

- Calcolo del valore da assegnare al bit 0
- Calcolo bit di maggioranza del registro
- Shift
- Calcolo bit di output

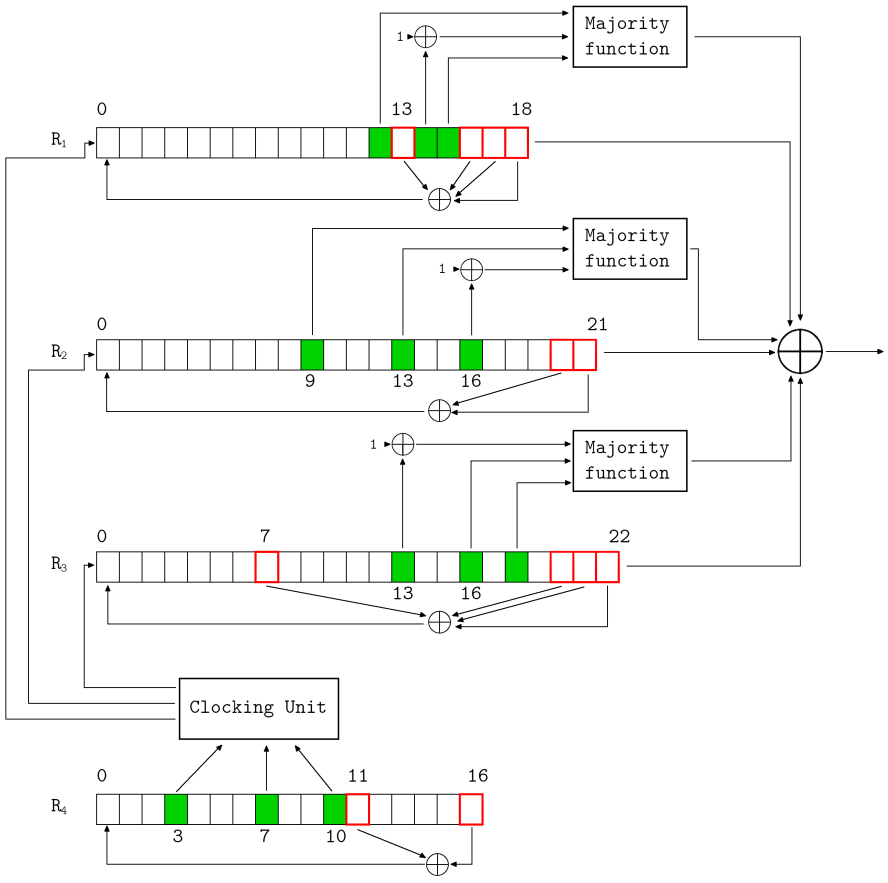


Figura 2.1: *Clock LFSR*
- Le caselle colorate in verde corrispondono ai bit utilizzati nelle funzione di maggioranza
- Le caselle contornate di rosso sono quelle utilizzate per le operazioni di XOR

Calcolo bit 0

Lo XOR viene eseguito su bit differenti in base al registro e al suo polinomio caratteristico.

- In R_1 vengono utilizzati i bit: 13, 16, 17 e 18
- In R_2 vengono utilizzati i bit: 20 e 21
- In R_3 vengono utilizzati i bit: 7, 20, 21 e 22
- In R_4 vengono utilizzati i bit: 11 e 16

In ogni registro il risultato dell'operazione viene scritto nel LSB dopo lo shift.

Calcolo bit di maggioranza

I registri R_1, R_2 e R_3 inviano 2 bit e il complementare di un altro bit ad una funzione di maggioranza. L'output di questa funzione verrà utilizzato per calcolare il bit di output del *clock*. Ogni registro dà in input alla funzione bit in posizioni diverse:

- $bitMaggioranzaR_1$: $R_1[12]$, $R_1[14]^C$, $R_1[15]$
- $bitMaggioranzaR_2$: $R_2[9]$, $R_2[13]$, $R_2[16]^C$
- $bitMaggioranzaR_3$: $R_3[13]^C$, $R_3[16]$, $R_3[18]$

Con $R_i[j]^C$ si indica il bit complementare di $R_i[j]$.

Shift

Ogni registro viene shiftato a destra di 1 bit. I bit più a destra vengono utilizzati per il calcolo del bit di output.

2.2.3 Calcolo bit di output

Il bit di output viene generato dall'operazione XOR dei bit usciti dalla fase di shift e dai bit di maggioranza dei singoli registri.

$$\begin{aligned} clockOutput = {} & bitShiftR_1 \oplus bitShiftR_2 \\ & \oplus bitShiftR_3 \oplus bitMaggioranzaR_1 \\ & \oplus bitMaggioranzaR_2 \oplus bitMaggioranzaR_3 \end{aligned}$$

2.2.4 Le fasi

Il cifrario A5/2 è diviso, come A5/1, in due macro fasi: la prima di inizializzazione e la seconda durante la quale vengono cifrati i dati inviati dal dispositivo mobile.

Inizializzazione

L'inizializzazione si suddivide in 5 fasi.

Prima fase

Nella prima fase vengono impostati tutti i bit dei registri R_1, R_2, R_3 e R_4 uguali a 0.

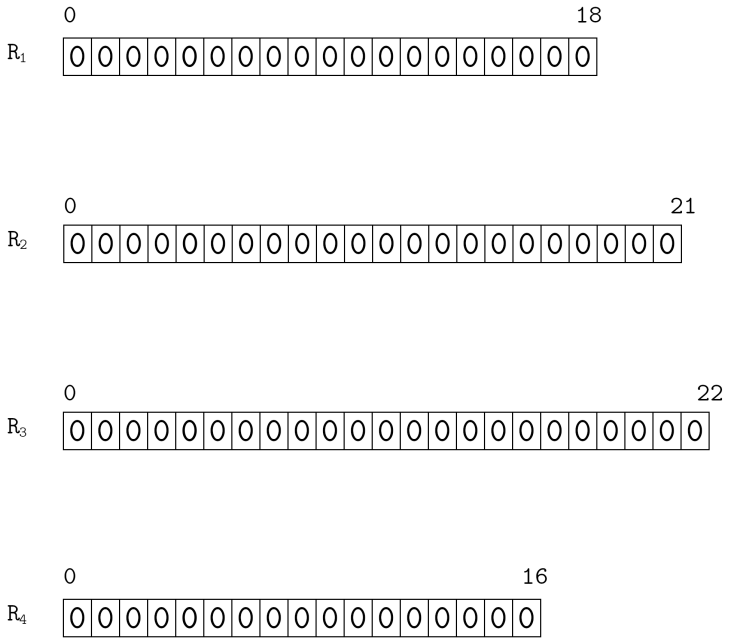


Figura 2.2: *Registri dopo la prima fase.*

Seconda fase

La seconda fase consiste nel combinare un bit della chiave segreta, tramite XOR, al bit meno significativo di ogni registro. Dopo queste operazioni i registri R_1, R_2, R_3 e R_4 eseguono un *clock* senza l'utilizzo della regola di *clock* di R_4 e scartando il bit di output prodotto. I passaggi descritti, compongono 1 *round* e vengono ripetuti per 64 volte. Ad ogni *round* il bit della chiave segreta utilizzato nei calcoli è quello nella posizione i-esima ($0 \leq i < 64$), dove i equivale al numero dell'iterazione.

for i := 0 **to** 63 **do**:

$R_1[0] = R_1[0] \oplus Key[i]$

$R_2[0] = R_2[0] \oplus Key[i]$

$R_3[0] = R_3[0] \oplus Key[i]$

$R_4[0] = R_4[0] \oplus Key[i]$

clock i 4 registri

Esempio del primo *round*

Le prime operazioni coinvolgono il bit meno significativo di ogni registro $R_1[0]$, $R_2[0]$, $R_3[0]$, $R_4[0]$ e il bit della chiave $Key[0]$ in quanto al primo *round* $i = 0$. Si ricorda che, dalla prima fase, i bit dei registri sono tutti uguali a 0. I calcoli eseguiti sono i seguenti:

$R_1[0] = R_1[0] \oplus Key[0] = 0 \oplus 1 = 1$

$R_2[0] = R_2[0] \oplus Key[0] = 0 \oplus 1 = 1$

$R_3[0] = R_3[0] \oplus Key[0] = 0 \oplus 1 = 1$

$R_4[0] = R_4[0] \oplus Key[0] = 0 \oplus 1 = 1$

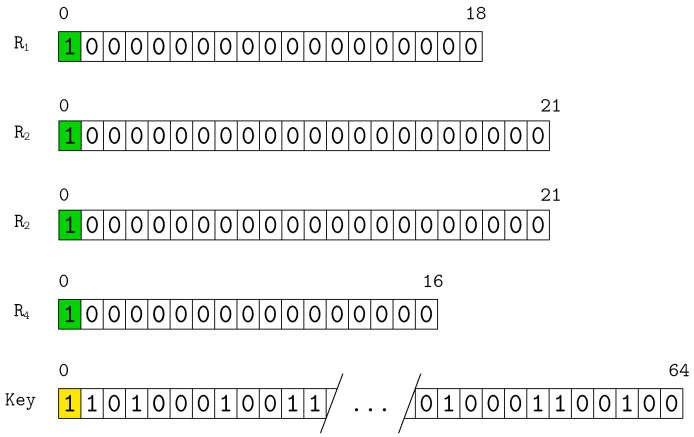


Figura 2.3: *Registri dopo l'operazione di XOR con la chiave.*
- Le caselle colorate in verde indicano i bit sovrascritti.
- La casella della chiave colorata in giallo indica l'i-esimo bit che viene utilizzato in questo round per le operazioni.

Dopo le operazioni con la chiave segreta i quattro registri devono eseguire il *clock*. Nell'immagine che segue sono evidenziati tutti i bit coinvolti nelle operazioni del calcolo del bit 0. Si ricorda che il bit meno significativo viene sovrascritto solamente dopo l'operazione di *shift* e l'output prodotto viene scartato (nell'esempio non viene mostrata la generazione del bit di output).

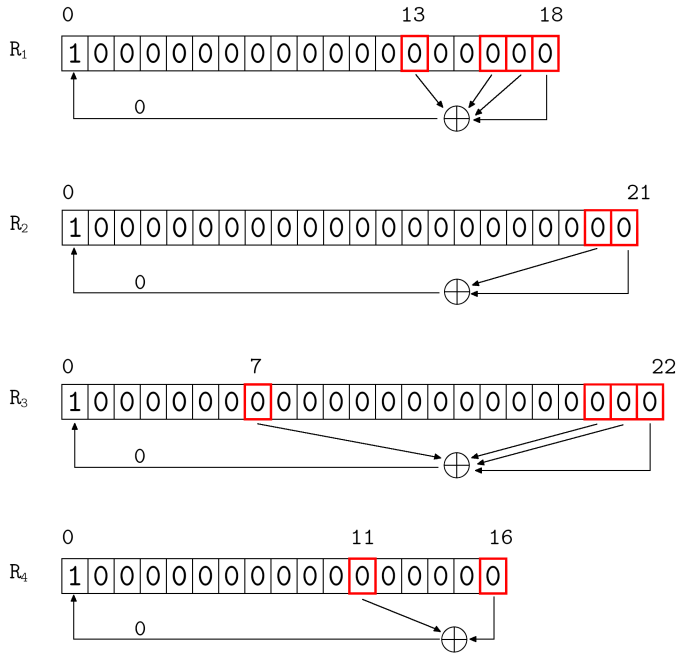


Figura 2.4: *Calcolo dei bit 0. I bit utilizzati per le operazioni sono quelli contornati in rosso.*

Nell'immagine seguente sono presenti i quattro registri dopo l'esecuzione del *clock*.

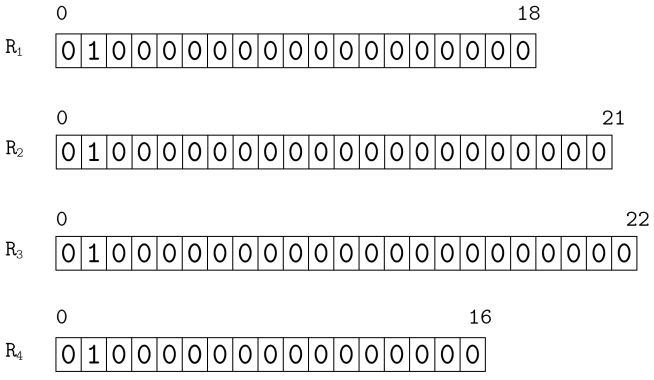


Figura 2.5: *Registri dopo l'esecuzione del clock, alla fine del primo round della seconda fase.*

A questo punto il *round* è finito e per completare la seconda fase occorre ripetere per 64 volte tutta la procedura, corrispondente ai passaggi illustrati nell'esempio.

Terza fase
La terza fase, molto simile alla seconda, consiste nel combinare un bit del frame pubblico, tramite XOR, al bit meno significativo di ogni registro. Dopo queste operazioni i registri R_1, R_2, R_3 e R_4 eseguono un *clock* senza l'utilizzo della regola di *clock* di R_4 e scartando il bit di output prodotto. I passaggi descritti, compongono 1 *round* e vengono ripetuti per 22 volte. Ad ogni *round* il bit del frame utilizzato nei calcoli è quello nella posizione i-esima ($0 \leq i < 22$), dove i equivale al numero dell'iterazione.

for i := 0 **to** 21 **do**:

$R_1[0] = R_1[0] \oplus Frame[i]$

$R_2[0] = R_2[0] \oplus Frame[i]$

$R_3[0] = R_3[0] \oplus Frame[i]$

$R_4[0] = R_4[0] \oplus Frame[i]$

clock i 4 registri

Esempio del primo *round*
Le prime operazioni coinvolgono il bit meno significativo di ogni registro $R_1[0]$, $R_2[0]$, $R_3[0]$, $R_4[0]$ e il bit del frame $Frame[0]$ in quanto al primo *round* $i = 0$. Si ricorda che i valori di partenza dei registri di questa fase sono stati computati nella fase precedente.

$R_1[0] = R_1[0] \oplus Frame[0] = 1 \oplus 1 = 0$

$R_2[0] = R_2[0] \oplus Frame[0] = 0 \oplus 1 = 1$

$R_3[0] = R_3[0] \oplus Frame[0] = 1 \oplus 1 = 0$

$R_4[0] = R_4[0] \oplus Frame[0] = 0 \oplus 1 = 1$

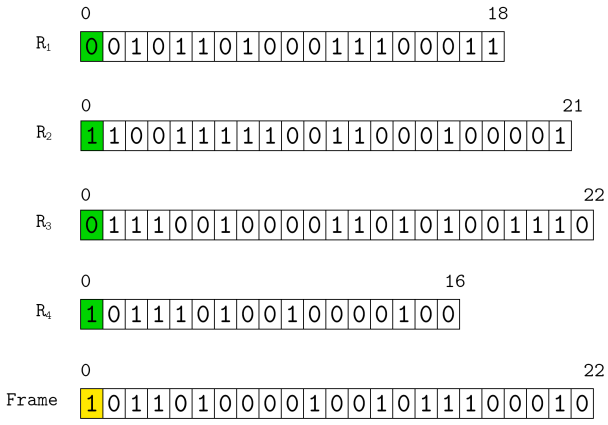


Figura 2.6: *Registri dopo l'operazione di XOR con il frame.*
- Le caselle colorate in verde indicano i bit sovrascritti.
- La casella della chiave colorata in giallo indica l'i-esimo bit che viene utilizzato in questo round per le operazioni.

Dopo le operazioni con il frame pubblico i quattro registri devono eseguire il *clock*. Nell'immagine che segue sono evidenziati tutti i bit coinvolti nelle operazioni del calcolo del bit 0. Si ricorda che il bit meno significativo viene sovrascritto solamente dopo l'operazione di *shift* e l'output prodotto viene scartato (nell'esempio non viene mostrata la generazione del bit di output).

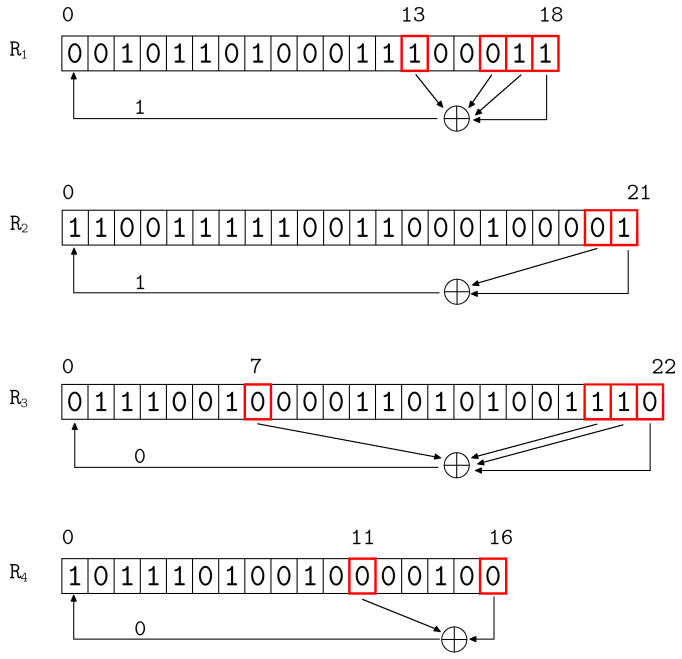


Figura 2.7: *Calcolo dei bit 0. I bit utilizzati per le operazioni sono quelli contornati in rosso.*

Nell'immagine seguente sono presenti i quattro registri dopo l'esecuzione del *clock*.

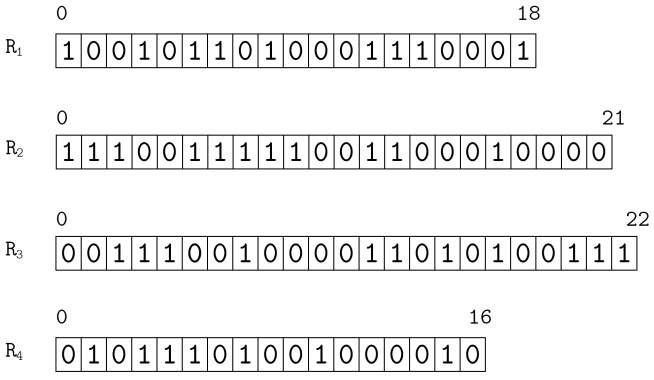


Figura 2.8: *Registri dopo l'esecuzione del clock, alla fine del primo round della terza fase.*

A questo punto il *round* è finito e per completare la terza fase occorre ripetere per 22 volte tutta la procedura, corrispondente ai passaggi illustrati nell'esempio.

Quarta fase
Nella quarta fase vengono impostati uguali ad 1 alcuni bit dei quattro registri.

$$R_1[15] = 1$$
$$R_2[16] = 1$$
$$R_3[18] = 1$$
$$R_4[10] = 1$$

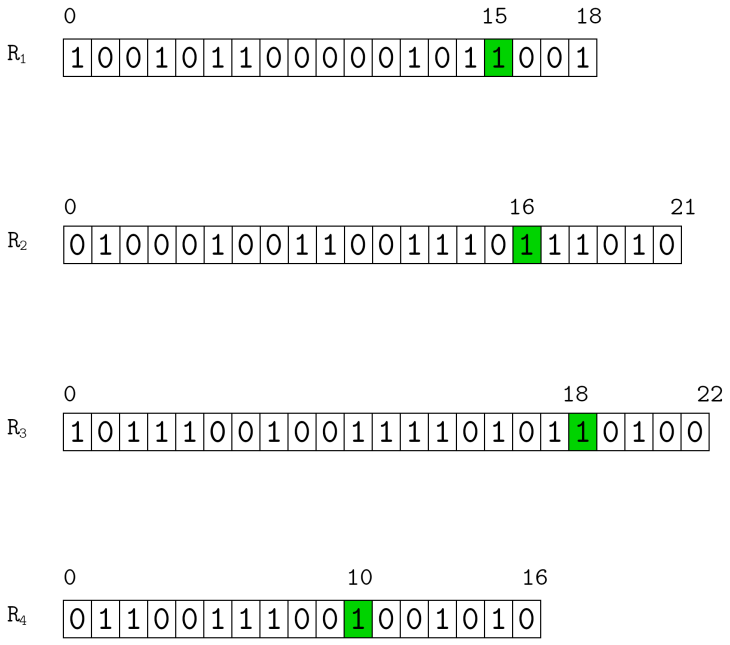


Figura 2.9: *Registri dopo la fase di assegnazione.*
- Le caselle colorate in verde sono quelle modificate.

Quinta fase
Nell'ultima fase i registri devono eseguire il *clock* mediante l'utilizzo della regola di *clock* del registro R_4, scartando l'output prodotto (nell'esempio non viene mostrata la generazione del bit di output). Questo passaggio, detto *round*, viene ripetuto per 99 volte.

for i := 0 **to** 98 **do**:
    clock secondo la regola di *clock* del registro R_4

Esempio del primo *round*
La prima operazione che viene effettuata è il calcolo del bit di maggioranza mediante il registro R_4, dal quale si può determinare quali registri devono eseguire il *clock*.

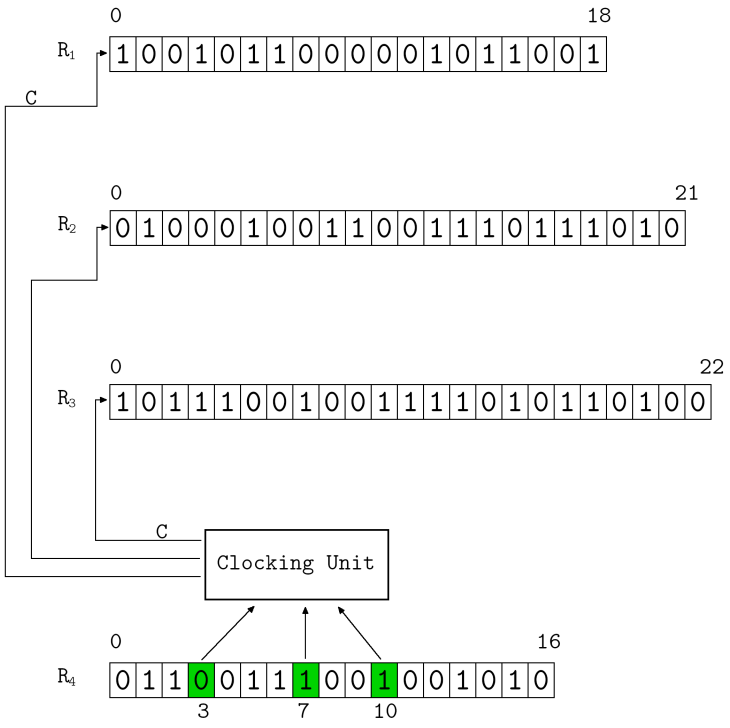


Figura 2.10: *Calcolo del bit di maggioranza.*
- Le caselle colorate in verde indicano quali bit sono stati utilizzati nell'operazione.

Il bit di maggioranza calcolato è uguale a 1, quindi solamente R_1, R_3 e R_4 eseguono il *clock*. Nell'immagine che segue sono evidenziati tutti i bit coinvolti nelle operazioni del calcolo del bit 0. Si ricorda che il bit meno significativo viene sovrascritto solamente dopo l'operazione di *shift*.

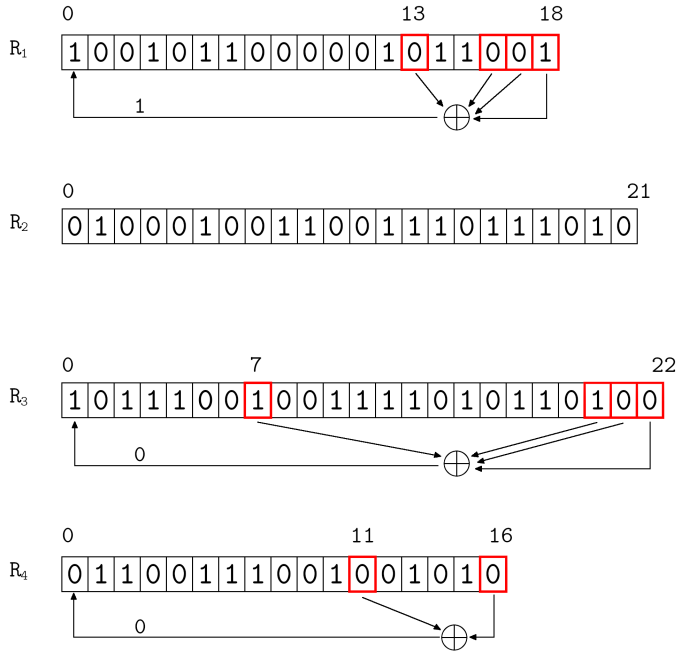


Figura 2.11: *Calcolo dei bit 0. I bit utilizzati per le operazioni sono quelli contornati in rosso.*

Nell'immagine seguente sono presenti i quattro registri dopo l'esecuzione del *clock*.

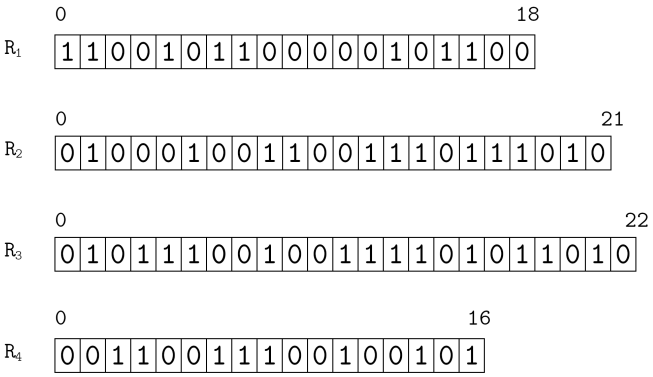


Figura 2.12: *Registri dopo l'esecuzione del clock, alla fine del primo round della quinta fase.*

A questo punto il *round* è finito e per completare la quinta fase occorre ripetere per 99 volte tutta la procedura, corrispondente ai passaggi illustrati nell'esempio.
Dopo le 99 iterazioni l'inizializzazione è terminata.

Cifratura

Dopo aver svolto l'inizializzazione i registri sono pronti per cifrare i dati. Vengono eseguiti ulteriori 228 *clock*, utilizzando la regola di *clock* del registro R_4. L'output prodotto da ogni *round* serve per produrre 1 bit della *key stream* di 228 bit, 114 bit servono per cifrare i dati dalla centrale fino al dispositivo mobile, mentre gli altri 114 bit vengono utilizzati per cifrare i dati dal telefono alla stazione, come in A5/1.

for i := 0 **to** 227 **do**:
    clock secondo la regola del registro R_4
    $keyStream[i] = clockOutput[i]$

114 bit del testo in chiaro vengono cifrati con i 114 bit della *key stream*.

$$keyStream \oplus plainText = cipherText$$

Per cifrare il *frame* successivo si ripetono le operazioni dal primo punto dell'inizializzazione utilizzando la stessa chiave segreta ma il frame viene incrementato di 1.

2.3 Attacchi

Analogamente al Capitolo 1.3 sono illustrati i principali attacchi effettuati nella storia al cifrario A5/2.

Anno	Autori	Tipologia	Richiede
1999	Goldberg, Wagner, e Green	Risoluzione sistema di equazioni lineari	Testo in chiario
2000	Slobodan Petrovic e Amparo Fuster-Sabater	Time-memory tradeoff	Testo cifrato
2003	Elad Barkan, Eli Biham e Nathan Keller	Error correction code	Testo cifrato
2003	Elad Barkan, Eli Biham e Nathan Keller	Man in the middle	

Tabella 2.3: *Attacchi nella storia*

Attacco di Goldberg, Wagner, e Green [13]

I ricercatori Goldberg, Wagner, e Green attaccarono A5/2 basandosi sulla risoluzione di un sistema di equazioni lineari. Per eseguire l'attacco è necessaria la conoscenza dello XOR dei testi in chiaro di 2 *frame*, distanti tra loro 2^{11} *frame*.

La complessità dell'attacco è di circa 2^{44} operazione bit-XOR: ci sono 2^{16} possibili stati del registro R_4 e sono necessari ulteriori 2^{28} XOR per risolvere il sistema binario lineare di 656 variabili ($656^3 \approx 2^{28}$). Esiste una versione ottimizzata dell'attacco con la quale la complessità è ridotta a 2^{28} operazioni bit-XOR ma necessita di uno stato pre-computazionale di 2^{46} calcoli bit-XOR con un spazio di memoria richiesto di $2^{27.8}$ byte ($\approx 233MB$).

Attacco di Petrovic e Amparo Sabater [14]

Trattasi di un attacco unicamente teorico.

Attacco di Barkan, Biham e Keller [13]

Nella connessione GSM viene utilizzato un sistema di correzione dell'errore prima della fase di encryption.

Questo attacco si concentra sul sistema di correzione dell'errore "Slow Associated Control Channel" (SACCH) ma è applicabile anche ad altri sistemi di correzione presenti nel GSM (come SDCCH). In SACCH il messaggio ha una dimensione totale di 456 bit e viene diviso in 4 *frame* che vengono cifrati ed inviati.

La complessità dell'attacco è di 2^{16} possibili stati, con un utilizzo della memoria di $2^{28.8}$ byte ($\approx 476MB$) e una fase di pre-computazione con complessità di 2^{47} operazioni bit-XOR.

Un altro attacco di questo tipo è stato effettuato su "Stand Alone Dedicated Control Channel" (SDCCH). L'hardware-only attack recupera immediatamente lo stato segreto iniziale di A5/2. Necessita 16 *frame* (solo 8 per SDCCH/8 canali) casuali di crittotesto e riesce a completare l'attacco in circa 1 secondo, senza l'utilizzo di uno stato di pre-computazione. L'attacco mira a indovinare lo stato di R_4 che richiede uno sforzo computazionale di 2^{16}. Questo è sufficiente per decriptare tutti i *frame* di una singola sessione, ma si potrebbe comunque trovare la chiave di sessione invertendo la fase di inizializzazione.

Man in the middle [15]

Questo attacco forza il telefono della vittima ad utilizzare A5/2, se supportato, o A5/0 che non apporta modifiche al testo in chiaro, violando quindi anche i cifrari più complessi A5/1 e A5/3.

Capitolo 3

A5/3

3.1 Introduzione

Introdotto nel 2002 come terza generazione di GSM, A5/3, come il suo predecessore A5/1, è un cifrario a flusso implementato per proteggere le connessioni mobile. A5/3 come tutti i cifrari A5 usufruisce di una chiave di 64 bit, di un frame di 22 bit ed è diviso in 2 fasi. La prima di inizializzazione della *key stream* e la seconda nella quale il testo in chiaro viene cifrato tramite XOR alla *key stream* creata precedentemente. La *key stream*, utilizzata dall'algoritmo, viene generata dalla funzione "KGCORE" che presenta al suo interno il cifrario a blocchi "Kasumi". Quest'ultimo è una versione semplificata e più veloce dell'algoritmo MISTY al fine di rendere il cifrario eseguibile dai dispositivi mobile.

3.2 Funzione KGCORE

La "Core Keystream Generator Function" (KGCORE) è una funzione che prende in input, oltre alla chiave e al frame, i parametri nella tabella che segue e restituisce come output un segmento di 228 bit.

Parametro	Valore
K_c	Chiave A5/3(64 bis)
COUNT	Frame (22 bit)
CO	Output stream
BLKCNT	Contatore (64 bit)
CA	0000 1111
CB	0000 0
CC	0000 0000 00 $\|\|$ COUNT
CD	0
CE	0000 0000 0000 0000
CK	$K_c \,\|\|\, K_c$ (Chiave 128 bit)
KM	Costante utilizzata per modificare la CK(128 bit)

Tabella 3.1: *Input e parametri di KGCORE*

Il cifrario a blocchi Kasumi, presente 5 volte all'interno della funzione, prende in input il valore di CK e un altro valore come rappresentato nell'immagine seguente.

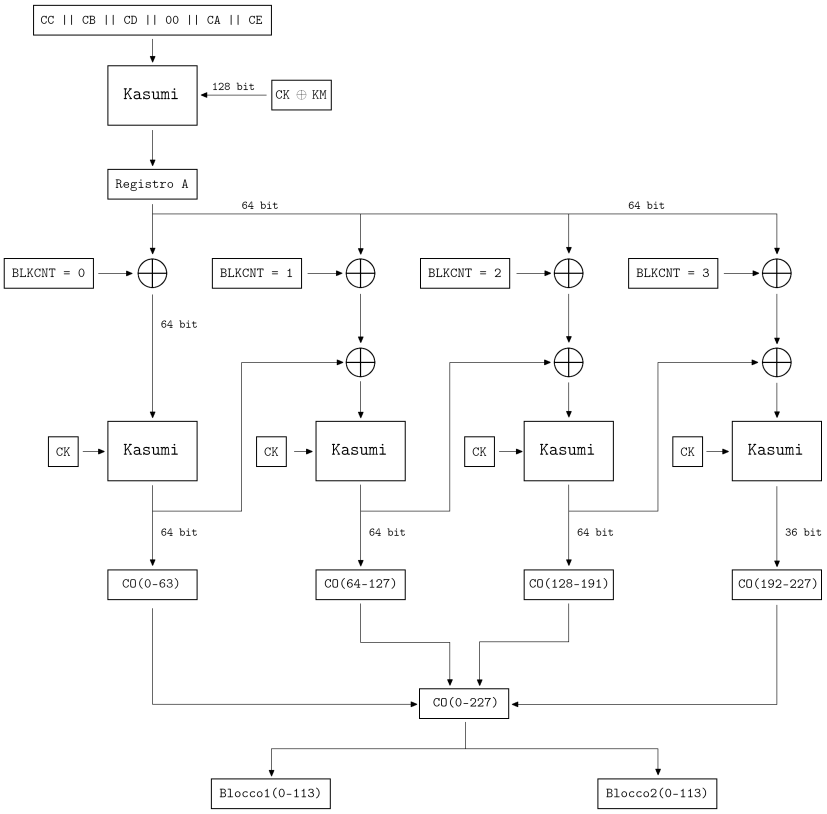


Figura 3.1: *Funzione KGCORE*

3.3 Kasumi

Kasumi è un cifrario Feistel che lavora su blocchi di 64 bit e accetta una chiave di 128 bit. Nell'algoritmo sono presenti 8 *round* nei quali vengono utilizzate due funzioni: FO che è una funzione Feistel a 3 *round* e FL che è una funzione a computazione lineare. L'ordine con le quali vengono chiamate cambia in base al ciclo. Nei *round* pari viene eseguita prima FO e poi FL mentre accade il contrario nei *round* dispari.

La chiave di 128 bit viene divisa in 8 parti di 16 bit l'una:

$$CK = K_1||K_2||K_3||K_4||K_5||K_6||K_7||K_8$$

La combinazione di K_i forma le chiavi:

- KL $(KL_{i,1}\|KL_{i,2})$
- KO $(KO_{i,1}\|KO_{i,2}\|KO_{i,3})$
- KI $(KI_{i,1}\|KI_{i,2}\|KI_{i,3})$

utilizzate nelle funzioni all'interno di Kasumi.

Round	$KL_{i,1}$	$KL_{i,2}$	$KO_{i,1}$	$KO_{i,2}$	$KO_{i,3}$	$KI_{i,1}$	$KI_{i,2}$	$KI_{i,3}$
1	$K_1 \lll 1$	K'_3	$K_2 \lll 5$	$K_6 \lll 8$	$K_7 \lll 13$	K'_5	K'_4	K'_8
2	$K_2 \lll 1$	K'_4	$K_3 \lll 5$	$K_7 \lll 8$	$K_8 \lll 13$	K'_6	K'_5	K'_1
3	$K_3 \lll 1$	K'_5	$K_4 \lll 5$	$K_8 \lll 8$	$K_1 \lll 13$	K'_7	K'_6	K'_2
4	$K_4 \lll 1$	K'_6	$K_5 \lll 5$	$K_1 \lll 8$	$K_2 \lll 13$	K'_8	K'_7	K'_3
5	$K_5 \lll 1$	K'_7	$K_6 \lll 5$	$K_2 \lll 8$	$K_3 \lll 13$	K'_1	K'_8	K'_4
6	$K_6 \lll 1$	K'_8	$K_7 \lll 5$	$K_3 \lll 8$	$K_4 \lll 13$	K'_2	K'_1	K'_5
7	$K_7 \lll 1$	K'_1	$K_8 \lll 5$	$K_4 \lll 8$	$K_5 \lll 13$	K'_3	K'_2	K'_6
8	$K_8 \lll 1$	K'_2	$K_1 \lll 5$	$K_5 \lll 8$	$K_6 \lll 13$	K'_4	K'_3	K'_7

Tabella 3.2: *Computazione della chiave*
- $K_i \lll j$ equivale alla rotazione a sinistra della parte K_i di j bit
- K'_i è uguale a K_i della chiave modificata da operazione di XOR con una costante

L'input di 64 bit viene diviso in 2 *stream* di 32 bit L0 e R0. Ad ogni *round* L_i e R_i avranno dei valori differenti secondo la regola seguente:

$$L_i = F_i(KL_i, KO_i, KI_i, L_{i-1}) \oplus R_{i-1}$$

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

Dove F_i varia se il *round* è pari o dispari:

$$Dispari \rightarrow F_i = FO(FL(L_{i-1}, KL_i), KO_i, KI_i)$$

$$Pari \rightarrow F_i = FL(FO(L_{i-1}, KO_i, KI_i), KL_i)$$

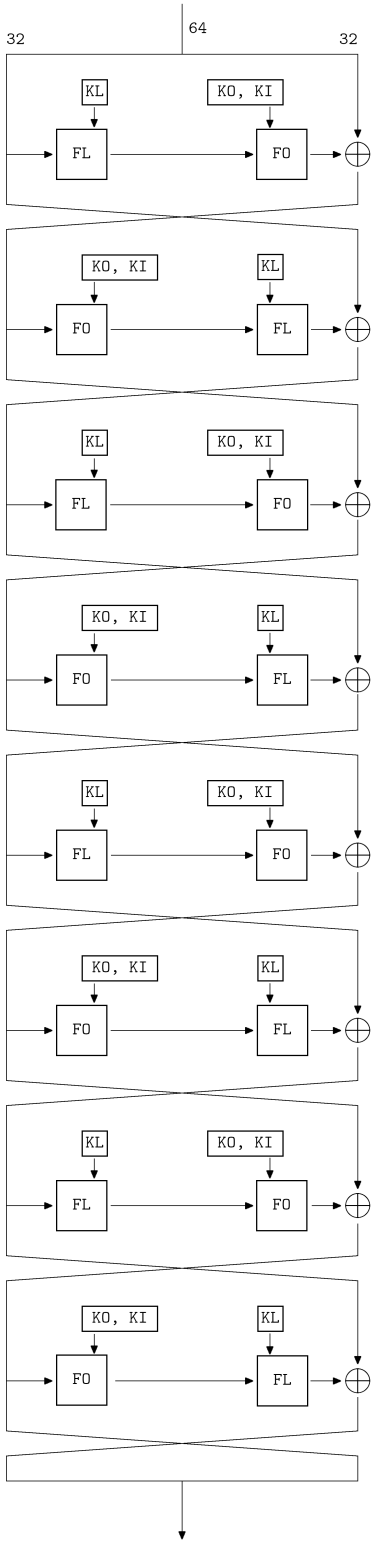


Figura 3.2: *Kasumi*

3.3.1 FL

FL è una funzione lineare molto semplice che prende in input 32 bit di dati e 32 bit di chiave KL. Entrambi gli input vengono divisi in due parti da 16 bit e utilizzati come nella figura sottostante.

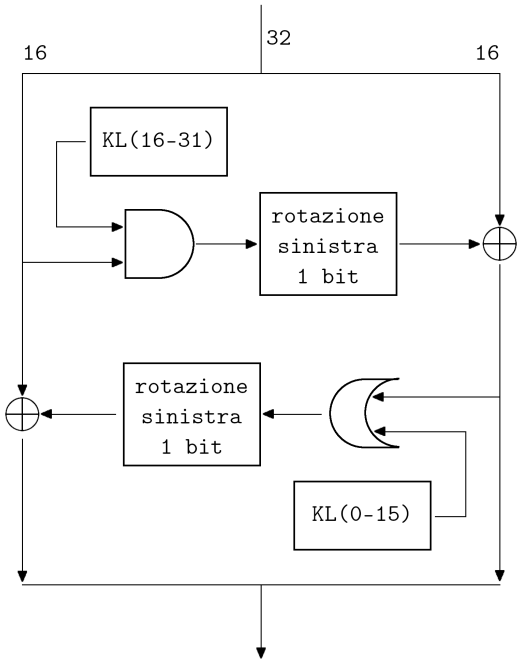


Figura 3.3: *FL*

3.3.2 FO

FO è una funzione Feistel a 3 *round* che prende in input 32 bit di dati e 96 bit di chiavi: KO 48 bit e KI 48 bit. I dati in input sono divisi in 2 parti di 16 bit l e r. Ad ogni *round* l viene modificato, utilizzando la funzione FI e operazioni di XOR, per generare la nuova r seconda la regola:

$$r_j = FI(KI, l_{j-1} \oplus KO_{i,j}) \oplus r_{j-1}$$

$$l_j = r_{j-1}$$

L'algoritmo di FO viene mostrato nella figura sotto riportata.

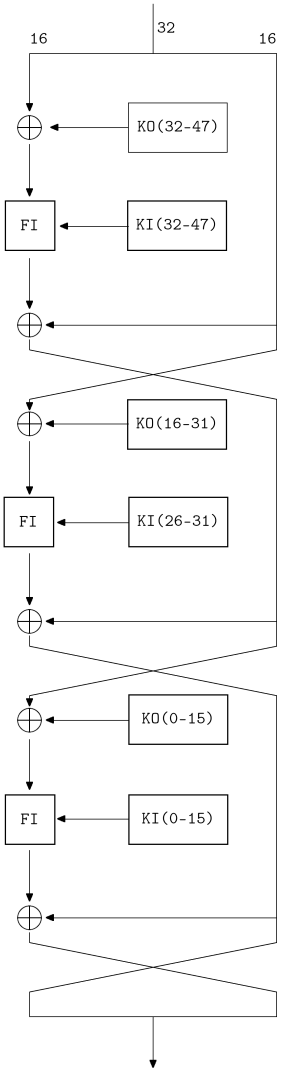


Figura 3.4: *FO*

3.3.3 FI

FI è una funzione che prende in input 16 bit di dati e 48 bit di chiave KI. La funzione che divide i dati in 2 parti (7 bit e 9 bit), è composta da 4 *round* e utilizza 2 S-box non lineari: S7 (S-box a 7 bit) nei *round* pari e S9 (S-box a 9 bit) nei *round* dispari. Essendo i dati divisi in 2 parti di dimensioni diverse, per completare tutte le operazioni sono necessarie due operazioni di aggiustamento:

$$Estensione0 = 00 \| r_j$$

$$Troncamento = LeastSignificant7(r_j)$$

L'algoritmo di FI viene mostrato nella figura sotto riportata.

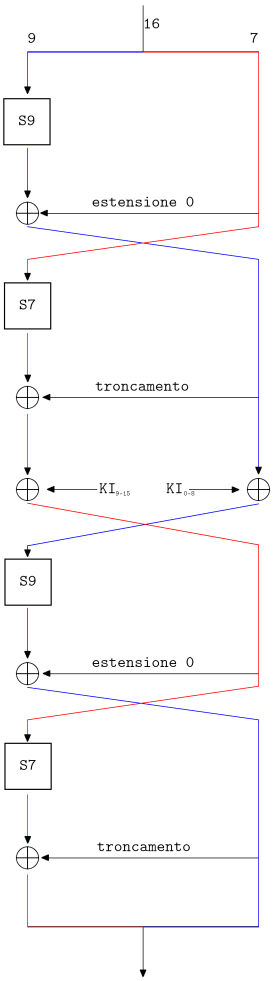


Figura 3.5: *FI*
- Segmento rosso = segmento a 7 bit
- Segmento blu = segmento a 9 bit

3.3.4 S-Box

S7 = {

```
  54,  50,  62,  56,  22,  34,  94,  96,  38,   6,  63,  93,   2,  18, 123,  33,
  55, 113,  39, 114,  21,  67,  65,  12,  47,  73,  46,  27,  25, 111, 124,  81,
  53,   9, 121,  79,  52,  60,  58,  48, 101, 127,  40, 120, 104,  70,  71,  43,
  20, 122,  72,  61,  23, 109,  13, 100,  77,   1,  16,   7,  82,  10, 105,  98,
 117, 116,  76,  11,  89, 106,   0, 125, 118,  99,  86,  69,  30,  57, 126,  87,
 112,  51,  17,   5,  95,  14,  90,  84,  91,   8,  35, 103,  32,  97,  28,  66,
 102,  31,  26,  45,  75,   4,  85,  92,  37,  74,  80,  49,  68,  29, 115,  44,
  64, 107, 108,  24, 110,  83,  36,  78,  42,  19,  15,  41,  88, 119,  59,   3
```

}

S9 = {

```
 167, 239, 161, 379, 391, 334,   9, 338,  38, 226,  48, 358, 452, 385,  90, 397,
 183, 253, 147, 331, 415, 340,  51, 362, 306, 500, 262,  82, 216, 159, 356, 177,
 175, 241, 489,  37, 206,  17,   0, 333,  44, 254, 378,  58, 143, 220,  81, 400,
  95,   3, 315, 245,  54, 235, 218, 405, 472, 264, 172, 494, 371, 290, 399,  76,
 165, 197, 395, 121, 257, 480, 423, 212, 240,  28, 462, 176, 406, 507, 288, 223,
 501, 407, 249, 265,  89, 186, 221, 428, 164,  74, 440, 196, 458, 421, 350, 163,
 232, 158, 134, 354,  13, 250, 491, 142, 191,  69, 193, 425, 152, 227, 366, 135,
 344, 300, 276, 242, 437, 320, 113, 278,  11, 243,  87, 317,  36,  93, 496,  27,
 487, 446, 482,  41,  68, 156, 457, 131, 326, 403, 339,  20,  39, 115, 442, 124,
 475, 384, 508,  53, 112, 170, 479, 151, 126, 169,  73, 268, 279, 321, 168, 364,
 363, 292,  46, 499, 393, 327, 324,  24, 456, 267, 157, 460, 488, 426, 309, 229,
 439, 506, 208, 271, 349, 401, 434, 236,  16, 209, 359,  52,  56, 120, 199, 277,
 465, 416, 252, 287, 246,   6,  83, 305, 420, 345, 153, 502,  65,  61, 244, 282,
 173, 222, 418,  67, 386, 368, 261, 101, 476, 291, 195, 430,  49,  79, 166, 330,
 280, 383, 373, 128, 382, 408, 155, 495, 367, 388, 274, 107, 459, 417,  62, 454,
 132, 225, 203, 316, 234,  14, 301,  91, 503, 286, 424, 211, 347, 307, 140, 374,
  35, 103, 125, 427,  19, 214, 453, 146, 498, 314, 444, 230, 256, 329, 198, 285,
  50, 116,  78, 410,  10, 205, 510, 171, 231,  45, 139, 467,  29,  86, 505,  32,
  72,  26, 342, 150, 313, 490, 431, 238, 411, 325, 149, 473,  40, 119, 174, 355,
 185, 233, 389,  71, 448, 273, 372,  55, 110, 178, 322,  12, 469, 392, 369, 190,
   1, 109, 375, 137, 181,  88,  75, 308, 260, 484,  98, 272, 370, 275, 412, 111,
 336, 318,   4, 504, 492, 259, 304,  77, 337, 435,  21, 357, 303, 332, 483,  18,
  47,  85,  25, 497, 474, 289, 100, 269, 296, 478, 270, 106,  31, 104, 433,  84,
 414, 486, 394,  96,  99, 154, 511, 148, 413, 361, 409, 255, 162, 215, 302, 201,
 266, 351, 343, 144, 441, 365, 108, 298, 251,  34, 182, 509, 138, 210, 335, 133,
 311, 352, 328, 141, 396, 346, 123, 319, 450, 281, 429, 228, 443, 481,  92, 404,
 485, 422, 248, 297,  23, 213, 130, 466,  22, 217, 283,  70, 294, 360, 419, 127,
 312, 377,   7, 468, 194,   2, 117, 295, 463, 258, 224, 447, 247, 187,  80, 398,
 284, 353, 105, 390, 299, 471, 470, 184,  57, 200, 348,  63, 204, 188,  33, 451,
  97,  30, 310, 219,  94, 160, 129, 493,  64, 179, 263, 102, 189, 207, 114, 402,
 438, 477, 387, 122, 192,  42, 381,   5, 145, 118, 180, 449, 293, 323, 136, 380,
  43,  66,  60, 455, 341, 445, 202, 432,   8, 237,  15, 376, 436, 464,  59, 461
```

}

3.4 Attacchi

Analogamente agli altri Capitolo sono illustrati i principali attacchi effettuati nella storia ai cifrari A5/3 e KASUMI.

Anno	Autori	Tipologia	Altro
2001	Kühn	Impossible differential attack	KASUMI 6 round
2001	Blunden	Related-key differential attack	KASUMI 6 round
2005	Eli Biham, Orr Dunkelman e Nathan Keller	Related-key rectangle (boomerang) attack	KASUMI 8 round
2010	Orr Dunkelman, Nathan Keller e Adi Shamir	Related-key attack (sandwich attack)	KASUMI 8 round
		Impossible differential attack	KASUMI primi 7 round
		Impossible differential attack	KASUMI ultimi 7 round
		Single-key attack	KASUMI 6 round
2019	Anand, Shanmugam e Santhoshini	Time-memory trafeoff - Rainbow attack	T5/3

Tabella 3.3: *Attacchi nella storia*

Attacco di Kühn - KASUMI 6 round [20]

Kühn è il primo ricercatore ad attaccare KASUMI. Questo attacco necessita 2^{55} dati di testo in chiaro e uno sforzo computazionale è di 2^{100}.

Attacco di Blunden - KASUMI 6 round [21]

Per eseguire questo attacco sono necessari 2^{19} dati di testo in chiaro e lo sforzo computazionale è di 2^{112}.

Attacco di Biham, Dunkelman e Keller - KASUMI 8 round [22]

Per eseguire questo attacco sono necessari $2^{54.6}$ dati di testo in chiaro e lo sforzo computazionale è di $2^{76.1}$.

Attacco di Dunkelman, Keller e Shamir - KASUMI 8 round [23]

L'attacco inizialmente era molto dispendioso con una complessità di 2^{76} ma, tramite l'utilizzo l'ottimizzazione delle procedure di calcolo, i ricercatori sono riusciti a diminuire la richiesta computazionale. La versione ottimizzata può calcolare una chiave completa di 128 bit usando 4 chiavi relative, la generazione di 2^{26} dati, 2^{30} byte di memoria e un tempo di 2^{32}. Il punto negativo dell'attacco è che richiede sia le chiavi correlate che il testo in chiaro scelto e questo non è applicabile nel modo in cui KASUMI viene utilizzato da A5/3.

Questo attacco si può effettuare su KASUMI ma non sul cifrario MISTY da cui è derivato.

Attacco a KASUMI primi 7 round [24]

Per eseguire questo attacco sono necessari 2^{62} dati di testo in chiaro e lo sforzo computazionale è di $2^{115.8}$.

Attacco a KASUMI ultimi 7 round [24]

Questo è l'attacco a KASUMI 7 *round* più efficiente. Per eseguire questo attacco sono necessari $2^{52.5}$ dati di testo in chiaro e lo sforzo computazionale è di $2^{114.3}$.

Attacco a KASUMI 6 round [25]

Per eseguire questo attacco sono necessari $2^{60.8}$ dati di testo in chiaro e lo sforzo computazionale è di $2^{65.4}$.

Attacco a T5/3 [26]

T5/3 è una versione semplificata del cifrario A5/3 che presenta le seguenti caratteristiche

	A5/3	T5/3
Input	64 bit	32 bit
Chiave	128 bit	64 bit
Output	2 blocchi da 114 bit	2 blocchi da 64 bit x2
Count	22 bit	11 bit
S-box	9 bit e 7 bit	5 bit e 3 bit

Tabella 3.4: *T5/3*

L'attacco ideato dai ricercatori Anand, Shanmugam e Santhoshini è basato sulle rainbow table. Lo studio analizza i differenti parametri, quali punto di distinzione, funzioni di riduzione e le collisioni. Le funzioni di riduzioni mappano dal crittotesto alla chiave nel keyspace e non impattano la complessità della generazione delle tabelle. Il punto di distinzione permette di ridurre il tempo di ricerca della chiave. Se il punto di distinzione è maggiore della lunghezza della catena ne consegue un aumento del tempo e delle collisioni per creare le rainbow table.

L'attacco si può effettuare in tempo reale ma al fine di generare le diverse tabelle necessita di una fase pre-computazionale:

- circa 30 giorni per generare le rainbow table
- circa 12 ore per generare le rainbow table con differenti reduction function

Per ridurre di un fattore 8 le tempistiche di quest'ultima fase è stato utilizzato un processore i7, 12 core. Questo processo si potrebbe ulteriormente migliorare tramite la programmazione CUDA fornita da GPU NVIDIA.

3.5 Attacchi nella Storia dei tre cifrari

Vengono presentati in una tabella riepilogativa i diversi attacchi nella storia effettuati ai cifrari descritti nei Capitoli precedenti.

Tabella 3.5: *Schema riepilogativo degli attacchi ai cifrari effettuati nella storia (pagina a fianco)*

Cifrario	Tipologia	Conoscere	Complessità	Problematiche
A5/1	Risoluzione sistema di equazioni lineari	Testo in chiaro	$2^{40.16}$	
A5/1	Divide et impera	Testo in chiaro	$2^{39.931}$	
A5/1	Time-memory tradeoff - Biased birthday attack	Testo in chiaro o solo cifrato	2^{42}	Fase di pre-computazione molto dispendiosa
A5/1	Time-memory tradeoff - Random subgraph attack	Testo in chiaro o solo cifrato	2^{48}	Fase di pre-computazione molto dispendiosa
A5/1	Correlation Attack	Testo in chiaro		
A5/1	Time-memory tradeoff	Testo cifrato		Generazione tabella di circa 3TB
A5/1	Rainbow attack	Testo cifrato		Generazione tabella di circa 2TB
A5/2	Risoluzione sistema di equazioni lineari	Testo in chiaro	2^{28}	Fase di pre-computazione dispendiosa (2^{46})
A5/2	Time-memory tradeoff	Testo cifrato		Attacco unicamente teorico
A5/2	Error correction code	Testo cifrato	2^{16}	Fase di pre-computazione dispondiosa (2^{47})
A5/2	Man in the middle	//	//	Attacco non più supportato
KASUMI	Impossible differential attack	Testo in chiaro	2^{100}	Conoscenza 2^{55} dati in chiaro
KASUMI	Related-key differential attack	Testo in chiaro	2^{112}	
KASUMI	Related-key attack rectangle (boomerang)	Testo in chiaro	$2^{76.1}$	Conoscenza $2^{54.6}$ dati in chiaro
KASUMI	Related-key attack (sandwich attack)	Testo in chiaro	2^{32}	
KASUMI	Impossible differential attack	Testo in chiaro	$2^{114.3}$	Conoscenza $2^{52.5}$ dati in chiaro
T5/3	Rainbow attack	Testo in chiaro		

Capitolo 4

Attacco a A5/1

Attacco di Biham e Dunkelman

In questo Capitolo viene descritto l'attacco base ideato dai due ricercatori Biham e Dunkelman, ritenuto il più efficiente tra quelli analizzati.

Il principio su cui si basa è quello di rimanere in attesa fino a quando non si verifica un evento che rivela molte informazioni. Al fine di poter trovare gli stati interni dei registri e quindi poter successivamente trovare la chiave è necessario conoscere l'*output stream* dell'algoritmo di A5/1.

É importante ricordare che, se consideriamo un utente, nell'algoritmo di A5/1 i primi due passaggi dell'inizializzazione hanno all'interno dei loro stati sempre gli stessi bit, in quanto all'inizio tutti i bit sono posti a 0 e successivamente subiscono un'operazione di XOR con la chiave segreta che non cambia mai per l'utente. Per questo non è necessario trovare la chiave segreta di cifratura ma è sufficiente scoprire la stato chiave, corrispondente allo stato dei 3 registri dopo lo XOR con la chiave a 64 bit.

4.1 Fase 1

La prima fase ci permette di ricavare 31 bit di informazioni riguardanti i registri.

Per fare ciò occorre assumere che per 10 *round* il registro R_3 non venga ciclato e si deve indovinare il bit di controllo del *clock* $R_3[10]$ e il bit di output $R_3[22]$.

Per i primi 10 *round*, dato che il registro R_3 non esegue il *clock*, si può ricavare dalla regola di maggioranza la seguente proprietà:

$$R_1[8] = R_2[10] \neq R_3[10]$$

Essendo $R_3[10]$ costante per 10 *round* si ottiene:

- $R_1[-1] \neq R_3[10] \Rightarrow 1$ bit di R_1
- $R_1[0-8] \neq R_3[10] \Rightarrow 9$ bit di R_1
- $R_2[1-10] \neq R_3[10] \Rightarrow 10$ bit di R_2

Da questa fase si riescono a calcolare ulteriori 11 bit di informazioni conoscendo l'*output stream* e il bit di output $R_3[22]$. L'*output stream*, come visto nel Capitolo 1, viene calcolata nel seguente modo:

$$outputBit = R_1[18] \oplus R_2[21] \oplus R_3[22]$$

e dato che si conosce $R_3[22]$ si può invertire la formula per ricavare 11 bit di informazioni riguardanti i registri R_1 e R_2:

$$outputBit \oplus R_3[22] = R_1[18] \oplus R_2[21]$$

Generalizzando sui 10 bit si può trovare la formula completa:

$$outputStream[0-10] \oplus R_3[22] = R_1[18-8] \oplus R_2[21-11]$$

Importante ricordare che in questo passaggio i bit dell'*output stream* vengono utilizzati da 0 a 10, mentre i bit dei registri R_1 e R_2 dal bit di output scendendo.

Questa operazione può essere svolta per 11 *round* anziché 10 in quanto anche se all'undicesimo *round* il registro R_3 dovesse eseguire il *clock*, per calcolare il bit di output verrebbe comunque utilizzato il medesimo $R_3[22]$.

Il costo della prima fase è di 2^2 in quanto devono essere indovinati 2 bit di R_3.

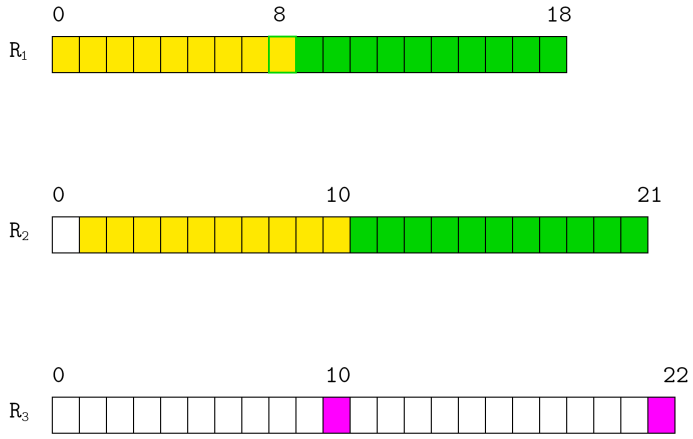


Figura 4.1: *Registri di A51 dopo la prima fase di attacco, stato iniziale*
- Le caselle colorate in magenta corrispondono ai bit che devono essere indovinati
- Le caselle colorate in giallo corrispondono ai bit trovati in seguito al clock
- Le caselle colorate in verde corrispondono ai bit che compongono gli 11 bit di informazioni calcolati da $outputBit \oplus R_3[22]$
- La casella $R_1[8]$ *rientra in entrambi i punti di calcolo con i colori verde e arancione*

4.2 Fase 2

Nella seconda fase l'obiettivo è quello di indovinare 9 bit di R_1 (18, 17, 16, 15, 14, 12, 11, 10, 9) e $R_2[0]$ al fine di poter calcolare tutti i bit dei registri R_1 e R_2.

Una volta trovati tutti i valori, occorre eseguire le operazioni seguenti. Nel registro R_1 il calcolo del bit 0 è dato dalla formula:

$$R_1 statoProssimo[0] = R_1[18] \oplus R_1[17] \oplus R_1[16] \oplus R_1[13]$$

In quanto dalla fase precedente sappiamo che $R_1 statoProssimo[0] \neq R_3[10]$, si hanno tutti i bit per poter invertire la formula e trovare $R_1[13]$:

$$R_1[13] = R_1[18] \oplus R_1[17] \oplus R_1[16] \oplus R_1 statoProssimo[0]$$

I bit di $R_2[11-21]$ vengono calcolati conoscendo i bit del primo registro e gli 11 bit trovati nella fase precedente ($outputStream[0-10] \oplus R_3[22] = R_1[18-8] \oplus R_2[21-11]$).

Si può continuare ad effettuare *clock* fino a quando R_3 non esegue un ciclo al fine di poter trovare il bit $R_3[21]$; ciò è possibile in quanto si conoscono completamente i primi due registri, l'*output stream* e il bit di controllo del *clock* $R_3[10]$.

Il costo della seconda fase è di 2^{10} perché devono essere indovinati 10 bit.

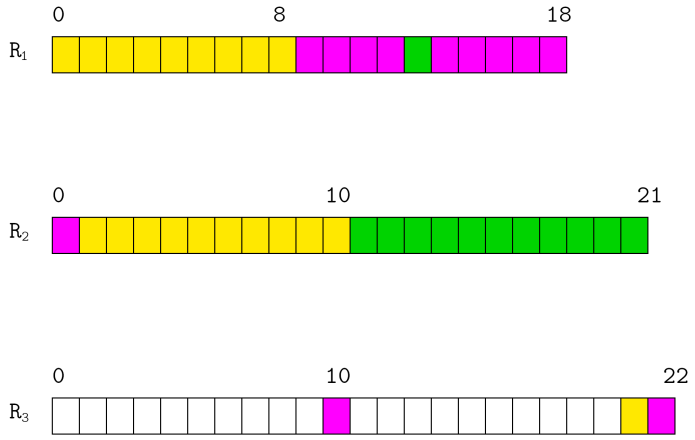


Figura 4.2: *Registri di A51 dopo la seconda fase di attacco, stato iniziale*
- Le caselle colorate in magenta corrispondono ai bit che devono essere indovinati
- Le caselle colorate in giallo corrispondono ai bit trovati in seguito al clock
- Le caselle colorate in verde corrispondono ai bit che sono stati calcolati durante la fase 2

4.3 Fase 3

Completate le prime due fasi si conoscono tutti i bit di R_1 e R_2 ma solo i bit $R_3[10]$, $R_3[21]$ e $R_3[22]$ del terzo registro.

A questo punto si devono indovinare i futuri 10 bit di controllo del *clock* $R_3[0-9]$.

Una volta terminata questa parte è sufficiente continuare ad effettuare *clock* per scoprire i rimanenti 11 bit di R_3 (dal 21 al 11), utilizzando la formula:

$$R_3[22] = R_1[18] \oplus R_2[21] \oplus OutpuBit$$

Occorre considerare che non bastano 11 *clock* per finire questa fase in quanto ogni registro viene ciclato in base alla regola della maggioranza. Dato che ogni registro ha una probabilità di *clock* pari a $\frac{3}{4}$, in media ci si aspetta servano 16 cicli di *clock* ($12 \div \frac{3}{4} = 16$) per trovare tutto R_3.

Così facendo si ha un possibile stato interno e bisogna verificarne la correttezza. Il costo ammortizzato per controllare se lo stato è corretto è uguale a 2 *clock* per ogni ipotesi.

$$2^{10} \cdot 2^4 \cdot 2 = 2^{15}$$

Il costo della fase 3 è di 2^{15}, in quanto devono essere indovinati 10 bit e occorre controllare la correttezza dello stato trovato.

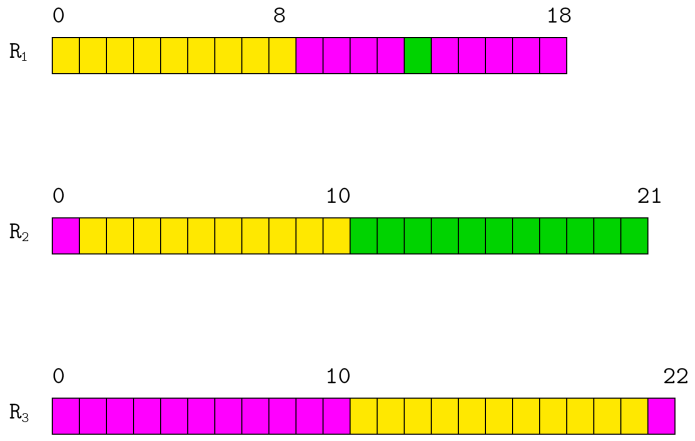


Figura 4.3: *Registri di A51 dopo la seconda fase di attacco, stato iniziale*
- Le caselle colorate in magenta corrispondono ai bit che devono essere indovinati
- Le caselle colorate in giallo corrispondono ai bit trovati in seguito al clock
- Le caselle colorate in verde corrispondono ai bit che sono stati calcolati durante la fase 2

4.4 Ritrovamento stato chiave

Grazie alle tre fasi precedenti si è riusciti ad avere un possibile stato interno che viene collocato nell'algoritmo di A5/1 subito dopo la fase di inizializzazione. Per risalire allo stato chiave, e quindi terminare l'attacco, bisogna effettuare due operazioni che implementano delle tecniche del ricercatore Golic [4]:

- 100 *reverse clock*
- *Reverse XOR frame*

Reverse clock

Il *reverse clock* è la funzione inversa rispetto al *clock*. Viene calcolato il nuovo MSB dipendentemente dal registro:

- $nuovoMSB\, R_1[18] = R_1[0] \oplus R_1[14] \oplus R_1[17] \oplus R_1[18]$
- $nuovoMSB\, R_2[21] = R_2[0] \oplus R_2[21]$
- $nuovoMSB\, R_3[22] = R_3[0] \oplus R_3[8] \oplus R_3[21] \oplus R_3[22]$

e successivamente i registri vengono shiftati a sinistra di 1 bit.

4.4.1 100 *reverse clock*

Questa fase consiste nel fare 100 *reverse clock* seguendo la regola di maggioranza.

for i := 99 **to** 0 **do**:
    reverse clock mediante l'utilizzo della regola di maggioranza

Regola di maggioranza in *reverse clock*

La regola di maggioranza in *reverse clock* è più complessa e dispendiosa rispetto alla regola normale in quanto occorre comprendere quali dei tre registri hanno eseguito un *clock* al *round* precedente e occorre controllare 2 bit per registro; in base a quanto emerso si devono seguire percorsi differenti. Utilizzano le tecniche di Golic [4].

Bit da controllare:

- R_1: 8, 9
- R_2: 10, 11
- R_3: 10, 11

Sia s_x il bit di maggioranza del registro $x = (R_1, R_2, R_3)$ mentre s'_x il bit successivo a s_x (esempio $x = R_1$ allora $s_x = R_1[8]$ e $s'_x = R_1[9]$) e $RC(x, y, z)$ il *reverse clock* dei registri (x, y, z). Siano (i, j, k) una permutazione dei registri R_1, R_2, R_3, quindi possono accadere sei eventi differenti:

- $\forall k$, se $s'_i = s'_j \neq s'_k = s_k$, allora $RC(i, j)$
- $\forall k$, se $s'_i = s'_j \neq s'_k \neq s_k$, allora $\nexists$ soluzione
- Se $s'_{R_1} = s'_{R_2} = s'_{R_3} = s_{R_1} = s_{R_2} = s_{R_3}$, allora $RC(R_1, R_2, R_3)$
- Se $s'_{R_1} = s'_{R_2} = s'_{R_3} \neq s_{R_1} = s_{R_2} = s_{R_3}$, allora $RC(R_1, R_2)$, $RC(R_1, R_3)$, $RC(R_2, R_3)$ e $RC(R_1, R_2, R_3)$
- $\forall k$, se $s'_{R_1} = s'_{R_2} = s'_{R_3} = s_i = s_j \neq s_k$, allora $RC(i, j)$ e $RC(R_1, R_2, R_3)$
- $\forall i$, se $s'_{R_1} = s'_{R_2} = s'_{R_3} = s_i \neq s_j = s_k$, allora $RC(i, j)$, $RC(i, k)$ e $RC(R_1, R_2, R_3)$

Dopo aver completato le operazioni di questa fase si possono ottenere una o più soluzioni dipendentemente dai casi trovati dalla regola di maggioranza in *reverse clock*, con un costo complessivo compreso tra $100\,clock \approx 2^{6.6}$ e $400\,clock \approx 2^{8.6}$, non rilevante per il calcolo delle complessità.

4.4.2 *Reverse* XOR frame

Questa fase è l'inversa rispetto alla terza sottofase dell'inizializzazione di A5/1 e occorre eseguire per 22 volte i passaggi seguenti. I 3 registri effettuano un *reverse clock* senza considerare la regola della maggioranza. Per calcolare il nuovo bit meno significativo di ogni registro viene effettuato uno XOR tra il bit 0 del registro e l'i-esimo bit $(22 > i \geq 0)$ del frame.

for i := 21 **to** 0 **do**:

reverse clock i 3 registri (senza l'utilizzo della regola di maggioranza)

$R_1[0] = R_1[0] \oplus Frame[i]$

$R_2[0] = R_2[0] \oplus Frame[i]$

$R_3[0] = R_3[0] \oplus Frame[i]$

Lo stato trovato corrisponde allo stato chiave, quindi l'attacco è terminato con successo.

Questo passaggio ha un costo costante di $22\,clock \approx 2^{4.4}$, non rilevante per il calcolo delle complessità.

4.5 Costo Totale

Il costo totale equivale a:

- Per la fase 1: 2^2
- Per la fase 2: 2^{10}
- Per la fase 3:
  - 2^{10} per indovinare i 10 bit
  - 2^4 numero stimato medio di *clock* per trovare i bit sconosciuti
- Costo ammortizzato per controllare la correttezza dello stato: 2 *clock*
  - tutti i casi devono fare almeno 1 *clock*
  - meta dei casi devono fare almeno 2 *clock*
  - un quarto dei casi devono fare almeno 3 *clock*
  - etc.
- Ritrovamento stato chiave: non rilevante

$$2^2 \cdot 2^{10} \cdot (2^{10} \cdot 2^4 \cdot 2) = 2^{27}$$

Il valore appena determinato equivale al costo totale solamente se si conosce l'esatta posizione in cui il registro R_3 non effettua il *clock* per 10 *round*. Questa condizione non è data a priori, quindi 2^{27} deve essere moltiplicato per il numero di posizioni iniziali necessari affinché si sia certi che l'evento richiesto si verifichi.

La probabilità che un registro non cicli è uguale a:

$$1 - \frac{3}{4} = \frac{1}{4}$$

Il requisito deve verificarsi per 10 *round*:

$$\left(\frac{1}{4}\right)^{10} = \frac{1}{2^{20}}$$

Per essere certi che l'evento si verifichi, è necessario esaminare al massimo 2^{20} differenti posizioni iniziali, considerando il maggior numero di *frame* possibili al fine di individuare il *frame* valido.

Considerando l'attacco complessivo, il costo totale aumenta notevolmente:

$$2^{27} \cdot 2^{20} = 2^{47}$$

Per questo motivo oltre che per l'hardware e il tempo a disposizione, si è deciso di implementare l'attacco, così come descritto nel Capitolo 5, considerando come ulteriore pre-condizione quella di avere un *frame* valido.

Capitolo 5

Attacco pratico a A5/1

L'attacco è stato scritto in linguaggio *C* in modo tale da avere delle performance elevate e quindi ridurre i tempi di computazione.

Tutto il codice descritto e illustrato in questo Capitolo può essere scaricato dalla mia pagina personale di GitHub al seguente link:

`https://github.com/itsraval/A51-attack`.

5.1 Struttura delle cartelle

Per rendere più agevoli eventuali miglioramenti e ottimizzare la visualizzazione del codice si è ritenuto opportuno suddividere il progetto in diversi file.

```
A51
├── compileBat
│   ├── compileA51.bat
│   ├── compileAttack.bat
│   ├── compileGenerateRealOutput.bat
│   └── compileRetrieveKey.bat
├── execution
├── headers
│   ├── A51LIB.h
│   ├── constant.h
│   ├── retrieveKey.h
│   └── treeState.h
├── output
├── textFiles
│   ├── A51.txt
│   └── keyRegisterState.txt
├── A51.c
├── attack.c
├── generateRealOutput.c
└── retriveKey.c
```

Nella cartella *compileBat* si possono trovare tutti i file batch che servono per eseguire il corrispettivo file.c, utilizzando la versione aggiornata degli altri file .c o .h necessari.

Nella cartella *execution* si possono trovare tutti i file .exe.

Nella cartella *headers* si possono trovare tutti i file .h e le rispettive implementazioni in file .c.

Nella cartella *output* si possono trovare tutti i file .o.

Nella cartella *textFiles* si possono trovare tutti i file che contengono gli output delle *printf* presenti nei file .c.

Nella cartella *A51* si possono trovare i 4 file principali.

- A51.c - algoritmo di A5/1
- attack.c - attacco ad A5/1
- generateRealOutput.c - utilizzato per testing e trovare l'*output stream* corretto
- retrieveKey.c - ultimi due passaggi dell'attacco

5.2 Il file di compilazione

Per avviare l'attacco è sufficiente eseguire il file *compileAttak.bat* presente nella cartella *compileBat*. Esso compila tutti gli headers e il sorgente main *attack.c*, quindi restituisce l'output nel file *keyRegisterState.txt* nella cartella *textFiles*.

Si è deciso di utilizzare un file .bat per l'esecuzione in modo tale che tutti i file necessari fossero aggiornati nel momento della compilazione dell'attacco e in modo da rendere più veloci ed efficienti le possibili modifiche e i testing.

```
gcc -w -c ..\headers\A51LIB.c -o ..\output\A51LIB.o ^
 && gcc -c ..\headers\treeState.c -o ..\output\treeState.o ^
 && gcc -c ..\retrieveKey.c -o ..\output\retrieveKey.o ^
 && gcc -c ..\attack.c -o ..\output\attack.o ^
 && gcc ..\output\attack.o ..\output\A51LIB.o ..\output\treeState.o
    ..\output\retrieveKey.o -o ..\execution\attack.exe ^
 && ..\execution\attack.exe > ..\textFiles\keyRegisterState.txt
```

5.3 Le costanti

In questo file sono presenti tutte le costanti riguardanti l'algoritmo di A5/1.

Le prime tre corrispondono alle dimensione dei 3 registri, seguite dalle posizioni dei bit per il calcolo della maggioranza; inoltre sono presenti anche le dimensioni di altri array.

```
#define R1LENGTH 19
#define R2LENGTH 22
#define R3LENGTH 23

#define R1MAJ 8
#define R2MAJ 10
#define R3MAJ 10

#define R1XORR2LENGTH 11

#define KEYLENGTH 64
#define FRAMELENGTH 22
#define OUTPUTLENGTH 114
```

5.4 Le funzioni di base di A5/1

In questa sezione vengono mostrate le funzioni di base utilizzate sia per implementare il cifrario A5/1, che l'attacco ad esso.

Calcolo bit complementare

```
int complementary(int x){
    // ritorna il valore complementare
    if(x == 0) return 1;
    else return 0;
}
```

Calcolo bit di maggioranza

La funzione seguente calcola il bit di maggioranza secondo la regola illustrata nel Capitolo 1.2.3.

```
int majorityBit(int r1[], int r2[], int r3[]){
    // calcolo bit di maggioranza
    if ((r1[R1MAJ] == r2[R2MAJ]) || (r1[R1MAJ] == r3[R3MAJ])){
        return r1[R1MAJ];
    } else{
        return r2[R2MAJ];
    }
}
```

Inizializzazione di un registro

L'array dato in input viene inizializzato con tutti i bit uguali a 0.

```
void initializeRegister (int r[], int length){
    // inizializza il registro  0
    for(int i=0; i<length; i++){
        r[i] = 0;
    }
}
```

Crea copia del registro

Questa funzione crea una copia, salvata in memoria, dell'array in input.

```
int* makeCopyRegister(int* r, int len){
    int* rc = malloc(len * sizeof(int));

    for(int i=0; i<len; i++){
        rc[i] = r[i];
    }
    return rc;
}
```

Stampa registro

```
void printRegister(int* r, int length){
    // stampa il registro
    printf("\nR");

    if(length == R1LENGTH){
        printf("1=[");
    }else if(length == R2LENGTH){
        printf("2=[");
    }else if(length == R3LENGTH){
        printf("3=[");
    }else{
        printf("=[");
    }

    for(int i=0; i<length-1; i++){
        printf("%d, ", r[i]);
    }
    printf("%d]", r[length-1]);
}
```

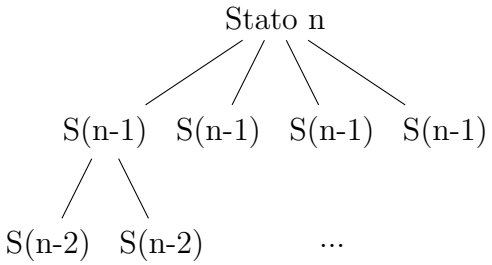
Clock del registro

La funzione seguente esegue il *clock* del registro dato in input, come illustrato nel Capitolo 1.2.2.

```
int clockRegister(int r[], int length){
    // clock registro
    int lastBit = r[length-1];
    int firstBit;

    // calcolo bit 0
    if (length == R1LENGTH){
        firstBit = r[13] ^ r[16] ^ r[17] ^ r[18];
    }else if (length == R2LENGTH){
        firstBit = r[20] ^ r[21];
    } else {
        firstBit = r[7] ^ r[20] ^ r[21] ^ r[22];
    }

    for (int i=length-1; i>0; i--){
        r[i] = r[i-1];
    }

    r[0] = firstBit;
    return lastBit;
}
```

5.5 L'albero degli stati

Per completare l'attacco e trovare lo stato chiave, il ricercatore Golic utilizza una struttura ad albero in modo da poter risalire allo stato precedente partendo da quello attuale. Si è ritenuto opportuno implementare la medesima struttura qui sotto rappresentata.



Struttura dello stato

Lo stato è composto dai 3 registri R_1, R_2, R_3 e da 4 puntatori.

```
#include <stdio.h>
#include <stdlib.h>
#include "constant.h"
#include "A51LIB.h"
#include "treeState.h"

typedef struct state{
    int* r1;
    int* r2;
    int* r3;
    struct state *s1;
    struct state *s2;
    struct state *s3;
    struct state *s4;
}state;
typedef struct state *State;
```

Creazione nuovo nodo

Quando viene creato un nuovo nodo, vengono inseriti i valori dei tre registri e i puntatori sono inizializzati a NULL in quanto il nuovo nodo sarà, in quel momento, una foglia dell'albero.

```
State newNode(int* r1, int* r2, int* r3){
    State node = malloc(sizeof(state));

    node->r1 = makeCopyRegister(r1, R1LENGTH);
    node->r2 = makeCopyRegister(r2, R2LENGTH);
    node->r3 = makeCopyRegister(r3, R3LENGTH);

    node->s1 = NULL;
    node->s2 = NULL;
    node->s3 = NULL;
    node->s4 = NULL;
    return node;
}
```

Stampa stato

```
void printState(State s){
    printRegister(s->r1, R1LENGTH);
    printRegister(s->r2, R2LENGTH);
    printRegister(s->r3, R3LENGTH);
    printf("\n%p - %p - %p - %p", s->s1, s->s2, s->s3, s->s4);
    printf("\n\n");
}
```

5.6 L'attacco

In questo Capitolo si espone la mia personale implementazione dell'attacco teorico ad A5/1 di Biham e Dunkelman.

Librerie utilizzate

Le librerie necessarie per il funzionamento dell'attacco sono mostrate nel codice sottostante. La libreria *"headers/retriveKey.h"* viene illustrata nel Capitolo 5.7 e contiene l'ultima parte dell'attacco, mentre le altre tre librerie sono state spiegate nei paragrafi precedenti.

```
#include <stdio.h>
#include <stdlib.h>
#include "headers/constant.h"
#include "headers/A51LIB.h"
#include "headers/treeState.h"
#include "headers/retriveKey.h"
```

Main

Nel main vengono salvati in memoria e inizializzati a 0 i tre registri R_1, R_2, R_3 e si creano gli array riguardanti l'*output stream* e il frame publlico di 22 bit. Infine si esegue la funzione attack, con la quale inizia la prima fase dell'attacco.

```
int main(){
    int* r1 = malloc(R1LENGTH * sizeof(int));
    int* r2 = malloc(R2LENGTH * sizeof(int));
    int* r3 = malloc(R3LENGTH * sizeof(int));

    int outputStream[OUTPUTLENGTH] = {1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1,
   1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1,
   0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1,
   0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1,
   1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1,
   0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1};

    int frame[FRAMELENGTH] = {1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0,
    1, 0, 0, 1, 1, 1, 0, 0};

    initializeRegister(r1, R1LENGTH);
    initializeRegister(r2, R2LENGTH);
    initializeRegister(r3, R3LENGTH);

    attack(r1, r2, r3, outputStream, frame);
}
```

5.6.1 Fase 1

Attack è la funzione che fa partire l'attacco. In essa si provano le 4 combinazioni dei bit $R_3[10]$ e $R_3[22]$, si completa la prima fase con la chiamata alla funzione *fase1* e si continua l'attacco con la funzione *fase2*.

```
void attack(int* r1, int* r2, int* r3, int* outputStream, int* frame){
    for(int r3Control=0; r3Control<2; r3Control++){
        for(int r3Output=0; r3Output<2; r3Output++){
            int* r1XORr2 = fase1(r1, r2, r3, outputStream, r3Control,
   r3Output);
            fase2(r1, r2, r3, r1XORr2, outputStream, frame);
        }
    }
}
```

Nella funzione seguente è implementa la fase 1, illustrata nel Capitolo 4.1. Come si può notare dopo l'assegnazione di $R_3[10]$ e $R_3[22]$ vengono calcolati 10 bit dei primi due registri e ulteriori 11 bit di informazioni riguardanti gli ultimi bit dei registri R_1 e R_2.

```
int* fase1(int* r1, int* r2, int* r3, int* outputStream, int r3Control,
    int r3Output){
    r3[10] = r3Control;
    r3[22] = r3Output;

    int compR3 = complementary(r3Control);
    for(int i=0; i<9; i++){
        r1[i] = compR3;
        r2[i+1] = compR3;
    }

    r2[10] = compR3;

    int* r1XORr2 = malloc(R1XORR2LENGTH * sizeof(int));
    initializeRegister(r1XORr2, R1XORR2LENGTH);

    for (int i=0; i<R1XORR2LENGTH; i++){
        r1XORr2[i] = r3[22] ^ outputStream[i];
    }
    return r1XORr2;
}
```

5.6.2 Fase 2

Nella seconda fase si assegna il valore al bit $R_2[11]$ e viene effettuata due volte una funzione ricorsiva, prima con *bit value* uguale a 0 e poi con valore pari 1.

```
1 void fase2(int* r1, int* r2, int* r3, int* r1XORr2, int* outputStream,
    int* frame){
2     r2[11] = r1[8] ^ r1XORr2[10];
3     fase2Recursive(r1, r2, r3, r1XORr2, outputStream, 18, 0, frame);
4     fase2Recursive(r1, r2, r3, r1XORr2, outputStream, 18, 1, frame);
5     return;
6 }
```

Nella funzione *fase2Recursive* vengono eseguite le operazioni illustrate nel Capitolo 4.2. La prima volta che viene chiamata ha in input $pos = 18$ perché si iniziano ad effettuare i calcoli partendo dal bit più significativo dei registri $R_1[18]$ e $R_2[18+3]$, scendendo fino alla posizione 9. Ad ogni chiamata un bit di R_1 e R_2 vengono assegnati (riga 3, 4).

Il caso base si ha quando la posizione è uguale a 9. Per entrambi i possibili valori di $R_2[0]$ vengono chiamate le funzioni $findR3\_21$, illustrata nel paragrafo successivo, seguita da *fase3*, che permette di proseguire l'attacco con la terza fase.

Se $pos > 9$ allora avvengono due chiamate ricorsive con $pos = pos - 1$; si utilizza nel primo caso $value = 0$, mentre nel secondo $value = 1$.

Se la posizione è uguale a 13, come prima operazione si decrementata la variabile *pos* di 1 in quanto i valori di $R_1[13]$ e $R_2[13+3]$ sono già stati calcolati al *round* con $pos = 16$ nel quale sono già presenti tutte le informazioni per computarli.

```
1 void fase2Recursive(int* r1, int* r2, int* r3, int* r1XORr2, int*
    outputStream, int pos, int value, int* frame){
2     if(pos==13) pos--;
3     r1[pos] = value;
4     r2[pos+3] = r1[pos] ^ r1XORr2[18-pos];
5
6     if(pos==9){
7         for(int r200=0; r200<2; r200++){
8             r2[0] = r200;
9             r3[21] = findR3_21(r1, r2, r3, outputStream);
10            fase3(r1, r2, r3, outputStream, frame);
11        }
12    }else{
13        if(pos==16){
14            r1[13] = r1[0] ^ r1[18] ^ r1[17] ^ r1[16];
15            r2[13+3] = r1[13] ^ r1XORr2[18-13];
16        }
17        fase2Recursive(r1, r2, r3, r1XORr2, outputStream, pos-1, 0,
    frame);
18        fase2Recursive(r1, r2, r3, r1XORr2, outputStream, pos-1, 1,
    frame);
19    }
20 }
```

Calcolo del bit $R_3[21]$

La funzione $findR3\_21$ ha il compito di trovare il valore di $R_3[21]$. Questo obiettivo viene raggiunto continuando a eseguire *clock* fino a quando il terzo registro non esegue un ciclo. Per calcolare il valore del bit si esegue una semplice operazione di XOR (riga 25). Per motivi legati alla ricorsione e all'utilizzo delle medesime variabili dei registri, tutte le operazione svolte in questa funzione vengono effettuate su copie dei registri originali in modo da non variarne il contenuto.

```
1  int findR3_21(int* r1, int* r2, int* r3, int* outputStream){
2      int* r1c = makeCopyRegister(r1, R1LENGTH);
3      int* r2c = makeCopyRegister(r2, R2LENGTH);
4      int* r3c = makeCopyRegister(r3, R3LENGTH);
5
6      int pos = 0;
7
8      while(1){
9          int majBit = majorityBit(r1c, r2c, r3c);
10         pos++;
11
12         if(majBit == r1c[R1MAJ]){
13             clockRegister(r1c, R1LENGTH);
14         }
15
16         if(majBit == r2c[R2MAJ]){
17             clockRegister(r2c, R2LENGTH);
18         }
19
20         if(majBit == r3c[R3MAJ]){
21             break;
22         }
23     }
24
25     int value = outputStream[pos] ^ r1c[18] ^ r2c[21];
26
27     free(r1c);
28     free(r2c);
29     free(r3c);
30     return value;
31 }
```

5.6.3 Fase 3

Nella terza fase viene eseguita due volte una funzione ricorsiva, prima con *bit value* uguale a 0 e poi con valore pari 1.

```
void fase3(int* r1, int* r2, int* r3, int* outputStream, int* frame){
    fase3Recursive(r1, r2, r3, outputStream, 9, 0, frame);
    fase3Recursive(r1, r2, r3, outputStream, 9, 1, frame);
}
```

Nella funzione *fase3Recursive* vengono effettuate le operazioni illustrate nel Capitolo 4.3. La prima volta che viene chiamata ha in input $pos = 9$ perché i bit che devono essere impostati in questa sottofase sono quelli con posizione da 0 a 9. Ad ogni chiamata viene impostato il bit di R_3 in posizione *pos* con valore *value*.

Il caso base si ha quando la posizione è uguale a 0. Si controlla la correttezza dello stato attuale dei registri tramite la funzione *correctState*. Se quest'ultima restituisce un valore affermativo allora la parte d'attacco studiata dai ricercatori Biham e Dunkelman è conclusa ed inizia la ricerca dello stato chiave, utilizzando le tecniche di Golic, con la funzione $reverse100Clock$.

Se $pos > 0$ allora avvengono due chiamate ricorsive con $pos = pos - 1$ e $value = 0$ nel primo caso e $value = 1$ nel secondo.

```
void fase3Recursive(int* r1, int* r2, int* r3, int* outputStream, int
    pos, int value, int* frame){
    r3[pos] = value;

    if(pos==0){
        int correctState = findR3_clock(r1, r2, r3, outputStream);

        if(correctState){
            State root = newNode(r1, r2, r3);
            reverse100Clock(root, frame);
        }
    }else{
        fase3Recursive(r1, r2, r3, outputStream, pos-1, 0, frame);
        fase3Recursive(r1, r2, r3, outputStream, pos-1, 1, frame);
    }
}
```

Calcolo dei bit rimanenti di R_3

In questa funzione vengono calcolati i bit non ancora individuati di R_3. Anche in questo caso, come nel *Calcolo del bit* $R_3[21]$, tutte le operazioni, ad eccezione della riga 25, vengono effettuate su copie dei registri originali in modo da non variarne il contenuto. Ad ogni *round*, se il terzo registro esegue un *clock*, viene impostato il valore del suo 21-esimo bit e viene ciclato, come da regola. Questo passaggio viene effettuato fino a quando R_3 non compie 12 *clock*.

```
1  int findR3_clock(int* r1, int* r2, int* r3, int* outputStream){
2      int* r1c = makeCopyRegister(r1, R1LENGTH);
3      int* r2c = makeCopyRegister(r2, R2LENGTH);
4      int* r3c = makeCopyRegister(r3, R3LENGTH);
5
6      int pos = 0;
7      int posR3 = 21;
8
9      while(1){
10         int majBit = majorityBit(r1c, r2c, r3c);
11         pos++;
12
13         if(majBit == r1c[R1MAJ]){
14             clockRegister(r1c, R1LENGTH);
15         }
16
17         if(majBit == r2c[R2MAJ]){
18             clockRegister(r2c, R2LENGTH);
19         }
20
21         if(majBit == r3c[R3MAJ]){
22             r3c[21] = outputStream[pos] ^ r1c[18] ^ r2c[21];
23             clockRegister(r3c, R3LENGTH);
24
25             r3[posR3] = outputStream[pos] ^ r1c[18] ^ r2c[21];
26
27             posR3--;
28             if(posR3==10){
29                 break;
30             }
31         }
32
33     }
34
35     int correctState = isCorrectState(r1c, r2c, r3c, outputStream,pos);
36
37     free(r1c);
38     free(r2c);
39     free(r3c);
40     return correctState;
41 }
```

Controllo correttezza dello stato

Questa funzione controlla che lo stato trovato, dato in input, sia corretto rispetto all'*output stream*. Restituisce 0 al primo errore trovato nello stato; se invece tutti i valori corrispondono all'*output stream*, lo stato è verificato e viene restituito 1.

```
int isCorrectState(int* r1, int* r2, int* r3, int* outputStream, int
    pos){
    int o1, o2, o3;

    while(pos<OUTPUTLENGTH){
        int majBit = majorityBit(r1, r2, r3);

        if(majBit == r1[R1MAJ]){
            o1 = clockRegister(r1, R1LENGTH);
        }else{
            o1 = r1[R1LENGTH-1];
        }

        if(majBit == r2[R2MAJ]){
            o2 = clockRegister(r2, R2LENGTH);
        }else{
            o2 = r2[R2LENGTH-1];
        }

        if(majBit == r3[R3MAJ]){
            o3 = clockRegister(r3, R3LENGTH);
        }else{
            o3 = r3[R3LENGTH-1];
        }

        if(outputStream[pos] != o1 ^ o2 ^ o3){
            return 0;
        }
        pos++;
    }

    return 1;
}
```

5.7 Il ritrovamento dello stato chiave

Librerie utilizzate

Le librerie necessarie per il ritrovamento dello stato chiave sono mostrate nel codice sottostante. La libreria *"headers/treeState.h"* è di fondamentale importanza al fine di riuscire a completare correttamente tutti i possibili *reverse clock*.

```
#include <stdio.h>
#include <stdlib.h>
#include "headers/constant.h"
#include "headers/A51LIB.h"
#include "headers/treeState.h"
```

Reverse clock

Il *reverse clock* è molto semplice e il suo comportamento è esposto nel Capitolo 4.4. Si calcola il nuovo MSB dipendentemente dal registro, si esegue lo shift a sinistra di 1 bit e si assegna il valore all'ultimo bit precedentemente calcolato.

```
void reverseClockRegister(int* r, int len){
    int lastBit;

    if(len == R1LENGTH){
        lastBit = r[0] ^ r[14] ^ r[17] ^ r[18];
    }else if(len == R2LENGTH){
        lastBit = r[0] ^ r[21];
    }else{
        lastBit = r[0] ^ r[8] ^ r[21] ^ r[22];
    }

    for(int i=0; i<len-1;i++){
        r[i] = r[i+1];
    }
    r[len-1] = lastBit;
}
```

5.7.1 100 *reverse clock*

Nelle funzioni seguenti vengono effettuate le operazione spiegate nel Capitolo 4.4.1. In *reverse*100*Clock* si esegue per 100 *round* il *reverse clock* considerando la regola di maggioranza.

```
void reverse100Clock(State s, int* frame){
    reverseClockMajBit(s, 100, frame);
}
```

Reverse calcolo bit di maggioranza

In questa funzione si calcola il bit di maggioranza in modo *reverse*.

```
1  int reverseMajBit(int r1[], int r2[], int r3[]){
2      // calcolo bit di maggioranza
3      if ((r1[R1MAJ+1] == r2[R2MAJ+1]) || (r1[R1MAJ+1] == r3[R3MAJ+1])){
4          return r1[R1MAJ+1];
5      } else{
6          return r2[R2MAJ+1];
7      }
8  }
```

La funzione seguente ha il compito di capire quali registri hanno i bit di maggioranza uguali al fine di fargli eseguire un *clock*. Viene utilizzata una sola volta alla riga 56 del $reverseClockMajBit$ nel caso in cui $s'_{R_1} = s'_{R_2} = s'_{R_3} = s_i = s_j \neq s_k$.

```
1  void clock2(int r1[], int r2[], int r3[]){
2      if(r1[R1MAJ]==r2[R2MAJ]){
3          reverseClockRegister(r1, R1LENGTH);
4          reverseClockRegister(r2, R2LENGTH);
5      }else if(r1[R1MAJ]==r3[R3MAJ]){
6          reverseClockRegister(r1, R1LENGTH);
7          reverseClockRegister(r3, R3LENGTH);
8      }else{
9          reverseClockRegister(r2, R2LENGTH);
10         reverseClockRegister(r3, R3LENGTH);
11     }
12 }
```

Questa funzione, utilizzata una sola volta alla riga 70 del $reverseClockMajBit$, ha come scopo quello di differenziare 3 casi che si possono verificare.

```
1  int clock3(int r1[], int r2[], int r3[]){
2      if(r1[R1MAJ]==r2[R2MAJ]){
3          return 3;
4      }else if(r1[R1MAJ]==r3[R3MAJ]){
5          return 2;
6      }else{
7          return 1;
8      }
9  }
```

Regola di maggioranza in *reverse clock*

La funzione ricorsiva $reverseClockMajBit$ ha l'obiettivo di eseguire il *reverse clock* dei registri R_1, R_2 ed R_3, seguendo la regola di maggioranza, come esposto nel Capitolo 4.4.1. La prima volta che viene chiamata ha in input $round = 100$; ad ogni ricorsione il $round$ viene decrementato di 1, fino ad arrivare al valore 0.

Il caso base si ha quando $round = 0$ che comporta il completamento dei *100 reverse clock* e l'inizio dell'ultima sottofase con la funzione $reverseXorFrame$.

Se $round > 0$ allora si calcola il $reverseMajBit$, si crea lo stato attuale dei registri e si eseguono i *clock* dipendentemente dalle regole illustrate nei Capitoli precedenti.

```
int reverseClockMajBit(State s, int round, int* frame){
    if(round == 0){
        reverseXorFrame(s, frame);
        printState(s);
        return 1;
    }

    int maj = reverseMajBit(s->r1, s->r2, s->r3);
    State node = newNode(s->r1, s->r2, s->r3);

    if((s->r1[R1MAJ+1] == maj) && (s->r2[R2MAJ+1] == maj) &&
    (s->r3[R3MAJ+1] != maj) && (s->r3[R3MAJ+1] == s->r3[R3MAJ])){
        // si' = sj' != sk' = sk
        reverseClockRegister(node->r1, R1LENGTH);
        reverseClockRegister(node->r2, R2LENGTH);
        s->s1 = node;
        reverseClockMajBit(node, round-1, frame);
    }else if((s->r1[R1MAJ+1] == maj) && (s->r3[R3MAJ+1] == maj) &&
    (s->r2[R2MAJ+1] != maj) && (s->r2[R2MAJ+1] == s->r2[R2MAJ])){
        // si' = sj' != sk' = sk
        reverseClockRegister(node->r1, R1LENGTH);
        reverseClockRegister(node->r3, R3LENGTH);
        s->s1 = node;
        reverseClockMajBit(node, round-1, frame);
    }else if((s->r2[R2MAJ+1] == maj) && (s->r3[R3MAJ+1] == maj) &&
    (s->r1[R1MAJ+1] != maj) && (s->r1[R1MAJ+1] == s->r1[R1MAJ])){
        // si' = sj' != sk' = sk
        reverseClockRegister(node->r2, R2LENGTH);
        reverseClockRegister(node->r3, R3LENGTH);
        s->s1 = node;
        reverseClockMajBit(node, round-1, frame);
    }else if((s->r1[R1MAJ+1] == maj) && (s->r2[R2MAJ+1] == maj) &&
    (s->r3[R3MAJ+1] != maj) && (s->r3[R3MAJ+1] != s->r3[R3MAJ])){
        // si' = sj' != sk' != sk Caso non possibile
        return 0;
    }else if((s->r1[R1MAJ+1] == maj) && (s->r3[R3MAJ+1] == maj) &&
    (s->r2[R2MAJ+1] != maj) && (s->r2[R2MAJ+1] != s->r2[R2MAJ])){
        // si' = sj' != sk' != sk Caso non possibile
        return 0;
```

```
35      }else if((s->r2[R2MAJ+1] == maj) && (s->r3[R3MAJ+1] == maj) &&
        (s->r1[R1MAJ+1] != maj) && (s->r1[R1MAJ+1] != s->r1[R1MAJ])){
36          // si' = sj' != sk' != sk Caso non possibile
37          return 0;
38      }else{
39          // caso con s1' = s2' = s3'
40          int r1m = s->r1[R1MAJ];
41          int r2m = s->r2[R2MAJ];
42          int r3m = s->r3[R3MAJ];
43
44          if((r1m == maj) && (r2m == maj) && (r3m == maj)){
45              // s1' = s2' = s3' = s1 = s2 = s3
46              reverseClockRegister(node->r1, R1LENGTH);
47              reverseClockRegister(node->r2, R2LENGTH);
48              reverseClockRegister(node->r3, R3LENGTH);
49              s->s1 = node;
50              reverseClockMajBit(node, round-1, frame);
51          }else if(((r1m == r2m) && (r1m == maj) && (r1m != r3m)) ||
52                   ((r1m == r3m) && (r1m == maj) && (r1m != r2m)) ||
53                   ((r2m == r3m) && (r2m == maj) && (r2m != r1m))){
54              // s1' = s2' = s3' = si = sj != sk
55              State node1 = newNode(s->r1, s->r2, s->r3);
56              clock2(node1->r1, node1->r2, node1->r3);
57              s->s1 = node1;
58              reverseClockMajBit(node1, round-1, frame);
59
60              State node2 = newNode(s->r1, s->r2, s->r3);
61              reverseClockRegister(node2->r1, R1LENGTH);
62              reverseClockRegister(node2->r2, R2LENGTH);
63              reverseClockRegister(node2->r3, R3LENGTH);
64              s->s2 = node2;
65              reverseClockMajBit(node2, round-1, frame);
66          }else if(((r1m == r2m) && (r1m != maj) && (r3m == maj)) ||
67                   ((r1m == r3m) && (r1m != maj) && (r2m == maj)) ||
68                   ((r2m == r3m) && (r2m != maj) && (r1m == maj))){
69              // s1' = s2' = s3' = si != sj = sk
70              int diff = clock3(s->r1, s->r2, s->r3);
71              State node1 = newNode(s->r1, s->r2, s->r3);
72              State node2 = newNode(s->r1, s->r2, s->r3);
73
74              if(diff==1){
75                  reverseClockRegister(node1->r1, R1LENGTH);
76                  reverseClockRegister(node1->r2, R2LENGTH);
77
78                  reverseClockRegister(node2->r1, R1LENGTH);
79                  reverseClockRegister(node2->r3, R3LENGTH);
80              }else if(diff==2){
81                  reverseClockRegister(node1->r1, R1LENGTH);
82                  reverseClockRegister(node1->r2, R2LENGTH);
83
84                  reverseClockRegister(node2->r2, R2LENGTH);
85                  reverseClockRegister(node2->r3, R3LENGTH);
```

```
            }else{
                reverseClockRegister(node1->r1, R1LENGTH);
                reverseClockRegister(node1->r3, R3LENGTH);

                reverseClockRegister(node2->r2, R2LENGTH);
                reverseClockRegister(node2->r3, R3LENGTH);
            }
            s->s1 = node1;
            reverseClockMajBit(node1, round-1, frame);

            s->s2 = node2;
            reverseClockMajBit(node2, round-1, frame);

            State node3 = newNode(s->r1, s->r2, s->r3);
            reverseClockRegister(node3->r1, R1LENGTH);
            reverseClockRegister(node3->r2, R2LENGTH);
            reverseClockRegister(node3->r3, R3LENGTH);
            s->s3 = node3;
            reverseClockMajBit(node3, round-1, frame);
        }else if((r1m != maj) && (r2m != maj) && (r3m != maj)){
            // s1' = s2' = s3' != s1 = s2 = s3
            State node1 = newNode(s->r1, s->r2, s->r3);
            reverseClockRegister(node1->r1, R1LENGTH);
            reverseClockRegister(node1->r2, R2LENGTH);
            s->s1 = node1;
            reverseClockMajBit(node1, round-1, frame);

            State node2 = newNode(s->r1, s->r2, s->r3);
            reverseClockRegister(node2->r1, R1LENGTH);
            reverseClockRegister(node2->r3, R3LENGTH);
            s->s2 = node2;
            reverseClockMajBit(node2, round-1, frame);

            State node3 = newNode(s->r1, s->r2, s->r3);
            reverseClockRegister(node3->r2, R2LENGTH);
            reverseClockRegister(node3->r3, R3LENGTH);
            s->s3 = node3;
            reverseClockMajBit(node3, round-1, frame);

            State node4 = newNode(s->r1, s->r2, s->r3);
            reverseClockRegister(node4->r1, R1LENGTH);
            reverseClockRegister(node4->r2, R2LENGTH);
            reverseClockRegister(node4->r3, R3LENGTH);
            s->s4 = node4;
            reverseClockMajBit(node4, round-1, frame);
        }else{
            printf("\nERRORE\n%d-%d-%d-%d\n", r1m, r2m, r3m, maj);
            return 0;
        }
    }
}
```

5.7.2 *Reverse* XOR frame

La funzione seguente è l'ultima che viene eseguita per completare l'attacco. In essa vengono effettuate le operazione esposte nel Capitolo 4.4.2.

```
void reverseXorFrame(State s, int* frame){
    for(int i=21; i>=0; i--){
        reverseClockRegister(s->r1, R1LENGTH);
        reverseClockRegister(s->r2, R2LENGTH);
        reverseClockRegister(s->r3, R3LENGTH);

        s->r1[0] = s->r1[0] ^ frame[i];
        s->r2[0] = s->r2[0] ^ frame[i];
        s->r3[0] = s->r3[0] ^ frame[i];
    }
}
```

5.8 Specifiche e durata

Attacco	Output Stream	Complessità	Tempo di esecuzione	Memoria occupata	CPU
Attacco effettuato	114 bit	2^{27}	15 secondi	0.5MB	12.2%
Attacco completo	114 bit $\cdot 2^{20}$	2^{47}	6 mesi (max)	$<$ 1.0MB	$<$ 15%

Tabella 5.1: *Entrambi gli attacchi riportati sono single thread. I valori dell'attacco completo sono stati calcolati a livello teorico partendo dai dati dell'attacco effettuato*

Come esposto nel Capitolo 4.5, l'attacco completo necessita di esaminare almeno 2^{20} differenti posizioni iniziali, al fine di essere sicuri che il registro R_3 non esegua *clock* per 10 *round*.

Nell'attacco completo il tempo di esecuzione può variare molto in quanto 2^{20} posizioni iniziali è il numero minimo necessario affinché si abbia la certezza che la pre-condizione si verifichi ma, l'evento può avvenire molto prima dell'ultima iterazione (es. se nel primo *frame* R_3 non esegue il *clock* per 10 *round* il tempo di esecuzione scende a 15 secondi).

Un fattore importante da considerare riguarda la parallelizzazione degli attacchi ai singoli *frame*. Essa riduce drasticamente il tempo di esecuzione ma ovviamente incrementa la memoria occupata e la percentuale di CPU utilizzata dipendentemente da quanti processi paralleli vengono eseguiti.

Numero di processi paralleli	Tempo massimo di esecuzione
1	6.06 mesi $\approx$ 182 giorni
10	18.2 giorni
20	9.10 giorni
25	7.28 giorni
50	3.64 giorni
100	1.82 giorni $\approx$ 43.69 ore
200	21.84 ore
500	8.73 ore

Tabella 5.2: All'aumentare del numero di processi paralleli, il tempo massimo di esecuzione diminuisce. Valori derivati dai numeri della Tabella 5.1

I test sono stati svolti su un laptop con le seguenti componenti:

Componente	Specifica
CPU	Intel i7-10750H, 6 core 12 thread
Memoria	16GB DDR4 2933MHz

Conclusioni

La tesi, elaborata all'interno del Dipartimento di Informatica, si è sviluppata dall'analisi del funzionamento di 3 cifrari del GSM: A5/1, A5/2 e A5/3 descritti nei primi tre Capitoli.

É stato effettuato un approfondimento in merito ai più noti attacchi in letteratura con il tentativo di stabilire quali fossero i più efficienti. Sono stati presi in considerazione vari parametri quali complessità, requisiti, problematiche ed eventuali tempistiche, se disponibili; a corredo sono state elaborate varie tabelle di analisi delle differenti tipologie di attacco.

Da detta analisi, si è deciso di approfondire lo studio riguardante l'attacco teorico di tipo "Known Plaintext" a A5/1 dei ricercatori Biham e Dunkelman, illustrato nel Capitolo 4, in quanto ritenuto il più efficiente.

Dopo un'accurata ricerca finalizzata ad una dettagliata conoscenza delle specifiche procedure di attacco, si è implementato, in linguaggio C, l'attacco al singolo *frame* dal quale si è riusciti a risalire fino allo stato chiave che permette di cifrare e decifrare velocemente ogni *frame* di uno specifico utente.

Ritengo che si siano ottenuti dei buoni risultati in quanto, avendo a disposizione il *frame* corretto, si è riusciti ad individuare in soli 15 secondi lo stato chiave.

Possibili sviluppi futuri riguardano l'esecuzione, in tempi ragionevoli, dell'attacco completo e non unicamente al singolo *frame*; si possono realizzare questi miglioramenti tramite l'esecuzione in parallelo degli attacchi ai singoli *frame* come descritto nel Capitolo 5.8.

Considerato che si dovrebbero parallelizzare interi processi e non semplici operazioni, si ritiene possa essere più efficiente l'utilizzo di CPU con un numero alto di CORE, rispetto ad una GPU.

Bibliografia

A5/1

[1] Upadhyay, D., Sharma, P., Valiveti, S. (2014). *Randomness analysis of A5/1 Stream Cipher for secure mobile communication*. International Journal Of Computer Science & Communication, 3, (pp. 95-100). http://www.csjournals.com/IJCSC/PDF5-1/21.%20darshana.pdf

[2] Maximov, A., Johansson, T., Babbage, S. (2004, August). *An improved correlation attack on A5/1*. In International Workshop on Selected Areas in Cryptography (pp. 1-18). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-540-30564-4_1.pdf

[3] Biham, E., Dunkelman, O. (2000, December). *Cryptanalysis of the A5/1 GSM stream cipher*. In International Conference on Cryptology in India (pp. 43-51). Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/3-540-44495-5_5

[4] Golić, J. D. (1997, May). *Cryptanalysis of alleged A5 stream cipher*. In International Conference on the Theory and Applications of Cryptographic Techniques (pp. 239-255). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-69053-0_17.pdf

[5] Stockinger, T. (2005). *Gsm network and its privacy-the a5 stream cipher*. CiteSeerX. http://www.nop.at/gsm_a5/GSM_A5.pdf

[6] Biryukov, A., Shamir, A., Wagner, D. (2000, April). *Real Time Cryptanalysis of A5/1 on a PC*. In International Workshop on Fast Software Encryption (pp. 1-18). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-44706-7_1.pdf

[7] Ekdahl, P., Johansson, T. (2003). *Another attack on A5/1*. IEEE transactions on information theory, 49, (pp. 284-289). https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1159783

[8] Barkan, E., Biham, E. (2005, August). *Conditional estimators: An effective attack on A5/1*. In International Workshop on Selected Areas in Cryptography (pp. 1-19). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/11693383_1.pdf

[9] Gendrullis, T., Novotný, M., Rupp, A. (2008, August). *A real-world attack breaking A5/1 within hours*. In International Workshop on Cryptographic Hardware and Embedded Systems (pp. 266-282). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007%252F978-3-540-85053-3_17.pdf

[10] Wikipedia. *A5/1*. https://it.wikipedia.org/wiki/A5/1

[11] Meyer, S. (2011). *Breaking GSM with rainbow Tables*. arXiv preprint arXiv:1107.1086. https://arxiv.org/ftp/arxiv/papers/1107/1107.1086.pdf

A5/2

[12] Bogdanov, A., Eisenbarth, T., Rupp, A. (2007, September). *A hardware-assisted realtime attack on A5/2 without precomputations*. In International Workshop on Cryptographic Hardware and Embedded Systems (pp. 394-412). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-540-74735-2_27.pdf

[13] Barkan, E., Biham, E., Keller, N. (2003, August). *Instant ciphertext-only cryptanalysis of GSM encrypted communication*. In Annual international cryptology conference (pp. 600-616). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-540-45146-4_35.pdf

[14] Petrovic, S., Fuster-Sabater, A. (2000). *CRYPTANALYSIS OF THE A5/2 ALGORITHM*. IACR Cryptol. ePrint Arch., 2000, 52. https://eprint.iacr.org/2000/052.pdf

[15] Kostrzewa, A. (2011). *Development of a man in the middle attack on the GSM Um-Interface*. Technische Universität Berlin Fakultät IV, Institut für Softwaretechnik und Theoretische Informatik. https://www.isti.tu-berlin.de/fileadmin/fg214/finished_theses/kostrzewa/diplom_kostrzewa.pdf

A5/3

[16] Wikipedia. *Kasumi*. https://en.wikipedia.org/wiki/KASUMI

[17] Vrentzos, E., Kostopoulos, G., Koufopavlou, O. (2006, June). *Hardware implementation of the A5/3 & A5/4 GSM encryption algorithms*. In Conference WAC. https://www.wacong.org/wac2006/allpapers/ifmip/ifmip_196.pdf

[18] Sankaliya, A. R., Mishra, V., Mandloi, A. (2011). *Implimentation of Cryptographic Algorithm for GSM and UMTS Systems*. International Journal of Network Security & Its Applications, 3, 81. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.211.1230&rep=rep1&type=pdf

[19] *Specification A5/3.* https://www.3gpp.org/ftp/tsg_sa/WG3_Security/TSGS3_26_Oxford/Docs/PDF/S3-020657.pdf

[20] Kühn, U. (2001, May). *Cryptanalysis of reduced-round MISTY.* In international conference on the theory and applications of cryptographic techniques (pp. 325-339). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-44987-6_20.pdf

[21] Blunden, M., Escott, A. (2001, April). *Related key attacks on reduced round KASUMI.* In International workshop on fast software encryption (pp. 277-285). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/3-540-45473-X_23.pdf

[22] Biham, E., Dunkelman, O., Keller, N. (2005, December). *A related-key rectangle attack on the full KASUMI.* In International Conference on the Theory and Application of Cryptology and Information Security (pp. 443-461). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/11593447_24.pdf

[23] Dunkelman, O., Keller, N., Shamir, A. (2010, August). *A practical-time related-key attack on the KASUMI cryptosystem used in GSM and 3G telephony.* In Annual cryptology conference (pp. 393-410). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-642-14623-7_21.pdf

[24] Jia, K., Li, L., Rechberger, C., Chen, J., Wang, X. (2012, August). *Improved cryptanalysis of the block cipher KASUMI.* In International Conference on Selected Areas in Cryptography (pp. 222-233). Springer, Berlin, Heidelberg. https://link.springer.com/content/pdf/10.1007/978-3-642-35999-6_15.pdf

[25] Saito, T. (2011). *A Single-Key Attack on 6-Round KASUMI.* IACR Cryptol. ePrint Arch., 2011, 584. https://eprint.iacr.org/2011/584.pdf

[26] Anand T., Shanmugam M., Santhoshini B. (2019). *Rainbow table attack on 3rd generation GSM Telephony secure algorithm - A5/3.* International Journal of Recent Technology and Engineering (IJRTE). https://www.ijrte.org/wp-content/uploads/papers/v7i5s4/E10170275S419.pdf