An Overview of eSTREAM Ciphers

Pratyay Mukherjee
Centre of Excellence in Cryptology
Indian Statistical Institute
Contents

1 Rabbit

1.1 Introduction ......................................................... 1

1.2 Specifications of Rabbit ........................................... 2

  1.2.1 Notation ..................................................... 2

  1.2.2 A high-level description .................................. 2

  1.2.3 Key-Setup Scheme ........................................... 4

  1.2.4 IV-Setup Scheme ............................................. 5

  1.2.5 Extraction Scheme .......................................... 6

  1.2.6 Next-State Function ........................................ 7

  1.2.7 Encryption/decryption Scheme ............................. 8

1.3 Security Properties of Rabbit ................................... 11

  1.3.1 KEY-SETUP Properties ...................................... 11

  1.3.2 IV Setup Properties ......................................... 13

  1.3.3 Partial Guessing ............................................ 14

  1.3.4 Algebraic Analysis ......................................... 15

  1.3.5 Correlation Analysis ........................................ 18

  1.3.6 Differential Analysis ........................................ 19

  1.3.7 Statistical Test ............................................. 20

  1.3.8 Mod $n$ Analysis ............................................ 21

  1.3.9 Period Length ................................................ 22

1.4 Cryptanalysis of Rabbit ........................................... 22
# CONTENTS

1.4.1 On a Bias of Rabbit ........................................... 22
1.4.2 Another Cryptanalysis based on FFT ......................... 23
1.4.3 Differential Fault Analysis of Rabbit ....................... 23

1.5 Performance Evaluations of Rabbit .............................. 23
   1.5.1 Intel Platforms ........................................... 24
   1.5.2 Power PC Platform ....................................... 25
   1.5.3 ARM7 Platform ........................................... 25
   1.5.4 MIPS 4Kc Platform ....................................... 26
   1.5.5 Hardware Performances ................................... 27

1.6 Strength and Advantages ......................................... 28
   1.6.1 Compact Design ........................................... 28
   1.6.2 High Security ............................................ 28

1.7 A few remarks about design ....................................... 28
   1.7.1 Design of Counter Systems ................................. 29
   1.7.2 Symmetry and Mixing .................................... 29

1.8 Conclusion .................................................... 29

1.9 A Simple C Implementation of Rabbit .......................... 30

2 Salsa 20

2.1 Introduction .................................................... 43

2.2 Specifications of Salsa20 ....................................... 43
   2.2.1 The quarterround function ................................. 44
   2.2.2 The rowround function ................................... 44
   2.2.3 The columnround function ................................. 45
   2.2.4 The doubleround function ................................ 45
   2.2.5 The littleendian function ................................. 45
   2.2.6 The Salsa20 hash function ............................... 45
   2.2.7 The Salsa20 expansion function ......................... 46
   2.2.8 The Salsa20 encryption function ......................... 47
2.3 Security Properties of Salsa20
  2.3.1 The cost of an attack
  2.3.2 Notes on the diagonal constants
  2.3.3 Diffusion in Salsa20
  2.3.4 Differential attacks
  2.3.5 Algebraic attacks
  2.3.6 Other attacks
2.4 Performance Evaluation of Salsa20
  2.4.1 Salsa20 on the AMD Athlon
  2.4.2 Salsa20 on the IBM PowerPC RS64 IV (Sstar)
  2.4.3 Salsa20 on the Intel Pentium III
  2.4.4 Salsa20 on the Intel Pentium 4 f12 (Willamette)
  2.4.5 Salsa20 on the Intel Pentium M
  2.4.6 Salsa20 on the Motorola PowerPC 7410 (G4)
  2.4.7 Salsa20 on the Sun UltraSPARC II
  2.4.8 Salsa20 on the Sun UltraSPARC III
  2.4.9 Salsa20 on next-generation CPUs
  2.4.10 Salsa20 on smaller platforms
2.5 Cryptanalysis of Salsa20
  2.5.1 Truncated differential cryptanalysis of five rounds of Salsa20
  2.5.2 Non-randomness in Salsa20
  2.5.3 Differential Cryptanalysis of Salsa20/8
  2.5.4 On the Salsa20 Core Function
2.6 Conclusion
2.7 A Simple C Implementation of Salsa20
3 HC-128
  3.1 Introduction
  3.2 Specifications of HC-128
3.2.1 Notation ................................................................. 78
3.2.2 Key and IV Set up .................................................... 79
3.2.3 Keystream Generation ................................................ 79

3.3 Security Properties of HC-128 ....................................... 79
3.3.1 Period length .......................................................... 81
3.3.2 Security of the secret key .......................................... 81
3.3.3 Security of the initialization process (Key-IV setup) ......... 81
3.3.4 Randomness of the keystream ...................................... 81

3.4 Implementation and Performance of HC-128 ...................... 83
3.4.1 The optimized implementation of HC-128 ....................... 83
3.4.2 The performance of HC-128 ........................................ 84

3.5 Cryptanalysis of HC-128 .............................................. 85
3.5.1 Approximating the Feedback Functions [Maitra et al. WCC 2009] ...................................................... 85
3.5.2 Extending the designer's cryptanalysis [Maitra et al. WCC 2009] ...................................................... 87
3.5.3 State Leakage in Keystream ........................................ 87
3.5.4 State Recovery from Partial State Exposure .................... 88
3.5.5 Differential Fault Analysis ......................................... 89

3.6 A New Variant of HC-128 to Resist Known Weaknesses .......... 90

3.7 Conclusion .............................................................. 91
3.8 A Simple C implementation of HC-128 ............................... 91

4 SOSEMANUK ............................................................... 95
4.1 Introduction ............................................................. 95
4.2 Specifications of SOSEMANUK ........................................ 96
4.2.1 SERPENT and derivatives ........................................ 96
4.2.2 The LFSR .......................................................... 97
4.2.3 Output transformation .............................................. 99
4.2.4 SOSEMANUK workflow .......................................... 99
4.2.5 Initialization of SOSEMANUK ...................................... 102
4.3 Design Rationale of SOSEMANUK

4.3.1 Key initialization and IV injection

4.3.2 The LFSR

4.3.3 The FSM

4.3.4 The output transformation

4.4 Security Properties of SOSEMANUK

4.4.1 Time-memory-data tradeoff attacks

4.4.2 Guess and determine attacks

4.4.3 Correlation attacks

4.4.4 Distinguishing attacks

4.4.5 Algebraic Attacks

4.5 Performances of SOSEMANUK

4.6 Cryptanalysis of SOSEMANUK

4.6.1 Improved Guess and Determine Attack on SOSEMANUK

4.6.2 Evaluation With Regard to Guess-and-Determine Attacks

4.6.3 Cryptanalysis of SOSEMANUK and SNOW 2.0 Using Linear Masks

4.6.4 Improved Linear Cryptanalysis of SOSEMANUK

4.6.5 A Byte-Based Guess and Determine Attack on SOSEMANUK

4.6.6 Differential Fault Analysis of SOSEMANUK

4.7 Conclusion

4.8 A Simple C Implementation of SOSEMANUK

5 Trivium

5.1 Introduction

5.2 Specifications of Trivium

5.2.1 Key stream generation

5.2.2 Key and IV setup

5.3 Implementation of Trivium

5.3.1 Hardware Implementation
5.3.2 Software Implementation ........................................ 141
5.3.3 Few other Hardware Implementations ...................... 141
5.4 Security Properties of Trivium ................................. 141
  5.4.1 Correlation ..................................................... 142
  5.4.2 Period .......................................................... 142
  5.4.3 Guess and Determine attacks ............................... 143
  5.4.4 Algebraic attacks .......................................... 143
  5.4.5 Resynchronization attacks ................................ 143
5.5 Cryptanalysis of Trivium ........................................ 143
  5.5.1 Cryptanalytic Results on Trivium ......................... 143
  5.5.2 Two Trivial Attacks on Trivium ............................ 144
  5.5.3 An Algebraic Analysis based on the Boolean SAT Problem 144
  5.5.4 Differential Fault Analysis of Trivium .................... 145
  5.5.5 Floating Fault Analysis of Trivium ....................... 146
  5.5.6 Algebraic Attack Against Trivium ......................... 146
  5.5.7 Cube attacks on Trivium ................................... 146
  5.5.8 Floating Fault Analysis of Trivium under Weaker Assumptions 147
  5.5.9 Hard Fault Analysis of Trivium ............................ 147
  5.5.10 Analysis of Trivium by a Simulated Annealing Variant 147
  5.5.11 The Cube Attack on Stream Cipher Trivium and Quadraticity Tests 148
  5.5.12 Improved Differential Fault Analysis of Trivium ........ 148
  5.5.13 Conditional Differential Cryptanalysis of Trivium ........ 149
5.6 Conclusion .......................................................... 149
5.7 A Simple C Implementation of Trivium ....................... 149

6 Grain v1 ................................................................. 155
  6.1 Introduction ..................................................... 155
  6.2 Specifications of Grain v1 .................................... 156
  6.2.1 Key and IV Initialization ................................. 157
6.2.2 Throughput Rate ................................................. 157
6.3 Security Properties of Grain ........................................ 159
  6.3.1 Linear Approximations ........................................ 159
  6.3.2 Algebraic Attacks ............................................. 159
  6.3.3 Time-Memory-Data Trade-off Attack ......................... 160
  6.3.4 Fault Attacks ............................................... 160
6.4 Design Choices of Grain ........................................... 160
  6.4.1 Size of the LFSR and the NFSR ............................ 160
  6.4.2 Speed Acceleration .......................................... 161
  6.4.3 Choice of $f(\cdot)$ ........................................... 161
  6.4.4 Choice of $g(\cdot)$ ........................................... 161
  6.4.5 Choice of output function .................................. 162
6.5 Hardware Performance of Grain-128 ............................... 162
6.6 Hardware Implementations of Grain ............................... 162
  6.6.1 An Improved Implementation of Grain ..................... 163
  6.6.2 Design and Implementation Based on SABL Logic .......... 163
  6.6.3 Comparison of Low-Power Implementations of Trivium and Grain ... 164
  6.6.4 Other comparative studies .................................. 164
6.7 Cryptanalysis of Grain ............................................ 164
  6.7.1 Slide Resynchronization Attack on the Initialization of Grain v1 .... 164
  6.7.2 Related-Key Chosen IV Attack on Grain-v1 and Grain-128 ........ 164
  6.7.3 Cryptanalysis of Grain using Time / Memory /Data Tradeoffs .......... 165
  6.7.4 Distinguishing Attack on Grain using Linear Sequential Circuit Method 165
  6.7.5 Analysis of Grain’s Initialization Algorithm .................. 165
  6.7.6 Fault Analysis of Grain-128 .................................. 165
  6.7.7 An Experimentally Verified Attack on Full Grain-128 ............ 166
  6.7.8 Breaking Grain-128 with Dynamic Cube Attacks ................ 166
  6.7.9 Fault analysis of Grain-128 by targeting NFSR ................. 167
6.8 Conclusion ...................................................... 167
7 MICKEY 2.0

7.1 Introduction ................................................. 171
7.2 Specifications of MICKEY 2.0 ................................. 172
  7.2.1 Input and Output Parameters ............................... 172
  7.2.2 Acceptable use ............................................ 172
  7.2.3 Components of the keystream generator .................. 172
  7.2.4 Key-loading and Initialization ............................. 174
  7.2.5 Generating keystream ....................................... 176
7.3 Design principles and Security Properties of MICKEY 2.0 ................................. 176
  7.3.1 The variable clocking of $R$ ............................... 176
  7.3.2 Motivation for the variable clocking ...................... 177
  7.3.3 Selection of clock control bits ............................. 178
  7.3.4 The $S$ register feedback function ....................... 178
  7.3.5 Algebraic Attacks .......................................... 179
  7.3.6 State Entropy .............................................. 180
  7.3.7 Output function ............................................ 180
7.4 Changes from MICKEY version 1 ............................. 180
  7.4.1 What are the changes from MICKEY version 1? ............ 180
  7.4.2 Time-Memory-Data (TMD) tradeoff, with or without BSW sampling ... 181
  7.4.3 State entropy loss and keystream coverage ................ 183
  7.4.4 Weak keys .................................................. 183
7.5 Performance of MICKEY 2.0 .................................... 184
7.6 Hardware Implementations of MICKEY .......................... 184
  7.6.1 On the Hardware Implementation of the MICKEY-128 Stream Cipher ... 184
  7.6.2 On the Parallelization of the MICKEY-128 2.0 ............... 185
  7.6.3 Implementation of parallel and reconfigurable MICKEY algorithm ... 185
  7.6.4 Comparison of FPGA-Targeted Hardware Implementations ........... 185
CONTENTS

7.6.5 FPGA Implementations of eSTREAM Phase-2 candidates .......................... 185
7.6.6 Hardware results for selected stream cipher candidates .......................... 186
7.6.7 Review of stream ciphers candidates from a low resource hardware perspective 186

7.7 Cryptanalysis of MICKEY 2.0 ...................................................................... 186
7.7.1 TMD-Tradeoff and State Entropy Loss Considerations of MICKEY .......... 186
7.7.2 Power Analysis Attacks on a Hardware Implementation of MICKEY ........ 187
7.7.3 Nonsmooth Cryptanalysis, with an Application to MICKEY ................. 187
7.7.4 Correlation Power Analysis Against Stream Cipher MICKEY v2 ............ 187

7.8 Conclusion ................................................................................................. 188

7.9 A Simple C Implementation of MICKEY 2.0 ........................................... 188
Chapter 1

Rabbit

1.1 Introduction

Rabbit is a synchronous stream cipher introduced in Fast Software Encryption [30] in 2003. It is one of the most potent candidate to the eSTREAM project [58]. The designers of the cipher targeted to use it in both software and hardware environments. The design is very strong as the designers provided the security analysis considering several possible attacks viz. algebraic, correlation, differential, guess-and-determine, and statistical attacks. Also they have presented very strong reasoning in [29] to conclude that these attacks are costlier than the brute-force exhaustive key search. In [10], Aumasson has shown the existence of a non-null bias in the pseudorandom key-stream generated by Rabbit, from the observation that the core function (will be explained later) is strongly unbalanced. But he concluded in that paper that, the distinguisher would take a time which is much greater than the cost of exhaustive key-search. So, until now, no big weakness of Rabbit has been found.

Briefly speaking, The Rabbit Algorithm takes 128-bit key and if necessary 64-bit IV as input. In each iteration it generates 128-bit output. The output is pseudo-random in the natural sense that they can not be distinguished from random strings of 128-bit with non-negligible probability (i.e. in efficient manner). The core of this cipher consists of 513 internal state bits. Obviously the output generated in each iteration is some combination of these state-bits. The 513 bits are divided into eight 32-bit state variable, eight 32-bit counter and one counter carry bit. The state functions which update these state variables are non-linear and thus build the basic of security provided by this cipher.

The design of rabbit enables faster implementation than common ciphers. Mostly bitwise operations like concatenation, bitwise xor, shifting are involved which explains its faster performances. A few costly operations like squaring are necessary to enhance the amount of
non-linearity. A key of size 128-bit can be used for encrypting up to $2^{64}$ blocks of plain-text. This means that for an attacker who does not know the key, it should not be possible to distinguish up to $2^{64}$ blocks of cipher-text output from the output of a truly random generator, using lesser steps than would be required for an exhaustive key search over $2^{128}$ keys.

1.2 Specifications of Rabbit

1.2.1 Notation

Although most of the notation used here are well-known, we provide the notations used here in tabular form:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\oplus$</td>
<td>Logical Exclusive OR.</td>
</tr>
<tr>
<td>$&amp;$</td>
<td>Logical AND.</td>
</tr>
<tr>
<td>$\ll/\ggg$</td>
<td>Left/Right Rotation.</td>
</tr>
<tr>
<td>$\ll/\gg$</td>
<td>Left/Right Shift.</td>
</tr>
<tr>
<td>$\odot$</td>
<td>Concatenation.</td>
</tr>
<tr>
<td>$A[g..h]$</td>
<td>Bit number g through h of A.</td>
</tr>
</tbody>
</table>

Also a few important things we mention to avoid unnecessary confusion. While numbering the bits, the least significant bit is denoted by 0 and hexadecimal numbers are prefixed conventionally by “0x”.

The internal state of the stream cipher consists of 513 bits as stated earlier. 512 bits are divided between eight 32-bit state variables $x_{j,i}$ and eight 32-bit counter variables $c_{j,i}$, where $x_{j,i}$ is the state variable of subsystem $j$ at iteration $i$, and $c_{j,i}$ denotes the corresponding counter variable. There is one counter carry bit $\phi_{7,i}$, which needs to be stored between iterations. Basically it stores the carry output of a summation which updates counters in each iteration (to be elaborated later). This counter carry bit is initialized to zero. The eight state variables and the eight counters are derived from the key at initialization which we explore next.

1.2.2 A high-level description

First we give a high-level description through a block diagram in Figure 1.1. Once the things becomes vivid pictorially, we can move to lower level with details of each block. For the time being we do not have to worry about those blocks. We just consider them as Black-boxes.

The Key-Setup, IV-Setup and Extraction are three main functional block of Rabbit. It must be noted that IV-Setup is optional. However, it can be noticed from the block-diagram that each of these main functional blocks interacts with the Next-State functions (a number
1.2. Specifications of Rabbit

Figure 1.1: The high-level description of Rabbit
of times) and then forwards its output to the next block. The Next-state function is the most important one which internally interacts with other functions viz. g-Function, Counter System. The Next-state function updates the internal states of rabbits combining them with updated counters. Now we discuss every functional block one by one.

1.2.3 Key-Setup Scheme

The Key-Setup scheme consists of three main parts. It takes the key as input and initializes them. Then it interacts with Next-State function several times. Finally to prevent key-recovery by inversion of the counter system, it re-initializes the counter system. The goal of the algorithm used in this step is to expand the input key (128-bit) into both the eight state variables and the eight counters such that there is a one-to-one correspondence between the key and the initial state variables $x_{j,0}$ and the initial counters $c_{j,0}$. The key, $K_{[127..0]}$, is divided into eight sub-keys: $k_0 = K_{[15..0]}$, $k_1 = K_{[31..16]}$, ..., $k_7 = K_{[127..112]}$. The state and counter variables are initialized from the sub-keys as follows:

$$x_{j,0} = \begin{cases} k_{(j+1)\text{mod} \ 8} \odot k_j & \text{for } j \text{ even} \\ k_{(j+5)\text{mod} \ 8} \odot k_{(j+4)\text{mod} \ 8} & \text{for } j \text{ odd} \end{cases}$$ (1.1)

and

$$c_{j,0} = \begin{cases} k_{(j+4)\text{mod} \ 8} \odot k_{(j+5)\text{mod} \ 8} & \text{for } j \text{ even} \\ k_j \odot k_{(j+1)\text{mod} \ 8} & \text{for } j \text{ odd} \end{cases}$$ (1.2)

Then, the system is iterated four times, according to the Next-state function, to diminish correlations between bits in the key and bits in the internal state variables. Finally, the counter variables are re-initialized according to:

$$c_{j,4} = c_{j,4} \oplus x_{((j+4)\text{mod} \ 8),4}$$ (1.3)

for all $j$, to prevent recovery of the key by inversion of the counter system.

We provide the summary of what this scheme does in a pseudo-code like manner which would help the reader to visualize the procedure in much easier way. (see Algorithm 1)
1.2. Specifications of Rabbit

Algorithm 1 KEY-SETUP

{Step-1: Initializing System....}
for $j = 0 \rightarrow 7$
  if ($j$ is even) then
    $x_{j,0} \leftarrow \text{CONCAT}(k_{(j+1) \mod 8}, k_{j})$
    $c_{j,0} \leftarrow \text{CONCAT}(k_{(j+4) \mod 8}, k_{(j+5) \mod 8})$
  else
    $x_{j,0} \leftarrow \text{CONCAT}(k_{(j+5) \mod 8}, k_{(j+4) \mod 8})$
    $c_{j,0} \leftarrow \text{CONCAT}(k_{j}, k_{(j+1) \mod 8})$
  end if
end for

{Step-2: Iterating System by Next-State Function....}
for $i = 0 \rightarrow 3$
  State[$x_{j,i+1, c_{j,i+1}}$] $\leftarrow$ NEXT-STATE(State[$x_{j,i, c_{j,i}}$]) $\forall j \in \{0, \ldots, 7\}$
end for

{Step-3: Re-initializing Counters....}
for $j = 0 \rightarrow 7$
  $c_{j,4} \leftarrow \text{XOR}(c_{j,4}, x_{(j+4) \mod 8}, 4)$
end for

1.2.4 IV-Setup Scheme

Now, after completion of Key-Setup, one can optionally run IV-setup scheme. The input of this part is the output from Key-Setup and a 64-bit IV. The internal states after key-setup is called the Master State. In this scheme, a copy of that Master State is modified. The IV-setup scheme works by modifying the counter state as function of the IV. This is done by XORing the 64-bit IV on all the 256 bits of the counter state. The 64 bits of the IV are denoted $IV^{[63..0]}$. The counters are modified as:

\[
\begin{align*}
  c_{0,4} &= c_{0,4} \oplus IV^{[31..0]} \\
  c_{2,4} &= c_{2,4} \oplus IV^{[63..32]} \\
  c_{4,4} &= c_{4,4} \oplus IV^{[31..0]} \\
  c_{6,4} &= c_{6,4} \oplus IV^{[63..32]} \\
  c_{1,4} &= c_{1,4} \oplus IV^{[63..48]} \odot IV^{[31..16]} \\
  c_{3,4} &= c_{3,4} \oplus IV^{[47..32]} \odot IV^{[15..0]} \\
  c_{5,4} &= c_{5,4} \oplus IV^{[63..48]} \odot IV^{[31..16]} \\
  c_{7,4} &= c_{7,4} \oplus IV^{[47..32]} \odot IV^{[15..0]} \quad (1.4)
\end{align*}
\]

The system is then iterated four times to make all state bits non-linearly dependent on all IV bits. This is essential to incorporate non-linearity in this scheme. Like the previous scheme,
this is done by calling the Next-state Function 4 times. The modification of the counter by the IV guarantees that all $2^{64}$ different IV vectors will lead to unique key-streams. The scheme has been summarized by a high-level pseudo-code in Algorithm 2.

Algorithm 2 IV-SETUP

\{Step-1: Modifying counters by input IV\ldots\}\}

for $j = 0 \rightarrow 7$ do

if $j = 0 \mod 4$ then

$c_{j,4} \leftarrow \text{XOR}(c_{j,4}, IV^{[31..0]})$

end if

if $j = 1 \mod 4$ then

$c_{j,4} \leftarrow \text{XOR}(c_{j,4}, \text{CONCAT}(IV^{[63..48]}, IV^{[31..16]}))$

end if

if $j = 2 \mod 4$ then

$c_{j,4} \leftarrow \text{XOR}(c_{j,4}, IV^{[63..32]})$

end if

if $j = 3 \mod 4$ then

$c_{j,4} \leftarrow \text{XOR}(c_{j,4}, \text{CONCAT}(IV^{[47..32]}, IV^{[15..0]}))$

end if

end for

\{Step-2: Iterating System by Next-State Function\ldots\}\}

for $i = 0 \rightarrow 3$ do

State$[x_{j,i+1}, c_{j,i+1}] \leftarrow \text{NEXT-STATE}(\text{State}[x_{j,i}, c_{j,i}]) \ \forall j \in \{0, \ldots, 7\}$

end for

1.2.5 Extraction Scheme

The Extraction Scheme takes the output from IV-Setup scheme whenever the later is used. Otherwise, it takes the output of Key-setup scheme as its input. In this scheme again the input state variable is iterated using Next-state function. And after each iteration the 128-bit output key-stream $s_i$ is extracted from 128-bit internal state variable i.e. $x_i$ as follows:

\begin{align*}
  s_i^{[15..0]} & = x_0^{[15..0]} \oplus x_5^{[31..16]} \\
  s_i^{[47..32]} & = x_2^{[15..0]} \oplus x_7^{[31..16]} \\
  s_i^{[79..64]} & = x_4^{[15..0]} \oplus x_1^{[31..16]} \\
  s_i^{[111..96]} & = x_6^{[15..0]} \oplus x_3^{[31..16]} \\
  s_i^{[31..16]} & = x_0^{[31..16]} \oplus x_3^{[15..0]} \\
  s_i^{[63..48]} & = x_2^{[31..16]} \oplus x_5^{[15..0]} \\
  s_i^{[95..80]} & = x_4^{[31..16]} \oplus x_7^{[15..0]} \\
  s_i^{[127..112]} & = x_6^{[31..16]} \oplus x_1^{[15..0]} \\
\end{align*}

Consequently the high-level pseudo-code has been given below: (see Algorithm 3)
1.2. Specifications of Rabbit

**Algorithm 3 Extraction Scheme**

```plaintext
for i = 0 → SIZE do
    {Iterate the system...}
    State[x,j,i+1, c,j,i+1] ← NEXT-STATE(State[x,j,i,c,j,i]) \( \forall j \in \{0, \ldots, 7\} \)
    {Generate Key-Stream as in eqn.(1.5)...}
    \( s_i \leftarrow \text{COMPUTE-KEY-STREAM}(x,j,i) \) \( \forall j \in \{0, \ldots, 7\} \)
end for
```

1.2.6 Next-State Function

Now we are on the verge of describing the most important part of this cipher which is nothing but the *Next-state Function*. Actually there are three steps which are performed in this function. First counters are updated according to the counter function, then the g-values are computed from the old state-variable and updated counter-variable. Then the state variables are updated from the newly computed g-values. For better modularity, the implementation can be thought of as the cascading call of three different functions which are doing those different tasks. The Next-state function calls the subroutine g-function which again calls the counter-updating function.

The counter-variables are updated by following equations:

\[
\begin{align*}
    c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} \mod 2^{32} \\
    c_{1,i+1} &= c_{1,i} + a_1 + \phi_{0,i+1} \mod 2^{32} \\
    c_{2,i+1} &= c_{2,i} + a_2 + \phi_{1,i+1} \mod 2^{32} \\
    c_{3,i+1} &= c_{3,i} + a_3 + \phi_{2,i+1} \mod 2^{32} \\
    c_{4,i+1} &= c_{4,i} + a_4 + \phi_{3,i+1} \mod 2^{32} \\
    c_{5,i+1} &= c_{5,i} + a_5 + \phi_{4,i+1} \mod 2^{32} \\
    c_{6,i+1} &= c_{6,i} + a_6 + \phi_{5,i+1} \mod 2^{32} \\
    c_{7,i+1} &= c_{7,i} + a_7 + \phi_{6,i+1} \mod 2^{32}
\end{align*}
\]

where the counter carry bit is given by the following equation:

\[
\phi_{j,i+1} = \begin{cases} 
    1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \text{ and } j = 0 \\
    1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \text{ and } j > 0 \\
    0 & \text{otherwise.}
\end{cases}
\]
The $a_j$ are constants having following values:

$$
\begin{align*}
    a_0 &= 0x4D34D34D \\
    a_1 &= 0xD34D34D3 \\
    a_2 &= 0x34D34D34 \\
    a_3 &= 0x4D34D34D \\
    a_4 &= 0x34D34D34D4 \\
    a_5 &= 0x34D34D34 \\
    a_6 &= 0x4D34D34D34 \\
    a_7 &= 0xD34D34D34D \\
\end{align*}
$$

In the next step, the g-values are computed with the updated counter values and the old state-variables. They are computed as:

$$
g_{j,i} = ((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)) \mod 2^{32}
$$

Finally, the internal state variables ($x'_{j,i}$) are computed as follows:

$$
\begin{align*}
    x_{0,i+1} &= g_{0,i} + (g_{7,i} \ll 16) + (g_{6,i} \ll 16) \mod 2^{32} \\
    x_{1,i+1} &= g_{1,i} + (g_{0,i} \ll 8) + g_{7,i} \mod 2^{32} \\
    x_{2,i+1} &= g_{2,i} + (g_{1,i} \ll 16) + (g_{0,i} \ll 16) \mod 2^{32} \\
    x_{3,i+1} &= g_{3,i} + (g_{2,i} \ll 8) + g_{1,i} \mod 2^{32} \\
    x_{4,i+1} &= g_{4,i} + (g_{3,i} \ll 16) + (g_{2,i} \ll 16) \mod 2^{32} \\
    x_{5,i+1} &= g_{5,i} + (g_{4,i} \ll 8) + g_{3,i} \mod 2^{32} \\
    x_{6,i+1} &= g_{6,i} + (g_{5,i} \ll 16) + (g_{4,i} \ll 16) \mod 2^{32} \\
    x_{7,i+1} &= g_{7,i} + (g_{6,i} \ll 8) + g_{5,i} \mod 2^{32} \\
\end{align*}
$$

Since the system is quite straightforward and self-illustrating, writing pseudo-code is redundant here. Instead we provide the graphical illustration below in figure 1.2 which is also given in [29].

### 1.2.7 Encryption/decryption Scheme

The extracted bits are XOR’ed with plaintext/ciphertext to encrypt/decrypt.

$$
\begin{align*}
    c_i &= p_i \oplus s_i \\
    p_i &= c_i \oplus s_i
\end{align*}
$$

where $c_i$ and $p_i$ are the $i^{th}$ 128-bit ciphertext and plaintext blocks respectively. So after adding this scheme to figure 1.1, the full block-diagram of rabbit will look like figure 1.3.
1.2. Specifications of Rabbit

Figure 1.2: Graphical Illustration of Next-State Function
Figure 1.3: Full Block-Diagram of Rabbit
1.3 Security Properties of Rabbit

Extensive security evaluations have been conducted on the Rabbit design. A full description of the results is presented in [30] and in a series of white papers, available in [46]. We summarize the security claims as follows:

- The cipher provides 128-bit security, i.e. a successful attack has to be more efficient than $2^{128}$ Rabbit trial encryptions.
- If IV is used, security for up to $2^{64}$ different IVs is provided, i.e. by requesting $2^{64}$ different IVs, the attacker does not gain an advantage over using the same IV.
- For a successful attack, the attacker has up to $2^{64}$ matching pairs of plaintext and ciphertext blocks available.

Here, we describe them briefly with adequate examples and practical demonstration in the next few subsections.

1.3.1 KEY-SETUP Properties

As explained in section 1.2.3, the Key-Setup scheme is divided into three major steps viz. Key Expansion, System Iteration & Counter Modification which are clearly shown in algorithm 1. Here we briefly describe the properties of those steps which make this scheme secure and solid.

- The Key Expansion stage guarantees a one-to-one correspondence between the key, the state and the counter, which prevents key redundancy. This can be easily observed from eqn. 1.1 and eqn. 1.2. It also distributes the key bits in an optimal way to prepare for the system iteration.

- The system iteration makes sure that after one iteration of the Next-State function, each key bit has affected all eight state variables. We provide the demonstration taking $k_0$ as example. In figure 1.4 we vividly show how this occurs. One can easily understand it with help of eqn. 1.6, eqn. 1.9 & eqn. 1.10. First we see that in Key-Expansion step, $k_0$ directly affects $x_{0,i}, x_{3,i}$ and $c_{4,i}, c_{7,i}$. When iterating the system, by eqn. 1.6, $c_{4,i+1}$ & $c_{7,i+1}$ got affected as well (this step is not explicitly shown in the diagram to avoid complications). By eqn. 1.9, one can clearly observe that $g_{0,i}, g_{3,i}, g_{4,i}$ & $g_{7,i}$ got affected when computing $g$-function. Finally, according to eqn. 1.10, every $x_{j,i+1}(\forall j = 0, \ldots, 7)$ got affected as every $g_{j,i}$ affects as many as three different $x'_{j,i+1}$s. The $x'_{j,i+1}$s which are affected by more than one $g'_{j,i}$s are shown by dashed line in the diagram. It also ensures that after two iterations of the Next-State function, all state bits are affected by all key
bits with a measured probability of 0.5. A safety margin is provided by iterating the system four times. A similar logic as figure 1.4 would apply to convince reader of these statements.

\[
\begin{align*}
  x_{0,i} & \quad x_{1,i} & \quad x_{2,i} & \quad x_{3,i} & \quad x_{4,i} & \quad x_{5,i} & \quad x_{6,i} & \quad x_{7,i} \\
  c_{0,i+1} & \quad c_{1,i+1} & \quad c_{2,i+1} & \quad c_{3,i+1} & \quad c_{4,i+1} & \quad c_{5,i+1} & \quad c_{6,i+1} & \quad c_{7,i+1} \\
  g_{0,i} & \quad g_{1,i} & \quad g_{2,i} & \quad g_{3,i} & \quad g_{4,i} & \quad g_{5,i} & \quad g_{6,i} & \quad g_{7,i} \\
  x_{0,i+1} & \quad x_{1,i+1} & \quad x_{2,i+1} & \quad x_{3,i+1} & \quad x_{4,i+1} & \quad x_{5,i+1} & \quad x_{6,i+1} & \quad x_{7,i+1}
\end{align*}
\]

**Figure 1.4: Example of key-bit affecting state variable after one iteration**

- Even if the counters are assumed to be known to the attacker, the counter modification (see eqn. 1.3) makes it hard to recover the key by inverting the counter system, as this would require additional knowledge of the state variables. It also destroys the one-to-one correspondence between key and counter, however, this should not cause a problem in practice with very high probability (will be explained later).

**Attack on the Key-Setup Function**

The highly *non-linear* dependence of both the state-bits and counter-bits upon key bits makes the attack based on partly guessing the key quite difficult. This is obvious as due to the high non-linearity, even if anyone can guess a considerable number of key-bits (say half) successfully, it would be very hard to correlate them to state-bits or counter-bits as they are complicated non-linear functions of key-bits. So, the problem of finding unknown bits from the non-linear functions, remains as hard as correctly guessing them. Furthermore, even if the counter bits were known after the counter modification, it is still hard to recover the key. Of course, knowing the counters would make other types of attacks easier.
Collision on Output

In case of Rabbit, the main concern is the non-linear map which is many-to-one. Due to this property, different keys could potentially result in the same key-stream. But, in this cipher, the Key expansion and System iteration were designed such that each key leads to unique counter values. But, the counter modification, we have discussed earlier to prevent counter recovery may result in equal counter values. Also, in this case one can verify easily that, assuming after four initial iterations, the inner state is essentially random and uncorrelated with counter system, the probability of collision is essentially given by Birthday Paradox. Which implies that, one collision is expected in 256-bit counter state for all $2^{128}$ keys. Henceforth, it can not pose a real threat.

Related-key Attack

Another possibility is related-key attack. Suppose, that a two keys $K$ and $\tilde{K}$ are related by the following relation:

$$K[i] = \tilde{K}[i+32].$$

Which implies that: $x_{j,0} = \tilde{x}_{j+2,0}$ & $c_{j,0} = \tilde{c}_{j+2,0}$. Now consider $K$ and $\tilde{K}$ are the same key. If the condition $a_{j,0} = a_{j+2,0}$ holds, then the Next-State function would preserve its property. But, one can easily verify that, the constants $a_j$’s are chosen in a way such that, $a_{j,0} \neq a_{j+2,0}$. So, the Next-State function will not preserve the related-key property. In this way the designers of Rabbit got rid of the possibility of this kind of attack.

1.3.2 IV Setup Properties

We start exploring the IV Setup Properties with the design rationale. Now, the security goal should be to justify an IV-length of 64 bits for encrypting up to $2^{64}$ plain-texts with same 128-bit key and no distinction from random bit pattern should be plausible by requesting up to $2^{64}$ IV setups. As explained in section 1.2.4, there are two stages in IV Setup routine:

- IV Addition : This is the intialization stage where it modifies the values of the counters.
- System Iteration : In this stage it calls Next-State function as many as four times.

Stage 1: IV Addition

In this stage the counter values are modified in such a way that it can be guaranteed that under an identical key, all $2^{64}$ possible different IVs will lead to unique key-streams. This obviously leads to the following observation: Each IV bit will affect the input of four different
$g$-functions in the first iteration. However, this is the maximal possible influence for a 64-bit IV.

**Stage 2 : System Iteration**

As explained in section 1.2.4, within the IV Setup routine, it calls the Next-State function four times. Essentially, it guarantees that, after one iteration, each IV bit affects all eight state variables. This is depicted in figure 1.5. Furthermore, to ensure non-linear dependence upon all IV bits, the system is iterated for three more times, four in total. Full security analysis can be found in the white papers provided by Cryptico Corp. in [6].

![Figure 1.5: Effect of IV-bits after one iteration of Next State function](image)

### 1.3.3 Partial Guessing

**Guess-and-Verify Attack**

This kind of attack is possible only when the output bits are predictable efficiently from a small set of inner-state bits. In [30], the designers of Rabbit showed the following result: Attacker must guess at least $2 \cdot 12$ input bytes for the different $g$-functions in order to verify against one byte. It is equivalent to guess 192 bits. Obviously it becomes harder than exhaustive key-search. So, from this result we can conclude that, it seems impossible to perform this attack by guessing fewer than 128 bits against output.
1.3. Security Properties of Rabbit

**Guess-and-Determine Attack**

There is another strategy of partial guessing which is known as *Guess-and-Determine Attack*. The strategy is very simple and obvious one, although the implementation is not. The main idea is to guess a few unknown bits of the cipher and from those deduce the remaining bits. Let us consider an attack scenario to make this idea more understandable. In that scenario the attacker works as follows:

- Attacker tries to reconstruct 512 bit of inner-state.
- Attacker observes 4 consecutive 128-bit cipher.
- Divide 32-bit counter and state variables into 8-bit variables.
- Construct an equation system that models state transition and output.
- Solve this equation system by guessing as few variables as possible.

Now, to analyze, we first find the efficiency of the strategy described above. We observe that, the efficiency directly depends on the number of variable guessed beforehand. Also, we see that the efficiency is lower-bounded by subsystems with smallest number of input variables affecting one output bit. In [30], the designers showed that, each byte of the next-state function depends on 12 input bytes (neglecting counters) or 24-bytes (including counters). From this result, we can conclude that, the attacker must guess more than 128 bits beforehand which is obviously no easier than the exhaustive key search. So, the designers of Rabbit proved that, this kind of attack is rather infeasible against Rabbit.

### 1.3.4 Algebraic Analysis

Algebraic Analysis is among the not-so-explored area of analysis in the literature. But, this technique may be fruitful against ciphers whose internal state is mainly updated in linear way (mainly LFSR-based). This kind of analysis has been discussed in detail in [4, 42, 43, 44]. But, while ciphers like Rabbit is concerned, in which the internal State bits updated in strongly non-linear fashion, then this kind of attacks are found to of no use. Still in [29], the algebraic analysis has been provided in details because of mainly two reasons. First, algebraic attacks are newly invented and it is not yet known to the cryptanalysts that, how (or at all) does it work. And secondly, it is yet unclear that, which properties of a cipher determine the resistance against algebraic attacks.
The Algebraic Normal Form

First we start with the definition of the Algebraic Normal Form (abbr. ANF). The algebraic normal form of a boolean function is defined as follows: Let \( f \equiv f(x_0, x_1, \ldots, x_n) \) from \( \{0,1\}^n \) to \( \{0,1\} \) be a boolean function. Then the ANF is given by:

\[
f(x_0, x_1, \ldots, x_n) = \sum_{u \in \{0,1\}^n} a_u \prod_{i=0}^{n-1} x_i^{u_i}
\]

where, \( u \equiv (u_0, u_1, \ldots, u_n) \in \{0,1\}^n \) and \( a_u \in \{0,1\} \) is given by:

\[
a_u = \sum_{\{x : x \land u = 0\}} f(x). \text{(Möbius Transform)}
\]

Also, we define another important think i.e. the degree of a monomial \( \prod_{i=0}^{n-1} x_i^{u_i} \) which is given by \( H(u) \) (Where \( H(u) \) is the hamming weight of \( u \)).

ANF of the g-Function

While analyzing the ANF of the g-functions which are responsible for the non-linearity of the cipher, it must be kept in mind that, one of the most important properties of non-linear building blocks is that, they have large number of monomials and good distribution of their degrees. We want to see how well that property holds for Rabbit. Full Analysis of algebraic properties of Rabbit can be found in [9]. Now, analysis shows that, in case of 32-bit random function the following property holds:

- **Average total no. of monomials**: \( 2^{31} \).
- **Average total no. of monomials including one given variable**: \( 2^{30} \)
- **Average no. of monomials not present in any 32-bit Random functions**: \( 1 \).

Importantly, the variance of the monomials not present in any 32-bit Random Function is also equal to 1. That implies the distribution is well concentrated near the average. Now, for the g-Functions of Rabbit, the analysis of the ANF of 32 boolean sub-function shows the following properties:

- **Algebraic degree** : At least 30.
- **No. of monomials** : \( 2^{24.5} \) to \( 2^{30.9} \).

The distribution of monomials are shown in figure 1.6 In figure 1.6, the smooth dark line
signifies the average random behavior and the margin including the standard deviation is shown by the dotted lines. From the figure, we observe that, some boolean function deviate significantly from the random case. Also, we can see that, all the boolean functions have a large number of monomials of high degree. Thorough analysis shows that, the total number of monomials that only occur once in the g-function is $2^{26.03}$ and the number of monomials do not occur at all is equal to $2^{26.2}$. Now, compare to random function, this is clearly a pretty large deviation. So, this information can be useful to analysis rabbit in future when the theory of Algebraic Analysis will evolve much.

The ANF of Full Cipher

Obviously due to the large key size the full analysis becomes infeasible with today’s computational power. So, the experimentation has to be done on some scaled-down version. The 8-bit version has been analyzed. More precisely, 32 output boolean functions have been studied as functions of 32-bit key. The ANFs are determined after 0,1,2,3,4 iterations and one extra in extraction. For, $0 + 1$ iteration, no. of monomials are found to be $\approx 2^{31}$. But, the result stabilizes after two iterations. That means, there are no more fluctuations in the output. The number of missing polynomials are 0, 1, 2, 3, 1 in iterations $(0+1)$, $\ldots$, $(4+1)$. Clearly the
data is in accordance with random function. Therefore, after a few iterations, all possible monomials are found in the output ANFs.

**Over-defined Equation Systems in the State**

Another strategy of algebraic analysis is to analyze the systems of equations in the state. Obviously, the system would be over-determined one. Also, to make the analysis feasible some simplistic assumption has been made. Throughout the analysis, the presence of counters has been ignored. Also, rotations are removed and additions are replaced by XOR. These assumptions make sense as considering all these factors, the system would likely to behave in the same way without them with high probability. Now, the inner state of Rabbit consists of 256 bits. That implies, we need output of at least two iterations to get non-linear system of 256 equations in 256 variables. Also, we observe that, everything except g-function is linear. So, we can calculate the number of monomials by the following method. The output of the first iteration is modeled as linear function. From this, 128 simple linear equations are obtained. These equations contain all 256 monomials of degree one. From the next iteration, the inner state bits are run through the g-functions. Experiments showed that, these equations contain

\[ 8 \cdot (2^{32} - 2^{26.2}) \approx 2^{34.97} \]  

monomials.

Now, coming to the non-linear system of equations, they are found to be very dense and also of much higher degree. It increases the number of variables to \( \approx 2^{35} \) which means \( 2^{35} - 2^8 \) extra equations are required to solve. Again, using further iteration would increase the number of monomials beyond \( 2^{128} \). Clearly, extra equations can not be obtained. So, we can conclude that, it is not feasible to successfully mount *Algebraic Attack* using inner state bits as variables.

**Over-defined Equation Systems in the Key**

Another possibility of algebraic attack is using the over-determined equation system in the Key. But, it turns out to be even more difficult than the states due to the presence of at least 5 round (9 if IV used) iterations of non-linear layer before first output. This observation leads to the fact that, ANF of full cipher has to be considered. From the earlier discussion, we recall that, after two iterations, ANF becomes indistinguishable from Random function *i.e.* no. of monomials becomes close to \( 2^{128} \). Eventually, we conclude that, solving such system is much harder than the brute-force key-search.

### 1.3.5 Correlation Analysis

*Correlation Attacks* are a special type of cryptanalysis in which the attacker tries to find the dependence of output bits on the input bits. Basically, the main target is to find the
correlation factor of output variables and input variables. There are a few variations \textit{viz. linear approximation} and \textit{second order approximation}.

**Linear Approximation**

The \textit{Linear Attack} attempts to find the best linear approximations between bits in input to the next-state function and extracted output. To achieve this, a special technique called \textit{Walsh-Hadamard Transform} is used assuming all inputs are linearly independent. Analysis of \(g\)-function gives the correlation coefficients of the cipher. It has been found that, the best linear approximation has correlation coefficient \(\approx 2^{-57.8}\) which implies, output from \(2^{114}\) iterations must be generated to distinguish from random function. The full analysis is provided in [95]. At the same time we comment that, the attack seems to be unlikely as it requires many large and usable correlation, only the best one is not sufficient.

**Second Order Approximation**

In this kind of approximation, the terms which has higher order than 2 are truncated. Suppose, the approximation of \(g^{[j]}\) is \(f^{[j]}\). The experimental results of second order approximation can be summarized as follows:

- Correlation Coefficient between \(f^{[j]}\) and \(g^{[j]}\) is less than \(2^{-9.5}\).
- Correlation Coefficient between \(f^{[j]} \oplus f^{[j+1]}\) and \(g^{[j]} \oplus g^{[j+1]}\) is \(\approx 2^{-2.72}\).
- Indicates that some higher order term vanishes when neighboring bits are XOR'ed.
- Constructing approximation from the results gives best one with correlation coefficient of \(2^{-26.4}\).

From the above result, after thorough observation, we can conclude that, it is not trivial to use second order approximation in linear analysis. Moreover, the analysis becomes very complex after including counter.

**1.3.6 Differential Analysis**

In a broad sense, \textit{Differential Analysis} is the study of how differences in an input can affect the resultant difference at the output. We first describe the principle and then go into more details about what happens in case of \textit{Rabbit}. First assume that, there exist two inputs \(x'\) and \(x''\) and their corresponding outputs \(y'\) and \(y''\) (all \(\in \{0, 1\}^n\)). Two different schemes are used. First one uses modulo subtraction where input difference and output difference are
defined by $\Delta x = x' - x'' (\text{mod} 2^n)$ and $\Delta y = y' - y'' (\text{mod} 2^n)$ respectively. Another one uses XOR difference where input difference and output difference are defined by $\Delta x = x' \oplus x''$ and $\Delta y = y' \oplus y''$ respectively.

**Differential of g-Function**

In [5], the full differential analysis has been made. We discuss the main points here. Ideally, all $2^{64}$ possible differentials should be analyzed which is clearly not feasible. Instead, smaller versions viz. 8, 10, 12, 14, 16 & 18-bit g-functions were considered in [5] to make the analysis feasible. Using the XOR difference, it has been found that, $\Delta x$ is characterized by a block of ones of size approx $\frac{3}{4}$th word length. Based on this observation, all input differences constituted by single blocks of ones were considered. Experiments showed the best result: Largest probability found = $2^{-11.57}$ for differential (0x007FFFFE, 0xFF001FFF).

In case of subtraction modulus difference, no such clear structure were observed. However, thorough observation showed that, the probabilities scale nicely with corresponding word-length. Assuming this scaling continues, which is the most likely fact, the differential with largest probability was expected to be of the order $2^{-17}$. Evidently, it is significantly lower compare to XOR difference. Also, higher order differentials were observed briefly. But severe problem arrived with higher order: Complexity went beyond computational power.

**Differential of Full Cipher**

In [9], the differential of Full Cipher was analyzed as a part of Algebraic Analysis. Observation says that, any characteristic will involve at least 8 g-functions. So, the transition matrices from smaller g-functions are analyzed. Experiments showed that, after four iterations, a steady state distribution of matrix element found for both schemes. The probability of best characteristic was found to be $\ll 2^{-64}$. Therefore, we can not expect any exploitable differential. Again for simplified version of Rabbit, higher order differential can be used to break IV Set-Up even for large number of iteration. For another simplified version including rotation, 3rd order differential were still found to has a high probability for one round. But, for more iteration, security increases quickly. Finally the use of XOR in g-function destroys applicability of higher order differential.

**1.3.7 Statistical Test**

Statistical Test is another important test for distinguishing a Pseudo-random generator from random generator. This tests are obviously not sufficient, but necessary. So, the designers of Rabbit performed different statistical tests like following:
1.3. Security Properties of Rabbit

(i) NIST Test Suit [107].

(ii) DIEHARD battery of tests [94]

(iii) ENT Test [123]

These tests are performed on internal state and extracted output. Even after conducting various tests on Key-Set-Up function, full cipher (8-bit version), no weakness has been found.

1.3.8 Mod $n$ Analysis

Mod $n$ Analysis is a form of partitioning cryptanalysis that exploits unevenness in how the cipher operates over equivalence classes (congruence classes) modulo $n$. The method was first suggested in 1999 by John Kelsey, Bruce Schneier, and David Wagner in [80]. Most importantly it is applicable to the cipher with bit rotation. Evidently, Rabbit is a candidate. The detail analysis is provided in [7]. The attack is based on the observation: $(x \ll 1) - 2x = 0 (\text{mod } n)$ iff $n = 3^a 5^b 17^c 257^d 65537^e$ when $x$ is 32-bit. ($a,b,c,d,e = 0/1$). It is applied to look for two types of attacks:

(i) Key Recovery.

(ii) Bias of output.

Key Recovery

As we have discussed in section 1.2, we can see that, besides rotation Rabbit uses operations like right-shift, addition modulo $2^{32}$, squaring and XOR. In [7], it has been observed that, there is no value of $n$ for which it is possible to analyze all the operations used in the state update of Rabbit. Eventually it leads to the conclusion that, it is impossible to construct ‘mod $n$’ model which implies that, we can not derive information about the internal state of the cipher.

Output Bias

Suppose, $C_i = \sum_j x_{j,i} (\text{mod } 3)$ & $G_i = \sum_j x_{j,i} (\text{mod } 2^{32})$. In [7], thorough experimentations shows the following results:

$$C_i = \begin{cases} 
0 & \text{with probability } 0.3324 \\
1 & \text{with probability } 0.3352 \\
2 & \text{with probability } 0.3324 
\end{cases}$$
and,

\[
G_i = \begin{cases} 
0 & \text{with probability 0.3337} \\
1 & \text{with probability 0.3337} \\
2 & \text{with probability 0.3326}
\end{cases}
\]

So, clearly they found non-uniformities in the distribution which could be analyzed thoroughly.

Now, if the bias remains noticeable after the output extraction then it is vulnerable to attack. But, since the property does not depend in any way on the value of the key, it is not possible to recover the key from it. Also, \( G_i \) \& \( G_i \mod 3 \) are not visible at output. A necessary condition in order to be able to see something at the output, is that there must be a link between the following output distributions:

(i) \( A(x_j) = \sum_j (x_{j,h}2^{16} + x_{j,l}) \mod 2^{32} \mod 3 \).

(ii) \( B(x_j) = (x_{j,h} \oplus x_{j+3,l}) \forall j \in \{0, \ldots, 7\} \).

But, the experimental results, showed in [7], says that the distributions are independent.

1.3.9 Period Length

Period length is obviously one important feature of the cipher. Most importantly, for stream cipher, the central characteristic is that, the exact lower bound can be provided. The counter system period length was found to be equal to \( 2^{256} - 1 \). Also, it has been proved that, the input to the g-functions has at least the same period. From this result, it follows that, a very pessimistic lower bound of \( 2^{215} \) can be guaranteed on the period of the state variable.

1.4 Cryptanalysis of Rabbit

In this section, we discuss few other results on cryptanalysis of Rabbit published recently.

1.4.1 On a Bias of Rabbit

This result has been published in 2007 by Aumasson. He analyzed mainly the g-function. In this paper several properties of Rabbit g-function has been observed and proved. He took scaled down versions viz. 8-bit, 16-bit as well as full 32-bit. Several biases of g-function has been shown both for 1-bit and n-bit pattern. Also the corresponding bias in key-stream has been calculated \( \approx 2^{-123.5} \). He concluded that, although g-function is strongly unbalanced (bias \( > 2^{-124.5} \)), the distinguisher requires \( 2^{247} \) 128-bit sample key-stream derived from random
keys, IVs. So the complexity of this attack is found to be much higher than exhaustive search. Clearly, these imbalance can not pose a real threat to Rabbit. Detail analysis can be found in [10].

1.4.2 Another Cryptanalysis based on FFT

After Aumasson’s work exact bias of Rabbit sub-block has been computed using Fast Fourier Transform (abbrv. FFT) method by Ling, Wang, Lu in 2008. Their work showed the best distinguishing attack with complexity $2^{158} \ll 2^{247}$. This work is an excellent endeavor as assuming knowledge of the relation between part of the internal states, this distinguishing attack can be extended to a key-recovery attack. Yet it remains an open challenge to further improve the distinguishing attack to complexity below $2^{128}$ which would be considered a true attack. Detail analysis can be found in the original paper ([91]).

1.4.3 Differential Fault Analysis of Rabbit

The differential fault analysis is one type of newly emerged side-channel analysis. This type of attack on Rabbit is first proposed in [82] very recently. They have used the following fault model: The attacker is assumed to be able to fault a random bit of the internal state of the cipher but cannot control the exact location of injected faults. Experiments showed that, it requires around 128–256 faults, precomputed table of size $2^{41.6}$ bytes and recovers the complete internal state of Rabbit in about $2^{38}$ steps. Detail analysis can be found in the original paper ([82]).

1.5 Performance Evaluations of Rabbit

Performance analysis is very important for practical use of any cipher. In the white papers ([8]), performance analysis is provided on various platforms viz. Pentium III, Pentium 4, PowerPC, ARM7 and MIPS 4Kc processors. A few important assumption has been made for this purpose: During the tests all data blocks (i.e. instance, data, key and iv) are 16-byte aligned. Furthermore, we assume that the size of the plaintext/ciphertext is a multiple of 16 bytes. All Rabbit functions were implemented with a standard C interface as described in [8]. Performance was measured by reading the processor clock counter before and after calling the procedure to be measured. While measuring memory, it should be kept in mind that, the presented memory requirements show the amount of memory allocated on the stack due to the calling convention (function arguments, return address and preserved registers) and temporary data. And, another thing is that, The code size includes the entire function i.e.
in addition to the algorithm itself it includes the function prolog and epilog. Next we explore performances analysis briefly in different platforms. We mostly use tabular representation for the convenience of the reader.

1.5.1 Intel Platforms

In Intel platforms, the performances were measured by running a speed-optimized version of Rabbit programmed in assembly language in-lined in C using MMX instructions. Intel C++ 7.0 compiler was used. The clock ticks are read just before and just after a call to the encryption or setup function as follows:

(i) \[
\text{start} = \text{read
clock	ick();}
\text{end} = \text{read
clock	ick();}
\text{overhead} = \text{end} - \text{start;}
\]

(ii) \[
\text{start} = \text{read
clock	ick();}
\text{key_setu\pt(...);}\]
\text{end} = \text{read
clock	ick();}
\text{key_setup_time} = \text{end} - \text{start} - \text{overhead;}

Pentium III Performances

Platform Specs:

(i) Desktop PC; 1.0 GHz; Intel 82815 chipset; Windows 2000.

(ii) Laptop ; 850 MHz; ALI M1621/M1533 chipset; Windows Me.

Performance Table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size</th>
<th>Memory</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Setup</td>
<td>794 bytes</td>
<td>32 bytes</td>
<td>307 cycles</td>
</tr>
<tr>
<td>IV-Setup</td>
<td>727 bytes</td>
<td>36 bytes</td>
<td>293 cycles</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>717 bytes</td>
<td>36 bytes</td>
<td>3.7 cycles/byte</td>
</tr>
<tr>
<td>PRNG</td>
<td>699 bytes</td>
<td>32 bytes</td>
<td>3.8 cycles/byte</td>
</tr>
</tbody>
</table>

Pentium 4 Performances

Platform Specs:

- Desktop PC; 1.7 GHz; Intel 82850 chipset; Windows 2000.
1.5. Performance Evaluations of Rabbit

Performance Table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size</th>
<th>Memory</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Setup</td>
<td>698 bytes</td>
<td>16 bytes</td>
<td>468 cycles</td>
</tr>
<tr>
<td>IV-Setup</td>
<td>688 bytes</td>
<td>20 bytes</td>
<td>428 cycles</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>762 bytes</td>
<td>28 bytes</td>
<td>5.1 cycles/byte</td>
</tr>
<tr>
<td>PRNG</td>
<td>710 bytes</td>
<td>24 bytes</td>
<td>5.2 cycles/byte</td>
</tr>
</tbody>
</table>

1.5.2 Power PC Platform

For Power PC, the task was complicated. Because, The register, holding the processor cycle count, is only accessible in kernel mode. So, to get rid of that, measurements had to be done based on the clock signal provided to the processor by the evaluation board. It is also important to note that, measuring 25 MHz clock ticks from the board instead of the actual 533 MHz processor clock causes lack of precision.

Power PC Performances

Platform Specs:

- 533 MHz PowerPC system.

Performance Table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size</th>
<th>Memory</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Setup</td>
<td>512 bytes</td>
<td>72 bytes</td>
<td>405 cycles</td>
</tr>
<tr>
<td>IV-Setup</td>
<td>444 bytes</td>
<td>72 bytes</td>
<td>298 cycles</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>440 bytes</td>
<td>72 bytes</td>
<td>3.8 cycles/byte</td>
</tr>
</tbody>
</table>

1.5.3 ARM7 Platform

Performance evaluation was also done using the ARMulator integrated in ARM Developer Suite 1.2. Similar to the approach which has been taken towards Pentium, here also timing values were obtained using clock() before and after calling the function in question. However, performance was measured in a simulated environment and thus may differ in real applications on a device using an ARM7 processor. But, at the same time, simplicity of Rabbit suggests minimal deviation. Another important point to note is that, here performance has been measured encrypting 4096 bytes of data.
ARM7 Performances

Platform Specs:

• ARMulator integrated in ARM Developers Suite 1.2.

Performance Table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size</th>
<th>Memory</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Setup</td>
<td>436 bytes</td>
<td>80 bytes</td>
<td>610 cycles</td>
</tr>
<tr>
<td>IV-Setup</td>
<td>408 bytes</td>
<td>80 bytes</td>
<td>624 cycles</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>368 bytes</td>
<td>48 bytes</td>
<td>9.58 cycles/byte</td>
</tr>
</tbody>
</table>

1.5.4 MIPS 4Kc Platform

To measure performances, assembly language versions of Rabbit has been written for the MIPS 4Kc processor. The platform was developed using the Embedded Linux Development Kit (ELDK), which includes GNU cross-development tools. The codes were written for little-endian as well as big-endian memory organization. However, techniques similar to Intel processors were used.

Little Endian Performances

Platform Specs:

• 150 MHz with a Linux operating system.

Performance Table:

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size</th>
<th>Memory</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Setup</td>
<td>856 bytes</td>
<td>32 bytes</td>
<td>749 cycles</td>
</tr>
<tr>
<td>IV-Setup</td>
<td>816 bytes</td>
<td>32 bytes</td>
<td>749 cycles</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>892 bytes</td>
<td>40 bytes</td>
<td>10.9 cycles/byte</td>
</tr>
</tbody>
</table>

Big Endian Performances

Platform Specs:

• 150 MHz with a Linux operating system.

Performance Table:
1.5. Performance Evaluations of Rabbit

<table>
<thead>
<tr>
<th>Function</th>
<th>Code Size</th>
<th>Memory</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key-Setup</td>
<td>960 bytes</td>
<td>32 bytes</td>
<td>749 cycles</td>
</tr>
<tr>
<td>IV-Setup</td>
<td>888 bytes</td>
<td>32 bytes</td>
<td>749 cycles</td>
</tr>
<tr>
<td>Encrypt/Decrypt</td>
<td>1052 bytes</td>
<td>40 bytes</td>
<td>13.5 cycles/byte</td>
</tr>
</tbody>
</table>

1.5.5 Hardware Performances

The hardware performance plays an important role in hardware specific applications. The simple structure and compact design of Rabbit are responsible for an excellent hardware performance in various platforms. We provide measurements from two different perspective:

(i) Area Optimized Performance.

(ii) Speed Optimized Performance.

Platform Specs:

- 0.18 µm CMOS.

Area Optimized Performance

Performance Table:

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Gate Count</th>
<th>Die Area</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>no CLA</td>
<td>3.8 K</td>
<td>0.044 mm²</td>
<td>88 Mbit/s</td>
</tr>
<tr>
<td>w/ CLA</td>
<td>4.1 K</td>
<td>0.048 mm²</td>
<td>500 Mbit/s</td>
</tr>
</tbody>
</table>

Speed Optimized Performance

For speed optimized case, it should be noted that, the multiplication consumes most time. All the other operations are done bitwise which are really very fast.

Performance Table:

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Gate Count</th>
<th>Die Area</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>28 K</td>
<td>0.32 mm²</td>
<td>3.7 Gbit/s</td>
</tr>
<tr>
<td>2</td>
<td>35 K</td>
<td>0.40 mm²</td>
<td>6.2 Gbit/s</td>
</tr>
<tr>
<td>4</td>
<td>57 K</td>
<td>0.66 mm²</td>
<td>9.3 Gbit/s</td>
</tr>
<tr>
<td>8</td>
<td>100K</td>
<td>1.16 mm²</td>
<td>12.4 Gbit/s</td>
</tr>
</tbody>
</table>
1.6 Strength and Advantages

In this section, we analyze Rabbit from more practical view point avoiding technical details mostly. Rabbit was a stream cipher with a complete new type of design. While, prior to Rabbit most of the stream ciphers were based on LFSR or S-box and thus vulnerable to different attacks due to the linear structure, it came as a cipher which does not need LFSR or S-box at all. Due to the presence of modular squaring in the Next-State function, it provides strong non-linear mixing of the inner state between two iterations. The main strength of Rabbit are as follows:

(i) Compact Design.

(ii) High Security.

1.6.1 Compact Design

One of the most important and useful feature of Rabbit is its compact design. All the arithmetic operations involved in the design are provided by modern processor. So it is entirely platform-independent. Also, it shows high speed performances on various platforms. For the same reason, the gate count remains low which is very useful in case of hardware specific operations. Since all the arithmetics in Rabbit are done in $GF(2^n)$, no look-up table is required which keeps memory requirement lower. The only thing it needs to store in the memory is the copy of inner state which can be easily accommodated in registers. Henceforth, the program can access them very fast.

1.6.2 High Security

As we have discussed thoroughly various security aspects of rabbit in section 1.3, we can say that, the design makes most wide-spread stream cipher attacks inapplicable. Summarizing, due to its high non-linearity, it prevents both linear and algebraic attack. In fact, it was evaluated against almost all known attacks by designers and others. Also, all the optimization were done carefully to avoid any possible weakness.

1.7 A few remarks about design

As the design were already discussed thoroughly in section 1.2, we briefly mention a few important points which makes it useful and distinct from other ciphers. Notice that, the core function of the cipher is $\{0,1\}^{32} \rightarrow \{0,1\}^{64}$ i.e. the squaring function. Now, due to squaring
the output size become 64-bit. Therefore, it was necessary to reduce this to 32-bit to maintain consistency. Now the obvious question arises: How to reduce the output to 32-bit? There were three options viz.

(i) To take only the higher 32 bits.

(ii) To take the middle 32 bits.

(iii) To take the XOR between higher 32 bits and lower 32 bits.

The third option was taken as it was found that, other two provides much more correlation coefficients. The obvious target to prevent correlation attack is to reduce the correlation between input bits and output bits.

1.7.1 Design of Counter Systems

While the counter system were being designed, it was observed that, updating inner states in non-linear fashion would result in unpredictable period length. So, to solve this problem, counters were added to inner states before running g-function. Again, it was found that, standard counter construction would result predictable bit pattern. To fix this, a simple and elegant solution was incorporated that is, carry feedback was used to destroy predictability. Also, the weak choice of the constant $A$ containing large strings of 0/1’s would make them vulnerable. So the $A$ are chosen by repeating 110100.

1.7.2 Symmetry and Mixing

As anyone can easily check out that, the design was made as symmetric as possible. Again, to prevent the attacker to decomposition, every block was constructed to provide maximum mixing. Moreover, most secure rotations are chosen after all possible rotations were analyzed to identify maximum mixing.

1.8 Conclusion

After all the discussion, in conclusion it can be confidently said that, Rabbit is a very strong cipher. In fact, among all the cipher who are the member of software profile of eSTREAM, Rabbit has the minimum number of attacks till date. However, current best attacks are much worse than the exhaustive search. Evidently, they can not pose any real threat. Also, it leaves room for researcher to work on the cryptanalysis of it in future. In future, with more development on algebraic attack, the algebraic weakness of g-function can be exploited. Also,
there could be endeavor to find output bias with reduced size or good differentials. Also, the excellent performance of Rabbit in various platforms makes it a really strong and practically usable stream cipher.

1.9 A Simple C Implementation of Rabbit

We present a simple C code of rabbit here. This is provided to make the reader understand the basic implementation. This is not optimized one. So this code can not be used for practical purpose. For the practically usable optimized code the reader must look into the codes submitted in the eSTREAM official portal at [58].

```c
/*Developer : Pratyay Mukherjee
   Email: pratyay85@gmail.com
*/

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
#define ROUND 3 /*Changing this value would result change in the number of output bytes*/
#define bigint unsigned long long int

/*show() takes an array of bigint as input and display them */
void show(bigint*elt, int I)
{
    int i,j;
    puts(""");
    for(i=0;i<I;++i)
        printf("%x\n",elt[i]);
    puts(""),
}
```
/*bintodec() converts a binary integer of some specified length into decimal */
long long unsigned int bintodec(unsigned char *bin, unsigned int size)
{
    int i,j,k;
    unsigned long long int temp=0;
    for(i=0;i<size;++i)
    {
        temp = 2*temp + bin[i];
    }
    return temp;
}
/*dectohex() converts a decimal int to a 8-digit hexadecimal int as a char array */
unsigned char* dectohex(unsigned long long int dec)
{
    int i=0,j,k,hh;
    unsigned char* hex,tt;
    unsigned long long int temp = dec;
    hex=(char*)malloc(sizeof(char)*8);
    for(i=0;i<8;++i)
    {
        hex[i]='0';
    }
    j=0;
    // It uses the standard conversion routine i.e. dividing by base=16 and
    // record the remainder
    while((temp >0)&&(j<8))
    {
        hh = temp%16;
        if((hh>=0)&(hh<=9))
            hex[j++]='0'+hh;
        else if((hh>=10)&(hh<=15))
            hex[j++]='A'+hh-10;
        else
        {
            puts("Something wrong computing HEXCODE");
            exit(1);
        }
        temp=temp/16;
    }
    for(i=0;i<4;++i)
    {

tt=hex[i];
hex[i]=hex[7-i];
hex[7-i]=tt;
}
return hex;
}

/*dectobin() is quite similar to dectohex() but much simpler. It converts a decimal
to binary form (char array) of specified bit*/
void dectobin(unsigned char* bin, unsigned long long int dec, int size)
{
    int i=0, j, k;
    unsigned char tt;
    unsigned long long int temp = dec;
    for (i=0; i<size; ++i)
        bin[i]=0;
    j=0;
    while (((temp >0) && (j<size))
    {
        bin[j++] = (unsigned char) temp%2;
        temp=temp/2;
    }
    for (i=0; i<(size/2); ++i)
    {
        tt=bin[i];
        bin[i]=bin[size-1-i];
        bin[size-1-i]=tt;
    }
}

/*show_binary() takes big binary integer as 2-d array form and display them in
   binary and corresponding hexadecimal and decimal form*/
void show_binary(unsigned char** elt, int I, int J)
{
    int i, j;
    unsigned char* hex_temp;
    hex_temp=(unsigned char*)malloc(sizeof(unsigned char)*(J/4));

    for (i=0; i<I; ++i)
    {
        for (j=0; j<J; ++j)
        {

printf("%d",elt[i][j]);
}
printf(" ");
}
puts(" ");

for(i=0;i<I;i++)
{
    hex_temp=dectohex(bintodec(elt[i],J));
    printf("%s	%lld\n",hex_temp,bintodec(elt[i],J));
}
puts(" ");

display_bytewise(bigint B, unsigned int size)
{
    int i,j,k,l;
    unsigned char*temp_hex;
    j=size/8;
    temp_hex=dectohex(B);
    for(k=0;k<j;++k)
    {
        for(l=0;l<2;++l)
        {
            printf("%c",temp_hex[2*(3-k)+l]);
        }
        printf(" ");
    }
    puts(" ");
}

/*copy_bigint() copies a big integer to another (in array form)*/
void copy_bigint(bigint*s,bigint*t,int size)
{
    int i;
    for(i=0;i<size;++i)
    s[i]=t[i];
}

/*initialize() sets 0 to all bits of a large binary number (in 2-d array form)*/
void initialize(unsigned char**elt,int I,int J)
{
```c
int i, j;
for (i = 0; i < I; ++i)
{
    for (j = 0; j < J; ++j)
    {
        elt[i][j] = 0;
    }
}

/* lrot_dec() rotates a 32-bit decimal number left by specified position */
bigint lrot_dec(bigint var, unsigned int rot)
{
    int i, j, k;
    bigint t1, t2;
    unsigned char* temp, *result, *temp1;

    t1 = (var << rot) & (Modulus - 1);

    t2 = (var >> (32 - rot)) & ((Modulus - 1) >> (32 - rot));

    return (t1 + t2) % Modulus;
}

/* add2() takes two 32*8 bit big binary numbers and add them and also output the boolean carry */
int add2(bigint* v1, bigint* v2, unsigned long long int carry_in, bigint* result)
{
    int i, j, k;
    unsigned char temp, temp_cy = 0;
    unsigned long long int cy_dec = 0;
    unsigned long long int var_dec = 0;
    cy_dec = carry_in;
    for (i = 0; i < 8; ++i)
    {
        var_dec = v1[i] + v2[i] + cy_dec;
        if (var_dec >= Modulus)
            cy_dec = 1;
        else
            cy_dec = 0;
    }
```

```c
var_dec = var_dec % Modulus;
result[i] = var_dec;

return cy_dec;

/*compute_g() computes the g() values and update the global variable G according to
 the specified equation*/
void compute_g()
{
  int i,j,k;
  unsigned long long int var1;
  unsigned long long int temp1=0,temp2=0,temp3=0,temp4=0,aux=0;
  unsigned char* temp;
  temp=(char*)malloc(sizeof(char)*32);

  for(i=0;i<8;++i)
  {
    temp2 = X[i];
    temp1 = temp2 + C[i];
    temp1 = temp1 % Modulus;
    temp2=temp1*temp1;
    temp3 = temp2 >> 32;
    temp2 = temp2 % Modulus;
    temp4 = temp2^temp3;
    temp4 = temp4 % Modulus;
    G[i]=temp4;
  }
}

/*NEXTSTATE() Update the global state variable X and counter C according to
equations*/
void NEXTSTATE()
{
  bigint*c_prev;
  int i,j;
  unsigned long long int k=0;
  unsigned char* temp1,*temp2;
  c_prev=(bigint*)malloc(sizeof(unsigned char)*8);
  temp1=(unsigned char*)malloc(sizeof(unsigned char)*32);
  temp2=(unsigned char*)malloc(sizeof(unsigned char)*32);
  copy_bigint(c_prev,C,8);
```
/*Update Counters : Add present counter value to constant A to store the result in C*/
k=add2(c_prev,A, cy, C);
cy=k; /*Sets the global carry cy*/

/*Update the variable G*/
compute_g();

/*Update X variable with newly calculated G and C values*/
X[0]= G[0] + lrot_dec(G[7],16) + lrot_dec(G[6],16);
X[1]= G[1] + lrot_dec(G[0],8) + G[7];
X[2]= G[2] + lrot_dec(G[1],16) + lrot_dec(G[0],16);
X[4]= G[4] + lrot_dec(G[3],16) + lrot_dec(G[2],16);
X[6]= G[6] + lrot_dec(G[5],16) + lrot_dec(G[4],16);

for(i=0;i<8;++i)
{
    X[i]=X[i] % Modulus ; /*All additions are mod 2^32*/
}

*/KEYSETUP() computes the initial X (global) and C (global) values with given Key (global)*/
void KEYSETUP()
{
    unsigned char **x,**c,**key; /*Temporary binary representation of corresponding variables*/
    int i,j;
    x=(unsigned char**)malloc(sizeof(unsigned char*)*8);
    for(i=0;i<8;++i)
    {
        x[i]=(unsigned char*)malloc(sizeof(unsigned char)*32);
    }
    initialize(x,8,32);
    c=(unsigned char**)malloc(sizeof(unsigned char*)*8);
    for(i=0;i<8;++i)
    {
        c[i]=(unsigned char*)malloc(sizeof(unsigned char)*32);
    }
1.9. A Simple C Implementation of Rabbit

```c
key=(unsigned char**)malloc(sizeof(unsigned char)*8);
for(i=0;i<8;++i)
{
    key[i]=(unsigned char*)malloc(sizeof(unsigned char)*16);
}
for(i=0;i<8;++i)
{
    dectobin(key[i],Key[i],16); /*Converting the global key (decimal) to
    local key (binary)*/
}
/*Initializing binary state variable x by key following specified equations*/
for(i=0;i<8;++i)
{
    if(i%2==0)
    {
        for(j=0;j<16;++j)
            x[i][j]=key[(i+1)%8][j];
        for(j=16;j<32;++j)
            x[i][j]=key[i][j-16];
    }
    else
    {
        for(j=0;j<16;++j)
            x[i][j]=key[(i+5)%8][j];
        for(j=16;j<32;++j)
            x[i][j]=key[(i+4)%8][j-16];
    }
    X[i]=bintodec(x[i],32); /*Converting local x (binary) to global X
    (decimal)*/
}
/*Initializing binary counter variable c by key following specified
   equations*/
for(i=0;i<8;++i)
{
    if(i%2==0)
    {
        for(j=0;j<16;++j)
            c[i][j]=key[(i+4)%8][j];
        for(j=16;j<32;++j)
            c[i][j]=key[(i+3)%8][j];
    }
    else
    {
        for(j=0;j<16;++j)
            c[i][j]=key[i][j-16];
        for(j=16;j<32;++j)
            c[i][j]=key[(i+5)%8][j-16];
    }
}
```

c[i][j]=key[(i+5)%8][j-16];
}
else
{
    for(j=0;j<16;++j)
        c[i][j]=key[i][j];
    for(j=16;j<32;++j)
        c[i][j]=key[(i+1)%8][j-16];
}
C[i]=bintodec(c[i],32); /*Converting local c (binary) to global C (decimal)*/

/*Iterating the system by calling NEXTSTATE() 4 times*/
for(i=0;i<4;++i)
{
    NEXTSTATE();
}

/*Updating the counter again to get rid of the possibility of recovering key by knowing counter*/
for(i=0;i<8;++i)
{
    C[i] ^= X[(i+4)%8];
}

/*KEYGEN() provides the user interface to choose between the given key. */
void KEYGEN()
{
    int i,j,k;
    unsigned long long int temp_key[8];

    /*Three different keys are hardcoded. User may change accordingly*/
    unsigned long long int key1[8] = { 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000 };
    unsigned long long int key2[8] = { 0xc3ac, 0xdc51, 0x62f1, 0x3bfc, 0x36fe, 0x2e3d, 0x1329, 0x9128 };
    unsigned long long int key3[8] = { 0x0043, 0xc09b, 0xab01, 0xe9e9, 0xc733, 0x87e0, 0x7415, 0x8395 };
puts("Choose your key :\n");
scanf("%d", &k);
if (k==1)
{
    for (i=0; i<8; ++i)
        temp_key[i]=key1[i];
}
else if (k==2)
{
    for (i=0; i<8; ++i)
        temp_key[i]=key2[i];
}
else if (k==3)
{
    for (i=0; i<8; ++i)
        temp_key[i]=key3[i];
}
else
{
    puts("Press between 1,2,3");
    KEYGEN();
}
copy_bigint(Key, temp_key, 8);
puts("-------------------------------");
puts("Input Key:\n");
for (i=0; i<8; ++i)
display_bytewise(Key[i], 16);
puts("-------------------------------");

/* IVSETUP() modifies the master state by modifying counter C (global) and also 
    serves the User interface to choose between three different IVs*/

void IVSETUP()
{
    int i, j, k;

    /* Three different IVs hard-coded. User may change them if necessary*/
    unsigned long long int IV1[4] ={ 0x0000, 0x0000, 0x0000, 0x0000};
    unsigned long long int IV2[4] ={ 0x7e59, 0xc126, 0xf575, 0xc373};
    unsigned long long int IV3[4] ={ 0x1727, 0xd2f4, 0x561a, 0xa6eb};
puts("Choose your IV:\n");
scanf("%d",&k);
if(k==1)
{
    for(i=0;i<4;++i)
        IV[i]=IV1[i];
}
else if(k==2)
{
    for(i=0;i<4;++i)
        IV[i]=IV2[i];
}
else if(k==3)
{
    for(i=0;i<4;++i)
        IV[i]=IV3[i];
}
else
{
    puts("Press between 1,2,3");
    IVSETUP();
}

puts("-----------------------------");
puts("Input IV:\n");
for(i=0;i<4;++i)
    display_bytewise(IV[i],16);
puts("-----------------------------");

/*Updating Counter variables by specified equations*/
C[0] = C[0] ^ ((IV[0])+(IV[1]<<16));

/*Iterating the system by calling NEXTSTATE() 4 times*/
for(i=0;i<4;++i)
    NEXTSTATE();
/*GENERATE() generates the 128-bit pseudorandom bit-stream in each iteration*/
void GENERATE()
{
    int i,j,k,l,m;

    /*Due to binary operations temporary binary variables are declared and
    allocated*/
    unsigned char **x,**c;
    x=(unsigned char**)malloc(sizeof(unsigned char*)*8);
    for(i=0;i<8;++i)
    {
        x[i]=(unsigned char*)malloc(sizeof(unsigned char)*32);
    }
    initialize(x,8,32);
    s=(unsigned char**)malloc(sizeof(unsigned char*)*8);
    for(i=0;i<8;++i)
    {
        s[i]=(unsigned char*)malloc(sizeof(unsigned char)*32);
    }
    initialize(s,8,32);

    puts("-------------------------------");
    puts("Output Stream:
");

    /*The system is iterated for 3 times to output 48-byte pseudorandom
    bit-stream */
    for(j=0;j<ROUND;++j)
    {
        NEXTSTATE();

        for(i=0;i<8;++i)
        {
            dectobin(x[i],X[i],32);/**<Decimal X (global) is converted to
            binary x (local)>>
        }

        /*Generate output by specified equations*/
        for(i=0;i<16;++i)
        {
            s[0][i+16]=x[0][i+16]~x[5][i];
            s[0][i]=x[0][i]~x[3][i+16];
            s[1][i+16]=x[2][i+16]~x[7][i];
        }
    }
\[ s[1][i] = x[2][i] \times x[5][i+16]; \]
\[ s[2][i+16] = x[4][i+16] \times x[1][i]; \]
\[ s[2][i] = x[4][i] \times x[7][i+16]; \]
\[ s[3][i+16] = x[6][i+16] \times x[3][i]; \]
\[ s[3][i] = x[6][i] \times x[1][i+16]; \]

```c
/* Display Output byte-wise in hexadecimal form*/
for (m=0; m<4; ++m) {
    display_bytewise(bintodec(s[m], 32), 32);
}
```

```c
puts("--------------------");
```

```c
/*main() function calls the sub-routines in order*/
int main() {
    int i=0, j;
    char k;
    KEYGEN();
    KEYSETUP();
    puts("Do you want to use IV? (Press 1 for 'yes' and any other key for 'no')\n");
    scanf("%d", &i);
    if (i==1)
        IVSETUP(); /* It may be skipped if IV is not required by the user*/
    GENERATE();
    return 0;
}
```
Chapter 2

Salsa 20

2.1 Introduction

Salsa 20 is a stream cipher submitted to eSTREAM [58] by Daniel Bernstein. It is built on a pseudorandom function based on 32-bit addition, bitwise addition (XOR) and rotation operations, which maps a 256-bit key, a 64-bit nonce (number used once), and a 64-bit stream position to a 512-bit output (a version with a 128-bit key also exists). This gives Salsa20 the unusual advantage that the user can efficiently seek to any position in the output stream. It offers speeds of around 4 – 14 cycles per byte in software on modern x86 processors, and reasonable hardware performance. It is not patented, and Bernstein has written several public domain implementations optimized for common architectures. The version selected in the eSTREAM profile has 12 rounds. So, it is called Salsa 20/12. Here we focus on the stream cipher Salsa20 in general.

In [17], the author himself provided a lot of security analysis. Also, in the eSTREAM website, a few other attacks has been provided. Still the security is quite intact. There are a lot of active researches going on around the crypto-community to break this very popular eSTREAM cipher.

In [20], the author has nicely provided the specs, design aspects, security analysis and the codes. Here we describe the design aspects from our own perspective to make everything the most readable and user-friendly. Also later in this chapter we describe a few recent attacks.

2.2 Specifications of Salsa20

The core of Salsa20 is a hash function with 64-byte input and 64-byte output. The hash function is used in counter mode as a stream cipher: Salsa20 encrypts a 64-byte block of plaintext by
hashing the key, nonce, and block number and xor’ing the result with the plaintext. As in [18], we describe the spec in a bottom-up manner starting from three simple operations on 4-byte words, continuing through the Salsa20 hash function, and finishing with the Salsa20 encryption function. We start with the basic block i.e. bytes which is an element of the set \{0, 1, \ldots, 255\}.

A word is an element of the set \{0, 1, \ldots, 2^{32} - 1\}. They are generally represented in hexadecimal notation. The sum of two words \(u\) and \(v\) is defined as \((u + v) \mod 2^{32}\).

### 2.2.1 The quarterround function

It takes a 4-word sequence as input and also returns a 4-word sequence. If \(y = \{y_0, y_1, y_2, y_3\}\) is the input, then \(\text{quarterround}(y) = z = \{z_0, z_1, z_2, z_3\}\) is defined as follows:

\[
\begin{align*}
    z_1 &= y_1 \oplus ((y_0 + y_3) \ll 7), \\
    z_2 &= y_2 \oplus ((z_1 + y_0) \ll 9), \\
    z_3 &= y_3 \oplus ((z_2 + z_1) \ll 13), \\
    z_0 &= y_0 \oplus ((z_3 + z_2) \ll 18).
\end{align*}
\]

One can visualize the quarterround function as modifying \(y\) in place i.e. first \(y_1\) changes to \(z_1\), then \(y_2\) changes to \(z_2\), then \(y_3\) changes to \(z_3\), then \(y_0\) changes to \(z_0\). Each modification is invertible, so the entire function is invertible.

### 2.2.2 The rowround function

It takes a 16-word sequence as input and returns a 16-word sequence. If \(y = \{y_0, y_1, \ldots, y_{15}\}\) is the input, then \(\text{rowround}(y) = z = \{z_0, z_1, \ldots, z_{15}\}\) is defined as follows:

\[
\begin{align*}
    (z_0, z_1, z_2, z_3) &= \text{quarterround}(y_0, y_1, y_2, y_3) \\
    (z_4, z_5, z_6, z_7) &= \text{quarterround}(y_5, y_6, y_7, y_4) \\
    (z_8, z_9, z_{10}, z_{11}) &= \text{quarterround}(y_{10}, y_{11}, y_8, y_9) \\
    (z_{12}, z_{13}, z_{14}, z_{15}) &= \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14})
\end{align*}
\]

One can visualize the input as square matrix:

\[
\begin{bmatrix}
    y_0 & y_1 & y_2 & y_3 \\
    y_4 & y_5 & y_6 & y_7 \\
    y_8 & y_9 & y_{10} & y_{11} \\
    y_{12} & y_{13} & y_{14} & y_{15}
\end{bmatrix}
\]
2.2. Specifications of Salsa20

The rowround function modifies the rows of the matrix in parallel by feeding a permutation of each row through the quarterround function. The order of modification is as shown in the equation 2.2.

2.2.3 The \textit{columnround} function

It is similar to the rowround function. It also takes a 16-word sequence as input and returns a 16-word sequence. If \( y = \{y_0, y_1, \ldots, y_{15}\} \) is the input, then \( \text{columnround}(y) = z = \{z_0, z_1, \ldots, z_{15}\} \) is defined as follows:

\[
\begin{align*}
(z_0, z_4, z_8, z_{12}) &= \text{quarterround}(y_0, y_4, y_8, y_{12}) \\
(z_5, z_9, z_{13}, z_{1}) &= \text{quarterround}(y_5, y_9, y_{13}, y_{1}) \\
(z_{10}, z_{14}, z_2, z_6) &= \text{quarterround}(y_{10}, y_{14}, y_2, y_6) \\
(z_{15}, z_3, z_7, z_{11}) &= \text{quarterround}(y_{15}, y_3, y_7, y_{11})
\end{align*}
\] (2.4)

Similar to the rowround function, here also the input can be visualized as a square matrix, but here the modification is done by columns instead. The columnround function is, from this perspective, simply the transpose of the rowround function: it modifies the columns of the matrix in parallel by feeding a permutation of each column through the quarterround function.

2.2.4 The \textit{doubleround} function

The doubleround function is simply a columnround followed by a rowround. Certainly, it takes a 16-word sequence as input and returns a 16-word sequence. If \( y = \{y_0, y_1, \ldots, y_{15}\} \) is the input, then \( \text{doubleround}(y) = z = \{z_0, z_1, \ldots, z_{15}\} = \text{rowround}(\text{columnround}(y)) \). One can visualize a double round as modifying the columns of the input in parallel, and then modifying the rows in parallel. Each word is modified twice.

2.2.5 The \textit{littleendian} function

If \( b = (b_0, b_1, b_2, b_3) \) is a 4-byte sequence then \( \text{littleendian}(b) \) is a word defined as follows:

\[
\text{littleendian}(b) = b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3
\] (2.5)

It is important to note that this function is invertible.

2.2.6 The Salsa20 hash function

This is the core of the cipher. It takes a 64-byte sequence as input and outputs another 64-byte sequence. In short it is defined as \( \text{Salsa20}(x) = x + \text{doubleround}^{10}(x) \) where each 4-byte
sequence is viewed as a word in little-endian form. In detail, it can be described as follows:

(Let \( x = x[0], x[1], ..., x[63] \))

\[
\begin{align*}
  x_0 &= \text{littleendian}(x[0], x[1], x[2], x[3]), \\
  x_1 &= \text{littleendian}(x[4], x[5], x[6], x[7]), \\
  &\vdots \\
  x_{15} &= \text{littleendian}(x[60], x[61], x[62], x[63]).
\end{align*}
\]

Define \((z_0, z_1, ..., z_{15}) = \text{doubleround}^{10}(x_0, x_1, ..., x_{15})\). Then \(\text{Salsa20}(x)\) is the concatenation of the following:

\[
\begin{align*}
  \text{littleendian}^{-1}(z_0 + x_0), \\
  \text{littleendian}^{-1}(z_1 + x_1), \\
  &\vdots \\
  \text{littleendian}^{-1}(z_{15} + x_{15})
\end{align*}
\]

### 2.2.7 The Salsa20 expansion function

If \(k\) is a 32-byte or 16-byte sequence and \(n\) is a 16-byte sequence then \(\text{Salsa20}_k(n)\) is a 64-byte sequence defined as follows:

Let

\[
\begin{align*}
  \sigma_0 &= (101, 120, 112, 97), \\
  \sigma_1 &= (110, 100, 32, 51), \\
  \sigma_2 &= (50, 45, 98, 121), \\
  \sigma_3 &= (116, 101, 32, 107)
\end{align*}
\]

and,

\[
\begin{align*}
  \tau_0 &= (101, 120, 112, 97), \\
  \tau_1 &= (110, 100, 32, 49), \\
  \tau_2 &= (54, 45, 98, 121), \\
  \tau_3 &= (116, 101, 32, 107)
\end{align*}
\]

Now if \(k_0, k_1, n\) are 16-byte sequences then \(\text{Salsa20}_{k_0,k_1}(n) = \text{Salsa20}(\sigma_0, k_0, \sigma_1, n, \sigma_2, k_1, \sigma_3)\). Else if \(k, n\) are 16-byte sequences then \(\text{Salsa20}_k(n) = \text{Salsa20}(\tau_0, k, \tau_1, n, \tau_2, k, \tau_3)\).

“Expansion” refers to the expansion of \((k, n)\) into \(\text{Salsa20}_k(n)\). It also refers to the expansion of \(k\) into a long stream of \(\text{Salsa20}_k\) outputs for various \(n\)’s and the constants \(\sigma_0\sigma_1\sigma_2\sigma_3\) and \(\tau_0\tau_1\tau_2\tau_3\) are “expand 32-byte k” and “expand 16-byte k” in ASCII.
2.2.8 The Salsa20 encryption function

The Encryption function of Salsa20 is based on all the above blocks. Let $k$ be a 32-byte or 16-byte sequence. Let $v$ be an 8-byte sequence. Let $m$ be an $l$-byte sequence for some $l \in 0, 1, \ldots, 2^{70}$. The Salsa20 encryption of $m$ with nonce $v$ under key $k$, denoted $Salsa_{20}^k(v) \oplus m$, is an $l$-byte sequence. Normally $k$ is a secret key (preferably 32 bytes); $v$ is a nonce, i.e., a unique message number; $m$ is a plaintext message; and $Salsa_{20}^k(v) \oplus m$ is a ciphertext message. Or $m$ can be a ciphertext message, in which case $Salsa_{20}^k(v) \oplus m$ is the original plaintext message. Formally, the function is written as follows:

$Salsa_{20}^k(v)$ is the $2^{70}$-byte sequence:

$Salsa_{20}^k(v, 0), Salsa_{20}^k(v, 1), Salsa_{20}^k(v, 2), \ldots, Salsa_{20}^k(v, 2^{64} - 1)$

Here $i$ is the unique 8-byte sequence $(i_0, i_1, \ldots, i_7)$ such that $i = i_0 + 2^8i_1 + 2^{16}i_2 + \ldots + 2^{56}i_7$. The formula $Salsa_{20}^k(v) \oplus m$ implicitly truncates $Salsa_{20}^k(v)$ to the same length as $m$. In other words,

$Salsa_{20}^k(v) \oplus (m[0], m[1], \ldots, m[l - 1]) = (c[0], c[1], \ldots, c[l - 1])$

where $c[i] = m[i] \oplus Salsa_{20}^k(v, i/64)[i \mod 64]$.

From the description in this section, it is easy to observe that, the definition of Salsa20 could easily be generalized from byte sequences to bit sequences, given an encoding of bytes as sequences of bits. However, there is no apparent application of this generalization.

2.3 Security Properties of Salsa20

The security of this cipher is analyzed nicely in [17] by Daniel Bernstein himself. We briefly describe the analysis here. For details analysis one may look into that document. If the Salsa20 key $k$ is a uniform random sequence of bytes, and the same nonce is never used for two different messages, then the Salsa20 encryption function is conjectured to produce ciphertexts that are indistinguishable from uniform random strings. At a lower level, the random function $n \rightarrow Salsa_{20}^k(n)$ from $\{0, 1, \ldots, 255\}^{16}$ to $\{0, 1, \ldots, 255\}^{64}$ is conjectured to be indistinguishable from uniform. This conjecture implies the first conjecture. The remaining part of this section explain why these conjectures are plausible, i.e., why Salsa20 is difficult to break. The Salsa20 design is quite conservative, allowing more confidence in these conjectures than in the analogous conjectures for some other functions.
Side-channel Attacks

Natural Salsa20 implementations take constant time on a huge variety of CPUs. There is no incentive for the authors of Salsa20 software to use variable-time operations such as S-box lookups. Timing attacks against Salsa20 are therefore just as difficult as pure cryptanalysis of the Salsa20 outputs. The operations in Salsa20 are also among the easiest to protect against power attacks and other side-channel attacks.

2.3.1 The cost of an attack

Assume that the target Salsa20 key \( k \) is a uniform random 32-byte sequence. Now, how to distinguish the Salsa20 ciphertexts from uniform random strings? The most obvious (and naive too) choice is brute-force. Consider a gigantic parallel machine with \( 2^{64} \) independent key-searching units, given a pair \((n, \text{Salsa20}_k(n))\) as input. One unit searches through \( 2^{192} \) keys in the time taken for \( 2^{192} \) Salsa20 hash-function evaluations; in the same amount of time, the entire machine operating in parallel searches through all \( 2^{256} \) keys, and is guaranteed to find the target key. The Salsa20 security conjecture is that one cannot simultaneously achieve a substantially better price, performance, and chance of success: there is no machine such that:

- costs substantially less than \( 2^{64} \) key-searching units,
- takes time substantially less than \( 2^{128} \) Salsa20 hash-function computations, and
- has chance substantially above \( 2^{-64} \) of distinguishing \( n \to \text{Salsa20}_k(n) \) from a uniform random function.

The words “substantially” keeps room for minor speed-ups and “distinguishing” is defined in the usual way.

Half-size keys

The security conjecture for 16-byte Salsa20 keys chops each exponent in half: there is no machine that costs substantially less than \( 2^{32} \) key-searching units, takes time substantially less than \( 2^{64} \) Salsa20 hash-function computations, and has probability substantially above \( 2^{-32} \) of distinguishing \( n \to \text{Salsa20}_k(n) \) from a uniform random function.

In [17], the author recommends to use 256-bit keys as a brute-force search through \( 2^{96} \) keys would be extremely expensive but is not inconceivable, and a success probability of \( 2^{-32} \) is not negligible. This recommendation has no relation with the security of Salsa20, rather it deals with the feasibility of the attack considering current computational power.
2.3.2 Notes on the diagonal constants

Each Salsa20 column round affects each column in the same way starting from the diagonal. Each Salsa20 row round affects each row in the same way starting from the diagonal. Consequently, shifting the entire Salsa20 hash-function input array along the diagonal has exactly the same effect on the output. The Salsa20 expansion function eliminates this shift structure by limiting the attacker’s control over the hash-function input. In particular, the input diagonal is always 0x61707865, 0x3320646e, 0x79622d32, 0x6b206574 which is different from all its nontrivial shifts. In other words, two distinct arrays with this diagonal are always in distinct orbits under the shift group.

Similarly, the Salsa20 hash-function operations are almost compatible with rotation of each input word by, say, 10 bits. Rotation changes the effect of carries that cross the rotation boundary, but it is consistent with all other carries, and with the Salsa20 operations other than addition. The Salsa20 expansion function also eliminates this rotation structure. The input diagonal is different from all its nontrivial shifts and all its nontrivial rotations and all nontrivial shifts of its nontrivial rotations. In other words, two distinct arrays with this diagonal are always in distinct orbits under the shift/rotate group.

2.3.3 Diffusion in Salsa20

The Salsa20 cipher has the nice diffusing property which is essential quality for a good stream cipher. We explain with an example. Consider computing the second block of the Salsa20 stream with nonce 0 and key (1, 2, 3, ..., 32). Rather than displaying the arrays produced by the second block computation, this section displays the xor between those arrays and the corresponding first-block arrays, to emphasize the “active” bits - the bits where the computations differ.

The Salsa20 hash function starts with a 4 × 4 input array whose only difference from the first block is the different block counter, as shown by the following xor:

\[
\begin{align*}
0x00000000, 0x00000000, 0x00000000, 0x00000000, \\
0x00000000, 0x00000000, 0x00000000, 0x00000000, \\
0x00000001, 0x00000000, 0x00000000, 0x00000000, \\
0x00000000, 0x00000000, 0x00000000, 0x00000000.
\end{align*}
\]

By the end of the first round, the difference has propagated to two other entries in the same column:
At this point there are still just a few active bits. The difference depends on a few carries but is still highly predictable. The second round then propagates the difference across columns:

\[
0x80040001, 0x00000000, 0x00000000, 0x00000000, \\
0x00000000, 0x00000000, 0x00000000, 0x00000000, \\
0x00000001, 0x00000000, 0x00000000, 0x00000000, \\
0x0000e000, 0x00000000, 0x00000000, 0x00000000.
\]

And, by the end of the third round, every word has been affected:

\[
0xedc5e0a9, 0x020000c0, 0x381f830c, 0x304888dc, \\
0x00000000, 0x00000000, 0x00000000, 0x00000000, \\
0x00000001, 0x00006000, 0x800c0001, 0x00000000, \\
0x0000e000, 0x01c00000, 0x040000d8, 0x01200f00.
\]

A substantial fraction of the bits are now active, although two words still have stretches of bits that were not (and were unlikely to be) active.

By the end of the fourth round, those last two stretches of inactivity have been eliminated:

\[
0x39545d5e, 0x0cc160d8, 0x301fb030, 0xa05208dc, \\
0xa240cc8b, 0x24e0120c, 0x2a030dc7, 0x0abe9b94e, \\
0x39ea409b, 0x08000000f, 0xc0f3bb828, 0x1c205f6d, \\
0xc6612ba5, 0x01c06a00, 0x02000018, 0x6745c36b.
\]

That is just 4 out of the 20 rounds in *Salsa20*. In every subsequent round, there are hundreds of active bits, for a total of more than 4000 active bits. Each of those 4000 active bits interacts with carries in a random-looking way, producing random-looking differences, which is not shown here.

### 2.3.4 Differential attacks

The idea of differential attack has been described in section 1.3.6 of the chapter 1. Now, suppose that there is a “small” difference \( n \oplus n' \) that has a perceptible chance of producing
a “small” state difference after several rounds of Salsa20. In other words: suppose that, for all the pairs \((n, n')\) having that difference, and for many keys \(k\), there is a “small” difference after several rounds of Salsa20. Then it should be possible to find at least one example of a qualifying \((n, n', k)\). But in [17], the author stated that, there is no reason to believe that one such example exists.

Salsa20 is quite different in this respect from ciphers such as AES where the input size is as large as the state size. AES has 16-byte inputs, 16-byte outputs, and (at least) 16-byte keys; there are \(2^{384}\) choices of \((n, n', k)\), so presumably there are more than \(2^{128}\) choices in which both of the 128-bit quantities \(n \oplus n'\) and \(AES_k(n) \oplus AES_k(n')\) are “small”. On the other hand, Salsa20 has 16-byte inputs, 64-byte outputs, and 32-byte keys; there are \(2^{512}\) choices of \((n, n', k)\), so there is no a-priori reason to believe that any of the choices have the 128-bit quantity \(n \oplus n'\) and the 512-bit quantity \(Salsa20_k(n) \oplus Salsa20_k(n')\) both being “small”. Hence if differential attack is not possible on this “class” of cipher which contains Salsa20 but not AES. So, while considering the possibility of differential attack, the AES is obviously more prone to the attack than Salsa20. And, AES has no such attack till date. So the possibility of differential attack on Salsa20 is even lower.

Clearly there are lot of difficulties to constitute a differential attack. Even with control over \(k\), it does not appear to be possible to keep a difference constrained within a small number of bits. The first two rounds

- convert a small change to \(x_6\) into large changes in \(x_5, x_8, x_9, x_{10}\) and smaller changes in \(x_0, x_2, x_3, x_4, x_6, x_7, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}\);

- convert a small change to \(x_7\) into medium-size changes in \(x_{13}, x_{14}, x_{15}\) and smaller changes in \(x_4, x_5, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}\);

- convert a small change to \(x_8\) into medium-size changes in \(x_0, x_2, x_3\) and smaller changes in \(x_1, x_8, x_9, x_{10}, x_{12}, x_{13}, x_{14}, x_{15}\); and

- convert a small change to \(x_9\) into large changes in \(x_0, x_4, x_5, x_7\) and smaller changes in \(x_1, x_2, x_3, x_6, x_8, x_9, x_{10}, x_{11}, x_{13}, x_{14}, x_{15}\).

Small combinations of these changes do not cancel many active bits. The author told that for every key, every input pair has more active bits after two rounds, and has thousands of active bits overall. Those thousands of active bits have thousands of random-looking interactions with carries.

Other notions of small differences, like using ‘−’ instead of ‘⊕’, for example, or ignoring some bits do not seem to help. Higher-order differential attacks do not seem to help. “Slide”
differentials, in which one compares an input array to e.g. the 2-round array for another input, does not work for the same basic reason.

2.3.5 Algebraic attacks

The idea of algebraic attack has been described in detail in section 1.3.4 of chapter 1. Intuitively, the target is to come up with a “small” set of equations satisfied by input states, output states, and (unknown) intermediate states, and then solve the equations or, for a distinguisher, see whether the equations have a solution. More generally, one might come up with equations that are usually satisfied, or sometimes satisfied, or satisfied noticeably more often for the cipher than for independent uniform random input and output bits. This broader perspective includes differential attacks, linear attacks, etc.

The author claims that, there does not seem to exist any “small” set of equations for the state bits in Salsa20. Each of the 320 32-bit additions in the Salsa20 computation requires dozens of quadratic equations, producing a substantially larger system of equations than are required to describe, for example, the bits in AES. Groebner-basis techniques (described in detail in [40]) for solving the AES-bit equations are, by the most optimistic estimates, slightly faster than brute-force search for a 256-bit key, but they use vastly more memory and thus have a much worse price-performance ratio. Algebraic attacks against Salsa20 appear to be even more difficult.

2.3.6 Other attacks

Weak-key attacks

Suppose that there is a special set of $2^{200}$ keys that are surprisingly easy to recognize i.e. they are found by a machine with comparable cost to $2^{56}$ key-searching units running for only as long as $2^{120}$ Salsa20 hash-function computations, rather than the obvious $2^{200}/2^{56} = 2^{144}$. That machine, when applied to a uniform random Salsa20 key, would have success probability $2^{200}/2^{256} = 2^{-56}$. This machine, being 28 times faster, 28 times less expensive, and 28 times more likely to succeed than the machine described in section 2.3.1, would violate the Salsa20 security conjecture.

This type of attack seems highly implausible for Salsa20. The Salsa20 key is mangled along with the input in an extremely complicated way. Any key differences rapidly spread through the entire Salsa20 state for the same reason that input differences do.
2.4. Performance Evaluation of Salsa20

Equivalent-key attacks

Let us assume that there exists an easily searched set $S$ of $2^{176}$ keys where each key $k \in S$ transforms inputs in the same way as $2^{24} - 1$ other keys. A machine with comparable cost to $2^{56}$ key-searching units, running for only as long as $2^{120}$ Salsa20 hash-function computations, searching through that set of $2^{176}$ keys, would actually be a distinguisher for $2^{200}$ keys, and would have success probability $2^{200}/2^{256} = 2^{-56}$. This machine would again violate the Salsa20 security conjecture. In other words, there is no need to make a separate conjecture regarding equivalent keys. This type of attack, like a weak-key attack, seems highly implausible for Salsa20.

Related-key attacks

The standard solutions to all the standard cryptographic problems—encryption, authentication, etc. are protocols that do not allow related-key attacks on the underlying primitives. The author claims to see no evidence that we can save time by violating this condition. He also says that it might be easily guessable that Salsa20 is highly resistant to related-key attacks but provides no guarantee.

2.4 Performance Evaluation of Salsa20

In [19], Bernstein himself discussed the performance issue in detail. That document discusses a range of benchmarks relevant to cryptographic speed; estimates Salsa20’s performance on those benchmarks; and explains, at a lower level, techniques to achieve this performance. Here we provide a overall idea of that. It was also stated that, Salsa20 provides consistent high speed in a wide variety of applications across a wide variety of platforms. Consistency means that, in each of these contexts, Salsa20 is not far from the existing fastest cryptographic function.

2.4.1 Salsa20 on the AMD Athlon

The Athlon has 7 usable integer registers, one of which is consumed by a round counter if the Salsa20 code is not completely unrolled. The Athlon is limited to 3 instructions per cycle and 2 memory operations per cycle. The small number of registers means that each round requires many load and stores. Loads can be absorbed into load-operate instructions, although they still count against the memory-operation bottleneck.

The main points are described here:
• The optimized code (implemented by the author himself) takes 29.25 Athlon cycles for a Salsa20 round, totalling 585 cycles (9.15 cycles/byte) for 20 rounds.

• It takes 645 cycles in total (10.08 cycles/byte) for the Salsa20 hash function, timed as 680 cycles with 35 cycles timing overhead.

• The timings are actually 655 or 656 cycles most of the time but 849 cycles on every eighth call, presumably because of branch mispredictions.

• The compiled code occupies 1248 bytes; Its main loop occupies 937 bytes and handles 4 rounds.

2.4.2 Salsa20 on the IBM PowerPC RS64 IV (Sstar)

The PowerPC RS64 IV has enough registers to avoid all loads and stores inside the hash-function rounds. The 16 words of hash-function input are loaded into separate registers; 4 quarter-rounds are performed in parallel, with 1 temporary register for each quarter-round; after 20 rounds, the input is loaded again, added to the round output, and stored. The obvious bottleneck is that the PowerPC RS64 IV is limited to 2 integer operations per cycle, with a rotate instruction counting as 2 operations. Each round has 64 operations and therefore takes at least 32 cycles, totalling 640 cycles (10.00 cycles/byte) for 20 rounds, even with fully unrolled code. The main points are described here:

• The author’s code takes 33 PowerPC RS64 IV cycles for each Salsa20 round, totalling 660 cycles (10.32 cycles/byte) for 20 rounds.

• It takes 756 cycles (11.82 cycles/byte) for the Salsa20 hash function, timed as 770 cycles with 14 cycles timing overhead.

• The compiled code for the Salsa20 hash function occupies 768 bytes; Its main loop occupies 392 bytes and handles 2 rounds.

2.4.3 Salsa20 on the Intel Pentium III

The Pentium III has 7 usable integer registers, one of which is consumed by a round counter if the Salsa20 code is not completely unrolled. The small number of registers means that each round requires many loads and stores. The Pentium III is limited to 3 operations per cycle. A store instruction counts as 2 operations. A load-operate instruction counts as 2 operations. The Pentium III is also limited to 2 integer operations per cycle. A store to the stack, and a subsequent load from the stack, can be replaced with a store to "MMX registers," and a subsequent load from MMX registers. The MMX store counts for only 1 operation, unlike a stack
store. On the other hand, the MMX load and the MMX store both count as integer operations, unlike a stack load and a stack store.

The main points are described here:

- The author’s code takes 37.5 Pentium III cycles for each Salsa20 round, totalling 750 cycles (11.72 cycles/byte) for 20 rounds.
- It takes 837 cycles (13.08 cycles/byte) for the Salsa20 hash function, timed as 872 cycles with 35 cycles timing overhead.
- The timings are actually 859 cycles most of the time but 908 cycles on every fourth call, presumably because of branch mispredictions.
- The compiled code for the Salsa20 hash function occupies 1280 bytes; Its main loop occupies 937 bytes and handles 4 rounds.

2.4.4 Salsa20 on the Intel Pentium 4 f12 (Willamette)

The Pentium 4 does badly with salsa20: it has a high latency for moving data between the 32-bit integer registers and the 64-bit MMX registers. The Pentium 4 f12 does better with, but other Pentium 4 CPUs have a high latency for reading data that was recently written to memory. So the most optimized code takes a completely different approach. The Pentium 4 has eight “XMM registers,” each of which can hold four 32-bit integers. The Pentium 4 has several XMM instructions: ADD, XOR, SHIFT and SHUFFLE. The Pentium 4 cannot perform two of these operations on the same cycle; it cannot perform two arithmetic operations (ADD, XOR) on adjacent cycles; it cannot perform two shift operations (SHIFT, SHUFFLE) on adjacent cycles. So, to solve this at the beginning of a column round, the code stores the input \((x_0, x_1, \ldots, x_{15})\) in four XMM registers. It performs a column round with the instructions, which just barely fit into 8 registers.

The main points are described here:

- The author’s code takes 48 Pentium 4 f12 (Willamette) cycles for each Salsa20 round, totalling 960 cycles (15 cycles/byte) for 20 rounds.
- It takes 1052 cycles (16.44 cycles/byte) for the Salsa20 hash function, timed as 1136 cycles with 84 cycles timing overhead.
- The compiled code for the Salsa20 hash function occupies 1144 bytes. Its main loop occupies 629 bytes and handles 4 rounds.
2.4.5 Salsa20 on the Intel Pentium M

The Pentium M has 7 usable integer registers, one of which is consumed by a round counter if the Salsa20 code is not completely unrolled. The small number of registers means that each round requires many loads and stores. The Pentium M is limited to 3 operations per cycle. Like the Pentium III, the Pentium M counts a load-operate instruction as 2 operations. Unlike the Pentium III, the Pentium M counts a store instruction as 1 operation. This difference means that code is slightly faster on the Pentium M than on the Pentium III, taking only about 36 cycles/round; it also means that a quite different sequence of instructions produces better results. A round ends up using 90 operations (taking at least 30 cycles): 16 additions, 16 rotations, 16 xor’s, 16 stores, and 26 loads. As for the 65-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

The main points are described here:

- The author’s code takes 33.75 Pentium M cycles for each Salsa20 round, totalling 675 cycles (10.55 cycles/byte) for 20 rounds.
- It takes 740 cycles (11.57 cycles/byte) for the Salsa20 hash function, timed as 790 cycles with 50 cycles timing overhead.
- The timings are actually 780 or 781 cycles most of the time but 856 cycles on every eighth call, presumably because of branch mispredictions.
- The compiled code for the Salsa20 hash function occupies 1248 bytes; Its main loop occupies 937 bytes and handles 4 rounds.

2.4.6 Salsa20 on the Motorola PowerPC 7410 (G4)

The PowerPC 7410, like the PowerPC RS64 IV, has enough registers to avoid all loads and stores inside the hash-function rounds. The obvious bottleneck is that the PowerPC 7410 is limited to 2 integer operations per cycle. The PowerPC 7410, unlike the PowerPC RS64 IV, counts a rotate instruction as 1 operation. Each round has 48 operations and therefore takes at least 24 cycles, totalling 480 cycles (7.50 cycles/byte) for 20 rounds, even with fully unrolled code.

The main points are described here:

- The author’s code takes 24.5 PowerPC 7410 cycles for each Salsa20 round, totalling 490 cycles (7.66 cycles/byte) for 20 rounds.
2.4. Performance Evaluation of Salsa20

- It takes approximately 570 cycles (8.91 cycles/byte) for the Salsa20 hash function, timed as approximately 584 cycles with 14 cycles timing overhead.

- The compiled code for the Salsa20 hash function occupies 768 bytes. Its main loop occupies 392 bytes and handles 2 rounds.

2.4.7 Salsa20 on the Sun UltraSPARC II

The UltraSPARC II handles each rotation with 3 integer operations: shift, shift, add. It is limited to 2 integer operations per cycle, and to 1 shift per cycle. Like the PowerPC, it has enough registers to avoid all loads and stores inside the hash-function rounds. A round has 80 integer operations: 32 adds, 32 shifts, 16 xors- and therefore takes at least 40 cycles. As for the 71-cycles-per-hash overhead: One can easily eliminate some of this overhead by merging the Salsa20 hash function with a higher-level encryption function.

The main points are described here:

- The author’s code takes 40.5 UltraSPARC II cycles for each Salsa20 round, totalling 810 cycles (12.66 cycles/byte) for 20 rounds.

- It takes 881 cycles (13.77 cycles/byte) for the Salsa20 hash function, timed as 892 cycles with 11 cycles timing overhead.

- The compiled code for the Salsa20 hash function occupies 936 bytes; Its main loop occupies 652 bytes and handles 2 rounds.

2.4.8 Salsa20 on the Sun UltraSPARC III

The UltraSPARC III is very similar to the UltraSPARC II. The UltraSPARC III documentation reports a few minor advantages that are not helpful for the Salsa20 computation: e.g., both integer operations in a cycle can be shifts. The disadvantages of the UltraSPARC III are not well documented.

The main points are described here:

- The author’s code takes 41 UltraSPARC III cycles for each Salsa20 round, totalling 820 cycles (12.82 cycles/byte) for 20 rounds.

- It takes 889 cycles (13.90 cycles/byte) for the Salsa20 hash function, timed as 905 cycles with 16 cycles timing overhead.

- The compiled code for the Salsa20 hash function occupies 936 bytes; Its main loop occupies 652 bytes and handles 2 rounds.
2.4.9 Salsa20 on next-generation CPUs

One can safely expect Salsa20 to perform well on tomorrow’s popular CPUs, for the same reason that Salsa20 achieves consistent high speed on a wide variety of existing CPUs. The basic operations in Salsa20—addition modulo 2^{32}, constant-distance 32-bit rotation, and 32-bit xor—are so simple, and so widely used, that they can be safely expected to remain fast on future CPUs. Consider, as an extreme example, the Pentium 4 f12, widely criticized for its “slow” shifts and rotations (fixed by the Pentium 4 f33); this CPU can still perform two 32-bit shifts per cycle using its XMM instructions. The accompanying communication in Salsa20—the addition, rotation, and xor modify 1 word out of 16 words, using 2 other words—is sufficiently small that it can also be expected to remain fast on future CPUs. Fast Salsa20 code is particularly easy to write if the 16 words, and a few temporary words, fit into registers; but, as illustrated by the Salsa20 implementation (detail in [20]) for the Pentium M, smaller register sets do not pose a serious problem.

Furthermore, Salsa20 can benefit from a CPU’s ability to perform several operations in parallel. For example, the PowerPC 7450 (G4e) documentation in [19] indicates that the PowerPC 7450 can perform 3 operations per cycle instead of the 2 performed by the PowerPC 7410. Latency does not become a bottleneck for Salsa20 unless the CPU’s latency/throughput ratio exceeds 4. One can imagine functions that use even simpler operations, and that have even less communication, and that support even more parallelism. But what really matters is that Salsa20 is simpler, smaller, and more parallel than the average computation. It is hard to imagine how a CPU could make Salsa20 perform badly without also hurting a huge number of common computations.

2.4.10 Salsa20 on smaller platforms

Building a Salsa20 circuit is straightforward. A 32-bit add-rotate-xor fits into a small amount of combinational logic. Salsa20’s 4 × 4 row-column structure is a natural hardware layout; the regular pattern of operations means that each quarter-round will have similar propagation delays. Of course, the exact speed of a Salsa20 circuit will depend on the amount of hardware devoted to the circuit. Similarly, the 32-bit operations in the Salsa20 computation can easily be decomposed into common 8-bit operations for a small CPU:

- A 32-bit xor can be decomposed into four 8-bit xors.
- A 32-bit addition can be decomposed into four 8-bit additions (with carry).
- A 32-bit rotation can be decomposed into, e.g., several 8-bit rotate-carry operations. The exact number of rotate-carry operations depends on how close the rotation distance is to
A multiple of 8.

An average Salsa20 word modification ends up taking about 20 8-bit arithmetic operations—about 20 cycles on a typical 8-bit CPU. If loads and stores consume another 32 cycles then 20 rounds of the Salsa20 hash function will take about 16640 cycles (260 cycles/byte). Salsa20 has no trouble fitting into the 128 bytes of memory on a typical 8-bit CPU.

2.5 Cryptanalysis of Salsa20

While Bernstein presented Salsa20, he announced a prize money of $1000 for the best attack. Crowley won the first prize in 2005. In this section we present a few important attacks including that “award winning” attack briefly.

2.5.1 Truncated differential cryptanalysis of five rounds of Salsa20

In [45], Crowley posted an attack on Salsa20/5. This attack uses many clusters of truncated differentials and requires $2^{165}$ work and $2^6$ plaintexts. In this paper, Crowley attack the Salsa20 PRF directly; the resulting attack on the Salsa20 stream cipher follows straightforwardly. Though many techniques of block cipher cryptanalysis are applicable to Salsa20, it has several features to defeat these techniques. First, the large block size allows for rapid diffusion without penalty of speed. Second, the attacker can control only four words of the sixteen-word input to the block cipher stage. Nevertheless, he is able to construct an attack based on multiple truncated differentials which breaks five rounds of the cipher. The attack works forwards from a small known input difference to a biased it 3 rounds later, and works 2 rounds backwards from an output after guessing 160 relevant key bits.

Crowley received the $1000 prize and presented his attack at the ECRYPT State-of-the-Art of Stream Ciphers workshop in Leuven.

2.5.2 Non-randomness in Salsa20

In [1], Fischer, Meier, Berbain, Biasee, and Robshaw reported a $2^{177}$ operation attack on Salsa20/6 and even a much faster attack on Salsa20/5, which clearly breaks Salsa20/5. In this paper, they analyze the key/IV setup of the eSTREAM Phase 2 candidates Salsa20 and TSC – 4. In the case of Salsa20 they demonstrate a key recovery attack on six rounds and observe non-randomness after seven. They investigate the initialization of Salsa20. They consider a set of well-chosen inputs $(K, IV)$ and compute the outputs $F(K, IV)$. Under an appropriate measure they aim to detect non-random behavior in the output. They accepted
the fact that, nothing in this paper affects the security of the full version of the cipher. However
they expect that the key can be recovered from five rounds of 128-bit Salsa20 with around $2^{81}$
operations and six rounds of $2^{56}$-bit Salsa20 with around $2^{177}$ operations. Both attacks would
require very moderate amount of texts. If we consider related-key attacks then the security of
seven rounds of 256-bit Salsa20 might be in question with around $2^{217}$ operations. However,
given divided opinions on such an attack model, they prefer to observe that a statistical
weakness has been observed over seven rounds. While they anticipate some progress, they are
doubtful that many more rounds can be attacked using the methods of this paper. Thus they
concluded with the fact that: Salsa20 still appears to be a conservative design.

2.5.3 Differential Cryptanalysis of Salsa20/8

In [120], Tsunoo, Saito, Kubo, Suzaki, and Nakashima reported a $2^{184}$ operation attack
on Salsa20/7 (and a much faster attack on Salsa20/6, clearly breaking Salsa20/6) at the
ECRYPT State-of-the-art of Stream Ciphers workshop in Bochum. It is reported that there
is a significant bias in the differential probability for Salsa20’s 4th round internal state. It
is further shown that using this bias, it is possible to break the 256-bit secret key 8-round
reduced Salsa20 model with a lower computational complexity than an exhaustive key search.
The cryptanalysis method exploits characteristics of addition, and succeeds in reducing the
computational complexity compared to previous methods. The attack works forwards from a
small known input difference to a biased bit 4 rounds later, and works 3 rounds backwards
from an output after guessing 171 highly relevant key bits.

2.5.4 On the Salsa20 Core Function

In [38], Julio Cesar Hernandez-Castro, Juan M. E. Tapiador, and Jean-Jacques Quisquater
pointed out some weaknesses in the Salsa20 core function that could be exploited to obtain up
to $2^{31}$ collisions for its full (20 rounds) version. They first find an invariant for its main building
block, the quarterround function, that is then extended to the rowround and columnround
functions. This allows them to find an input subset of size $2^{32}$ for which the Salsa20 core
behaves exactly as the transformation $f(x) = 2x$. An attacker can take advantage of this for
constructing $2^{31}$ collisions for any number of rounds. They finally show another weakness in
the form of a differential characteristic with probability one that proves that the Salsa20 core
does not have 2nd pre-image resistance.
2.6 Conclusion

After all the discussion, in conclusion it can be confidently said that, Salsa20 is a very strong cipher. In spite of a few attacks we have discussed here, the full round of Salsa20/12 (which is practically the version they have in eSTREAM portfolio) is still considered to be pretty secure. But, definitely it leaves room for researcher to work on the cryptanalysis of it in future. Due to its structural simplicity it is considered to be one of the most popular cipher in the eSTREAM portfolio.

2.7 A Simple C Implementation of Salsa20

```c
/* Developer : Pratyay Mukherjee
   Email: pratyay85@gmail.com */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define Modulus 4294967296
#define ROUND 1
#define KEYSIZE 32
#define bigint unsigned long long int

const bigint sigma[4][4]={{101, 120, 112, 97},{110, 100, 32, 51},{50, 45, 98, 121},{116, 101, 32, 107}}; // The constants
const bigint tau[4][4]={{101, 120, 112, 97},{110, 100, 32, 49},{54, 45, 98, 121},{116, 101, 32, 107}}; // The constants
unsigned long long int
X_dec[8], G_dec[8], Key_dec[8], C_dec[8], A_dec[8], cy=0, IV[4]; // Defining the Global Variables

/* lrot_dec() rotates a 32-bit decimal number left by specified position */
bigint lrot_dec(bigint var, unsigned int rot)
{
    int i, j, k;
    bigint t1, t2;
    unsigned char * temp, *result, *temp1;
    t1=(var<<rot)&(Modulus-1);
```
t2=(var>>(32-rot))&((Modulus-1)>>)(32-rot));

return (t1+t2)%Modulus;
}

/*qround() takes a 4-word sequence (as an array of bigint) as input and returns a
4-word sequence (as an array of bigint) as output after performing Quarterround
operation*/
bigint* qround(bigint* s)
{
    int i;
    bigint* t;
    unsigned char* temp;
    temp = (unsigned char*)malloc(sizeof(unsigned char)*32);
    t = (bigint*)malloc(sizeof(bigint)*4);

    t[1] = s[1] ^ lrot_dec((s[0]+s[3])&(Modulus-1),7);
    t[2] = s[2] ^ lrot_dec((s[0]+t[1])&(Modulus-1),9);
    t[0] = s[0] ^ lrot_dec((t[2]+t[3])&(Modulus-1),18);

    return t;
}

/*rowround() takes a 16-word sequence (as an array of bigint) as input and returns
a 16-word sequence (as an array of bigint) as output after performing Rowround
operation*/
bigint* rowround(bigint* s)
{
    bigint *temp,*out,*t;
    temp=(bigint*)malloc(sizeof(bigint)*4);
    t=(bigint*)malloc(sizeof(bigint)*16);

    temp[0]=s[0];
    temp[1]=s[1];
    temp[2]=s[2];
    temp[3]=s[3];

    out=qround(temp);

    t[0]=out[0];
    t[1]=out[1];
    t[2]=out[2];
2.7. A Simple C Implementation of Salsa20

```c
    t[3]=out[3];
    temp[0]=s[5];
    temp[1]=s[6];
    temp[2]=s[7];
    temp[3]=s[4];
    out=qround(temp);  
    t[5]=out[0];
    t[6]=out[1];
    t[7]=out[2];
    t[4]=out[3];
    temp[0]=s[10];
    temp[1]=s[11];
    temp[2]=s[8];
    temp[3]=s[9];
    out=qround(temp);  
    t[10]=out[0];
    t[11]=out[1];
    t[8]=out[2];
    t[9]=out[3];
    temp[0]=s[15];
    temp[1]=s[12];
    temp[2]=s[13];
    temp[3]=s[14];
    out=qround(temp);  
    t[15]=out[0];
    t[12]=out[1];
    t[13]=out[2];
    t[14]=out[3];

    return t;
```

/*colround() takes a 16-word sequence (as an array of bigint) as input and returns 
a 16-word sequence (as an array of bigint) as output after performing
Columnround operation*/

bigint* colround(bigint* s)
{
     bigint *temp,*out,*t;
     temp=(bigint* )malloc(sizeof(bigint)*4);

     t=(bigint*)malloc(sizeof(bigint)*16);

     temp[0]=s[0];
     temp[1]=s[4];
     temp[2]=s[8];
     temp[3]=s[12];

     out=qround(temp);

     t[0]=out[0];
     t[4]=out[1];
     t[8]=out[2];
     t[12]=out[3];

     temp[0]=s[5];
     temp[1]=s[9];
     temp[2]=s[13];
     temp[3]=s[1];

     out=qround(temp);

     t[5]=out[0];
     t[9]=out[1];
     t[13]=out[2];
     t[1]=out[3];

     temp[0]=s[10];
     temp[1]=s[14];
     temp[2]=s[2];
     temp[3]=s[6];

     out=qround(temp);

     t[10]=out[0];
     t[14]=out[1];
     t[2]=out[2];
     t[6]=out[3];
```
temp[0]=s[15];
temp[1]=s[3];
temp[2]=s[7];
temp[3]=s[11];

out=qround(temp);

t[15]=out[0];
t[3]=out[1];
t[7]=out[2];
t[11]=out[3];

return t;
}

/*doubleround() takes a 16-word sequence (as an array of bigint) as input and
returns a 16-word sequence (as an array of bigint) as output after performing
Doublerounf operation which is nothing but a Columnround() followed by a
Rowround()*/
bigint* doubleround(bigint*s)
{
    return(rowround(colround(s)));
}

/*littleendian() takes a 4-byte sequence (array of bigint) and outputs a word*/
bigint* littleendian(bigint*s)
{
}

/*lit_end_inv() is nothing but the inverse opeartion of littleendiian() that is
taking an word as input it outputs a 4-byte sequence (array of bigint)*/
bigint* lit_end_inv(bigint s)
{
    int i,j;
    bigint*t;
    t=(bigint*)malloc(sizeof(bigint)*4);

    for(i=0;i<4;++i)
    {
        t[i]=s%256;
    }
```
s=s/256;
}
return t;
}

/*copy_bigint() takes two array of bigint and their size as input and copy the
source array to the target */
void copy_bigint(bigint*s,bigint*t,int size)
{
    int i;
    for(i=0;i<size;++i)
        s[i]=t[i];
}

/*salsa_hash() takes 64-byte sequence as input and produces another 64-bytes output
after performing several operations*/
bigint* salsa_hash(bigint*s)
{
    int i,j,k;
    bigint*x,*z,*t,*temp,*y;
    x=(bigint*)malloc(sizeof(bigint)*16);
    y=(bigint*)malloc(sizeof(bigint)*16);
    t=(bigint*)malloc(sizeof(bigint)*16);
    temp=(bigint*)malloc(sizeof(bigint)*4);

    //Step1: Applying littleendian()
    for(i=0;i<16;i++)
    {
        temp[0]=s[4*i];
        temp[1]=s[4*i+1];
        temp[2]=s[4*i+2];
        temp[3]=s[4*i+3];
        x[i]=littleendian(temp);
    }

    for(i=0;i<4;++i)
    {
        temp[i]=0;
    }
    copy_bigint(y,x,16);//copying the values in a another array
    for(i=0;i<10;++i)
    {
z=doubleround(y);//performing doubleround 10 times on one copy
copy_bigint(y,z,16);

}
for(i=0;i<16;++i)
{

temp=lit_end_inv((z[i]+x[i]));  //applying lit_end_inv()
t[4*i]=temp[0];
t[4*i+1]=temp[1];
t[4*i+2]=temp[2];
t[4*i+3]=temp[3];
}

return t;
}

/*It takes three inputs. The first input is a 16-byte sequence. The second input may
be 16-byte or 32-bytes and this is determined by the third input which acts as
an indicator. It anyway outputs a 64-byte sequence after performing some
operations and also using the constant values defined atop */
b bigint* salsa_exp(bigint* n,b bigint k[2][16],int ind)
{
  int i,j;
  bigint*s,*t;
  s=(bigint*)malloc(sizeof(bigint)*64);
  if(ind==2)//If second input is 32-byte.
  {
    j=0;
    i=0;
    for(j=0;j<4;j++)
    {
      s[i++]=sigma[0][j];
    }
    for(j=0;j<16;++j)
    {
      s[i++]=k[0][j];
    }
    for(j=0;j<4;j++)
    {
      s[i++]=sigma[1][j];
    }
  }
  for(j=0;j<16;++j)
  {
    s[i++]=n[j];
  }
for(j=0; j<16; j++)
{
    s[i++]=n[j];
}
for(j=0; j<4; j++)
{
    s[i++]=sigma[2][j];
}
for(j=0; j<16; j++)
{
    s[i++]=k[1][j];
}
for(j=0; j<4; j++)
{
    s[i++]=sigma[3][j];
}

else if(ind==1)//If second input is 16-byte
{
    j=0;
    i=0;
    for(j=0; j<4; j++)
    {
        s[i++]=tau[0][j];
    }

    for(j=0; j<16;++j)
    {
        s[i++]=k[0][j];
    }
    for(j=0; j<4; j++)
    {
        s[i++]=tau[1][j];
    }
    for(j=0; j<16; j++)
    {
        s[i++]=n[j];
    }
    for(j=0; j<4; j++)
    {
        s[i++]=tau[2][j];
    }
    for(j=0; j<16; j++)
    {
        s[i++]=n[j];
    }
}
2.7. A Simple C Implementation of Salsa20

```c
{
    s[i++]=k[0][j];
}
for(j=0;j<4;j++)
{
    s[i++]=tau[3][j];
}

else
{
    puts("Can not expand !!!");
    exit(1);
}
t=salsa_hash(s);
return t;

//match() takes two bigint array as input and outputs 1 if they are same and 0
//otherwise*/
unsigned int match(bigint* s,bigint*t,int size)
{
    int i,j,k;
    for(i=0;i<size;++i)
    {
        if(s[i]!=t[i])
            return 0;
    }
    return 1;
}

/*main function */
int main()
{
    int i,j,ii,jj,kk;
    bigint *t,tt;

    /*Several Examples given in the spec are tested here for all the functions
    individually, Functions which are tested in a particular input are given
    below*/
```
bigint in1[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000}, // qround()
out1[4] = {0x00000000, 0x00000000, 0x00000000, 0x00000000}, // qround()
in2[4] = {0x00000000, 0x00000000, 0x00000001, 0x00000000}, // qround()
out2[4] = {0x80040000, 0x00000000, 0x00000001, 0x00002000}, // qround()
in3[4] = {0xd3917c5b, 0x55f1c407, 0x52a58a7a, 0x8f887a3b}, // qround()
out3[4] = {0x3e2f308c, 0xd90a8f36, 0x6ab2a923, 0x2883524c}, // qround()

in4[16] = {0x08521bd6, 0x1fe88837, 0xbb2aa576, 0x3aa26365,
0xc54c6a5b, 0x2fc74c2f, 0x6dd39cc3, 0xda0a64f6,
0x90a2f23d, 0x067f95a6, 0x06b35f61, 0x41e4732e,
0xe859c100, 0xea4d84b7, 0x0f619bff, 0xbcc6e965a}, // rowround()
out4[16] = {0xa890d39d, 0x65d71596, 0xe9487daa, 0xc8ca6a86,
0x949d2192, 0x764b7754, 0xe408d9b9, 0x7a41b4d1,
0x3402e183, 0x3c3af432, 0x50669f96, 0xd89ef0a8,
0x0040ede5, 0xb545fbce, 0x2d57ed4f, 0x1818882d}, // rowround()
in5[16] = {0x08521bd6, 0x1fe88837, 0xbb2aa576, 0x3aa26365,
0xc54c6a5b, 0x2fc74c2f, 0x6dd39cc3, 0xda0a64f6,
0x90a2f23d, 0x067f95a6, 0x06b35f61, 0x41e4732e,
0xe859c100, 0xea4d84b7, 0x0f619bff, 0xbcc6e965a}, // colround()
out5[16] = {0x8c9d190a, 0xce8e4c90, 0x1ef8e9d3, 0x1326a71a,
0x90a20123, 0xead3c4f3, 0x63a09a10, 0xf0708d69,
0x789b010c, 0xd195a681, 0x8eb7d5504, 0xa774135c,
0x481c2027, 0x53a8e4b5, 0x441f89c5, 0x3f78c9c8}, // colround()
in6[16] = {0xde501066, 0x6f9ebbf7, 0xe4fbb9db, 0x454e3f57,
0xb75540d3, 0x43e93a4c, 0x3a6f2a0a, 0x726d6b36,
0x9243f484, 0x91451e8, 0x4fa9d247, 0x8dc8dee11,
0x054bf545, 0x254dd653, 0xd94216b6, 0x67b276c1}, // doubleround()
out6[16] = {0xccaaaf672, 0x23d960f7, 0x9153e63a, 0xcd96ad0,
0x50440492, 0xf07cad19, 0xae344aa0, 0xf4fd2fda,
0xca531c29, 0x8e7943db, 0xac1600cd, 0xd503ca00,
0xa74b2ad6, 0xbc331c5c, 0x1dda24c7, 0xeee92877}, // doubleround()
in7[4] = {86, 75, 30, 9}, // littleendian()
out7 = 0x091e4b56, // littleendian()
in8[64] = {211, 159, 13, 115, 76, 55, 82, 183, 3, 117, 222,
37, 191, 187, 234, 136,
2.7. A Simple C Implementation of Salsa20

49, 237, 179, 48, 1, 106, 178, 219, 175, 199, 166, 48, 86, 16, 179, 207,
31, 240, 32, 63, 15, 83, 93, 161, 116, 147, 48, 113, 238, 55, 204, 36,
79, 201, 235, 79, 3, 81, 156, 47, 203, 26, 244, 243, 88, 118, 104, 54}, //salsa_hash()
out8[64]={109, 42, 178, 168, 156, 240, 248, 238, 168, 196, 190, 203,
26, 110, 170, 154,
29, 29, 150, 26, 150, 30, 235, 249, 190, 163, 251, 48, 69, 144, 51, 57,
118, 40, 152, 157, 180, 57, 27, 94, 107, 42, 236, 35, 27, 111, 114, 114,
219, 236, 232, 135, 111, 155, 110, 18, 24, 232, 95, 158, 179, 19, 48, 202}, //salsa_hash()
in9[64]= {88, 118, 104, 54, 79, 201, 235, 79, 3, 81, 156, 47, 203,
26, 244, 243,
191, 187, 234, 136, 211, 159, 13, 115, 76, 55, 82, 183, 3, 117, 222, 37,
86, 16, 179, 207, 49, 237, 179, 48, 1, 106, 178, 219, 175, 199, 166, 48,
238, 55, 204, 36, 31, 240, 32, 63, 15, 83, 93, 161, 116, 147, 48, 113}, //salsa_hash()
out9[64]={179, 19, 48, 202, 219, 236, 232, 135, 111, 155, 110, 18, 24, 232,
95, 158,
26, 110, 170, 154, 109, 42, 178, 168, 156, 240, 248, 238, 168, 196, 190, 203,
69, 144, 51, 57, 29, 29, 150, 26, 150, 30, 235, 249, 190, 163, 251, 48,
27, 111, 114, 114, 118, 40, 152, 157, 180, 57, 27, 94, 107, 42, 236, 35}, //salsa_hash()
out10[64]={ 69, 37, 68, 39, 41, 15, 107, 193, 255, 139, 122,
6, 170, 233, 217, 98,
89, 144, 182, 106, 21, 51, 200, 65, 239, 49, 222, 34, 215, 114, 40, 126,
104, 197, 7, 225, 197, 153, 31, 2, 102, 78, 76, 176, 84, 245, 246, 184,
177, 160, 133, 130, 6, 72, 149, 119, 192, 195, 132, 236, 234, 103, 246, 74}, //salsa_exp() with 32-byte key
out11[64]={ 39, 173, 46, 248, 30, 200, 82, 17, 48, 67, 254, 239, 37, 18,
13, 247,
241, 200, 61, 144, 10, 55, 50, 185, 6, 47, 246, 253, 143, 86, 187, 225,
134, 85, 110, 246, 161, 163, 43, 235, 231, 94, 171, 51, 145, 214, 112, 29,
14, 232, 5, 16, 151, 140, 183, 141, 171, 9, 122, 181, 104, 182, 177, 193}; //salsa_exp() with 16-byte key

bigint
k[2][16]={ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16},
{201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216}};
key
bigint n[16]={101, 102, 103, 104, 105, 106, 107, 108, 0, 0, 0, 0, 0, 0, 0, 0}; //nonce used

to generate Pseudorandom stream.
bigint
used for testing purpose
bigint half[8];
/*Testing qround*/
puts("Testing Quarterround:");
t=qround(in1);

if (match(t,out1,4))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

t=qround(in2);

if (match(t,out2,4))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

t=qround(in3);

if (match(t,out3,4))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

/*Testing rowround*/
puts("Testing Rowround:");

t=rowround(in4);

if (match(t,out4,16))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

/*Testing colround*/
puts("Testing Columnround:");
t=colround(in5);

if (match(t,out5,16))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

/*Testing doubleround*/
puts("Testing Douleround:");
t=doubleround(in6);

if(match(t,out6,16))
    puts("SUCCESS!!");
else{
    puts("FAILURE");
}

/*Testing littleendian*/
puts("Testing Littleendian:");
tt=littleendian(in7);

if(out7==tt)
    puts("SUCCESS!!");
else{
    puts("FAILURE");
}

/*Testing salsa_hash*/
puts("Testing Salsa_hash:");

t=salsa_hash(in8);

if(match(t,out8,64))
    puts("SUCCESS!!");
else{
    puts("FAILURE");
}

t=salsa_hash(in9);

if(match(t,out9,64))
    puts("SUCCESS!!");
else{
    puts("FAILURE");
}
/*Testing salsa_exp*/
puts("Testing Salsa expansion:");
t=salsa_exp(n_test,k,2);
if(match(t,out10,64))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

t=salsa_exp(n_test,k,1);
if(match(t,out11,64))
puts("SUCCESS!!");
else{
    puts("FAILURE");
}

puts("Output Stream:");
for(ii=0;ii<ROUND;++ii)
{
    puts(" ");
    j=ii;
    jj=0;
    while(j>0)
    {
        n[7+jj]=j%256; //changing last half of the nonce value.
        ++jj;
        j=j/256;
    }

t=salsa_exp(n,k,KEYSIZE/16);
for(i=0;i<64;++i)
{

    printf("%x\t",t[i]);

}
puts(" ");
}

return 0;
2.7. A Simple C Implementation of Salsa20

}
Chapter 3

HC-128

3.1 Introduction

The HC-128 algorithm is a software-efficient (profile 1), synchronous stream cipher designed by Hongjun Wu. The cipher makes use of a 128-bit key and 128-bit initialization vector; its secret state consists of two tables, each with 512 registers of 32 bits in length. At each step, one register of one of the tables is updated using a non-linear feedback function, while one 32-bit output is generated from the non-linear output filtering function. The cipher specification states that $2^{64}$ key-stream bits can be generated from each key/IV pair.

The HC-128 stream cipher offers a very impressive performance in software applications where one wishes to encrypt large streams of data. For example, it can encrypt at the speed of 3.52 cycles/byte on Pentium M processors, or 2.86 cycles/byte on AMD Athlon 64 processors. However, since HC-128 is a table-driven algorithm, there is a cost in the time to initialize the cipher (key and IV setup requires around 27,300 clock cycles). As a result, for applications that require frequent reinitialization, there can be a significant performance penalty. This indicates that the HC-128 stream cipher should be a very strong performer for link-level streamed applications, but a relatively poor performer for typical packetized applications.

3.2 Specifications of HC-128

The specification of HC-128 is described in detail in [56]. Here we briefly describe the specs. HC-128 consists of two secret tables, each one with 512 32-bit elements. At each step we update one element of a table with non-linear feedback function. All the elements of the two tables get updated every 1024 steps. At each step, one 32-bit output is generated from the non-linear output filtering function. From a 128-bit key and a 128-bit initialization vector,
HC-128 generates keystream with length up to $2^{64}$ bits.

### 3.2.1 Notation

We start with the notation we use in this chapter. Notations are described in the following table 3.1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td>addition mod $2^{32}$</td>
</tr>
<tr>
<td>$\Box$</td>
<td>subtraction mod 512</td>
</tr>
<tr>
<td>$\oplus$</td>
<td>bitwise xor</td>
</tr>
<tr>
<td>$|$</td>
<td>concatenation</td>
</tr>
<tr>
<td>$\ll / \gg$</td>
<td>left/right shift</td>
</tr>
<tr>
<td>$\ll / \gg$</td>
<td>left/right rotate</td>
</tr>
</tbody>
</table>

Table 3.1: Symbols and their meaning used in this chapter

Two tables $P$ and $Q$ are used in HC-128. The key and the initialization vector of HC-128 are denoted as $K$ and $IV$. We denote the keystream being generated as $s$. The details are provided in table 3.2.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>A table with 512 32-bit elements. Each element is denoted as $P[i]$, with $0 \leq i \leq 511$.</td>
</tr>
<tr>
<td>$Q$</td>
<td>A table with 512 32-bit elements. Each element is denoted as $Q[i]$, with $0 \leq i \leq 511$.</td>
</tr>
<tr>
<td>$k$</td>
<td>The 128-bit key of HC-128.</td>
</tr>
<tr>
<td>$IV$</td>
<td>The 128-bit initialization vector of HC-128.</td>
</tr>
<tr>
<td>$s$</td>
<td>The keystream being generated from HC-128. The 32-bit output of the $i$-th step is denoted as $s_i$. $s = s_0 \parallel s_1 \parallel s_3 \parallel \ldots$.</td>
</tr>
</tbody>
</table>

Table 3.2: Notations and their explanation

HC-128 is the simplified version of HC-256 (detail in [125]) for 128-bit security. There are six functions being used in HC-256. $f_1(x)$ and $f_2(x)$ are the same as the $\sigma_0^{(256)}(x)$ and $\sigma_1^{(256)}(x)$ being used in the message schedule of SHA-256 (details in [108]). For $h_1(x)$, $Q$ is used as a S-box, whereas $P$ is used in the same purpose for $h_2(x)$. The functions are described in the following table 3.3

Where $x$ is a 32-bit word and $x = x_3 \parallel x_2 \parallel x_1 \parallel x_0$. 
### 3.3 Security Properties of HC-128

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x)$</td>
<td>$(x \gg 7) \oplus (x \gg 18) \oplus (x \gg 3)$</td>
</tr>
<tr>
<td>$f_2(x)$</td>
<td>$(x \gg 17) \oplus (x \gg 19) \oplus (x \gg 10)$</td>
</tr>
<tr>
<td>$g_1(x, y, z)$</td>
<td>$((x \gg 10) \oplus (z \gg 23)) + (y \gg 8)$</td>
</tr>
<tr>
<td>$g_2(x, y, z)$</td>
<td>$((x \ll 10) \oplus (z \ll 23)) + (y \ll 8)$</td>
</tr>
<tr>
<td>$h_1(x)$</td>
<td>$Q[x_0] + Q[256 + x_2]$</td>
</tr>
<tr>
<td>$h_2(x)$</td>
<td>$P[x_0] + P[256 + x_2]$</td>
</tr>
</tbody>
</table>

Table 3.3: The Functions and their description

#### 3.2.2 Key and IV Set up

The process starts with the initialization \textit{i.e.} with the Key and IV setup algorithms. In this step, the key and the initialization vector are expanded into $P$ and $Q$. and the cipher is run 1024 steps. The setup algorithm is described by the pseudo-code in Algorithm 4. Let $K = K_0 \parallel K_1 \parallel K_2 \parallel K_3$ and $IV = IV_0 \parallel IV_1 \parallel IV_2 \parallel IV_3$. Also, let, $K_{i+4} = K_i$ and $IV_{i+4} = IV_i$ for $0 \leq i \leq 4$.

The initialization process completes and the cipher is ready to generate keystream.

#### 3.2.3 Keystream Generation

At each step, one element of a table is updated and one 32-bit output is generated. Each S-box is used to generate only 512 outputs, then it is updated in the next 512 steps. The keystream generation algorithm of HC-128 is given in algorithm 5.

#### 3.3 Security Properties of HC-128

According to the author’s note in [56] the security analysis of HC-128 is similar to that of HC-256. The output and feedback functions of HC-128 are non-linear, so it is impossible to apply the fast correlation attacks and algebraic attacks to recover the secret key of HC-128. The large secret S-box of HC-128 is updated during the keystream generation process, so it is very difficult to develop linear relations linking the input and output bits of the S-box. In this section, we analyze the period of HC-128, the security of the secret key and the initialization process and the randomness of the keystream.
Algorithm 4 KEY-IV-SETUP

{Step-1: Expanding key and IV into an array $W_i$ ($0 \leq i \leq 1279$)}

for $i = 0 \to 7$ do
  $W_i \leftarrow K_i$
end for

for $i = 8 \to 15$ do
  $W_i \leftarrow IV_{i-8}$
end for

for $i = 16 \to 1279$ do
  $W_i \leftarrow f_2(W_{i-2}) + W_{i-7} + f_1(W_{i-15}) + W_{i-16} + i$
end for

{Step-2: Update the tables $P$ and $Q$ with the array $W$}

for $i = 0 \to 511$ do
  $P[i] \leftarrow W_{i+256}$
  $Q[i] \leftarrow W_{i+768}$
end for

{Step-3: Run the cipher for 1024 steps and use the outputs to replace the table elements....}

for $i = 0 \to 511$ do
  $P[i] \leftarrow (P[i] + g_1(P[i \boxplus 3], P[i \boxplus 10], P[i \boxplus 511])) \oplus h_1(p[i \boxplus 12])$
  $Q[i] \leftarrow (Q[i] + g_2(P[i \boxplus 3], P[i \boxplus 10], P[i \boxplus 511])) \oplus h_2(p[i \boxplus 12])$
end for

Algorithm 5 KEYSTREAM-GENERATION

{Assume $N$ bits are required....}

for $i = 0 \to N$ do
  $j = i \mod 512$
  if $(i \mod 1024) < 512$ then
    $P[j] \leftarrow (P[j] + g_1(P[j \boxplus 3], P[j \boxplus 10], P[j \boxplus 511]))$
    $s_i = h_1(P[j \boxplus 12] \oplus P[j])$
  else
    $Q[j] \leftarrow (Q[j] + g_1(Q[j \boxplus 3], Q[j \boxplus 10], Q[j \boxplus 511]))$
    $s_i = h_2(Q[j \boxplus 12] \oplus Q[j])$
  end if
  $i \leftarrow i + 1$
end for
3.3. Security Properties of HC-128

3.3.1 Period length

The 32778-bit state of HC-128 ensures that the period of the keystream is extremely large. But the exact period of HC-128 is difficult to predict. The average period of the keystream is estimated to be much more than $2^{256}$. The large number of states also eliminates the threat of the time-memory-data trade-off attack on stream ciphers (see [24] for details).

3.3.2 Security of the secret key

The author notices that the output function and the feedback function of HC-128 are non-linear. The non-linear output function leaks small amount of partial information at each step. The non-linear feedback function ensures that the secret key can not be recovered from those leaked partial information.

3.3.3 Security of the initialization process (Key-IV setup)

The initialization process of the HC-128 consists of two stages, as given in subsection 3.2.2. The key and IV are expanded into $P$ and $Q$. At this stage, every bit of the key/IV affects all the bits of the two tables and any difference in the related keys/IVs results in uncontrollable differences in $P$ and $Q$. Note that the constants in the expansion function at this stage play significant role in reducing the effect of related keys/IVs. After the expansion, the cipher is run for 1024 steps and using the outputs to update the $P$ and $Q$. After the initialization process, the expectation is that any difference in the keys/IVs would not result in biased keystream.

3.3.4 Randomness of the keystream

Since the key-size of HC-128 is 128 bit, clearly the distinguishing attack on HC-128 requires more than $2^{128}$ outputs. The analysis is given below.

We recall that at the $i$-th step, if $(i \mod 1024) < 512$, the table $P$ is updated as:

$$P[i \mod 512] \leftarrow (P[i \mod 512] + g_1(P[i \oplus 3], P[i \oplus 10], P[i \oplus 511]))$$

(3.1)

We know that, $s_i = h_1(P[i \oplus 12]) \oplus P[i \mod 512]$ for $10 \leq (i \mod 1024) \leq 511$, this feedback function can be written alternatively as:

$$s_i \oplus h_1(z_i) = (s_{i-1024} \oplus h'_1(z_{i-1024}))$$

(3.2)

$$+ g_1(s_{i-3} \oplus h_1(z_{i-3}), s_{i-10} \oplus h_1(z_{i-10}), s_{i-1023} \oplus h'_1(z_{i-1023}))$$

where $h_1(x)$ and $h'_1(x)$ indicate two different functions since they are related to different S-boxes; $z_j$ denotes the $P[j \oplus 12]$ at the $j$-th step.
Similarly, for $1024 \times 10 < i, j < 1024 \times 10 + 511$ and $j \neq i$, we obtain.

$$s_{j}^{0} \oplus s_{j-1024}^{0} \oplus s_{j-10}^{8} \oplus s_{j-1023}^{23} = (h_{1}(z_{j}))^{0} \oplus (h'_{1}(z_{j-1024}))^{0} \oplus (h_{1}(z_{j-3}))^{10} \oplus (h_{1}(z_{j}))^{8} \oplus (h'_{1}(z_{j-1023}))^{23}$$

(3.4)

For the left sides of equation 3.3 and equation 3.4 to be equal, i.e. for the following equation:

$$s_{i}^{0} \oplus s_{i-1024}^{0} \oplus s_{i-10}^{8} \oplus s_{i-1023}^{23} = s_{j}^{0} \oplus s_{j-1024}^{0} \oplus s_{j-10}^{8} \oplus s_{j-1023}^{23}$$

(3.5)
to hold, we require that,

$$s_{i}^{0} \oplus s_{i-1024}^{0} \oplus s_{i-10}^{8} \oplus s_{i-1023}^{23} = s_{j}^{0} \oplus s_{j-1024}^{0} \oplus s_{j-10}^{8} \oplus s_{j-1023}^{23}$$

(3.6)

And approximating equation 3.6, we get

$$H(x_{1}) = H(x_{2})$$

(3.7)

where $H$ denotes a random secret 80-bit-to-1-bit S-box, $x_{1}$ and $x_{2}$ are two 80-bit random inputs, $x_{1} = z_{i} \parallel z_{i-3} \parallel z_{i-10} \parallel z_{i-1023}$ and $x_{2} = z_{j} \parallel z_{j-3} \parallel z_{j-10} \parallel z_{j-1023}$ where $\parallel$ indicates the concatenation of the least significant byte and the second most significant byte of $z$. We state following theorem without proof (stated and proved in [56]) which gives the collision rate of the outputs of $H(x)$.

**Theorem 3.3.1.** Let $H$ be an $m$-bit-to-$n$-bit S-box and all those $n$-bit elements are randomly generated, where $m \geq n$. Let $x_{1}$ and $x_{2}$ be two $m$-bit random inputs to $H$. Then $H(x_{1}) = H(x_{2})$ with probability $2^{-m} + 2^{-n} - 2^{-m-n}$.

According to the Theorem 3.3.1, equation 3.7 holds with probability $\frac{1}{2} + 2^{-81}$. So equation 3.5 holds with probability $\frac{1}{2} + 2^{-81}$. After testing the validity of $2^{164}$ equations 3.5, the output of the cipher can be distinguished from random signal with success rate 0.9772 (with false negative rate and false positive rate as 0.0228). Note that only about $2^{17}$ equations 3.5 can be obtained from every 512 outputs this distinguishing attack requires about $2^{156}$ outputs.
We note that the attack above only deals with the least significant bit in equation 3.1. It may be possible to consider the rest of the 31 bits bit-by-bit. But due to the effect of the two \textquoteleft+\textquoteright operations in the feedback function, the attack exploiting those 31 bits is not as effective as that exploiting the least significant bit. Thus more than $2^{151}$ outputs are needed in this distinguishing attack.

It may be possible that the distinguishing attack against HC-128 can be improved in the future. However, it is very unlikely that security goal of the designer can be breached since the security margin is extremely large. They conjecture that it is computationally impossible to distinguish $2^{64}$ bits keystream of HC-128 from random bitstream.

### 3.4 Implementation and Performance of HC-128

The optimized implementation of HC-128 is similar to that of HC-256. On the Pentium M processor, the speed of HC-128 reaches 3.05 cycles/byte, while the speed of HC-256 is about 4.4 cycles/byte.

#### 3.4.1 The optimized implementation of HC-128

In the optimized code, loop unrolling is used and only one branch decision is made for every 16 steps. The details of the implementation are given below. The feedback function of $P$ is given as:

$$
P[i \mod 512] = P[i \mod 512] + P[i \mod 10] + g_1(P[i \mod 3], P[i \mod 511]) \tag{3.8}$$

A register $X$ containing 16 elements is introduced for $P$. If $(i \mod 1024) < 512$ and $i \mod 16 = 0$, then at the beginning of the $i$-th step, $X[j] = P[(i - 16 + 15) \mod 512]$. In the 16 steps starting from the $i$-0th step, the $P$ and $X$ are updated as follows:
\[ P[i] = P[i] + g_1(X[13], X[6], P[i+1]); \]  
\[ X[0] = P[i]; \]  
\[ P[i+1] = P[i+1] + g_1(X[14], X[7], P[i+2]); \]  
\[ X[1] = P[i+1]; \]  
\[ P[i+2] = P[i+2] + g_1(X[15], X[8], P[i+3]); \]  
\[ X[2] = P[i+2]; \]  
\[ P[i+3] = P[i+3] + g_1(X[0], X[9], P[i+4]); \]  
\[ X[3] = P[i+3]; \]  
\[ \ldots \]  
\[ P[i+14] = P[i+14] + g_1(X[11], X[4], P[i+15]); \]  
\[ X[14] = P[i+14]; \]  
\[ P[i+15] = P[i+15] + g_1(X[12], X[5], P[(i+1) \mod 512]); \]  
\[ X[15] = P[i+15]; \]  

Note that at the \( i \)-th step, two elements of \( P[i \oplus 10] \) and \( P[i \oplus 3] \) can be obtained directly from \( X \). Also for the output function \( s_i = h_1(P[i \oplus 12] \oplus P[i \mod 1024]) \), the \( P[i \oplus 12] \) can be obtained from \( X \). In this implementation, there is no need to compute \( i \oplus 3 \), \( i \oplus 10 \) and \( i \oplus 12 \).

A register \( Y \) with 16 elements is used in the implementation of the feedback function of \( Q \) in the same way as that given above.

### 3.4.2 The performance of HC-128

**Encryption Speed**

The designer uses the C codes (available in [56]) submitted to the eSTREAM to measure the encryption speed. The processor used in the measurement is the Intel Pentium M (1.6 GHz, 32 KB Level 1 cache, 2 MB Level 2 cache). Using the eSTREAM performance testing framework, the highest encryption speed of HC-128 is 3.05 cycles/byte with the compiler gcc (there are three optimization options leading to this encryption speed: k8 O3-ual-ofp, prescott O2-ofp and athon O3-ofp). Using the Intel C++ Compiler 9.1 in Windows XP (SP2), the speed is 3.3 cycles/byte. Using the Microsoft Visual C++ 6.0 in Windows XP (SP2), the speed is 3.6 cycles/byte.
Initialization Process

The key setup of HC-128 requires about 27,300 clock cycles. There are two large S-boxes in HC-128. In order to eliminate the threat of related key/IV attack, the tables should be updated with the key and IV thoroughly and this process requires a lot of computations. It is thus undesirable to use HC-128 in the applications where key (or IV) is updated very frequently.

3.5 Cryptanalysis of HC-128

3.5.1 Approximating the Feedback Functions [Maitra et al. WCC 2009]

In the WCC 2009 workshop, Maitra et al. presented an approximation of the feedback function which we describe here. The analysis is based on the linear approximation of the feedback function. The binary addition is approximated to $\text{XOR}$. We start with a few definitions.

Let $X^{(i)}$ denote the $i$-th bit of an integer $X$, $i \geq 0$ ($i = 0$ stands for the LSB) and $X_1, X_2, \ldots, X_n$ be $n$ independent and uniformly distributed integers. Then define,

$$S_n = \sum_{k=1}^{n} X_k$$

$$L_n = \bigoplus_{k=1}^{n} X_k$$

Also denote $p_n^i = \Pr(S_n^{(i)} = L_n^{(i)})$ and $p_n = \lim_{i \to \infty} p_n^i$. Now by theoretical analysis, the following observations are made:

- $p_n^0 = 1$ implies the LSB is same for both modulo sum and $\text{XOR}$.
- $p_n^1 = \frac{1}{2} + \frac{1}{2^{n+1}}$ implies $p_2 = \frac{1}{2}$.
- $p_n^2 = \frac{1}{3}(1 + \frac{1}{2^{n-1}})$ implies $p_3 = \frac{1}{3}$.
- For even $n$, $p_n = \frac{1}{2}$.
- For odd $n$,
  - $p_n \to \frac{1}{2}$ as $n \to \infty$.
  - For small $n$, $p_n$ may not be close to $\frac{1}{2}$.

The detailed analysis on approximating addition by $\text{XOR}$ can be found in [113].
In 2009, Maitra et al. showed some approximation of the feedback function of HC-128. From section 3.2, notice that the cipher uses two similar functions $g_1, g_2$.

$$g_1(x, y, z) = ((x \gg 10) \oplus (z \gg 23) + (y \gg 8))$$

$$g_2(x, y, z) = ((x \ll 10) \oplus (z \ll 23) + (y \ll 8))$$

While updating, two binary addition operations are used: One inside $g_1$ (or $g_2$) and another outside as mentioned below.

$$P[j] \leftarrow (P[j] + g_1(P[j] \boxdot 3, P[j] \boxdot 10, P[j] \boxdot 511))$$

$$Q[j] \leftarrow (Q[j] + g_2(Q[j] \boxdot 3, Q[j] \boxdot 10, Q[j] \boxdot 511))$$

These two binary addition operations in the update operations of $g_1$ and $g_2$ are source of high non-linearity. But better linear approximation is found using the following result:

Suppose $X_1, X_2, X_3$ are three $n$-bit numbers with $S = (X_1 + X_2 + X_3) \mod 2^n$. Then, for $0 \leq b \leq n - 1$, $\Pr(S_b = X_1^b \oplus X_2^b \oplus X_3^b) = p_b$, where $p_b = \frac{1}{3}(1 + \frac{1}{2^{b-1}})$ that is,

$$p_b = \begin{cases} 
1 & \text{if } b = 0 \\
\frac{1}{2} & \text{if } b = 1 \\
\frac{1}{3} & \text{(approx.) if } 2 \leq b \leq n - 1 
\end{cases} \quad (3.12)$$

Now from section 3.2, we notice that, during the keystream generation part of HC-128, the array $P$ is updated as follows:

$$P'_{updated}[i] = P[i] + ((P[i \boxdot 3] \gg 10) \oplus (P[i \boxdot 511] \gg 23)) + (P[i \boxdot 10] \gg 8) \quad (3.13)$$

Suppose $P'_{updated}[i]$ is the updated value of $P[i]$ when we replace the binary ‘+’ by XOR. Then for $0 \leq b \leq n - 1$, the $b$-th bit of the updated value would be given by

$$(P'_{updated}[i])^b = (P[i]^b \oplus (P[i \boxdot 3])^{10+b} \oplus (P[i \boxdot 511])^{23+b} \oplus (P[i \boxdot 10])^{8+b}) \quad (3.14)$$

Again for $0 \leq b \leq n - 1$, we have,

$$\Pr(P'_{updated}[i]^b = P_{updated}[i]^b) = p_b \quad (3.15)$$

Let, $s_i = P_{updated}[i] \oplus h_1(P[i \boxdot 12])$ and $\psi = P'_{updated}[i] \oplus h_1(P[i \boxdot 12])$. Then for $0 \leq b \leq n-1$, we have,

$$\Pr((s_i)^b = (\psi)^b) = p_b \quad (3.16)$$

But we have the value of $p_b$ as provided in equation 3.12. In this way the technique works.
3.5.2 Extending the designer’s cryptanalysis [Maitra et al. WCC 2009]

In section 3.3, we have described the bias which the designer himself found. Again, in the WCC 2009 workshop, Maitra et al. extended the analysis. The designer found the bias in the least significant bit only. Maitra et al. showed that, the same distinguisher would also work for the other bits too. The equation 3.3 from section 3.3 can be modified as follows:

$$s^b_i \oplus s^b_{i-1024} \oplus s^{10+b}_{i-3} \oplus s^{8+b}_{i-10} \oplus s^{23+b}_{i-1023}$$

$$= (h_1(z_i))^b \oplus (h'_1(z_{i-1024}))^b \oplus (h_1(z_{i-3}))^{10+b} \oplus (h_1(z_{i-10}))^{8+b} \oplus (h'_1(z_{i-1023}))^{23+b}$$

This equation holds with probabilities: $p_0 = 1$ for $b = 0$ (Designer’s case), $p_1 = \frac{1}{2}$ for $b = 1$ and $p_b = \frac{1}{3}$ for $2 \leq b \leq 31$. In short we can write the following equation:

$$\Pr(\Psi^b_i = H^b_i) = p_b$$

where,

$$\Psi^b_i = s^b_i \oplus s^b_{i-1024} \oplus s^{10+b}_{i-3} \oplus s^{8+b}_{i-10} \oplus s^{23+b}_{i-1023}$$

$$H^b_i = (h_1(z_i))^b \oplus (h'_1(z_{i-1024}))^b \oplus (h_1(z_{i-3}))^{10+b} \oplus (h_1(z_{i-10}))^{8+b} \oplus (h'_1(z_{i-1023}))^{23+b}$$

For $2 \leq b \leq 31$, the following equation holds

$$\Pr(\Psi^b_i = H^b_i \oplus 1) = 1 - p_b$$

They proved the following theorem which we state here.

**Theorem 3.5.1.** The following equation holds as a generic case of Wu’s distinguisher

$$\text{Pr}(\Psi^b_i = \Psi^b_j) = \begin{cases} 
\frac{1}{2} + 2^{-81} & \text{if } b = 0; \\
\frac{1}{2} & \text{if } b = 1; \\
\frac{1}{2} + 2^{-81} & \text{if } 2 \leq b \leq 31
\end{cases}$$

The case $b = 0$ corresponds to Wu’s LSB-based distinguisher. Generically, one can mount distinguisher of around the same order for each of the 30 bits corresponding to $b = 2, 3, \ldots, 31$ based on the bias $\frac{1}{2} + 2^{-81}$. The observed bias for higher bits is little less compared to the LSB, so the distinguisher will require around $(9)^2$ time of keystream words, i.e. $81 \cdot 2^{155}$.

3.5.3 State Leakage in Keystream

In 2007, Dunkelman noticed a small observation on HC-128 and posted in the forum at [51]. His observation shows that the keystream words of HC-128 leak information regarding secret states. He reported that,
\[
\Pr(s_j \oplus s_{j+1} = P[j] \oplus P[j+1]) \approx 2^{-16} \quad (3.21)
\]

The probability is much higher than the random association probability \(2^{-31}\) of two 32 bit integers. Later in WCC 2009, again Maitra et al. showed some improvements over this result. They consider a block of 512 many keystream words corresponding to array \(P\). For \(0 \leq u \neq v \leq 511\),

\[
\Pr((s_u \oplus s_v) = (P[u] \oplus P[v])) \approx 2^{-16} \quad (3.22)
\]

This happens due to the bias in the equality of \(h_1(\cdot)\) for two different inputs. The function \(h_1\) uses only 16 bits as input.

Better probability is obtained when a block of 512 keystream words is considered corresponding to array \(P\). For any \(u, v\), \(0 \leq u \neq v \leq 500\), if \(((s_u^{(0)} = s_v^{(0)}) \& (s_u^{(2)} = s_v^{(2)}))\), then,

\[
\Pr((s_{u+12} \oplus s_{v+12}) = (P[u+12] \oplus P[v+12])) \approx 2^{-15} \quad (3.23)
\]

Thus observing the keystream words, one can get better information.

3.5.4 State Recovery from Partial State Exposure

In [109], Maitra, Paul and Raizada from ISI Kolkata, has produced very important result regarding the state recovery from partial exposure. They recovered full state of HC-128 assuming half of the state is known. Here we briefly discuss their strategy.

First the keystream is generated in blocks of 512 words. Now, consider four consecutive blocks \(B_1, B_2, B_3, B_4\) such that,

- Block \(B_1\) : \(P\) unchanged, \(Q\) updated.
- Block \(B_2\) : \(P\) updated to \(P_N\), \(Q\) unchanged.
- Block \(B_3\) : \(P_N\) unchanged, \(Q\) updated to \(Q_N\).
- Block \(B_4\) : Used only for verifying the correctness.

It is assumed that, the half state \(i.e.\) \(P\) is known. The target is to construct the full state \(i.e.\) \((P_N, Q_N)\). The procedure is outlined as follows:

- Phase-1 : Get \(P_N\) from \(P\).
• Phase-2 : Part of $Q$ from $P_N$ is constructed. This phase requires modeling the problem as finding the largest connected component in a random bipartite graph.

• Phase-3 : Construct tail of $Q$ from its parts.

• Phase-4 : Complete $Q_N$ from tail of $Q$.

• Phase-5 : Verification.

The detail can be found in [109]. While analyzing the data and time complexity of the technique, the observations are as follows:

• For the First Phase, we do not need any keystream word.

• For each of the Second, Third, Fourth and Fifth Phases, we need a separate block of 512 keystream words.

• Thus, the required amount of data is $4 \cdot 512 = 2^{11}$ no. of 32-bit keystream words, giving a data complexity $2^{16}$.

• It can be proved that the time complexity to be $2^{42}$. This includes:
  
  – Time to find the largest component.
  
  – Time for computing Phases 3, 4 and 5 for each of $2^{32}$ guesses of the selected node in the largest component.

3.5.5 Differential Fault Analysis

In [83], Kircanski and Youssef presented a differential fault attack against HC-128. They used a standard model: the attacker is able to fault a random word of the inner state tables $P$ and $Q$ but can not control its exact location nor its new faulted value. The attacker is able to reset the cipher arbitrary number of times. To perform the attack, the faults are induced while the cipher is in state 268 instead of state 0. Such a choice reduces the number of required faults to perform the attack. The aim of the attack is to recover the $P$ and $Q$ tables of the cipher in step $i = 1024$.

Now we briefly describe the main idea: Assume that the fault occurred at $Q[f]$, while the cipher is in state $i = 268$. The faulty value $Q'[f]$ is surely not referenced is during steps $i = 0, \ldots, 511$, it follows that $P'[l] = P[l], l = 0, \ldots, 511$. Also according to the update rule, the values $Q[j], Q[j \sqcup 3], Q[j \sqcup 10]$ and $Q[j \sqcup 511]$ are referenced, the the first time in which $Q'[f]$ will be referenced is during the state in which $Q[f - 1]$ is updated i.e. in step $i = 512 + f - 1$. Thus, $Q[f - 1] \neq Q'[f - 1]$. Now, if the fault occurs at $Q[f]$, then $s_j = s'_j$.
holds for $512 \leq j < 512 + f - 1$. The first difference occurs at $i = 512 + j$, $j = f - 1$, after the value $Q[f - 1]$ is affected and then referenced for the output in the same step.

Here are the main steps:

- Repeat the following steps until all of the P, Q words have been faulted at least once.
  - Reset the cipher, iterate it for 268 steps and then induce the fault.
  - Store the resulting faulty keystream words $s'_i$, $i = 268, \ldots, 1535$
- Recover the $h(\cdot)$ input values.
- Recover the inner state, bit by bit, in 32 phases.

The attack requires 7968 fault injections. Complexity of the state recovery is dominated by that of solving a set of 32 systems of linear equations over $\mathbb{Z}_2$ in 1024 variables.

### 3.6 A New Variant of HC-128 to Resist Known Weaknesses

In [109], Maitra, Paul and Raizada from ISI Kolkata proposed a new variant of HC-128 which not only avoids the previously known weaknesses, but also weakness discovered in this work. It has been observed that, all known weaknesses exploit the fact that $h_1(\cdot)$ as well as $h_2(\cdot)$ makes use of only 16 bits from the 32-bit input. To get rid of this, they replace $h_1$ and $h_2$ as follows:

\[
\begin{align*}
  h_{N1}(x) &= (P[x^{(0)}] + P[256 + x^{(2)}]) \oplus x. \\
  h_{N2}(x) &= (Q[x^{(0)}] + Q[256 + x^{(2)}]) \oplus x.
\end{align*}
\] (3.24)

This technique uses all bits in $h_1$ and $h_2$ and thus remove the weakness. The second thing they does is they makes updates of $P$ and $Q$ independent by including a randomly chosen word from the $Q$ array in the update of $P$ array elements and a randomly chosen word from the $P$ array while updating the $Q$ array elements as follows:

\[
\begin{align*}
  g_{N1}(x, y, z) &= ((x \gg 10) \oplus (z \gg 23)) + Q[(y \gg 7) \land 1FF]. \\
  g_{N2}(x, y, z) &= ((x \ll 10) \oplus (z \ll 23)) + P[(y \gg 7) \land 1FF].
\end{align*}
\] (3.25)

Now, due to nesting of $P$ and $Q$ in each other’s update, firstly, the internal state would be preserved even if half the internal state elements are revealed, since knowledge of one of
3.7 Conclusion

$P$ and $Q$ in one block would not reveal the same array in any previous or subsequent blocks. Secondly, for the fault analysis, if a fault occurs at $Q[f]$ in the block in which $P$ is updated, then $Q[f]$ is not referenced until step $f-1$ of the next block (in which $Q$ would be updated). This assumption does not hold in the new design. Also, since the new $h(\cdot)$ functions use all the 32-bit words of their input arguments, the existing distinguisher cannot be mounted.

Here we give a performance of the new design compare to HC-128 and HC-256 in a machine with Intel(R) Pentium(R) D CPU, 2.8 GHz Processor Clock, 2048 KB Cache Size, 1 GB DDR RAM on Ubuntu 7.04 (Linux 2.6.20-17-generic) OS using the gcc-3.4 prescott O3-ofp compiler.

<table>
<thead>
<tr>
<th>Stream Encryption (cycles/byte)</th>
<th>HC-128</th>
<th>New Proposal</th>
<th>HC-256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.13</td>
<td>4.29</td>
<td>4.88</td>
</tr>
</tbody>
</table>

3.7 Conclusion

In conclusion it can be said that, HC-128 has several advantages viz. very simple and understandable design. But a few important weaknesses has been found which can pose much greater threat in future. An alternative design has also been discussed where the weaknesses are taken care of with a small compromise in performance. In future some more new variations may come up with stronger security and better performance.

3.8 A Simple C implementation of HC-128

We present a understandable C-implementation here. For more optimized version reader may look into the eSTREAM website here [58].

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define ROUNDS 16

unsigned int P[512], Q[512], K[8], IV[8], W[1280];

/* Functions and Operations Used */
```
#define sl3(x, n) (((x) << n) ^ ((x) >> (32-n)))
#define sr3(x, n) (((x) >> n) ^ ((x) << (32-n)))
#define f1(x) (sr3((x), 7)) ^ (sr3((x), 18)) ^ ((x) >> 3)
#define f2(x) (sr3((x), 17)) ^ (sr3((x), 19)) ^ ((x) >> 10)
#define g1(x, y, z) (((sr3((x), 10)) ^ (sr3((z), 23))) + (sr3((y), 8)))
#define g2(x, y, z) (((sl3((x), 10)) ^ (sl3((z), 23))) + (sl3((y), 8)))

unsigned int h1(unsigned int x){
    unsigned int y;
    unsigned char a, c;
    a = (unsigned char) (x);
    c = (unsigned char) ((x) >> 16);
    y = (Q[a])+(Q[256+c]);
    return y;
}

unsigned int h2(unsigned int x){
    unsigned int y;
    unsigned char a, c;
    a = (unsigned char) (x);
    c = (unsigned char) ((x) >> 16);
    y = (P[a])+(P[256+c]);
    return y;
}

/* Minus modulo 512 */

int mm(int i, int j){
    int temp;

    if (i >= 512)
        i = i%512;
    if (j >= 512)
        j = j%512;

    if (i >= j)
        temp = i-j;
    else
        temp = 512 + i-j;

    return temp;
}
/* The key scheduling */

void ksa(){
    int i;

    for (i = 0; i < 8; i++)
        W[i] = K[i];
    for (i = 0; i < 8; i++)
        W[i+8] = IV[i];
    for (i = 16; i < 1280; i++)
        W[i] = (f2(W[i-2])) + W[i-7] + (f1(W[i-15])) + W[i-16] + i;

    for (i = 0; i < 512; i++){
        P[i] = W[i+256];
        Q[i] = W[i+768];
    }

    for (i = 0; i < 512; i++)
        P[i] = (P[i] + g1(P[mm(i, 3)], P[mm(i, 10)], P[mm(i, 511)]))^(h1(P[mm(i, 12)]));

    for (i = 0; i < 512; i++)
        Q[i] = (Q[i] + g2(Q[mm(i, 3)], Q[mm(i, 10)], Q[mm(i, 511)]))^(h2(Q[mm(i, 12)]));
}

main(){
    unsigned int i, j, s;

    srand48(time(NULL));

    /* Generate Key and IV */
    /* Three sets of key-iv pairs given in the HC-128 document for checking */

    for (i = 0; i < 4; i++){
        K[i] = (unsigned int) (4294967296.0*drand48());
        IV[i] = (unsigned int) (4294967296.0*drand48());
    }

\[
\begin{align*}
\text{K[0]} &= \text{K[1]} = \text{K[2]} = \text{K[3]} = 0; \\
\text{IV[0]} &= 1; \text{IV[1]} = \text{IV[2]} = \text{IV[3]} = 0;
\end{align*}
\]

\[
\begin{align*}
\text{K[0]} &= \text{0x55}; \text{K[1]} = \text{K[2]} = \text{K[3]} = 0; \\
\text{IV[0]} &= \text{IV[1]} = \text{IV[2]} = \text{IV[3]} = 0;
\end{align*}
\]

/* Expand Key and IV */

for (i = 0; i < 4; i++){
    K[i+4] = K[i];
    IV[i+4] = IV[i];
}

/* Call the Key Scheduling Subroutine */

ksa();

i = 0;

while(1){
    /* As may keystream words you need */
    if (i >= ROUNDS) break;
    j = i%512;
    if (i%1024 < 512)
        P[j] = (P[j] + g1(P[mm(j,3)], P[mm(j, 10)], P[mm(j, 511)]));
        s = (h1(P[mm(j, 12)])) ^ P[j];
    else
        Q[j] = (Q[j] + g2(Q[mm(j,3)], Q[mm(j, 10)], Q[mm(j, 511)]));
        s = (h2(Q[mm(j, 12)])) ^ Q[j];
    printf("%8x ", s);
    if (i > 0 && i%4 == 3) printf("\n");
    i++;
}
Chapter 4

SOSEMANUK

4.1 Introduction

SOSEMANUK is a software-efficient (profile 1), synchronous stream cipher proposed by Berbain et al. in [15]. The cipher has a variable key length, ranging from 128 to 256 bits, and takes an initial value (IV) of 128 bits in length. However, for any key length the cipher is only claimed to offer 128-bit security. SOSEMANUK uses similar design principles to the stream cipher SNOW 2.0 (details in [53]) and block cipher SERPENT (details in [22]). The design of SOSEMANUK aims to fix some potential structural weaknesses of SNOW 2.0, while providing better performance by decreasing the size of the internal state.

As with the stream cipher SNOW 2.0, SOSEMANUK has two main components: a linear feedback register (LFSR) and a finite state machine (FSM). The LFSR operates on 32-bit words and has length 10; at every clock a new 32-bit word is computed. The FSM has two 32-bit memory registers: at each step the FSM takes as input words from the LFSR, updates the memory registers and produces a 32-bit output. On every four consecutive output words from the FSM an output transformation, based on the block cipher SERPENT, is applied. The resulting four 32-bit output words are \texttt{XOR}-ed with four outputs from the LFSR to produce four 32-bit words of keystream.

Regarding its performance in software, SOSEMANUK can encrypt long data streams at 5.60 cycles/byte on Pentium M and at 4.07 cycles/byte on AMD Athlon 64 X2. For more information about eSTREAM ciphers performance in software, refer to the eSTREAM testing framework page at [55]. For a more extensive comparison with many other stream ciphers on several different architectures, refer to the eBACS stream cipher software timings page maintained by D. Bernstein at [21].
4.2 Specifications of SOSEMANUK

Since SOSEMANUK is built on the block cipher SERPENT, we start the specification with description of SERPENT and its derivative.

4.2.1 SERPENT and derivatives

SERPENT (details in [22]) is a block cipher proposed as an AES candidate. SERPENT operates over blocks of 128 bits which are split into four 32-bit words, which are then combined in so-called “bitslice” mode. SERPENT can thus be defined as working over quartets of 32-bit words. We number SERPENT input and output quartets from 0 to 3, and write them in the order: \((Y_3, Y_2, Y_1, Y_0)\). \(Y_0\) is the least significant word, and contains the least significant bits of the 32 4-bit inputs to the SERPENT S-boxes. When SERPENT output is written into 16 bytes, the \(Y_i\) values are written following the little-endian convention (least significant byte first), and \(Y_0\) is output first, then \(Y_1\), and so on.

From SERPENT we define two primitives called \(\text{Serpent1}\) and \(\text{Serpent24}\).

\(\text{Serpent1}\)

A SERPENT rounds consist of, in that order:

- a subkey addition, by bitwise exclusive or;
- S-box application (which is expressed as a set of bitwise combinations between the four running 32-bit words, in bitslice mode);
- a linear bijective transformation (which amounts to a few \(\text{XOR}\)s, shifts and rotations in bitslice mode).

\(\text{Serpent1}\) is one round of SERPENT, without the key addition and the linear transformation. SERPENT uses eight distinct S-boxes (see [15] for details), numbered from \(S_0\) to \(S_7\) on 4-bit words. We define \(\text{Serpent1}\) as the application of \(S_2\), in bitslice mode. This is the third S-box layer of SERPENT. \(\text{Serpent1}\) takes four 32-bit words as input, and provides four 32-bit words as output.

\(\text{Serpent24}\)

\(\text{Serpent24}\) is SERPENT reduced to 24 rounds, instead of the 32 rounds of the full version of SERPENT. \(\text{Serpent24}\) is equal to the first 24 rounds of SERPENT, where the last round (the
24th) is a complete one and includes a complete round with the linear transformation and an XOR with the 25th subkey. In other words, the 24th round of Serpent24 is thus equivalent to the thirty-second round of SERPENT, except that it contains the linear transformation and that the 24th and 25th subkeys are used (32nd and 33rd subkeys in SERPENT). Thus, the last round equation on Page 224 in [22] is as follows:

\[ R_{23}(X) = L(\hat{S}_{23}(X \oplus \hat{K}_{23}) \oplus \hat{K}_{24}) \] (4.1)

Serpent24 uses only 25 128-bit subkeys, which are the first 25 subkeys produced by the SERPENT key schedule. In SOSEMANUK, Serpent24 is used for the initialization step, only in encryption mode. Decryption is not used.

4.2.2 The LFSR

Another important ingredient of SOSEMANUK is the Linear Feedback Shift Register (abbrv. as LFSR). Now we describe the LFSR involved here.

Underlying finite field

Most of the stream cipher internal state is held in a LFSR containing 10 elements of \( \mathbb{F}_{2^{32}} \), the field with \( 2^{32} \) elements. The elements of \( \mathbb{F}_{2^{32}} \) are represented exactly as in SNOW 2.0. We recall this representation here. Let \( \mathbb{F}_2 \) denote the finite field with 2 elements. Let \( \beta \) be a root of the primitive polynomial \( Q(X) = X^8 + X^7 + X^5 + X^3 + 1 \) on \( \mathbb{F}_2[X] \). We define the field \( \mathbb{F}_{2^8} \) as the quotient \( \mathbb{F}_2[X]/Q(X) \). Each element in \( \mathbb{F}_{2^8} \) is represented using the basis \( (\beta^7, \beta^6, \ldots, \beta, 1) \). Since the chosen polynomial is primitive, then \( \beta \) is a multiplicative generator of all invertible elements of \( \mathbb{F}_{2^8} \): every non-zero element in \( \mathbb{F}_{2^8} \) is equal to \( \beta^k \) for some integer \( k \) (0 \( \leq k \leq 254 \)). Any element in \( \mathbb{F}_{2^8} \) is identified with an 8-bit integer by the following bijection:

\[ \phi : \mathbb{F}_{2^8} \to \{0, 1, \ldots, 255\} \] (4.2)

\[ x = \sum_{i=0}^{7} x_i \beta^i \to \sum_{i=0}^{7} x_i 2^i \]

where each \( x_i \) is either 0 or 1. For instance, \( \beta^{23} \) is represented by the integer \( \phi(\beta^{23}) = 0x81 \) (in hexadecimal). Therefore, the addition of two elements in \( \mathbb{F}_{2^8} \) corresponds to a bitwise XOR between the corresponding integer representations. The multiplication by \( \beta \) is a left shift by one bit of the integer representation, followed by an XOR with a fixed mask if the most significant bit dropped by the shift equals 1.
Let $\alpha$ be a root of polynomial $P(x) = X^4 + \beta^{23}X^3 + \beta^{245}X^2 + \beta^{48}X + \beta^{239}$ on $F_{2^8}[X]$. The field $F_{2^{32}}$ is then defined as the quotient $F_{2^8}[X]/P(X)$, i.e., its elements are represented with the basis $(\alpha^3, \alpha^2, \alpha, 1)$. Any element in $F_{2^{32}}$ is identified with a 32-bit integer by the following bijection:

$$
\psi : F_{2^{32}} \rightarrow \{0, 1, \ldots, 2^{32} - 1\} \quad (4.3)
$$

$$
y = \sum_{i=0}^{3} y_i \alpha^i \rightarrow \sum_{i=0}^{3} \phi(y_i)2^{8i}
$$

Thus, the addition of two elements in $F_{2^{32}}$ corresponds to a bitwise XOR between their integer representations. This operation will hereafter be denoted by $\oplus$. SOSEMANUK also uses multiplications and divisions of elements in $F_{2^{32}}$ by $\alpha$. Multiplication of $z \in F_{2^{32}}$ by $\alpha$ corresponds to a left shift by 8 bits of $\psi(z)$, followed by an XOR with a 32-bit mask which depends only on the most significant byte of $\psi(z)$. Division of $z \in F_{2^{32}}$ by $\alpha$ is a right shift by 8 bits of $\psi(z)$, followed by an XOR with a 32-bit mask which depends only on the least significant byte of $\psi(z)$.

![Figure 4.1: The LFSR](image)

**Definition of the LFSR**

The LFSR operates over elements of $F_{2^{32}}$. The initial state, at $t = 0$, entails the ten 32-bit values $s_1$ to $s_0$. At each step, a new value is computed, with the following recurrence:

$$
s_{t+10} = s_{t+9} \oplus \alpha^{-1}s_{t+3} \oplus \alpha s_t, \forall t \geq 1 \quad (4.4)
$$

and the register is shifted (see Figure 4.1 for an illustration of the LFSR).

The LFSR is associated with the following feedback polynomial:

$$
\pi(X) = \alpha X^{10} + \alpha^{-1}X^7 + X + 1 \in F_{2^{32}}[X] \quad (4.5)
$$

Since the LFSR is non-singular and since $\pi$ is a primitive polynomial, the sequence of 32-bit words $(s_t)_{t \geq 1}$ is periodic and has maximal period $(2^{320} - 1)$.
4.2. Specifications of SOSEMANUK

The Finite State Machine

The Finite State Machine (FSM) is a component with 64 bits of memory, corresponding to two 32-bit registers $R_1$ and $R_2$. At each step, the FSM takes as inputs some words from the LFSR state; it updates the memory bits and produces a 32-bit output. The FSM operates on the LFSR state at time $t \geq 1$ as follows:

$$FSM_t : (R_{1_{t-1}}, R_{2_{t-1}}, s_{t+1}, s_{t+8}, s_{t+9}) \rightarrow (R_{1_t}, R_{2_t}, f_t) \quad (4.6)$$

where,

$$R_{1_t} = (R_{2_{t-2}} + \text{mux}(\text{lsb}(R_{1_{t-1}}), s_{t+1}, s_{t+1} \oplus s_{t+8})) \mod 2^{32} \quad (4.7)$$

$$R_{2_t} = \text{Trans}(R_{1_{t-1}})$$

$$f_t = (s_{t+9} + R_{1_t} \mod 2^{32}) \oplus R_{2_t}$$

where $\text{lsb}(x)$ is the least significant bit of $x$, $\text{mux}(c,x,y)$ is equal to $x$ if $c = 0$, or to $y$ if $c = 1$. The internal transition function $\text{Trans}$ on $\mathbb{F}_{2^{32}}$ is defined by

$$\text{Trans}(z) = (M \times z \mod 2^{32}) \ll 7 \quad (4.8)$$

where $M$ is the constant value $0x54655307$ (the hexadecimal expression of the first ten decimals of $\pi$) and $\ll$ denotes bitwise rotation of a 32-bit value (by 7 bits here).

4.2.3 Output transformation

The outputs of the FSM are grouped by four, and $\text{Serpent1}$ is applied to each group; the result is then combined by $\text{XOR}$ with the corresponding dropped values from the LFSR, to produce the output values $z_t$:

$$(z_{t+3}, z_{t+2}, z_{t+1}, z_t) = \text{Serpent1}(f_{t+3}, f_{t+2}, f_{t+1}, f_t) \oplus (s_{t+3}, s_{t+2}, s_{t+1}, s_t) \quad (4.9)$$

Four consecutive rounds of Sosemanuk are depicted in Figure 4.2.

4.2.4 SOSEMANUK workflow

The SOSEMANUK cipher combines the FSM and the LFSR to produce the output values $z_t$. Time $t = 0$ designates the internal state after initialization; the first output value is $z_1$. Figure 4.3 gives a graphical overview of SOSEMANUK.

At time $t \geq 1$, we perform the following operations:
Figure 4.2: The output transformation on four consecutive rounds of SOSEMANUK.
4.2. Specifications of SOSEMANUK

- The FSM is updated: \( R_{1t}, R_{2t} \) and the intermediate value \( f_t \) are computed from \( R_{1t-1}, R_{2t-1}, s_{t+1}, s_{t+8} \) and \( s_{t+9} \).

- The LFSR is updated: \( s_{t+10} \) is computed, from \( s_t, s_{t+3} \) and \( s_{t+9} \). The value \( s_t \) is sent to an internal buffer, and the LFSR is shifted.

Figure 4.3: An overview of SOSEMANUK

Once every four steps, four output values \( z_t, z_{t+1}, z_{t+2} \) and \( z_{t+3} \) are produced from the accumulated values \( f_t, f_{t+1}, f_{t+2}, f_{t+3} \) and \( s_t, s_{t+1}, s_{t+2}, s_{t+3} \). Thus, SOSEMANUK produces 32-bit values. We recommend encoding them into groups of four bytes using the little-endian convention, because it is faster on the most widely used high-end software platform (x86-compatible PC), and because SERPENT uses that convention.

Therefore, the first four iterations of SOSEMANUK are as follows.

- The LFSR initial state contains values \( s_1 \) to \( s_{10} \); no value \( s_0 \) is defined. The FSM initial state contains \( R_{10} \) and \( R_{20} \).

- During the first step \( R_{11}, R_{21} \) and \( f_1 \) are computed from \( R_{10}, R_{20}, s_2, s_9 \) and \( s_{10} \).

- The first step produces the buffered intermediate values \( s_1 \) and \( f_1 \).
- During the first step, the feedback word $s_{11}$ is computed from $s_{10}, s_4$ and $s_1$, and the internal state of the LFSR is updated, leading to a new state composed of $s_2$ to $s_{11}$.

- The first four output values are $z_1, z_2, z_3$ and $z_4$, and are computed using one application of Serpent1 over $(f_4, f_3, f_2, f_1)$, whose output is combined by XORs with $(s_4, s_3, s_2, s_1)$.

### 4.2.5 Initialization of SOSEMANUK

The SOSEMANUK initialization process is split into two steps:

- The key schedule, which processes the secret key but does not depend on the IV.
- The IV injection, which uses the output of the key schedule and the IV. This initializes the stream cipher internal state.

As initialization, SOSEMANUK generates the initial value of internal state, using the key schedule of SERPENT and Serpent24. Figure 4.4 illustrates the initialization of SOSEMANUK.

SOSEMANUK takes secret key KEY as an input to key schedule of SERPENT, to generate twenty-five 128-bit subkeys. Though the secret key of SOSEMANUK is variable, ranging from 128-bit up to 256-bit, SERPENT’s secret key is also variable, ranging from 1-bit from 256-bit. Thus, key scheduler can be operated in accordance with the secret key length. After the subkey generation, initial vector IV is taken as an input to Serpent24. Then, intermediate data of
rounds 12 and 18 of Serpent24, and output data of round 24 of Serpent24 are used as initial values of internal state. Providing that $Y^{12}, Y^{18},$ and $Y^{24}$ denote the outputs of rounds 12, 18, and 24, respectively, these three data are substituted in the equations below, as the initial values of their respective registers. Here, LFSR register and FSM register at the completion of initialization are represented by $(s_{10}, s_9, \ldots, s_1)$ and $(R_{10}, R_{20})$, respectively.

\[
\begin{align*}
  s_7 || s_8 || s_9 || s_{10} &= Y^{12} \\
  R_{10} || s_5 || R_{20} || s_6 &= Y^{18} \\
  s_1 || s_2 || s_3 || s_4 &= Y^{24}
\end{align*}
\]

### 4.3 Design Rationale of SOSEMANUK

#### 4.3.1 Key initialization and IV injection

**Underlying Principle**

A first property of the initialization process is that it is split into two distinct steps: the key schedule which does not depend on the IV, and the IV injection which generates the initial state of the generator from the IV and from the output of the key schedule. Then, the IV setup for a fixed key is less expensive than a complete key setup, improving the common design since changing the IV is more frequent than changing the secret key.

A second characteristic of SOSEMANUK is that the IV setup is derived from the application of a block cipher over the IV. If we consider the function $F_K$ which maps a $n$-bit IV to the first $n$ bits of output stream generated from the key $K$ and the IV, then $F_K$ must be computationally indistinguishable from a random function over $\mathbb{F}_2^n$. Hence, the computation of $F_K$ cannot “morally” be faster than the best known PRF over $n$-bit blocks. It so happens that the fastest known PRF use the same implementation techniques that the fastest known Pseudo-Random Permutations (which are block ciphers), and amount to the equivalent performance.

Since SOSEMANUK stream generation is very fast, the generation of $n$ stream bits takes little time compared to a computation of a robust PRP over a block of $n$ bits. Following this path of reasoning, the designers decided to use a block cipher as the foundation of the IV setup for SOSEMANUK: the IV setup itself cannot be much faster than the application of a block cipher, and the security requirements for that step are much similar to what is expected from a block cipher.
Choice of Block Cipher

The block cipher used in the IV setup is derived from SERPENT for the following reasons:

- SERPENT has been thoroughly analyzed during the AES selection process and its security is well-understood.
- SERPENT needs no static data tables, and hence adds little or no data cache pressure.
- The SERPENT round function is optimized for operation over data represented as 32-bit words, which is exactly how data is managed within SOSEMANUK. Using SERPENT implies no tedious byte extraction from 32-bit words, or recombinations into such words.
- The best linear bias and differential bias for a 6-round version of SERPENT are $2^{-28}$ and $2^{-58}$ respectively. (details in [22]). Thus, 12 rounds should provide appropriate security.
- Two consecutive outputs of data are spaced with six inner rounds in order to prevent the existence of relations between the bits of the initial state and the secret key bits which could be used in an attack.

4.3.2 The LFSR

LFSR Length

The following things are important to note about the length of the LFSR used in the design of SOSEMANUK:

- The LFSR length should ideally be small. Because, in that case it would be easier to map to registers.
- For efficient LFSR implementation, the LFSR must not be physically shifted; moving data around contributes nothing to actual security, and takes time.
  - If $n$ is the LFSR length, then $kn$ steps (for some integer $k$) must be “unrolled”, so that at each step only one LFSR cell is modified.
  - Since Serpent1 operates over four successive output values, $kn$ corresponds to $\text{lcm}(4, n)$ and it should be kept as small as possible, since a higher code size increases code cache pressure.
- The length 10 is chosen (8 is also suitable but prone to guess-and-determine attack). So, the outcome is 324 bit ($10 \times 32 + 2 \times 32$) internal state. Also, it is important to notice that only 20 rounds of unrolling is kept for maximum efficiency.
LFSR Feedback

The design criteria for the feedback polynomial are similar to those used in SNOW 2.0. Since the feedback polynomial must be as sparse as possible, the designers chose as in SNOW 2.0 a primitive polynomial of the form

\[ \pi(X) = c_0 X^{10} + c_a X^{n-a} + c_b X^{n-b} + 1 \]  

(4.11)

where \( 0 < a < b < 10 \). Some important points are:

- The coefficients \( c_0, c_a \) and \( c_b \) preferably lie in \( \{1, \alpha, \alpha - 1\} \) which are the elements corresponding to an efficient multiplication in \( \mathbb{F}_{2^{32}} \).

- \( \{c_0, c_a, c_b\} \) must contain at least two distinct non-binary elements; otherwise, a multiple of \( \pi \) with binary coefficients can be easily constructed (details in [52, 69]), providing an equation which holds for each single bit position.

- \( a \) and \( b \) must be coprime with LFSR length. Otherwise, for instance if \( d = \gcd(a, 10) > 1 \), the corresponding recurrence relation

\[ s_{t+10} = c_b s_{t+b} + c_a s_{t+a} + c_0 s_t \]  

(4.12)

involves three terms of a decimated sequence \( (s_{dt+i})_{t>0} \) (for some integer \( i \)), which can be generated by an LFSR of length \( n/d \).

- The values \( a = 3, b = 9, c_0 = \alpha, c_3 = \alpha^{-1} \) AND \( c_9 = 1 \) satisfies all the conditions.

4.3.3 The FSM

The Trans function

The Trans function is chosen according to the following implementation criteria:

- The multiplication is done in 32 bit which provides excellent “data mixing”.

- Only two basic operations are used \( \text{viz.} \) \( \text{MULT} \) and \( \text{SHIFT} \)– both are easy logical steps.

- A great advantage is no static table (S-box) is not required which saves lot of memory.

- The operations involved in the \( \text{Trans} \) functions are incompatible with the other operations used in the FSM (addition over \( \mathbb{Z}_{2^{32}}, \text{XOR} \)) because actually, mixing operations on the ring and on the vector space disables associativity and commutativity laws.
The mux operation

- The mux operation aims at increasing the complexity of fast correlation and algebraic attacks, since it decimates the FSM input sequence in an irregular fashion.

- Moreover, this operation can be implemented efficiently with either control bit extension and bitwise operations, or an architecture specific “conditional move” opcode.

It is fitting that both LFSR elements \( s_{t+c} \) and \( s_{t+d} \) (with \( c \leq d \)) in the mux operation are not involved in the recurrence relation. Otherwise the complexity of guess-and-determine attacks might be reduced. The distance \( (d - c) \) between those elements must be co-prime with the LFSR length since they must not be expressed as a decimated sequence with a lower linear complexity. Here, \( d - c = 7 \) is chosen. Finally, it must be impossible for the inputs of the mux operation at two different steps correspond to the same element in the LFSR sequence. For this reason, the mux operation outputs either \( s_{t+c} \) or \( s_{t+c} \oplus s_{t+d} \). If \( s_{t+c} \oplus s_{t+d} \) is the input of the FSM at time \( t \), the possible inputs at time \( (t + d - c) \) are \( s_{t+d} \) and \( s_{t+d} \oplus s_{t+2d-c} \), which do not match any previous input. It is worth noticing that this property does not hold anymore if the mux outputs either \( s_{t+c} \) or \( s_{t+d} \).

4.3.4 The output transformation

The output transformation derived from Serpent1 aims at mixing four successive outputs of the FSM in a nonlinear way. As a consequence, any 32-bit keystream word produced by SOSEMANUK depends on four consecutive intermediate values \( f_t \). As a result, recovering any single output of the FSM, \( f_t \), in a guess-and-determine attack requires the knowledge of at least four consecutive words from the LFSR sequence, \( s_t, s_{t+1}, s_{t+2}, s_{t+3} \).

The following properties have also been taken into account in the choice of output transformation.

- Both nonlinear mixing operations involved in SOSEMANUK (the Trans operation and the Serpent1 used in bitslice mode) do not provide any correlation probability or linear property on the least significant bits that could be used to mount an attack.

- From an algebraic point of view, those operations are combined to produce nonlinear equations.

- No linear relation can be directly exploited on the least significant bit of the values \((f_t, f_{t+1}, f_{t+2}, f_{t+3})\), only quadratic equations with more variables than the number of possible equations.
• The linear relation between \( s_t \) and \( \text{Serpent}1 (f_t, f_{t+1}, f_{t+2}, f_{t+3}) \) prevents SOSEMANUK from SQUARE-like attacks.

Finally, the fastest SERPENT S-box (\( S_2 \)) has been chosen in \( \text{Serpent}1 \) from an efficiency point of view. But, \( S_2 \) also guarantees that there is no differential-linear relation on the least significant bit (the “most linear” one in the output of the FSM).

### 4.4 Security Properties of SOSEMANUK

The designers claimed 128-bit security of SOSEMANUK. The details can be found in [15].

#### 4.4.1 Time-memory-data tradeoff attacks

Due to the choice of the length of the LFSR (more than twice the key length), the time-memory-data tradeoff attacks described in [12, 25, 64] are impracticable. Moreover, since these TMDTO attacks aim at recovering the internal state of the cipher, recovering the secret key requires the additional cost of an attack against \( \text{Serpent}24 \). The best time-memory data tradeoff attack is the Hellman’s one (details in [71]) which aims at recovering a pair (\( K, IV \)). For a 128-bit secret key and a 128-bit IV, its time complexity is equal to \( 2^{128} \) cipher operations.

#### 4.4.2 Guess and determine attacks

The main weaknesses of SNOW 1.0 are related to this type of attacks (two at least have been exhibited [69]). They essentially exploit a particular weakness in the linear recurrence equation. This does not hold anymore for the new polynomial choice in SNOW 2.0 and for the polynomial used in SOSEMANUK which involve non-binary multiplications by two different constants. The attack shown in [69] also exploited a “trick” coming from the dependence between the values \( R_1_{t-1} \) and \( R_1_t \). This trick is avoided in SNOW 2.0 (because there is no direct link between those two register values anymore) and in SOSEMANUK.

The best guess and determine attack we have found on SOSEMANUK is as follows.

• Guess at time \( t, s_t, s_{t+1}, s_{t+2}, s_{t+3}, R_{1_{t-1}} \) and \( R_{2_{t-1}} \) (6 words).

• Compute the corresponding outputs of the FSM \( (f_t, f_{t+1}, f_{t+2}, f_{t+3}) \).

• Compute \( R_{2_t} = \text{Trans}(R_{1_{t-1}}) \) and \( R_{1_t} \) from Equation 4.7 if \( \text{lsb}(R_{1_{t-1}}) = 1 \) (this can be done only with probability 1/2).

• From \( f_t = (s_{t+9} + R_{1_t} \mod 2^{32}) \oplus R_{2_t} \), compute \( s_{t+9} \).
• Compute $R_{1,t+1}$ from the knowledge of both $s_{t+2}$ and $s_{t+9}$; compute $R_{2,t+1}$. Compute $s_{t+10}$ from $f_{t+1}$, $R_{1,t+1}$ and $R_{2,t+1}$.

• Compute $R_{1,t+2}$ from $s_{t+3}$ and $s_{t+10}$; compute $R_{2,t+2}$. Compute $s_{t+11}$ from $f_{t+2}$, $R_{1,t+2}$ and $R_{2,t+2}$. Now, $s_{t+4}$ can be recovered due to the feedback relation at time $t+1$:

$$\alpha^{-1} s_{t+4} = s_{t+11} \oplus s_{t+10} \oplus \alpha s_{t+1}$$  \hspace{1cm} (4.13)

• Compute $R_{1,t+3}$ from $s_{t+4}$ and $s_{t+11}$ ; compute $R_{2,t+2}$. Compute $s_{t+12}$ from $f_{t+3}$, $R_{1,t+3}$ and $R_{2,t+3}$. Compute $s_{t+5}$ by the feedback relation at time $t+2$:

$$\alpha^{-1} s_{t+5} = s_{t+12} \oplus s_{t+11} \oplus \alpha s_{t+2}$$  \hspace{1cm} (4.14)

At this point, the LFSR words $s_t, s_{t+1}, s_{t+2}, s_{t+3}, s_{t+4}, s_{t+5}, s_{t+9}$ are known. Three elements $(s_{t+6}, s_{t+7}, s_{t+8})$ remain unknown. To complete the full 10 words state of the LFSR, we need to guess 2 more words, $s_{t+6}$ and $s_{t+7}$ since each $f_{t+i}, 4 \leq i \leq 7$, depends on all 4 words $s_{t+4}, s_{t+5}, s_{t+6}$ and $s_{t+7}$. Therefore, this attack requires the guess of 8 32-bit words, leading to a complexity of $2^{56}$.

The designers claim that there is no better guess-and-determine attack against SOSEMANUK. The main reason is that Serpent1 used in bitslice mode requires the knowledge of at least four consecutive words from the LFSR sequence when recovering any single output of the FSM. Note that the previous attack on an LFSR of length eight enables the recovery of the entire internal state of the cipher from the guess of six words only.

4.4.3 Correlation attacks

In order to find a relevant correlation in SOSEMANUK, the following questions can be addressed:

• does there exist a linear relation at bit level between some input and output bits?

• does there exist a particular relation between some input bit vector and some output bit vector?

In the first case, two linear relations could be exhibited at the bit level. In the first, the least significant bit of $s_{t+9}$ was “conserved”, since the modular addition over $Z_{2^{32}}$ is a linear operation on the least significant bit. The second linear relation induced by the FSM concerns the least significant bit of $s_{t+1}$ or of $s_{t+1} \oplus s_{t+8}$ (used to compute $R_{1,t}$ ) or the seventh bit of $R_{2,t}$ computed from $s_t$ or of $s_t \oplus s_{t+7}$. We here use that $R_{2,t} = \text{Trans}(R_{1,t-1})$ and $R_{1,t-1} = R_{2,t-2} + (\text{stor}(s_t \oplus s_{t+7})) \mod 2^{32}$. 
4.4. Security Properties of SOSEMANUK

No linear relation holds after applying Serpent1 and there are too many unknown bits to exploit a relation on the outputs words due to the bitslice design. Moreover, a fast correlation attack seems to be impracticable because the mux operation prevents certainty in the dependence between the LFSR states and the observed keystream.

4.4.4 Distinguishing attacks

A distinguishing attack by D. Coppersmith, S. Halevi and C. Jutla (see [41] for details) against the first version of SNOW used a particular weakness of the feedback polynomial built on a single multiplication by $\alpha$. This property does not hold for the choice of the new polynomial in SNOW 2.0 and for the polynomial used in SOSEMANUK where multiplication by $\alpha - 1$ is also included.

In [124], D. Watanabe, A. Biryukov and C. De Cannière have mounted a new distinguishing attack on SNOW 2.0 with a complexity about $2^{225}$ operations using multiple linear masking method. They construct 3 different masks $\Gamma_i = \Gamma_i \cdot \alpha$ and $\Gamma_i = \Gamma_i \cdot (\Gamma_i \cdot (\gamma \oplus t) = \Gamma_i \cdot (\gamma \oplus t)$ for $i = 1, 2$ and 3, where $S'$ is the transition function of the FSM in SNOW 2.0. In the case of SNOW 2.0, the hardest hypothesis to satisfy is the first one defined on $y = S'(x)$. In case of SOSEMANUK, we need $Pr(\Gamma_i \cdot Trans(x) = \Gamma_i \cdot x)_{i=1,2,3}$ to be high. But, we also need that $\forall i = 1, 2, 3$, the relation:

$$ (\Gamma_i', \Gamma_i', \Gamma_i', \Gamma_i') \cdot (x_1, x_2, x_3, x_4) = Serpent1(\Gamma_i, \Gamma_i, \Gamma_i, \Gamma_i) \cdot (x_1, x_2, x_3, x_4)$$  \hspace{1cm} (4.15)

for some $\Gamma_i \in \mathbb{F}_2^{12}$, holds with a high probability.

Due to the bitslice design chosen for Serpent1, it seems very difficult to find such a mask. Therefore, the attack described in [124] could not be applied directly on SOSEMANUK.

4.4.5 Algebraic Attacks

Let us consider, as in [23], the initial state of the LFSR at bit level:

$$(s_{10}, \ldots, s_1) = (s_{10}, s_{10}, s_{0}, s_{0}, s_{31}, s_{31}, s_{0})$$  \hspace{1cm} (4.16)

Then, the outputs of SOSEMANUK at time $t \geq 1$ could be written:

$$F^t(s_{31}^{10}, s_6^1) = (z_t, z_{t+1}, z_{t+2}, z_{t+3})$$  \hspace{1cm} (4.17)
where $F$ is a vectorial Boolean function from $\mathbb{F}_2^{320}$ into $\mathbb{F}_2^{128}$ that could be seen as 128 Boolean functions $F_j, \forall j \in [0, \ldots, 127]$ from $\mathbb{F}_2^{320}$ into $\mathbb{F}_2$.

Let us study the degree of an $F_j$ function depending on a particular bit of the output or on a linear combination of output bits because it is not possible to directly compute the algebraic immunity of each function $F_j$ due to the very large number of variables (320 input bits). According to the designers, the following remarks prevent the existence of low degree relations between the inputs and the outputs of $F_j$.

- The output bit $i$ after the modular addition on $\mathbb{Z}_2^{32}$ is of degree $i + 1$. (as described in [32]).

- The output bit $i$ after the $Trans$ mapping is of degree $i + 1 - 7 \mod 32$, $\forall i = 6$ and equal to 32 for $i = 6$. (as described in [32])

- The $mux$ operation does not enable to determine with probability one the exact number of bits of the initial state involved in the algebraic relation.

- The algebraic immunity of the SERPENT S-box $S_2$ at 4-bit word level is equal to 2. (see [100] for details on algebraic immunity)

The designer claims that, under the remarks stated above, the algebraic attack against SOSEMANUK is intractable.

### 4.5 Performances of SOSEMANUK

This section is devoted to the software performance of SOSEMANUK. It compares the performance of SOSEMANUK and of SNOW 2.0 on several architectures (see Table 4.1) for the keystream generation and the key setup.

All the results presented for SOSEMANUK have been computed using the reference C implementation supplied by the designers which can be found at [57].

**Code size.** The main unrolled loop implies a code size between 2 and 5 KB depending on the platform and the compiler. Therefore, the entire code fits in the L1 cache.

**Static data.** The reference C implementation uses static data tables with a total size equal to 4 KB. This amount is 3 times smaller than the size of static data required in SNOW 2.0, leading to a lower date cache pressure.

**Key setup.** Recall that the key setup (the subkey generation given by Serpent24) is made once and that each new IV injection for a given key corresponds to a small version of the block cipher SERPENT.
### Table 4.1: Comparison between Sosemanuk and SNOW 2.0: number of cycles per 32-bit word for keystream generation on several architectures

<table>
<thead>
<tr>
<th>CISC target</th>
<th>parameters</th>
<th>SOSEMANUK (cycles/W)</th>
<th>SNOW2.0 (cycles/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Frequency</td>
<td>Memory</td>
<td>Compiler</td>
</tr>
<tr>
<td>Pentium 3</td>
<td>800 MHz</td>
<td>376 MB</td>
<td>GCC 3.2.2</td>
</tr>
<tr>
<td>Pentium 4M</td>
<td>2.3 GHz</td>
<td>503 MB</td>
<td>GCC 3.2.2</td>
</tr>
<tr>
<td>Pentium 4 (prescot)</td>
<td>2.6 GHz</td>
<td>1 GB</td>
<td>GCC 3.2.2</td>
</tr>
<tr>
<td>Pentium 4 (nocona)</td>
<td>3.2 GHz</td>
<td>1 GB</td>
<td>ICC 8.1</td>
</tr>
<tr>
<td>Athlon XP 1800+</td>
<td>1.5 GHz</td>
<td>256 MB</td>
<td>GCC 3.4.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RISC target</th>
<th>parameters</th>
<th>SOSEMANUK (cycles/W)</th>
<th>SNOW2.0 (cycles/W)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Frequency</td>
<td>Memory</td>
<td>Compiler</td>
</tr>
<tr>
<td>G4 (PPC7455 v3.3)</td>
<td>1GHz</td>
<td>500 MB</td>
<td>GCC 3.3</td>
</tr>
<tr>
<td>G5 (PPC 970)</td>
<td>2GHz</td>
<td>1 GB</td>
<td>GCC 3.3</td>
</tr>
<tr>
<td>Alpha EV67</td>
<td>500 MHz</td>
<td>256 MB</td>
<td>GCC 3.4.0</td>
</tr>
<tr>
<td>Alpha EV6</td>
<td>500 MHz</td>
<td>256 MB</td>
<td>GCC 2.95.2</td>
</tr>
<tr>
<td>Alpha EV6</td>
<td>500 MHz</td>
<td>256 MB</td>
<td>DEC CC 5.9</td>
</tr>
<tr>
<td>Alpha EV5</td>
<td>500 MHz</td>
<td>384 MB</td>
<td>GCC 2.95.2</td>
</tr>
<tr>
<td>Alpha EV5</td>
<td>500 MHz</td>
<td>256 MB</td>
<td>DEC CC 5.9</td>
</tr>
<tr>
<td>Ultrasparc III</td>
<td>1.2 GHz</td>
<td>4 GB</td>
<td>GCC 3.4.0</td>
</tr>
<tr>
<td>Ultrasparc III</td>
<td>1.2 GHz</td>
<td>4 GB</td>
<td>CC Forte 5.4</td>
</tr>
<tr>
<td>MIPS R5900</td>
<td>167 MHz</td>
<td>32 MB</td>
<td>GCC 2.95</td>
</tr>
</tbody>
</table>

The performance of the key setup and of the IV setup in SOSEMANUK are directly derived from the performance of SERPENT (details in [62]). Due to intellectual property aspects, the designer’s implementation does not re-use the best implementation of SERPENT. However, the performance given in [96] leads to the following results on a Pentium 4:

- key setup $\approx 900$ cycles.
- IV setup $\approx 480$ cycles.

These estimations for the IV setup (resp. key setup) performance corresponds to about $3/4^{th}$ of the best published performance for SERPENT encryption (resp. for SERPENT key schedule). The key setup in SNOW 2.0 is done for each IV. It is assumed to take around 900 cycles on a Pentium4 (details in [53]) (the SNOW 2.0 reference implementation provides about 900 cycles on a G4 processor).

**Keystream generation.** Table 4.1 presents the performance of the keystream generation for SOSEMANUK. The reference implementation of the SNOW 2.0 cipher has been bench-
marked on the same computers in order to compare both ciphers. Table 4.1 mentions the bus frequency and the amount of RAM, but these parameters are not relevant in this context. During benchmarks, steps were taken to the effect that no memory access is supposed to be performed outside of the innermost cache level (so-called L1 cache, which is located directly on the processor). Hence external RAM size and speed do not matter here. Even if SNOW 2.0 remains faster on CISC architecture using GCC, SOSEMANUK overtakes SNOW 2.0 on the other platforms (the RISC ones) due to a better design for the mappings of data on the processor registers and a lower data cache pressure.

4.6 Cryptanalysis of SOSEMANUK

4.6.1 Improved Guess and Determine Attack on SOSEMANUK

In [3], Ahmadi, Eghlidos and Khazaee showed an improved guess-and-determine attack on SOSEMANUK with a complexity of $O(2^{226})$. This implies that, the cipher has still the 128-bit security claimed by the authors, but does not provide full security when the key of length more than 226 bits is used. Like a standard guess-and-determine attack, they attempted to obtain the states of all cells of the whole cipher system by guessing the contents of some of them initially and comparing the resulting key sequence with the running key sequence. Based on the design method of Advanced GD attacks, they first analyzed SOSEMANUK by considering some simplifying assumptions on MUX and Serpent1 which leads to an attack with the complexity of $O(2^{160})$. Next, they modified the attack by taking into account the real MUX and Serpent1 which results in an attack with computational complexity of $O(2^{226})$ on the cipher.

4.6.2 Evaluation With Regard to Guess-and-Determine Attacks

In [121], Tsunoo, Saito, Shigeri, Suzaki, Ahmadi, Eghlidos and Khazaeei described the results of the guess and determine attack made on SOSEMANUK. The attack method enables to determine all of 384-bit internal state just after the initialization, using only 24-word keystream. This attack needs about $2^{224}$ computations. Thus, when secret key length is longer than 224-bit, it needs less computational effort than an exhaustive key search, to break SOSEMANUK. The results show that the cipher has still the 128-bit security as claimed by its designers.

4.6.3 Cryptanalysis of SOSEMANUK and SNOW 2.0 Using Linear Masks

In [88], Lee, Lee and Park showed a cryptanalysis of SOSEMANUK and SNOW 2.0 using linear mask. Basically, they presented a correlation attack on SOSEMANUK with complexity less
than $2^{150}$. They showed that, by combining linear approximation relations regarding the FSM update function, the FSM output function and the keystream output function, it is possible to derive linear approximation relations with correlation $-2^{-21.41}$ involving only the keystream words and the LFSR initial state. Using such linear approximation relations, they mounted a correlation attack with complexity $2^{147.88}$ and success probability 99% to recover the initial internal state of 384 bits. Basically they have mounted the attack by combining linear masking method with techniques in [16] using fast Walsh transform to recover the initial LFSR state of Grain. Most importantly, the time, data and memory complexity are all less than $2^{150}$.

### 4.6.4 Improved Linear Cryptanalysis of SOSEMANUK

In [39], Cho and Hermelin proposed and improved linear cryptanalysis of SOSEMANUK. They applied the generalized linear masking technique to SOSEMANUK and derive many linear approximations holding with the correlations of up to $2^{-25.5}$. They showed that, the data complexity of the linear attack on SOSEMANUK can be reduced by a factor of $2^{10}$ if multiple linear approximations are used. Since SOSEMANUK claims 128-bit security, this attack would not be a real threat on the security of SOSEMANUK.

In this paper the authors improved the attack by Lee et al. (details in [88]) described in section 4.6.3. They derived the best linear approximation of SOSEMANUK by the generalized linear masking method which was applied to the distinguishing attack on SNOW 2.0 by Nyberg et al. (details in [105]). Their results show that the best linear approximation of SOSEMANUK is not a single but multiple. Moreover, many linear approximations have the same order of magnitude of the correlations as the highest one. If Lee et al.’s attack uses such multiple linear approximations holding with strong correlations, the data complexity of the attack can be reduced significantly. On the other hand, the time complexity of the attack is not much affected since the total amount of linear approximations is determined by the correlation of the dominant linear approximations. They estimated that the best attack requires around $2^{135.7}$ keystream bits with the time complexity $2^{147.4}$ and memory complexity $2^{146.8}$.

### 4.6.5 A Byte-Based Guess and Determine Attack on SOSEMANUK

In [60], Feng, Liu, Zhou, Wu and Feng presented a new byte-based guess and determine attack on SOSEMANUK, where they view a byte as a basic data unit and guess some certain bytes of the internal states instead of the whole 32-bit words during the execution of the attack. Surprisingly, their attack only needs a few words of known key stream to recover all the internal states of SOSEMANUK, and the time complexity can be dramatically reduced to $\mathcal{O}(2^{176})$. Since SOSEMANUK has a key with the length varying from 128 to 256 bits, this result showed that when the length of an encryption key is larger than 176 bits, this guess and
determine attack is more efficient than an exhaustive key search.

4.6.6 Differential Fault Analysis of SOSEMANUK

In [54], Salehani, Kircanski and Youssef presented a fault analysis attack on SOSEMANUK. The fault model in which they analyzed the cipher is the one in which the attacker is assumed to be able to fault a random inner state word but cannot control the exact location of injected faults. This attack, which recovers the secret inner state of the cipher, requires around 6144 faults, work equivalent to around $2^{48}$ SOSEMANUK iterations and a storage of around $2^{38.17}$ bytes.

4.7 Conclusion

SOSEMANUK is basically not very popular due to its complicated structure. Also, there are several attacks already demonstrated and some of them may pose real threat too. But the design idea may be used to develop a much secure stream cipher in future.

4.8 A Simple C Implementation of SOSEMANUK

Here we present a easy implementation in C. The main goal is to simplify the thing to implementors. Obviously it is not optimized. For more optimized and sophisticated version we recommend the reader to look into the codes provided in the eSTREAM portal at [58].

```c
/*
 * Developer: Sorav Sen Gupta
   email : sg.sourav@gmail.com
 * ---------------------------------------------
 * Usage:
 * 1. Compile the code using gcc or cc
 * 2. Run the executable to get 2 test vectors
 * 3. Modify Key and IV in main() function
 * ---------------------------------------------
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#if UINT_MAX >= 0xFFFFFFFF
```
typedef unsigned int unum32;
#else
typedef unsigned long unum32;
#endif
#define ONE32 ((unum32)0xFFFFFFFF)
#define T32(x) ((x) & ONE32)

/*
 * Two structures are used:
 *  
 * -- "sosemanuk_key_context" holds the processed secret key. The contents 
 * of this structure depends only on the key, not the IV.
 * 
 * -- "sosemanuk_run_context" holds the current cipher internal state. This
 * structure is initialized using the "sosemanuk_key_context" structure, and
 * the IV; it is updated each time some output is produced.
 */

typedef struct {
  /*
   * Sub-keys for Serpent24.
   */
  unum32 sk[100];
} sosemanuk_key_context;

typedef struct {
  /*
   * Internal cipher state.
   */
  unum32 s00, s01, s02, s03, s04, s05, s06, s07, s08, s09;
  unum32 r1, r2;

  /*
   * Buffering: the stream cipher produces output data by 
   * blocks of 640 bits. buf[] contains such a block, and
   * "ptr" is the index of the next output byte.
   */
  unsigned char buf[80];
  unsigned ptr;
} sosemanuk_run_context;
/*
 * 32-bit data decoding, little endian.
 */

static unum32 decode32le(unsigned char *data)
{
    return (unum32)data[0]
    | ((unum32)data[1] << 8)
    | ((unum32)data[2] << 16)
    | ((unum32)data[3] << 24);
}

/*
 * 32-bit data encoding, little-endian.
 */

static void encode32le(unsigned char *dst, unum32 val)
{
    dst[0] = val & 0xFF;
    dst[1] = (val >> 8) & 0xFF;
    dst[2] = (val >> 16) & 0xFF;
    dst[3] = (val >> 24) & 0xFF;
}

/*
 * Left-rotation by n bits (0 < n < 32).
 */
#define ROTL(x, n) (T32(((x) << (n)) | T32((x) >> (32 - (n)))))

#ifndef _S0_H
#define _S0_H

#define S0(r0, r1, r2, r3, r4) do {
    r3 ^= r0; r4 = r1; \
    r1 &= r3; r4 ^= r2; \
    r1 ^= r0; r0 |= r3; \
    r0 ^= r4; r4 ^= r3; \
    r3 ^= r2; r2 |= r1; \
}

#endif

/*****************************/

/*
 * Serpent S-boxes, implemented in bitslice mode.
 */

#define S0(r0, r1, r2, r3, r4) do {
    r3 ^= r0; r4 = r1; \
    r1 &= r3; r4 ^= r2; \
    r1 ^= r0; r0 |= r3; \
    r0 ^= r4; r4 ^= r3; \
    r3 ^= r2; r2 |= r1; \
}
4.8. A Simple C Implementation of SOSEMANUK

```c
#define S1(r0, r1, r2, r3, r4) do { \
    r0 = ~r0; r2 = ~r2; \
    r4 = r0; r0 &= r1; \
    r2 ^= r0; r0 |= r3; \
    r3 ^= r2; r1 ^= r0; \
    r0 ^= r4; r4 |= r1; \
    r1 ^= r3; r4 ^= r3; \
} while (0)

#define S2(r0, r1, r2, r3, r4) do { \
    r4 = r0; r0 &= r2; \
    r0 ^= r3; r2 ^= r1; \
    r2 ^= r0; r3 |= r4; \
    r3 ^= r1; r4 ^= r2; \
    r1 = r3; r3 |= r4; \
    r3 ^= r0; r0 &= r1; \
    r4 ^= r0; r1 ^= r3; \
    r1 ^= r4; r4 = ~r4; \
} while (0)

#define S3(r0, r1, r2, r3, r4) do { \
    r4 = r0; r0 |= r3; \
    r3 ^= r1; r1 &= r4; \
    r4 ^= r2; r2 ^= r3; \
    r3 &= r0; r4 |= r1; \
    r3 ^= r4; r0 ^= r1; \
    r4 &= r0; r1 ^= r3; \
    r4 ^= r2; r1 |= r0; \
    r1 ^= r2; r0 ^= r3; \
    r2 = r1; r1 |= r3; \
    r1 ^= r0; \
} while (0)
```

#define S4(r0, r1, r2, r3, r4) do { 
    r1 ^= r3; r3 = ~r3; 
    r2 ^= r3; r3 ^= r0; 
    r4 = r1; r1 &= r3; 
    r1 ^= r2; r4 ^= r3; 
    r0 ^= r4; r2 &= r3; 
    r2 ^= r0; r0 &= r1; 
    r3 ^= r0; r4 |= r1; 
    r4 ^= r2; r2 &= r3; 
    r0 = ~r0; r4 ^= r2; 
    r0 ^= r2; r2 &= r3; 
    r0 ^= r0; r4 ^= r2; 
    } while (0)

#define S5(r0, r1, r2, r3, r4) do { 
    r0 ^= r1; r1 ^= r3; 
    r3 = ~r3; r4 = r1; 
    r1 & r0; r2 ^= r3; 
    r1 ^= r2; r2 |= r4; 
    r4 ^= r3; r3 &= r1; 
    r3 ^= r2; r4 ^= r1; 
    r4 ^= r2; r2 &= r0; 
    r0 &= r3; r2 = ~r2; 
    r0 ^= r4; r4 &= r3; 
    r2 ^= r4; 
    } while (0)

#define S6(r0, r1, r2, r3, r4) do { 
    r2 = ~r2; r4 = r3; 
    r3 &= r0; r0 ^= r4; 
    r3 ^= r2; r2 |= r4; 
    r1 ^= r3; r2 ^= r0; 
    r0 |= r1; r2 ^= r1; 
    r4 ^= r0; r0 |= r3; 
    r0 ^= r2; r4 ^= r3; 
    r4 ^= r0; r3 = ~r3; 
    r2 &= r4; 
    r2 ^= r3; 
    } while (0)

#define S7(r0, r1, r2, r3, r4) do { 
    r4 = r1; r1 |= r2; 
    r1 ^= r3; r4 ^= r2; 
    r2 ^= r1; r3 |= r4; 

4.8. A Simple C Implementation of SOSEMANUK

```c
r3 &= r0; r4 ^= r2; \
r3 ^= r1; r1 |= r4; \n```

```c
while (0)
/*
The Serpent linear transform.
*/
#define SERPENT_LT(x0, x1, x2, x3) do { \
x0 = ROTL(x0, 13); \
x2 = ROTL(x2, 3); \
x1 = x1 ^ x0 ^ x2; \
x3 = x3 ^ x2 ^ T32(x0 << 3); \
x1 = ROTL(x1, 1); \
x3 = ROTL(x3, 7); \
x0 = x0 ^ x1 ^ x3; \
x2 = x2 ^ x3 ^ T32(x1 << 7); \
x0 = ROTL(x0, 5); \
x2 = ROTL(x2, 22); \
} while (0)
/* ----------------------------------------------------------------------- */

```c
void
sosemanuk_schedule(sosemanuk_key_context *kc, unsigned char *key, size_t key_len) {
    /*
    This key schedule is actually a truncated Serpent key schedule.
    The key-derived words (w_i) are computed within the eight
    local variables w0 to w7, which are reused again and again.
    */
#define SKS(S, o0, o1, o2, o3, d0, d1, d2, d3) do { \
        unum32 r0, r1, r2, r3, r4; \
        r0 = w ## o0; \
        r1 = w ## o1; \
        r2 = w ## o2; \
        r3 = w ## o3; \
```
S(r0, r1, r2, r3, r4); \
kc->sk[i ++] = r # d0; \
kc->sk[i ++] = r # d1; \
kc->sk[i ++] = r # d2; \
kc->sk[i ++] = r # d3; \
} while (0)

#define SKS0 SKS(S0, 4, 5, 6, 7, 1, 4, 2, 0)
#define SKS1 SKS(S1, 0, 1, 2, 3, 2, 0, 3, 1)
#define SKS2 SKS(S2, 4, 5, 6, 7, 2, 3, 1, 4)
#define SKS3 SKS(S3, 0, 1, 2, 3, 1, 2, 3, 4)
#define SKS4 SKS(S4, 4, 5, 6, 7, 1, 4, 0, 3)
#define SKS5 SKS(S5, 0, 1, 2, 3, 1, 3, 0, 2)
#define SKS6 SKS(S6, 4, 5, 6, 7, 0, 1, 4, 2)
#define SKS7 SKS(S7, 0, 1, 2, 3, 4, 3, 1, 0)

#define WUP(wi, wi5, wi3, wi1, cc) do { \
  unum32 tt = (wi) ^ (wi5) ^ (wi3) ^ (wi1) ^ (0x9E3779B9 ^ (unum32)(cc)); \
  (wi) = ROTL(tt, 11); \
} while (0)

#define WUP0(cc) do { \
  WUP(w0, w3, w5, w7, cc); \
  WUP(w1, w4, w6, w0, cc + 1); \
  WUP(w2, w5, w7, w1, cc + 2); \
  WUP(w3, w6, w0, w2, cc + 3); \
} while (0)

#define WUP1(cc) do { \
  WUP(w4, w7, w1, w3, cc); \
  WUP(w5, w0, w2, w4, cc + 1); \
  WUP(w6, w1, w3, w5, cc + 2); \
  WUP(w7, w2, w4, w6, cc + 3); \
} while (0)

unsigned char wbuf[32];
register unum32 w0, w1, w2, w3, w4, w5, w6, w7;
int i = 0;

/*
 * The key is copied into the wbuf[] buffer and padded to 256 bits
 * as described in the Serpent specification.
/*
 * (key_len == 0 || key_len > 32) {
 * fprintf(stderr, "invalid key size: \%lu\n",
 * (unsigned long)key_len);
 * exit(EXIT_FAILURE);
 * }
 * memcpy(wbuf, key, key_len);
 * if (key_len < 32) {
 * wbuf[key_len] = 0x01;
 * if (key_len < 31)
 * memset(wbuf + key_len + 1, 0, 31 - key_len);
 * }
 *
 * size_t u;
 *
 * printf("\nKey input:\t");
 * for (u = 0; u < key_len; u ++)
 * printf("%02X", key[u]);
 * printf("\n");
 *
 * w0 = decode32le(wbuf);
 * w1 = decode32le(wbuf + 4);
 * w2 = decode32le(wbuf + 8);
 * w3 = decode32le(wbuf + 12);
 * w4 = decode32le(wbuf + 16);
 * w5 = decode32le(wbuf + 20);
 * w6 = decode32le(wbuf + 24);
 * w7 = decode32le(wbuf + 28);
 *
 * printf("Key final:\t%08lx %08lx %08lx %08lx\n\t%08lx %08lx %08lx %08lx\n",
 * (unsigned long)w7, (unsigned long)w6,
 * (unsigned long)w5, (unsigned long)w4,
 * (unsigned long)w3, (unsigned long)w2,
 * (unsigned long)w1, (unsigned long)w0);
 * printf("\n");
 *
 * WUP0(0); SKS3;
 * WUP1(4); SKS2;
 * WUP0(8); SKS1;
 * WUP1(12); SKS0;
 * WUP0(16); SKS7;
 * WUP1(20); SKS6;
 * WUP0(24); SKS5;
void sosemanuk_init(sosemanuk_run_context *rc, sosemanuk_key_context *kc, unsigned char *iv, size_t iv_len)
{

    /*
     * The Serpent key addition step.
     */

    #define KA(zc, x0, x1, x2, x3) do {
        x0 ^= kc->sk[(zc)];
        x1 ^= kc->sk[(zc) + 1];
    } while (0)

    WUP1(28); SKS4;
    WUP0(32); SKS3;
    WUP1(36); SKS2;
    WUP0(40); SKS1;
    WUP1(44); SKS0;
    WUP0(48); SKS7;
    WUP1(52); SKS6;
    WUP0(56); SKS5;
    WUP1(60); SKS4;
    WUP0(64); SKS3;
    WUP1(68); SKS2;
    WUP0(72); SKS1;
    WUP1(76); SKS0;
    WUP0(80); SKS7;
    WUP1(84); SKS6;
    WUP0(88); SKS5;
    WUP1(92); SKS4;
    WUP0(96); SKS3;

    #undef SKS
    #undef SKS0
    #undef SKS1
    #undef SKS2
    #undef SKS3
    #undef SKS4
    #undef SKS5
    #undef SKS6
    #undef SKS7
    #undef WUP
    #undef WUP0
    #undef WUP1
}
x2 ^= kc->sk[(zc) + 2];
  x3 ^= kc->sk[(zc) + 3];
} while (0)

/*
* One Serpent round.
* zc = current subkey counter
* S = S-box macro for this round
* i0 to i4 = input register numbers (the fifth is a scratch register)
* o0 to o3 = output register numbers
*/
#define FSS(zc, S, i0, i1, i2, i3, i4, o0, o1, o2, o3) do {
  KA(zc, r ## i0, r ## i1, r ## i2, r ## i3);
  S(r ## i0, r ## i1, r ## i2, r ## i3, r ## i4);
  SERPENT_LT(r ## o0, r ## o1, r ## o2, r ## o3);
} while (0)

/*
* Last Serpent round. Contrary to the "true" Serpent, we keep
* the linear transformation for that last round.
*/
#define FSF(zc, S, i0, i1, i2, i3, i4, o0, o1, o2, o3) do {
  KA(zc, r ## i0, r ## i1, r ## i2, r ## i3);
  S(r ## i0, r ## i1, r ## i2, r ## i3, r ## i4);
  SERPENT_LT(r ## o0, r ## o1, r ## o2, r ## o3);
  KA(zc + 4, r ## o0, r ## o1, r ## o2, r ## o3);
} while (0)

register unum32 r0, r1, r2, r3, r4;
unsigned char ivtmp[16];

if (iv_len >= sizeof ivtmp) {
  memcpy(ivtmp, iv, sizeof ivtmp);
} else {
  if (iv_len > 0)
    memcpy(ivtmp, iv, iv_len);
  memset(ivtmp + iv_len, 0, (sizeof ivtmp) - iv_len);
}

size_t u;

printf("IV input:\t");
for (u = 0; u < 16; u ++)
printf("%02X", ivtmp[u]);
printf("\n");

/*
 * Decode IV into four 32-bit words (little-endian).
 */
long r0 = decode32le(ivtmp);
long r1 = decode32le(ivtmp + 4);
long r2 = decode32le(ivtmp + 8);
long r3 = decode32le(ivtmp + 12);

printf("IV final: \t%08lX %08lX %08lX %08lX\n",
(unsigned long)r3, (unsigned long)r2,
(unsigned long)r1, (unsigned long)r0);

/*
 * Encrypt IV with Serpent24. Some values are extracted from the
 * output of the twelfth, eighteenth and twenty-fourth rounds.
 */
FSS(0, S0, 0, 1, 2, 3, 4, 1, 4, 2, 0);
FSS(4, S1, 1, 4, 2, 0, 3, 2, 1, 0, 4);
FSS(8, S2, 2, 1, 0, 4, 3, 0, 4, 1, 3);
FSS(12, S3, 0, 4, 1, 3, 2, 4, 1, 3, 2);
FSS(16, S4, 4, 1, 3, 2, 0, 1, 0, 4, 2);
FSS(20, S5, 1, 0, 4, 2, 3, 0, 2, 1, 4);
FSS(24, S6, 0, 2, 1, 4, 3, 0, 2, 3, 1);
FSS(28, S7, 0, 2, 3, 1, 4, 4, 1, 2, 0);
FSS(32, S0, 4, 1, 2, 0, 3, 1, 3, 2, 4);
FSS(36, S1, 1, 3, 2, 4, 0, 2, 1, 4, 3);
FSS(40, S2, 2, 1, 4, 3, 0, 4, 3, 1, 0);
FSS(44, S3, 4, 3, 1, 0, 2, 3, 1, 0, 2);
rc->s09 = r3;
rc->s08 = r1;
rc->s07 = r0;
rc->s06 = r2;

FSS(48, S4, 3, 1, 0, 2, 4, 1, 4, 3, 2);
FSS(52, S5, 1, 4, 3, 2, 0, 4, 2, 1, 3);
FSS(56, S6, 4, 2, 1, 3, 0, 4, 2, 0, 1);
FSS(60, S7, 4, 2, 0, 1, 3, 3, 1, 2, 4);
FSS(64, S0, 3, 1, 2, 4, 0, 1, 0, 2, 3);
FSS(68, S1, 1, 0, 2, 3, 4, 2, 1, 3, 0);
rc->r1 = r2;
rc->s04 = r1;
rc->r2 = r3;
rc->s05 = r0;

FSS(72, S2, 2, 1, 3, 0, 4, 3, 0, 1, 4);
FSS(76, S3, 3, 0, 1, 4, 2, 0, 1, 4, 2);
FSS(80, S4, 0, 1, 4, 2, 3, 1, 3, 0, 2);
FSS(84, S5, 1, 3, 0, 2, 4, 3, 2, 1, 0);
FSS(88, S6, 3, 2, 1, 0, 4, 3, 2, 4, 1);
FSF(92, S7, 3, 2, 4, 1, 0, 0, 1, 2, 3);
rc->s03 = r0;
rc->s02 = r1;
rc->s01 = r2;
rc->s00 = r3;

printf("\n=====================================================
")
printf("\nInitial LFSR state:\n\n	s[0] = %08lX",
(unsigned long)rc->s00);
printf("ts[1] = %08lX", (unsigned long)rc->s01);
printf("ts[2] = %08lX", (unsigned long)rc->s02);
printf("ts[3] = %08lX", (unsigned long)rc->s03);
printf("ts[4] = %08lX", (unsigned long)rc->s04);
printf("ts[5] = %08lX", (unsigned long)rc->s05);
printf("ts[6] = %08lX", (unsigned long)rc->s06);
printf("ts[7] = %08lX", (unsigned long)rc->s07);
printf("ts[8] = %08lX", (unsigned long)rc->s08);
printf("ts[9] = %08lX", (unsigned long)rc->s09);
printf("\nInitial FSM state:\R1 = %08lX \tR2 = %08lX\n",
(unsigned long)rc->r1, (unsigned long)rc->r2);

#undef KA
#undef FSS
#undef FSF

} /*
 * Multiplication by alpha: alpha * x = T32(x << 8) ^ mul_a[x >> 24]
 */
static unum32 mul_a[] = {
0x00000000, 0xE19FCF13, 0x6B973726, 0x8A08F835,
0xD6876E4C, 0x3718A15F, 0xBD10596A, 0x5C8F9679,
0x05A7DC98, 0xE438138B, 0x6E30EBBE, 0x8F0F24AD,
0xD320B2D4, 0x32BF7DC7, 0xB8B785F2, 0x59284AE1,
0x0AE71199, 0xEB78DE8A, 0x617026BF, 0x80EFE9AC,
0xC2E04CD7, 0x237F83C4, 0xA9777BF1, 0x48E8B4E2,
0x11C0FE03, 0x64D7FA27, 0x85483534, 0x1E803302,
0x1B27EF9A, 0x23F65714, 0x95DE1DF5, 0x7441D2E6,
0xC749704F, 0x26D85F5C, 0xA3906416, 0xE0AD5D2A,
0x1467229B, 0x80F8ED88, 0x09E6FDAE, 0x1FF69D7D,
0x28CE449F, 0x9F518B8C, 0x8B5E2EF7, 0x7B011FE7,
0x22955066, 0x81B93F6E, 0x6026F07D, 0x5592540F,
0x0F40CD01, 0x83E9AB05, 0x4452AA0C, 0x92D5C440,
0xDD36A917, 0x573E5122, 0x734A0B53, 0x9732D5D9,
0xB7F748F3, 0x76DEAF21, 0x9F390C6C, 0x4EB5BB95,
4.8. A Simple C Implementation of SOSEMANUK

A Simple C Implementation of SOSEMANUK

0x4B12670D, 0x9A8DA81E, 0x2085502B, 0xC11A9F38, 0x9D950941, 0x7C0AC652, 0xF6023E67, 0x179DF174, 0x78FBCC08, 0x9964031B, 0x136CFB2E, 0x21D386E9, 0x721CDD91, 0x93831282, 0x198BEAB7, 0xF81425A4, 0xA49BB3DD, 0x45047CCE, 0xCF0C84FB, 0x2E9348E8, 0x77BB0109, 0x9624CE1A, 0x1C2C362F, 0xFDB3F93C, 0xA13C6F45, 0x40A3A056, 0xCAAB5863, 0xB2349770, 0xC69CEE93, 0x80D32180, 0x070BD9B5, 0xE69416A6, 0xBA1B80DF, 0x5B844FC0, 0xD18CB7F9, 0x301378EA, 0x693B320B, 0x88A4FD18, 0x02AC052D, 0xE333CA3E, 0xBFBC5C47, 0x5E239354, 0xD42B6609, 0x35B4A472, 0x667BFF0A, 0x87E43019, 0x0DECC82C, 0xEC73073F, 0xB0FC9146, 0x51635E55, 0xDB6BA660, 0x3AF46973, 0x63DC2392, 0x8243EC81, 0x084B14B4, 0xE9D4DBA7, 0xB55B4DDE, 0x54C482CD, 0xDECC7AF8, 0x3F53B5EB

static unum32 mul_ia[] = {
    0x00000000, 0x180F40CD, 0x301E8033, 0x2811C0FE, 0x603CA966, 0x7833E9AB, 0x50222955, 0x482D6998, 0xC078FBCC, 0xD877BB01, 0xF0667BFF, 0xE8693B32, 0xA04452AA, 0xB84B1267, 0x905AD299, 0x88559254, 0x29F05F31, 0x31FF1FFC, 0x19EEDF02, 0x01E19FCF, 0x49CCF657, 0x51C3B69A, 0x79D27664, 0x61DD36A9, 0xE988A4FD, 0xF187E430, 0xD99624CE, 0xC1996403, 0x89B40D9B, 0x91BB4D56, 0xB9A8DA80, 0xA15C65, 0x5249BE62, 0x4A46FEAF, 0x62573E51, 0x7A587E9C, 0x32751704, 0x2A7A57C9, 0x026B9737, 0x1A64D7FA, 0x923145AE, 0x8A3E563, 0xA22FC59D, 0xBA208550, 0xF20DECC8, 0xEA02AC05, 0xC2136CFB, 0xDA1C2C36, 0x7BB9E153, 0x63B6A19E, 0x4BA76160, 0x53A821AD, 0x1B854835, 0x038A08F8, 0x2B9BC806, 0x339488CB, 0xBBC11A9F, 0xA3CE5A52, 0xA8B4D9AC, 0x93D0DA61, 0xDCB7B3F9, 0x3CF23F34, 0xEBC333CA, 0xF3EC7307, 0xA492D5C4, 0x9CB9D509, 0x94C855F7, 0x8C83153A, 0xC4AE7CA2, 0xDA136C6F, 0xF4B0F91C, 0xECB5F5C5, 0x64EA2E08, 0x7CE56EC5, 0x54F4AE3B, 0x4CFBEEF6,
4.8. A Simple C Implementation of SOSEMANUK

0xFEDECC7A, 0xE6D18CB7, 0xCEC04C49, 0xD6CF0C84, 0x9EE2651C, 0x86ED25D1, 0xAEFCE52F, 0xB6F3A5E2

*/

* Compute the next block of bits of output stream. This is equivalent
* to one full rotation of the shift register.
*/

static void
sosemanuk_internal(sosemanuk_run_context *rc)
{
    /*
     * MUL_A(x) computes alpha * x (in F_{2^{32}}).
     * MUL_G(x) computes 1/alpha * x (in F_{2^{32}}).
     */
    #define MUL_A(x) (T32((x) << 8) ^ mul_a[(x) >> 24])
    #define MUL_G(x) (((x) >> 8) ^ mul_ia[(x) & 0xFF])
    #define XMUX(c, x, y) (((c) & 0x1) ? ((x) ^ (y)) : (x))

    /*
     * FSM() updates the finite state machine.
     */
    #define FSM(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9) do {
        unum32 tt, or1; 
        tt = XMUX(r1, s ## x1, s ## x8); 
        or1 = r1; 
        r1 = T32(r2 + tt); 
        tt = T32(or1 * 0x54655307); 
        r2 = ROTL(tt, 7); 
        PFSM; 
    } while (0)

    /*
     * LRU updates the shift register; the dropped value is stored
     * in variable "dd".
     */
    #define LRU(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, dd) do {
        dd = s ## x0; 
        s ## x0 = MUL_A(s ## x0) ^ MUL_G(s ## x3) ^ s ## x9; 
        PLFSR(dd, s ## x1, s ## x2, s ## x3, s ## x4, s ## x5, 
             s ## x6, s ## x7, s ## x8, s ## x9, s ## x0); 
    }
while (0) /*
* CC1 stores into variable "ee" the next intermediate word
* (combination of the new states of the LFSR and the FSM).
*/
#define CC1(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, ee) do { \ 
  ee = T32(s ## x9 + r1) ^ r2; \ 
  PCCVAL(ee); \ 
} while (0)

/*
* STEP computes one internal round. "dd" receives the "s_t"
* value (dropped from the LFSR) and "ee" gets the value computed
* from the LFSR and FSM.
*/
#define STEP(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, dd, ee) do { \ 
  FSM(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9); \ 
  LRU(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, dd); \ 
  CC1(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, ee); \ 
} while (0)

/*
* Apply one Serpent round (with the provided S-box macro), XOR
* the result with the "v" values, and encode the result into
* the destination buffer, at the provided offset. The "x*"
* arguments encode the output permutation of the "S" macro.
*/
#define OUTWORD_BASE (rc->buf)
#define SRD(S, x0, x1, x2, x3, ooff) do { \ 
  PSPIN(u0, u1, u2, u3); \ 
  S(u0, u1, u2, u3, u4); \ 
  PSPOUT(u ## x0, u ## x1, u ## x2, u ## x3); \ 
  encode32le(OUTWORD_BASE + ooff, u ## x0 ^ v0); \ 
  encode32le(OUTWORD_BASE + ooff + 4, u ## x1 ^ v1); \ 
  encode32le(OUTWORD_BASE + ooff + 8, u ## x2 ^ v2); \ 
  encode32le(OUTWORD_BASE + ooff + 12, u ## x3 ^ v3); \ 
  POUT(OUTWORD_BASE + ooff); \ 
} while (0)

#define PFSM (void)0
#define PLFSR(dd, x1, x2, x3, x4, x5, x6, x7, x8, x9, x0) (void)0
#define PCCVAL(ee) (void)0
#define PSPIN(x0, x1, x2, x3) (void)0
#define PSPOUT(x0, x1, x2, x3) (void)0
#define POUT(buf) (void)0

unum32 s00 = rc->s00;
unum32 s01 = rc->s01;
unum32 s02 = rc->s02;
unum32 s03 = rc->s03;
unum32 s04 = rc->s04;
unum32 s05 = rc->s05;
unum32 s06 = rc->s06;
unum32 s07 = rc->s07;
unum32 s08 = rc->s08;
unum32 s09 = rc->s09;
unum32 r1 = rc->r1;
unum32 r2 = rc->r2;
unum32 u0, u1, u2, u3, u4;
unum32 v0, v1, v2, v3;

STEP(00, 01, 02, 03, 04, 05, 06, 07, 08, 09, v0, u0);
STEP(01, 02, 03, 04, 05, 06, 07, 08, 09, 00, v1, u1);
STEP(02, 03, 04, 05, 06, 07, 08, 09, 00, 01, v2, u2);
STEP(03, 04, 05, 06, 07, 08, 09, 00, 01, 02, v3, u3);
SRD(S2, 2, 3, 1, 4, 0);
STEP(04, 05, 06, 07, 08, 09, 00, 01, 02, 03, v0, u0);
STEP(05, 06, 07, 08, 09, 00, 01, 02, 03, 04, v1, u1);
STEP(06, 07, 08, 09, 00, 01, 02, 03, 04, 05, v2, u2);
STEP(07, 08, 09, 00, 01, 02, 03, 04, 05, 06, v3, u3);
SRD(S2, 2, 3, 1, 4, 16);
STEP(08, 09, 00, 01, 02, 03, 04, 05, 06, 07, v0, u0);
STEP(09, 00, 01, 02, 03, 04, 05, 06, 07, 08, v1, u1);
STEP(00, 01, 02, 03, 04, 05, 06, 07, 08, 09, v2, u2);
STEP(01, 02, 03, 04, 05, 06, 07, 08, 09, 00, v3, u3);
SRD(S2, 2, 3, 1, 4, 32);
STEP(02, 03, 04, 05, 06, 07, 08, 09, 00, 01, v0, u0);
STEP(03, 04, 05, 06, 07, 08, 09, 00, 01, 02, v1, u1);
STEP(04, 05, 06, 07, 08, 09, 00, 01, 02, 03, v2, u2);
STEP(05, 06, 07, 08, 09, 00, 01, 02, 03, 04, v3, u3);
SRD(S2, 2, 3, 1, 4, 48);
STEP(06, 07, 08, 09, 00, 01, 02, 03, 04, 05, v0, u0);
STEP(07, 08, 09, 00, 01, 02, 03, 04, 05, 06, v1, u1);
```c
STEP(08, 09, 00, 01, 02, 03, 04, 05, 06, 07, v2, u2);
STEP(09, 00, 01, 02, 03, 04, 05, 06, 07, 08, v3, u3);
SRD(S2, 2, 3, 1, 4, 64);

rc->s00 = s00;
rc->s01 = s01;
rc->s02 = s02;
rc->s03 = s03;
rc->s04 = s04;
rc->s05 = s05;
rc->s06 = s06;
rc->s07 = s07;
rc->s08 = s08;
rc->s09 = s09;
rc->r1 = r1;
rc->r2 = r2;
}

#define XOR(x, y) x ^ y

/*
 * Combine buffers in1[] and in2[] by XOR, result in out[]. The length
 * is "data_len" (in bytes). Partial overlap of out[] with either in1[]
 * or in2[] is not allowed. Total overlap (out == in1 and/or out == in2)
 * is allowed.
 */

static void
xorbuf(const unsigned char *in1, const unsigned char *in2, unsigned char *out, size_t data_len)
{
    while (data_len -- > 0)
        *out ++ = *in1 ++ ^ *in2 ++;
}

/* see sosemanuk.h */

void
sosemanuk_prng(sosemanuk_run_context *rc, unsigned char *out, size_t out_len)
{
    if (rc->ptr < (sizeof rc->buf)) {
        size_t rlen = (sizeof rc->buf) - rc->ptr;

        if (rlen > out_len)
            rlen = out_len;

        memcpy(out, rc->buf + rc->ptr, rlen);
        out += rlen;
    }
```
out_len -= rlen;
rc->ptr += rlen;
}
while (out_len > 0) {
    sosemanuk_internal(rc);
    if (out_len >= sizeof rc->buf) {
        memcpy(out, rc->buf, sizeof rc->buf);
        out += sizeof rc->buf;
        out_len -= sizeof rc->buf;
    } else {
        memcpy(out, rc->buf, out_len);
        rc->ptr = out_len;
        out_len = 0;
    }
}

/* see sosemanuk.h */
void
sosemanuk_encrypt(sosemanuk_run_context *rc,
                   unsigned char *in,
                   unsigned char *out,
                   size_t data_len)
{
    if (rc->ptr < (sizeof rc->buf)) {
        size_t rlen = (sizeof rc->buf) - rc->ptr;
        if (rlen > data_len)
        {            rlen = data_len;
            xorbuf(rc->buf + rc->ptr, in, out, rlen);
            in += rlen;
            out += rlen;
            data_len -= rlen;
            rc->ptr += rlen;
        }
    }
    while (data_len > 0) {
        sosemanuk_internal(rc);
        if (data_len >= sizeof rc->buf) {
            xorbuf(rc->buf, in, out, sizeof rc->buf);
            in += sizeof rc->buf;
            out += sizeof rc->buf;
            data_len -= sizeof rc->buf;
        } else {
            xorbuf(rc->buf, in, out, data_len);
            rc->ptr = data_len;
        }
    }
}
/*
 * Generate 160 bytes of stream with the provided key and IV.
 */
static void
maketest(int tvn, unsigned char *key, size_t key_len, unsigned char *iv, size_t iv_len)
{
    sosemanuk_key_context kc;
    sosemanuk_run_context rc;

    unsigned char tmp[160];
    unsigned u;

    printf("\n====================================================================\n");
    printf("Test vector %d for SOSEMANUK", tvn);
    printf("\n====================================================================\n");
    sosemanuk_schedule(&kc, key, key_len);
    sosemanuk_init(&rc, &kc, iv, iv_len);
    sosemanuk_prng(&rc, tmp, sizeof tmp);

    printf("\n====================================================================\n");
    printf("\nOutput keystream:\n");
    for (u = 0; u < sizeof tmp; u++) {
        if ((u & 0x0F) == 0)
            printf("\n   ");
        printf(" %02X", (unsigned)tmp[u]);
    }
    printf("\n====================================================================\n");
    printf("\n");
}

int
main(void)
{
    static unsigned char key1[] = {
        0xA7, 0xC0, 0x83, 0xFE, 0xB7
    }
    data_len = 0;
}
}
};

static unsigned char iv1[] = {
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};

maketest(1, key1, sizeof key1, iv1, sizeof iv1);

static unsigned char key2[] = {
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77,
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF
};

static unsigned char iv2[] = {
    0x88, 0x99, 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF,
    0x00, 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77
};

maketest(2, key2, sizeof key2, iv2, sizeof iv2);

return 0;
}
Chapter 5

Trivium

5.1 Introduction

The Trivium algorithm is a hardware-efficient (profile 2), synchronous stream cipher designed by Christophe De Cannière and Bart Preneel. The cipher makes use of a 80-bit key and 80-bit initialization vector (IV); its secret state has 288 bits, consisting of three interconnected non-linear feedback shift registers of length 93, 84 and 111 bits, respectively. The cipher operation consists of two phases: the key and IV set-up and the keystream generation. Initialization is very similar to keystream generation and requires 1152 steps of the clocking procedure of Trivium. The keystream is generated by repeatedly clocking the cipher, where in each clock cycle three state bits are updated using a non-linear feedback function, and one bit of keystream is produced and output. The cipher specification states that $2^{64}$ keystream bits can be generated from each key/IV pair.

The Trivium stream cipher was designed to be compact in constrained environments and fast in applications that requires a high throughput. In particular, the cipher’s design is such that the basic throughput can be improved through parallelization (allowing computing 64 iterations at once), without an undue increase to the area required for its implementation. For instance, for 0.13 $\mu$m Standard Cell CMOS the gate count is 2599 NAND gates for one bit of output and 4921 NAND gates for the full parallelization (see [65] for more details). A 64-bit implementation in 0.25 $\mu$m 5-metal CMOS technology yields a throughput per area ratio of 129 GBit/s-mm$^2$ (see [68] for more details), which is higher than for any other eSTREAM portfolio cipher. Hardware performance of all profile-2 eSTREAM candidates (phase 3) was described in Good and Benaissa’s paper at SASC 2008 (details in [67]). Prototype quantities of an ASIC containing all phase-3 hardware candidates was designed and fabricated on 0.18 $\mu$m CMOS, as part of the eSCARGOT project at [106].
Although Trivium does not target software applications, the cipher is still reasonably efficient on a standard PC. For more information about eSTREAM ciphers performance in software, refer to the eSTREAM testing framework page here [55].

5.2 Specifications of Trivium

Trivium was designed as an exercise in exploring how far a stream cipher can be simplified without sacrificing its security, speed or flexibility. While simple designs are more likely to be vulnerable to simple, and possibly devastating, attacks (which is why the designers strongly discourage the use of Trivium at this stage), they certainly inspire more confidence than complex schemes, if they survive a long period of public scrutiny despite their simplicity. In this section we describe the specifications which can also be found at [35].

5.2.1 Key stream generation

The proposed design contains a 288-bit internal state denoted by \((s_1, \ldots, s_{288})\). The keystream generation consists of an iterative process which extracts the values of 15 specific state bits and uses them both to update 3 bits of the state and to compute 1 bit of key stream \(z_i\). The state bits are then rotated and the process repeats itself until the requested \(N \leq 2^{64}\) bits of key stream have been generated. A complete description is given by the following simple pseudo-code depicted in Algorithm 6:

\[
\begin{align*}
\text{Algorithm 6 KEYS\textsc{tream-GENERATION}} \\
\text{for } i = 1 \rightarrow N \text{ do} \\
& t_1 \leftarrow s_{66} + s_{93} \\
& t_1 \leftarrow s_{162} + s_{177} \\
& t_1 \leftarrow s_{243} + s_{288} \\
& z_i \leftarrow t_1 + t_2 + t_3 \\
& t_1 \leftarrow t_1 + s_91 \cdot s_92 + s_{171} \\
& t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264} \\
& t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69} \\
& (s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92}) \\
& (s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176}) \\
& (s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287}) \\
\text{end for}
\end{align*}
\]

Note that here, and in the rest of this document, the ‘+’ and ‘\cdot’ operations stand for addition and multiplication over GF(2) (i.e., \texttt{XOR} and \texttt{AND}), respectively. A graphical representation of
the key stream generation process can be found in Figure 5.1.

![Graphical Representation of Trivium](image)

Figure 5.1: Graphical Representation of Trivium

### 5.2.2 Key and IV setup

The algorithm is initialized by loading an 80-bit key and an 80-bit IV into the 288-bit initial state, and setting all remaining bits to 0, except for $s_{286}, s_{287}, \text{and } s_{288}$. Then, the state is rotated over 4 full cycles, in the same way as explained above, but without generating key stream bits. This is summarized as a pseudo-code in Algorithm 7:
Algorithm 7 KEY-IV-SETUP

$$(s_1, s_2, \ldots, s_{93}) \leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0)$$

$$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$$

$$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$$

for $i = 1 \rightarrow 4 \cdot 288$ do

\begin{align*}
  t_1 & \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{171} \\
  t_1 & \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{264} \\
  t_1 & \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{69} \\
  (s_1, s_2, \ldots, s_{93}) & \leftarrow (t_3, s_1, \ldots, s_{92}) \\
  (s_{94}, s_{95}, \ldots, s_{177}) & \leftarrow (t_1, s_{94}, \ldots, s_{176}) \\
  (s_{178}, s_{179}, \ldots, s_{288}) & \leftarrow (t_2, s_{178}, \ldots, s_{287})
\end{align*}

end for

5.3 Implementation of Trivium

5.3.1 Hardware Implementation

Trivium is a hardware oriented design focused on flexibility. It aims to be compact in environments with restrictions on the gate count, power-efficient on platforms with limited power resources, and fast in applications that require high-speed encryption. The requirement for a compact implementation suggests a bit-oriented approach. It also favors the use of a nonlinear internal state, in order not to waste all painfully built up nonlinearity at the output of the key stream generator. In order to allow power-efficient and fast implementations, the design must also provide a way to parallelize its operations. In the case Trivium, this is done by ensuring that any state bit is not used for at least 64 iterations after it has been modified. This way, up to 64 iterations can be computed at once, provided that the 3 AND gates and 11 XOR gates in the original scheme are duplicated a corresponding number of times. This allows the clock frequency to be divided by a factor 64 without affecting the throughput.

Based on [87], we can compute an estimation of the gate count for different degrees of parallelization. The results are listed in Table 5.1.

<table>
<thead>
<tr>
<th>Components</th>
<th>1-bit</th>
<th>8-bit</th>
<th>16-bit</th>
<th>32-bit</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip-flops</td>
<td>288</td>
<td>288</td>
<td>288</td>
<td>288</td>
<td>288</td>
</tr>
<tr>
<td>AND gates</td>
<td>3</td>
<td>24</td>
<td>48</td>
<td>96</td>
<td>192</td>
</tr>
<tr>
<td>XOR gates</td>
<td>11</td>
<td>88</td>
<td>176</td>
<td>352</td>
<td>704</td>
</tr>
<tr>
<td>NAND gate count</td>
<td>3488</td>
<td>3712</td>
<td>3968</td>
<td>4480</td>
<td>5504</td>
</tr>
</tbody>
</table>

Table 5.1: Estimated gate counts of 1-bit to 64-bit hardware implementations
5.4. Security Properties of Trivium

5.3.2 Software Implementation

Despite the fact that Trivium does not target software applications, the cipher is still reasonably efficient on a standard PC. The measured performance of the reference C-code on an 1.5 GHz Xeon processor can be found in Table 5.2.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream generation</td>
<td>12 cycles/byte</td>
</tr>
<tr>
<td>Key Setup</td>
<td>55 cycles</td>
</tr>
<tr>
<td>IV Setup</td>
<td>2050 cycles</td>
</tr>
</tbody>
</table>

Table 5.2: Measured performance on an Intel® Xeon™ CPU 1.5 GHz

5.3.3 Few other Hardware Implementations

A low-cost implementation of Trivium

In [101], Nele Mentens, Jan Genoe, Bart Preneel and Ingrid Verbauwhede described the implementation of two Trivium cores on a single chip. The cores are realized in a 5-metal 0.35µm AMIS technology. The first core on the chip is an automatically placed and routed standard cell core. The second one is a custom design using dynamic logic and C2 MOS flipflops. The goal of this paper was to evaluate and compare the size of the cores based on the lay-out results. The lay-out of the custom design shows a significant size reduction compared to the standard cell design.

Few Other comparative results

Other comparative results are discussed in section 7.6.4, section 6.6.3, section 7.6.5, section 7.6.6 and section 7.6.7.

5.4 Security Properties of Trivium

In this section we briefly discuss some of the cryptographic properties of Trivium. For a more detailed analysis of the cipher, we refer to the paper [47]. The security requirement we impose on Trivium is that any type of cryptographic attack should not be significantly easier to apply to trivium than to any other imaginable stream cipher with the same external parameters (i.e., any cipher capable of generating up to $2^{64}$ bits of key stream from an 80-bit secret key and an 80-bit IV). Unfortunately, this requirement is not easy to verify, and henceforth the designers
provided arguments why they believe that certain common types of attacks are not likely to affect the security of the cipher.

5.4.1 Correlation

From a brief observation one can easily find linear correlations between key stream bits and internal state bits, since $z_i$ is simply defined to be equal to $s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$. However, as opposed to LFSR based ciphers (e.g. SOSEMANUK), Trivium’s state evolves in a nonlinear way, and it is not clear how the attacker should combine these equations in order to efficiently recover the state.

An easy way to find correlations of the second type is to follow linear trails through the cipher and to approximate the outputs of all encountered AND gates by 0. However, the positions of the taps in Trivium have been chosen in such a way that any trail of this specific type is forced to approximate at least 72 AND gate outputs. An example of a correlated linear combination of key stream bits obtained this way is:

$$z_1 + z_{16} + z_{28} + z_{43} + z_{46} + z_{55} + z_{61} + z_{73} + z_{88} + z_{124} + z_{133} + z_{142} + z_{202} + z_{211} + z_{220} + z_{289}$$

If we assume that the correlation of this linear combination is completely explained by the specific trail we considered, then it would have a correlation coefficient of $2^{-72}$. Detecting such a correlation would require at least $2^{144}$ bits of key stream, which is well above the security requirement.

Other more complicated types of linear trails with larger correlations might exist, but at this stage it seems unlikely that these correlations will exceed $2^{-40}$. This issue is discussed in more details in the paper [47].

5.4.2 Period

Because of the fact that the internal state of Trivium evolves in a nonlinear way, its period is hard to determine. Still, a number of observations can be made. First, if the AND gates are omitted (resulting in a completely linear scheme), one can show that any key/IV pair would generate a stream with a period of at least $2^{96-3} - 1$. This has no immediate implications for Trivium itself, but it might be seen as an indication that the taps have been chosen properly.

Secondly, Trivium’s state is updated in a reversible way, and the initialization of $(s_{178}, \ldots, s_{288})$ prevents the state from cycling in less than 111 iterations. If we believe that Trivium behaves as a random permutation after a sufficient number of iterations, then all cycle lengths up to
2^{288} would be equiprobable, and hence the probability for a given key/IV pair to cause a cycle smaller than 2^{80} would be 2^{-208}.

5.4.3 Guess and Determine attacks

In each iteration of Trivium, only a few bits of the state are used, despite the general rule-of-thumb that sparse update functions should be avoided. As a result, guess and determine attacks are certainly a concern. A straightforward attack would guess \((s_{25}, \ldots, s_{93})\), \((s_{97}, \ldots, s_{177})\), and \((s_{244}, \ldots, s_{288})\), 195 bits in total, after which the rest of the bits can immediately be determined from the key stream.

5.4.4 Algebraic attacks

Trivium seems to be a particularly attractive target for algebraic attacks. The complete scheme can easily be described with extremely sparse equations of low degree. However, its state does not evolve in a linear way, and hence the efficient linearization techniques used to solve the systems of equations generated by LFSR based schemes will be hard to apply. However, other techniques might be applicable and their efficiency in solving this particular system of equations needs to be investigated.

5.4.5 Resynchronization attacks

Another type of attacks are resynchronization attacks, where the adversary is allowed to manipulate the value of the IV, and tries to extract information about the key by examining the corresponding key stream. Trivium tries to preclude this type of attacks by cycling the state a sufficient number of times before producing any output. It can be shown that each state bit depends on each key and IV bit in a nonlinear way after two full cycles (i.e., \(2 \cdot 288\) iterations). We expect that two more cycles will suffice to protect the cipher against resynchronization attacks.

5.5 Cryptanalysis of Trivium

5.5.1 Cryptanalytic Results on Trivium

In [111], Håvard Raddum used a novel technique to try to solve a system of equations associated with Trivium. Due to the short key-length compared to the size of the internal state of Trivium (80 to 288 bits), no efficient attack on full trivium was obtained. But, the reduced versions corresponding to the design’s 'basic construction' was broken by this approach. In this paper
they set up systems of sparse equations describing the full Trivium and reduced versions, and tried to solve them by using a new technique described in [112]. By this approach they showed that the full Trivium is still not broken, but that reduced versions with two registers instead of three is broken significantly faster than exhaustive search. Also, since their approach was algebraic in nature (solving equation systems) the attack requires very little known key-stream, as opposed to most other types of attacks that typically requires enormous amounts of known key-stream. This makes this kind of attack much more threatening in a real-world setting.

5.5.2 Two Trivial Attacks on Trivium

In [98], Alexander Maximov and Alex Biryukov observed a class of Trivium-like designs. They proposed a set of techniques that one can apply in cryptanalysis of such constructions. The first group of methods is for recovering the internal state and the secret key of the cipher, given a piece of a known keystream. Their attack is more than $2^{30}$ faster than the best known attack till then. Another group of techniques allows to gather statistics on the keystream, and to build a distinguisher.

They studied two designs: the original design of Trivium and a truncated version Bivium, which follows the same design principles as the original. They showed that the internal state of the full Trivium can be recovered in time around $c \cdot 2^{83.5}$, and for Bivium this complexity is $c \cdot 2^{36.1}$. Moreover, a distinguisher for Bivium with working time $2^{32}$ was presented, the correctness of which had been verified by simulations.

5.5.3 An Algebraic Analysis based on the Boolean SAT Problem

In [99], Cameron McDonald, Chris Charnes, and Josef Pieprzyk focused on an algebraic analysis which uses the boolean satisfiability problem in propositional logic. For reduced variants of the cipher viz. Bivium, this analysis recovers the internal state with a minimal amount of keystream observations.

In this paper they considered the problem of solving a system of non-linear equations over $\mathbb{F}_2$ as a corresponding SAT-problem of propositional logic. That is, they converted the algebraic equations describing the cipher into a propositional formula in conjunctive normal form (CNF). They used a SAT-solver to solve the resulting SAT-problem, which allowed them under certain conditions to recover the key. They needed to guess a subset of the state variables in order to reduce the complexity of the system, before it can be solved by a SAT-solver. The solution returned by the SAT-solver is the remaining unknown state variables. Once the entire state is known, the cipher is clocked backwards to recover the key. The characteristic feature of this type of attack is that only minimal amount of observed keystream are required in order to
5.5. Cryptanalysis of Trivium

Due to the unpredictable behavior and complexity of the SAT-solvers, the results obtained in this paper are derived from experiments. Both attacks on Bivium require only 177 bits of keystream. The average attack time on Bivium-A is 21 seconds. The average complexity of the attack on Bivium-B is $2^{42.7}$ seconds. Both of these attack are faster than an exhaustive key search. The attack complexity on Trivium is worse than an exhaustive search.

5.5.4 Differential Fault Analysis of Trivium

In [72], Michal Hojsík and Bohuslav Rudolf presented differential fault analysis of Trivium and proposed two attacks on Trivium using fault injection. They supposed that an attacker can corrupt exactly one random bit of the inner state and that he can do this many times for the same inner state. This can be achieved e.g. in the CCA scenario. During experimental simulations, having inserted 43 faults at random positions, they were able to disclose the trivium inner state and afterwards the private key. This is the first time differential fault analysis is applied to a stream cipher based on shift register with non-linear feedback.

Since they supposed that an attacker can inject a fault only to a random position, they also described a simple method for fault position determination. Afterwards knowing the corresponding faulty keystream, they directly recovered few inner state bits and obtain several linear equations in inner state bits. Just by repeating this procedure for the same inner state but for different (randomly chosen) fault positions they recovered the whole cipher inner state, and clocking it backwards they were able to determine the secret key. The drawback of this simple approach is that they needed many fault injections to be done in order to have enough equations.

To decrease number of faulty keystreams needed (i.e. to decrease the number of fault injections needed), they also used quadratic equations given by a keystream difference. But did not use all quadratic equations, but just those which contains only quadratic monomials of a special type, where the type follows directly from the cipher description. In this way they were able to recover the whole trivium inner state using approximately 43 fault injections. As mentioned above, presented attacks require many fault injections to the same Trivium inner state. This can be achieved in the chosen-ciphertext scenario, assuming that the initialization vector is the part of the cipher input. In this case, an attacker will always use the same cipher input (initialization vector and ciphertext) and perform the fault injection during the deciphering process. Hence, proposed attacks could be described as chosen-ciphertext fault injection attacks.

They did not consider usage of any sophisticated methods for solving systems of polynomial equations (e.g. Gröbner basis algorithms). They worked with simple techniques which
naturally raised from the analysis of the keystream difference equations. Hence the described attacks are easy to implement. This also shows how simple is to attack Trivium by differential fault injection.

5.5.5 Floating Fault Analysis of Trivium

In [73] Michal Hojsík and Bohuslav Rudolf again presented an improvement of the previous attack in [72]. It requires only 3.2 one-bit fault injections in average to recover the Trivium inner state (and consequently its key) while in the best case it succeeds after 2 fault injections. They termed this attack floating fault analysis since it exploits the floating model of the cipher. The use of this model leads to the transformation of many obtained high-degree equations into linear equations. This work showed how a change of the cipher representation may result in much better attack.

5.5.6 Algebraic Attack Against Trivium

In [114], Ilaria Simonetti, Ludovic Perret and Jean Charles Faugère presented some basic results comparing a basic Gröbner basis attack against trivium and its truncated versions Bivium-A and Bivium-B. They showed how to generate a system of equations over \( \mathbb{F}_2 \) for Trivium and Bivium. They used two method: first, they added three variables (or two for Bivium) for each clock of the cipher; in the second one they used as variables only the 288 bits (or 177 bits for Bivium) of the internal state at the beginning. In the last section they used these two approaches and computed the Gröbner basis of the system. They gave some experimental complexity results, which are comparable with the previous known results.

5.5.7 Cube attacks on Trivium

In [14], S. S. Bedi and N. Rajesh Pillai discussed cube attack proposed in [122, 49]. Independent verification of the equations given in [49] and [122] were carried out. Experimentation showed that the precomputed equations were not general. They are holding when applied to the class of IVs for which they were computed — where IV bits at locations other than those corresponding to the cube are fixed at 0. When these IV bits are fixed at some other values, the relations do not hold. The probable cause for this is given and an extra step to the method for equation generation is suggested to take care of such cases.
5.5.8 Floating Fault Analysis of Trivium under Weaker Assumptions

In [75], Hu Yupu, Gao Juntao and Liu Qing presented an improvement of the previous attack [73]. In this paper, the attack is under the following weaker and more practical assumption:

- The fault injection can be made for the state at a random time.
- The positions of the fault bits are from random one of 3 NFSRs, and from a random area within 8 neighboring bits.

They presented a checking method, by which either the injecting time and fault positions can be determined, or the state differential at a known time can be determined. Each of these two determinations is enough for floating attack. After the determination, the attacker can averagely obtain 67.167 additional linear equations from 82 original quadratic equations, and obtain 66 additional quadratic equations from 66 original cubic equations. A modification of this model is similarly effective with the model of Michal Hojsík and Bohuslav, in [73] for the floating attack.

5.5.9 Hard Fault Analysis of Trivium

In [76], Yupu Hu, Fengrong Zhang, and Yiwei Zhang considered another type of fault analysis of stream cipher, which is to simplify the cipher system by injecting some hard faults. They called it hard fault analysis. They presented the following results about such attack to Trivium.

In Case 1 with the probability not smaller than 0.2396, the attacker can obtain 69 bits of 80-bits-key. In Case 2 with the probability not smaller than 0.2291, the attacker can obtain all of 80-bits-key. In Case 3 with the probability not smaller than 0.2291, the attacker can partially solve the key. In Case 4 with non-negligible probability, the attacker can obtain a simplified cipher, with smaller number of state bits and slower non-linearization procedure. In Case 5 with non-negligible probability, the attacker can obtain another simplified cipher. Besides, these 5 cases can be checked out by observing the key stream.

5.5.10 Analysis of Trivium by a Simulated Annealing Variant

In [31], Julia Borghoff, Lars R. Knudsen, and Krystian Matusiewicz proposed a new method of solving certain classes of systems of multivariate equations over the binary field and its cryptanalytical applications. They showed how heuristic optimization methods such as hill climbing algorithms can be relevant to solving systems of multivariate equations. A characteristic of equation systems that may be efficiently solvable by the means of such algorithms is provided. As an example, they investigated equation systems induced by the problem of
recovering the internal state of the stream cipher Trivium. They proposed an improved variant of the simulated annealing method that seems to be well-suited for this type of system and provided some experimental results.

In this paper they also investigated systems of sparse multivariate equations. The important additional requirement they made is that each variable appears only in a very limited number of equations. The equation system generated by the key stream generation algorithm of the stream cipher Trivium satisfies those properties and examined in this paper as the main example. The fully determined Trivium systems consists of 954 equations in 954 variables. Solving this system allows us to recover the 288-bit initial state.

This approach considered the problem of finding a solution for the system as an optimization problem and then applies an improved variant of simulated annealing to it. As opposed to the XL and XSL algorithms, the simulated annealing algorithm does not increase the size of the problem, it does not generate more nor change the existing equations. The only additional requirement is an objective function, called the cost function, that should be minimized.

With their experiments demonstrated in this work, they were not able to break Trivium in the cryptographic sense which means with a complexity equivalent to less than $2^{80}$ key setups and the true complexity of their method against Trivium is unknown. However, if the Trivium system purely as a multivariate quadratic Boolean system in 954 variables is considered then the system will be solved significantly faster than brute force, namely in around $2^{210}$ bit flips which is roughly equivalent to $2^{203}$ evaluations of the system. This shows that their variant of simulated annealing seems to be a promising tool for solving non-linear Boolean equation systems with certain properties.

5.5.11 The Cube Attack on Stream Cipher Trivium and Quadraticity Tests

In [104], Piotr Mroczkowski and Janusz Szmidt developed quadraticity tests within the cube attack and applied them to a variant of stream cipher Trivium reduced to 709 initialization rounds. Using this method the full 80-bit secret key could be obtained. In this way it eliminates the stage of brute force search of some secret key bits which occurred in the previous cube attacks [49].

5.5.12 Improved Differential Fault Analysis of Trivium

In [103], Mohamed Saied Emam Mohamed, Stanislav Bulygin, and Johannes Buchmann provided an example of combining DFA attacks and algebraic attacks. They used algebraic methods to improve the DFA of Trivium [73]. Their improved DFA attack recovers the inner state of Trivium by using only 2 fault injections and only 420 keystream bits.
5.5.13 Conditional Differential Cryptanalysis of Trivium

In [85], Simon Knellwolf, Willi Meier, and María Naya-Plasencia presented an improved technique of conditional differential cryptanalysis by using automatic tools to find and analyze the involved conditions. Using these improvements they cryptanalyzed the stream cipher Trivium and the KATAN family of lightweight block ciphers. For both ciphers they obtained new cryptanalytic results which were the best known at that time. For reduced variants of Trivium they obtained a class of weak keys that can be practically distinguished up to 961 of 1152 rounds.

The most relevant cryptanalytic results on Trivium are obtained by cube attacks [49] and by cube testers [116, 11]. In this paper, the analysis can be seen as a refinement of cube testers. Table 5.3 summarizes the results and compares them to existing analysis.

<table>
<thead>
<tr>
<th>Rounds</th>
<th>Complexity</th>
<th># Keys</th>
<th>Types of Attack</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>767</td>
<td>$2^{45}$</td>
<td>all</td>
<td>key-recovery</td>
<td>[49]</td>
</tr>
<tr>
<td>790</td>
<td>$2^{31}$</td>
<td>all</td>
<td>distinguisher</td>
<td>[11]</td>
</tr>
<tr>
<td>806</td>
<td>$2^{44}$</td>
<td>all</td>
<td>distinguisher</td>
<td>[116]</td>
</tr>
<tr>
<td>767</td>
<td>$2^{45}$</td>
<td>$2^{31}$</td>
<td>distinguisher</td>
<td>This Paper</td>
</tr>
<tr>
<td>767</td>
<td>$2^{45}$</td>
<td>$2^{26}$</td>
<td>distinguisher</td>
<td>This Paper</td>
</tr>
</tbody>
</table>

Table 5.3: Cryptanalytic results for Trivium.

5.6 Conclusion

Trivium has the simplest structure among all the ciphers in eSTREAM. May be due to the simplicity many cryptanalysts targeted it. Essentially, there is a trade off: the simplicity versus security. However, lot of new techniques of cryptanalysis are invented to attack this cipher. Also, it must be kept in mind that, the goal of the designers of Trivium was rather different from making the strongest cipher. Their simple approach obviously founded a platform to construct a much stronger but simple cipher in future.

5.7 A Simple C Implementation of Trivium

Here we present one simple C implementation of the cipher. For more sophisticated and optimized implementation the reader may look into the eSTREAM page at [58].

/****************************
Developer: Goutam Paul
email: goutam.paul@ieee.organization
#include<stdio.h>

#define ROUND 64 * 8

char key[81], iv[81], s[289], t1, t2, t3;

void getkey() {
    int i;
    char d, b, hkey[21], hiv[21];

    printf("Enter the 80-bit key as 20 hex digits: ");
    scanf("%s",hkey);

    printf("Enter the 80-bit IV as 20 hex digits: ");
    scanf("%s",hiv);

    // Convert the hex string hkey[0...19] to binary and store it in key[1...80]
    for(i=0;i<20;i+=2) {
        if(hkey[i]=='0' & hkey[i]<='9') d=hkey[i]-'0';
        else if (hkey[i]=='a' & hkey[i]<='f') d=hkey[i]-'a'+10;
        else if (hkey[i]=='A' & hkey[i]<='F') d=hkey[i]-'A'+10;
        b=d/8; d=d%8; key[4*i+8]= b;
        b=d/4; d=d%4; key[4*i+7]= b;
        b=d/2; d=d%2; key[4*i+6]= b;
        b=d; key[4*i+5]= b;
    }

    // Convert the hex string hiv[0...19] to binary and store it in iv[1...80]
    for(i=0;i<20;i+=2) {
        if(hiv[i]=='0' & hiv[i]<='9') d=hiv[i]-'0';
        else if (hiv[i]=='a' & hiv[i]<='f') d=hiv[i]-'a'+10;
        else if (hiv[i]=='A' & hiv[i]<='F') d=hiv[i]-'A'+10;
        b=d/8; d=d%8; key[4*i+8]= b;
        b=d/4; d=d%4; key[4*i+7]= b;
        b=d/2; d=d%2; key[4*i+6]= b;
        b=d; key[4*i+5]= b;
    }


```c
else if (hiv[i] >= 'A' && hiv[i] <= 'F') d = hiv[i] - 'A' + 10;
  b = d / 8; d = d % 8; iv[4 * i + 8] = b;
else if (hiv[i] >= '0' && hiv[i] <= '9') d = hiv[i] - '0';
else if (hiv[i] >= 'a' && hiv[i] <= 'f') d = hiv[i] - 'a' + 10;
else if (hiv[i] >= 'A' && hiv[i] <= 'F') d = hiv[i] - 'A' + 10;
  b = d / 8; d = d % 8; iv[4 * i + 7] = b;
  b = d / 2; d = d % 2; iv[4 * i + 6] = b;
  b = d; iv[4 * i + 5] = b;

if (hiv[i + 1] >= '0' && hiv[i + 1] <= '9') d = hiv[i + 1] - '0';
else if (hiv[i + 1] >= 'a' && hiv[i + 1] <= 'f') d = hiv[i + 1] - 'a' + 10;
else if (hiv[i + 1] >= 'A' && hiv[i + 1] <= 'F') d = hiv[i + 1] - 'A' + 10;
  b = d / 8; d = d % 8; iv[4 * i + 4] = b;
  b = d / 2; d = d % 2; iv[4 * i + 3] = b;
  b = d; iv[4 * i + 2] = b;
  b = d; iv[4 * i + 1] = b;
}
}

void updatestate() {
  int i;

  t1 = (t1 + s[91] * s[92] + s[171]) % 2;
  t2 = (t2 + s[175] * s[176] + s[264]) % 2;
  t3 = (t3 + s[286] * s[287] + s[69]) % 2;

  for (i = 92; i >= 1; i--) s[i + 1] = s[i];
  s[1] = t3;

  for (i = 176; i >= 94; i--) s[i + 1] = s[i];
  s[94] = t1;

  for (i = 287; i >= 178; i--) s[i + 1] = s[i];
  s[178] = t2;
}

void ksa() {
  int i;

  getkey();

  // Initialize the internal state
  for (i = 1; i <= 80; i++) s[i] = key[81 - i];
  for (i = 81; i <= 93; i++) s[i] = 0;
```
for(i = 1; i <= 80; i++) s[93+i] = iv[81-i];

for(i = 174; i <= 285; i++) s[i] = 0;
s[286] = s[287] = s[288] = 1;

// Update state for 4 * 288 rounds without any output

for(i = 1; i <= 4 * 288; i++) {
    t1 = (s[66] + s[93])%2;
    t2 = (s[162] + s[177])%2;
    t3 = (s[243] + s[288])%2;
    updatestate();
}

main() {
    int r;
    char z[ROUND], n1, n2;

    ksa();

    // Generate keystream

    for(r = 0; r < ROUND; r++) {
        t1 = (s[66] + s[93])%2;
        t2 = (s[162] + s[177])%2;
        t3 = (s[243] + s[288])%2;
        z[r] = (t1 + t2 + t3)%2;
        updatestate();
    }

    // Print keystream output as hex digits

    printf("\nFirst %d keystream bits: ", ROUND);
    for(r=0;r<ROUND;r+=8) {
        if(r%128 == 0) printf("\n\t\t");
        n1=z[r]+z[r+1]*2+z[r+2]*4+z[r+3]*8;
        n2=z[r+4]+z[r+5]*2+z[r+6]*4+z[r+7]*8;
        printf("%X%X",n2,n1);
    }
    printf("\n");
5.7. A Simple C Implementation of Trivium
Chapter 6

Grain v1

6.1 Introduction

Grain is best described as a family of hardware-efficient (profile 2), synchronous stream ciphers. The cipher’s initial version (described in detail in [70]) used an 80-bit key and a 64-bit initialization vector but analysis in the early stages of the eSTREAM effort compromised its security (see [16] for details). The revised specification, Grain v1, described two stream ciphers: one for 80-bit (with 64-bit initialization vector) and another for 128-bit keys (with 80-bit initialization vector). Elegant and simple, Grain v1 has been an attractive choice for cryptanalysts and implementors alike with two shift registers: one with linear feedback and the second with non-linear feedback; being the essential feature of the algorithm family. These registers, and the bits that are output, are coupled by means of very lightweight, but judiciously-chosen boolean functions.

For the version that takes 80-bit keys, the specification given by Grain v1 is the currently recommended one. However, cryptanalysis of the 128-bit version of Grain v1 has led to the proposal of a new version called Grain 128a (see [2] for details) very recently. This variant also specifies some additional registers to enable the calculation of a message authentication code in addition to generating a keystream. While Grain 128a retains the elegance of earlier versions of the cipher, in its fastest implementation it now occupies more space (2700 GE) and runs at half the speed of Grain v1. However, the design of the Grain family allows for an ingenious multiplication of throughput speed, though at the cost of a minor increase in the space consumed. Hardware performance of all profile-2 eSTREAM candidates (phase 3) was described in Good and Benaissa’s paper at SASC 2008 (see [67] for details). Prototype quantities of an ASIC containing all phase-3 hardware candidates was designed and fabricated on 0.18 µm CMOS, as part of the eSCARGOT project at [106].
Like many stream ciphers, there is some cost incurred during initialization and the impact of this will depend on the intended application and the likely size of the messages being encrypted.

6.2 Specifications of Grain v1

Here we briefly discuss the specifications of Grain-128. It is describe thoroughly in [70] and in [93]. An overview of the different blocks used in the cipher can be found in Figure 6.1 and the specification will refer to this figure.

The cipher consists of three main building blocks, namely an LFSR, an NFSR and an output function. The content of the LFSR is denoted by \( s_i, s_{i+1}, \ldots, s_{i+127} \). Similarly, the content of NFSR is denoted by \( b_i, b_{i+1}, \ldots, b_{i+127} \). The feedback polynomial of the LFSR, denoted by \( f(x) \), is a primitive polynomial of degree 128. It is defined as follows:

\[
f(x) = 1 + x^{32} + x^{47} + x^{58} + x^{90} + x^{121} + x^{128}.
\]

To remove any possible ambiguity we also give the corresponding update function of the LFSR as,

\[
s_{i+128} = s_i + s_{i+7} + s_{i+38} + s_{i+70} + s_{i+81} + s_{i+96}.
\]

The nonlinear feedback polynomial of the NFSR, \( g(x) \), is the sum of one linear and one bent function. It is defined as,

\[
g(x) = 1 + x^{32} + x^{37} + x^{72} + x^{102} + x^{128} + x^{44} + x^{60} + x^{61} + x^{125} + x^{63} + x^{67} + x^{69} + x^{101} + x^{80} + x^{88} + x^{110} + x^{111} + x^{115} + x^{117}.
\]

Again, to remove any possible ambiguity we also write the corresponding update function of the NFSR. In the update function below, note that the bit \( s_i \) which is masked with the

Figure 6.1: An overview of Grain-128
input to the NFSR is included, while omitted in the feedback polynomial.

\[
b_{i+128} = s_i + b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + b_{i+11}b_{i+13} + b_{i+17}b_{i+18} + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84}. \tag{6.4}
\]

The 256 memory elements in the two shift registers represent the state of the cipher. From this state, 9 variables are taken as input to a Boolean function, \( h(x) \). Two inputs to \( h(x) \) are taken from the NFSR and seven are taken from the LFSR. This function is of degree 3 and very simple. It is defined as,

\[
h(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_9x_4x_8 \tag{6.5}
\]

where the variables \( x_0, x_1, x_2, x_3, x_4, x_5, x_7 \) and \( x_8 \) correspond to the tap positions \( b_{i+12}, s_{i+8}, s_{i+13}, s_{i+20}, b_{i+95}, s_{i+42}, s_{i+69}, s_{i+79}, s_{i+95} \) respectively. The output function is defined as,

\[
z_i = \sum_{j \in A} b_{i+j} + h(x) + s_{i+93} \tag{6.6}
\]

where \( A = \{2, 15, 36, 45, 64, 73, 89\} \).

### 6.2.1 Key and IV Initialization

Before keystream is generated the cipher must be initialized with the key and the IV. Let the bits of the key, \( k \), be denoted \( k_i \), \( 0 \leq i \leq 127 \) and the bits of the IV be denoted \( IV_i \), \( 0 \leq i \leq 95 \). Then the initialization of the key and IV is done as follows. The 128 NFSR elements are loaded with the key bits, \( b_i = k_i \), \( 0 \leq i \leq 127 \), then the first 96 LFSR elements are loaded with the IV bits, \( s_i = IV_i \), \( 0 \leq i \leq 95 \). The last 32 bits of the LFSR is filled with ones, \( s_i = 1 \), \( 96 \leq i \leq 127 \). After loading key and IV bits, the cipher is clocked 256 times without producing any keystream. Instead the output function is fed back and \( \text{xored} \) with the input, both to the LFSR and to the NFSR. Figure 6.2 depicts the process visually.

### 6.2.2 Throughput Rate

Both shift registers are regularly clocked so the cipher will output 1 bit/clock. Using regular clocking is an advantage compared to stream ciphers which uses irregular clocking or decimation of the output sequence, since no hardware consuming output buffer is needed. Regular clocking is also an advantage when considering side-channel attacks. It is possible to increase the speed of the cipher at the expense of more hardware. This is an important feature of the Grain family of stream ciphers compared to many other stream ciphers. Increasing the speed can very easily be done by just implementing the small feedback functions, \( f(x) \) and \( g(x) \), and the
output function several times. In order to simplify this implementation, the last 31 bits of the shift registers, \( s_i \), \( 97 \leq i \leq 127 \) and \( b_i \), \( 97 \leq i \leq 127 \) are not used in the feedback functions or in the input to the output function. This allows the speed to be easily multiplied by up to 32 if a sufficient amount of hardware is available. For more discussion about the hardware implementation of Grain-128, we refer to section 6.5. An overview of the implementation when the speed is doubled can be seen in Figure 6.3. Naturally, the shift registers also need to be implemented such that each bit is shifted \( t \) steps instead of one when the speed is increased by a factor \( t \). By increasing the speed by a factor 32, the cipher will output 32 bits/clock. Since, in the key initialization, the cipher is clocked 256 times, the possibilities to increase the speed is limited to factors \( \leq 32 \) that are divisible by 256. The number of clockings needed in the key initialization phase is then \( \frac{256}{t} \). Since the output and feedback functions are small, it is quite feasible to increase the throughput in this way.
6.3 Security Properties of Grain

6.3.1 Linear Approximations

Linear sequential circuit approximations was first introduced by Golić in [63]. It is shown that it is always possible to find a linear function of output bits that is unbalanced. For linear approximations, the designer of Grain studied the structure of the Grain design in general. They considered an arbitrary choice of functions \( g(\cdot) \), \( h(\cdot) \) and \( f(\cdot) \). The number of taps taken from the two registers in the function \( h(\cdot) \) is also arbitrary. Here, the function \( f(\cdot) \) is a primitive generating polynomial used for the LFSR. A Boolean nonlinear function \( g(\cdot) \) is applied to generate a new state of the NFSR. Finally, the keystream is the output of another Boolean function \( h(\cdot) \). Note that, to simplify notation, the function \( h(\cdot) \) in this section also includes the linear terms added in the output function.

The results in this section was first given in [97] as follows. Let \( A_g(\cdot) \) and \( A_h(\cdot) \) be linear approximations for \( g(\cdot) \) and \( h(\cdot) \) with the biases \( g \) and \( h \), respectively. That is,

\[
\Pr\{A_g(\cdot) = g(\cdot)\} = \frac{1}{2} + \epsilon_g, \tag{6.7}
\]
\[
\Pr\{A_h(\cdot) = h(\cdot)\} = \frac{1}{2} + \epsilon_h, \tag{6.8}
\]

Then, there exists a time invariant linear combination of the keystream bits and LFSR bits, such that this equation has the following bias:

\[
\epsilon = 2^{\eta(A_h) + \eta(A_g) - 1} \cdot \epsilon_g^{\eta(A_g)} \cdot \epsilon_h^{\eta(A_h)} \tag{6.9}
\]

where \( \eta(a(\cdot)) \) is the number of the NFSR state variables used in some function \( a(\cdot) \). This bias can not immediately be used in cryptanalysis since also the LFSR has to be taken into account. However, as soon as the bias is large, a distinguishing or even a key-recovery attack can be mounted by \( e.g. \), finding a low weight parity check equation for the LFSR. When we talk about correlation attacks of different kinds, it has been shown in [97] that the strength of Grain is directly based on the difficulty of the general decoding problem (GDP), well-known as a hard problem.

6.3.2 Algebraic Attacks

In Grain-128, an NFSR is used to introduce nonlinearity together with the function \( h(\cdot) \). Solving equations for the initial 256 bit state is not possible due to the nonlinear update of the NFSR. The algebraic degree of the output bit expressed in initial state bits will be large in general and also varying in time. And the designer’s claim is that, this will defeat any algebraic attack on the cipher.
6.3.3 Time-Memory-Data Trade-off Attack

A generic time-memory-data trade-off attack on stream ciphers costs $O(2^{n/2})$, (see [24] for details), where $n$ is the number of inner state variables in the stream cipher. In Grain-128, the two shift registers are of size 128 each so the total number of state variables is 256. Thus, the designers claimed that, expected complexity of a time-memory-data trade-off attack should not be lower than $O(2^{128})$.

6.3.4 Fault Attacks

While considering the fault attack, the designers made the strongest assumption possible, namely that the adversary can introduce one single fault in a location of the LFSR that he can somehow determine. Note that this assumption may not be at all realistic. They aimed to look at the input-output properties for $h(\cdot)$, and to get information about the inputs from known input-output pairs. As long as the difference does not propagate to position $b_{i+95}$ the difference that can be observed in the output is coming only from inputs of $h(\cdot)$ from the LFSR. If the attacker is able to reset the cipher many times, each time introducing a new fault in a known position that he can guess from the output difference, then we can not preclude that he will get information about a subset of state bits in the LFSR. Considering the more realistic assumption that the adversary is not able to control the number of faults that have been inserted then it seems more difficult to determine the induced difference from the output differences. It is also possible to introduce faults in the NFSR. These faults will never propagate to the LFSR, but the faults introduced here will propagate nonlinearly in the NFSR and their evolution will be harder to predict. Thus, introducing faults into the NFSR seems more difficult than into the LFSR.

6.4 Design Choices of Grain

In this section we briefly give the reasonings behind the choices for the parameters used in Grain-128 according to the designers. Section 6.3 clearly shows that a proper choice of design parameters is important.

6.4.1 Size of the LFSR and the NFSR

The size of the key in Grain-128 is 128 bits. Because of the simple and generic time-memory-data trade-off attack, the internal state must be at least twice as large as the size of the key. Therefore, the LFSR and the NFSR to be of size 128 bits was chosen.
6.4.2 Speed Acceleration

Although the binary hardware implementation of Grain is small and fast, its speed can still be increased significantly. The functions \( f(\cdot) \), \( g(\cdot) \), and \( h(\cdot) \) can be implemented several times, so that several bits can be produced in parallel at the same time. In Grain-128 the designers explicitly allowed up to 32 times speed acceleration. Many software oriented ciphers are word based with a word size of 32 bits. These ciphers output 32 bits in every clock or iteration. If needed, Grain-128 can also be implemented to output 32 bits/clock. For a simple implementation of this speed acceleration the functions \( f(\cdot) \), \( g(\cdot) \), and \( h(\cdot) \) should not use variables taken from the first 31 taps of the LFSR and the NFSR. Obviously, speed acceleration is a trade-off between speed and hardware complexity. Speed can additionally be increased even more, by allowing the internal state to be increased proportionally. For more discussion on the throughput, see Section 6.2.2.

6.4.3 Choice of \( f(\cdot) \)

This function is the generating polynomial for the LFSR, thus, it must be primitive. It has been shown (in e.g., [36]) that if the function \( f(\cdot) \) is of low weight, there exist different correlation attacks. Therefore, the number of taps to be used for the generating function \( f(\cdot) \) should be larger than five. A large number of taps is also undesirable due to the complexity of the hardware implementation.

6.4.4 Choice of \( g(\cdot) \)

This Boolean function is used for the NFSR, generating a nonlinear relation of the state of the register. The design of this function must be carefully chosen so that the attack given in Section 6.3.1 will not be possible. Recall that the bias of the output will depend on the number of terms in the best linear approximation of \( g(\cdot) \). It will also depend on the bias of this approximation. To increase the number of terms in the best linear approximation, the resiliency of the function must be high. On the other hand, to have as small bias as possible in the best approximation, the function should have high nonlinearity. It is well known that a bent function has the highest possible nonlinearity. However, bent functions can not be balanced. In order to have both high resiliency and nonlinearity, a highly resilient (linear) function is used together with a bent function. The bent function \( b(\cdot) \) is the chosen function.

\[
b(x) = x_0x_1 + x_2x_3 + x_4x_5 + x_6x_7 + x_8x_9 + x_{10}x_{11} + x_{12}x_{13}.
\]  

This function has nonlinearity 8128. To increase the resiliency, 5 linear terms are added to the function. This will result in a balanced function with resiliency 4 and nonlinearity
\(2^5 \cdot 8128 = 260096\). This is an easy way to construct functions with high resiliency and nonlinearity. Another important advantage of this function is that it is very small and cheap to implement in hardware. The best linear approximation is any linear function using at least all the linear terms. There are 214 such functions and they have bias \(g = 2^{-8}\).

### 6.4.5 Choice of output function

The output function consists of the function \(h(x)\) and terms added linearly from the two shift registers. This guarantees that the output will depend on the state of both registers. The function \(h(x)\) takes input from both the LFSR and the NFSR. Similar to the function \(g(\cdot)\), the bias of the output will depend on the number of terms in the best linear approximation of this function and also the bias of this approximation. Hence, this function has the same design criteria as \(g(\cdot)\). The function \(h(x)\) has nonlinearity 240 and since in total 8 variables are added linearly the output function has in total nonlinearity \(2^8 \cdot 240 = 61440\). The function \(h(x)\) is not balanced and the best linear approximations have bias \(\epsilon_h = 2^{-5}\). There are in total 256 linear approximations with this bias.

### 6.5 Hardware Performance of Grain-128

The Grain family of stream ciphers is designed to be very small in hardware. In this section we give an estimate of the gate count resulting from a hardware implementation of the cipher. The gate count for a function depends on the complexity and functionality. The numbers are no natural constants and will depend on the implementation in an actual chip. Usually, the gate count is based on a 2 input nand gate which is defined to have gate count 1. Hence, the gate count can be seen as the equivalent number of nand gates in the implementation. Table 6.1 lists the equivalent gate count for the building blocks used in our estimation. The total gate count for the different functions can be seen in Table 6.2. This is just an estimate and the numbers are not exact, \(e.g.,\) the multiplexers needed in order to switch between key/IV loading, initialization and keystream generation are not included in the count. Also, two extra \texttt{xors} are needed in key initialization mode. However, excluding these things results in insignificant deviations from the real values. The exact number of gates needed for each function will depend on the implementation anyway.

### 6.6 Hardware Implementations of Grain

In this section we discuss a few other hardware implementations apart from the designers.
### 6.6.1 An Improved Implementation of Grain

In [92], Shohreh Sharif Mansouri and Elena Dubrova showed how to further improve the hardware efficiency of Grain stream cipher. By transforming the NLFSR of Grain from its original Fibonacci configuration to the Galois configuration and by introducing a clock division block, they doubled the throughput of the 80 and 128-bit key 1bit/cycle architectures of Grain with no area penalty.

### 6.6.2 Design and Implementation Based on SABL Logic

In [110], R. Ebrahimi Atani, W. Meier, S. Mirzakuchaki and S. Ebrahimi Atani provided a brief overview of hiding countermeasures. In this paper, they exploited Sense Amplifier Based Logic (SABL) to counteract power analysis in Grain stream cipher. Power traces of the resulting circuits exhibit that SABL significantly reduces the signal to noise ratio (SNR).

Simulations showed DPA resistivity of SABL implementation of Grain-128 has a major improvement. The paper presented the tradeoffs involved in designing the architecture, the design for performance issues and the possibilities for future development.

---

### Table 6.1: The Gate Count Used For Different Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Gate Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAND2</td>
<td>1</td>
</tr>
<tr>
<td>NAND3</td>
<td>1.5</td>
</tr>
<tr>
<td>XOR2</td>
<td>2.5</td>
</tr>
<tr>
<td>D flip flop</td>
<td>8</td>
</tr>
</tbody>
</table>

### Table 6.2: The Estimated Gate Count in an Actual Implementation

<table>
<thead>
<tr>
<th>Building Block</th>
<th>Speed Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1x</td>
</tr>
<tr>
<td>LFSR</td>
<td>1024</td>
</tr>
<tr>
<td>NFSR</td>
<td>1024</td>
</tr>
<tr>
<td>$f(\cdot)$</td>
<td>12.5</td>
</tr>
<tr>
<td>LFSR</td>
<td>37</td>
</tr>
<tr>
<td>LFSR</td>
<td>35.5</td>
</tr>
<tr>
<td>Total</td>
<td>2133</td>
</tr>
</tbody>
</table>
6.6.3 Comparison of Low-Power Implementations of Trivium and Grain

In [59], Martin Feldhofer provided a comparison of the Grain and Trivium. They evaluated these algorithms concerning their feasibility to implement them for low-power applications in RFID systems. A triple of parameters which includes the chip area, the power consumption, and the number of clock cycles for encrypting a fixed amount of data is introduced which allows a fair comparison of the proposals. The datapaths of the implementations are presented in detail and the synthesis results are shown. A comparison of the results of Grain and Trivium with an AES implementation shows that the chip area of Trivium is slightly smaller while Grain requires less clock cycles for encrypting 128 bits of data. The low-power implementations of the stream ciphers require only a fourth of the mean current consumption of the AES algorithm.

6.6.4 Other comparative studies

Other comparative studies which include hardware implementation of Grain are discussed in section 7.6.4, section 7.6.5, section 7.6.6 and section 7.6.7.

6.7 Cryptanalysis of Grain

6.7.1 Slide Resynchronization Attack on the Initialization of Grain v1

In [86], Özgül Küçük presented an attack in which they found keys and initial values of the stream cipher Grain v1. For any \((K, IV)\) pair there exist related \((K', IV')\) pair with probability \(1/2^2\) that generates 1-bit shifted keystream. Their method can be applied to various stream ciphers. They called it slide resynchronization attack because it is the application of slide attack (see [27] for more details on slide attacks) on block ciphers to the initialization of stream ciphers. Although this does not result in an efficient key recovery attack yet, it indicates a weakness in the initialization which could be overcomed with a little effort.

6.7.2 Related-Key Chosen IV Attack on Grain-v1 and Grain-128

In [89], Lee, Jeong, Sung and Hong extended the above related-key chosen IV attack (proposed in [86]) which finds related keys and IVs of Grain that generate the 1-bit shifted keystream sequence. In this paper, they extended the above attack and proposed related-key chosen IV attacks on Grain-v1 and Grain-128. The attack on Grain-v1 recovers the secret key with \(2^{22.59}\) chosen IVs, \(2^{31.39}\)-bit keystream sequences and \(2^{27.01}\) computational complexity.
6.7.3 Cryptanalysis of Grain using Time / Memory / Data Tradeoffs

In [28], T.E. Bjøstad showed that Grain has a low resistance to BSW sampling, leading to tradeoffs that in the active phase recover the internal state of Grain v1 using $O(2^{71})$ time and memory, and $O(2^{53.5})$ bits of known keystream. While the practical significance of these tradeoffs may be arguable due to the precomputation costs involved, their existence clearly violate stated design assumptions in the Grain specification, and parallels may be drawn to the similar cryptanalytic results on and the subsequent tweak of MICKEY v1.

6.7.4 Distinguishing Attack on Grain using Linear Sequential Circuit Method

In [81], Shahram Khazaei, Mahdi M. Hasanzadeh and Mohammad S. Kiaei, derived a linear function of consecutive keystream bits which is held with correlation coefficient of about $2^{-63.7}$ using the linear sequential circuit approximation method, introduced by Golić in 1994 (see [63] for more details). Then using the concept of so-called generating function, they turned it into a linear function with correlation coefficient of $2^{-29}$ which shows that the output sequence of Grain can be distinguished from a purely random sequence using about $O(2^{61.4})$ bits of the output sequence with the same time complexity. A preprocessing phase for computing a trinomial multiple of a certain primitive polynomial with degree 80 is needed which can be performed using time and memory complexities of $O(2^{40})$.

6.7.5 Analysis of Grain’s Initialization Algorithm

In [34], Christophe De Cannière, Özgül Küçük and Bart Preneel analyzed the initialization algorithm of Grain. They pointed out the existence of a sliding property in the initialization algorithm of the Grain family, and showed that it could be used to reduce by half the cost of exhaustive key search (currently the most efficient attack on both Grain v1 and Grain-128). Also they analyzed the differential properties of the initialization, and mounted several attacks, including a differential attack on Grain v1 which recovers one out of $2^9$ keys using two related keys and $2^{55}$ chosen IV pairs.

6.7.6 Fault Analysis of Grain-128

In [37], Alexandre Berzati, Cecile Canovas, Guilhem Castagnos, Blandine Debraize, Louis Goubin, Aline Gouget, Pascal Paillier and Stephanie Salgado introduced a fault attack on GRAIN-128 based on a realistic fault model and explores possible improvements of the attack.

The model they assumed is as follows: The adversary is assumed to be able to flip exactly one bit lying in one position in the LFSR without choosing its location but at a chosen point
in time. Fault injection is performed \textit{e.g.}, by lighting up the device with laser beams. The attacker has only partial control on the locations of the faults but he is assumed to be able to inject a fault over and over again at his will at the same position. In addition, the attacker is assumed to have full control over timing. The attacker is also assumed to be able to reset the cryptographic device to its original state and then apply another randomly chosen fault to the same device. Assuming this model, they proposed a fault attack on GRAIN-128 by which, with an average number of 24 consecutive faults in the LFSR state, they could recover the secret key within a couple of minutes of off-line computation. They also proposed some realistic countermeasures which protect GRAIN-128 at low extra cost.

\subsection*{6.7.7 An Experimentally Verified Attack on Full Grain-128}

In \cite{48}, Itai Dinur, Tim Güneysu, Christof Paar, Adi Shamir and Ralf Zimmermann described the first single-key attack which can recover the full key of the full version of Grain-128 for arbitrary keys by an algorithm which is significantly faster than exhaustive search (by a factor of about $2^{38}$). It was based on a new version of a cube tester, which used an improved choice of dynamic variables to eliminate the previously made assumption that ten particular key bits are zero. In addition, the new attack is much faster than the previous weak-key attack, and has a simpler key recovery process. Since it is extremely difficult to mathematically analyze the expected behavior of such attacks, they implemented it on RIVYERA, which is a new massively parallel reconfigurable hardware, and tested its main components for dozens of random keys. These tests experimentally verified the correctness and expected complexity of the attack, by finding a very significant bias in this new cube tester for about 7.5\% of the keys they tested. This was the first time that the main components of a complex analytical attack were successfully realized against a full-size cipher with a special-purpose machine. Moreover, it was also the first attack that truly exploits the configurable nature of an FPGA-based cryptanalytic hardware.

\subsection*{6.7.8 Breaking Grain-128 with Dynamic Cube Attacks}

In \cite{50}, Itai Dinur and Adi Shamir presented a new variant of cube attacks called a \textit{dynamic cube attack}. Whereas standard cube attacks (see details in \cite{49}) find the key by solving a system of linear equations in the key bits, this attack recovers the secret key by exploiting distinguishers obtained from cube testers. Dynamic cube attacks can create lower degree representations of the given cipher, which makes it possible to attack schemes that resist all previously known attacks. Their first attack runs in practical time complexity and recovers the full 128-bit key when the number of initialization rounds in Grain-128 is reduced to 207. Their second attack broke a Grain-128 variant with 250 initialization rounds and is faster than exhaustive search.
by a factor of about $2^{28}$. Finally, they presented an attack on the full version of Grain-128 which can recover the full key but only when it belongs to a large subset of 2-10 of the possible keys. This attack is faster than exhaustive search over the $2^{118}$ possible keys by a factor of about $2^{15}$.

### 6.7.9 Fault analysis of Grain-128 by targeting NFSR

In [79], Sandip Karmakar showed that Grain-128 can also be attacked by inducing faults in the NFSR. The attack requires about 56 fault injections for NFSR and a computational complexity of about $2^{21}$.

### 6.8 Conclusion

Evidently, Grain-128 had some real weaknesses and naturally, it has a number of attacks which with the complexity much lower than the exhaustive key-search. But, Grain-v1 is a newly renovated version which has not yet suffered from that many attacks until now. Due to its simplicity, Grain v1 is also a popular stream cipher when considering implementations on hardware platforms.

### 6.9 A Simple C++ implementation of Grain v1

Here we present a simple understandable C++ implementation. For more sophisticated and optimized version of codes the reader is strongly recommended to look into the submitted C code in the eSTREAM portal at [58].

```c
/****************************
Developer: Subhadeep Banik
   email : s.banik_r@isical.ac.in
****************************/

// In this implementation of GRAIN the most significant bit of the 1st hex value is
treated as index 0
#include<stdio.h>
#include<conio.h>
main()
{
   int lfsr[80],nfsr[80],tl,tn,zr[80];
   static char ki[21],IV[17];
   int t,tt,i,ie1,ie2,ie,a0;
```
int op;

printf("Enter Initial Key in HEX 20 bit :");
scanf("%s",ki);
// ki stores the secret key
printf("Enter Initial vector in HEX 16 bit :");
scanf("%s",IV);
// IV stores the initial vector

// convert the string ki to binary and store it in the NFSR
for(i=0;i<20;i+=2)
{
    if(ki[i]>'0' && ki[i]<'9') ie =ki[i]-'0';
    else if (ki[i]>'a' && ki[i]<'f') ie=ki[i]-'a'+10;
    else if (ki[i]>'A' && ki[i]<'F') ie=ki[i]-'A'+10;
    a0=ie/8; ie=ie%8; nfsr[4*i]=a0;
    a0=ie/4; ie=ie%4; nfsr[4*i+1]=a0;
    a0=ie/2; ie=ie%2; nfsr[4*i+2]=a0;
    a0=ie; nfsr[4*i+3]=a0;
    if(ki[i+1]>'0' && ki[i+1]<'9') ie =ki[i+1]-'0';
    else if (ki[i+1]>'a' && ki[i+1]<'f') ie=ki[i+1]-'a'+10;
    else if (ki[i+1]>'A' && ki[i+1]<'F') ie=ki[i+1]-'A'+10;
    a0=ie/8; ie=ie%8; nfsr[4*i+4]=a0;
    a0=ie/4; ie=ie%4; nfsr[4*i+5]=a0;
    a0=ie/2; ie=ie%2; nfsr[4*i+6]=a0;
    a0=ie; nfsr[4*i+7]=a0;
}

// convert the string IV to binary and store it in the LFSR
for(i=0;i<16;i+=2)
{
    if(IV[i]>'0' && IV[i]<'9') ie =IV[i]-'0';
    else if (IV[i]>'a' && IV[i]<'f') ie=IV[i]-'a'+10;
    else if (IV[i]>'A' && IV[i]<'F') ie=IV[i]-'A'+10;
    a0=ie/8; ie=ie%8; lfsr[4*i]=a0;
    a0=ie/4; ie=ie%4; lfsr[4*i+1]=a0;
    a0=ie/2; ie=ie%2; lfsr[4*i+2]=a0;
    a0=ie; lfsr[4*i+3]=a0;
if (IV[i+1] >= '0' && IV[i+1] <= '9') ie = IV[i+1] - '0';
else if (IV[i+1] >= 'a' && IV[i+1] <= 'f') ie = IV[i+1] - 'a' + 10;
else if (IV[i+1] >= 'A' && IV[i+1] <= 'F') ie = IV[i+1] - 'A' + 10;
a0 = ie / 8; ie = ie % 8; lfsr[4*i+4] = a0;
a0 = ie / 4; ie = ie % 4; lfsr[4*i+5] = a0;
a0 = ie / 2; ie = ie % 2; lfsr[4*i+6] = a0;
a0 = ie; lfsr[4*i+7] = a0;
}
// The last 16 bits of LFSR are initialized to ones to avoid all zero state
for (i=64;i<80;i++) lfsr[i] = 1;

// initialisation process
for (tt=0;tt<160;tt++) {
    for (i=0;i<=78;i++) lfsr[i] = lfsr[i + 1]; lfsr[79] = (tl + op) % 2;
    for (i=0;i<=78;i++) nsr[i] = nsr[i + 1]; nsr[79] = (tn + op) % 2;
}
// Stream generation for the first 80 clocks
for (tt=0;tt<80;tt++) {
    for (i=0;i<=78;i++) lfsr[i] = lfsr[i + 1]; lfsr[79] = (tl + op) % 2;
    for (i=0;i<=78;i++) nsr[i] = nsr[i + 1]; nsr[79] = (tn + op) % 2;
}
// zr stores output

for (i=0; i<=78; i++) lfsr[i] = lfsr[i+1]; lfsr[79] = tl;
for (i=0; i<=78; i++) nfsr[i] = nfsr[i+1]; nfsr[79] = tn;
}

// print zr in hex
printf("\n");
for (t=0; t<80; t+=8)
{

    printf("%x%x", ie1, ie2);
}

getch();
Chapter 7

MICKEY 2.0

7.1 Introduction

MICKEY 2.0 is a hardware-efficient (profile 2), synchronous stream cipher designed by Steve Babbage and Matthew Dodd. The cipher makes use of a 80-bit key and an initialization vector with up to 80 bits in length. The name MICKEY is short for "Mutual Irregular Clocking KEYstream generator". The cipher secret state consists of two 100-bit shift registers, one linear and one nonlinear, each of which is irregularly clocked under control of the other. The specific clocking mechanisms contribute cryptographic strength while still providing guarantees on period and pseudorandomness. The cipher specification states that each key can be used with up to $2^{40}$ different IVs of the same length, and that $2^{40}$ keystream bits can be generated from each key/IV pair. The designers have also specified a scaled-up version of the cipher called MICKEY-128 2.0, which takes a 128-bit key and an IV up to 128 bits.

MICKEY 2.0 can be implemented with a particularly small hardware footprint, making it a good candidate where low gate count or low power are the primary requirements. The irregular clocking means that it cannot readily be parallelized so as to run at high speed in software. Hardware performance of all profile-2 eSTREAM candidates (phase 3) was described in Good and Benaissa’s paper at SASC 2008 (see [65] for more details). Prototype quantities of an ASIC containing all phase-3 hardware candidates was designed and fabricated on 0.18 $\mu$m CMOS, as part of the eSCARGOT project (see [106] for more details).

It has been noted, e.g. by Gierlichs et al. at SASC 2008 (see [61] for details), that straightforward implementations of the MICKEY ciphers are likely to be susceptible to timing or power analysis attacks, where these are relevant. Otherwise there have been no known cryptanalytic advances against MICKEY 2.0 or MICKEY-128 2.0 since the publication of the eSTREAM portfolio.
7.2 Specifications of MICKEY 2.0

7.2.1 Input and Output Parameters

MICKEY 2.0 takes two input parameters:

- an 80-bit secret key $K$, whose bits are labelled $k_0, \ldots, k_{79}$;
- an initialization variable $IV$, anywhere between 0 and 80 bits in length, whose bits are labelled $iv_0, \ldots, iv_{l-1}$, where $l$ is the length of $IV$.

The keystream bits output by MICKEY 2.0 are labelled $z_0, z_1, \ldots$. Ciphertext is produced from plaintext by bitwise XOR with keystream bits, as in most stream ciphers.

7.2.2 Acceptable use

The maximum length of keystream sequence that may be generated with a single $(K, IV)$ pair is $2^{40}$ bits. It is acceptable to generate $2^{40}$ such sequences, all from the same $K$ but with different values of $IV$. It is not acceptable to use two initialization variables of different MICKEY 2.0 specification lengths with the same $K$. And it is not, of course, acceptable to reuse the same value of $IV$ with the same $K$.

7.2.3 Components of the keystream generator

The Registers

The generator is built from two registers $R$ and $S$. Each register is 100 stages long, each stage containing one bit. We label the bits in the registers $r_0, \ldots, r_{99}$ and $s_0, \ldots, s_{99}$ respectively. Broadly speaking, the reader may think of $R$ as the linear register and $S$ as the non-linear register.

Clocking the register $R$

Define a set of feedback tap positions for $R$:

$$RTAPS = \{0, 1, 3, 4, 5, 6, 9, 12, 13, 16, 19, 20, 21, 22, 25, 28, 37, 38, 41, 42, 45, 46, 50, 52, 54, 56, 58, 60, 61, 63, 64, 65, 66, 67, 71, 72, 79, 80, 81, 82, 87, 88, 89, 90, 91, 92, 94, 95, 96, 97\}$$

We define an operation $\text{CLOCK}_R(R, INPUT\_BIT\_R, CONTROL\_BIT\_R)$ as the following pseudocode in Algorithm 8:
Algorithm 8 CLOCK_R

{Let \( r_0, \ldots, r_{99} \) be the state of the register \( R \) before clocking, and let \( r'_{0}, \ldots, r'_{99} \) be the state of the register \( R \) after clocking.}\}

\( FEEDBACK\_BIT = r_{99} \oplus INPUT\_BIT\_R \)

for \( i = 1 \rightarrow 99 \) do
\( r'_i = r_{i-1} \)
\( r'_0 = 0 \)
end for

for \( i = 0 \rightarrow 99 \) do
    if \( i \in RTAPS \) then
        \( r'_i = r'_i \oplus FEEDBACK\_BIT \)
    end if
end for

if \( CONTROL\_BIT\_R = 1 \) then
    for \( i = 0 \rightarrow 99 \) do
        \( r'_i = r'_i \oplus r_i \)
    end for
end if

---

| \( i \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| \( COPO_0 \) | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| \( COPO_1 \) | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 |
| \( FI_0 \) | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| \( FI_1 \) | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |

| \( i \) | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 |
| \( COPO_0 \) | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| \( COPO_1 \) | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 |
| \( FI_0 \) | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| \( FI_1 \) | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |

| \( i \) | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 |
| \( COPO_0 \) | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| \( COPO_1 \) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| \( FI_0 \) | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \( FI_1 \) | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |

| \( i \) | 75 | 76 | 77 | 78 | 79 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 |
| \( COPO_0 \) | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| \( COPO_1 \) | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| \( FI_0 \) | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| \( FI_1 \) | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

Figure 7.1: Initialization Values for register \( S \)
Clocking the register \( S \)

Define four sequences \( COMP_0, \ldots, COMP_{98}, COMP_1, \ldots, COMP_{98}, FB_0, \ldots, FB_{99}, FB_1, \ldots, FB_{199} \) in figure 7.1.

We define an operation \( \text{CLOCK}_S(S, INPUT\_BIT\_S, CONTROL\_BIT\_S) \) as pseudocode in Algorithm 9:

**Algorithm 9 CLOCK\_S**

\[
\begin{align*}
\{ & \text{Let } s_0, \ldots, s_{99} \text{ be the state of the register } S \text{ before clocking, and let } \\
& s_0, \ldots, s_{99} \text{ be the state of the register after clocking. We will also use } \\
& \hat{s}_0, \ldots, \hat{s}_{99} \text{ as intermediate variables to simplify the specification.} \\
& \text{FEEDBACK\_BIT} = s_{99} \oplus INPUT\_BIT\_S \\
\text{for } i = 1 \rightarrow 98 \text{ do} \\
& \hat{s}_i = s_{i-1} \oplus ((s_i \oplus COMP_0_i)(s_{i+1} \oplus COMP_1_i)) \\
& \hat{s}_0 = 0 \\
& \hat{s}_{99} = s_{98} \\
\text{end for} \\
\text{if } CONTROL\_BIT\_S = 0 \text{ then} \\
& \text{for } i = 0 \rightarrow 99 \text{ do} \\
& s'_i = \hat{s}_i \oplus (FB_0_i \cdot \text{FEEDBACK\_BIT}) \\
& \text{end for} \\
\text{else} \\
& \text{if } CONTROL\_BIT\_S = 1 \text{ then} \\
& \text{for } i = 0 \rightarrow 99 \text{ do} \\
& s'_i = \hat{s}_i \oplus (FB_1_i \cdot \text{FEEDBACK\_BIT}) \\
& \text{end for} \\
& \text{end if} \\
& \text{end if} \\
\end{align*}
\]

Clocking the overall generator

We define an operation \( \text{CLOCK}_{KG}(R, S, MIXING, INPUT\_BIT) \) as a pseudocode in Algorithm 10:

### 7.2.4 Key-loading and Initialization

The registers are initialized from the input variables according to pseudocode given in Algorithm 11:
Algorithm 10 C\textsc{lock\_}\textsc{kg} \\
\textsc{Control\_bit\_r} = s_{34} \oplus r_{67} \\
\textsc{Control\_bit\_s} = s_{67} \oplus r_{33} \\
\textbf{if} \textsc{mixing} = \textsc{true} \textbf{then} \\
\quad \textsc{input\_bit\_r} = \textsc{input\_bit} \oplus s_{50} \\
\textbf{else} \\
\quad \textbf{if} \textsc{mixing} = \textsc{false} \textbf{then} \\
\quad \quad \textsc{input\_bit\_r} = \textsc{input\_bit} \\
\quad \textbf{end if} \\
\textbf{end if} \\
\textsc{input\_bit\_s} = \textsc{input\_bit} \\
\textsc{clock\_r}(r, \textsc{input\_bit\_r}, \textsc{control\_bit\_r}) \\
\textsc{clock\_s}(s, \textsc{input\_bit\_s}, \textsc{control\_bit\_s})

Algorithm 11 \textsc{key-load-initialization} \\
\{\textsc{load in iv}\} \\
\textbf{for} \ i = 0 \rightarrow \textsc{iv\_length} - 1 \ \textbf{do} \\
\quad \textsc{clock\_kg} ( r, s, \textsc{mixing} = \textsc{true}, \textsc{input\_bit} = iv_i ) \\
\textbf{end for} \\
\{\textsc{load in k}\} \\
\textbf{for} \ i = 0 \rightarrow 79 \ \textbf{do} \\
\quad \textsc{clock\_kg} ( r, s, \textsc{mixing} = \textsc{true}, \textsc{input\_bit} = k_i ) \\
\textbf{end for} \\
\{\textsc{preclock}\} \\
\textbf{for} \ i = 0 \rightarrow 99 \ \textbf{do} \\
\quad \textsc{clock\_kg} ( r, s, \textsc{mixing} = \textsc{true}, \textsc{input\_bit} = 0 ) \\
\textbf{end for}
### 7.2.5 Generating keystream

Having loaded and initialized the registers, the keystream bits $z_0, \ldots, z_{L-1}$ are generated according to Algorithm 12:

**Algorithm 12** KEYSTREAM-GENERATION

```plaintext
for $i = 0 \rightarrow L - 1$ do
    $z_i = r_0 \oplus s_0$
    $\text{CLOCK}_{\text{KG}} (R, S, \text{MIXING} = \text{FALSE}, \text{INPUT\_BIT} = 0)$
end for
```

### 7.3 Design principles and Security Properties of MICKEY 2.0

#### 7.3.1 The variable clocking of $R$

When $\text{CONTROL\_BIT\_R} = 0$, the clocking of $R$ is a standard linear feedback shift register clocking operation (with Galois-style feedback, following the primitive characteristic polynomial $C_R(x) = x^{100} + \sum_{i \in RTAPS} x^i$, with $\text{INPUT\_BIT\_R}$ XORed into the feedback).

If we represent elements of the field $GF(2^{100})$ as polynomials $\sum_{i=0}^{99} r_i x^i$ modulo $C_R(x)$, then shifting the register corresponds to multiplication by $x$ in the field.

![Diagram](image.png)

**Figure 7.2:** Clocking the $R$ register with $\text{CONTROL\_BIT\_R} = 0$

When $\text{CONTROL\_BIT} = 1$, as well as shifting each bit in the register to the right, it is also XORed back into the current stage, as shown in Figure 7.3. This corresponds to multiplication by $x + 1$ in the same field.

The characteristic polynomial $C_R(x)$ has been chosen so that $C_R(x)|x^{J} + x + 1$, where $J = 2^{50} - 157$. Thus, clocking the register with $\text{CONTROL\_BIT\_R} = 1$ is equivalent to clocking the register $J$ times.

This technique – a simple operation, related to the standard linear register clocking oper-
7.3. Design principles and Security Properties of MICKEY 2.0

Figure 7.3: Clocking the $R$ register with $CONTROL\_BIT\_R = 1$

ation but equivalent to making the register jump by clocking it $J$ times, is due to Cees Jansen (see [78] for details). In [78], Jansen presents the technique applied to LFSRs with Fibonacci-style clocking, but it is clear that the same approach is valid with Galois-style clocking.

7.3.2 Motivation for the variable clocking

Stream ciphers making use of variable clocking often lend themselves to statistical attacks, in which the attacker guesses how many times the register has been clocked at a particular time. There are a number of characteristics of a cipher design that may make such attacks possible. But for MICKEY 2.0 the designers took care of them. The principles behind the design of MICKEY 2.0 are as follows:

- to take all of the benefits of variable clocking, in protecting against many forms of attack;
- to guarantee period and local randomness;
- subject to those, to reduce the susceptibility to statistical attacks as far as possible

More details can be obtained in the eSTREAM portal at [118] or in the article [13].

In MICKEY 2.0, the register $R$ acts as the engine, ensuring that the state of the generator does not repeat within the generation of a single keystream sequence, and ensuring good local statistical properties. The influence of $R$ on the clocking of $S$ also prevents $S$ from becoming stuck in a short cycle. If the jump index $J < 260$, then the state of $R$ will not repeat during the generation of a maximum length (240-bit) keystream sequence; and if $J > 240$, then some efficient attack could be avoided. The designers chose the jump index $J$ as to have the largest possible value subject to $J < 250$. 
7.3.3 Selection of clock control bits

The designers deliberately chose the clock control bits for each register to be derived from both registers, in such a way that knowledge of either register state is not sufficient to tell the attacker how either register will subsequently be clocked. This helps to guard against guess and determine or divide and conquer attacks.

7.3.4 The S register feedback function

For any fixed value of \texttt{CONTROL.BIT.S}, the clocking function of S is invertible (so that the space of possible register values is not reduced by clocking S). The designer’s goal for the clocking function of S can be stated as follows. Assume that the initial state of S is randomly selected, and that the sequence of values of \texttt{CONTROL.BIT.S} applied to the clocking of S are also randomly selected. Then consider the sequence \((s_{0}(i) : i = 0, 1, 2, \ldots)\). (\(s_{0}(i)\) actually implies the contents of \(s_{0}\) after the generator has been clocked \(i\) times.) We want to avoid any strong affine relations in that sequence – that is, there should not be a set \(I\) such that the value \(p = \sum_{i \in I} s_{0}(i)\) is especially likely to be equal to 0 (or to 1) as the initial state and \texttt{CONTROL.BIT.S} range over all possible values.

The reason for this design goal is to avoid attacks based on establishing a probabilistic linear model (\textit{i.e.} a set \(I\) as described above) that would allow a linear combination of keystream bits to be strongly correlated to a combination of bits only from the (\textit{“linear”, “weaker”}) \(R\).
register. The designer’s thought was here especially of distinguishing attacks.

Although it was pretty nontrivial to meet the design goal, the designers claimed some good reason to convince the reader that they have met it. At least, earlier proposals they considered for $S$ were weaker in this regard. They modelled a number of constructions on a scaled down version of $S$, and looked for the strongest linear relations holding over relatively short sequences $(s_0(i))$, and they found that the construction they had chosen performed well. In particular, their construction preserves local randomness, in the sense that, if the initial state is uniformly random, then a sequence of 100 successive bits $s_0(i)$ will also be uniformly random. So no sum of fewer than 101 successive bits $s_0(i)$ will be equal to 0 with probability distinct from $\frac{1}{2}$. From their empirical analysis, they claimed to believe that the strongest bias would come from a combination selected from precisely 101 successive bits $s_0(i)$.

![Figure 7.5: Clocking the $S$ register](image)

7.3.5 Algebraic Attacks

Algebraic attacks usually become possible when the keystream is correlated to one or more linearly clocking registers, whose clocking is either entirely predictable or can be guessed.

The designers had taken care that the attacker cannot eliminate the uncertainty about the clocking of either register by guessing a small set of values. (By illustrative contrast, some attacks on LILI-128 (see [115] for details) were possible because the state of the 39-stage register could be guessed, and then the clocking of the 89-stage register became known.) Furthermore, each keystream bit produced by MICKEY 2.0 is not correlated to the contents of either one register (so in particular not to the “linear register” $R$).
7.3.6 State Entropy

The generator is subject to variable clocking under control of bits from within the generator. This results in a reduction of the entropy of the overall generator state: some generator states after clocking have two or more possible preimages, and some states have no possible preimages. This is discussed further in section 7.4.3.

The fact that the control bit for each register is derived by XORing bits from both registers, and hence is uncorrelated to the state of the register it controls, is a crucial feature of the design: it means that clocking the overall generator does not reduce the entropy of either one register state.

7.3.7 Output function

MICKEY 2.0 uses a very simple output function \((r_0 \oplus s_0)\) to compute keystream bits from the register states.

Also, the designers considered more complex alternatives, e.g. of the form \(r_0 \oplus g(r_1, \ldots, r_{99}) \oplus s_0 \oplus h(s_1, \ldots s_{99})\) for some Boolean functions \(g\) and \(h\). Although these might increase the security margin against some types of attack, they preferred to keep the output function simple and elegant, and rely instead on the mutual irregular clocking of the registers.

7.4 Changes from MICKEY version 1

In MICKEY version 1, the \(R\) and \(S\) registers were each 80 stages long (instead of 100). The overall state size was thus 160 bits, for an algorithm supporting an 80-bit secret key. MICKEY version 1 was, deliberately, a minimalist algorithm with very little “padding” to bolster its security margin.

The best cryptanalytic efforts against MICKEY version 1 are by Jin Hong and Woo-Hwan Kim (see [74] for more details). They considered three areas of (arguable) vulnerability. The revisions in MICKEY 2.0 had been precisely targeted at addressing the issues raised in [74].

7.4.1 What are the changes from MICKEY version 1?

The changes are very simple: the two registers have each been increased from 80 stages to 100 stages. Some detailed values, such as control bit tap locations, have been scaled accordingly.
7.4.2 Time-Memory-Data (TMD) tradeoff, with or without BSW sampling

Let \( N \) be the size of the keystream generator state space (so \( 2^{160} \) for MICKEY version 1). Let \( X \) be the set of all possible keystream generator states. Let \( f : X \to Y \) be the function that maps a generator state to the first \( \log_2(N) \) bits of keystream produced. Suppose the attacker has harvested a large number of \( \log_2(N) \)-bit keystream sequences \( y_i \in Y \), and wants to identify a keystream generator state \( x \in X \) such that \( f(x) = y_i \) for some \( i \).

**BSW tradeoff**

The Biryukov-Shamir TMD (see [25] for more details) algorithm succeeds with high probability if the following conditions are satisfied:

\[
TM^2 D^2 = N^2 \quad \text{and} \quad 1 \leq D^2 \leq T
\]

(7.1)

where \( T \) is the online time complexity, \( M \) is the memory requirement, and \( D \) is the number of keystream sequences available to the attacker. The offline time complexity is \( P = N/D \).

**BSW sampling**

When we say that we can perform BSW sampling (more details available in [26]) with a sampling factor \( W \), we mean that:

- there is a subset \( X' \subset X \) with cardinality \( N/W \), and it is easy to generate elements of \( X' \).
- if \( Y' \) is the image of \( X' \) under \( f \), then it is easy to recognize elements of \( Y' \).

The attacker may consider only those keystream sequences that are elements of \( Y' \), and apply the BS tradeoff to the problem of inverting the restricted function \( f' : X' \to Y' \). If the total number of keystream sequences available to the attacker is \( D \), only roughly \( D/W \) of these will fall in \( y' \) and so be usable; on the other hand, the size of the set of preimages is now \( N/W \) instead of \( N \). The conditions for success become

\[
TM^2 \left( \frac{D}{W} \right)^2 = \left( \frac{N}{W} \right)^2 \quad \text{and} \quad 1 \leq \left( \frac{D}{W} \right)^2 \leq T
\]

(7.2)

i.e.

\[
TM^2 D^2 = N^2 \quad \text{and} \quad W^2 \leq D^2 \leq TW^2
\]

and the offline time complexity remains \( P = \frac{(N/W)}{D/W} = \frac{N}{D} \). Also, very importantly, the number of table lookups in the online attack is reduced by a factor \( W \), which greatly reduces the actual time it takes.
TMD tradeoff against MICKEY version 1

In [74], Hong and Kim showed that BSW sampling can be performed on MICKEY version 1 with a sampling factor $W = 2^{27}$. This allows a TMD tradeoff attack to be performed with the following complexity, for instance:

- unfiltered data complexity $D = 2^{60}$, e.g. $2^{20}$ keystream sequences each of length roughly $2^{40}$ bits; filtering these by BSW sampling means that the attack is performed against a reduced set of $D/W = 2^{33}$ keystream sequences;
- search space of reduced size $N/W = 2^{133}$;
- time complexity $T = 2^{66}$;
- memory complexity $M = 2^{67}$;
- offline time complexity $P = 2^{100}$.

So we have an attack whose online time, data and memory complexities are all less than the key size of $2^{80}$. However, the one-off precomputation time complexity is greater than $2^{80}$. Other parameter values are possible, but the precomputation time is always greater than $2^{80}$.

There is no consensus as to whether this constitutes a successful attack. Some authors seem to ignore precomputation time completely, and consider only online complexity to matter; others would say that an attack requiring overall complexity greater than exhaustive search is of no practical significance. Although, the designers’ inclination was more towards the second view, they recognized that some will deem the cipher less than fully secure if such attacks exist.

MICKEY 2.0

In MICKEY 2.0, the state size $N = 2^{200}$. Thus, for any BS tradeoff attack, with or without BSW sampling, if $TM^2D^2 = N^2$ then at least one of $T, M$ or $D$ must be at least $2^{80}$. So no attack is possible with online complexity faster than exhaustive key search.

Earlier researches recommended that the state size of a keystream generator should be at least twice the key size, to protect against what is now usually called the Babbage-Golic TMD attack. By making the state size at least $2.5$ times the key size, robust protection against the Biryukov-Shamir TMD attack was also provided, with or without BSW sampling.
BSW sampling of MICKEY 2.0

It is still possible to perform BSW sampling on MICKEY 2.0. No attempt has been made by the designers to prevent this.

7.4.3 State entropy loss and keystream convergence

The variable clocking mechanism in MICKEY means that the state entropy reduces as the generator is clocked. This is fundamental to the MICKEY design philosophy. For MICKEY version 1, in [74] Hong and Kim showed that this entropy loss can result in the convergence of distinct keystream sequences within the parameters of legitimate use of the cipher. For example, if $V$ keystream sequences of length $2^{40}$ are generated from different $(K, IV)$ pairs, then for large enough $V$ there will be state collisions – and of course, once identical states are reached, subsequent keystream sequences are identical. An exact analysis seems difficult, but it appears that $V$ may not have to be much larger than $2^{22}$ before collisions will begin to occur.

This uncomfortable property holds because, after the generator has been run for long enough to produce a $2^{40}$-bit sequence, the state entropy will have reduced by nearly 40 bits, from the initial $2^{160}$ to only just over $2^{120}$. Because 120 is less than twice the key size, the designers began to see collisions within an amount of data less than the key size.

In MICKEY 2.0, the state size is 200 bits, and the maximum permitted length of a single keystream sequence is $2^{40}$ bits. After the generator has been run for long enough to produce a $2^{40}$-bit sequence, the entropy will still be just over 160 bits. This is twice the key size, and so the problem persists no longer.

7.4.4 Weak keys

There was an obvious “lock-up” state for the register $R$: if the key and IV loading and initialization leaves $R$ in the all zero state, then it will remain permanently in that state. For MICKEY version 1 the designers reasoned as follows:

“It is clear that, if an attacker assumes that this is the case, she can readily confirm her assumption and deduce the remainder of the generator state by analysing a short sequence of keystream. But, because this can be assumed to occur with probability roughly $2^{-80}$ – much less than the probability for any guessed secret key to be correct – we do not think it necessary to prevent it (and so in the interests of efficiency we do not do so)”.

In [74], Hong and Kim pointed out that there is also a lock-up state for the register $S$. If the key and IV loading and initialization leaves $S$ in this particular state, then it will remain permanently in that state, irrespective of the values of the clock control bits. The probability
of a "weak state" in MICKEY version 1 is thus roughly $2^{-79}$ which is greater than $2^{-80}$.

It is undoubtedly much easier to try two candidate secret keys, with a success probability of $2^{-79}$, than to mount an attack based on these possible weak states. So the designers argued that it is not necessary to guard against their occurrence. But anyway, with MICKEY 2.0 the increased register lengths mean that the probability of a weak state goes down to roughly $2^{-99}$, which is clearly too small to be taken into account.

### 7.5 Performance of MICKEY 2.0

MICKEY 2.0 is not designed for notably high speeds in software, although it is straightforward to implement it reasonably efficiently. The designers' own reasonably efficient implementation generated $10^8$ bits of keystream in 3.81 seconds, using a PC with a 3.4 GHz Pentium 4 processor.

Also the designers stated that, there might be scope for more efficient software implementations that produce several bits of keystream at a time, making use of look-up tables to implement the register clocking and keystream derivation.

### 7.6 Hardware Implementations of MICKEY

In this section we describe a few efficient hardware implementation apart from designers'.

#### 7.6.1 On the Hardware Implementation of the MICKEY-128 Stream Cipher

In [84], Paris Kistos gave some idea on the Hardware Implementation of the MICKEY-128. In this paper the implementation on hardware of MICKEY-128 is investigated. MICKEY-128 with a 128-bit key is aimed at area-restricted hardware environments where a key size of 128 bits is required. An efficient hardware implementation of the cipher was presented in this paper.

MICKEY-128 has two major advantages: (i) the low hardware complexity, which results in small area and (ii) the high level of security. FPGA device was used for the performance demonstration. Some of the first results of implementing the stream cipher on an FPGA were reported. A maximum throughput equal to 170 Mbps can be achieved, with a clock frequency of 170 MHz.
7.6.2 On the Parallelization of the MICKEY-128 2.0

In [117], Stefan and Mitchell, using a novel mathematical interpretation of the algorithm, presented a method of parallelizing the stream cipher to produce an n-bit keystream output. They demonstrated a high-throughput (560 Mbps), area-efficient (392 slices) two-way parallelized implementation on the Xilinx Virtex-II Pro FPGA.

7.6.3 Implementation of parallel and reconfigurable MICKEY algorithm

In [102], Li Miao, Xu Jinfu, Dai Zibin, Yang Xiaohui and Qu Hongfei proposed a parallel and dynamic reconfigurable hardware architecture of MICKEY algorithm, which can satisfy the different characteristics of MICKEY-80, MICKEY-128 and MICKEY-128 2.0 algorithms. The three algorithms are exactly the same in design principle, so according to different reconfigurable parameters, they can be implemented in one chip. As to different parallel methods, detailed comparison and analysis are performed. The design has been realized using Altera’s FPGA. Synthesis, placement and routing of parallel and reconfigurable design have accomplished on 0.18 µm CMOS process. The result proves the maximum throughput can achieve 1915.8 Mbps.

7.6.4 Comparison of FPGA-Targeted Hardware Implementations

In [77], discussed FPGA hardware implementations of all eSTREAM phase 3 hardware stream cipher candidates (profile 2) and some of their derivatives. The designs are optimized for maximum throughput per unit area as well as minimum area, and targeted for Xilinx Spartan 3 FPGAs. The results have found that the Grain and Trivium families of ciphers have demonstrated relative implementation efficiency compared to the rest of the cipher candidates; Mickey also provided a balance of low area with high throughput per area.

7.6.5 FPGA Implementations of eSTREAM Phase-2 candidates

In [33], Philippe Bulens and Kassem Kalach evaluated the hardware performance of these algorithms in the reconfigurable hardware Xilinx Virtex-II devices. Based on their implementations (that mainly confirm previous results), they discussed the respective interest of the focused candidates and suggest certain guidelines for their comparison.
7.6.6 Hardware results for selected stream cipher candidates

In [67], T. Good and M. Benaissa presented hardware implementation and performance metrics for the candidate stream ciphers in the Phase II Hardware Focus. Quantitative consideration is also given to all candidate ciphers as to whether any should be added to the Hardware Focus set. In this treatment, only the submissions without licensing restrictions have been considered. The results are presented in tabular and graphical format together with some recommendations aimed at simplifying the implementation task for future engineers and a priority order for cryptanalysis, solely from a hardware perspective, is presented.

7.6.7 Review of stream ciphers candidates from a low resource hardware perspective

In [66], T. Good, W. Chelton and M. Benaissa presented hardware implementation and analysis of a carefully selected sub-set of the candidate stream ciphers submitted to the eSTREAM project. Only the submissions without licensing restrictions have been considered. The sub-set of six was defined based on memory requirements versus the Advanced Encryption Standard and any published security analysis. A number of complete low resource designs for each of the candidates are presented together with FPGA results for both Xilinx Spartan II and Altera Cyclone FPGAs, ASIC results in terms of throughput, area and power are also included. The results are presented in tabular and graphical format. The graphs are further annotated with different cost functions in terms of throughput and area to simplify the identification of the lowest resource designs. Based on these results, the short-listed six ciphers are classified.

7.7 Cryptanalysis of MICKEY 2.0

There are a few cryptanalysis attempts on MICKEY 2.0. Most of the attack targeted MICKEY version-1. In section 7.4, the correction made to that version is described. In this section we describe the attacks briefly.

7.7.1 TMD-Tradeoff and State Entropy Loss Considerations of MICKEY

In [74], Kim and Hong gave three weaknesses of MICKEY. A small class of weak keys were found and also they showed that, the time/memory/data tradeoff is applicable. They also showed that, the state update function reduces entropy of the internal state as it is iterated., resulting in keystreams that start out differently but become merged together towards the end.

In section 7.4, we have already discussed how these problems in MICKEY version 1 has
been fixed by the designers in MICKEY 2.0

7.7.2 Power Analysis Attacks on a Hardware Implementation of MICKEY

In [126], Hongxu Zhao and Shiqi Li first implemented a communication interface between the ASIC and a PC which is used to interact with the ASIC and facilitates to collect the power traces for further analysis. Afterwards, side-channel attack has been used to reveal the complete secret key. The most effort has been put into applying several Differential Power Analysis techniques on the implementation of the MICKEY-128 algorithm. Additionally, the comparison among these different methods was discussed.

7.7.3 Nonsmooth Cryptanalysis, with an Application to MICKEY

In [119], Elmar Tischhauser presented a new approach to the cryptanalysis of symmetric algorithms based on non-smooth optimization. They developed this technique as a novel way of dealing with nonlinearity over $\mathbb{F}_2$ by modeling the equations corresponding to the algorithm as a continuous optimization problem that avoids terms of higher degree. The resulting problems are not continuously differentiable, but can be approached with techniques from non-smooth analysis. Applied to the stream cipher MICKEY, which is part of the eSTREAM final portfolio, this method can solve instances corresponding to the full cipher, although with time complexity greater than brute force. Finally, they compared this approach to classical pseudo-Boolean programming.

7.7.4 Correlation Power Analysis Against Stream Cipher MICKEY v2

In [90], Liu, Gu, Guo and Zheng discussed correlation power analysis attack against stream cipher MICKEY v2. In such attacks, they used Hamming-Distance model to simulate the power consumption. Hamming-Distance model is a more accurate description to power consumption than other models such as Hamming-Weight, bit model etc. Generally, Hamming-Distance model is used to map the transitions that occur at the cells’ outputs of a CMOS circuit to the values of power consumption. In this attack, they proposed the Hamming-Distance model based on internal nodes of $\text{XOR}$ gates considering that the basic structure of MICKEY v2 is a two-input and a three-input $\text{XOR}$ gate. They simulate the power which is coming from not only the output of gate but also the internal nodes. Then they designed the attack way to MICKEY v2 by this model. And finally they simulated the result of attacking. The result shows that it needs only few or ten power traces during initialization to reveal the secret key by using weakness of MICKEY v2 initialization when resynchronization.
7.8 Conclusion

In conclusion, it can be said that, although MICKEY version-1 has suffered lot of threats, those were fixed in MICKEY 2.0 and there is no significant threat against the version 2.0 till now. So, while considering hardware implementation, MICKEY 2.0 could be a really good choice.

7.9 A Simple C Implementation of MICKEY 2.0

Here we provide a simple C implementation of MICKEY 2.0 This is provided to make the reader familiar to the implementation. For more sophisticated implementations the reader must look into the eSTREAM portal at [58].

```c
/*************************************************
Developer: Pratyay Mukherjee
email: pratyay85@gmail.com
***************************************************/

#include <stdio.h>

typedef unsigned char uchar;

@Bean
Constants as defined by the algorithm are converted into masks that can be
directly XOR-ed when required*/
uchar rtaps[13] = {0xde, 0x4c, 0x9e, 0x48, 0x6, 0x66, 0x2a, 0xad, 0xf1, 0x81, 0xe1,
  0xfb, 0xc0};
uchar comp0[13] = {0xc, 0x5e, 0x95, 0x56, 0x90, 0x15, 0x42, 0xe1, 0x57, 0xfd, 0x7e,
  0xa0, 0x60};
uchar comp1[13] = {0x59, 0x79, 0x46, 0xbb, 0xc6, 0xb8, 0x45, 0xc7, 0xeb, 0xbc,
  0x43, 0x89, 0x80};
uchar fb0[13] = {0xf5, 0xfe, 0x5f, 0xf9, 0x81, 0xc9, 0x52, 0xf5, 0x40, 0x1a, 0x37,
  0x39, 0x80};
uchar fb1[13] = {0x59, 0x79, 0x46, 0xbb, 0xc6, 0xb8, 0x45, 0xc7, 0xeb, 0xbc,
  0x43, 0x89, 0x80};
uchar fb1[13] = {0xf5, 0xfe, 0x5f, 0xf9, 0x81, 0xc9, 0x52, 0xf5, 0x40, 0x1a, 0x37,
  0x39, 0x80};
uchar fb1[13] = {0x59, 0x79, 0x46, 0xbb, 0xc6, 0xb8, 0x45, 0xc7, 0xeb, 0xbc,
  0x43, 0x89, 0x80};

@Bean
/*Specified key and IV values along with respective output keystream values
Key = 12 34 56 78 9a bc de f0 12 34
IV = 21 43 65 87
Keystream = 98 21 e1 0c 5e d2 8d 32 bb c3 d1 fb 15 e9 3a 15
*/
```
Key = f1 1a 56 27 ce 43 b6 1f 89 12
IV = 9c 53 2f 8a c3 ea 4b 2e a0 f5
Keystream = 21 a0 43 66 19 cb 9f 3f 6f 1f b3 03 f5 6a 09 a9

Key = 3b 80 fc 8c 47 5f c2 70 fa 26
IV =
Keystream = 6b 67 68 6f 57 0e 87 5f fb 25 92 af 90 24 1b 1c

/*The three sets of input are provided below. Un-comment one and comment out the
rest to test
with that particular test set*/

/*Test set #1*/
uchar Key[10] = {0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0, 0x12, 0x34};
uchar IV[4] = {0x21, 0x43, 0x65, 0x87};
uchar IVlength = 4;

/*Test set #2*/
/*uchar Key[10] = {0xf1, 0x1a, 0x56, 0x27, 0xce, 0x43, 0xb6,0x1f, 0x89, 0x12};
uchar IV[10] = {0x9c, 0x53, 0x2f, 0x8a, 0xc3, 0xea, 0x4b, 0x2e, 0xa0, 0xf5};
uchar IVlength = 10;*/

/*Test set #3*/
/*uchar Key[10] = {0x3b, 0x80, 0xfc, 0x8c, 0x47, 0x5f, 0xc2, 0x70, 0xfa, 0x26};
uchar IV[0] = {};
uchar IVlength = 0;*/

uchar KeyStream[16];

struct mickey_registers
{
    uchar R[13];
    uchar S[13];
};
typedef struct mickey_registers mickey;

/*Clocking the register R*/
void clock_r(mickey *m, uchar Input_Bit_R, uchar Control_Bit_R)
{

uchar Feedback_Bit, i;
uchar carry_bits[13];

Feedback_Bit = ((m->R[12] & 16)>>4) ^ Input_Bit_R;

/*Carry bits are required to perform right shift across successive variables of the character array*/
carry_bits[0] = 0;
for (i=0 ; i<12 ; i++)
    carry_bits[i+1] = (m->R[i] & 1)<<7;

if (Control_Bit_R)
    for (i=0 ; i<13 ; i++)
        m->R[i] ^= (m->R[i]>>1) ^ carry_bits[i];
else
    for (i=0 ; i<13 ; i++)
        m->R[i] = (m->R[i]>>1) ^ carry_bits[i];

if (Feedback_Bit)
    for (i=0 ; i<13 ; i++)
        m->R[i] ^= rtaps[i];
}

/*Clocking the register S*/
void clock_s(mickey *m, uchar Input_Bit_S, uchar Control_Bit_S)
{
    uchar Feedback_Bit, i;
    uchar carry_bits_right[13];
    uchar carry_bits_left[13];
    uchar temp;

    Feedback_Bit = ((m->S[12] & 16)>>4) ^ Input_Bit_S;

    /*Carry bits are required to perform right and left shifts across successive variables of the character array*/
carry_bits_right[0] = 0;
for (i=0 ; i<12; i++)
    carry_bits_right[i+1] = (m->S[i] & 1)<<7;
carry_bits_left[12] = 0;
for (i=1 ; i<13 ; i++)
    carry_bits_left[i-1] = (m->S[i] & 128)>>7;
/*Generating "S hat"*/
    temp = (m->S[12]>>1) & 0x10;
    for (i=0 ; i<13 ; i++)
        m->S[i] = ((m->S[i]>>1) ^ carry_bits_right[i]) ^ ((m->S[i] ^ comp0[i]) & (m->S[i]<<1 ^ carry_bits_left[i] ^ comp1[i]));
    /* 0th bit of S-hat = 0 */
    m->S[0] &= 0x7f;
    /* 99th bit of S-hat = 98th bit of S */
    m->S[12] &= 0xef;
    m->S[12] ^= temp;

    if (Feedback_Bit)
        {
            if (Control_Bit_S)
            {
                for (i=0 ; i<13 ; i++)
                    m->S[i] ^= fb1[i];
            }
            else
            {
                for (i=0 ; i<13 ; i++)
                    m->S[i] ^= fb0[i];
            }
        }
    /*Clocking the overall generator*/
    void clock_kg(mickey *m, uchar Mixing, uchar Input_Bit)
    {
        uchar Control_Bit_R, Control_Bit_S, Input_Bit_R, Input_Bit_S;

        Control_Bit_R = ((m->S[4] & 32)>>5) ^ ((m->R[8] & 16)>>4);
        Control_Bit_S = ((m->S[8] & 16)>>4) ^ ((m->R[4] & 64)>>6);

        if (Mixing)
            Input_Bit_R = Input_Bit ^ ((m->S[6] & 32)>>5);
        else
            Input_Bit_R = Input_Bit;

        Input_Bit_S = Input_Bit;
clock_r(m, Input_Bit_R, Control_Bit_R);
clock_s(m, Input_Bit_S, Control_Bit_S);
}

/*Prints the output keystream*/
void print_keystream()
{
    int i;
    printf("\nKey Stream\n");
    for (i=0 ; i<16 ; i++)
        printf("%x ",KeyStream[i]);
    printf("\n");
}

void main()
{
    mickey m;
    uchar i, j, Input_Bit;

    /*Initialise*/
    for (i=0 ; i<13 ; i++)
    {
        m.S[i] = 0;
        m.R[i] = 0;
    }

    /*Load IV*/
    int counter = 0;
    for (i=0 ; i<IVlength ; i++)
    {
        for (j=0 ; j<8 ; j++)
        {
            Input_Bit = (IV[i]>>(7-j)) & 1;
            clock_kg(&m, 1, Input_Bit);
            counter++;
        }
    }

    /*Load Key*/
    for (i=0 ; i<10 ; i++)
    {
        for (j=0 ; j<8 ; j++)
        {

Input_Bit = (Key[i]>>(7-j)) & 1;
clock_kg(&m, 1, Input_Bit);

} 
/*Preclock*/
for (i=0 ; i<100 ; i++)
clock_kg(&m, 1, 0);

/*Generate Key Stream*/
for (i=0 ; i<16 ; i++)
{
  KeyStream[i] = 0;
  for (j=0 ; j<8 ; j++)
  {
    KeyStream[i] ^= ((m.R[0] ^ m.S[0]) & 128)>>j;
    clock_kg(&m, 0, 0);
  }
}

/*Print KeyStream*/
print_keystream();

}
Bibliography


[58] eSTREAM. the ecrypt stream cipher project. http://www.ecrypt.eu.org/stream/.


