Analysis of the LTE Security Algorithm ZUC with SAT Solver

Ibrahim Awel Ahmed

Submitted to the
Institute of Graduate Studies and Research
in partial fulfillment of the requirements for the Degree of

Master of Science
in
Computer Engineering

Eastern Mediterranean University
February 2015
Gazimağusa, North Cyprus

Approval of the Institute of Graduate Studies and Research

Prof. Dr. Serhan Çiftçioğlu
Acting Director

I certify that this thesis satisfies the requirements as a thesis for the degree of Master of Science in Computer Engineering.

Prof. Dr. Işık Aybay
Chair, Department of Computer Engineering

We certify that we have read this thesis and that in our opinion it is fully adequate in scope and quality as a thesis for the degree of Master of Science in Computer Engineering.

Assoc. Prof. Dr. Alexander Chefranov
Supervisor

Examining Committee

1. Assoc. Prof. Dr. Alexander Chefranov
2. Asst. Prof. Dr. Gürcü Öz
3. Asst. Prof. Dr. Önsen Toygar

ABSTRACT

Long Term Evolution (LTE) the 4th generation (4G) mobile broadband radio network is designed with special attention given to security. In order to secure the communication over the air radio network of LTE, three confidentiality and integrity cryptographic algorithms are approved by 3rd Generation Partnership Project (3GPP). ZUC, which is one of the algorithms is the third alternative to LTE. ZUC is designed using inherited properties from SNOW3G cryptographic algorithm with some improvements. However, it has been found that related keys, which are result of weak key state exists in its predecessor. Moreover, from the view point of Security Algorithms Group of Experts (SAGE) advancement in cryptanalysis may have effect on both ZUC and its predecessor, due to their design similarity, and SNOW3G weak key property. This thesis analyzed the latest version of ZUC; the analysis is to check the existence of weak key state in ZUC at the end of initialization of Linear Feedback Shift Register (LFSR) with key and initialization vector (IV). The analysis is done by Boolean satisfiability problem solver (SAT solver) program, emerging logical cryptographic algorithms analysis technique. For the analysis the key initialization procedure equations of the algorithm are converted to SAT instance, which is special input format for SAT solvers in Conjunctive Normal Form (CNF), and are fed to SAT Solver. The result showed that the latest version of ZUC LFSR after the initialization is not initialized with same value that indicates there is no weak key state problem in the key generation procedure of ZUC algorithm.

Keywords: ZUC, weak key, Satisfiability, SAT solver, Logical cryptanalysis

ÖZ

Dördüncü nesil mobil geniş bant radyo ağı olan Uzun Vadeli Dönüşüm (LTE), özellikle güvenlik konusu dikkate alınarak tasarlanmıştır. LTE'nin hava radyo ağı üzerinden iletişim sağlamak üzere üç şifreleme algoritması 3. Nesil Ortaklık Projesi (3GPP) tarafından onaylanmıştır. Bu algoritmalardan biri olan ZUC, LTE'nin üçüncü alternatifidir. ZUC, SNOW3G şifreleme algoritmasından bazı özellikleri alarak, ilgili özelliklerin geliştirilmesiyle tasarlanmıştır. Ancak, ilgili anahtarlarla ilgili olarak, öncülü olan SNOW3G'de zayıf anahtar durumunun mevcut olduğu görülmüştür. Buna ek olarak, Güvenlik Algoritmaları Grubu Uzmanları'nın (SAGE) bakış açısına göre kripto analizi alanındaki iyileştirmelerin ZUC ve öncülü üzerinde etkili olabilir. Bunun nedeni ise tasarım benzerlikleri ve SNOW3G zayıf anahtar özellikleridir. Bu çalışma, ZUC'un son sürümünü analiz etmeyi amaçlamıştır. Analiz, anahtar ve başlatma vektörü (IV) ile doğrusal geribesleme öteleme kaydının (LFSR) başlatılmasının sonunda, zayıf anahtar durumunu denetlemek amacıyla yapılmıştır. Şifreleme algoritmalarının analizinde yükselmekte olan bir yöntem olarak Boolean sağlanabilirlik problem çözücü (SAT çözücü) programı kullanılmıştır. Algoritmanın anahtar başlatma prosedürü denklemleri, Bağlaçlı Normal Biçim'de (CNF) SAT çözücüleri için özel bir girdi biçimi olan SAT örneklerine dönüştürülmüştür ve ardından SAT Çözücüsü'ne yerleştirilmiştir. Sonuçlara göre, ZUC LFSR'nin son sürümünün aynı değerlerle başlatılmadığı ortaya çıkmış ve ZUC algoritmasında zayıf anahtar durumunun mevcut olmadığı anlaşılmıştır.

Anahtar Sözcükler: ZUC, zayıf anahtar, Sağlanabilirlik, SAT çözücü, Mantıksal kripto analizi.

Dedicated to My Family

ACKNOWLEDGMENT

With the supervision of Assoc. Prof. Dr. Alexander Chefranov in my work, getting valuable advice, follow up, corrections, suggestion, and improvements, I learn a lot from this thesis. It is a pleasure for me to say thank you so much to my supervisor Assoc. Prof. Dr. Alexander Chefranov.

My thankfulness extended to professors at EMU who shared me their knowledge. In addition, I am indebted to express my thankfulness for my families whom behind my studies with their support, encouragement and helped me to advance in knowledge.

TABLE OF CONTENTS

ABSTRACT iii

ÖZ iv

ACKNOWLEDGMENT vi

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF SYMBOLS OR LIST OF ABBREVIATIONS xi

1 INTRODUCTION 1

  1.1 Problem Definition 2

  1.2 Methodology of Analysis 4

  1.3 Organization of the thesis 4

2 BOOLEAN SATISFIABILITY (SAT) PROBLEMS 6

  2.1 Boolean Satisfiability problems 6

  2.2 SAT Solvers 9

    2.2.1 Conjunctive Normal Form 9

    2.2.2 DIMACS Format 10

3 ZUC ALGORITHM STRUCTURE 14

  3.1 Structure of ZUC 14

    3.1.1 LFSR 15

    3.1.2 Bit Reorganization (BR): 15

    3.1.3 Non-linear function (*F*) 16

  3.2 Execution 17

    3.2.1 Initialization stage 17

    3.2.2 Working stage 18

3.2.3 Keystream Generation 18
3.2.4 Encryption/Decryption 18
4 IMPLEMENTATION AND EXPERIMENTAL RESULTS 20
4.1 Building CNF 20
4.2.1 Exclusive OR CNF 21
4.2.2 Modular addition 2^m 22
4.2.3 Modular addition 2^m CNF 23
4.2.4 S-Box CNF 24
4.2.5 Rotation – K bit cyclic shift CNF 26
4.3 Generating ZUC SAT instance CNF 26
4.4 Experiment conducted 31
4.4.1 User manual to the program 31
4.4.2 Experment results analysis 35
5 CONCLUSION 38
REFERENCES 39
APPENDICES 42
Appendix A: Test parameters 43
Appendix B: Test cases variables assignment 46
Appendix C: Sample XOR CNF Generator Code 47
Appendix D: Sample S-Box S0 CNF Generator Code 49

LIST OF TABLES

Table 1: Possible values for 10 binary variables.. 8

Table 2: Truth table Boolean function 21

Table 3: Truth table Exclusive OR.. 22

Table 4: Truth table Boolean addition of two 1-bit operands with carry bits 23

Table 5: Partial view truth table ZUC S-Boxes S0 / S1 input and output................. 25

Table 6: Partial view of S-box 0 first output column CNF 25

Table 7: Mapping operand to SAT variable.. 27

Table 8: Partial view of Exclusive OR CNF generation 28

Table 9: Partial views of generated DIMACS file 29

Table 10: Mapping s-box inputs to SAT variable.. 30

Table 11: Partial view of generated DIMACS of S-Box 0 30

Table 12: Different Key and IV for test scenarios 36

Table 13: Experimental result key 1 and IV 1 36

Table 14: Experimental result key 2 and IV 2 37

Table 15: Experimental result key 3 and IV 3 37

Table 16: Experimental result key 4 and IV 4 37

Table 17: Experimental result key 5 and IV 5 37

Table A1: Test 1 parameters………………………………………………………...43

Table A2: Test 2 parameters……………………………………………………......43

Table A3: Test 3 parameters………………………………………………………...44

Table A4: Test 4 parameters………………………………………………………...45

LIST OF FIGURES

Figure 1: Satisfiable input for MinSat SAT Solver [9] 11

Figure 2: SATISFIABLE output of MinSat SAT Solver [9] 12

Figure 3: UNSATISFIABLE input and output of MinSat SAT Solver [9] 13

Figure 4: Structure of ZUC Algorithm [6] 14

Figure 5: Boolean addition of two 1-bit operands with carry bits [15]...................... 23

Figure 6: Encryption/decryption test case 1……………….…………………….................32

Figure 7: Saved DIMACS files………..……………….…………………….................33

Figure 8: Input to online SAT Solver [9]………………….……………….…..............34

Figure 9: Output of online SAT Solver [9]……………….……………………............. 35

LIST OF SYMBOLS OR LIST OF ABBREVIATIONS

AES	Advanced Encryption Standard
BR	Bit-reorganization
CNF	Conjunctive Normal Form
DIMACS	Center for Discrete Mathematics and Theoretical Computer Science
EPS	Evolved Packet System - LTE and SAE
EEA	EPS Encryption Algorithm
EIA	EPS Integrity Algorithm
128-EEA3	128-bit EEA third algorithm
FSM	Finite State Machine
GF (q)	Galois field with q elements
3G	3rd Generations of mobile communications technology
4G	4th Generations of mobile communications technology
3GPP	3rd Generation Partnership Project
IV	Initialization Vector
LFSR	Linear Feedback Shift Register
LTE	Long Term Evolution (radio network)
SAGE	Security Algorithms Group of Experts
SATISFIABILITY or SAT	Boolean Satisfiability Problem
SNOW3G	Stream cipher for the 3GPP encryption algorithms
QoS	Quality of Service
ZUC	Stream cipher algorithm (named after Chinese Mathematician and astronomer Zu Chongzhi)

$+$	The addition of two integers.
$=$	The assignment operator
mod	The modulo operation of integers
$\oplus$	The bit-wise exclusive-OR operation of integers
$\boxplus$	The modulo 2^{32} addition
a||b	The concatenation of strings a and b
aH	The leftmost 16 bits of integer a
aL	The rightmost 16 bits of integer a
$a<<<_n k$	The k-bit cyclic shift of the n bit register ato the left
a>> 1	The l-bit right shift of integer a
$(a_1, a_2, \ldots, a_n) \rightarrow (b_1, b_2, \ldots, b_n)$	The assignment of the values of a_i to b_i in parallel
$\rightarrow$	Implication
$\leftrightarrow$	Double implication
$\neg$	Negation / NOT
$\overline{x}$	Negation of x
$\vee$	OR
$\wedge$	AND
$\lceil x \rceil$	Upper bound of x

Chapter 1

INTRODUCTION

Long Term Evolution (LTE) the 4th Generation (4G) mobile broadband radio network technology designed by the 3rd Generation Partnership Project (3GPP), telecommunication standard and development body [1], to fulfill the growing demands for huge broadband throughput, definite availability, wide coverage of mobility, and variety of Quality of Service (QoS) levels, with special attention given to security. Security is a critical issue for both 3G and 4G mobile networks, [2] and in particular strict security as one of the design objectives for LTE [3]; therefore, 3GPP specifications recommend encrypted communication over the air interface of LTE mobile network. Moreover, three confidentiality and data integrity cryptographic algorithms are approved by 3GPP [4].

The approved confidentiality and data integrity cryptographic algorithms are SNOW3G, Advanced Encryption Standard (AES), and ZUC. SNOW3G have been working for the 3G. The big expectation of strict security added AES as second alternative. In 2010, new algorithm ZUC proposed as a third addition to LTE [4] and the 2011 version of ZUC added to LTE Advanced [4, 2].

ZUC is stream cipher, which takes 128-bit keys to encrypt/decrypt blocks of data in the range of 1 to 2^{32} bits [5, 6]. The encryption/decryption is adding key stream bits generated by ZUC into plain message or cipher message. ZUC algorithm involves three logical layers first layer is Linear Feedback Shift Register (LFSR) that contain 16 stages shift registers, and second layer bit reorganization (BR) layer, and third layer nonlinear transformation layer (F) [6]. Each of the layers of ZUC discussed further in Chapter 3.

ZUC algorithm has design similarity to SNOW3G. SNOW3G cipher consists of a combination of a LFSR and a Finite State Machine (FSM) where the LFSR also feeds the next state function of the FSM. During evaluation, weaknesses were discovered in SNOW 1.0 and the authors have developed a new version, version 2.0 of the cipher that solves the weaknesses and improves the performance. During SAGE evaluation, the design was further modified to increase its resistance against algebraic attacks with the result named SNOW3G. It has been found that related keys exist both for SNOW 2.0 and SNOW3G [7].

1.1 Problem Definition

ZUC algorithm design requirement was to make it different from its predecessor algorithms, so that an attack on either is unlikely on the others too. However, ZUC architectural design is not fully different from the previous 3GPP approved algorithm i.e. SNOW3G. Therefore, according to SAGE the design requirements not fully fulfilled, because ZUC inherits properties from SNOW3G. Nonetheless, SAGE considered that it was not only inherited some features, but also added strength too, and hence accepted the design. However, SAGE had doubt that both algorithms might be affected with advancement in cryptanalysis [4].

Unfortunately, the 2010 public evaluation found weakness, then again newer version in 2011 released making amendment to flaws identified, followed by six-month public evaluation which ended with no problem reported. Finally, SAGE approved the 2011 version to 3GPP as confidentiality and integrity protection for LTE Advanced mobile network [4].

ZUC considered resistant from different types of cryptanalytic attacks [4] among them an attack related key used, which referred as weak key. Generally, weak key is a key that leads to extraction of the key bits used for the encryption/decryption from the input bits and output bits relationship of a cryptographic algorithm [8]. In context to ZUC, keys considered weak if the Key and initialization Vector (IV) used for the initialization of the LFSR initializes all the cells of the LFSR with same value at the end of initialization, which is called all-p state. The pair that leads to such state is called weak (key IV) pair and the Key is called weak key [4].

The strength of ZUC comes mainly from its nonlinear process during the key steam generation that makes an attack complex and difficult. Though considering SAT solver's efficacy in solving complex problems [8] and SAGE's initial idea that advancement in cryptanalysis may affect ZUC and its predecessor as premise [4], the objective of this thesis is to analyze ZUC with SAT solver, analyzing security algorithms with different mechanisms exposes hidden weakness there by gives direction for either amendment or replacement. SAT solvers alone cannot break modern cryptographic algorithms; however, they are useful at enhancing cryptanalysis, and their combination to other cryptanalytic techniques seems promising [8]. The analysis mechanism of ZUC algorithm is specifically related to key loading and initialization procedure to find, or prove the absence of, weak key state on ZUC using SAT solver state of the are cryptanalysis method emerging.

1.2 Methodology of Analysis

SAT solvers can help to analyze cryptographic algorithms key generation process mechanism to verify its strength and weakness [8]. Therefore, the analysis is done with SAT Solver program. For this thesis MiniSat SAT Solver available online at [9] is used. SAT solver cryptanalysis requires representation of the algorithm or parts of it into format that SAT solvers understand called SAT instance. SAT instance is Conjunctive Normal Form (CNF) representation of the cryptographic algorithm equations that involve ARX algorithms (i.e. operations based on NOT, XOR, AND, OR, as well as addition modulo 2^n and left rotation) [8]. In addition to ARX operators, CNF of S-box also required for algorithms that involve S-box like ZUC.

The next stage is generating SAT instances while the algorithm is running, and stores the CNF to file. Then feed to SAT solver and if the formed CNF equation is satisfiable indicates possibility of weak key, else, weak key is not exit [8].

1.3 Organization of the thesis

In Chapter 1, we have covered introduction, problem definition and methodology of analysis. The subsequent chapters are as follows:

Chapter 2 discuses Boolean satisfiability problem, the foundations of SAT solvers.

Chapter 3, discuses ZUC algorithm general structure and its operation

Chapter 4 discusses the implementation of ZUC algorithm followed by SAT instance generation from the source code, and then SAT solver analysis on the generated SAT instances.

Chapter 5 concludes the thesis.

Finally, at the end of appendices section experimental test parameters, test cases sample codes, and instance generator sample codes are included.

Chapter 2

BOOLEAN SATISFIABILITY (SAT) PROBLEMS

This Chapter discussed SAT problems, followed by SAT problem solvers referred as SAT Solvers. Then CNF, the standard input for most SAT solvers discussed. The CNF input representation for SAT Solvers called Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) format and its syntax discussed. Finally, practical examples are supplied.

2.1 Boolean Satisfiability problems

Boolean satisfiability problems are mathematical logical problems that try to find possible solution that makes a given Boolean function result true. In other word we are interested in getting values that satisfy Boolean equation as in equation (2.1) shown bellow, so that the result becomes 1 i.e. TRUE. A Boolean expression result becomes TRUE is referred as SATISFIABLE, meaning there are possible solution that fulfil the given expression. Otherwise, it is called UNSATISFIABLE i.e. the expression is always 0 indicates FALSE, i.e. there is no possible input for the problem that make the result of it becomes TRUE [10, 13].

$$f = \bigwedge_{i=1}^{n} Ci, \qquad where \quad C = \bigvee_{i=1}^{m} Li, \quad L \in \{0,1\} \tag{2.1}$$

L (Literal) a propositional variable or its negation,

and C (Clause) is literal or disjunction of literals.

For example, let’s consider simple Boolean formula *f* which is conjunction between two Boolean variables *a*, and *b*.

$$f = (a \wedge \neg b)$$

f is *SATISFIABLE* i.e. ($f = 1 = (a \wedge \neg b)$), because there are possible values that the result $f = (a \wedge \neg b) = TRUE$, when $a = TRUE$, and $b = FALSE$ satisfy the formula. In contrast, $f = (a \wedge \neg a)$, *f* is *UNSATISFIABLE* i.e. ($f = (a \wedge \neg a) = FALSE$), since there is no possible value that makes the result *f TRUE*, in both cases when $a = TRUE$ and a = FALSE, $f = (a \wedge \neg a)$ is always *FALSE*, hence *UNSATISFIABLE*.

Let’s consider another example with number of Boolean variables increasing to 5 variables *a1,a2,a3,a4,a5* having the following clauses of the five Boolean variables:

$$(\neg a2 \vee a5)$$

$$(a4 \vee \neg a5)$$

$$(a1 \vee \neg a3 \vee a4)$$

$$(a1 \vee a2)$$

To check the satisfiability we can create collectively combining each of them as a Boolean formula as follows:

$$f = (\neg a2 \vee a5) \wedge (a4 \vee \neg a5) \wedge (a1 \vee \neg a3 \vee a4) \wedge (a1 \vee a2)$$

Now the objective is to find possible Boolean values that satisfy the formula, so that *f* becomes TRUE (1).

$$1 = (\neg a2 \vee a5) \wedge (a1 \vee \neg a3 \vee a4) \wedge (a4 \vee \neg a5) \wedge (a1 \vee a2)$$

Possible values that satisfy the given equation substituted in each variable a_i of the Boolean equation are:

$$a1=1 \qquad a2=0 \qquad a3=1 \qquad a4=1 \qquad a5=1$$

$$f = (1 \vee 1) \wedge (1 \vee 0 \vee 1) \wedge (1 \vee 0) \wedge (1 \vee 0)$$

SAT problems looks easy just as we are trying the possible values to get the result, but the problem arises when the number of Boolean variable increase, and becomes complex. Also SAT problems are NP-Complete. NP-Complete problems get harder and harder as the problem becomes larger and will be difficult to solve. So we can generalize it as follows for n variables at worst case we have 2^n exhaustive search to get the possible values that make the give Boolean equation TRUE [10].

For example for 10 variables, it needs 2^{10} =1024 exhaustive search with possible configuration values shown in Table 1.

Table 1: Possible values for 10 binary variables

2^{10} Possible values	
1	0000000000
2	0000000001
3	0000000010
4	0000000011
.	.
.	.
1024	111111111

Similarly increasing the variables to 1000 then we need to have:

2^{1000} =1.07150860718626732094842504906e+301 Possible values.

Therefore, an efficient alternative required to solve these kind complex Boolean equations. Satisfiability solvers in short SAT Solvers, deal in such problems and decides whether the give equation is satisfiable or not [11]. SAT solvers and how they work is explained in the subsequent sections.

2.2 SAT Solvers

SAT Solvers are programs that determine whether Boolean expression has solution or not, using mathematical methods. SAT solver program tries to find all possible values for the variables that can make the given expression as a whole true, and gives the possible satisfiable values. During the search, if there are no values that make the expression true it returns unsatisfiable . To find satisfying values for a given clause of an expression SAT solver apply depth-first search algorithm based on backtracking the details of how it work can be found in [12]. Mostly SAT Solvers input is Boolean expressions in CNF with special input format called DIMACS format [11]. CNF and DIMACS are explained in the next sections.

2.2.1 Conjunctive Normal Form

CNF Terminologies: CNF is formed from the following three basic blockes: term, clause, and expression [11].

Term ⇨ A term is a Boolean variable like ($a2$) or the negation of it like ($\neg a2$**).** Term also called literal.

Clause ⇨ A clause is the disjunction of terms, terms joined by OR. Clause can contain single terms, and clause may not have repetition of Bboolean variables.

Expression ⇨ An expression is the conjunction of clauses, clauses joined by AND.

Definition: CNF is expression formed by conjunction of clauses, clauses connected by AND as in (2.1), and each clause composed of disjunction of terms. As the symbols indicate the disjunctions of the terms (L_i) forms the clause C and the conjunction of clauses(C_i) gives the final result of *f in CNF*.

Example: This expression ($a \vee b \vee c$) ($\neg a \vee b \vee \neg c$) is in CNF since the conjunction implicitly expressed by parenthesis. To satisfy the expression "*SATISFIABLE*"

(TRUE) results of every clause must be TRUE. Since it is conjunction of clauses any one of the clauses is FALSE, the expression as a whole is "*UNSATISFIABLE*" (FALSE) [11], which is the basics behind that SAT solvers use to find the possible solutions.

2.2.2 DIMACS Format

The input format for SAT Solvers is the referred as DIMACS file format which is normal text file. As SAT Solvers requirement the CNF of a given Boolean expression is represented in DIMACS file. In DIMACS the variables are not directly written rather each variable is assigned mapping decimal number as representation of the variable and negation of it expressed by "-" and the two operators of CNF disjunction and conjunction are represented by space and 0 respectively. The syntax for the DIMACS in CNF is as follows [11]:

➔ ***p cnf <No_OF_VARIABLES> < No_OF_CLAUSE>***

- ⇨ First line must start with character "p" followed by space then "cnf" again space the number of variables space at the end number of clauses.
- ⇨ Character "c" to make comment
- ⇨ In this format variables of the equation under consideration are given natural number representation, and integers are assigned for literals. For example, the variable *a*2 represented by 2 and –*a*2 by -2.
- ⇨ The DIMACS in CNF operator disjunction represented by character space
- ⇨ Number zero (0) indicates ends a clause and represents the conjunction.

The next examples show the DIMACS File format for Boolean equation.

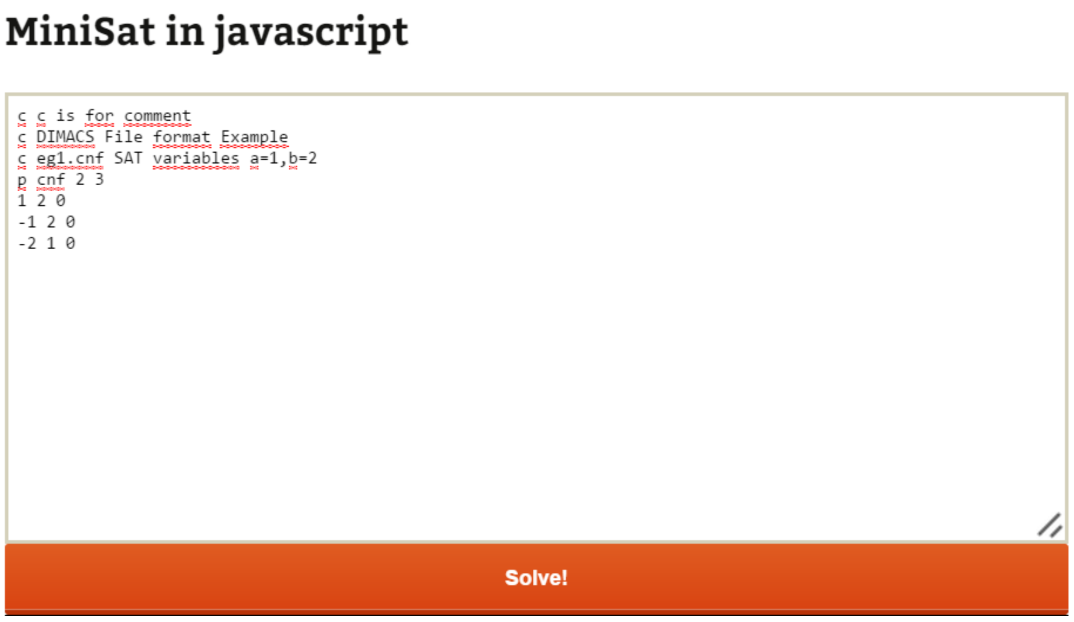
SATISFIABLE Example

```
f =(a | b) & (-a | b) & (-b | a)
```

Each clause independently re-written as follows, it contains two variables a, and b and three clauses:

```
(a | b) &
(-a | b) &
(-b | a)
```

Convert the variables into the mapped SAT variables *a*=1 and *b*=2 and feed to SAT solver as shown in the Figure 1.



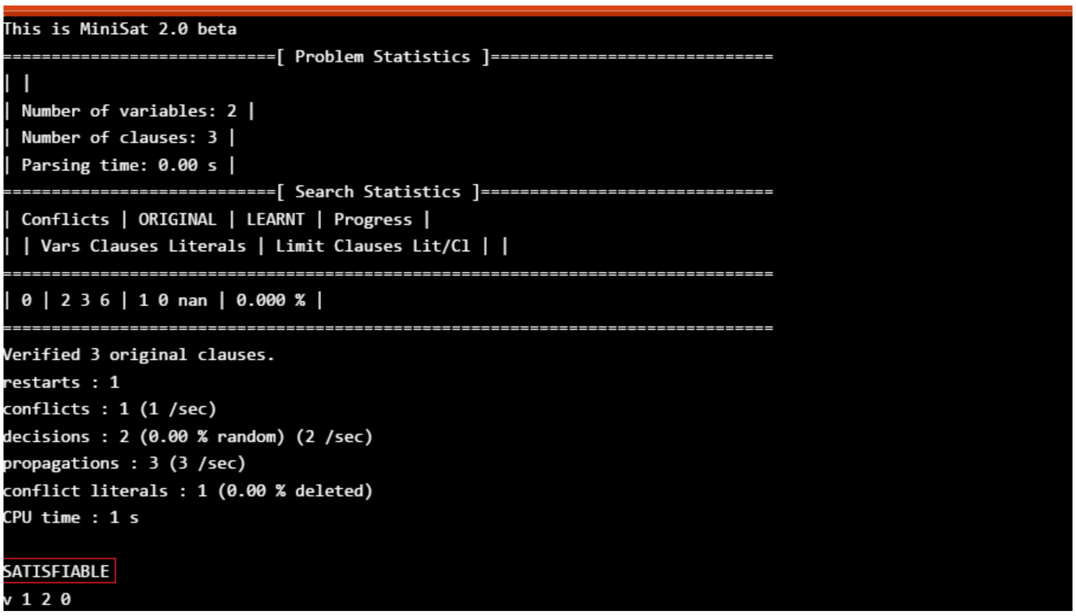
```
c c is for comment
c DIMACS File format Example
c eg1.cnf SAT variables a=1,b=2
p cnf 2 3
1 2 0
-1 2 0
-2 1 0
```

Figure 1: Satisfiable input for MinSat SAT Solver [9]

As shown in Firure 2. from the execution of the expression *f* possible values that make the it SATISFIABLE are *a*=1, and *b*=2. Positive result implies true; *a*=true, and *b*=true.

f =(a | b) & (-a | b) & (-b | a)

true = (true | true) & (-true | true) & (-true | true)



```
This is MiniSat 2.0 beta
==========================[ Problem Statistics ]============================
| |
| Number of variables: 2 |
| Number of clauses: 3 |
| Parsing time: 0.00 s |
==========================[ Search Statistics ]=============================
| Conflicts | ORIGINAL | LEARNT | Progress |
| | Vars Clauses Literals | Limit Clauses Lit/Cl | |
===============================================================================
| 0 | 2 3 6 | 1 0 nan | 0.000 % |
===============================================================================
Verified 3 original clauses.
restarts : 1
conflicts : 1 (1 /sec)
decisions : 2 (0.00 % random) (2 /sec)
propagations : 3 (3 /sec)
conflict literals : 1 (0.00 % deleted)
CPU time : 1 s

SATISFIABLE
v 1 2 0
```

Figure 2: SATISFIABLE output of MinSat SAT Solver [9]

UNSATISFIABLE Example

f =[-*b* & (-*a* | -*c*) & (*a* | *b*) & (*b* |*c*)]

Each clause independently re-written as follows, it contains three variables *a*, *b*, and *c* and four clauses:

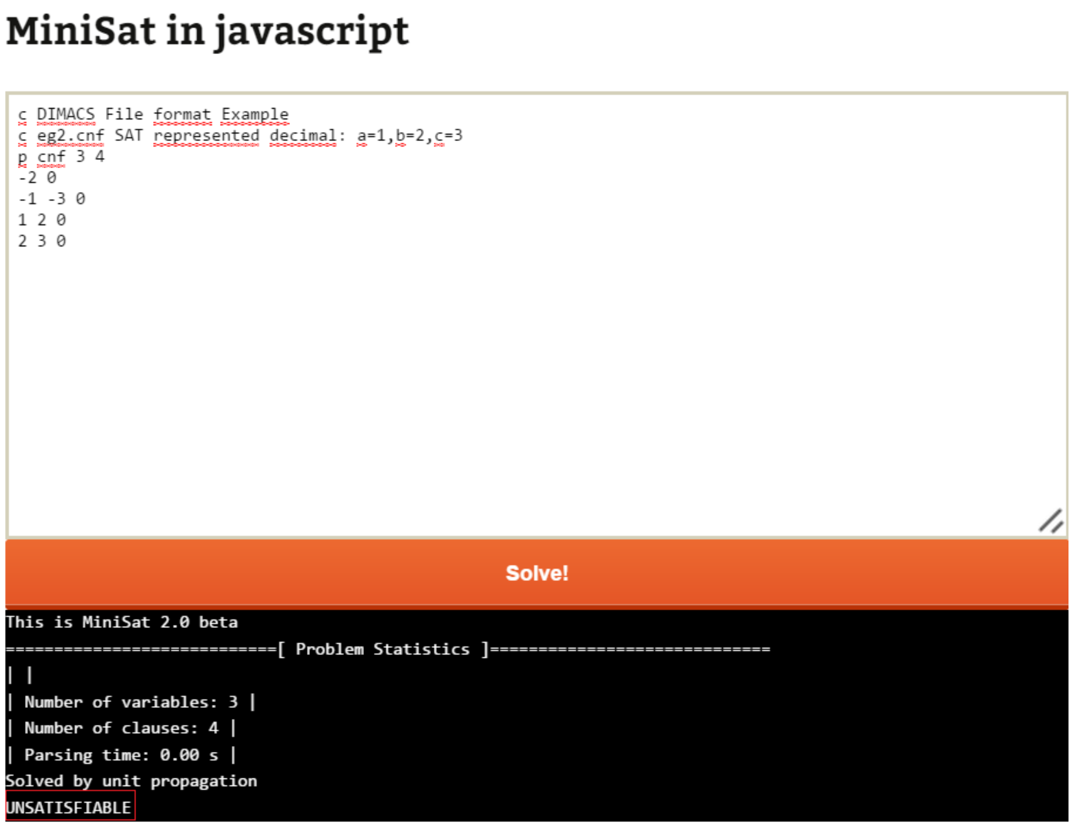
-*b* &

(-*a* | -*c*) &

(*a* | *b*) &

(*b* |*c*)

Convert the variables into mapped SAT variables *a*=1, *b*=2, and *c*=3. Then feed to SAT solver as shown in the Figure 3.


MiniSat in javascript

```
c DIMACS File format Example
c eg2.cnf SAT represented decimal: a=1,b=2,c=3
p cnf 3 4
-2 0
-1 -3 0
1 2 0
2 3 0
```

```
This is MiniSat 2.0 beta
============================[ Problem Statistics ]=============================
| |
| Number of variables: 3 |
| Number of clauses: 4 |
| Parsing time: 0.00 s |
Solved by unit propagation
UNSATISFIABLE
```

Figure 3: UNSATISFIABLE input and output of MinSat SAT Solver [9]

From the execution of the expression *f* There is no possible value that will make the given expression *f* true, hence UNSATISFIABLE.

Chapter 3

ZUC ALGORITHM STRUCTURE

In this Chapter, the general structure and execution of ZUC algorithm in briefly presented for further details refer [4, 5, 6, 13], official 3GPP specifications. The new stream cipher ZUC is a world–oriented stream cipher, takes a 128-bit secret key and a 128-bit IV as input, and outputs keystreams of 32-bit words, which are used to encryption/decryption [5].

3.1 Structure of ZUC

As shown in the Figure 4. The general structure of ZUC algorithm involves three logical layers: LFSR, BR, and F [6]. Each logical layer of the algorithm briefly discussed bellow.

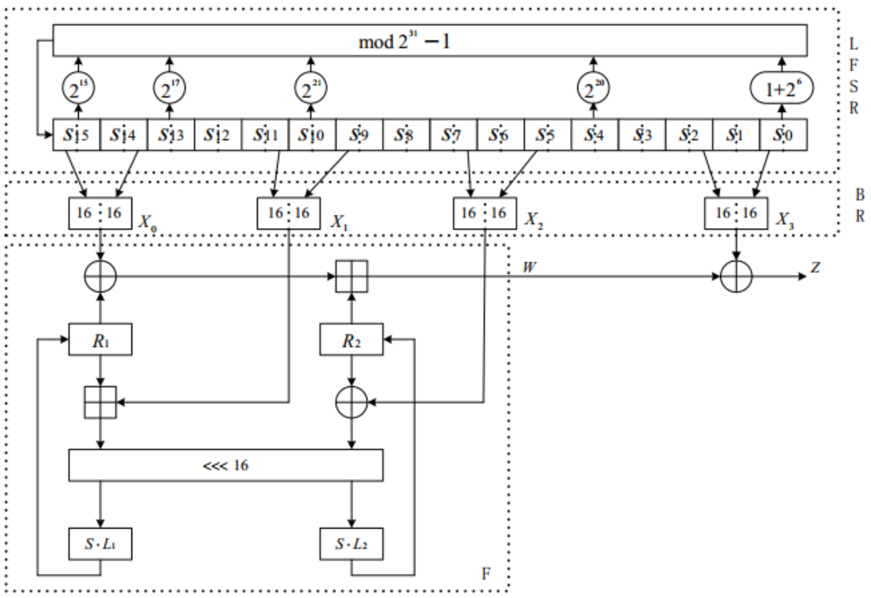


Figure 4: Structure of ZUC Algorithm [6]

3.1.1 LFSR

LFSR has 16 cells each having 31-bits denoted by (s_0, s_1,…, s_{14},s_{15}) each can take values 1.2.3…,2^{31}-1. LFSR involves two modes of operation initialization and working modes.

1) Initialization mode

In this mode for the initialization of LFSR values, (s_0, s_1,…, s_{14},s_{15}), it takes a 31-bit input from output of *F* discarding 1 right most bit *u*=*W*>>1, details shown below:

```
LFSRWithInitialisationMode (u) {
        $v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0 \bmod (2^{31}-1)$        (3.1)
        $s_{15} = u + v \bmod (2^{31}-1)$        (3.2)
        if $s_{16} = 0$ then set $s_{16}$ $2^{31}-1$
        $(s_1, s_2,\dots, s_{15},s_{16}) \rightarrow (s_0, s_1,\dots, s_{14},s_{15})$
}
```

2) Working mode

In this mode the initialization of LFSR values (s_0, s_1,…, s_{14},s_{15}) it takes no input details shown below:

```
LFSRWithWorkMode () {
        $v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + (1+2^8)s_0 \bmod (2^{31}-1)$
        if $s_{16} = 0$ then set $s_{16}$ $2^{31}-1$
        $(s_1, s_2,\dots, s_{15},s_{16}) \rightarrow (s_0, s_1,\dots, s_{14},s_{15})$
}
```

3.1.2 Bit Reorganization (BR):

The BR cells denoted by *X0, X1, X2* and *X3* in the second logical layer. Each of the BR cells takes 32 bit from LFSR Cells Left half (H) or Right Half (L). Then *X0, X1, X2* will be input for F (nonlinear function) and X3 will be input for key stream production.

```
Bitreorganization (){
        $X0 = S15H \| S14L$
        $X1 = S11L \| S9H$
        $X2 = S7L \| S5H$
        $X3 = S2L \| S0H$
}
```

3.1.3 Non-linear function (*F*)

The non-linear function denoted by F in the logical layer has 32-bits cells *R1* and *R2*. As it mentioned above the first three words, *X0, X1, X2* formed in the *BR* stage are input to *F* and then *F* output 32-bit (*W*). The equations shown below are under the *F*.

```
F(X0, X1, X2) {
        $W = (X0 \oplus R_1) \boxplus R2$          (3.3)
        $W1 = R1 \boxplus X1$                       (3.4)
        $W2 = R2 \oplus X2$                         (3.5)
        $R1 = S(L1(W1L \| W2H))$                    (3.6)
        $R2 = S(L2(W2L \| W1H))$                    (3.7)
}
```

In the equations (3.6) and (3.7) S is S-box and L is linear transformer. Linear transformer (L) defined bellow in (3.8), and (3.9), and S-box defined in (3.10).

1) The linear transforms *L*1 and *L*2

According to the definition below, it transforms the input to *x*

$$L_1(X) = X \oplus (X <<<_{32} 2) \oplus (X <<<_{32} 10) \oplus (X <<<_{32} 18) \oplus (X <<<_{32} 24) \quad (3.8)$$

$$L_2(X) = X \oplus (X <<<_{32} 8) \oplus (X <<<_{32} 14) \oplus (X <<<_{32} 22) \oplus (X <<<_{32} 30) \quad (3.9)$$

2) S-Boxes

The S-box ***S*** The 32×32 S-box S is composed of 4 juxtaposed 8×8 S-boxes, i.e., $S=(S_0,S_1,S_2,S_3)$,where$S_0=S_2$, $S_1=S_3$ S-Boxes 0, S-Boxes defined in [6].

For example

For an 8-bit input x in hex as x=h||l, the first hex half (h) represents row and the second hex half (l) column of S_0 (or S_1).

S_0 (0x24) =0xE4 and S_1 (0x68)=0x23.

Similarly for 32-bit input x divide it into four 8-bits (4 bytes) and apply the same as above for each.

$X= x_0 \| x_1 \| x_2 \| x_3$, $Y = y_0 \| y_1 \| y_2 \| y_3$ where x_i and y_i are all bytes, i=0, 1, 2, 3. Then we have $y_i=S_i(x_i)$, i=0, 1, 2, 3

Let X=0x2468ACE8 be a 32-bit input, and Y its 32-bit output. Then divide each byte get from the output from the appropriate S-box then it will give as follows:

$Y=S(X) =S_0$ (0x24) $\|S_1$ (0x68) $\|S_2$ (0xAC) $\|S_3$ (0xE8) = 0xE423E17B.

In general for S-Boxes 0 and 1

$$Y = Si(x), i=0,1 \quad (3.10)$$

3.2 Execution

ZUC execution has two stages: the initialization and working stage.

3.2.1 Initialization stage

The initialization starts by key loading and cells R_1 and R_2 initialized to be all 0. The 16 cells of LFSR each with 31-bit is initialized by expanding the 128-bit Key and IV into 16 each having one byte, and constant value 240-bit expanded into 16 having 15-bit each then it loads to LFSR as follows:

$$k = k_0 \| k_1 \| k_2 \| \ldots \| k_{15} \text{ and}$$

$$iv = iv_0 \| iv_1 \| iv_2 \| \ldots \| iv_{15}$$

$$D= d_0 \| d_1 \| \ldots \| d_{15}, \qquad \text{Constant value}$$

$s_i = k_i || di || iv_i$ where k_i and iv_i, $0<i<15$, are all bytes.

Then it runs the following functions for 32 rounds:

1. *Bitreorganization ()*
2. $W=F(X_0, X_1, X_2)$
3. *LFSRWithInitialisationMode (w>> 1)*

3.2.2 Working stage

Following the initialization stage the working stage. First, it runs the following only once: it discards the output of *W* nonlinear function *F*:

1. *Bitreorganization ()*
2. $F(X_0, X_1, X_2)$
3. *LFSRWithInitialisationMode ()*

3.2.3 Keystream Generation

Then 32-bit key stream (***Z***) production commences the key stream generation involves the following:

1. *Bitreorganization ()*
2. $\boldsymbol{Z}= F(X_0, X_1, X_2) \oplus X_3$
3. *LFSRWithInitialisationMode* ()

For the encryption/decryption of a message with LENGTH bits, it generates

$L=\lceil LENGTH/32 \rceil$ words.

3.2.4 Encryption/Decryption

Both encryption/decryption perform by exclusive-OR message M and cipher message C by the keystream Z generated in the above step.

$M = M[0] \| M[1] \| M[2] \| \ldots \| M[LENGTH-1]$ be the input bit stream of length LENGTH and

$C = C[0] \parallel C[1] \parallel C[2] \parallel \ldots \parallel C[\text{LENGTH-1}]$ be the corresponding output bit stream of length LENGTH, where M[i] and C[i] are bits, i=0,1,2,…,LENGTH-1. Then

$C[i] = M[i] \oplus z[i]$,i=0,1,2,…,LENGTH-1.

$M[i] = C[i] \oplus z[i]$,i=0,1,2,…,LENGTH-1.

Chapter 4

IMPLEMENTATION AND EXPERIMENTAL RESULTS

First, ZUC Confidentiality algorithm written in C programming language is taken from 3GPP specification documentations [5, 6]. Then the CNF generators added to the source code without affecting the encryption decryption process. CNF generators can be prepared converting each operator in the algorithm [8]. The ZUC Confidentiality algorithm is composed of different operators exclusive OR, modular addition 2^m, and rotation – K bit cyclic shift. When the operators execute the method defined for the particular operator appends the CNF of the operator into DIMACS file. In addition to operators CNF for S-Boxes is also generates during the S-Boxes execution.

4.1 Building CNF

The conversion of Boolean equations into CNF format done based on truth table of given equation or applying De'morgans laws. General case how to generate CNF from truth table mentioned is discussed in this section, so that it can be applied for all similar cases in the implementation. To build CNF from truth table we need to consider all the possible inputs and outputs of the Boolean equation [16]. For instance, let us take an arbitrary example of a Boolean function with three variables *f (x, y, z),* variables *x* and *y* represent the two operands and *z* the result of the operations, like ($z = x \oplus y$) and the function output is either true "1" or false "0". Let us also assume the function in consideration have the truth Table 2.

Table 2: Truth table Boolean function

Row No.	*x*	*y*	*z*	*f (x, y, z)*
1	0	0	0	1
2	0	0	1	0
3	0	1	0	0
4	0	1	1	1
5	1	0	0	0
6	1	0	1	1
7	1	1	0	1
8	1	1	1	0

To generate CNF from truth table, first, check the result of the function with the given variables for each row. When the result of the function is "0" from the same row of the corresponding variables will be connected by disjunction (logical OR), if the value of the truth table is "1" the negation of the variable will be taken this makes what is known clause of the CNF. Similarly for the rest of the rows and connect each clause of the rows by conjunction (logical AND) this finally gives the CNF of the equation.

$$f = (x \lor y \lor \overline{z}) \land (x \lor \overline{y} \lor \mathrm{z}) \land (\overline{x} \lor y \lor \mathrm{z}) \land (\overline{x} \lor \overline{y} \lor \overline{z})$$

For all the equations similar approach will be applied to generate the CNF of cryptographic equations of ZUC algorithm containing cryptographic equations: (3.1), (3.2), (3.3), (3.4), (3.5), (3.6), (3.7), (3.8), (3.9), and (3.10), and having the operators exclusive OR, modular addition 2^m, S-Box, and rotation – K bit cyclic shift. The CNF of each operators is defined using truth table.

4.2.1 Exclusive OR CNF

To build the Exclusive OR CNF Table 3 is used with similar method is applied above.

Table 3: Truth table Exclusive OR

Rows No.	**X**	**Y**	**z**	**z = x ⊕ y**
1	0	0	0	1
2	0	0	1	0
3	0	1	0	0
4	0	1	1	1
5	1	0	0	0
6	1	0	1	1
7	1	1	0	1
8	1	1	1	0

CNF of $z = x \oplus y$ is:

$$(x \lor y \lor \neg z) \land (x \lor \neg y \lor z) \land (\neg x \lor y \lor z) \land (\neg x \lor \neg y \lor \neg z) \qquad (4.1)$$

4.2.2 Modular addition 2^m

Modular addition is part of many cryptographic algorithms, for instance SNOW 3G, and ZUC. Specifically two operands modular addition is the focus of this thesis as the cryptographic equations of ZUC involve fixed length two operand modular additions. Modular addition is based on Galois field GF (2^m) i.e. addition n operands having m-bit and values can only be {0, 1}. As Figure 5 (left) depicts to perform two operands modular addition it is important to have three operands (*x, y, and CarryIn*), i.e. the two operands plus the carry bit, and the expected outputs (*Sum, CarryOut*) result sum and the carry bit output. Similarly, for n bit operands as Figure 5 (right) shows it can be generalized as the addition for each bit as (x_i, y_i, *and CarryIni)* and its result outputs are *(Sumi, CarryOuti)* [15,16].

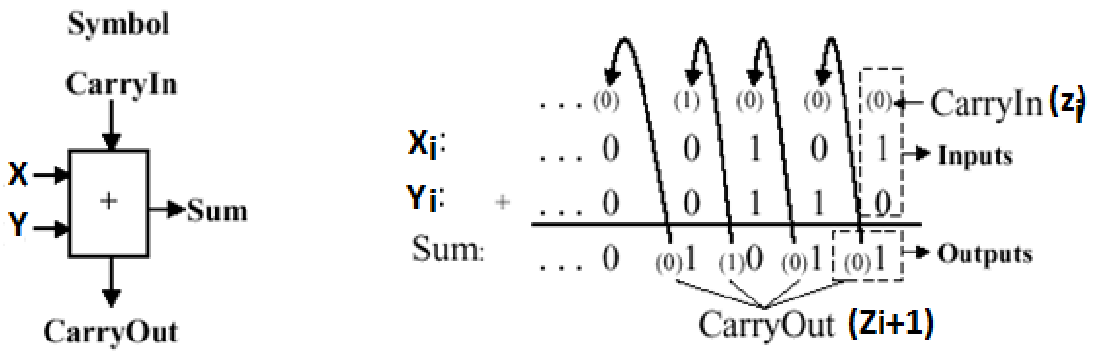


Figure 5: Boolean addition of two 1-bit operands with carry bits [15]

4.2.3 Modular addition 2^m CNF

Base on the above addition method we can generalize and represent the result for two operands all possible inputs and outputs of the modular addition as shown in Table 4.

Table 4: Truth table Boolean addition of two 1-bit operands with carry bits

Rows No.	Carry In z_i	Operand1 x_i	Operand2 y_i	Carry Out z_{i+1}	Sum s_i
1	0	0	0	0	0
2	0	0	1	0	1
3	0	1	0	0	1
4	0	1	1	1	0
5	1	0	0	0	1
6	1	0	1	1	0
7	1	1	0	1	0
8	1	1	1	1	1

From the above truth table, representation of the Modular addition in CNF is as follows:

$$s_i = (z_i \lor x_i \lor y_i) \land (z_i \lor \neg x_i \lor \neg y_i) \land (\neg z_i \lor x_i \lor \neg y_i) \land (\neg z_i \lor \neg x_i \lor y_i) \quad (4.2)$$

$$z_{i+1} = (z_i \lor x_i \lor y_i) \land (z_i \lor x_i \lor \neg y_i) \land (z_i \lor \neg x_i \lor y_i) \land (\neg z_i \lor x_i \lor y_i) \quad (4.3)$$

The final modular addition in CNF is the CNF of (4.2) ∧ (4.3).

Addition $_{2^m CNF} = [(4.2) \wedge (4.3)]$ (4.4)

4.2.4 S-Box CNF

The CNF generated based on truth table of the s-boxes Table 5 (It shows partial view of the 256 rows), which are representation of the S-Boxes S_0 and S_1. The truth tables have 256 rows that constructed from the 16x16 s-box, that are directly taken from the 3rd Generation Partnership Project (3GPP) ZUC specification document [6], and its equivalent binary that is necessary for the generation of the CNF made from the input and output of the s-boxes. The 8-bit outputs of the truth table is represented by variables **y1, y2, y3, y4, y5, y6, y7, y8** similarly 8-bit input also represented by variables **x1, x2, x3, x4, x5, x6, x7, x8**. The CNF representation for each of the eight output variables made according to truth Table 5 of the s-box CNF generated by traversing through each 256 row of the output variable. According to CNF rule from truth table, during the traverse when the output variable truth table value is "0" from the same row of the corresponding 8-bit input variables connected by disjunction (logical or) when the value is "1" it negation is taken. Finally the generated disjunction variables connected by conjunction (logical and) to the next one. Let us consider the first output (**y1**) variable CNF automatically generated shown below by (4.5), similarly for rest of output variables y2, y3, y4, y5, y6, and y7, y8 their CNF constructed and joined by conjunction to get the final **CNF of the S-box0** as shown in (4.6). Due to the big size of the CNF of S-boxes Table 6 shows partial view of S-box 0 and same method applied on the others y2, y3, y4, y5, y6, and y7, y8.

Table 5: Partial view truth table ZUC S-Boxes S0 / S1 input and output

No	S-Box S0/S1 Input		S-Box S0 Output		S-Box S1 Output	
		X1 X2 X3 X4 X5 X6 X7 X8		y1 y2 y3 y4 y5 y6 y7 y8		y1 y2 y3 y4 y5 y6 y7 y8
1	0x00	0 0 0 0 0 0 0 0	0x3e	0 0 1 1 1 1 1 0	0x55	0 1 0 1 0 1 0 1
2	0x01	0 0 0 0 0 0 0 1	0x72	0 1 1 1 0 0 1 0	0xc2	1 1 0 0 0 0 1 0
3	0x02	0 0 0 0 0 0 1 0	0x5b	0 1 0 1 1 0 1 1	0x63	0 1 1 0 0 0 1 1
…	…	…	…	…	…	…
…	…	Rows omitted	…	…	…	…
…	…	…	…	…	…	…
		Rows omitted				
…	…	…	…	…	…	…
…	…	…	…	…	…	…
255	0xfe	1 1 1 1 1 1 1 0	0x34	0 0 1 1 0 1 0 0	0xe2	1 1 1 0 0 0 1 0
256	0xff	1 1 1 1 1 1 1 1	0x60	0 1 1 0 0 0 0 0	0xf2	1 1 1 1 0 0 1 0

Table 6: Partial view of S-box 0 first output column CNF

y1 = $[(x1 \lor x2 \lor x3 \lor x4 \lor x5 \lor x6 \lor x7 \lor x8) \land$ (4.5)

$(x1 \lor x2 \lor x3 \lor x4 \lor x5 \lor x6 \lor x7 \lor \neg x8) \land$

$(x1 \lor x2 \lor x3 \lor x4 \lor x5 \lor x6 \lor \neg x7 \lor x8) \land$

...

Clauses omitted

…

$(\neg x1 \lor \neg x2 \lor \neg x3 \lor \neg x4 \lor \neg x5 \lor \neg x6 \lor \neg x7 \lor x8) \land$

$(\neg x1 \lor \neg x2 \lor \neg x3 \lor \neg x4 \lor \neg x5 \lor \neg x6 \lor \neg x7 \lor \neg x8)]$

$f_{\text{S-Boxes0 CNF}} = \bigwedge_{i=1}^{8}(yi)$ defined based on (4.5) and all outputs (4.6)

By applying similar method, we can generate CNF for S-Box 1 we finally get:

$f_{\text{S-Boxes1 CNF}} = \bigwedge_{i=1}^{8}(yi)$ (4.7)

4.2.5 Rotation – K bit cyclic shift CNF

The K bits cyclic shift also called rotation. CNF generated based on De Morgan's law implication and double implication.

$c \leftrightarrow (a<<<_{32}k)$ (4.8)

$a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$ De Morgan's law of double implication (4.9)

$(a \rightarrow b) = \neg a \vee b$ De Morgan's law of implication (4.10)

$c \rightarrow (a<<<_{32}k) \wedge (a<<<_{32}k) \rightarrow c$ applying by (4.8)

CNF Rotation: $(\neg c \vee (a<<<_{32}k)) \wedge ((a<<<_{32}k)\ c))$ applying by (4.9) (4.11)

4.3 Generating ZUC SAT instance CNF

CNF is generated for each key IV input; during execution of the program the CNF generator for each operator in the program takes operands (variables) of the operation convert them into binary. Then for bit values of 1 it sends positive and for 0 negative of the SAT variables, which are tracked with numbers [8]. For example the execution of the exclusive OR operator defined in (4.1) for a single occurrence in the algorithm it generates 32x4 clauses composed of the logical variables x_i, y_i, z_i represented interms of the SAT variable numbers. These SAT variables start from 1 and increments for every logical operator execution in a bit level. Then the CNF variables are stored into DIMACS file appending on the previous.

next Table 7 taken from the first round execution of exclusive OR operator executing equation defined by (3.1) for particular input values x,y,z given in the table. It shows the input variables in binary and the corresponding SAT variables as decimal number. In SAT variables mapping binary 1 represented with positive and 0 with negative decimal numbers.

Table 7: Mapping operand to SAT variable

Inputs	*X*=e08f9a00, *y*=0, *z*= e08f9a00
X	e08f9a00
xi binary	1 1 1 0 0 0 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0
xi SAT variable	1 2 3 -4 -5 -6 -7 -8 9 -10 -11 -12 13 14 15 16 17 -18 -19 20 21 -22 23 -24 -25 -26 -27 -28 -29 -30 -31 -32
Y	0
yi binary	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
yi SAT variable	-1 -2 -3 -4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -20 -21 -22 -23 -24 -25 -26 -27 -28 -29 -30 -31 -32
Z	e08f9a00
zi binary	1 1 1 0 0 0 0 0 1 0 0 0 1 1 1 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0
zi SAT variable	1 2 3 -4 -5 -6 -7 -8 9 -10 -11 -12 13 14 15 16 17 -18 -19 20 21 -22 23 -24 -25 -26 -27 -28 -29 -30 -31 -32

Now let us apply the SAT variables and generate SAT instance for equation (4.1) for 32 times as follows:

$f_{\oplus \text{ CNF}} = \bigwedge_{i=1}^{32} (xi \lor yi \lor \neg zi) \land (xi \lor \neg yi \lor zi) \land (\neg xi \lor yi \lor zi) \land (\neg xi \lor \neg yi \lor \neg zi)$

Applying the SAT variables, it will give as shown in Table 8. The table shows partially not all the 32x4 clauses.

Table 8: Partial view of Exclusive OR CNF generation

$(1 \vee -1 \vee \neg 1) \wedge (1 \vee \neg{-1} \vee 1) \wedge (\neg 1 \vee -1 \vee 1) \wedge (\neg 1 \vee \neg{-1} \vee \neg 1)$
$(2 \vee -2 \vee \neg 2) \wedge (2 \vee \neg{-2} \vee c_2) \wedge (\neg 2 \vee -2 \vee 2) \wedge (\neg 2 \vee \neg{-2} \vee \neg 2)$
$(3 \vee -3_1 \vee \neg 3) \wedge (3 \vee \neg{-3} \vee 3) \wedge (\neg 3 \vee -3 \vee 3) \wedge (\neg 3 \vee \neg{-3} \vee \neg 3)$
…
Clauses omitted
….

We have seen sample representation for exclusive or operator similarly, all operators defined by (4.4), (4.6), and (4.10) according to shown below:

$$f_{\boxplus\mathrm{CNF}}=\bigwedge_{i=1}^{32} (zi \vee xi \vee yi) \wedge (zi \vee \neg xi \vee \neg yi) \wedge (\neg zi \vee xi \vee \neg yi) \wedge (\neg zi \vee \neg xi \vee yi)(zi \vee xi \vee yi) \wedge (zi \vee xi \vee \neg yi) \wedge (zi \vee \neg xi \vee yi) \wedge (\neg zi \vee xi \vee yi)$$

$f_{\text{S-Boxes CNF}}$ = Defined by equations (4.6) and (4.6) for S-box 0 and S-Box 1 respectively.

$f_{<<<k\,\mathrm{CNF}} = \bigwedge_{i=1}^{32}(ci \vee \neg a(i + k \bmod 32)) \wedge (\neg ai \vee c(i + k \bmod 32))$

After traversing through the Initialization procedures of ZUC for three rounds including the last state of the initialization we will have DIMACS CNF with 3584 SAT variables and 48128 clauses generated Table 9 shows parts of the CNF instances generated.

Table 9: Partial views of generated DIMACS file

p cnf 3584 48128	c 3	c 5
c XOR Start	3 -3 -3 0	-5 -5 5 0
c 1	3 3 3 0	-5 5 -5 0
1 -1 -1 0	-3 -3 3 0	5 -5 -5 0
1 1 1 0	-3 3 -3 0	5 5 5 0
-1 -1 1 0	c 4	c 6
-1 1 -1 0	-4 -4 4 0	-6 -6 6 0
c 2	-4 4 -4 0	-6 6 -6 0
2 -2 -2 0	4 -4 -4 0	6 -6 -6 0
2 2 2 0	4 4 4 0	6 6 6 0
-2 -2 2 0	…	…
-2 2 -2 0 …		
48092	Clauses	Omitted
…	…	…
….	….	….
3582 3582 3582 0	-3583 -3583 3583 0	-3584 3584 3584 0
3582 3582 -3582 0	-3583 -3583 -3583 0	-3584 3584 -3584 0
3582 -3582 3582 0	-3583 3583 3583 0	-3584 -3584 3584 0
-3582 3582 3582 0	3583 -3583 3583 0	3584 3584 3584 0

Next S-box specifically taken out of the above-generated DIMACS file and shown below in Table 10 for specific input given, the SAT variable takes eight inputs by mapping to SAT variable, in this case from current position of SAT variable counter from 641 to 648 and generates the CNF clause shown in Table 11. Based on the rules defined for S-box 0 above in (4.5), and (4.6).

Table 10: Mapping s-box inputs to SAT variable

Inputs hex	*x*=9d
x binary	1 0 0 1 1 1 0 1
x SAT variable	641 -642 -643 644 645 646 -647 648

Table 11: Partial view of generated DIMACS of S-Box 0

```
c y1 Start
641 -642 -643 644 645 646 -647 648 0
641 -642 -643 644 645 646 -647 -648 0
641 -642 -643 644 645 646 647 648 0
….
Clauses omitted
…..
-641 642 643 -644 -645 -646 -647 -648 0
-641 642 643 -644 -645 -646 647 648 0
-641 642 643 -644 -645 -646 647 -648 0
c y1 End Clauses : 128
….
….
c y2 Start
641 -642 -643 644 645 646 -647 648 0
641 -642 -643 644 645 -646 647 648 0
641 -642 -643 644 645 -646 647 -648 0
…..
…..
-641 642 643 -644 645 -646 647 648 0
-641 642 643 -644 -645 646 647 648 0
-641 642 643 -644 -645 -646 647 648 0
c y2 End Clauses : 128
….
….
c y8 Start
641 -642 -643 644 645 646 -647 648 0
641 -642 -643 644 645 646 -647 -648 0
641 -642 -643 644 645 -646 -647 648 0
…
…
-641 642 643 -644 -645 -646 -647 -648 0
-641 642 643 -644 -645 -646 647 648 0
-641 642 643 -644 -645 -646 647 -648 0
c y8 End Clauses : 128
```

4.4 Experiment conducted

The test of availability of weak key state on ZUC algorithm is performed by generating CNF representation of the equations of the key initializations represented in DIMACS format, then executed with Online SAT solver available in [9]. Based on the weak sate definition in [8, 4] and explained above in Chapter 1 section 1.3. The CNF is generated for each round of the initialization and the final state assumed as weak key state appended to the CNF DIMACS file, and finally the SAT instance executed with SAT solver. The existence possible solution is indication of the availability of weak key, else if SAT solver returns UNSATISFIABLE indicates there is no weak key state.

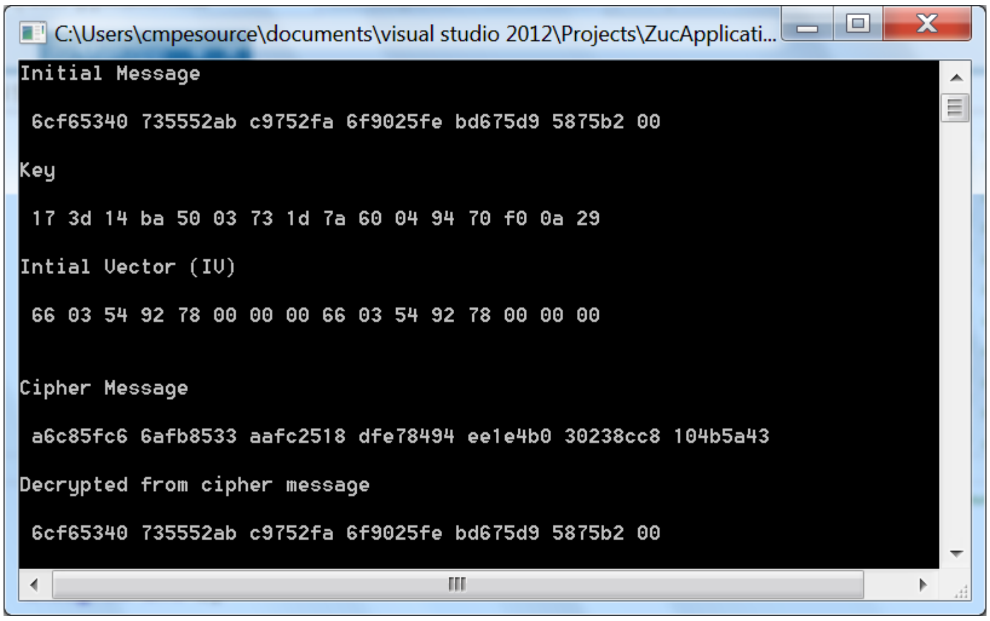
Therefore, ZUC algorithm is implemented as per the requirement in [5,6], and for the key initialization procedure the ARX operators SAT instance generator is added to the code without affecting the operation of the encryption decryption process. Then according to the above weak key definition for each ARX operators key initialization procedure with key IV SAT instance is generated considered as input, and the final status of the key IV initialization SAT instance as output of the initialization. Then generated SAT instance execute with SAT Solver. SAT solver tries to get possible solution for this relation from the given input output relation if there is possible solution it implies the key is weak; otherwise, the key is not weak.

4.4.1 User manual to the program

The first step of the analysis is generating SAT instance. The SAT instance generation process simultaneously proceeds while performing the encryption/decryption process. Therefore, all the parameters of ZUC algorithm those are necessary for the encryption/decryption need to be properly assign. The following steps summarize the process:

1. Assignment of the confidentiality keys, IV, and the other variables of ZUC algorithm like COUNT, BEARER, DIRECTION, and LENGTH. In addition, during encryption supply the plain text M, and during decryption supply the cipher message C. all values used are found in implementer's test data in [13], and partially supplied in appendix A Table A1 – Table A4. Sample assignment taken from the source code shown in Appendix B the complete view can be found in the source code from CD of the thesis.

2. Then build the solution to make the compilation from the environment of development in this case visual Studio 2012 and run the program. The program dose the encryption/decryption for the given input as shown bellow in Figure 6.

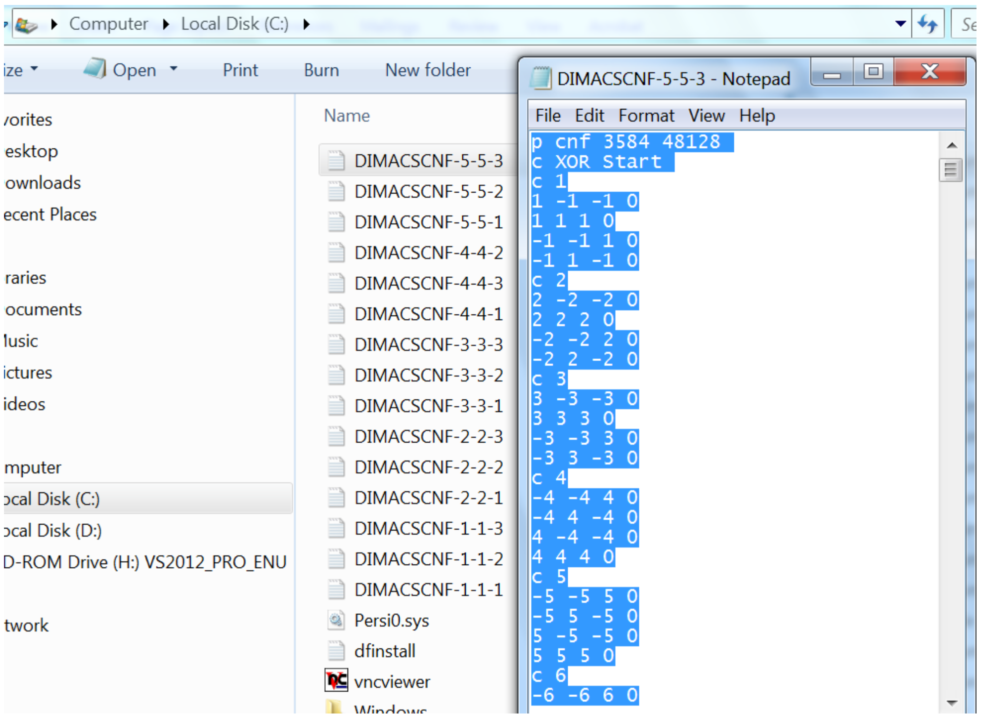

C:\Users\cmpesource\documents\visual studio 2012\Projects\ZucApplicati...

```
Initial Message
 6cf65340 735552ab c9752fa 6f9025fe bd675d9 5875b2 00
Key
 17 3d 14 ba 50 03 73 1d 7a 60 04 94 70 f0 0a 29
Intial Vector (IV)
 66 03 54 92 78 00 00 00 66 03 54 92 78 00 00 00

Cipher Message
 a6c85fc6 6afb8533 aafc2518 dfe78494 ee1e4b0 30238cc8 104b5a43
Decrypted from cipher message
 6cf65340 735552ab c9752fa 6f9025fe bd675d9 5875b2 00
```

Figure 6: Encryption/decryption test case 1

3. Simultaneously the program generates the SAT instances and save in DIMACS file in the specified location in this case c drive, with the given file name in this case DIMACS as shown in Figure 7.
4. Open the DIMACS file from the saved location copy all contains of the file that you want to test.

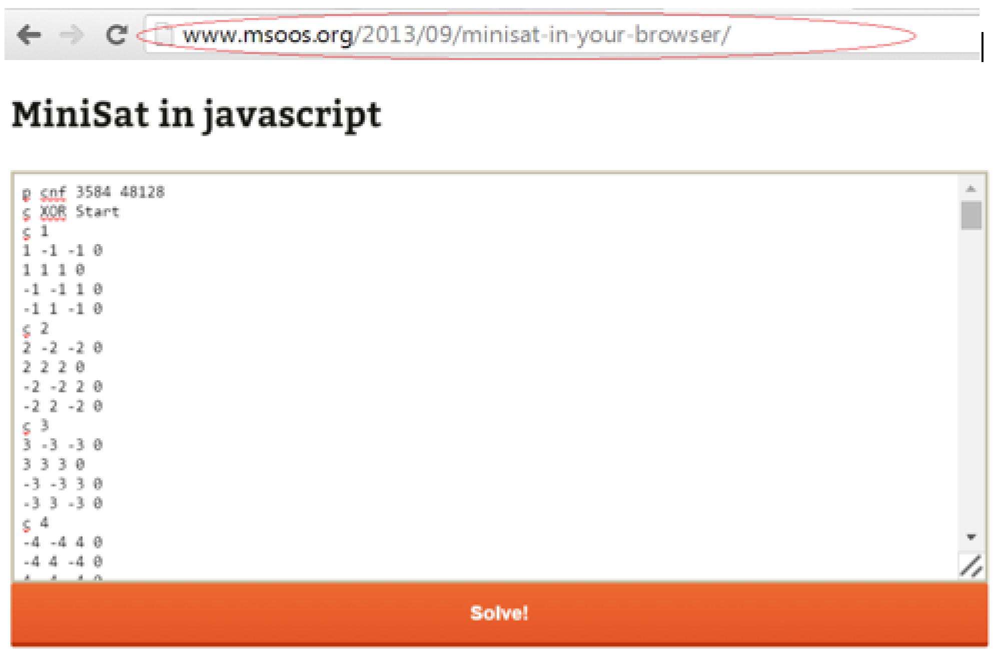


DIMACSCNF-5-5-3
DIMACSCNF-5-5-2
DIMACSCNF-5-5-1
DIMACSCNF-4-4-2
DIMACSCNF-4-4-3
DIMACSCNF-4-4-1
DIMACSCNF-3-3-3
DIMACSCNF-3-3-2
DIMACSCNF-3-3-1
DIMACSCNF-2-2-3
DIMACSCNF-2-2-2
DIMACSCNF-2-2-1
DIMACSCNF-1-1-3
DIMACSCNF-1-1-2
DIMACSCNF-1-1-1
Persi0.sys
dfinstall
vncviewer

```
p cnf 3584 48128
c XOR Start
c 1
1 -1 -1 0
1 1 1 0
-1 -1 1 0
-1 1 -1 0
c 2
2 -2 -2 0
2 2 2 0
-2 -2 2 0
-2 2 -2 0
c 3
3 -3 -3 0
3 3 3 0
-3 -3 3 0
-3 3 -3 0
c 4
-4 -4 4 0
-4 4 -4 0
4 -4 -4 0
4 4 4 0
c 5
-5 -5 5 0
-5 5 -5 0
5 -5 -5 0
5 5 5 0
c 6
-6 -6 6 0
```

Figure 7: Saved DIMACS files

5. Then open the online SAT solver as shown in Figure 8 past it to SAT solver execution window and click solve.

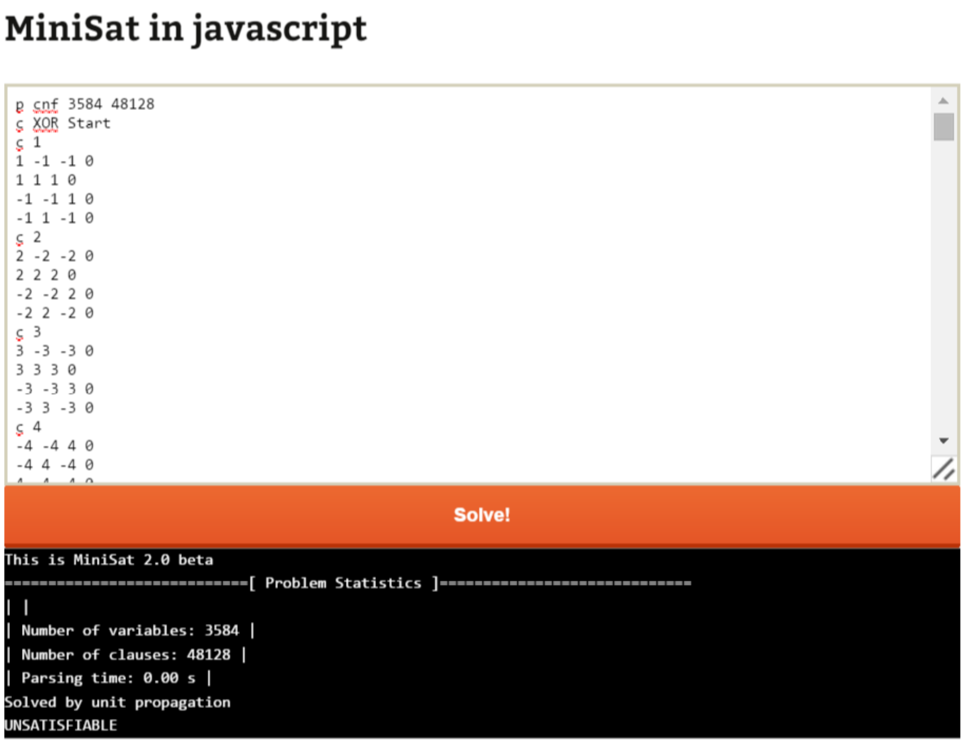

www.msoos.org/2013/09/minisat-in-your-browser/

MiniSat in javascript

```
p cnf 3584 48128
c XOR Start
c 1
1 -1 -1 0
1 1 1 0
-1 -1 1 0
-1 1 -1 0
c 2
2 -2 -2 0
2 2 2 0
-2 -2 2 0
-2 2 -2 0
c 3
3 -3 -3 0
3 3 3 0
-3 -3 3 0
-3 3 -3 0
c 4
-4 -4 4 0
-4 4 -4 0
```

Figure 8: Input to online SAT Solver [9]

6. The result of execution of the SAT Solver shall be as follows shown in Figure 9. The output of the SAT solver interpreted if SATISFIABLE it returns the possible values, else UNSATISFIABLE implies no possible value(s) that satisfy the given SAT instance input that is indication that there is no weak key state.



```
MiniSat in javascript

p cnf 3584 48128
c XOR Start
c 1
1 -1 -1 0
1 1 1 0
-1 -1 1 0
-1 1 -1 0
c 2
2 -2 -2 0
2 2 2 0
-2 -2 2 0
-2 2 -2 0
c 3
3 -3 -3 0
3 3 3 0
-3 -3 3 0
-3 3 -3 0
c 4
-4 -4 4 0
-4 4 -4 0
```

```
This is MiniSat 2.0 beta
============================[ Problem Statistics ]=============================
| |
| Number of variables: 3584 |
| Number of clauses: 48128 |
| Parsing time: 0.00 s |
Solved by unit propagation
UNSATISFIABLE
```

Figure 9: Output of online SAT Solver [9]

4.4.2 Experment results analysis

The experiment done with different key and IV combinations. The five keys shown in Table 12 taken from [13], implementers test data document. The Keys are given directly and the IV constructed from the IV building block variables. These Keyes are used during the SAT instance generation. In addition, 50 more randomly generated keys are used. For readability purpose each key is assigned labels: key1, key2, key3, key4, and key5, similarly the initialization vectors are also labeled as IV1,IV2,IV3,IV4,and IV5.

Table 12: Different Key and IV for test scenarios

Key/IV	No	Values(hex)
Key	1	Test Key 1 from [13] , Implementer's Test Data
IV	1	66 03 54 92 78 00 00 00 66 03 54 92 78 00 00 00
Key	2	Test Key 2 from [13] , Implementer's Test Data
IV	2	00 05 68 23 c4 00 00 00 00 05 68 23 c4 00 00 00
Key	3	Test Key 3 from [13] , Implementer's Test Data
IV	3	76 45 2e c1 14 00 00 00 76 45 2e c1 14 00 00 00
Key	4	Test Key 4 from [13] , Implementer's Test Data
IV	4	e4 85 0f e1 84 00 00 00 e4 85 0f e1 84 00 00 00
Key	5	Test Key 5 from [13] , Implementer's Test Data
IV	5	27 38 cd aa d0 00 00 00 27 38 cd aa d0 00 00 00

The SAT instances are generated for the initialization procedure of the ZUC algorithm with different key and IV. The SAT instances generated during the initialization procedure for one round, two rounds, and three rounds initializations. For each round the last state of the initialization appended as weak key state. Tables 13 to 17 summarizes the SAT instance generated for initialization of one round 1792 SAT variable having 24064 clauses, for two rounds of initialization 2688 SAT variable having 36096 clauses, and for three rounds of initialization 3584 SAT variable having 48128 clauses generated. Then the CNF generated stored in DIMACS file executed with SAT solver the existence of possible solution is indication of the availability of weak key. In the test conducted for all cases, SAT solver returns UNSATISFIABLE indicates there is no weak key state in ZUC algorithm.

Table 13: Experimental result key 1 and IV 1

Rounds	Key	IV	No.Variables	No.Clauses	Satisfiability
1	1	1	1792	24064	UNSATISFIABLE
2	1	1	2688	36096	UNSATISFIABLE
3	1	1	3584	48128	UNSATISFIABLE

Table 14: Experimental result key 2 and IV 2

Rounds	Key	IV	No.Variables	No.Clauses	Satisfiability
1	2	2	1792	24064	UNSATISFIABLE
2	2	2	2688	36096	UNSATISFIABLE
3	2	2	3584	48128	UNSATISFIABLE

Table 15: Experimental result key 3 and IV 3

Rounds	Key	IV	No.Variables	No.Clauses	Satisfiability
1	3	3	1792	24064	UNSATISFIABLE
2	3	3	2688	36096	UNSATISFIABLE
3	3	3	3584	48128	UNSATISFIABLE

Table 16: Experimental result key 4 and IV 4

Rounds	Key	IV	No.Variables	No.Clauses	Satisfiability
1	4	4	1792	24064	UNSATISFIABLE
2	4	4	2688	36096	UNSATISFIABLE
3	4	4	3584	48128	UNSATISFIABLE

Table 17: Experimental result key 5 and IV 5

Rounds	Key	IV	No.Variables	No.Clauses	Satisfiability
1	5	5	1792	24064	UNSATISFIABLE
2	5	5	2688	36096	UNSATISFIABLE
3	5	5	3584	48128	UNSATISFIABLE

Chapter 5

CONCLUSION

In this thesis ZUC the LTE Advanced confidentiality and integrity cryptographic algorithm recommended by SAGE to be the third alternative algorithm, in addition to SNOW3G, and AES investigated for the existence of weak key. Weak key state in general is a key that leads to extraction of the key bits used for the encryption/decryption. Specifically to ZUC, keys considered weak if the Key and initialization Vector (IV) used for the initialization of the LFSR initializes all the cells of the LFSR with same value at the end of initialization. In this analysis, existence of weak key in ZUC is checked with the help of SAT solver that is the state of the art cryptanalysis tool which is emerging. For the test SAT instance for different Key and IV combination, test parameters are provided by 3GPP ZUC implementers test data documentation and in addition fifty keys were randomly generated. The SAT instance generated for three rounds of the initialization procedure and the last round of the initialization state of LFSR were considered as weak key. Then for each scenario of the Key IV combination, SAT instance results were generated and then executed with SAT solver. The result for the entire test return UNSATISFIABLE that means LFSR is not initialized with the same values therefore weak key does not exist in the key generation procedure of ZUC algorithm. Hence, ZUC is a strong algorithm and the right choice for the high security demand of the 4G network.

REFERENCES

[1] “3GPP,” The 3rd Generation Partnership Project, [Online]. Available: http://www.3gpp.org/. [Accessed 15 January 2015].

[2] Orhanou, Ghizlane, S. El Hajji, A. Lakbabi, and Y. Bentaleb. "Analytical evaluation of the stream cipher ZUC." In Multimedia Computing and Systems (ICMCS), International Conference, pp. 927-930. IEEE, 2012

[3] Escudero-Andreu, C.P. Raphael and D.J. Parish. "Analysis and Design of Security for Next Generation 4G Cellular Networks." In Proceedings of the 13th Annual Post Graduate Symposium on the Convergence of Telecommuni-Cations, Networking and Broad-Casting (PGNET), pp. 25-26, 2012.

[4] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 4: Design and Evaluation Report, ETSI/SAGE Specification, Version: 2.0 (September 9, 2011).

[5] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3.Document 1: 128-EEA3 and 128-EIA3 Specification. ETSI/SAGE Specification, Version: 1.7 (December30, 2011).

[6] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3. Document 2: ZUC Specification, ETSI/SAGE Specification, Version: 1.6 (June 28, 2011).

[7] “SNOW,” WIKIPEDIA, [Online]. Available: http://en.wikipedia.org/wiki/SNOW/ .[Accessed 24 February 2015].

[8] Fr´ed´eric L., Jr N. J., and Heul D. V. "Applications of SAT Solvers in Cryptanalysis: Finding Weak Keys and Preimages." Journal on Satisfiability, Boolean Modeling and Computation, Vol. 9, pp.1-25, 2014.

[9] “MiniSat in your browser,” Wonderings of sat geek, [Online]. Available: http://www.msoos.org/2013/09/minisat-in-your-browser/.[Accessed 15 January 2015].

[10] Qasem M., “SAT and MAX-SAT for the Lay-Researcher,” Public Authority for Applied Education and Training College of Technological Studies, [Online]. Available: http://www.mqasem.net/sat/sat/index.php. [Accessed 15 January 2015].

[11] Wheeler D. A., “MiniSAT User Guide: How to use the MiniSAT SAT Solver,” Wonderings of satgeek,[Online]. Available: http://www.dwheeler.com/essays/minisat-user-guide.html. [Accessed 15 January 2015].

[12] Soos M., Nohl K., and Castelluccia C. "Extending SAT solvers to cryptographic problems." In Theory and Applications of Satisfiability Testing-SAT, Springer Berlin Heidelberg, pp. 244–257, 2009.

[13] Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 & 128-EIA3 Document 3: Implementer's Test Data, ETSI/SAGE Specification, Version: 1.1 (January 4, 2011).

[14] "Boolean satisfiability problem," WIKIPEDIA, [Online].Available: http://en.wikipedia.org/wiki/Boolean_satisfiability_problem. [Accessed 12 January 2015].

[15] Schmalz M.S., "Organization of Computer Systems Course Notes," University of Florida, [Online]. Available: http://www.cise.ufl.edu/~mssz/CompOrg/CDA-arith.html. [Accessed 12 January 2015].

[16] Stallings, William. "Cryptography and network security, principles and practices", fifth Edtion, New York, USA: Practice Hall, 2011

[17] Legendre F., Dequen G., and Krajecki M. "Logical Reasoning to Detect Weaknesses About SHA-1 and MD4/5." IACR Cryptology ePrint Archive, 2014.

APPENDICES

Appendix A: Test parameters

Table A1: Test1 parameters

Key	(hex) 17 3d 14 ba 50 03 73 1d 7a 60 04 94 70 f0 0a 29
Count	(hex) 66035492
Bearer	(hex) f
Direction	(hex) 0
Length	193 bits
Plaintext:	(hex) 6cf65340 735552ab 0c9752fa 6f9025fe 0bd675d9 005875b2 00000000
Ciphertext:	(hex) a6c85fc6 6afb8533 aafc2518 dfe78494 0ee1e4b0 30238cc8 00000000

Table A2: Test 2 parameters

Key	(hex) e5 bd 3e a0 eb 55 ad e8 66 c6 ac 58 bd 54 30 2a
Count	(hex) 56823
Bearer	(hex) 18
Direction	(hex) 1
Length	800 bits
Plaintext:	(hex) 14a8ef69 3d678507 bbe7270a 7f67ff50 06c3525b 9807e467 c4e56000 ba338f5d 42955903 67518222 46c80d3b 38f07f4b e2d8ff58 05f51322 29bde93b bbdcaf38 2bf1ee97 2fbf9977 bada8945 847a2a6c 9ad34a66 7554e04d 1f7fa2c3 3241bd8f 01ba220d
Ciphertext:	(hex) 131d43e0 dea1be5c 5a1bfd97 1d852cbf 712d7b4f 57961fea 3208afa8 bca433f4 56ad09c7 417e58bc 69cf8866 d1353f74 865e8078 1d202dfb 3ecff7fc bc3b190f e82a204e d0e350fc 0f6f2613 b2f2bca6 df5a473a 57a4a00d 985ebad8 80d6f238 64a07b01

Table A3: Test 3 parameters

Key	(hex) d4 55 2a 8f d6 e6 1c c8 1a 20 09 14 1a 29 c1 0b
Count	(hex) 76452ec1
Bearer	(hex) 2
Direction	(hex) 1
Length	1570 bits
Plaintext:	(hex) 38f07f4b e2d8ff58 05f51322 29bde93b bbdcaf38 2bf1ee97 2fbf9977 bada8945 847a2a6c 9ad34a66 7554e04d 1f7fa2c3 3241bd8f 01ba220d 3ca4ec41 e074595f 54ae2b45 4fd97143 20436019 65cca85c 2417ed6c bec3bada 84fc8a57 9aea7837 b0271177 242a64dc 0a9de71a 8edee86c a3d47d03 3d6bf539 804eca86 c584a905 2de46ad3 fced6554 3bd90207 372b27af b79234f5 ff43ea87 0820e2c2 b78a8aae 61cce52a 0515e348 d196664a 3456b182 a07c406e 4a207912 71cfeda1 65d535ec 5ea2d4df 40000000
Ciphertext:	(hex) 8383b022 9fcc0b9d 2295ec41 c977e9c2 bb72e220 378141f9 c8318f3a 270dfbcd ee6411c2 b3044f17 6dc6e00f 8960f97a facd131a d6a3b49b 16b7babc f2a509eb b16a75dc ab14ff27 5dbeeea1 a2b155f9 d52c2645 2d0187c3 10a4ee55 beaa78ab 4024615b a9f5d5ad c7728f73 560671f0 13e5e550 085d3291 df7d5fec edded559 641b6c2f 585233bc 71e9602b d2305855 bbd25ffa 7f17ecbc 042daae3 8c1f57ad 8e8ebd37 346f71be fdbb7432 e0e0bb2c fc09bcd9 6570cb0c 0c39df5e 29294e82 703a637f 80000000

Table A4: Test 4 parameters

Key	(hex) db 84 b4 fb cc da 56 3b 66 22 7b fe 45 6f 0f 77
Count	(hex) e4850fe1
Bearer	(hex) 10
Direction	(hex) 1
Length	2798 bits
Plaintext:	(hex) e539f3b8 973240da 03f2b8aa 05ee0a00 dbafc0e1 82055dfe 3d7383d9 2cef40e9 2928605d 52d05f4f 9018a1f1 89ae3997 ce19155f b1221db8 bb0951a8 53ad852c e16cff07 382c93a1 57de00dd b125c753 9fd85045 e4ee07e0 c43f9e9d 6f414fc4 d1c62917 813f74c0 0fc83f3e 2ed7c45b a5835264 b43e0b20 afda6b30 53bfb642 3b7fce25 479ff5f1 39dd9b5b 995558e2 a56be18d d581cd01 7c735e6f 0d0d97c4 ddc1d1da 70c6db4a 12cc9277 8e2fbbd6 f3ba52af 91c9c6b6 4e8da4f7 a2c266d0 2d001753 df089603 93c5d568 88bf49eb 5c16d9a8 0427a416 bcb597df 5bfe6f13 890a07ee 1340e647 6b0d9aa8 f822ab0f d1ab0d20 4f40b7ce 6f2e136e b67485e5 07804d50 4588ad37 ffd81656 8b2dc403 11dfb654 cdead47e 2385c343 6203dd83 6f9c64d9 7462ad5d fa63b5cf e08acb95 32866f5c a787566f ca93e6b1 693ee15c f6f7a2d6 89d97417 98dc1c23 8e1be650 733b18fb 34ff880e 16bbd21b 47ac0000
Ciphertext:	(hex) 4bbfa91b a25d47db 9a9f190d 962a19ab 323926b3 51fbd39e 351e05da 8b8925e3 0b1cce0d 12211010 95815cc7 cb631950 9ec0d679 40491987 e13f0aff ac332aa6 aa64626d 3e9a1917 519e0b97 b655c6a1 65e44ca9 feac0790 d2a321ad 3d86b79c 5138739f a38d887e c7def449 ce8abdd3 e7f8dc4c a9e7b733 14ad310f 9025e619 46b3a56d c649ec0d a0d63943 dff592cf 962a7efb 2c8524e3 5a2a6e78 79d62604 ef268695 fa400302 7e22e608 30775220 64bd4a5b 906b5f53 1274f235 ed506cff 0154c754 928a0ce5 476f2cb1 020a1222 d32c1455 ecaef1e3 68fb344d 1735bfbe deb71d0a 33a2a54b 1da5a294 e679144d df11eb1a 3de8cf0c c0619179 74f35c1d 9ca0ac81 807f8fcc e6199a6c 7712da86 5021b04c e0439516 f1a526cc da9fd9ab bd53c3a6 84f9ae1e 7ee6b11d a138ea82 6c5516b5 aadf1abb e36fa7ff f92e3a11 76064e8d 95f2e488 2b5500b9 3228b219 4a475c1a 27f63f9f fd264989 a1bc0000

Appendix B: Test cases variables assignment

```
#include <stdio.h>
#include "zuc.c"
typedef unsigned char u8;
typedef unsigned int  u32;

void testCase1()
{
              //Test Set 1
              //Confidentiality Key - CK
              u8 CK[KEY_SIZE]={0x17 ,0x3d ,0x14 ,0xba ,0x50 ,0x03 ,0x73 ,0x1d
              ,0x7a ,0x60 ,0x04 ,0x94 ,0x70 ,0xf0 ,0x0a ,0x29};
              //IV parameters
              u32 COUNT = 0x66035492;
              u32 BEARER =0xf;
              u32 DIRECTION = 0x0;
              u32 LENGTH = 193;
              //Plaintext - M
              //MESSAGE_SIZE = 7;
              u32 M[MESSAGE_SIZE] = {0x6cf65340 ,0x735552ab ,0x0c9752fa
              ,0x6f9025fe ,0x0bd675d9 ,0x005875b2 ,0x00000000};
              //ciphertext - C
              u32 C[MESSAGE_SIZE] = {0xa6c85fc6 ,0x6afb8533 ,0xaafc2518
              ,0xdfe78494 ,0x0ee1e4b0 ,0x30238cc8 ,0x00000000};
}



void testCase2()
{
              //Test Set 2
              //Confidentiality Key - CK
              u8 CK[KEY_SIZE]={0xe5, 0xbd, 0x3e, 0xa0, 0xeb, 0x55, 0xad,
              0xe8, 0x66, 0xc6, 0xac, 0x58, 0xbd, 0x54, 0x30, 0x2a};
              //IV parameters
              u32 COUNT = 0x56823;
              u32 BEARER =0x18;
              u32 DIRECTION = 0x1;
              u32 LENGTH = 800;
              //Plaintext - M
              //MESSAGE_SIZE = 25;
              u32 M[MESSAGE_SIZE]={0x14a8ef69, 0x3d678507, 0xbbe7270a,
              0x7f67ff50, 0x06c3525b, 0x9807e467, 0xc4e56000, 0xba338f5d,
               0x42955903, 0x67518222, 0x46c80d3b, 0x38f07f4b, 0xe2d8ff58,
              0x05f51322, 0x29bde93b, 0xbbdcaf38,
               0x2bf1ee97, 0x2fbf9977, 0xbada8945, 0x847a2a6c, 0x9ad34a66,
              0x7554e04d, 0x1f7fa2c3, 0x3241bd8f,
               0x01ba220d  };
              //ciphertext - C
              u32 C[MESSAGE_SIZE]={0x131d43e0, 0xdea1be5c, 0x5a1bfd97,
              0x1d852cbf, 0x712d7b4f, 0x57961fea, 0x3208afa8, 0xbca433f4,
              0x56ad09c7, 0x417e58bc, 0x69cf8866, 0xd1353f74, 0x865e8078,
              0x1d202dfb, 0x3ecff7fc, 0xbc3b190f,
              0xe82a204e, 0xd0e350fc, 0x0f6f2613, 0xb2f2bca6, 0xdf5a473a,
              0x57a4a00d, 0x985ebad8, 0x80d6f238,
              0x64a07b01 };
}
```

Appendix C: Sample XOR CNF Generator Code

```
void xORCNFGenerator(u32 a, u32 b,u32 c)
{
       u32 i,xORCNFClauseCount=0;
       CNFVariable();

       decimalToBinary(a,32);

       for(i=0;i<32;i++)
              if(binaryResult[i]==0)
                     CNFVariable_a[i] = CNFVariable_a[i]*-1;

       decimalToBinary(b,32);

       for(i=0;i<32;i++)
              if(binaryResult[i]==0)
                     CNFVariable_b[i] = CNFVariable_b[i]*-1;

       decimalToBinary(c,32);

       for(i=0;i<32;i++)
              if(binaryResult[i]==0)
                     CNFVariable_c[i] = -1* CNFVariable_c[i];

       //CNF of c ⇔ (a ⊕ b)

       writeCharToDIMACSCNF('c');
       writeCharToDIMACSCNF(' ');
       writeCharToDIMACSCNF('X');
       writeCharToDIMACSCNF('O');
       writeCharToDIMACSCNF('R');
       writeCharToDIMACSCNF(' ');
       writeCharToDIMACSCNF('S');
       writeCharToDIMACSCNF('t');
       writeCharToDIMACSCNF('a');
       writeCharToDIMACSCNF('r');
       writeCharToDIMACSCNF('t');
       writeCharToDIMACSCNF(' ');
       writeCharToDIMACSCNF('\n');

       for (i = 0; i < 32; i++)
       {
              writeCharToDIMACSCNF('c');
              writeCharToDIMACSCNF(' ');
              writeNoToDIMACSCNF(++xORCNFClauseCount);
              writeCharToDIMACSCNF('\n');
              writeNoToDIMACSCNF(CNFVariable_a[i]);
              writeCharToDIMACSCNF(' ');
              writeNoToDIMACSCNF(CNFVariable_b[i]);
              writeCharToDIMACSCNF(' ');
              writeNoToDIMACSCNF(-1*CNFVariable_c[i]);
              writeCharToDIMACSCNF(' ');
              writeCharToDIMACSCNF('0');
              writeCharToDIMACSCNF('\n');
              CNFTotalClausesCount++;

              writeNoToDIMACSCNF(CNFVariable_a[i]);
              writeCharToDIMACSCNF(' ');
              writeNoToDIMACSCNF(-1*CNFVariable_b[i]);
              writeCharToDIMACSCNF(' ');
```

```
            writeNoToDIMACSCNF(CNFVariable_c[i]);
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF('0');
            writeCharToDIMACSCNF('\n');
            CNFTotalClausesCount++;

            writeNoToDIMACSCNF(-1*CNFVariable_a[i]);
            writeCharToDIMACSCNF(' ');
            writeNoToDIMACSCNF(CNFVariable_b[i]);
            writeCharToDIMACSCNF(' ');
            writeNoToDIMACSCNF(CNFVariable_c[i]);
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF('0');
            writeCharToDIMACSCNF('\n');
            CNFTotalClausesCount++;

            writeNoToDIMACSCNF(-1*CNFVariable_a[i]);
            writeCharToDIMACSCNF(' ');
            writeNoToDIMACSCNF(-1*CNFVariable_b[i]);
            writeCharToDIMACSCNF(' ');
            writeNoToDIMACSCNF(-1*CNFVariable_c[i]);
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF('0');
            writeCharToDIMACSCNF('\n');
            CNFTotalClausesCount++;
        }

        writeCharToDIMACSCNF('c');
        writeCharToDIMACSCNF(' ');
        writeCharToDIMACSCNF('X');
        writeCharToDIMACSCNF('O');
        writeCharToDIMACSCNF('R');
        writeCharToDIMACSCNF(' ');
        writeCharToDIMACSCNF('e');
        writeCharToDIMACSCNF('n');
        writeCharToDIMACSCNF('d');
        writeCharToDIMACSCNF(' ');
        writeCharToDIMACSCNF('\n');

}
```

Appendix D: Sample S-Box S0 CNF Generator Code

```
void sBox0CNFGenerator(u32 sBoxinput)
{
       u32 i,j,k,CNFClausePerColumnCount=0;
       CNFSBoxVariable();

       decimalToBinary(sBoxinput,8);

       //makes the variable Format cnf variable according to the the binary
       //input of the box.
       for(i=0;i<8;i++)
              if(binaryResult[i]==0)
                     CNFVariable_SBox[i] = -1*CNFVariable_SBox[i];

       //8 CNF for representing each output columns of S-Box truth table
       for ( j = 0; j < 8; j++)
       {
              CNFClausePerColumnCount=0;

              writeCharToDIMACSCNF('c');
              writeCharToDIMACSCNF(' ');
              writeCharToDIMACSCNF('o');
              writeNoToDIMACSCNF(j+1);
              writeCharToDIMACSCNF(' ');
              writeCharToDIMACSCNF('S');
              writeCharToDIMACSCNF('t');
              writeCharToDIMACSCNF('a');
              writeCharToDIMACSCNF('r');
              writeCharToDIMACSCNF('t');
              writeCharToDIMACSCNF(' ');
              writeCharToDIMACSCNF('\n');
              //traverse through the column
              for ( i = 0; i < 256; i++)
              {
                     //if only the value of the output column = 0 Generate
                     //the CNF
                     if(S0_Binary[i][j]=='0')
                     {
                            // 8 input columns of the of S-Box truth table
                            for ( k = 8; k < 16; k++)
                            {
                                   //if the value of the input column = 1
                                   //negate the //variable Format
                                   if(S0_Binary[i][k]=='1')
                                   {

                            writeNoToDIMACSCNF(-1*CNFVariable_SBox[k-8]);
                                   }
                                   else
                                   {
                                   //if the value of the input column = 0 do
                                   //nothing the variable Format
                            writeNoToDIMACSCNF(CNFVariable_SBox[k-8]);
                                   }
                                   writeCharToDIMACSCNF(' ');
                            }
                            writeCharToDIMACSCNF('0');
                            writeCharToDIMACSCNF('\n');
                            CNFTotalClausesCount++; //total number of cluses
                            CNFClausePerColumnCount++; // clauses per column
```

```
                }
            }
            writeCharToDIMACSCNF('c');
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF('o');
            writeNoToDIMACSCNF(j+1);
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF('E');
            writeCharToDIMACSCNF('n');
            writeCharToDIMACSCNF('d');
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF('C');
            writeCharToDIMACSCNF('l');
            writeCharToDIMACSCNF('a');
            writeCharToDIMACSCNF('u');
            writeCharToDIMACSCNF('s');
            writeCharToDIMACSCNF('e');
            writeCharToDIMACSCNF('s');
            writeCharToDIMACSCNF(' ');
            writeCharToDIMACSCNF(':');
            writeCharToDIMACSCNF(' ');
            writeNoToDIMACSCNF(CNFClausePerColumnCount);
            writeCharToDIMACSCNF('\n');
        }
    }
```