ETSI/SAGE **Specification**	**Version: 2.1** **Date: 16th March 2009**

Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2.
Document 1: UEA2 and UIA2 Specification

Document History		
V1.0	**10th January 2006**	**Publication**
V1.1	**6th September 2006**	**No change to the algorithm specification at all, just removal of an unwanted page header**
V2.1	**16th March 2009**	**Improvement of C code (SP-090140)**

PREFACE

This specification has been prepared by the 3GPP Task Force, and gives a detailed specification of the 3GPP confidentiality algorithm ***UEA2*** and the 3GPP integrity algorithm ***UIA2***.

This document is the first of four, which between them form the entire specification of 3GPP Confidentiality and Integrity Algorithms:

- Specification of the 3GPP Confidentiality and Integrity Algorithms ***UEA2*** & ***UIA2***.
  Document 1: ***UEA2*** and ***UIA2*** Algorithm Specifications.

- Specification of the 3GPP Confidentiality and Integrity Algorithms ***UEA2*** & ***UIA2***.
  Document 2: **SNOW 3G** Algorithm Specification.

- Specification of the 3GPP Encryption and Confidentiality Algorithms ***UEA2*** & ***UIA2***.
  Document 3: Implementors' Test Data.

- Specification of the 3GPP Encryption and Confidentiality Algorithms ***UEA2*** & ***UIA2***.
  Document 4: Design Conformance Test Data.

The normative part of the specification of the ***UEA2*** (confidentiality) and ***UIA2*** (integrity) algorithms is in the main body of this document. The annexes to this document are purely informative.

The informative section of this document includes four informative annexes: Annex 1 contains remarks about the mathematical background of some functions of ***UIA2***. Annex 2 contains implementation options for some functions of ***UIA2***. Annex 3 contains illustrations of functional elements of the algorithms, while Annex 4 contains an implementation program listing of the cryptographic algorithm specified in the main body of this document, written in the programming language C.

The normative section of the specification of the stream cipher (**SNOW 3G**) on which they are based is in the main body of Document 2. The annexes to that document, and Documents 3 and 4 above, are purely informative.

Blank Page

TABLE OF CONTENTS

REFERENCES

[1] 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Security Architecture (3G TS 33.102 version 6.3.0).

[2] 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Cryptographic Algorithm Requirements; (3G TS 33.105 version 6.0.0).

[3] Specification of the 3GPP Confidentiality and Integrity Algorithms;
Document 1: ***f8*** and ***f9*** specifications; (3GPP TS35.201 Release 6).

[4] Specification of the 3GPP Confidentiality and Integrity Algorithms ***UEA2*** & ***UIA2***.
Document 1: ***UEA2*** and ***UIA2*** specifications.

[5] Specification of the 3GPP Confidentiality and Integrity Algorithms ***UEA2*** & ***UIA2***.
Document 2: **SNOW 3G** specification.

[6] Specification of the 3GPP Confidentiality and Integrity Algorithms ***UEA2*** & ***UIA2***.
Document 3: Implementors' Test Data.

[7] Specification of the 3GPP Confidentiality and Integrity Algorithms ***UEA2*** & ***UIA2***.
Document 4: Design Conformance Test Data.

[8] P. Ekdahl and T. Johansson, "A new version of the stream cipher SNOW", in Selected Areas in Cryptology (SAC 2002), LNCS 2595, pp. 47–61, Springer-Verlag.

NORMATIVE SECTION

This part of the document contains the normative specification of the Confidentiality and Integrity algorithms.

1. OUTLINE OF THE NORMATIVE PART

Section 2 introduces the algorithm and describes the notation used in the subsequent sections.

Section 3 specifies the confidentiality algorithm ***UEA2.***

Section 4 specifies the integrity algorithm ***UIA2.***

2. INTRODUCTORY INFORMATION

2.1. Introduction

Within the security architecture of the 3GPP system there are standardised algorithms for confidentiality (***f8***) and integrity (***f9***). A first set of algorithms for ***f8*** and ***f9*** (***UEA1*** and ***UIA1***) has already been specified [3]. A second set of algorithms for ***f8*** and ***f9*** (***UEA2*** and ***UIA2***) are fully specified here: The second set of these algorithms is based on the **SNOW 3G** algorithm that is specified in a companion document [5].

The confidentiality algorithm ***UEA2*** is a stream cipher that is used to encrypt/decrypt blocks of data under a confidentiality key **CK**. The block of data may be between 1 and 2^{32}bits long. The algorithm uses **SNOW 3G** as a keystream generator

The integrity algorithm ***UIA2*** computes a 32-bit MAC (Message Authentication Code) of a given input message using an integrity key **IK**. The message may be between 1 and 2^{32} bits long. The approach adopted uses **SNOW 3G.**

Note: for both UEA2 and UIA2, the length limit of 2^{32} bits is intended to be a safe value: comfortably lower than any point at which security of the algorithms starts to fail, but comfortably enough for any anticipated application.

2.2. Notation

2.2.1. Radix

We use the prefix **0x** to indicate **hexadecimal** numbers.

2.2.2. Conventions

We use the assignment operator ‘=’, as used in several programming languages. When we write

<variable> = <expression>

we mean that <variable> assumes the value that <expression> had before the assignment took place. For instance,

$$x = x + y + 3$$

means

(new value of x) becomes (old value of x) + (old value of y) + 3.

2.2.3. Bit/Byte ordering

All data variables in this specification are presented with the most significant bit (or byte) on the left hand side and the least significant bit (or byte) on the right hand side. Where a variable is broken down into a number of sub-strings, the left most (most significant) sub-string is numbered 0, the next most significant is numbered 1 and so on through to the least significant.

For example an n-bit **MESSAGE** is subdivided into 64-bit substrings $\mathbf{MB_0}$**,** $\mathbf{MB_1}$**,** $\mathbf{MB_2}$**, ….** So if we have a message:

0x0123456789ABCDEFFEDCBA98765432108654381AB594FC28786404C50A37…

we have:

$\mathbf{MB_0}$ = 0x0123456789ABCDEF

$\mathbf{MB_1}$ = 0xFEDCBA9876543210

$\mathbf{MB_2}$ = 0x86545381AB594FC2

$\mathbf{MB_3}$ = 0x8786404C50A37…

In binary this would be:

000000010010001101000101011001111000100110101011110011011110111111111110

with $\mathbf{MB_0}$ = 0000000100100011010001010110011110001001101010111100110111101111

$\mathbf{MB_1}$ = 1111111011011100101110101001100001110110010101000011001000010000

$\mathbf{MB_2}$ = 1000011001010100010100111000000110101011010110010100111111000010

$\mathbf{MB_3}$ = 1000011110000110010000000100110001010000101000110111…

2.2.4. List of Symbols

=	The assignment operator.
⊕	The bitwise exclusive-OR operation
||	The concatenation of the two operands.
$\lceil x \rceil$	The smallest integer greater than or equal to the real number *x*.
$\&_n$	The bitwise AND operation in an n-bit register.
$<<_n t$	t-bit left shift in an n-bit register.
$>>_n t$	t-bit right shift in an n-bit register

2.3. List of Variables

BEARER the 5-bit input to the ***UEA2*** function.

CK the 128-bit confidentiality key.

COUNT the 32-bit time variant input to the ***UEA2*** and ***UIA2*** functions (COUNT-C for ***UEA2*** and COUNT-I for ***UIA2***)

DIRECTION the 1-bit input to both the ***UEA2*** and ***UIA2*** functions indicating the direction of transmission (uplink or downlink).

FRESH the 32-bit random input to the ***UIA2*** function.

IBS the input bit stream to the ***UEA2*** function.

IK the 128-bit integrity key.

KS[i] the i^{th} bit of keystream produced by the keystream generator.

LENGTH the input to the ***UEA2*** and ***UIA2*** functions which specifies the number of bits in the input bitstream ($1-2^{32}$).

MAC-I the 32-bit message authentication code (MAC) produced by the integrity function ***UIA2***.

MESSAGE the input bitstream of LENGTH bits that is to be processed by the ***UIA2*** function.

OBS the output bit stream from the ***UEA2*** function.

$z_1, z_2, \ldots$ the 32-bit words forming the keystream sequence of **SNOW 3G**. The word produced first is z_1, the next word z_2 and so on.

3. CONFIDENTIALITY ALGORITHM *UEA2*

3.1. Introduction

The confidentiality algorithm ***UEA2*** is a stream cipher that encrypts/decrypts blocks of data between 1 and 2^{32} bits in length.

3.2. Inputs and Outputs

The inputs to the algorithm are given in Table 1, the output in Table 2:

Parameter	Size (bits)	Comment
COUNT-C	32	Frame dependent input COUNT-C[0]…COUNT-C[31]
BEARER	5	Bearer identity BEARER[0]…BEARER[4]
DIRECTION	1	Direction of transmission DIRECTION[0]
CK	128	Confidentiality key CK[0]….CK[127]
LENGTH	Unspecified	The number of bits to be encrypted/decrypted
IBS	LENGTH	Input bit stream IBS[0]….IBS[LENGTH-1]

Table 1. ***UEA2*** inputs

Parameter	Size (bits)	Comment
OBS	LENGTH	Output bit stream OBS[0]….OBS[LENGTH-1]

Table 2. ***UEA2*** output

3.3. Components and Architecture

The keystream generator is based on **SNOW 3G** that is specified in [5]. **SNOW 3G** is a word oriented stream cipher and generates a keystream in multiples of 32-bits.

3.4. Initialisation

In this section we define how the keystream generator is initialised with the key variables before the generation of keystream bits.

All variables have length 32 and are presented with the most significant bit on the left hand side and the least significant bit on the right hand side.

$\mathbf{K_3 = CK[0] \,||\, CK[1] \,||\, CK[2] \,||\, \ldots \,||\, CK[31]}$

$\mathbf{K_2 = CK[32] \,||\, CK[33] \,||\, CK[34] \,||\, \ldots \,||\, CK[63]}$

$\mathbf{K_1 = CK[64] \,||\, CK[65] \,||\, CK[66] \,||\, \ldots \,||\, CK[95]}$

$\mathbf{K_0 = CK[96] \,||\, CK[97] \,||\, CK[98] \,||\, \ldots \,||\, CK[127]}$

$\mathbf{IV_3 = COUNT\text{-}C[0] \,||\, COUNT\text{-}C[1] \,||\, COUNT\text{-}C[2] \,||\, \ldots \,||\, COUNT\text{-}C[31]}$

$\mathbf{IV_2 = BEARER[0] \,||\, BEARER[1] \,||\, \ldots \,||\, BEARER[4] \,||\, DIRECTION[0] \,||\, 0 \,||\, \ldots \,||\, 0}$

$\mathbf{IV_1 = COUNT\text{-}C[0] \,||\, COUNT\text{-}C[1] \,||\, COUNT\text{-}C[2] \,||\, \ldots \,||\, COUNT\text{-}C[31]}$

$\mathbf{IV_0 = BEARER[0] \,||\, BEARER[1] \,||\, \ldots \,||\, BEARER[4] \,||\, DIRECTION[0] \,||\, 0 \,||\, \ldots \,||\, 0}$

SNOW 3G is initialised as described in document [5].

3.5. Keystream Generation

Set **L** = $\lceil$**LENGTH** / 32$\rceil$.

SNOW 3G is run as described in document [5] to produce the keystream consisting of the 32-bit words $\mathbf{z}_1 \ldots \mathbf{z}_L$. The word produced first is $\mathbf{z}_1$, the next word $\mathbf{z}_2$ and so on.

The sequence of keystream bits is **KS[0] … KS[LENGTH-1]**, where **KS[0]** is the most significant bit and **KS[31]** is the least significant bit of $\mathbf{z}_1$, **KS[32]** is the most significant bit of $\mathbf{z}_2$ and so on.

3.6. Encryption/Decryption

Encryption/decryption operations are identical operations and are performed by the exclusive-OR of the input data (**IBS**) with the generated keystream (**KS**).

For each integer $\boldsymbol{i}$ with $0 \le \boldsymbol{i} \le$ **LENGTH**-1 we define:

$\mathbf{OBS}[i] = \mathbf{IBS}[i] \oplus \mathbf{KS}[i]$.

4. INTEGRITY ALGORITHM *UIA2*

4.1. Introduction

The integrity algorithm ***UIA2*** computes a Message Authentication Code (MAC) on an input message under an integrity key ***IK***. The message may be between 1 and 2^{32} bits in length.

For ease of implementation the algorithm is based on the same stream cipher (**SNOW 3G**) as is used by the confidentiality algorithm ***UEA2***.

4.2. Inputs and Outputs

The inputs to the algorithm are given in table 3, the output in table 4:

Parameter	Size (bits)	Comment
COUNT-I	32	Frame dependent input COUNT-I[0]…COUNT-I[31]
FRESH	32	Random number FRESH[0]…FRESH[31]
DIRECTION	1	Direction of transmission DIRECTION[0]
IK	128	Integrity key IK[0]…IK[127]
LENGTH	64	The number of bits to be ‘MAC’d
MESSAGE	LENGTH	Input bit stream

Table 3. ***UIA2*** inputs

Parameter	Size (bits)	Comment
MAC-I	32	Message authentication code MAC-I[0]…MAC-I[31]

Table 4. ***UIA2*** output

4.3. Components and Architecture

4.3.1. SNOW 3G

The integrity function uses **SNOW 3G** that is specified in [5]. **SNOW 3G** is a word oriented stream cipher and generates from the key and an initialisation variable five 32-bit-words $\mathbf{z}_1$, $\mathbf{z}_2$, $\mathbf{z}_3$, $\mathbf{z}_4$ and $\mathbf{z}_5$.

4.3.2. MULx

MULx maps 128 bits to 64 bits. Let V and c be 64-bit input values. Then MULx is defined:
If the leftmost (i.e. the most significant) bit of V equals 1, then

$\mathrm{MULx}(V, c) = (V \ll_{64} 1) \oplus c,$

else

$\mathrm{MULx}(V, c) = V \ll_{64} 1.$

4.3.3. MULxPOW

MULxPOW maps 128 bits and a positive integer i to 64 bit. Let V and c be 64-bit input values, then MULxPOW(V, i, c) is recursively defined:

If i equals 0, then

$\mathrm{MULxPOW}(V, i, c) = V,$

else

$\mathrm{MULxPOW}(V, i, c) = \mathrm{MULx}(\mathrm{MULxPOW}(V, i - 1, c), c).$

4.3.4. MUL

MUL maps 192 bits to 64 bit. Let V, P and c be 64-bit input values.

Then the 64-bit output *result* of MUL(V, P, c) is computed as follows:

- *result* = 0.
- for i = 0 to 63 inclusive
  - if $(P \gg_{64} i) \mathbin{\&_{64}} \mathrm{0x01}$ equals 0x01, then
    $result = result \oplus \mathrm{MULxPOW}(V, i, c).$

4.4. Initialisation

In this section we define how the keystream generator is initialised with the key and initialisation variables before the generation of keystream bits.

All variables have length 32 bits and are presented with the most significant bit on the left hand side and the least significant bit on the right hand side.

$\mathbf{K_3} = \mathbf{IK}[0] \;\|\; \mathbf{IK}[1] \;\|\; \mathbf{IK}[2] \;\|\; \ldots \;\|\; \mathbf{IK}[31]$

$\mathbf{K_2} = \mathbf{IK}[32] \;\|\; \mathbf{IK}[33] \;\|\; \mathbf{IK}[34] \;\|\; \ldots \;\|\; \mathbf{IK}[63]$

$\mathbf{K_1} = \mathbf{IK}[64] \;\|\; \mathbf{IK}[65] \;\|\; \mathbf{IK}[66] \;\|\; \ldots \;\|\; \mathbf{IK}[95]$

$\mathbf{K_0} = \mathbf{IK}[96] \;\|\; \mathbf{IK}[97] \;\|\; \mathbf{IK}[98] \;\|\; \ldots \;\|\; \mathbf{IK}[127]$

$\mathbf{IV_3} = \mathbf{COUNT\text{-}I}[0] \;\|\; \mathbf{COUNT\text{-}I}[1] \;\|\; \mathbf{COUNT\text{-}I}[2] \;\|\; \ldots \;\|\; \mathbf{COUNT\text{-}I}[31]$

$\mathbf{IV_2} = \mathbf{FRESH}[0] \;\|\; \mathbf{FRESH}[1] \;\|\; \mathbf{FRESH}[2] \;\|\; \ldots \;\|\; \mathbf{FRESH}[31]$

$\mathbf{IV_1} = \mathbf{DIRECTION}[0] \oplus \mathbf{COUNT\text{-}I}[0] \;\|\; \mathbf{COUNT\text{-}I}[1] \;\|\; \mathbf{COUNT\text{-}I}[2] \;\|\; \ldots \;\|\; \mathbf{COUNT\text{-}I}[31]$

$\mathbf{IV_0} = \mathbf{FRESH}[0] \;\|\; \mathbf{FRESH}[1] \;\|\; \ldots \;\|\; \mathbf{FRESH}[15] \;\|\; \mathbf{FRESH}[16] \oplus \mathbf{DIRECTION}[0] \;\|\; \mathbf{FRESH}[17] \;\|\; \ldots \;\|\; \mathbf{FRESH}[31]$

SNOW 3G is initialised as described in document [5].

4.5. Calculation

Set $\mathbf{D} = \lceil \mathbf{LENGTH} / 64 \rceil + 1$.

SNOW 3G is run as described in document [5] in order to produce 5 keystream words $\mathbf{z}_1$, $\mathbf{z}_2$, $\mathbf{z}_3$, $\mathbf{z}_4$, $\mathbf{z}_5$.

Set $\mathbf{P} = \mathbf{z}_1 \| \mathbf{z}_2$

and $\mathbf{Q} = \mathbf{z}_3 \| \mathbf{z}_4$.

Let **OTP**[0], **OTP**[1], **OTP**[2], …, **OTP**[31] be bit-variables such that

$$\mathbf{z}_5 = \mathbf{OTP}[0] \,\|\, \mathbf{OTP}[1] \,\|\ldots\|\, \mathbf{OTP}[31],$$

i.e. **OTP**[0] is the most and **OTP**[31] the least significant bit of $\mathbf{z}_5$.

For $0 \le i \le \mathbf{D} - 3$ set

$$\mathbf{M}_i = \mathbf{MESSAGE}[64i] \,\|\, \mathbf{MESSAGE}[64i+1] \,\|\ldots\|\, \mathbf{MESSAGE}[64i+63].$$

Set

$$\mathbf{M}_{\mathbf{D}-2} = \mathbf{MESSAGE}[64(\mathbf{D}-2)] \,\|\, \ldots \,\|\, \mathbf{MESSAGE}[\mathbf{LENGTH}-1] \,\|\, 0\ldots0.$$

Let **LENGTH**[0], **LENGTH**[1], …, **LENGTH**[63] be the bits of the 64-bit representation of **LENGTH**, where **LENGTH**[0] is the most and **LENGTH**[63] is the least significant bit.

Set $\mathbf{M}_{\mathbf{D}-1} = \mathbf{LENGTH}[0] \,\|\, \mathbf{LENGTH}[1] \,\|\, \ldots \,\|\, \mathbf{LENGTH}[63]$.

Compute the function Eval_M:

- Set the 64-bit variable **EVAL** = 0.
- for $i = 0$ to **D** – 2 inclusive:
  - $\mathbf{EVAL} = \mathrm{Mul}(\mathbf{EVAL} \oplus \mathbf{M}_i, \mathbf{P}, \text{0x000000000000001b})$.

Set $\mathbf{EVAL} = \mathbf{EVAL} \oplus \mathbf{M}_{D-1}$

Now we multiply **EVAL** by **Q**:

$$\mathbf{EVAL} = \mathrm{Mul}(\mathbf{EVAL}, \mathbf{Q}, \text{0x000000000000001b}).$$

Let $\mathbf{EVAL} = \mathbf{e}_0 \,\|\, \mathbf{e}_1 \,\|\, \ldots \,\|\, \mathbf{e}_{63}$ with $\mathbf{e}_0$ the most and $\mathbf{e}_{63}$ the least significant bit.

For $0 \le i \le 31$, set

$$\mathbf{MAC\text{-}I}[i] = \mathbf{e}_i \oplus \mathbf{OTP}[i].$$

The bits $\mathbf{e}_{32}$, …, $\mathbf{e}_{63}$ are discarded.

INFORMATIVE SECTION

This part of the document is purely informative and does not form part of the normative specification of the Confidentiality and Integrity algorithms.

ANNEX 1
Remarks about the mathematical background of some operations of the *UIA2* Algorithm

1.1. The function EVAL_M

The first part (the function EVAL_M) of the calculations for the ***UIA2*** algorithm corresponds to the evaluation of a polynomial at a secret point: From the bits and the length of **MESSAGE** a polynomial $\mathbf{M} \in GF(2^{64})[X]$ is defined. This polynomial is evaluated at the point $\mathbf{P} \in GF(2^{64})$ defined by $\mathbf{z_1} \| \mathbf{z_2}$.

This can be seen as follows:

Consider the Galois Field $GF(2^{64})$ where elements of the field are represented as polynomials over GF(2) modulo the irreducible polynomial $x^{64} + x^4 + x^3 + x + 1$.

Variables consisting of 64 bits can be mapped to this field by interpreting the bits as the coefficients of the corresponding polynomial.

For example for $0 \leq i \leq \mathbf{D}\text{-}3$ the variable
$\mathbf{M}_i$ **=** **MESSAGE**[64*i*] || **MESSAGE**[64*i*+1] **||**...|| **MESSAGE**[64*i*+62] || **MESSAGE**[64*i*+63]
is interpreted as
MESSAGE[64*i*]x^{63}+ **MESSAGE**[64*i*+1]x^{62} + ... + **MESSAGE**[64*i*+62]x +
MESSAGE[64*i*+63].

Construct the polynomial **M** of degree **D**-1 in $GF(2^{64})[X]$ as
$\mathbf{M}(X) = \mathbf{M_0}X^{\mathbf{D}-1} + \mathbf{M_1}X^{\mathbf{D}-2} + \ldots + \mathbf{M}_{\mathbf{D}-2}X + \mathbf{M}_{\mathbf{D}-1}$.

Evaluate the polynomial **M** at the point **P**, i.e. compute
$\mathbf{M(P)} = \mathbf{M}_0\mathbf{P}^{\mathbf{D}-1} + \mathbf{M}_1\mathbf{P}^{\mathbf{D}-2} + \ldots + \mathbf{M}_{\mathbf{D}-2}\mathbf{P} + \mathbf{M}_{\mathbf{D}-1} = (\ldots(\mathbf{M}_0\mathbf{P} + \mathbf{M}_1)\mathbf{P} + \mathbf{M}_2)\mathbf{P} + \ldots + \mathbf{M}_{\mathbf{D}-2})\mathbf{P} + \mathbf{M}_{\mathbf{D}-1}$.

This is done in the function Eval_M in 4.5.

1.2. The function MUL(V, P, c)

The function MUL(*V*, *P*, c) (see 4.3.4) corresponds to a multiplication of *V* by *P* in $GF(2^{64})$. Here $GF(2^{64})$ is described as $GF(2)(\beta)$ where β is a root of the GF(2)[x] polynomial $x^{64} + c_0x^{63} + \ldots + c_{62}x + c_{63}$ and $c = c_0 \| c_1 \| \ldots \| c_{63}$.

ANNEX 2
Implementation options for some operations of the *UIA2* Algorithm

The function MUL (see 4.3.4) can be implemented using table lookups. This might accelerate execution of the function EVAL_M, as for the evaluation of the polynomial only multiplication by a constant factor *P* is needed.

There are different possible sizes for the tables. Here we use 8 tables with 256 entries, but for example it is also possible to use 16 tables with 16 entries.

In order to execute MUL by table-lookups first Pre_Mul_P (see 2.1) is executed, which generates the tables. Then in MUL_P (see 2.2) the multiplication is performed by 8 table-lookups and an xor of the results.

Hence in 4.5 instead of **EVAL** = Mul(EVAL $\oplus$ $\mathbf{M}_i$, **P**, 0x1b) we can use **EVAL** = Mul_P(**EVAL** $\oplus$ $\mathbf{M}_i$).

2.1. Procedure Pre_Mul_P

In order to be able to compute Mul_P (see 2.2) the procedure Pre_Mul_P is executed once before the first call of Mul_P.
Pre_Mul_P computes from the 64-bit input **P** eight tables PM[0], PM[1], …, PM[7]. Each of these tables contains 256 entries PM[j][0], PM[j][1], …, PM[j][255] with 64 bits.

For $0 \le \mathrm{j} \le 7$ and $0 \le X \le 255$ the value PM[j][*X*] corresponds to $X\,\mathbf{P}\,\mathrm{x}^{8\mathrm{j}}$.

Let *r* be the 64-bit value 0x000000000000001b.

- The tables are computed as follows:
  PM[0][0] = PM[1][0] = PM[2][0] = PM[3][0] = PM[4][0] = PM[5][0] = PM[6][0] = PM[7][0] = 0.

- PM[0][1] = **P**.

- for *i* = 1 to 63 inclusive:

  - $\mathrm{PM}[i >>_8 3][1 <<_8 (i \,\&_8\, 0\mathrm{x}07)] = \mathrm{PM}[(i - 1) >>_8 3][1 <<_8 ((i - 1) \,\&_8\, 0\mathrm{x}07)] <<_{64} 1$.

  - if the leftmost bit of $\mathrm{PM}[(i - 1) >>_8 3][1 << ((i - 1) \,\&_8\, 0\mathrm{x}07)]$ equals 1, then
    $\mathrm{PM}[i >>_8 3][1 <<_8 (i \,\&_8\, 0\mathrm{x}07)] = \mathrm{PM}[i >>_8 3][1 << (i \,\&_8\, 0\mathrm{x}07)] \oplus r$.

- for *i* = 0 to 7 inclusive

  - for *j* = 1 to 7 inclusive

    - for $k = 1$ to $(1 <<_8 j) - 1$ inclusive

      - $\mathrm{PM}[i][(1 <<_8 j) + k] = \mathrm{PM}[i][1 <<_8 j] \oplus \mathrm{PM}[i][k]$.

2.2. Function Mul_P

The function Mul_P maps a 64-bit input *X* to a 64-bit output.

Let $X = X_0 \parallel X_1 \parallel X_2 \parallel X_3 \parallel X_4 \parallel X_5 \parallel X_6 \parallel X_7$, with X_0 the most and X_7 the least significant byte.

Compute Mul_P(X) as

$$\begin{aligned}\text{Mul\_P}(X) = {} & \text{PM}[0][X_7] \oplus \text{PM}[1][X_6] \oplus \text{PM}[2][X_5] \oplus \text{PM}[3][X_4] \oplus \text{PM}[4][X_3] \oplus \\ & \text{PM}[5][X_2] \oplus \text{PM}[6][X_1] \oplus \text{PM}[7][\mathrm{X}_0].\end{aligned}$$

ANNEX 3
Figures of the *UEA2* and *UIA2* Algorithms

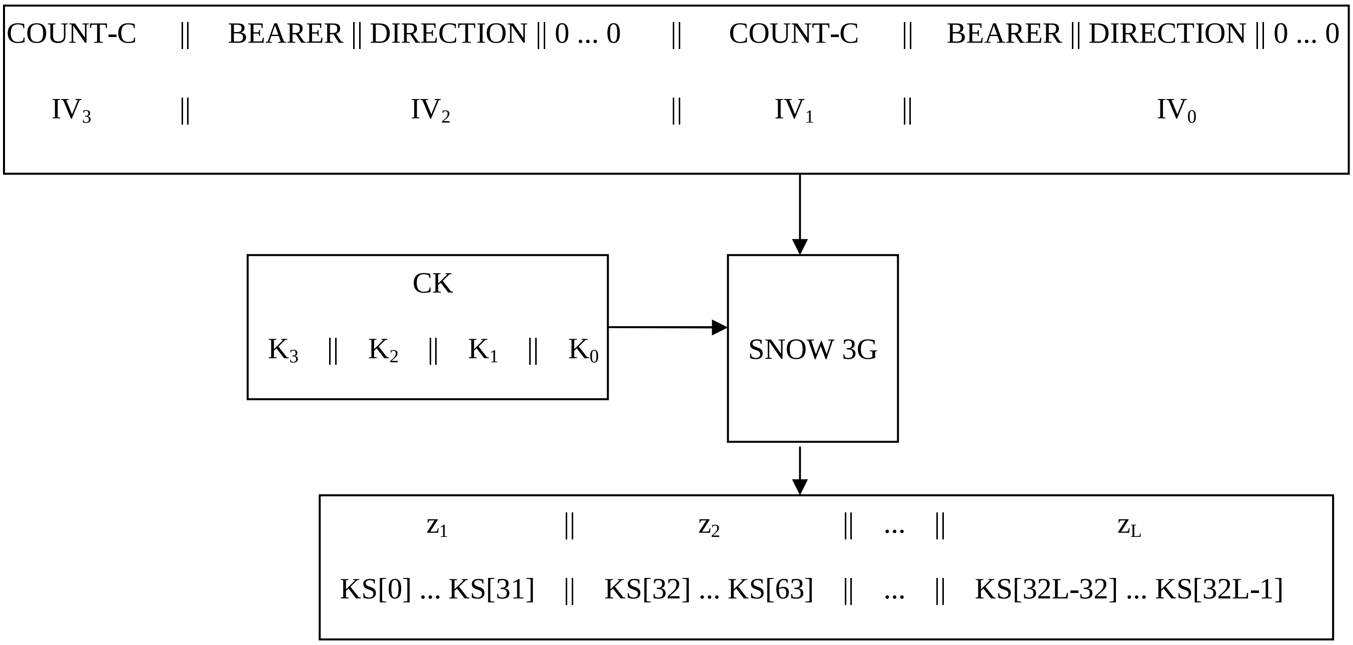
COUNT-C || BEARER || DIRECTION || 0 ... 0 || COUNT-C || BEARER || DIRECTION || 0 ... 0
IV_3 || IV_2 || IV_1 || IV_0
CK
K_3 || K_2 || K_1 || K_0
SNOW 3G
z_1 || z_2 || ... || z_L
KS[0] ... KS[31] || KS[32] ... KS[63] || ... || KS[32L-32] ... KS[32L-1]

Figure 1: *UEA2* Keystream Generator

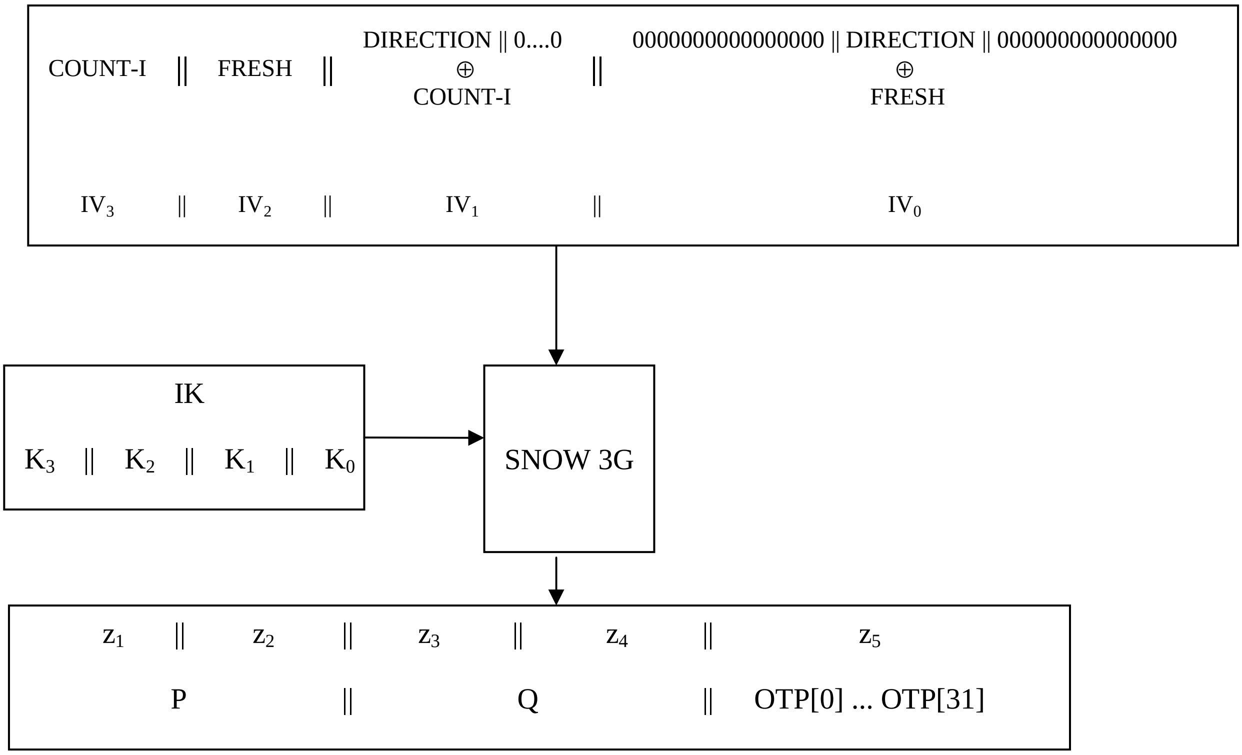
COUNT-I || FRESH || DIRECTION || 0....0 ⊕ COUNT-I || 0000000000000000 || DIRECTION || 000000000000000 ⊕ FRESH
IV3 || IV2 || IV1 || IV0
IK
K3 || K2 || K1 || K0
SNOW 3G
z1 || z2 || z3 || z4 || z5
P || Q || OTP[0] ... OTP[31]

Figure 2: *UIA2* Integrity function, part 1

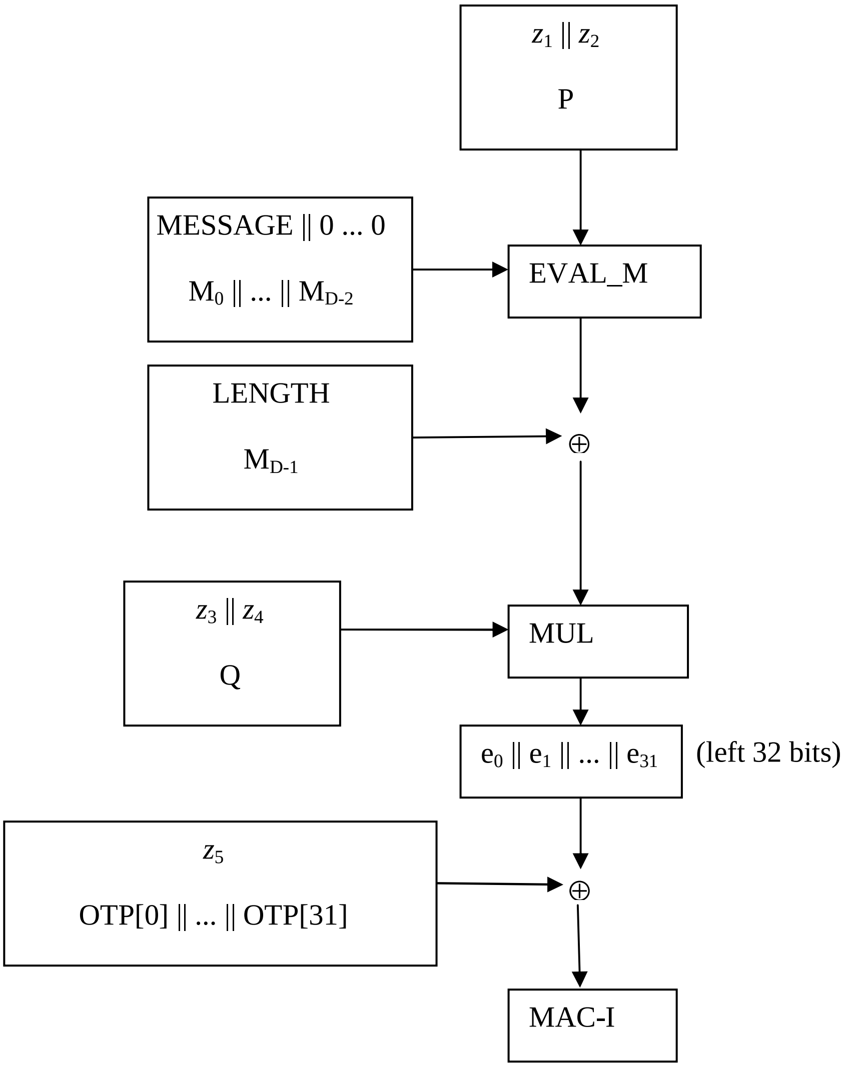

Figure 3: *UIA2* Integrity function, part 2

ANNEX 4
Simulation Program Listing

4.1. UEAII

4.1.1 Header File

```
/*---------------------------------------------------------
 *                  f8.h
 *---------------------------------------------------------*/
#ifndef F8_H_
#define F8_H_

#include "SNOW_3G.h"

/* f8.
 * Input key: 128 bit Confidentiality Key.
 * Input count:32-bit Count, Frame dependent input.
 * Input bearer: 5-bit Bearer identity (in the LSB side).
 * Input dir:1 bit, direction of transmission.
 * Input data: length number of bits, input bit stream.
 * Input length: 32 bit Length, i.e., the number of bits to be encrypted or
 *               decrypted.
 * Output data: Output bit stream. Assumes data is suitably memory
 * allocated.
 * Encrypts/decrypts blocks of data between 1 and 2^32 bits in length as
 * defined in Section 3.
 */

void f8( u8 *key, u32 count, u32 bearer, u32 dir, u8 *data, u32 length );

#endif
```

4.1.2 Code

```
/*---------------------------------------------------------
 *                  f8.c
 *---------------------------------------------------------*/
#include "f8.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


/* f8.
 * Input key: 128 bit Confidentiality Key.
 * Input count:32-bit Count, Frame dependent input.
 * Input bearer: 5-bit Bearer identity (in the LSB side).
 * Input dir:1 bit, direction of transmission.
 * Input data: length number of bits, input bit stream.
 * Input length: 32 bit Length, i.e., the number of bits to be encrypted or
 *               decrypted.
 * Output data: Output bit stream. Assumes data is suitably memory
 * allocated.
 * Encrypts/decrypts blocks of data between 1 and 2^32 bits in length as
 * defined in Section 3.
 */

void f8( u8 *key, u32 count, u32 bearer, u32 dir, u8 *data, u32 length )
{
  u32 K[4],IV[4];
  int n = ( length + 31 ) / 32;
  int i=0;
  u32 *KS;

  /*Initialisation*/

  /* Load the confidentiality key for SNOW 3G initialization as in section
3.4. */
  for (i=0; i<4; i++)
    K[3-i] = (key[4*i] << 24) ^ (key[4*i+1] << 16) ^ (key[4*i+2] << 8) ^
(key[4*i+3]);
```

```
  /* Prepare the initialization vector (IV) for SNOW 3G initialization as in
section 3.4. */
  IV[3] = count;
  IV[2] = (bearer << 27) | ((dir & 0x1) << 26);

  IV[1] = IV[3];
  IV[0] = IV[2];

  /* Run SNOW 3G algorithm to generate sequence of key stream bits KS*/
  Initialize(K,IV);

  KS = (u32 *)malloc(4*n);
  GenerateKeystream(n,(u32*)KS);

  /* Exclusive-OR the input data with keystream to generate the output bit
stream */
  for (i=0; i<n; i++)
  {
    data[4*i+0] ^= (u8) (KS[i] >> 24) & 0xff;
    data[4*i+1] ^= (u8) (KS[i] >> 16) & 0xff;
    data[4*i+2] ^= (u8) (KS[i] >>  8) & 0xff;
    data[4*i+3] ^= (u8) (KS[i]      ) & 0xff;
  }

  free(KS);
}

/* End of f8.c */
```

4.2. UIAII

4.2.1 Header File

```
/*---------------------------------------------------------
 *                   f9.h
 *---------------------------------------------------------*/
#ifndef F9_H_
#define F9_H_

#include "SNOW_3G.h"

/* f9.
 * Input key: 128 bit Integrity Key.
 * Input count:32-bit Count, Frame dependent input.
 * Input fresh: 32-bit Random number.
 * Input dir:1 bit, direction of transmission (in the LSB).
 * Input data: length number of bits, input bit stream.
 * Input length: 64 bit Length, i.e., the number of bits to be MAC'd.
 * Output  : 32 bit block used as MAC
 * Generates 32-bit MAC using UIA2 algorithm as defined in Section 4.
 */

u8* f9( u8* key, u32 count, u32 fresh, u32 dir, u8 *data, u64 length);

#endif
```

4.2.2 Code

```
/*---------------------------------------------------------
 *                   f9.c
 *---------------------------------------------------------*/
#include "f9.h"
#include <stdio.h>
#include <math.h>
#include <string.h>


/* MUL64x.
 * Input V: a 64-bit input.
 * Input c: a 64-bit input.
 * Output : a 64-bit output.
 * A 64-bit memory is allocated which is to be freed by the calling
 * function.
 * See section 4.3.2 for details.
 */

u64 MUL64x(u64 V, u64 c)
{
   if ( V & 0x8000000000000000 )
    return (V << 1) ^ c;
   else
    return V << 1;

}

/* MUL64xPOW.
 * Input V: a 64-bit input.
 * Input i: a positive integer.
 * Input c: a 64-bit input.
 * Output : a 64-bit output.
 * A 64-bit memory is allocated which is to be freed by the calling
function.
 * See section 4.3.3 for details.
 */

u64 MUL64xPOW(u64 V, u8 i, u64 c)
{
   if ( i == 0)
    return V;
   else
    return MUL64x( MUL64xPOW(V,i-1,c) , c);
}

/* MUL64.
 * Input V: a 64-bit input.
 * Input P: a 64-bit input.
 * Input c: a 64-bit input.
 * Output : a 64-bit output.
 * A 64-bit memory is allocated which is to be freed by the calling
 * function.
 * See section 4.3.4 for details.
 */

u64 MUL64(u64 V, u64 P, u64 c)
{
   u64 result = 0;
   int i = 0;

   for ( i=0; i<64; i++)
   {
    if( ( P>>i ) & 0x1 )
       result ^= MUL64xPOW(V,i,c);
   }

   return result;
}

/* mask8bit.
 * Input n: an integer in 1-7.
 * Output : an 8 bit mask.
 * Prepares an 8 bit mask with required number of 1 bits on the MSB side.
 */
u8 mask8bit(int n)
{
  return 0xFF ^ ((1<<(8-n)) - 1);
}

/* f9.
 * Input key: 128 bit Integrity Key.
 * Input count:32-bit Count, Frame dependent input.
 * Input fresh: 32-bit Random number.
 * Input dir:1 bit, direction of transmission (in the LSB).
 * Input data: length number of bits, input bit stream.
 * Input length: 64 bit Length, i.e., the number of bits to be MAC'd.
 * Output  : 32 bit block used as MAC
 * Generates 32-bit MAC using UIA2 algorithm as defined in Section 4.
 */
u8* f9( u8* key, u32 count, u32 fresh, u32 dir, u8 *data, u64 length)
{
  u32 K[4],IV[4], z[5];
  u32 i=0,D;
  static u8 MAC_I[4] = {0,0,0,0}; /* static memory for the result */
  u64 EVAL;
  u64 V;
  u64 P;
  u64 Q;
  u64 c;

  u64 M_D_2;
  int rem_bits = 0;

  /* Load the Integrity Key for SNOW3G initialization as in section 4.4. */
  for (i=0; i<4; i++)
    K[3-i] = (key[4*i] << 24) ^ (key[4*i+1] << 16) ^ (key[4*i+2] << 8) ^
(key[4*i+3]);

  /* Prepare the Initialization Vector (IV) for SNOW3G initialization as in
section 4.4. */
  IV[3] = count;
  IV[2] = fresh;
  IV[1] = count ^ ( dir << 31 ) ;
  IV[0] = fresh ^ (dir << 15);

  z[0] = z[1] = z[2] = z[3] = z[4] = 0;

  /* Run SNOW 3G to produce 5 keystream words z_1, z_2, z_3, z_4 and z_5. */
  Initialize(K,IV);
  GenerateKeystream(5,z);


  P = (u64)z[0] << 32 | (u64)z[1];
  Q = (u64)z[2] << 32 | (u64)z[3];

  /* Calculation */

  if ((length % 64) == 0)
    D = (length>>6) + 1;
  else
    D = (length>>6) + 2;
  EVAL = 0;

  c = 0x1b;


  /* for 0 <= i <= D-3 */
  for (i=0;i<D-2;i++)
  {
     V = EVAL ^ ( (u64)data[8*i  ]<<56 | (u64)data[8*i+1]<<48 |
(u64)data[8*i+2]<<40 | (u64)data[8*i+3]<<32 |
                  (u64)data[8*i+4]<<24 | (u64)data[8*i+5]<<16 |
(u64)data[8*i+6]<< 8 | (u64)data[8*i+7] );
     EVAL = MUL64(V,P,c);
  }

  /* for D-2 */
  rem_bits = length % 64;
  if (rem_bits == 0)
     rem_bits = 64;

  M_D_2 = 0;
  i = 0;
  while (rem_bits > 7)
  {
    M_D_2 |= (u64)data[8*(D-2)+i] << (8*(7-i));
    rem_bits -= 8;
    i++;
  }
  if (rem_bits > 0)
    M_D_2 |= (u64)(data[8*(D-2)+i] & mask8bit(rem_bits)) << (8*(7-i));

  V = EVAL ^ M_D_2;
  EVAL = MUL64(V,P,c);

  /* for D-1 */
  EVAL ^= length;
  /* Multiply by Q */
  EVAL = MUL64(EVAL,Q,c);


  for (i=0; i<4; i++)
    MAC_I[i] = (mac32 >> (8*(3-i))) & 0xff;

  return MAC_I;
}

/* End of f9.c */

/*------------------------------------------------------------------------*/
```