Contents

Foreword ..................................................................................................................................................4

0 Scope ..................................................................................................................................................5

NORMATIVE SECTION ............................................................................................................................5
1 Outline of the normative part ...............................................................................................................5
1.1 References ......................................................................................................................................5

2 Introductory information ..................................................................................................................6
2.1 Introduction ....................................................................................................................................6
2.1.1 Radix ..........................................................................................................................................6
2.1.2 Conventions ............................................................................................................................6
2.1.3 Bit/Byte ordering .....................................................................................................................7
2.1.4 List of Symbols .......................................................................................................................7
2.2 List of Variables ...........................................................................................................................7

3 Core function KGCORE ..................................................................................................................8
3.1 Introduction ....................................................................................................................................8
3.2 Inputs and Outputs .......................................................................................................................8
3.3 Components and Architecture .....................................................................................................9
3.4 Initialisation ...................................................................................................................................9
3.5 Keystream Generation ..................................................................................................................9

4 A5/3 algorithm for GSM encryption .............................................................................................10
4.1 Introduction ...................................................................................................................................10
4.2 Inputs and Outputs .......................................................................................................................10
4.3 Function Definition .....................................................................................................................10

5 A5/3 algorithm for ECSD encryption .............................................................................................11
5.1 Introduction ...................................................................................................................................11
5.2 Inputs and Outputs .......................................................................................................................11
5.3 Function Definition .....................................................................................................................11

6 GEA3 algorithm for GPRS encryption ...........................................................................................12
6.1 Introduction ...................................................................................................................................12
6.2 Inputs and Outputs .......................................................................................................................12
6.3 Function Definition .....................................................................................................................12

INFORMATIVE SECTION ....................................................................................................................13

A.1 Introduction ....................................................................................................................................14
A.2 Inputs and Outputs .......................................................................................................................14
A.3 Function Definition .....................................................................................................................14

Annex B (informative): Figures of the Algorithms .............................................................................16

Annex C (informative): Simulation Program Listings ........................................................................20

Annex D (informative): Change history .............................................................................................27
Foreword

This Technical Specification has been produced by the 3rd Generation Partnership Project (3GPP).

The contents of the present document are subject to continuing work within the TSG and may change following formal TSG approval. Should the TSG modify the contents of the present document, it will be re-released by the TSG with an identifying change of release date and an increase in version number as follows:

Version x.y.z

where:

x  the first digit:
   1  presented to TSG for information;
   2  presented to TSG for approval;
   3  or greater indicates TSG approved document under change control.

y  the second digit is incremented for all changes of substance, i.e. technical enhancements, corrections, updates, etc.

z  the third digit is incremented when editorial only changes have been incorporated in the document.
0 Scope

This specification has been prepared by the 3GPP Task Force, and gives a detailed specification of the A5/3 encryption algorithms for GSM and ECSD, and of the GEA3 encryption algorithm for GPRS.

This document is the first of three, which between them form the entire specification of the A5/3 and GEA3 algorithms:


The normative part of the specification of the A5/3 and GEA3 algorithms is in the main body of this document. The annexes to this document are purely informative. Annex A gives a specification of the 3GPP f8 confidentiality algorithm, bringing out the (deliberate) commonality between it and the A5/3 and GEA3 algorithms. Annex B contains illustrations of functional elements of the algorithms, while Annex C contains an implementation program listing of the cryptographic algorithms specified in the main body of this document, written in the programming language C.

Documents 2 and 3 above are also purely informative.

The normative part of the specification of the block cipher (KASUMI) on which the A5/3 and GEA3 algorithms are based can be found in TS 35.202 [5].

NORMATIVE SECTION

This part of the document contains the normative specifications of the A5/3 and GEA3 encryption algorithms.

1 Outline of the normative part

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

Section 3 specifies a core function KGCORE.

Section 4 specifies the encryption algorithm A5/3 for GSM in terms of the function KGCORE.

Section 5 specifies the encryption algorithm A5/3 for ECSD in terms of the function KGCORE.

Section 6 specifies the encryption algorithm GEA3 for GPRS in terms of the function KGCORE.

1.1 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication, edition number, version number, etc.) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies. In the case of a reference to a 3GPP document (including a GSM document), a non-specific reference implicitly refers to the latest version of that document in the same Release as the present document.
2 Introductory information

2.1 Introduction

In this document are specified three ciphering algorithms: \texttt{A5/3} for GSM, \texttt{A5/3} for ECSD, and \texttt{GEA3} for GPRS (including EGPRS). The algorithms are stream ciphers that are used to encrypt/decrypt blocks of data under a confidentiality key \(K_c\). Each of these algorithms is based on the \texttt{KASUMI} algorithm that is specified in reference TS 35.202 [5]. \texttt{KASUMI} is a block cipher that produces a 64-bit output from a 64-bit input under the control of a 128-bit key. The algorithms defined here use \texttt{KASUMI} in a form of output-feedback mode as a keystream generator.

The three algorithms are all very similar. We first define a core keystream generator function \texttt{KGCORE} (section 3); we then specify each of the three algorithms in turn (sections 4, 5 and 6) in terms of this core function.

2.1 Notation

2.1.1 Radix

We use the prefix \(0x\) to indicate \texttt{hexadecimal} numbers.

2.1.2 Conventions

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

\[
<\text{variable}> = <\text{expression}>
\]

we mean that \(<\text{variable}>\) assumes the value that \(<\text{expression}>\) 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.

3GPP
2.1.3 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 STRING is subdivided into 64-bit substrings $SB_0, SB_1, ... SB_n$ so if we have a string:

$$0x0123456789ABCDDEFFCBA98765432108654381AB594FC28786404C50A37...$$

we have:

- $SB_0 = 0x0123456789ABCDEF$
- $SB_1 = 0xFE9876543210$
- $SB_2 = 0x8654381AB594FC2$
- $SB_3 = 0x8786404C50A37...$

In binary this would be:

```
00000001011000101011001100101111000100110101111011111111110...
```

with $SB_0 = 0000000101100010101100110010111000100110101111011111111110...

- $SB_1 = 1111111101111101111101111111001101001100011010110101100001100100001000001000000100100010100110110101100100111111111111110$
- $SB_2 = 1000011000100011000101001101100000110101101010010100100111011110110010000110010000100000100100010100110111...$

2.1.4 List of Symbols

- $=$ The assignment operator.
- $\oplus$ The bitwise exclusive-OR operation
- $\|$ The concatenation of the two operands.
- $KASUMI[x]_k$ The output of the $KASUMI$ algorithm applied to input value $x$ using the key $k$.
- $X[i]$ The $i^{th}$ bit of the variable $X$. ($X = X[0] \| X[1] \| X[2] \| ...$).
- $Y[i]$ The $i^{th}$ octet of the variable $Y$. ($Y = Y[0] \| Y[1] \| Y[2] \| ...$).
- $Z_i$ The $i^{th}$ 64-bit block of the variable $Z$. ($Z = Z_0 \| Z_1 \| Z_2 \| ...$).

2.3 List of Variables

- $A$ a 64-bit register that is used within the $KGCORE$ function to hold an intermediate value.
- $BLKCNT$ a 64-bit counter used in the $KGCORE$ function.
- $BLOCK1$ a string of keystream bits output by the $A5/3$ algorithm — 114 bits for GSM, 348 bits for ECSD.
- $BLOCK2$ a string of keystream bits output by the $A5/3$ algorithm — 114 bits for GSM, 348 bits for ECSD.
- $BLOCKS$ an integer variable indicating the number of successive applications of $KASUMI$ that need to be performed.
- $CA$ an 8-bit input to the $KGCORE$ function.
- $CB$ a 5-bit input to the $KGCORE$ function.
- $CC$ a 32-bit input to the $KGCORE$ function.
- $CD$ a 1-bit input to the $KGCORE$ function.
CE a 16-bit input to the **KGCORE** function.

CK a 128-bit input to the **KGCORE** function.

CL an integer input to the **KGCORE** function, in the range \(1 \ldots 2^{19}\) inclusive, specifying the number of output bits for **KGCORE** to produce.

CO the output bitstream (CL bits) from the **KGCORE** function.

COUNT a 22-bit frame dependent input to both the GSM and ECSD **A5/3** algorithms.

DIRECTION a 1-bit input to the **GEA3** algorithm, indicating the direction of transmission (uplink or downlink).

INPUT a 32-bit frame dependent input to the **GEA3** algorithm.

**K_C** the cipher key that is an input to each of the three cipher algorithms defined here. Although at the time of writing the standards specify that **K_C** is 64 bits long, the algorithm specifications here allow it to be of any length between 64 and 128 inclusive, to allow for possible future enhancements to the standards.

KLEN the length of **K_C** in bits, between 64 and 128 inclusive (see above).

KM a 128-bit constant that is used to modify a key. This is used in the **KGCORE** function.

KS\[i\] the \(i^{th}\) bit of keystream produced by the keystream generator in the **KGCORE** function.

KSB\[i\] the \(i^{th}\) block of keystream produced by the keystream generator in the **KGCORE** function. Each block of keystream comprises 64 bits.

M an input to the **GEA3** algorithm, specifying the number of octets of output to produce.

OUTPUT the stream of output octets from the **GEA3** algorithm.

### 3 Core function **KGCORE**

#### 3.1 Introduction

In this section we define a general-purpose keystream generation function **KGCORE**. The individual encryption algorithms for GSM, GPRS and ECSD will each be defined in subsequent sections by mapping the relevant inputs to the inputs of **KGCORE**, and mapping the output of **KGCORE** to the relevant output.

#### 3.2 Inputs and Outputs

The inputs to **KGCORE** are given in table 1, the output in table 2:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>8 bits CA[0]...CA[7]</td>
</tr>
<tr>
<td>CB</td>
<td>5 bits CB[0]...CB[4]</td>
</tr>
<tr>
<td>CC</td>
<td>32 bits CC[0]...CC[31]</td>
</tr>
<tr>
<td>CD</td>
<td>A single bit CD[0]</td>
</tr>
<tr>
<td>CE</td>
<td>16 bits CE[0]...CE[15] (see Note 1 below)</td>
</tr>
<tr>
<td>CK</td>
<td>128 bits CK[0]...CK[127]</td>
</tr>
<tr>
<td>CL</td>
<td>An integer in the range (1 \ldots 2^{19}) inclusive, specifying the number of output bits to produce</td>
</tr>
</tbody>
</table>
Table 2: KGCORE output

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CO</td>
<td>CL bits CO[0]…CO[CL-1]</td>
</tr>
</tbody>
</table>

NOTE 1: All the algorithms specified in this document assign a constant, all-zeroes value to CE. More general use of CE is, however, available for possible future uses of KGCORE.

### 3.3 Components and Architecture

(See figure B.1, Annex B).

The function KGCORE is based on the block cipher KASUMI that is specified in TS 35.202 [5]. KASUMI is used in a form of output-feedback mode and generates the output bitstream in multiples of 64 bits.

The feedback data is modified by static data held in a 64-bit register A, and an (incrementing) 64-bit counter BLKCNT.

### 3.4 Initialisation

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

We set the 64-bit register A to CC || CB || CD || 0 0 || CA || CE

i.e. $A = CC[0]…CC[31] CB[0]…CB[4] CD[0] 0 0 CA[0]…CA[7] CE[0]…CE[15]$

We set the key modifier KM to 0x55555555555555555555555555555555

We set KSB0 to zero.

One operation of KASUMI is then applied to the register A, using a modified version of the confidentiality key.

$$A = KASUMI[A] \oplus KM$$

### 3.5 Keystream Generation

Once the keystream generator has been initialised in the manner defined in section 3.4, it is ready to be used to generate keystream bits. The keystream generator produces bits in blocks of 64 at a time, but the number CL of output bits to produce may not be a multiple of 64; between 0 and 63 of the least significant bits are therefore discarded from the last block, depending on the total number of bits specified by CL.

So let BLOCKS be equal to (CL/64) rounded up to the nearest integer. (For instance, if CL = 128 then BLOCKS = 2; if CL = 129 then BLOCKS = 3.)

To generate each keystream block (KSB) we perform the following operation:

For each integer $n$ with $1 \leq n \leq$ BLOCKS we define:

$$KSB_n = KASUMI[A \oplus BLKCNT \oplus KSB_{n-1}]_{CK}$$

where $BLKCNT = n-1$

The individual bits of the output are extracted from $KSB_1$ to $KSB_{BLOCKS}$ in turn, most significant bit first, by applying the operation:

For $n = 1$ to BLOCKS, and for each integer $i$ with $0 \leq i \leq 63$ we define:

$$CO((n-1)*64)+i = KSB_n[i]$$
4 A5/3 algorithm for GSM encryption

4.1 Introduction

The GSM A5/3 algorithm produces two 114-bit keystream strings, one of which is used for uplink encryption/decryption and the other for downlink encryption/decryption.

We define this algorithm in terms of the core function KGCORE.

4.2 Inputs and Outputs

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bits)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>22</td>
<td>Frame dependent input COUNT[0]…COUNT[21]</td>
</tr>
<tr>
<td>Kc</td>
<td>KLEN</td>
<td>Cipher key Kc[0]…Kc[KLEN-1], where KLEN is in the range 64…128 inclusive (see Notes 1 and 2 below)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bits)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCK1</td>
<td>114</td>
<td>Keystream bits BLOCK1[0]…BLOCK1[113]</td>
</tr>
<tr>
<td>BLOCK2</td>
<td>114</td>
<td>Keystream bits BLOCK2[0]…BLOCK2[113]</td>
</tr>
</tbody>
</table>

NOTE 1: The specification of the A5/3 algorithm only allows KLEN to be of value 64.
NOTE 2: It must be assumed that Kc is unstructured data — it must not be assumed, for instance, that any bits of Kc have predetermined values.

4.3 Function Definition

(See figure B.2, Annex B).

We define the function by mapping the GSM A5/3 inputs onto the inputs of the core function KGCORE, and mapping the output of KGCORE onto the outputs of GSM A5/3.

So we define:

CA[0]…CA[7] = 0 0 0 0 1 1 1 1
CB[0]…CB[4] = 0 0 0 0 0
CC[0]…CC[9] = 0 0 0 0 0 0 0 0 0 0
CC[10]…CC[31] = COUNT[0]…COUNT[21]
CD[0] = 0
CE[0]…CE[15] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CK[0]…CK[KLEN-1] = Kc[0]…Kc[KLEN-1]
If KLEN < 128 then

CK[KLEN]…CK[127] = Kc[0]…Kc[127 – KLEN]

(So in particular if KLEN = 64 then CK = Kc || Kc)

CL = 228
Apply **KGCORE** to these inputs to derive the output CO[0]…CO[227].

Then define:

\[
\text{BLOCK1}[0]…\text{BLOCK1}[113] = \text{CO}[0]…\text{CO}[113] \\
\text{BLOCK2}[0]…\text{BLOCK2}[113] = \text{CO}[114]…\text{CO}[227]
\]

### 5 A5/3 algorithm for ECSD encryption

#### 5.1 Introduction

The ECSD A5/3 algorithm produces two 348-bit keystream strings, one of which is used for uplink encryption/decryption and the other for downlink encryption/decryption.

We define this algorithm in terms of the core function **KGCORE**.

#### 5.2 Inputs and Outputs

The inputs to the algorithm are given in table 5, the output in table 6:

<table>
<thead>
<tr>
<th>Table 5: ECSD A5/3 inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>COUNT</td>
</tr>
<tr>
<td>(K_C)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 6: ECSD A5/3 outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter</td>
</tr>
<tr>
<td>BLOCK1</td>
</tr>
<tr>
<td>BLOCK2</td>
</tr>
</tbody>
</table>

**NOTE 1:** The specification of the A5/3 algorithm only allows \(KLEN\) to be of value 64.

**NOTE 2:** It must be assumed that \(K_C\) is unstructured data — it must not be assumed, for instance, that any bits of \(K_C\) have predetermined values.

#### 5.3 Function Definition

(See figure B.3, Annex B).

We define the function by mapping the ECSD A5/3 inputs onto the inputs of the core function **KGCORE**, and mapping the output of **KGCORE** onto the outputs of ECSD A5/3.

So we define:

\[
\begin{align*}
\text{CA}[0]…\text{CA}[7] &= 1 1 1 1 0 0 0 0 \\
\text{CB}[0]…\text{CB}[4] &= 0 0 0 0 0 \\
\text{CC}[0]…\text{CC}[9] &= 0 0 0 0 0 0 0 0 0 0 \\
\text{CC}[10]…\text{CC}[31] &= \text{COUNT}[0]…\text{COUNT}[21] \\
\text{CD}[0] &= 0 \\
\text{CE}[0]…\text{CE}[15] &= 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 
\end{align*}
\]
CK[0]…CK[KLEN-1] = Kc[0]…Kc[KLEN-1]

If KLEN < 128 then

CK[KLEN]…CK[127] = Kc[0]…Kc[127 – KLEN]

(So in particular if KLEN = 64 then CK = Kc || Kc)

CL = 696

Apply KGCORE to these inputs to derive the output CO[0]…CO[695].

Then define:

BLOCK1[0]…BLOCK1[347] = CO[0]…CO[347]

BLOCK2[0]…BLOCK2[347] = CO[348]…CO[695]

### 6 GEA3 algorithm for GPRS encryption

#### 6.1 Introduction

The GPRS GEA3 algorithm produces an M-byte keystream string. M can vary; in this specification we assume that M will never exceed $2^{16} = 65536$.

We define this algorithm in terms of the core function KGCORE.

#### 6.2 Inputs and Outputs

The inputs to the algorithm are given in table 7, the output in table 8:

**Table 7: GEA3 inputs**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bits)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>32</td>
<td>Frame dependent input INPUT[0]...INPUT[31]</td>
</tr>
<tr>
<td>DIRECTION</td>
<td>1</td>
<td>Direction of transmission indicator DIRECTION[0]</td>
</tr>
<tr>
<td>Kc</td>
<td>KLEN</td>
<td>Cipher key Kc[0]...Kc[KLEN-1], where KLEN is in the range 64...128 inclusive (see Notes 1 and 2 below)</td>
</tr>
<tr>
<td>M</td>
<td></td>
<td>Number of octets of output required, in the range 1 to 65536 inclusive</td>
</tr>
</tbody>
</table>

**Table 8: GEA3 outputs**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bits)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUTPUT</td>
<td>8M</td>
<td>Keystream octets OUTPUT(0)...OUTPUT(M-1)</td>
</tr>
</tbody>
</table>

NOTE 1: The specification of the GEA3 algorithm only allows KLEN to be of value 64.

NOTE 2: It must be assumed that Kc is unstructured data — it must not be assumed, for instance, that any bits of Kc have predetermined values.

#### 6.3 Function Definition

(See figure B.4, Annex B).

We define the function by mapping the GEA3 inputs onto the inputs of the core function KGCORE, and mapping the output of KGCORE onto the outputs of GEA3.

So we define:
CA[0]...CA[7] = 1 1 1 1 1 1 1 1
CB[0]...CB[4] = 0 0 0 0 0
CC[0]...CC[31] = INPUT[0]...INPUT[31]
CD[0] = DIRECTION[0]
CE[0]...CE[15] = 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
CK[0]...CK[KLEN-1] = K_c[0]...K_c[KLEN-1]

If KLEN < 128 then
CK[KLEN]...CK[127] = K_c[0]...K_c[127 – KLEN]

(So in particular if KLEN = 64 then CK = K_c || K_c)

CL = 8M

Apply KGCORE to these inputs to derive the output CO[0]...CO[8M-1].

Then for 0 ≤ i ≤ M-1 define:

OUTPUT[i] = CO[8i]...CO[8i + 7]

where CO[8i] is the most significant bit of the octet.

---

INFORMATIVE SECTION

This part of the document is purely informative and does not form part of the normative specification of A5/3 and GEA3.
Annex A (informative): Specification of the 3GPP Confidentiality Algorithm \( f_8 \)

A.1 Introduction

The algorithms defined in this specification have been designed to have much in common with the 3GPP confidentiality algorithm, to ease simultaneous implementation of multiple algorithms. To clarify this, a specification of \( f_8 \) is given here in terms of the core function \( \text{KG} \text{CORE} \). For the definitive specification of \( f_8 \), the reader is referred to TS 35.202 [5].

A.2 Inputs and Outputs

The inputs to the algorithm are given in table A.1, the output in table A.2:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bits)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>32</td>
<td>Frame dependent input COUNT[0]…COUNT[31]</td>
</tr>
<tr>
<td>BEARER</td>
<td>5</td>
<td>Bearer identity BEARER[0]…BEARER[4]</td>
</tr>
<tr>
<td>DIRECTION</td>
<td>1</td>
<td>Direction of transmission DIRECTION[0]</td>
</tr>
<tr>
<td>CK</td>
<td>128</td>
<td>Confidentiality key CK[0]…CK[127]</td>
</tr>
<tr>
<td>LENGTH</td>
<td></td>
<td>The number of bits to be encrypted/decrypted (1-20000)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Size (bits)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>KS</td>
<td>1-20000</td>
<td>Keystream bits KS[0]…KS[LENGTH-1]</td>
</tr>
</tbody>
</table>

NOTE: The definitive specification of \( f_8 \) includes a bitstream IBS amongst the inputs, and gives the output as a bitstream OBS, both of these bitstreams are \( \text{LENGTH} \) bits long. OBS is obtained by the bitwise exclusive-or of IBS and KS. We present just the keystream generator part of \( f_8 \) here, for closer comparison with \( A5/3 \) and GEA3.

A.3 Function Definition

(See figure 5, Annex B).

We define the function by mapping the \( f_8 \) inputs onto the inputs of the core function \( \text{KG} \text{CORE} \), and mapping the output of \( \text{KG} \text{CORE} \) onto the outputs of \( f_8 \).

So we define:

\[
\begin{align*}
CA[0]…CA[7] &= 00000000 \\
CC[0]…CC[31] &= COUNT[0]…COUNT[31] \\
CD[0] &= DIRECTION[0] \\
CE[0]…CE[15] &= 0000000000000000 \\
CK[0]…CK[127] &= CK[0]…CK[127] \\
CL &= LENGTH
\end{align*}
\]
Apply $\text{KGCORE}$ to these inputs to derive the output $\text{CO}[0]...\text{CO}[\text{LENGTH}-1]$.

Then define:

$$\text{KS}[0]...\text{KS}[\text{LENGTH}-1] = \text{CO}[0]...\text{CO}[\text{LENGTH}-1]$$
Annex B (informative):
Figures of the Algorithms

NOTE: BLKCNT is specified as a 64-bit counter so there is no ambiguity in the expression
A ⊕ BLKCNT ⊕ KSBₙ₋₁ where all operands are of the same size. In a practical implementation, where the
keystream generator is required to produce no more than a certain number of bits, only the least significant
few bits of the counter need to be realised.

Figure B.1: KGCORE Core Keystream Generator Function
Figure B.2: GSM A5/3 Keystream Generator Function

Figure B.3: ECSD A5/3 Keystream Generator Function
Figure B.4: GEA3 Keystream Generator Function

Figure B.5: 3GPP f8 Keystream Generator Function
Table B.1: GSM A5/3, ECSD A5/3, GEA3 and f8 in terms of KG CORE

<table>
<thead>
<tr>
<th></th>
<th>GSM A5/3</th>
<th>ECSD A5/3</th>
<th>GEA3</th>
<th>f8</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA</td>
<td>0 0 0 1 1 1 1</td>
<td>1 1 1 0 0 0 0</td>
<td>1 1 1 1 1 1 1</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>CB</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
<td>BEARER</td>
</tr>
<tr>
<td>CC</td>
<td>0...0</td>
<td></td>
<td>COUNT</td>
<td>0...0</td>
</tr>
<tr>
<td>CD</td>
<td>0</td>
<td>0</td>
<td>DIRECTION</td>
<td>DIRECTION</td>
</tr>
<tr>
<td>CE</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CK</td>
<td>KC repeated to fill 128 bits</td>
<td>KC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>BLOCK1</td>
<td></td>
<td>BLOCK2</td>
<td>BLOCK1</td>
</tr>
</tbody>
</table>
Annex C (informative):
Simulation Program Listings

kasumi.h

/*----------------------------- Kasumi.h ------------------------------------*/
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned int u32;
/*----- a 64-bit structure to help with endian issues -----*/
typedef union {
    u32 b32[2];
    u16 b16[4];
    u8  b8[8];
} REGISTER64;
/*------------- prototypes --------------------------------*/
void KeySchedule(u8 *key);
void Kasumi(u8 *data);

kasumi.c

/*----------------------------- Kasumi.c ------------------------------------*/
/* A sample implementation of KASUMI, the core algorithm for the
* 3GPP Confidentiality and Integrity algorithms.
* This has been coded for clarity, not necessarily for efficiency.
* This will compile and run correctly on both Intel (little endian)
* and Sparc (big endian) machines. (Compilers used supported 32-bit ints).
* Version 1.1 08 May 2000*/

#include "Kasumi.h"

/*--------- 16 bit rotate left ------------------------------------------*/
#define ROL16(a,b) (u16)((a<<b)|(a>>(16-b)))
/*------- unions: used to remove "endian" issues ------------------------*/
typedef union {
    u32 b32;
    u16 b16[2];
    u8  b8[4];
} DWORD;
typedef union {
    u16 b16;
    u8  b8[2];
} WORD;
/*-------- globals: The subkey arrays -----------------------------------*/
static u16 KLi1[8], KLi2[8];
static u16 KOi1[8], KOi2[8], KOi3[8];
static u16 KIi1[8], KIi2[8], KIi3[8];
/*----------------------------- FI() ------------------------------------*
* The FI function (fig 3). It includes the S7 and S9 tables.*/

3GPP
* Transforms a 16-bit value.

static u16 FI( u16 in, u16 subkey )
{
    u16 nine, seven;
    static u16 S7[] = {
        54, 50, 62, 56, 23, 47, 59, 79, 55, 55, 53, 9, 123, 33, 91, 49,
        113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113,
        113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113,
        113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113,
        113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113,
        113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113, 113,
    };
    static u16 S9[] = {
        167, 239, 161, 379, 393, 327, 324, 24, 456, 267, 157, 460, 488, 426, 309, 229,
        439, 506, 208, 271, 349, 401, 434, 236, 16, 209, 359, 52, 56, 120, 199, 277,
        465, 416, 252, 287, 246, 63, 83, 305, 420, 345, 153, 502, 65, 61, 244, 282,
        280, 383, 373, 128, 382, 408, 155, 495, 367, 388, 274, 107, 459, 417, 62, 454,
        132, 225, 203, 316, 234, 14, 301, 91, 503, 286, 424, 211, 347, 307, 140, 374,
        35, 103, 125, 427, 19, 214, 453, 146, 498, 314, 444, 230, 256, 329, 198, 285,
        50, 116, 78, 410, 10, 205, 510, 171, 231, 45, 139, 467, 29, 86, 505, 32,
        7, 494, 131, 254, 490, 431, 238, 411, 325, 149, 473, 40, 119, 174, 355,
        185, 233, 389, 71, 448, 273, 372, 55, 110, 178, 322, 12, 469, 392, 369, 190,
        1, 109, 375, 137, 181, 88, 75, 308, 260, 484, 98, 272, 370, 275, 412, 111,
        336, 318, 4, 504, 492, 259, 304, 77, 337, 435, 21, 357, 303, 332, 483, 18,
        47, 85, 25, 497, 474, 289, 100, 269, 296, 478, 270, 106, 31, 104, 433, 84,
        414, 486, 394, 96, 99, 154, 511, 148, 413, 363, 409, 255, 162, 215, 302, 201,
        266, 351, 343, 144, 441, 365, 108, 298, 251, 34, 182, 509, 138, 210, 353, 135,
        311, 352, 328, 141, 396, 346, 123, 319, 450, 281, 429, 228, 443, 481, 92, 404,
        485, 422, 248, 297, 23, 213, 130, 466, 22, 217, 283, 70, 29, 360, 419, 127,
        322, 377, 7, 468, 194, 2, 117, 295, 463, 258, 224, 447, 247, 187, 80, 398,
        284, 353, 105, 390, 299, 471, 470, 184, 57, 200, 348, 63, 204, 188, 33, 451,
        97, 30, 310, 219, 94, 160, 129, 493, 64, 179, 263, 102, 189, 207, 114, 402,
        438, 477, 387, 122, 192, 42, 381, 5, 145, 118, 180, 449, 293, 323, 136, 380,
        43, 66, 60, 455, 341, 445, 202, 432, 8, 237, 15, 376, 436, 464, 59, 461};

/* The sixteen bit input is split into two unequal halves, */
* nine bits and seven bits – as is the subkey */  
nine = (u16)(in>>7);
seven = (u16)(in&0x7F);

/* Now run the various operations */

nine = (u16)[S9[nine]  ^ seven];
seven = (u16)[S7[seven] ^ (nine & 0x7F)];

seven ^= (subkey>>9);
nine ^= (subkey&0x1FF);
nine = (u16)[S9[nine]  ^ seven];
seven = (u16)[S7[seven] ^ (nine & 0x7F)];
in = (u16)((seven<<9) + nine);

return( in );


static u32 FO( u32 in, int index )
{  

/* This FO() function. */
* Transforms a 32-bit value. Uses <index> to identify the
* appropriate subkeys to use. */

3GPP
u16 left, right;

/* Split the input into two 16-bit words */
left = (u16)(in>>16);
right = (u16) in;

/* Now apply the same basic transformation three times */
left ^= KOi1[index];
left = FI( left, KIi1[index] );
left ^= right;
right ^= KOi2[index];
right = FI( right, KIi2[index] );
right ^= left;
left ^= KOi3[index];
left = FI( left, KIi3[index] );
left ^= right;

in = (((u32)right)<<16)+left;
return( in );
}

/*---------------------------------------------------------------------
* FL()
* The FL() function.
* Transforms a 32-bit value. Uses <index> to identify the
* appropriate subkeys to use.
*---------------------------------------------------------------------*/
static u32 FL( u32 in, int index )
{
    u16 l, r, a, b;

    /* split out the left and right halves */
l = (u16)(in>>16);
r = (u16)(in);

    /* do the FL() operations */
a = (u16) (l & KLi1[index]);
r ^= ROL16(a,1);
b = (u16)(r | KLi2[index]);
l ^= ROL16(b,1);

    /* put the two halves back together */
in = (((u32)l)<<16) + r;
return( in );
}

/*---------------------------------------------------------------------
* Kasumi()
* the Main algorithm (fig 1). Apply the same pair of operations
* four times. Transforms the 64-bit input.
*---------------------------------------------------------------------*/
void Kasumi( u8 *data )
{
    u32 left, right, temp;
    DWORD *d;
    int n;

    /* Start by getting the data into two 32-bit words (endian correct) */
    d = (DWORD*)data;
    left  = (((u32)d[0].b8[0])<<24)+(((u32)d[0].b8[1])<<16)+
    +((u32)d[0].b8[2]<<8)+((u32)d[0].b8[3]));
    right = (((u32)d[1].b8[0])<<24)+(((u32)d[1].b8[1])<<16)+
    n = 0;
do{
    temp = FL( left, n );
    temp = FO( temp, n++ );
    right ^= temp;

    temp = FL( right, n );
    temp = FO( temp, n++ );
    left ^= temp;

    /* Finish by combining the two halves together */
in = (((u32)right)<<16)+left;
return( in );
}
temp = FO( right, n );
left ^= temp;
}while( n<=7 );

/* return the correct endian result */
d[0].b8[0] = (u8)(left>>24);  d[1].b8[0] = (u8)(right>>24);
d[0].b8[1] = (u8)(left>>16);  d[1].b8[1] = (u8)(right>>16);
d[0].b8[2] = (u8)(left>>8);    d[1].b8[2] = (u8)(right>>8);
d[0].b8[3] = (u8)(left);      d[1].b8[3] = (u8)(right);
*/

/*---------------------------------------------------------------------
* KeySchedule()
* Build the key schedule. Most "key" operations use 16-bit
* subkeys so we build u16-sized arrays that are "endian" correct.
*---------------------------------------------------------------------*/
void KeySchedule( u8 *k )
{
    static u16 C[] = {0x0123,0x4567,0x89AB,0xCDEF,0xFEDC,0xBA98,0x7654,0x3210};
    u16 key[8], Kprime[8];
    WORD *k16;
    int n;

    /* Start by ensuring the subkeys are endian correct on a 16-bit basis */
    k16 = (WORD *)k;
    for( n=0; n<8; ++n )
        key[n] = (u16)((k16[n].b8[0]<<8) + (k16[n].b8[1]));

    /* Now build the K'[] keys */
    for( n=0; n<8; ++n )
        Kprime[n] = (u16)(key[n] ^ C[n]);

    /* Finally construct the various sub keys */
    for( n=0; n<8; ++n )
    {
        KLi1[n] = ROL16(key[n],1);
        KLi2[n] = Kprime[(n+2)&0x7];
        KOi1[n] = ROL16(key[(n+1)&0x7],5);
        KOi2[n] = ROL16(key[(n+5)&0x7],8);
        KOi3[n] = ROL16(key[(n+6)&0x7],13);
        KLi[n] = Kprime[(n+4)&0x7];
        KLi2[n] = Kprime[(n+3)&0x7];
        KLi3[n] = Kprime[(n+7)&0x7];
    }
}

/*---------------------------------------------------------------------
* end of kasumi.c
*---------------------------------------------------------------------*/

kgcore.c

/*---------------------------------------------------------------------
* KGcore
*---------------------------------------------------------------------*/

/* A sample implementation of KGcore, the heart of the
 * A5/3 algorithm set.
 * This has been coded for clarity, not necessarily for
 * efficiency.
 * This will compile and run correctly on both Intel
 * (little endian) and Sparc (big endian) machines.
 * Version 0.1  13 March 2002
 *---------------------------------------------------------------------*/
#include "kasumi.h"
#include <stdio.h>
```c
void KGcore( u8 ca, u8 cb, u32 cc, u8 cd, u8 *ck, u8 *co, int cl ) { 
  REGISTER64 A;  /* the modifier */
  REGISTER64 temp; /* The working register */
  int i, n;
  u8 key[16],ModKey[16];  /* Modified key */
  u16 blkcnt;  /* The block counter */

  /* Copy over the key */
  for( i=0; i<16; ++i )
    key[i] = ck[i];

  /* Start by building our global modifier */
  temp.b32[0]  = temp.b32[1]  = 0;
  A.b32[0]     = A.b32[1]     = 0;

  /* initialise register in an endian correct manner*/
  A.b8[0]  = (u8) (cc>>24);
  A.b8[1]  = (u8) (cc>>16);
  A.b8[2]  = (u8) (cc>>8);
  A.b8[3]  = (u8) (cc);
  A.b8[4]  = (u8) (cb<<3);
  A.b8[4] |= (u8) (cd<<2);
  A.b8[5]  = (u8) ca;

  /* Construct the modified key and then "kasumi" A */
  for( n=0; n<16; ++n )
    ModKey[n] = (u8)(ck[n] ^ 0x55);
  KeySchedule( ModKey );
  Kasumi( A.b8 ); /* First encryption to create modifier */
  /* Final initialisation steps */
  blkcnt = 0;
  KeySchedule( key );

  /* Now run the key stream generator */
  while( cl > 0 )
  {
    /* First we calculate the next 64-bits of keystream */
    /* XOR in A and BLKCNT to last value */
    temp.b32[0] ^= A.b32[0];
    temp.b32[1] ^= A.b32[1];
    temp.b8[7] ^= blkcnt;

    /* KASUMI it to produce the next block of keystream */
    Kasumi( temp.b8 );

    /* Set <n> to the number of bytes of input data *
    * we have to modify. (=8 if length <= 64) */
    if( cl >= 64 )
      n = 8;
    else
      n = (cl+7)/8;

    /* copy out the keystream */
    for( i=0; i<n; ++i )
      *co++ = temp.b8[i];
    cl -= 64;  /* done another 64 bits */
    ++blkcnt;  /* increment BLKCNT */
  }
}
/* KGcore()*/
*/ Given ca, cb, cc, cd, ck, cl generate c0 */
/*---------------------------------------------------------*/
*/ KGcore() */
*/ Given ca, cb, cc, cd, ck, cl generate c0 */
/*---------------------------------------------------------*/
*/ KGcore() */
*/ Given ca, cb, cc, cd, ck, cl generate c0 */
/*---------------------------------------------------------*/
A5/3

/*---------------------------------------------*
 * A5/3                                               *
 * A sample implementation of A5/3, the functions of the *
 * A5/3 algorithm set.                                  *
 * This has been coded for clarity, not necessarily for *
 * efficiency.                                          *
 * This will compile and run correctly on both Intel    *
 * (little endian) and Sparc (big endian) machines.      *
 * Version 0.1 13 March 2002                           *
 /*---------------------------------------------*/

#include "kasumi.h"
#include <stdlib.h>

void KGcore( u8 ca, u8 cb, u32 cc, u8 cd, u8 *ck, u8 *co, int cl );

/*--------------------------------------------------------------------------*/
* BuildKey()                                                             *
* The KGcore() function expects a 128-bit key. This                     *
* function builds that key from shorter length keys.                    *
/*------------------------------------------------------------------------*/
static u8 *BuildKey( u8 *k, int len )
{
    static u8 ck[16];    /* Where the key is built */
    int i, n, sf;
    u8 mask[]={0x1,0x3,0x7,0xF,0x1F,0x3F,0x7F,0xFF};
    i = (len+7)/8;       /* Round to nearest byte */
    if (i > 16 ) i = 16;    /* limit to 128 bits */
    for( n=0; n<i; ++n ) /* copy over the key */
        ck[n] = k[n];
    sf = len%8;        /* Any odd key length? */
    if( len < 128 )
    {
        n = 0;
        if( sf ) /* Doesn't align to byte boundaries */
        {
            ck[i-1] &= mask[sf];
            ck[i-1] += ck[0]<<sf;
            while( i<16 )
            {
                ck[i] = (ck[n]>>(8-sf)) + (ck[n+1]<<sf);
                ++n;
                ++i;
            }
        } else
        {
            while( i<16 )
                ck[i++] = ck[n++];
        }
    }
    return( ck );
}

/*---------------------------------------------*/
* The basic A5/3 functions.                          *
* These follow a standard layout:                   *
* - From the supplied key build the 128-bit required by *
* KGcore()                                             *
* - Call the KGcore() function with the appropriate *
* parameters                                          */
/* - Take the generated Keystream and repackage it
 * in the required format.
 */

/*---------------------------------------------------------
* The standard GSM function
*---------------------------------------------------------*/
void GSM( u8 *key, int klen, int count, u8 *block1, u8 *block2 )
{
    u8 *ck, data[32];
    int i;
    ck=BuildKey( key, klen );
    KGcore( 0x0F, 0, count, 0, ck, data, 228 );
    for( i=0; i<15; ++i )
    {
        block1[i] = data[i];
        block2[i] = (data[i+14]<<2) + (data[i+15]>>6);
    }
    block1[14] &= 0xC0;
    block2[14] &= 0xC0;
}

/*---------------------------------------------------------
* The standard GSM ECSD function
*---------------------------------------------------------*/
void ECSD( u8 *key, int klen, int count, u8 *block1, u8 *block2 )
{
    u8 *ck, data[87];
    int i;
    ck=BuildKey( key, klen );
    KGcore( 0xF0, 0, count, 0, ck, data, 696 );
    for( i=0; i<44; ++i )
    {
        block1[i] = data[i];
        block2[i] = (data[i+43]<<4) + (data[i+44]>>4);
    }
    block1[43] &= 0xF0;
    block2[43] &= 0xF0;
}

/*---------------------------------------------------------
* The standard GEA3 function
*---------------------------------------------------------*/
void GEA3( u8 *key, int klen, u32 input, u8 direction, u8 *block, int m )
{
    u8 *ck, *data;
    int i;

    data = malloc( m );
    ck=BuildKey( key, klen );
    KGcore( 0xFF, 0, input, direction, ck, data, m*8 );
    for( i=0; i<m; ++i )
    {
        block[i] = data[i];
    }
    free( data );
}

/*---------------------------------------------------------
* End of A53f.c
*---------------------------------------------------------*/

a53f.h

void GSM( u8 *key, int klen, int count, u8 *block1, u8 *block2 );
void ECSD( u8 *key, int klen, int count, u8 *block1, u8 *block2 );
void GEA3( u8 *key, int klen, u32 input, u8 direction, u8 *block, int m );

3GPP
Annex D (informative):
Change history

<table>
<thead>
<tr>
<th>Date</th>
<th>TSG #</th>
<th>TSG Doc.</th>
<th>CR</th>
<th>Rev</th>
<th>Subject/Comment</th>
<th>Old</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002-05</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>ETSI SAGE first publication</td>
<td>SAGE V1.0</td>
<td></td>
</tr>
<tr>
<td>2002-07</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Agreed at SA WG3 #24 for presentation to TSG SA #17 for approval. Converted into 3GPP TS format as TS 55.216 (Technically equivalent to SAGE V1.0)</td>
<td>SAGE V1.0</td>
<td>1.0.0</td>
</tr>
<tr>
<td>2002-09</td>
<td>SP-17</td>
<td>SP-020506</td>
<td>-</td>
<td>-</td>
<td>Approved for Release 6 - version 6.0.0</td>
<td>1.0.0</td>
<td>6.0.0</td>
</tr>
<tr>
<td>2002-12</td>
<td>SP-18</td>
<td>SP-020721</td>
<td>001</td>
<td>-</td>
<td>EGPRS algoritm</td>
<td>6.0.0</td>
<td>6.1.0</td>
</tr>
<tr>
<td>2003-09</td>
<td>SP-19</td>
<td>SP-030490</td>
<td>002</td>
<td>-</td>
<td>Clarification on the usage of the Key length</td>
<td>6.1.0</td>
<td>6.2.0</td>
</tr>
</tbody>
</table>