ETSI TS 135 231 V12.1.0 (2014-10)

Universal Mobile Telecommunications System (UMTS);
LTE;
Specification of the TUAK algorithm set:
A second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*;
Document 1: Algorithm specification
(3GPP TS 35.231 version 12.1.0 Release 12)

Reference
DTS/TSGS-0335231vc10

Keywords
LTE, SECURITY, UMTS

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://ipr.etsi.org).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

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

The present document may refer to technical specifications or reports using their 3GPP identities, UMTS identities or GSM identities. These should be interpreted as being references to the corresponding ETSI deliverables.

The cross reference between GSM, UMTS, 3GPP and ETSI identities can be found under http://webapp.etsi.org/key/queryform.asp.

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**may not**", "**need**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Contents

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.

Introduction

The present document is one of three, which between them form the entire specification of the example algorithms, entitled:

- 3GPP TS 35.231: "Specification of the Tuak algorithm set: A second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*;
  Document 1: Algorithm specification ".

- 3GPP TS 35.232: "Specification of the Tuak algorithm set: A second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*;
  Document 2: Implementers" test data".

- 3GPP TS 35.233: "Specification of the Tuak algorithm Set: A second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*;
  Document 3: Design conformance test data".

1 Scope

The present document and the other Technical Specifications in the series, TS 35.232 [15] and 35.233 [16] contain an example set of algorithms which could be used as the authentication and key generation functions ***f1***, ***f1****, ***f2***, ***f3***, ***f4***, ***f5*** and ***f5**** for 3GPP systems. All seven functions are operator-specifiable rather than being fully standardised and other algorithms could be envisaged.

2 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*.

[1] 3GPP TS 33.102: "3G Security; Security Architecture3G Security; Specification of the MILENAGE algorithm set: An example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 2: Algorithm specification.[3] "The KECCAK Reference", version 3.0, 14 January 2011, G. Bertoni, J. Daemen, M. Peeters, G. van Aasche, (available at http://keccak.noekeon.org/Keccak-reference-3.0.pdf).

[4] "KECCAK Implementation Overview", version 3.2, 29 May 2012, G. Bertoni, J. Daemen, M. Peeters, G. van Aasche, R. van Keer (available at http://keccak.noekeon.org/Keccak-implementation-3.2.pdf).

[5] "SAKURA: a flexible coding for tree hashing", 3 June 2013, G. Bertoni, J. Daemen, M. Peeters, G. van Aasche, (available at http://keccak.noekeon.org/Sakura.pdf).

[6] "Securing the AES finalists against Power Analysis Attacks", in FSE 2000, Seventh Fast Software Encryption Workshop, Thomas S. Messerges, ed. Schneier, Springer Verlag, 2000.

[7] "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", P. C. Kocher, in CRYPTO'96, Lecture Notes in Computer Science #1109, Springer Verlag, 1996.

[8] "Side Channel Cryptanalysis of Product Ciphers", in ESORICS'98, Lecture Notes in Computer Science #1485, Springer Verlag, 1998, J. Kelsey, B. Schneier, D. Wagner, C. Hall.

[9] "DES and differential power analysis", in CHES'99, Lecture Notes in Computer Science #1717, Springer Verlag, 1999, L. Goubin, J. Patarin.

[10] "Differential Power Analysis", in CRYPTO'99, Lecture Notes in Computer Science #1666, Springer Verlag, 1999, P. Kocher, J. Jaffe, B. Jun.

[11] "On Boolean and Arithmetic Masking against Differential Power Analysis", in CHES'00, Lecture Notes in Computer Science series, Springer Verlag, 2000, L. Goubin, J.-S. Coron.

[12] 3GPP TS 33.401: "3GPP System Architecture Evolution (SAE); Security architecture".

[13] ETSI TS 103 383: "Smart Cards; Embedded UICC; Requirements Specification".

[14] 3GPP TR 21.905: "Vocabulary for 3GPP specifications".

[15] 3GPP TS 35.232: "3G Security; Specification of the Tuak Algorithm Set: a Second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 2: Implementers" test data".

[16] 3GPP TS 35.233: "3G Security; Specification of the Tuak Algorithm Set: a second example algorithm set for the 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5 and f5*; Document 3: Design conformance test data".

3 Definitions

3.1 Definitions

For the purposes of the present document, the terms and definitions given in TR 21.905 [14] and the following apply. A term defined in the present document takes precedence over the definition of the same term, if any, in TR 21.905 [14].

Tuak: The name of this algorithm set is "Tuak". It should be pronounced like "too-ack".

3.2 Symbols

=	The assignment operator
⊕	The bitwise exclusive-OR operation
||	The concatenation of the two operands
X[i]	The i^{th} bit of the variable **X**. (**X = X[0] || X[1] || X[2] || …..**)
Π	the permutation Keccak-f[1600] (See clause 5.2 and annex C)

The following represent variables used in the algorithm:

AK	a 48-bit anonymity key that is the output of either of the functions *f5* and *f5**
AMF	a 16-bit authentication management field that is an input to the functions *f1* and *f1**
CK	a 128-bit or 256-bit confidentiality key that is the output of the function *f3*
IK	a 128-bit or 256-bit integrity key that is the output of the function *f4*
IN	a 1600-bit value that is used as the input to the permutation Π when computing the functions *f1*, *f1**, *f2*, *f3*, *f4*, *f5* and *f5**
INSTANCE	an 8-bit value that is used to specify different modes of operation and different parameter lengths within the algorithm set
K	a 128-bit or 256-bit subscriber key that is an input to the functions *f1*, *f1**, *f2*, *f3*, *f4*, *f5* and *f5**
MAC-A	a 64-bit, 128-bit or 256-bit network authentication code that is the output of the function *f1*
MAC-S	a 64-bit, 128-bit or 256-bit resynchronization authentication code that is the output of the function *f1**
OP	Operator Variant Algorithm Configuration Field (used in MILENAGE)
OUT	a 1600-bit value that is taken as the output of the permutation Π when computing the functions *f1*, *f1**, *f2*, *f3*, *f4*, *f5* and *f5**
RAND	a 128-bit random challenge that is an input to the functions *f1*, *f1**, *f2*, *f3*, *f4*, *f5* and *f5**
RES	a 32-bit, 64-bit, 128-bit or 256-bit signed response that is the output of the function *f2*
SQN	a 48-bit sequence number that is an input to either of the functions *f1* and *f1**. (For *f1** this input is more precisely called SQN_{MS}.) See informative Annex C of [1] for methods of encoding sequence numbers
SQN_{MS}	(See SQN)
TOP	a 256-bit Operator Variant Algorithm Configuration Field that is a component of the functions *f1*, *f1**, *f2*, *f3*, *f4*, *f5* and *f5**
TOP_C	a 256-bit value derived from TOP and K and used within the computation of the functions

4 Preliminary information

4.1 Introduction

Within the security architecture of the 3GPP system there are seven security functions related to authentication and key agreement: f1, f1*, f2, f3, f4, f5 and f5*. The operation of these functions falls within the domain of one operator, and the functions are therefore to be specified by each operator rather than being fully standardized. The algorithms specified in the present document are examples that may be used by an operator who does not wish to design his own.

The algorithm specified is called Tuak (pronounced "too-ack").

It is not mandatory that the particular algorithms specified in the present document are used.

The inputs and outputs of all seven algorithms are defined in clause 4.4.

4.2 Notation

4.2.1 Radix

The prefix 0x is used to indicate hexadecimal numbers.

4.2.2 Bit-numbering for inputs and outputs

3GPP TS 33.102 [1] includes the following convention. (There is similar text in the specification of MILENAGE, as defined in 3GPP TS 35.206 [2]):

All data variables in the present document are presented with the most significant substring on the left hand side and the least significant substring on the right hand side. A substring may be a bit, byte or other arbitrary length bit string. Where a variable is broken down into a number of substrings, the left-most (most significant) substring is numbered 0, the next most significant is numbered 1, and so on through to the least significant.

So, for example, RAND[0] is the most-significant bit of RAND and RAND[127] is the least significant bit of RAND.

This convention applies to all inputs and outputs to Tuak, as listed in tables 1 to 9 below.

However, internally to the Tuak specification variables are simply treated as indexed bit strings, without a specific indication of bit, byte or word order.

4.2.3 Assignment operations

The assignment operator '='is used in many programming languages. Thus:

<*variable*> = <*expression*>

It means 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.

Also

<*variables*> = <*expressions*>

for lists of variables and expressions, then the left-most variable assumes the value the left-most expression had before the assignment took place, the next left-most variable assumes the value the next left-most expression had before the assignment took place, and so on.

For instance,

x[0]..*x*[2] = 3, 4, 5

means

(new value of *x*[0]) becomes 3,
(new value of *x*[1]) becomes 4,
(new value of *x*[2]) becomes 5.

Whereas:

x[0]..x[2] = y[2]..y[0]

means

(new value of *x*[0]) becomes (old value of *y*[2]),
(new value of *x*[1]) becomes (old value of *y*[1]),
(new value of *x*[2]) becomes (old value of *y*[0]).

4.2.4 Void

4.3 Void

5 Inputs and outputs

5.1 Tuak inputs and outputs

The inputs to Tuak are given in tables 1 and 2, the outputs in tables 3 to 9 below.

There are a few differences from the inputs and outputs to MILENAGE [2].

We allow tThe key K may be 128 bits or 256 bits. MAC-A and MAC-S may be 64, 128 or 256 bits. RES may be 32, 64, 128 or 256 bits. CK and IK may be 128 or 256 bits. Existing 3GPP specifications (see [1] and [12]) do not support all these possibilities, but they are included in Tuak for future flexibility in case future releases of these specifications may want to support them.

NOTE 1: The 3G security architecture specification [1] calls the output of the f1 function 'MAC' while the present document and [2] call it 'MAC-A'.

Any sizes for the parameters K, MAC-A, MAC-S, RES, CK and IK mentioned in the present document shall not be supported nor used in entities defined in 3GPP specifications until these specifications explicitly allow their use.

In any particular implementation, the parameters shall have a fixed length, chosen in advance. For example an operator may fix K at length 256 bits, RES at length 64 bits, CK and IK at length 128 bits. As the lengths do not vary with input, they are not specified as formal input parameters.

Table 1: Inputs to *f1* and *f1

Parameter	Size (bits)	Comment
K	128 or 256	Subscriber key K[0]…K[127] or K[0]…K[255]
RAND	128	Random challenge RAND[0]…RAND[127]
SQN	48	Sequence number SQN[0]…SQN[47] (for ***f1**** this input is more precisely called SQN_{MS})
AMF	16	Authentication management field AMF[0]…AMF[15]

Table 2: Inputs to f2, f3, f4, f5 and f5*

Parameter	Size (bits)	Comment
K	128 or 256	Subscriber key K[0]…K[127] or K[0]…K[255]
RAND	128	Random challenge RAND[0]…RAND[127]

Table 3: f1 output

Parameter	Size (bits)	Comment
MAC-A	64, 128 or 256	Network authentication code MAC-A[0]…MAC-A[63] or MAC-A[0]…MAC-A[127] or MAC-A[0]…MAC-A[255]

Table 4: *f1 output**

Parameter	Size (bits)	Comment
MAC-S	64, 128 or 256	Resynch authentication code MAC-S[0]…MAC-S[63] or MAC-S[0]…MAC-S[127] or MAC-S[0]…MAC-S[255]

Table 5: *f2* output

Parameter	Size (bits)	Comment
RES	32, 64, 128 or 256	Response RES[0]…RES[31] or RES[0]…RES[63] or RES[0]…RES[127] or RES[0]…RES[255]

Table 6: *f3* output

Parameter	Size (bits)	Comment
CK	128 or 256	Confidentiality key CK[0]…CK[127] or CK[0]…CK[255]

Table 7: *f4* output

Parameter	Size (bits)	Comment
IK	128 or 256	Integrity key IK[0]…IK[127] or IK[0]…IK[255]

Table 8: *f5* output

Parameter	Size (bits)	Comment
AK	48	Anonymity key AK[0]…AK[47]

Table 9: *f5 output**

Parameter	Size (bits)	Comment
AK	48	Resynch anonymity key AK[0]…AK[47]

NOTE 2: Both f5 and f5* outputs are called AK according to [1]. In practice only one of them at a time will be calculated in any given call to the authentication and key agreement algorithms.

5.2 Keccak and its inputs and outputs

This clause refers to the Keccak reference specification [3]. Use is made of the permutation Keccak-f[1600], which is abbreviated to Π, and defined formally in Annex C.

We use Strings **IN**[0] .. **IN**[1599] and **OUT**[0] .. **OUT**[1599] are used to represent the input and output of Π. As in [3], these are treated as simple bit strings. However, to support efficient implementations of Keccak (see [4]), inputs are mapped to **IN** and outputs are extracted from **OUT** in such a way that bits of input and output should not need to be reversed within bytes for such implementations.

The Keccak specification includes the concept of a security parameter which the designers call "capacity". Based on the designers' recommendations, a formal capacity of 512 bits is used: all input strings to the Keccak permutation shall be padded to 1088 bits, and then have 512 zero bits appended. The padding used to extend the input string to 1088 bits is the "1 0* 1" padding defined in [3], immediately preceded by "1 1 1 1" for consistency with Sakura coding and domain separation (see e.g. " SAKURA: a flexible coding for tree hashing" [5], start of section 6 for the Sakura coding, and

Table 4 for the domain separation coding). Note that our input strings before padding are always shorter than 832 bits, with the remaining bits of IN always the same in all modes, and output is only ever extracted from the first 832 bits of OUT – so in practice the effective capacity of the construction is at least 1600 - 832 = 768 bits.

5.3 Other inputs and substrings

MILENAGE uses a 128-bit value **OP**, and derives a 128-bit value $\mathbf{OP_C}$. **OP** is an Operator Variant Algorithm Configuration Field.

For Tuak a 256-bit Operator Variant Algorithm Configuration Field is specified, **TOP**; and a derived 256-bit value $\mathbf{TOP_C}$.

The following internal variables are defined in the algorithm definition:

- A 56-bit string **ALGONAME**[0] .. **ALGONAME**[55], with an arbitrary fixed value. This is specified as the ASCII representation of the string "TUAK1.0": to be explicit, **ALGONAME**[0] .. **ALGONAME**[55] = 0,1,0,1,0,1,0,0, 0,1,0,1,0,1,0,1, 0,1,0,0,0,0,0,1, 0,1,0,0,1,0,1,1, 0,0,1,1,0,0,0,1, 0,0,1,0,1,1,1,0, 0,0,1,1,0,0,0,0
- An 8-bit string **INSTANCE**[0] .. **INSTANCE**[7] which will be given different values for different algorithms within the set.

The internal variable **INSTANCE** is coded using the following schema (sections 6 gives the exact details) :

INSTANCE[0] .. **INSTANCE**[1] indicate which function is being implemented

INSTANCE[2]…**INSTANCE**[4] indicate the length of the MAC-A/MAC-S or RES output, or they are all set to zero when deriving $\mathbf{TOP_C}$

INSTANCE[5] .. **INSTANCE**[7] indicate whether the CK/IK/K lengths are 256 bit.

6 Definition of the example algorithms

6.1 Derivation of TOP_C

The **INSTANCE** variable is constructed as follows:

INSTANCE[0] .. **INSTANCE**[6] = 0,0,0,0,0,0,0

INSTANCE[7] = 0 if the length of K is 128 bits

= 1 if the length of K is 256 bits

The 1600-bit value **IN** is then constructed as follows:

IN[0] .. **IN**[255] = **TOP**[255] .. **TOP**[0]

IN[256] .. **IN**[263] = **INSTANCE**[7] .. **INSTANCE**[0]

IN[264] .. **IN**[319] = **ALGONAME**[55] .. **ALGONAME**[0]

IN[i] = 0 for $320 \leq i \leq 511$

IN[512] .. **IN**[767] = **K**[255] .. **K** [0] if the length of K is 256 bits

IN[512] .. **IN**[639] = **K**[127] .. **K** [0] if the length of K is 128 bits

IN[i] = 0 for $640 \leq i \leq 767$ if the length of K is 128 bits

IN[i] = 1 for $768 \leq i \leq 772$

IN[i] = 0 for $773 \leq i \leq 1086$

IN[1087] = 1

IN[i] = 0 for 1088 ≤ i ≤ 1599

Then apply the permutation:

OUT = Π(**IN**)

And extract $\mathbf{TOP_C}$ as follows:

$\mathbf{TOP_C}$[0] .. $\mathbf{TOP_C}$[255] = **OUT**[255] .. **OUT**[0]

6.2 Specification of the function f1

The internal **INSTANCE** variable is constructed as follows:

INSTANCE[0] .. **INSTANCE**[1] = 0,0

INSTANCE[2] .. **INSTANCE**[4] = 0,0,1 if the MAC-A length is 64 bits

= 0,1,0 if the MAC-A length is 128 bits

= 1,0,0 if the MAC-A length is 256 bits

INSTANCE[5] .. **INSTANCE**[6] = 0,0

INSTANCE[7] = 0 if the length of K is 128 bits

= 1 if the length of K is 256 bits

The 1600-bit value **IN** is then constructed as follows:

IN[0] .. **IN**[255] = $\mathbf{TOP_C}$[255] .. $\mathbf{TOP_C}$[0]

IN[256] .. **IN**[263] = **INSTANCE**[7] .. **INSTANCE**[0]

IN[264] .. **IN**[319] = **ALGONAME**[55] .. **ALGONAME**[0]

IN[320] .. **IN**[447] = **RAND**[127] .. **RAND**[0]

IN[448] .. **IN**[463] = **AMF**[15] .. **AMF**[0]

IN[464] .. **IN**[511] = **SQN**[47] .. **SQN**[0]

IN[512] .. **IN**[767] = **K**[255] .. **K** [0] if the length of K is 256 bits

IN[512] .. **IN**[639] = **K**[127] .. **K** [0] if the length of K is 128 bits

IN[i] = 0 for 640 ≤ i ≤ 767 if the length of K is 128 bits

IN[i] = 1 for 768 ≤ i ≤ 772

IN[i] = 0 for 773 ≤ i ≤ 1086

IN[1087] = 1

IN[i] = 0 for 1088 ≤ i ≤ 1599

Then apply the permutation:

OUT = Π(**IN**)

And extract function output as follows.

Output of ***f1*** = **MAC-A**, where

MAC-A[0] .. **MAC-A**[63] = **OUT**[63] .. **OUT**[0] if the MAC-A length is 64 bits

MAC-A[0] .. **MAC-A**[127] = **OUT**[127] .. **OUT**[0] if the MAC-A length is 128 bits

[0] .. **MAC-A**[255] = **OUT**[255] .. **OUT**[0] if the MAC-A length is 256 bits

6.3. Specification of the function f1*

The internal **INSTANCE** variable is constructed as follows. The construction is very similar to ***f1***, except that **INSTANCE**[0] is 1 rather than 0:

INSTANCE[0] .. **INSTANCE**[1] = 1,0

INSTANCE[2] .. **INSTANCE**[4] = 0,0,1 if the MAC-S length is 64 bits

= 0,1,0 if the MAC-S length is 128 bits

= 1,0,0 if the MAC-S length is 256 bits

INSTANCE[5] .. **INSTANCE**[6] = 0,0

INSTANCE[7] = 0 if the length of K is 128 bits

= 1 if the length of K is 256 bits

The 1600-bit value **IN** is then constructed in exactly the same way as for ***f1*** (but note **IN**[263] will be 1 rather than 0).

Then the permutation:

OUT = Π(**IN**)

is applied and the function output is extracted as follows.

Output of ***f1**** = **MAC-S**, where

MAC-S[0] .. **MAC-S**[63] = **OUT**[63] .. **OUT**[0] if the MAC-S length is 64 bits

MAC-S[0] .. **MAC-S**[127] = **OUT**[127] .. **OUT**[0] if the MAC-S length is 128 bits

MAC-S[0] .. **MAC-S**[255] = **OUT**[255] .. **OUT**[0] if the MAC-S length is 256 bits

6.4 Specification of the functions f2, f3, f4 and f5

The internal **INSTANCE** variable is constructed as follows:

INSTANCE[0] .. **INSTANCE**[1] = 0,1

INSTANCE[2] .. **INSTANCE**[4] = 0,0,0 if the length of RES is 32 bits

= 0,0,1 if the length of RES is 64 bits

= 0,1,0 if the length of RES is 128 bits

= 1,0,0 if the length of RES is 256 bits

INSTANCE[5] = 0 if the length of CK is 128 bits

= 1 if the length of CK is 256 bits

INSTANCE[6] = 0 if the length of IK is 128 bits

= 1 if the length of IK is 256 bits

INSTANCE[7] = 0 if the length of K is 128 bits

= 1 if the length of K is 256 bits

The 1600-bit value **IN** is then constructed as follows:

IN[0] .. **IN**[255] = $\mathbf{TOP}_C$[255] .. $\mathbf{TOP}_C$[0]

IN[256] .. **IN**[263] = **INSTANCE**[7] .. **INSTANCE**[0]

IN[264] .. **IN**[319] = **ALGONAME**[55] .. **ALGONAME**[0]

IN[320] .. **IN**[447] = **RAND**[127] .. **RAND**[0]

IN[i] = 0 for $448 \leq i \leq 511$

IN[512] .. **IN**[767] = **K**[255] .. **K** [0] if the length of K is 256 bits

IN[512] .. **IN**[639] = **K**[127] .. **K** [0] if the length of K is 128 bits

IN[i] = 0 for $640 \leq i \leq 767$ if the length of K is 128 bits

IN[i] = 1 for $768 \leq i \leq 772$

IN[i] = 0 for $773 \leq i \leq 1086$

IN[1087] = 1

IN[i] = 0 for $1088 \leq i \leq 1599$

Then the permutation:

OUT = Π(**IN**)

is applied and the function outputs are extracted as follows:

Output of ***f2*** = **RES**, where:

RES[0] .. **RES**[31] = **OUT**[31] .. **OUT**[0] if the RES length is 32 bits

RES[0] .. **RES**[63] = **OUT**[63] .. **OUT**[0] if the RES length is 64 bits

RES[0] .. **RES**[127] = **OUT**[127] .. **OUT**[0] if the RES length is 128 bits

RES[0] .. **RES**[255] = **OUT**[255] .. **OUT**[0] if the RES length is 256 bits

Output of ***f3*** = **CK**, where:

CK[0] .. **CK**[127] = **OUT**[383] .. **OUT**[256] if the CK length is 128 bits

CK[0] .. **CK**[255] = **OUT**[511] .. **OUT**[256] if the CK length is 256 bits

Output of ***f4*** = **IK**, where:

IK[0] .. **IK**[127] = **OUT**[639] .. **OUT**[512] if the IK length is 128 bits

IK[0] .. **IK**[255] = **OUT**[767] .. **OUT**[512] if the IK length is 256 bits

Output of ***f5*** = **AK**, where:

AK[0] .. **AK**[47] = **OUT**[815] .. **OUT**[768]

6.5 Specification of the function f5*

The internal **INSTANCE** variable is constructed as follows:

INSTANCE[0] .. **INSTANCE**[1] = 1,1

INSTANCE[2] .. **INSTANCE**[4] = 0,0,0

INSTANCE[5] .. **INSTANCE**[6] = 0,0

INSTANCE[7] = 0 if the length of K is 128 bits

= 1 if the length of K is 256 bits

The 1600-bit value **IN** is then constructed in exactly the same way as for ***f2, f3, f4,*** and ***f5*** (but note that **IN**[263] will be 1 rather than 0, and that **IN**[257], **IN**[258], **IN**[259], **IN**[260] , **IN**[261] will all be 0).

Then the permutation:

OUT = Π(**IN**)

is applied and the function output is extracted as follows:

Output of ***f5**** = **AK**, where:

AK[0] .. **AK**[47] = **OUT**[815] .. **OUT**[768]

7 Implementation considerations

7.1 TOP_C computed on or off the UICC?

It will be seen in clause 6.1 that $\mathbf{TOP}_C$ is computed from **OP** and **K**, and that it is only $\mathbf{TOP}_C$, not **TOP**, that is ever used in subsequent computations.

As for $\mathbf{OP}_C$ in MILENAGE, it is recommended that $\mathbf{TOP}_C$ be computed off the UICC where possible, and that $\mathbf{TOP}_C$ rather than **TOP** be loaded to the UICC for use in subsequent computations. This should also apply when updating an embedded UICC (eUICC) as defined in [13]: the value of $\mathbf{TOP}_C$ (and not **TOP**) should be loaded to the eUICC in conjunction with the new K and other operator customization parameters.

This gives the following benefits:

- The complexity of the algorithms run on the UICC is reduced.
- It is more likely that **TOP** can be kept secret. (If **TOP** is stored on the UICC, it only takes one UICC to be reverse engineered for **TOP** to be discovered and published. But it should be difficult for someone who has discovered even a large number of ($\mathbf{TOP}_C$, **K**) pairs to deduce **TOP**. That means that the $\mathbf{TOP}_C$ associated with any other value of **K** will be unknown, which may make it harder to mount some kinds of cryptanalytic and forgery attacks. The algorithms are designed to be secure whether or not **TOP** is known to the attacker, but a secret **TOP** is one more hurdle in the attacker"s path.)

7.2 Further customization

TOP obviously allows for some degree of operator customization.

Further, as described in clause 5.1, the lengths of **K**, and of **MAC-A/MAC-S**, **RES**, **CK** and **IK** can be chosen by the operator, although they have to be fixed in each particular implementation of Tuak. In a flexible implementation (e.g. UICC), these operator-chosen parameter lengths could be loaded to the UICC in conjunction with the associated **K** and $\mathbf{TOP}_C$.

Where compatibility is required with existing 3GPP specifications, the operator shall set the length of the K to 128 bits, the length of the RES to between 32 and 128 bits, the length of the **MAC-A/MAC-S** to 64 bits, the length of **CK** to 128 bits, and the length of **IK** to 128 bits.

If an even more secure version of this algorithm is required, this could be done by adding extra applications of the Keccak permutation before extracting output. These would be used in the derivation of $\mathbf{TOP}_C$ (clause 6.1), and each of the algorithms ***f1, f1*, f2-f5, f5**** (clauses 6.2 to 6.5). In each case, instead of:

Construct **IN**

OUT = Π(**IN**)

Extract outputs

the approach could be

Construct **IN**

OUT = Π(Π(**IN**))

Extract outputs

or

Construct **IN**

OUT = Π(Π(Π(**IN**)))

Extract outputs

or however many extra applications of the permutation are required. Again, in a flexible implementation (e.g. UICC), the number of iterations of Π may be loaded to the UICC as an operator-chosen parameter.

7.3 Resistance to side channel attacks

When these algorithms are implemented on a UICC, consideration should be given to protecting them against side channel attacks such as differential power analysis (DPA). [4, 6, 7, 8, 9, 10, 11] may be useful references.

Annex A (normative): Tuak diagrams

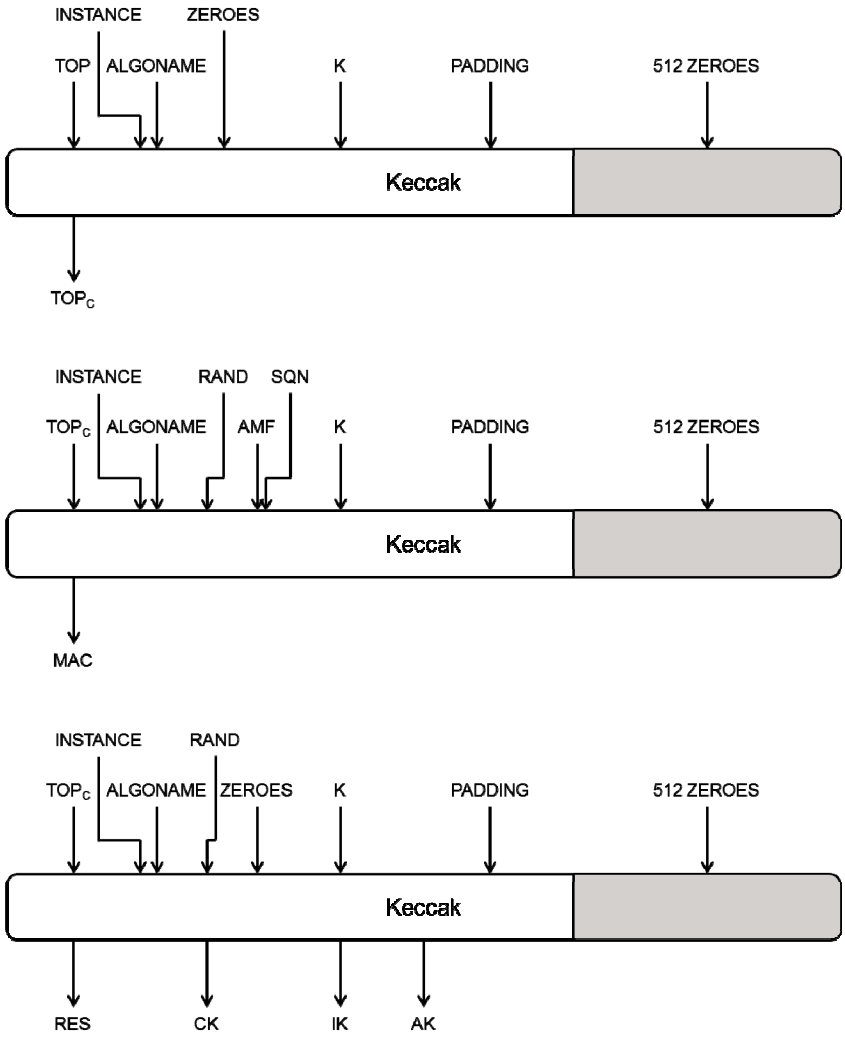

Figure A.1 Tuak operation

The first diagram illustrates the derivation of $\mathbf{TOP_C}$.

The second diagram illustrates the derivation of either **MAC-A** (using the ***f1*** function) or **MAC-S** (using the ***f1**** function), with different values of the **INSTANCE** byte in each case.

The third diagram illustrates the derivation of **RES** (using the ***f2*** function), **CK** (using ***f3***), **IK** (using ***f4***) and **AK** (using ***f5***) or alternatively the derivation of **AK** using ***f5**** (in which case the other three outputs should be ignored).

In all cases it is assumed that just one iteration of the Keccak permutation is used (see clause 7.2).

Note the 512-bit "capacity" of Keccak: only zeroes are input to the rightmost 512 bits, and no outputs are extracted from the rightmost 512 bits.

Annex B (informative): TuakApplication Programme Interface (AP) in ANSI CI

```
/*  ------------------------------------------------------------
    Constants and Typedefs
    ------------------------------------------------------------
*/
typedef unsigned char    uint8;
static const uint8  ALGONAME[] = "TUAK1.0";

uint8   TOP[32];               /* Operator's Configuration     */
uint8   KEY_sz;                /* = 16/32 bytes                    */
uint8   RES_sz;                /* = 4/8/16/32 bytes                */
uint8   CK_sz;                 /* = 16/32 bytes                    */
uint8   IK_sz;                 /* = 16/32 bytes                    */
uint8   MAC_sz;                /* = 8/16/32 bytes                  */
uint8   KeccakIterations;      /* >=1, number of iterations        */

/* ------------------------------------------------------------
   TUAK API Declaration
   ------------------------------------------------------------
*/
void TUAK_ComputeTOPC(  uint8 *key,         /* in,  uint8[KEY_sz]   */
                        uint8 *TOPC     /* out, uint8[32]       */
                      );

void TUAK_f1                (   uint8 *key,     /* in,  uint8[KEY_sz]   */
                                uint8 *rand,        /* in,  uint8[16]       */
                                uint8 *sqn,     /* in,  uint8[6]            */
                                uint8 *amf,     /* in,  uint8[2]            */
                                uint8 *mac          /* out, uint8[MAC_sz]   */
                            );

void TUAK_f2345         (   uint8 *key,     /* in,  uint8[KEY_sz]   */
                                uint8 *rand,        /* in,  uint8[16]       */
                                uint8 *res,     /* out, uint8[RES_sz]   */
                                uint8 *ck,          /* out, uint8[CK_sz]    */
                                uint8 *ik,          /* out, uint8[IK_sz]    */
                                uint8 *ak           /* out, uint8[6]            */
                            );

void TUAK_f1s               (   uint8 *key,     /* in,  uint8[KEY_sz]   */
                                uint8 *rand,        /* in,  uint8[16]       */
                                uint8 *sqn,     /* in,  uint8[6]            */
                                uint8 *amf,     /* in,  uint8[2]            */
                                uint8 *mac          /* out, uint8[MAC_sz]   */
                            );

void TUAK_f5s               (   uint8 *key,     /* in,  uint8[KEY_sz]   */
                                uint8 *rand,        /* in,  uint8[16]       */
                                uint8 *ak           /* out, uint8[6]            */
                            );
```

Annex C (normative): Specification of the Keccak permutation used within Tuak

The following specification of the permutation Π is extracted from the Keccak reference specification [3], section 1.2.

The permutation Π is described as a sequence of operations on a state a that is a three-dimensional 5 x 5 x 64 array of elements of GF(2). The expression $a[x][y][z]$ with $x, y \in \mathbf{Z}_5$ and $z \in \mathbf{Z}_{64}$, denotes the bit in position (x, y, z), with the indices starting from zero. The array a is mapped to the bits of an input and output string s by $s[64.(5y + x) + z] = a[x][y][z]$, so that s is a 1600-bit string, indexed from bit 0 up to bit 1599. Note that expressions in the x and y coordinates should be taken modulo 5 and expressions in the z coordinate modulo 64. It is possible to omit the $[z]$ index, both the $[y][z]$ indices, or all three indices, implying that the statement is valid for all values of the omitted indices.

Π is an iterated permutation, consisting of a sequence of 24 rounds R, indexed by i_r from 0 to 23. Each round consists of five steps, $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$, performed in the following order:

$\theta : a[x][y][z] \leftarrow a[x][y][z] + \Sigma^4_{y'=0} a[x - 1][y'][z] + \Sigma^4_{y'=0} a[x + 1][y'][z - 1]$,

$\rho : a[x][y][z] \leftarrow a[x][y][z - (t + 1)(t + 2)/2]$,

with t satisfying $0 \le t < 24$ and

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \text{ in } GF(5)_{2x2}$$

or $t = -1$ if $x = y = 0$,

so that t is given by the following table:

x=	0	1	2	3	4
y=0	-1	0	18	6	12
y=1	7	23	2	9	22
y=2	1	3	17	16	20
y=3	13	8	4	5	15
y=4	19	10	21	14	11

and $(t + 1)(t + 2)/2 \in \mathbf{Z}_{64}$ is given by:

x=	0	1	2	3	4
y=0	0	1	62	28	27
y=1	36	44	6	55	20
y=2	3	10	43	25	39
y=3	41	45	15	21	8
y=4	18	2	61	56	14

$\pi : a[x][y] \leftarrow a[x'][y']$, with

$$\begin{matrix} x & 0\ 1 & x' \\ y & 2\ 3 & y' \end{matrix},$$

$\chi : a[x] \leftarrow a[x] + (a[x + 1] + 1)a[x + 2]$,

$\iota : a \leftarrow a + \mathrm{RC}[i_r]$.

GF(2). With the exception of
the value of the round constants $\mathrm{RC}[i_r]$, these rounds are identical.

The round constants are given by (with the first index denoting the round number):

$\mathrm{RC}[i_r][0][0][2^j - 1] = \mathrm{rc}[j + 7i_r]$ for all $0 \le j \le 6$,
$\mathrm{RC}[i_r][x][y][z] = 0$ for all other values of x, y, z

The values $\mathrm{rc}[t] \in GF(2)$ are defined as the output of a binary linear feedback shift register (LFSR):

$\mathrm{rc}[t] = (x^t \bmod x^8 + x^6 + x^5 + x^4 + 1) \bmod x$ in $GF(2)[x]$,

so that $\mathrm{rc}[j + 7i_r]$ is given by the following table:

j=	0	1	2	3	4	5	6

j=	0	1	2	3	4	5	6

i_r=0	1	0	0	0	0	0	0
i_r=1	0	1	0	1	1	0	0
i_r=2	0	1	1	1	1	0	1
i_r=3	0	0	0	0	1	1	1
i_r=4	1	1	1	1	1	0	0
i_r=5	1	0	0	0	0	1	0
i_r=6	1	0	0	1	1	1	1
i_r=7	1	0	1	0	1	0	1
i_r=8	0	1	1	1	0	0	0
i_r=9	0	0	1	1	0	0	0
i_r=10	1	0	1	0	1	1	0
i_r=11	0	1	1	0	0	1	0

i_r=12	1	1	1	1	1	1	0
i_r=13	1	1	1	1	0	0	1
i_r=14	1	0	1	1	1	0	1
i_r=15	1	1	0	0	1	0	1
i_r=16	0	1	0	0	1	0	1
i_r=17	0	0	0	1	0	0	1
i_r=18	0	1	1	0	1	0	0
i_r=19	0	1	1	0	0	1	1
i_r=20	1	0	0	1	1	1	1
i_r=21	0	0	0	1	1	0	1
i_r=22	1	0	0	0	0	1	0
i_r=23	0	0	1	0	1	1	1

Annex D (informative): Example source code for Tuak (ANSI C)

Tuak is built on top of the Keccak function, as described in annex C. In the following example code, a Tuak configuration is defined as a set of global variables, so that these can be initialized or modified at run-time, and there is no need to fix them in the code as static constants. The read/write data to the Keccak state, including the padding, will depend on the Keccak state representation. Included in this example are 8-bit, 32-bit and 64-bit implementations of the Keccak function, each having a different representation of the state.

The Tuak f-functions do similar calculations in their core, therefore, it was possible to combine most of the computations in one function called TUAK_Main(). Lengths are given in bytes, rather than bits, in order to support 8-bit environments (e.g., length=256 does not fit in type `uint8`). All the example codes are endianness-free, i.e. can be used on both big and little endian machines.

```
/* This code may be freely used or adapted.

   This implementation of TUAK is endianness-free.
   It supports 64-bit, 32-bit and 8-bit environments.
*/

/* ---------------------------------------------------------------------
   Constants, typedefs, macros, compilation settings
   ---------------------------------------------------------------------
*/
/* This macro selects Keccak_f implementation instance - 8/32/64-bit version */
#define KECCAK_VERSION_BITS     32

/* Depending on the version of Keccak we do:
   - map KECCAK_F macro to relevant Keccak_f (8/32/64-bit) instance
   - declare Keccak"s state INOUT[] as global, for simplicity
   - define method for TUAK padding TUAK_ADD_PADDING()
*/
#if KECCAK_VERSION_BITS==64
    static uint64   INOUT[25];  /* state to Keccak_f for 64-bit version */
    extern void     Keccak_f_64(uint64 *s);
#   define KECCAK_F     Keccak_f_64
#   define TUAK_ADD_PADDING()       INOUT[12] = 0x1FULL, INOUT[16] = (0x01ULL<<63)
#elif KECCAK_VERSION_BITS==32
    static uint32   INOUT[50];  /* state to Keccak_f for 32-bit version  */
    extern void     Keccak_f_32(uint32 *s);
#   define KECCAK_F     Keccak_f_32
#   define TUAK_ADD_PADDING()       INOUT[24] = 0x1FUL, INOUT[33] = 0x80000000
#elif KECCAK_VERSION_BITS==8
    static uint8        INOUT[200]; /* state to Keccak_f for 8-bit version  */
    extern void     Keccak_f_8(uint8 s[200]);
#   define KECCAK_F     Keccak_f_8
#   define TUAK_ADD_PADDING()       INOUT[96] = 0x1F, INOUT[135] = 0x80
#else
#   error The requested version of Keccak_f is not implemented!
#endif

static const uint8  ALGONAME[] = "TUAK1.0";

void TUAK_ComputeTOPC(uint8*, uint8*);

/* ---------------------------------------------------------------------
   TUAK Instance Configuration
   if dynamic   => can be set/modified on run-time
   if constants => fixed instance of the algorithm
   ---------------------------------------------------------------------
*/
uint8   TOP[32];                        /* Operator's Configuration     */
uint8   KEY_sz              = 16;   /* = 16/32 bytes                        */
uint8   RES_sz              = 8;    /* = 4/8/16/32 bytes                */
uint8   CK_sz                   = 32;   /* = 16/32 bytes                        */
uint8   IK_sz                   = 32;   /* = 16/32 bytes                        */
uint8   MAC_sz              = 16;   /* = 8/16/32 bytes                  */
uint8   KeccakIterations    = 1;    /* >=1, number of Keccak_f iterations   */

/* ---------------------------------------------------------------------
   PUSH_DATA / PULL_DATA, TUAK_Main()
```

```
   ------------------------------------------------------------------
*/
void PUSH_DATA(const uint8 * data, uint8 n, uint8 location)
{   while(n--)
#if KECCAK_VERSION_BITS==64
        INOUT[location>>3] |= ((uint64)data[n]) << ((location++ & 7)<<3);
#elif KECCAK_VERSION_BITS==32
        INOUT[location>>2] |= ((uint32)data[n]) << ((location++ & 3)<<3);
#elif KECCAK_VERSION_BITS==8
        INOUT[location++] = data[n];   /* Note: reversed order of bytes */
#endif
}

void PULL_DATA(uint8 * data, uint8 n, uint8 location)
{   while(n--)
#if KECCAK_VERSION_BITS==64
        data[n] = (uint8)(INOUT[location>>3] >> ((location++ & 7)<<3));
#elif KECCAK_VERSION_BITS==32
        data[n] = (uint8)(INOUT[location>>2] >> ((location++ & 3)<<3));
#elif KECCAK_VERSION_BITS==8
        data[n] = INOUT[location++];   /* Note: reversed order of bytes */
#endif
}

/* Universal function used by TUAK API   functions */
void TUAK_Main      (   uint8 instance,     /* in, uint8                */
                        uint8 *rand,            /* in, uint8[16]            */
                        uint8 *amf,         /* in, uint8[2]         */
                        uint8 *sqn,         /* in, uint8[6]         */
                        uint8 *key              /* in, uint8[16/32]     */
                    )
{   uint8 i, TOPC[32];
    TUAK_ComputeTOPC(key, TOPC);               /* compute TOPC             */
    memset((uint8*)INOUT    , 0, 200);         /* clean INOUT          */

    PUSH_DATA(TOPC      , 32,  0);          /* TOPC                 */
    PUSH_DATA(&instance  , 1 , 32);         /* INSTANCE             */
    PUSH_DATA(ALGONAME   , 7 , 33);         /* ALGONAME             */
    PUSH_DATA(rand       , 16, 40);         /* RAND                 */
    if(amf) PUSH_DATA(amf, 2 , 56);         /* AMF , if !=NULL      */
    if(sqn) PUSH_DATA(sqn, 6 , 58);         /* SQN , if !=NULL      */
    PUSH_DATA(key, (instance & 1)?32:16, 64);   /* KEY-128/256 bits     */

    TUAK_ADD_PADDING();                     /* Padding bits 768-1087    */

    for(i=0; i<KeccakIterations; ++i)
        KECCAK_F(INOUT);
}

/* ------------------------------------------------------------------
   TUAK API Definition
   ------------------------------------------------------------------
*/
void TUAK_ComputeTOPC(  uint8 *key,         /* in,  uint8[16/32]        */
                        uint8 *TOPC         /* out, uint8[32]           */
                     )
{   uint8 i, inst = KEY_sz>>5;
    memset(INOUT, 0, 200);
    PUSH_DATA(TOP       , 32, 0 );                  /* TOP                          */
    PUSH_DATA(&inst     , 1 , 32);                  /* INSTANCE for TOPC        */
    PUSH_DATA(ALGONAME , 7 , 33);                   /* ALGONAME                 */
    PUSH_DATA(key   , KEY_sz, 64);                  /* KEY-128/256              */
    TUAK_ADD_PADDING();                             /* Padding bits 768-1087    */

    for(i=0; i<KeccakIterations; ++i)
        KECCAK_F(INOUT);

    PULL_DATA(TOPC, 32, 0);                         /* get the result           */
}

void TUAK_f1            (   uint8 *key,             /* in,  uint8[KEY_sz]       */
                            uint8 *rand,                /* in,  uint8[16]           */
                            uint8 *sqn,             /* in,  uint8[6]            */
                            uint8 *amf,             /* in,  uint8[2]            */
                            uint8 *mac                  /* out, uint8[MAC_sz]       */
                        )
{   TUAK_Main( (KEY_sz>>5) | MAC_sz, rand, amf, sqn, key);
    PULL_DATA(mac, MAC_sz, 0);
```

```
}

void TUAK_f2345     (   uint8 *key,             /* in,  uint8[KEY_sz]       */
                        uint8 *rand,                /* in,  uint8[16]           */
                        uint8 *res,             /* out, uint8[RES_sz]       */
                        uint8 *ck,                  /* out, uint8[CK_sz]        */
                        uint8 *ik,                  /* out, uint8[IK_sz]        */
                        uint8 *ak                   /* out, uint8[6]                */
                    )
{   TUAK_Main( (KEY_sz>>5) | ((IK_sz>>4)&0x02) | ((CK_sz>>3)&0x04)
  | (RES_sz&0x38) | 0x40, rand, 0, 0, key);
    PULL_DATA(res, RES_sz, 0 );
    PULL_DATA(ck , CK_sz , 32);
    PULL_DATA(ik , IK_sz , 64);
    PULL_DATA(ak , 6     , 96);
}

void TUAK_f1s       (   uint8 *key,                 /* in,  uint8[KEY_sz]       */
                        uint8 *rand,                    /* in,  uint8[16]           */
                        uint8 *sqn,                 /* in,  uint8[6]                */
                        uint8 *amf,                 /* in,  uint8[2]                */
                        uint8 *mac                      /* out, uint8[MAC_sz]       */
                    )
{   TUAK_Main( (KEY_sz>>5) | MAC_sz | 0x80, rand, amf, sqn, key);
    PULL_DATA(mac, MAC_sz, 0);
}

void TUAK_f5s       (   uint8 *key,                 /* in,  uint8[KEY_sz]       */
                        uint8 *rand,                    /* in,  uint8[16]           */
                        uint8 *ak                       /* out, uint8[6]                */
                    )
{   TUAK_Main( (KEY_sz>>5) | 0xc0, rand, 0, 0, key);
    PULL_DATA(ak, 6, 96);
}
```

Annex E (informative): Example source code for Keccak (ANSI C)

Comments to the example code for a 64-bit implementation of Keccak

This 64-bit implementation of the Keccak permutation follows the specification in annex C. It has 24 rounds of 5 basic steps in the order Theta, Rho, Pi, Chi, Iota. The state of Keccak is 1600 bits, represented as 25 blocks of 64 bits each (`uint64 s[25]`). Such a block is referred to as a "lane" in the Keccak reference specification [3].

In the example code, bit n=0..1599 of Keccak state is mapped to the state representation array as:

bit(n) = (s[n/64]>>(n%64))&1. The mapping between s[w] and a[x][y][z] is: a[x][y][z] = (s[5y+x]>>z)&1, for any triple x, y=0..4, z=0..63 (and w=5y+x).

An efficient way to implement Keccak is to code it as an update function. For the purpose of efficiency, there are three pre-computed constant tables Rho[25], Pi[25], Iota[24] (74 bytes in total). Run-time requires 43+sizeof(uint64*) bytes of stack, excluding possible alignment of input arguments.

In the Theta function, it is necessary to perform the following computation:

$\theta : a[x][y][z] \leftarrow a[x][y][z] + \Sigma^{4}_{y'=0} a[x-1][y'][z] + \Sigma^{4}_{y'=0} a[x+1][y'][z-1]$.

To do this, first compute 5 sums $\Sigma^{4}_{y'=0} a[x][y'][z]$ for each x=0..4 (and all z=0..63) and store them in five temporary 64-bit variables (`uint64 t[5]`). Such an operation as a[x][y][z] ←a[x][y][z-1] is just a rotation of the corresponding 64-bit value of s by 1 to the left, and a[x][y][z] ←a[x′][y′][z] is a corresponding assignment s[w]←s[w']. This way, in the second loop of Theta, run over y=0..4 and update 25 words of s based on pre-computed temporary sums.

The Rho function is just a circular rotation of bits for each of 64-bit words individually of the state s[25]. The number of bits a word s[n] is to be rotated by is Rho[n].

The Pi function is a full-cycle permutation of 24 words of s, excluding s[0]. To avoid duplicating the state for this operation, remember the first word in the permutation chain s[1] somewhere in a temporary variable T, then perform s[1] ← s[Pi[1]], where the value Pi[1] is the index of the word that has to go to s[1]. In the next step, assign s[Pi[1]]←s[Pi[Pi[1]]], and so on. After 23 such steps the permutation is almost complete, except that the saved value should be located to the last referenced place s[Pi^23[1]]←T.

In the Chi function, reuse 5 temporary 64-bit words to compute somewhat mixed Boolean expressions; the function is straightforward to implement.

The Iota function updates 7 bits indexed by 0, 1, 3, 7, 15, 31, 63, of only one word s[0], by XORing them with some constant values that depend on the current round index (r=0..23). Each cell of the pre-computed constant table Iota[24] encodes those 7 bits in one byte (`uint8`) in the following way.

Bits related to indices 0, 1, 3, 7 stay in their correct place of the byte Iota[r]. Bits related to indices 15, 31, 63 are mapped to bits 4, 5, 6 of the byte Iota[r], correspondingly. When XORing, the lower bits 0-7 appear well mapped (`Iota[round]&0x8B`), and the upper bits 31-63 are received by the relevant three shifts of that byte to the left to certain positions (note, the byte Iota[r] is first converted to uint64). It is also possible to OR all the shifts first, and then to mask those 7 bits before XORing the result to s[0]. This trick is possible since ORing does not overlap those 7 bits, but the result of ORing will, however, interfere some of the other 57 bits, and by the final masking with the constant `0x800000008000808BULL`, remove that unwanted influence.

Comments to the example code for an 8-bit implementation of Keccak

The 8-bit implementation of Keccak is basically a step-by-step refactored version of the 64-bit version. The state of Keccak is now represented as 200 bytes of 8 bits each (`uint8 s[200]`). Each block of 8 bytes is a mapping of a relevant 64-bit word in the 64-bit version. The Keccak state is mapped to the state representation array as: bit(n) = (s[n/8]>>(n%8))&1, for n=0..1599.

A 64-bit word (that is now represented as 8 bytes) can be rotated by 1-7 bits easily –just save the first byte of the 8-byte block, then shift and propagate 1-7 bits from one byte to the next, and, finally, complete the operation at the end by using the saved byte.

However, in the Rho function, the rotation parameter - *r* bits - can be any value in the range 1..63. Then perform the following technique. Split *r* as *r*=8*A+B, where A is the number of full bytes of a 64-bit word to be rotated, and B is the remaining number of bits of the rotation value *r*. In the first loop, rotate an 8-byte block of s by A bytes, and store the result in some other temporary 8-byte block (in `t[0..7]`). In the second step. copy that temporary block `t[0..7]` back to the relevant location of the state s, and, in parallel, perform the rotation of the block by B bits in the way mentioned earlier.

Comments to the example code for a 32-bit implementation of Keccak

The 32-bit implementation of Keccak is yet another step-by-step refactoring of the 64-bit version. The state of Keccak is represented as `uint32[50]`, where each two 32-bit consecutive words are mapped to one 64-bit word. i.e., state64[k] = (state32[2*k+1]<<32) | state32[2*k], for k=0..24.

In the Rho function, every such pair of 32-bit words (A; B), where A=state32[2*k] and B=state32[2*k+1], for some k=0..24, needs to be rotated by n=0..63 bits. The resulting pair will be (A'; B') = (A<<n | B>>(32-n); B<<n | A>>(32-n)), if n<32. In case n>=32 then take n%32 as the shifting parameter for each of the 32-bit words, and the resulting A' and B' are swapped. The refactoring of other functions is straightforward.

```
/* This code may be freely used or adapted.
*/
typedef unsigned char           uint8;
typedef unsigned long           uint32;
typedef unsigned long long  uint64;

const uint8 Rho[25]     = {0,1,62,28,27,36,44,6,55,20,3,10,43,25,39,41,45,
   15,21,8,18,2,61,56,14};

const uint8 Pi[25]      = {0,6,12,18,24,3,9,10,16,22,1,7,13,19,20,4,5,11,17,
   23,2,8,14,15,21};

const uint8 Iota[24]    = {1,146,218,112,155,33,241,89,138,136,57,42,187,203,
   217,83,82,192,26,106,241,208,33,120};

#define ROTATE64(value, n)  \
((((uint64)(value))<<(n)) | (((uint64)(value))>>(64-(n))))

/* ---------------------------------------------------------------------
   64-bit version of Keccak_f(1600)
   ---------------------------------------------------------------------
*/
void Keccak_f_64(uint64 *s)
{   uint64 t[5];
    uint8 i, j, round;

    for(round=0; round<24; ++round)
    {   /* Theta function */
        for(i=0; i<5; ++i)
            t[i] = s[i] ^ s[5+i] ^ s[10+i] ^ s[15+i] ^ s[20+i];
        for(i=0; i<5; ++i, s+=5)
        {   s[0] ^= t[4] ^ ROTATE64(t[1], 1);
            s[1] ^= t[0] ^ ROTATE64(t[2], 1);
            s[2] ^= t[1] ^ ROTATE64(t[3], 1);
            s[3] ^= t[2] ^ ROTATE64(t[4], 1);
            s[4] ^= t[3] ^ ROTATE64(t[0], 1);
        }
        s -= 25;
```

```
        /* Rho function */
        for(i=1; i<25; ++i)
            s[i] = ROTATE64(s[i], Rho[i]);

        /* Pi function */
        for(t[1] = s[i=1]; (j=Pi[i]) > 1; s[i]=s[j], i=j);
        s[i] = t[1];

        /* Chi function */
        for(i=0; i<5; ++i, s += 5)
        {   t[0] = (~s[1]) & s[2];
            t[1] = (~s[2]) & s[3];
            t[2] = (~s[3]) & s[4];
            t[3] = (~s[4]) & s[0];
            t[4] = (~s[0]) & s[1];
            for(j=0; j<5; ++j) s[j] ^= t[j];
        }
        s -= 25;

        /* Iota function */
        t[0] = Iota[round];
        *s ^= (t[0] | (t[0]<<11) | (t[0]<<26) | (t[0]<<57))
& 0x8000000080008008ULL; /* set & mask bits 0,1,3,7,15,31,63 */
    }
}


/* ----------------------------------------------------------------------
   8-bit version of Keccak_f(1600)
   ----------------------------------------------------------------------
*/
void Keccak_f_8(uint8 s[200])
{   uint8 t[40], i, j, k, round;

    for(round=0; round<24; ++round)
    {   /* Theta function */
        for(i=0; i<40; ++i)
            t[i]=s[i]^s[40+i]^s[80+i]^s[120+i]^s[160+i];
        for(i=0; i<200; i+=8)
            for(j = (i+32)%40, k=0; k<8; ++k)
                s[i+k] ^= t[j+k];
        for(i=0; i<40; t[i] = (t[i]<<1)|j, i+=8)
            for(j = t[i+7]>>7, k=7; k; --k)
                t[i+k] = (t[i+k]<<1)|(t[i+k-1]>>7);
        for(i=0; i<200; i+=8)
            for(j = (i+8)%40, k=0; k<8; ++k)
                s[i+k] ^= t[j+k];

        /* Rho function */
        for(i=8; i<200; i+=8)
        {   for(j = Rho[i>>3]>>3, k=0; k<8; ++k)    /* j:=bytes to shift, s->t      */
                t[(k+j)&7] = s[i+k];
            for(j = Rho[i>>3]&7, k=7; k; --k)      /* j:=bits  to shift, t->s   */
                s[i+k] = (t[k]<<j) | (t[k-1]>>(8-j));
            s[i] = (t[0]<<j) | (t[7]>>(8-j));
        }

        /* Pi function */
        for(k=8; k<16; ++k) t[k] = s[k];        /* =memcpy(t+8, s+8, 8)                      */
        for(i=1; (j=Pi[i])>1; i=j)
            for(k=0; k<8; ++k)                      /* =memcpy(s+(i<<3), s+(j<<3), 8)   */
                s[(i<<3)|k] = s[(j<<3)|k];
        for(k=0; k<8; ++k)                          /* =memcpy(s+(i<<3), t+8, 8)            */
            s[(i<<3)|k] = t[k+8];

        /* Chi function */
        for(i=0; i<200; i+=40)
        {   for(j=0; j<40; ++j)
                t[j]=(~s[i+(j+8)%40]) & s[i+(j+16)%40];
            for(j=0; j<40; ++j) s[i+j]^=t[j];
        }

        /* Iota function */
        k = Iota[round];
        s[0] ^= k & 0x8B;           /* bits 0, 1, 3, 7 */
        s[1] ^= (k<<3)&0x80;        /* bit 15 */
        s[3] ^= (k<<2)&0x80;        /* bit 31 */
        s[7] ^= (k<<1)&0x80;        /* bit 63 */
```

```
    }
}

/* ----------------------------------------------------------------------
   32-bit version of Keccak_f(1600)
   ----------------------------------------------------------------------
*/
void Keccak_f_32(uint32 *s)
{   uint32 t[10];
    uint8 i, j, round, k;

    for(round=0; round<24; ++round)
    {   /* Theta function */
        for(i=0; i<10; ++i)
            t[i] = s[i] ^ s[10+i] ^ s[20+i] ^ s[30+i] ^ s[40+i];
        for(i=0; i<5; ++i)
            for(j=8, k=2; ; j%=10, k=(k+2)%10)
            {   *s++ ^= t[j++] ^ ((t[k]<<1)|(t[k+1]>>31));
                *s++ ^= t[j++] ^ ((t[k+1]<<1)|(t[k]>>31));
                if(j==8) break;
            }
        s -= 50;

        /* Rho function */
        for(i=2; i<50; i+=2)
        {   k = Rho[i>>1] & 0x1f;
            t[0] = (s[i+1] << k) | (s[i] >> (32-k));
            t[1] = (s[i] << k) | (s[i+1] >> (32-k));
            k = Rho[i>>1] >> 5;
            s[i] = t[1-k], s[i+1] = t[k];
        }

        /* Pi function */
        for(i=2, t[0]=s[2], t[1]=s[3]; (j=(Pi[i>>1]<<1))>2; i=j)
            s[i]=s[j], s[i+1]=s[j+1];
        s[i]=t[0], s[i+1]=t[1];

        /* Chi function */
        for(i=0; i<5; ++i, s+=10)
        {   for(j=0; j<10; ++j)
                t[j] = (~s[(j+2)%10]) & s[(j+4)%10];
            for(j=0; j<10; ++j)
                s[j] ^= t[j];
        }
        s -= 50;

        /* Iota function */
        t[0] = Iota[round];
        s[0] ^= (t[0] | (t[0]<<11) | (t[0]<<26)) & 0x8000808B;
        s[1] ^= (t[0]<<25) & 0x80000000;
    }
}
```

Annex F (informative): Change history

Change history							
Date	**TSG #**	**TSG Doc.**	**CR**	**Rev**	**Subject/Comment**	**Old**	**New**
Dec 2013					*Version after approval*	*1.1.0*	*12.0.0*
Dec 2013					*Update in Introduction with the spec numbers*	*12.0.0*	*12.0.1*
Sep-2014	*SP-64*	*SP-140316*	*001*	*2*	*Overall editorial modification to the Tuak specification TS 35.231*	*12.0.1*	*12.1.0*

History

Document history		
V12.1.0	October 2014	Publication