ETSI TS 135 235 V19.0.0 (2026-01)

TECHNICAL SPECIFICATION

5G;
Specification of the MILENAGE-256 algorithm set;
An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f3, f4, f5, f5* and f5;**
Document 2: algorithm specification
(3GPP TS 35.235 version 19.0.0 Release 19)

Reference
DTS/TSGS-0335235vj00

Keywords
5G,SECURITY

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 - APE 7112B
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° w061004871

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are 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 IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, 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.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM**® and the GSM logo are trademarks registered and owned by the GSM Association.

Legal Notice

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. These shall be interpreted as being references to the corresponding ETSI deliverables.

The cross reference between 3GPP and ETSI identities can be found at 3GPP to ETSI numbering cross-referencing.

Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**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.

In the present document, modal verbs have the following meanings:

shall indicates a mandatory requirement to do something

shall not indicates an interdiction (prohibition) to do something

The constructions "shall" and "shall not" are confined to the context of normative provisions, and do not appear in Technical Reports.

The constructions "must" and "must not" are not used as substitutes for "shall" and "shall not". Their use is avoided insofar as possible, and they are not used in a normative context except in a direct citation from an external, referenced, non-3GPP document, or so as to maintain continuity of style when extending or modifying the provisions of such a referenced document.

should indicates a recommendation to do something

should not indicates a recommendation not to do something

may indicates permission to do something

need not indicates permission not to do something

The construction "may not" is ambiguous and is not used in normative elements. The unambiguous constructions "might not" or "shall not" are used instead, depending upon the meaning intended.

can indicates that something is possible

cannot indicates that something is impossible

The constructions "can" and "cannot" are not substitutes for "may" and "need not".

will indicates that something is certain or expected to happen as a result of action taken by an agency the behaviour of which is outside the scope of the present document

will not indicates that something is certain or expected not to happen as a result of action taken by an agency the behaviour of which is outside the scope of the present document

might indicates a likelihood that something will happen as a result of action taken by some agency the behaviour of which is outside the scope of the present document

might not indicates a likelihood that something will not happen as a result of action taken by some agency the behaviour of which is outside the scope of the present document

In addition:

is (or any other verb in the indicative mood) indicates a statement of fact

is not (or any other negative verb in the indicative mood) indicates a statement of fact

The constructions "is" and "is not" do not indicate requirements.

Introduction

The present document contains a 256-bit example of set of algorithms, collectively called MILENAGE-256, which may be used as the authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**. It is not mandatory to use the particular algorithms specified in this document – all eight functions are operator-specifiable rather than being fully standardised. Operators electing to employ this example set can further personalise the algorithms (as described in the text).

An additional function, f5**, which is optional to implement and use, is also provided. This function, when used, replaces the use of f5*, and then serves to protect against some new attacks that have been recently discovered.

The present document is one of four documents, which collectively comprise the entire specification of the example authentication and key generation algorithms. Namely:

- 3GPP TS 35.234 [2]: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 1: MILENAGE-256 General".
- **3GPP TS 35.235: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 2: MILENAGE-256 Algorithm Specification".**
- 3GPP TS 35.236 [3]: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 3: Implementors' Test and Design Conformance Test Data".
- 3GPP TR 35.937 [4]: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 4: Summary and Results of Design and Evaluation".

1 Scope

This document contains a detailed specification of the general framework for the MILENAGE-256 algorithm set, together with specification for the cryptographic kernel used to instantiate the algorithm set.

The main new requirement for the MILENAGE-256 algorithm set, compared to previous 3GPP authentication and key generation functions, is to provide a 256-bit target security level, mainly motivated by future proofing 3GPP networks in case larger scale quantum computers become practical in the future. While this level of security can already be provided by the previously defined TUAK algorithm set [10], having another algorithm set, based on a different cryptographic kernel, provides a fallback, in case of future advances in conventional (non-quantum-computing based) cryptanalysis of the hash-function Keccak, the kernel used in TUAK.

The framework for MILENAGE-256 largely mirrors that of the previously defined MILENAGE algorithm set [9], but with a few important differences as discussed later in the present document.

In terms of the cryptographic kernel, MILENAGE-256 requires the use of a kernel mapping 256-bit inputs to 256-bit outputs, under the control of a 256-bit secret key. The present document provides kernel based on direct use of the Rijndael-256-256 block cipher with 256-bit block and key size.

The reader may recall that Rijndael was the candidate algorithm selected by NIST as the Advanced Encryption Standard, though the option to use a 256-bit block size was not adopted as part of the NIST AES requirements [5].

The algorithm set is named MILENAGE-256 since the intention is to provide full 256-bit security, which requires the use of 256-bit keys. Nonetheless MILENAGE-256 supports both 128-bit and 256-bit keys, to facilitate the transition to 256-bit security. If MILENAGE-256 is employed in a context containing components that do not yet support 256-bit keys, 128-bit keys can initially be employed, though it is recommended that the implementation supports a mechanism for transitioning to 256-bit keys in the future.

Associated test data for the MILENAGE-256 example algorithm set appears in a partner document comprising detailed test data, covering modes and the example kernel, and general design conformance test data [3].

Provisions are further made so that operators who so desire can customise the algorithm set to different degrees, providing some level of separation/isolation between implementations used by different operators.

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 TR 21.905: "Vocabulary for 3GPP Specifications".

[2] 3GPP TS 35.234: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 1: MILENAGE-256 General".

[3] 3GPP TS 35.236: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 3: Implementors' Test and Design Conformance Test Data".

[4] 3GPP TR 35.937: "Specification of the MILENAGE-256 algorithm set: An example set of 256-bit 3GPP authentication and key generation functions f1, f1*, f2, f2, f3, f5, f5, f5* and f5**; Document 4: Summary and Results of Design and Evaluation".

[5] 3GPP TS 33.102: "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Security Architecture".

[6] 3GPP TS 33.105: "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Cryptographic Algorithm Requirements".

[7] The Advanced Encryption Standard (AES), NIST FIPS 197, NIST, 2001.

[8] Rijndael information page, NIST archived AES submissions, https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived- crypto-projects/aes-development#rijndael

[9] 3GPP TS 35.205: "3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; 3G Security; Specificati[-on 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 1: General".

[10] 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".

[11] R. Borgaonkar, "New Privacy Threat on 3G, 4G, and Upcoming 5G AKA Protocols", in Proceedings on Privacy Enhancing Technologies 2019(3):108-127. Also available at https://eprint.iacr.org/2018/1175.pdf (published online: July 2019).

[12] M. Brisfors, S. Forsmark, and E. Dubrova, "How Deep Learning Helps Compromising USIM", in P.-Y. Liardet, N. Mentens (Eds): Proceedings of the 19th Smart Card Research and Advanced Application Conference (CARDIS'2020), LNCS 12609, Springer Verlag, pp. 135-150.

[13] J. Daemen and V. Rijmen, "The design of Rijndael", Springer Verlag, 2002.

[14] Henri Gilbert, "The Security of One-Block-to-Many Modes of Operation", Proceedings of FSE 2003: 376-395.

[15] L. Goubin and J.-S. Coron, "On Boolean and arithmetic masking against differential power analysis", in Ç. K. Koç, C. Paar (Eds): Proceedings of CHES '00, LNCS 1965, Springer Verlag, pp. 231-237.

[16] L. Goubin and J. Patarin, "DES and differential power analysis", in CHES´'99, LNCS 1717, Springer Verlag, pp. 158-172.

[17] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side Channel Cryptanalysis of Product Ciphers", in Procedings of ESORICS'98, LNCS 1485, Springer Verlag, pp. 97-110.

[18] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", in N. Koblitz (Ed): Proceedings of CRYPTO'96, LNCS 1109, Springer Verlag, pp. 104-113.

[19] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis", in M. Wiener (Ed), Proceedings of CRYPTO '99, LNCS 1666, Springer-Verlag, pp. 388-397.

[20] A. Maximov and M. Näslund, "Security analysis of the Milenage-construction based on a PRF", Cryptology ePrint Archive, available at https://eprint.iacr.org/2023/607.

[21] T. S. Messerges, "Securing the AES finalists against Power Analysis Attacks", in B. Schneier (Ed): Proceedings of the Seventh Fast Software Encryption Workshop (FSE '00), LNCS 1978, Springer Verlag, pp. 150-164.

[22] R. Wang, H. Wang, E. Dubrova, and M. Brisfors, "Advanced Far Field EM Side- Channel Attack on AES", in Proceedings of the 7th ACM on Cyber-Physical System Security Workshop (CPSS '21), pp. 29-39.

[23] Shay Gueron, White Paper "Intel Advanced Encryption Standard (AES) New Instructions Set" https://www.intel.com/content/dam/doc/white-paper/advanced- encryption-standard-new-instructions-set-paper.pdf

3 Definitions of terms, symbols and abbreviations

3.0 Introductory information

The security architecture of the 3GPP system currently includes seven security functions ***f1, f1*, f2, f3, f4, f5,*** and ***f5****. These authentication and key generation functions reside in the domain of network operators. Accordingly, the functions are not fully standardised and individual operators may design and implement their own set. The algorithms specified in this document, collectively referred to as MILENAGE-256, provide an example set of functions for operators that prefer not to design their own algorithms.

One additional algorithm, labelled ***f5**,*** has been provided to protect against so-called re- synch attacks [9]. Use of this function, which is optional and decided by the operator, shall be mutually exclusive to the use of ***f5****.

The inputs and outputs of all eight algorithms are defined in clause 6.

3.1 Terms

For the purposes of the present document, the terms given in TR 21.905 [1] 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 [1].

3.2 Symbols

For the purposes of the present document, the following symbols apply:

{ }	Curly brackets are used to denote values of array type. See clause 3.2.4 for further details
0_A	The byte array of size 16 with all zeros, {0, 0,…, 0}
1_A	The byte array of size 16 with the integer 1 in the first byte,{1,0, … , 0}
2_A	The byte array of size 16 with the integer 1 in the first byte,{2,0, … , 0}
$=$	The assignment operator
$==$	The equality comparison operator, returns True or False
:=	The definition operator
$\oplus$	The bitwise exclusive-OR operation. For byte arrays, this operates over bytes in parallel
$a \gg l$	The integer *a*, shifted by *l* bits to the right (without rotation). This is equivalent to division by 2^l without remainder
$a \ll l$	The integer *a*, shifted by *l* bits to the left (without rotation). This is equivalent to multiplication by 2^l
$\|\|$	The concatenation of two operands. Concatenation of integers is done differently from concatenation of arrays
$[a \ldots b]$	The closed integer interval (*a* and *b* are included). Observe the special meaning when used in the context of arrays as described in clause 3.4.
$\mathbb{N}_n$	The set of natural numbers representable by *n* bits
$\{\mathbb{N}_n\}^k$	The set of arrays of size *k* containing natural numbers, each representable by *n* bits
$\{\mathbb{N}_n\}^{k,l}$	The set of matrices of size *k* rows by *l* columns containing natural numbers, each representable by *n* bits
$GF(2^8)$	The Galois-field with 2^8 elements
$\lfloor r \rfloor$	The floor function. Returns the largest integer, smaller than or equal to $r \in \mathbb{R}$
$a \bmod b$	The remainder of division when dividing *a* by *b*. Depending on the context, *a* and *b* can be integers or values of the field $GF(2^n)$ for some n. The result is considered to have the same bit-size as *b*
$\text{bin}_n(a)$	The *n*-bit representation of the integer *a*
length(*s*)	The size in bytes of the array of ASCII encoded characters (string) *s*
$\max(a, b, \ldots)$	The largest of the integer values *a*, *b*, ….
$\min(a, b, \ldots)$	The smallest of the integer values *a*, *b*, ….
$c\ ?\ x : y$	Selection operation, results in the value/expression *x* if condition *c* is true, and results in *y* otherwise
AES-n	AES with n-bit key
$\text{PRF}_\mathbf{K}$	Pseudo-random function defined by key **K**
Rijndael-b-n	The Rijndael block cipher with b-bit blocks and n-bit key

3.3 Abbreviations

For the purposes of the present document, the abbreviations given in TR 21.905 [1] and the following apply. An abbreviation defined in the present document takes precedence over the definition of the same abbreviation, if any, in TR 21.905 [1].

3GPP	3rd Generation Partnership Project
AES	Advanced Encryption Standard
AKA	Authentication and Key Agreement
ASCII	American Standard Code for Information Interchange DPA Differential Power Analysis
MAC	Message Authentication Code
MDPH	Merkle-Damgård with Permutation and Hirose compression function PRF Pseudo-random function
PRP	Pseudo-random permutation
UE	User Equipment
USIM	User Services Identity Module

3.4 Radix

Unless otherwise noted, integer values are represented in decimal. We use the prefix **0x** to indicate **hexadecimal** integers. Binary numbers are written $(a)_2$, for example $(101)_2 = 5$.

3.5 Bit ordering, arrays and related operations

This specification utilises different sets of integers, e.g. $\mathbb{N}_8$, $\mathbb{N}_{16}$ and $\mathbb{N}_{128}$ (the set of natural numbers representable with 8, 16 and 128 bits, respectively). For a number $n \in \mathbb{N}_d$, the **most significant bit** is denoted n_{msb}A bit is an element of $\mathbb{N}_1$.

EXAMPLE: The integer value of an integer $n \in \mathbb{N}_d$ having precisely one non-zero bit in bit position *I*, for some $i \in [0 \ldots d-1]$, is 2^i.

In the present document, there is a need to construct a byte $b \in \mathbb{N}_8$ from a collection of elements from smaller domains $\mathbb{N}_{n0}$, $\mathbb{N}_{n1}$, $\mathbb{N}_{n2}$, ... where $n_0 + n_1 + n_2 \ldots = 8$. The concatenation of $a_i \in \mathbb{N}_{ni}$, $i \in [0 \ldots k-\!-1]$, $n_i \in [1 \ldots 7]$ in to $b \in \mathbb{N}_8$ is denoted by

$b = a_{k-1} \parallel a_{k-2} \parallel \cdots \parallel a_1 \parallel a_0$,

where a_0 becomes the **least significant part** of b.

According to the conventions defined above, the set of arrays of size *k* of *n*-bit integers are denoted by $\{ \mathbb{N}_n \}^k$. An array $A \in \{ \mathbb{N}_n \}^k$ is indexed $\{A[0], A[1], \ldots, A[k-2], A[k-1]\}$, where $A[0]$ is the first element of the array. Every array is written from left to right, with the first element A[0] to the left.

EXAMPLE: A = {0, 1, 2, 3} is an array with A[0] = 0, A[1] = 1, and so forth.

Concatenation of arrays is done from left to right, meaning that the left-most operands are given the lower indices in the result.

EXAMPLE: Let A= {1, 2, 3, 4} and B = {5, 6, 7, 8}. Then C = A ‖ B, will be the array {1, 2, 3, 4, 5, 6, 7, 8}, i.e. C[0] = 1, C[1] = 2, …, C[4] = 5, …, C[7] = 8.

Observe that this is different to how integers are concatenated, as described above. To make the usage of the symbol || unambiguous, it is stressed that ‖ is to be understood as integer concatenation, if and only if the elements aj to be concatenated are given in the format $bin_{nj}(a_j)$ for some $n_j \in [1 \ldots 7]$. In all other cases, use of ‖ denotes array concatenation.

Contrary to the principle of concatenation, a sub-array is constructed as follows, given an array $A = \{ A[0], A[1], \ldots, A[k-2], A[k-1] \}$:

$A[a \ldots b] := \{ A[a], A[a+1], \ldots, A[b-1], A[b] \}$,

for $0 \leq a \leq b \leq k-1$. Thus, $A[a \ldots b]$ has b – a +1 elements.

$\{\mathbb{N}_n\}^*$ denotes the set of variable-length arrays whose elements are natural numbers representable by *n* bits.

EXAMPLE: An arbitrarily long array of message bytes is an element of $\{\mathbb{N}_8\}^*$, the set of variable-length arrays containing bytes as elements.

The PRF kernel specification of clause 11, for the purpose of implementing finite field arithmetic, sometimes interprets byte arrays as representing elements of an extension of the field $GF(2^8)$. All details of how to perform these operations are provided in said clause.

Conversions from arrays of smaller domains into arrays of larger domains (and vice versa) are **not** done in a consistent way. Both Little Endianness and Big Endianness are used throughout the present document. This is due to being backwards compatible with the existing 3GPP protocols, as well as keeping the order for externally defined algorithms. The specification aims to be very explicit about the bits and bytes ordering.

If byte array operands to the $\oplus$ operation are of different lengths, the shorter one shall be appended to the right, via the above defined || operation, by as many zero-valued bytes as needed to make the lengths equal, and the result of the operation shall then be interpreted as having the size of the larger operand.

EXAMPLE: If $A = \{A[0], A[1], A[2]\}$ and $B = \{B[0], B[1], B[2], B[3], B[4]\}$ then $A \oplus B$

$= (A \,||\, \{0, 0\}) \oplus B = \{A[0] \oplus B[0], A[1] \oplus B[1], A[2] \oplus B[2], B[3], B[4]\}$.

4 Structure

This specification is organised as follows:

- Clause 3 introduces symbols and notation used in the subsequent clauses.
- Clause 5 provides a summary of all variables (inputs, outputs, and intermediary values) used in the algorithm specification.
- Clause 6 defines the set of supported and allowed parameter sizes for the implementation of the algorithms.
- Clause 7 explains how the algorithms are designed as a framework in such a way that various "customising components" can be selected in order to customise the algorithm for a particular operator.
- Clause 8 defines the example algorithms.
- Clause 9 explains various options and considerations for implementation of the algorithms, including considerations to be borne in mind when modifying the customising components.
- Illustrative pictures are given in clause 10.
- Clauses 11 provide specification of the cryptographic kernel that is used in the definition of the example algorithms.

As a complement, Annexes of the test data document [3] contain source code in the C/C++ programming language.

5 List of variables

5.1 Size variables

Table 5.1-1: Size variables

Name	Comment
AK_{SZ}	The size in bytes of the anonymity key **AK** and the anonymity re-synch key **AK***.
CK_{SZ}	The size in bytes of the confidentiality key **CK**.
IK_{SZ}	The size in bytes of the integrity key **IK**.
K_{SZ}	The size in bytes of the subscriber key **K**.
MAC_{SZ}	The size in bytes of the authentication codes **MAC-A** and **MAC-S**.
$RAND_{SZ}$	The size in bytes of the random challenge **RAND**.
RES_{SZ}	The size in bytes of the signed response **RES**.
SQN_{SZ}	The size in bytes of the sequence number **SQN**.

5.2 Specified general AKA input/ouput variables

Table 5.2-1: Specified general AKA input/output variables

Name	Comment
AK	An anonymity key that is output by the function ***f5***.
AK*	An anonymity resynchronisation key that is output by the functions ***f5**** or ***f5***.**
AMF	Two bytes of authentication management field that is input to the functions ***f1*** and ***f1****.
CK	A confidentiality key that is the output of the function ***f3***.
IK	An integrity key that is the output of the function ***f4***.
K	A subscriber key that is an input to the functions ***f1***, ***f1****, ***f2***, ***f3***, ***f4***, ***f5***, ***f5, f5**** and ***f5*****.
MAC-A	A network authentication code that is output by the function ***f1***.
MAC-S	A resynchronisation authentication code that is output by the function ***f1****.
RAND	A random challenge that is an input to the functions ***f1***, ***f1****, ***f2***, ***f3***, ***f4***, ***f5***, ***f5**** and ***f5*****.
RES	A signed response that is an output of the function ***f2***.
SQN	A sequence number that is an input to either of the functions ***f1*** and ***f1****. (For ***f1**** this input is more precisely called $\mathbf{SQN_{MS}}$)

5.3 MILENAGE-256 specific input variables

Table 5.3-1: MILENAGE-256 input variables

Name	Comment	
ALGONAME	An array of bytes specifying a name of the complete algorithm set. The name shall be ASCII-coded and shall have at most 31 ASCII characters.	
f-index (or fi)	Index that labels the different constituent functions **f1*, f1, f2, f3, f4, f5**, **f5*** and **f5****. For brevity and subscripting, also denoted by fi.	
	f1* : fi = 0	**f4** : fi = 4
	f1 : fi = 1	**f5** : fi = 5
	f2 : fi = 2	**f5*** : fi = 6
	f3 : fi = 3	**f5**** : fi = 7
c_i	Where $i \in [0 \ldots 7]$. For each i, c_i is a 16-byte array that constitutes the operator-customisable constant associated with the specific AKA function having the corresponding f-index. Used during the computation of the constituent functions.	
OP	An array of 32 bytes consisting of the Operator Variant Algorithm Configuration Field. The value is decided by the operator.	
OP_C	An array of 32 bytes consisting of a computed value, derived from *OP, ALGONAME*, K_{SZ}, and **K**.	

5.4 Additional variables and functions used for MILENAGE-256 computation

The individual functions ***f1, f1*, f2, f3, f4, f5***, ***f5*** and ***f5***** in the MILENAGE-256 algorithm set are implemented using a common cryptographic kernel in the form of a pseudo-random function PRF. Additionally, the selected size of the input and output parameters of the algorithm set can vary. For each given (fixed) value of the set of selected parameter sizes, each of the individual functions ***f1, f1****, …, etc, is referred to as an *instance* of the common kernel. An instance is uniquely determined by the *f-index* of the ***f***-function being computed, combined with the set of input/output parameter sizes relevant for that ***f***-function.

EXAMPLE: ***f3*** and ***f4*** are considered separate instances, even if all their input/output parameters have the same sizes. Further, an ***f3*** implementation producing outputs of size n_1 is considered as a separate instance from an ***f3*** implementation producing outputs of size n_2, ($n_1 \neq n_2$), even if all other parameter sizes of the algorithm set are the same.

For notational convenience, the following variables, used during the computation of these instances, are defined.

Table 5.4-1: Additional variables and functions used for MILENAGE-256 computation

Name	Type	Comment
IN_i	$\{\mathbb{N}_8\}^{32}$	Where $i \in [0 \ldots 7]$. Instance-specific value constructed within the computation of the constituent functions.
OUT_i	$\{\mathbb{N}_8\}^{32}$	Where $i \in [0 \ldots 7]$. Instance-specific intermediate value from which the outputs of the constituent functions are obtained by truncating the array.
$TEMP$	$\{\mathbb{N}_8\}^{32}$	An array of bytes consisting of a temporary value constructed within the computation of the constituent functions.
V	$\{\mathbb{N}_8\}^{32}$	An array of bytes consisting of a temporary value used in the computation of OP_C.

NOTE: Specification of the kernel functions (PRF) in clause 11 also use a number of internal variables. Since the use of kernel functions specific to the present document is not mandatory, no listing of kernel-specific variables is produced here.

6 Algorithm inputs and outputs

Inputs to the different algorithms are given in tables 6-2 and 6-3, below, and outputs appear in table 5-4.

Table 1 describes the supported sizes for input and output variables whose sizes are allowed to vary. For instance, the byte-length of the subscriber key **K** shall be specified by the key size variable K_{SZ}, which shall take one of the two values 16 or 32 (bytes), corresponding to keys **K** of size 128-bits and 256-bits, respectively. Existing 3GPP specifications do not support all the variable sizes specified in table 1 but they are included for future flexibility in case future specifications want to support them. For 3GPP usage, adopting a specific size for any of the variable-length parameters **K, SQN, AK, AK*, RAND, RES, CK, IK, MAC-A** and **MAC-S** mentioned in the present document requires a 3GPP specification explicitly allowing the selected size.

In any particular implementation, it is recommended that the parameters have a fixed length, chosen in advance.

EXAMPLE: An operator could 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, the length variables are not specified as formal input arguments.

Refer to clause 9.3 for cases where it could be motivated to allow some parameters to vary in size during operation.

Table 6-1: Permitted/supported parameter size ranges

Name	Type	Permitted values	Comment
AK_{SZ}	$\mathbb{N}_4$	6, 7, …, 12	The size in bytes of the anonymity key **AK** and the anonymity resynchronisation key **AK***.
CK_{SZ}	$\mathbb{N}_6$	16, 17, ..., 32	The size in bytes of the confidentiality key **CK**.
IK_{SZ}	$\mathbb{N}_6$	16, 17, ..., 32	The size in bytes of the integrity key **IK**.
K_{SZ}	$\mathbb{N}_6$	16, 32	The size in bytes of the subscriber key **K**.
MAC_{SZ}	$\mathbb{N}_6$	8, 9, ..., 32	The size in bytes of the authentication codes **MAC-A** and **MAC-S**.
$RAND_{SZ}$	$\mathbb{N}_6$	16, 18, …, 32 (even values only)	The size in bytes of the random challenge **RAND**.
RES_{SZ}	$\mathbb{N}_6$	4, 5, …, 32	The size in bytes of the signed response **RES**.
SQN_{SZ}	$\mathbb{N}_4$	6, 7, …, 12	The size in bytes of the sequence number **SQN**.

Each of the size-parameters in table 6-1 is assigned a numerical value that specifies the size (in bytes) of the corresponding parameter. Entries in the "Permitted values" column list the candidate values available for assignment.

The shortest values supported above agree with the shortest values supported in TS 33.102 [5]. Regarding RES_{SZ}, TS 33.102 [5] allows the response **RES** to take any size in the range 4 to 16 bytes.

Table 6-2: Inputs to f1 and f1*

Name	Type	Comment
AMF	$\{\mathbb{N}_8\}^2$	Two bytes of authentication management field **AMF**[0], **AMF**[1].
K	$\{\mathbb{N}_8\}^{Ksz}$	An array of subscriber key material {**K**[0], …, **K**[$K_{SZ}-1$]}.
RAND	$\{\mathbb{N}_8\}^{RANDsz}$	An array of random challenge bytes {**RAND**[0],…,**RAND**[$RAND_{SZ}-1$]}.
SQN	$\{\mathbb{N}_8\}^{SQNsz}$	A sequence number that is an input to either of the functions ***f1*** and ***f1****. (For ***f1**** this input is more precisely called **SQN**$_{MS}$)

Table 6-3: Inputs to f2, f3, f4, f5, f5* and f5**

Name	Type	Comment
K	$\{\mathbb{N}_8\}^{Ksz}$	An array of subscriber key material {**K**[0],…,**K**[$K_{SZ}-1$]}.
RAND	$\{\mathbb{N}_8\}^{RANDsz}$	An array of random challenge bytes {**RAND**[0],…,**RAND**[$RAND_{SZ}-1$]}.
MAC-S	$\{\mathbb{N}_8\}^{MACsz}$	An array of bytes consisting of the resynchronisation authentication code, {**MAC-S**[0],…,**MAC-S**[MAC_{SZ}-1]}. This is only used as input to ***f5*****.

NOTE 1: Besides the inputs stated in tables 6-2 and 6-3, the outputs of each ***f***-function also dependent on the MILENAGE-256 framework-common parameters as specified in clause 5.3.

Table 6-4: f-function outputs

f-function	Output Name	Type	Comment
f1*	**MAC-S**	$\{\mathbb{N}_8\}^{MACsz}$	An array of bytes consisting of the resynchronisation authentication code, {**MAC- S**[0],…,**MAC-S**[$MAC_{SZ}-1$]}.
f1	**MAC-A**	$\{\mathbb{N}_8\}^{MACsz}$	An array of bytes consisting of the network authentication code, {**MAC-A**[0],…,**MAC- A**[$MAC_{SZ}-1$]}.
f2	**RES**	$\{\mathbb{N}_8\}^{RESsz}$	An array of bytes consisting of the response, {**RES**[0],…,**RES**[$RES_{SZ}-1$]}.
f3	**CK**	$\{\mathbb{N}_8\}^{CKsz}$	An array of bytes consisting of the confidentiality key, {**CK**[0],…,**CK**[$CK_{SZ}-1$]}.
f4	**IK**	$\{\mathbb{N}_8\}^{IKsz}$	An array of bytes consisting of the integrity key, {**IK**[0],…,**IK**[$IK_{SZ}-1$]}.
f5	**AK**	$\{\mathbb{N}_8\}^{AKsz}$	An array of bytes consisting of the anonymity key, {**AK**[0],…,**AK**[$AK_{SZ}-1$]}.
f5*, f5**	**AK***	$\{\mathbb{N}_8\}^{AKsz}$	An array of bytes consisting of the resynchronisation anonymity key, {**AK***[0],…,**AK***[$AK_{SZ}-1$]}.

NOTE 2: ***f1**** need not always be computed. It is computed only when a resynchronisation procedure is needed.

NOTE 3: Both ***f5*** and ***f5**** outputs are called **AK** according to TS 33.102 [5]. A choice has been made in the present document to distinguish them as **AK** and **AK***, to avoid confusion. When a synchronisation failure occurs, then both the network as well as the UE will need to compute both **AK** and **AK***, in that order. When ***f5*** ** is used, this function is used instead of ***f5**** to derive **AK***.

All MILENAGE-256 ***f***-functions also depend on encoded versions of the size parameters for all variable-sized input/output variables relevant for the given function. Encoding methods for these size parameters and the ***f***-function dependence on these encodings are specified in clause 8.

EXAMPLE: Inputs to the function ***f1*** include the variables **K**, **RAND** and **SQN**. The function ***f1*** outputs the variable **MAC-A**. Each of these variables admits variable sizes, as specified by a corresponding size parameter. Accordingly, encodings of the size parameters K_{SZ}, $RAND_{SZ}$, SQN_{SZ}, and MAC_{SZ} are incorporated into the implementation of the function ***f1***, as described in clause 8.

7 The algorithm framework and the specific example algorithm

A complete instance of the MILENAGE-256 algorithm set shall be defined by the following components:

- A keyed PRF which takes an input $X \in \{\mathbb{N}_8\}^{32}$, a key $\mathbf{K} \in \{\mathbb{N}_8\}^{32}$ and returns an output $Y \in \{\mathbb{N}_8\}^{32}$, denoted $Y = \mathrm{PRF}(X)$.
- An up to thirty-two byte array (including a zero termination byte) ASCII character encoding *ALGONAME* specifying a name for the overall algorithm set. *ALGONAME* should be unique for each context in which the MILENAGE-256 framework is adopted.
- A specific set of parameter sizes compliant with the options defined in table 6-1.
- A value $OP \in \{\mathbb{N}_8\}^{32}$, which is an Operator Variant Algorithm Configuration Field (see clauses 8.1 and 9.1).
- For each **K**, one specific set of values assigned to the eight customisable constants $c_i \in \{\mathbb{N}_8\}^{16}$, $i \in [0 \ldots 7]$. The same set of values may be assigned for all **K**. The default recommendation is that operators employ the same set of c_i values for all keys **K**.

NOTE: A simple mechanism was included to allow personalisation of the algorithms, when used by different operators. Each operator can freely select their own value for OP. The algorithm set is designed to be secure whether or not OP is publicly known. However, operators could see some advantage in keeping their value of OP secret (see clause 9.1). The constants c_i allow a specific operator to further customise the algorithms (see clause 9.2). Equivalent security properties apply for the set of values c_i as for *OP*.

The intention is that a chosen set of c_i values enable operator-customisable separation among distinct ***f***-functions, whereas the separation between distinct subscribers is accomplished by their unique subscriber key **K**. This recommendation aligns with the fact that security of the algorithm set strictly requires distinct (secret) subscriber keys **K** for each subscriber but does not depend on the particular c_0 values selected, nor their secrecy. Operators may therefore set all c_i equal (or even set all c_i to zero) if the use-case context motivates such a simple choice. Nonetheless, operators may elect to use distinct c_i values to achieve more-granular customised ***f***-function separation, if desired. Default values for c_i and discussion on customisation is provided in clauses 8 and 9.

Though the c_i constants for the functions ***f5**** and ***f5***** are labelled distinctly, as c_6 and c_7, respectively, operators may select $c_6 = c_7$ since there is no need to create additional separation between ***f5**** and ***f5*****. The choice $c_6 = c_7$ is consistent with the requirement that operators shall use only one of the functions ***f5**** or ***f5*****, with the latter enabling protection against re-synch attacks. Nonetheless operators may customise the separation of ***f5**** and ***f5***** by using distinct c_6 and c_7, if desired.

The present document defines a default value for *ALGONAME*, intended for 3GPP usage. For usage in other contexts, it could be desirable to customise the algorithm set by assigning an alternative value for *ALGONAME*. However, except for specifying the format and maximum length, it is outside the scope of this document to define how this name is chosen in general.

EXAMPLE: *ALGONAME* could be assigned by a standardisation organisation that adopts MILENAGE-256 for usage.

The specific MILENAGE-256 example algorithm set defined in the present document includes one realisation of the kernel $\mathrm{PRF}_K(X)$ for completeness and to provide the potential to reuse existing cryptographic implementations. Nonetheless, the algorithm was deliberately designed to ensure that this component can be interchangeably replaced by any operator preferring to create a customised algorithm set, as long the selected PRF is secure. Hence, this document actually defines an algorithm framework, and the example algorithm fits within this framework. The complete algorithm is defined in clause 8. Specifically, in clause 8.1 the framework is defined in terms of a general 256-bit cryptographic kernel. Then, in clause 8.2, a particular block-cipher based kernel is selected, thereby providing a complete specification that fit in the aforementioned framework.

8 Definition of the example algorithm

8.1 General algorithm framework

The MILENAGE-256 algorithm set comprises eight functions, ***f1, f1*, f2, f3, f4, f5***, ***f5**** and ***f5*****. The function ***f5***** is provided as an option to protect against the resynchronisation attack [11] and shall then be used instead of ***f5****. Each of these eight functions is associated with an *f*-index, ranging from 0 to 7 (inclusive) and specified in clause 5.3. This *f*-index is below captured by the explicit variable *f-index*, or sometimes (for brevity of notation) in a variable *fi*. Below, the computation of these functions is presented in terms of an intermediate variable $TEMP$, a set of constructed input values $IN_i, i \in [0 \ldots 7]$, and corresponding output values OUT_i. The computation of ***f1, f1*, f2, f3, f4, f5***, ***f5**** and ***f5***** also depends on the 256- bit value OP_C, which shall be derived from OP and **K** as follows:

$$OPC := OP \oplus PRF\boldsymbol{K}(PRF\boldsymbol{K}(OP) \oplus V) \quad (EQ\ 1)$$

where,

$V[0]$ = 0x01 if Ksz == 32 and $V[0]$ = 0x00 if Ksz == 16.

$V[j]$ = *ALGONAME*[*j-1*] for j = 1,…, length(*ALGONAME*), and

$V[j]$ = 0x00 for j = length(*ALGONAME*) + 1, …, 31.

For 3GPP usage, *ALGONAME* shall be assigned the following (ASCII encoded) default value:

ALGONAME := "MILENAGE2.0".

NOTE: This implies that *ALGONAME*, encoded as an array of bytes, will be assigned values *ALGONAME*[0] = 77, *ALGONAME*[1] = 73, … , *ALGONAME*[9] = 46, *ALGONAME*[10] = 48, and length(*ALGONAME*) = 11.

Given OP_C computed according to (EQ 1), an intermediate byte array *TEMP* ∈ $\{\mathbb{N}8\}^{32}$ shall be then computed as:

$TEMP$:= PRF$_K$(***RAND*** ⊕ OP_C) ⊕ OP_C.

If **RAND** has fewer bytes than OP_C, i.e. if $RAND_{SZ} < 32$, zero-valued bytes shall be appended to **RAND**, as per definition of the ⊕ operation in clause 3.5, so that its length matches that of OP_C.

Computation of the ***f***-functions also uses an intermediate function MAKE_INS, which produces a one-byte output defined as follows:

MAKE_INS(*f-index*, $RAND_{SZ}$, K_{SZ}) :=

bin_3(*f-index*) ‖ bin_4(RAND$_{SZ}^{-2}$/2) ‖ $bin_1(K_{SZ} \gg 5)$.

Computing all of the ***f***-functions corresponds to computing eight output arrays OUT_i^*, $i \in [0 \ldots 7]$ which shall be defined as follows:

OUT_i: = PRF$_K$($TEMP \oplus IN_i$) ⊕ OPC, (EQ 2)

where INi ∈ {ℕ8}32, $i \in [0 \ldots 7]$, shall be as defined below.

8.2 Specification of individual functions

8.2.1 Default values for c0, …, c7

Default values (unless otherwise specified in the implementation) of c_0, $i \in [0 \ldots 7]$ shall be

$c_i[j] = 0$, for $j \in [0 \ldots 14]$ and,

$c_i[15] = (i == 0)\ ?\ 0 : (1 \ll (i - 1))$

NOTE: The values specified above follow the same pattern as the default values defined for MILENAGE-128.

8.2.2 Specification of the functions *f1 and f1**

For $i = fi \in [0, 1]$ (corresponding to the *f-index* of the function being computed), construct

IN_1 as follows. Set:

IN_1 [0] = MAKE_INS(fi, $RAND_{SZ}$, K_{SZ}),

IN_1 [1] = $bin_3(SQN_{SZ} - 5)$ ‖ $bin_5(MAC_{SZ} - 1)$,

IN_1 [2] = **AMF**[0],

IN_1 [3] = **AMF**[1],

IN_1 [4 + j] = **SQN**[j], $j \in [0 \ldots SQN_{SZ} - 1]$,

if $SQN_{SZ} < 12$, then $IN_i[4 + j] = 0$, $j \in [SQN_{SZ} \ldots 11]$,

$IN_i[16 + j] = c_i[j]$, $j \in [0 \ldots 15]$.

To compute ***f1*** (**MAC-A)**:

Use IN_1 to compute OUT_1 according to (EQ 2) of clause 8.1.1.

Take **MAC-A**$[0 \ldots MAC_{SZ} - 1] = OUT_1[0 \ldots MAC_{SZ} - 1]$.

And, to compute ***f1**** (**MAC-S)**:

Use IN_0 to compute OUT_0 according to (EQ 2) of clause 8.1.1.

Take **MAC-S**$[0 \ldots MAC_{SZ} - 1] = OUT_0[0 \ldots MAC_{SZ} - 1]$.

8.2.3 Specification of the function *f2*

For *f-index* == 2, construct IN_2 as follows. Set:

IN_2 [0] = MAKE_INS(2, $RAND_{SZ}$, K_{SZ}),

IN_2 [1] = $bin_3(0) \parallel bin_5(RES_{SZ} - 1)$,

IN_2 [2 … 15] = 0,

IN_2 [16 + j] = $c_2[j]$, $j \in [0 \ldots 15]$.

To compute ***f2*** **(RES)**:

Use IN_2 to compute OUT_2 according to (EQ 2) of clause 8.1.1.

Take **RES**$[0 \ldots RES_{SZ} - 1] = OUT_2[0 \ldots RES_{SZ} - 1]$.

8.2.4 Specification of the function *f3*

For *f-index* == 3, construct IN_3 as follows. Set:

$IN_3[0]$ = MAKE_INS(3, $RAND_{SZ}$, K_{SZ}),

IN_3 [1] = $bin_3(0) \parallel bin_5(CK_{SZ} - 1)$,

IN_3 [2 … 15] = 0,

IN_3 [16 + j] = $c_3[j]$, $j \in [0 \ldots 15]$.

To compute ***f3*** (**CK)**:

Use IN_3 to compute OUT_3 according to (EQ 2) of clause 8.1.1.

Take **CK**$[0 \ldots CK_{SZ} - 1] = OUT_3[0 \ldots CK_{SZ} - 1]$.

8.2.5 Specification of the function *f4*

For *f-index* == 4, construct IN_4 as follows. Set:

$IN_4[0]$ = MAKE_INS(4, $RAND_{SZ}$, K_{SZ}),

IN_4 [1] = $bin_3(0) \parallel bin_5(IK_{SZ} - 1)$,

IN_4 [2 … 15] = 0,

IN_4 [16 + j] = $c_4[j]$, $j \in [0 \ldots 15]$.

To compute ***f4*** (**IK)**:

Use IN_4 to compute OUT_4 according to (EQ 2) of clause 8.1.1.

Take **IK**$[0 \ldots IK_{SZ} - 1] = OUT_4[0 \ldots IK_{SZ} - 1]$.

8.2.6 Specification of the function *f5*

For *f-index* == 5, construct OUT_5 as follows. Set:

$IN_5[0]$ = MAKE_INS(5, $RAND_{SZ}$, K_{SZ}),

$IN_5[1] = bin_5(0) \parallel bin_3(AK_{SZ} - 5)$,

$IN_5[2 \dots 15] = 0$,

$IN_5[16 + j] = c_5[j],\ j \in [0 \dots 15]$.

To compute ***f5*** (**AK)**:

Use IN_5 to compute OUT_5 according to (EQ 2) of clause 8.1.1.

Take **AK**$[0 \dots AK_{SZ} - 1] = OUT_5[0 \dots AK_{SZ} - 1]$.

8.2.7 Specification of the function *f5** and *f5***

The function ***f5**** or the function ***f5***** shall be used to generate **AK***. The function ***f5***** shall replace the function ***f5**** when re-synch attack prevention is enabled.

If implementing ***f5****, for *f-index* == 6, construct IN_6 as follows. Set:

$IN_6[0]$ = MAKE_INS(6, $RAND_{SZ}$, K_{SZ}),

$IN_6[1] = bin_5(0) \parallel bin_3(AK_{SZ} - 5)$,

$IN_6[2 \dots 15] = 0$,

$IN_6[16 + j] = c_6[j],\ j \in [0 \dots 15]$.

To compute ***f5**** (**AK*)**:

Use IN_6 to compute OUT_6 according to (EQ 2) of clause 8.1.1.

Take **AK***$[0 \dots AK_{SZ} - 1] = OUT_6[0 \dots AK_{SZ} - 1]$.

Alternatively, if implementing ***f5*****, for *f-index* == 7, construct IN_7 as follows. Set:

$IN_7[0]$ = MAKE_INS(7, $RAND_{SZ}$, K_{SZ}),

- $IN_7[1] = bin_5(MAC_{SZ} - 1) \parallel bin_3(AK_{SZ} - 5)$,
- $IN_7[2 \dots 15] = 0$,
- $IN_7[16 + j] = c_7[j],\ j \in [0 \dots 15]$,
- Additionally, set
  - $IN_7[j + 2] = IN_7[j + 2] \oplus$ **MAC-S**$[j]$, for $j \in [0 \dots \min(29, MAC_{SZ} - 1)]$.

NOTE: The effect of min(29, MAC_{SZ} -1) in the last line above has the effect of truncating **MAC-S** values that are larger than 240 bits. This is done to ensure instance separation relative to the other ***f***-functions and relative to implementations with other parameter settings by leaving the first two bytes of IN_B unaffected by **MAC-S**.

To compute ***f5***** (**AK*)**:

- Use IN_7 to compute OUT_7 according to (EQ 2) of clause 8.1.1.
- Take **AK***$[0 \dots AK_{SZ} - 1] = OUT7[0 \dots AK_{SZ} - 1]$.

8.3 Comments on the *f*-function specifications

Above, for any value *fi* = *f-index* ∈ [0 ... 7], the first byte of IN_0, namely $IN_i[0]$, is produced by MAKE_INS and always given by:

$IN_i[0] = bin_3(fi) \parallel bin_4((RAND_{SZ} - 2)/2) \parallel bin_1(K_{SZ} \gg 5)$,

where the three components encode the invoked function, the random element size, and the subscriber key size, respectively. The other size parameters, namely

SQN_{SZ}, MAC_{SZ}, RES_{SZ}, CK_{SZ}, IK_{SZ}, and AK_{SZ}, are incorporated into the functions ***f1***, ***f1****, ***f2***, ***f3***, ***f4***, ***f5*** , ***f5**** and ***f5***** via the byte $IN_i[1]$. This is summarised in table 8.3-1 below.

Table 8.3-1: Bit positioning for the first two bytes of the *f*-functions

***f*-function**	$IN_i[0] =$ **MAKE_INS**$(i, RAND_{SZ}, K_{SZ})$								$IN_i[1]$							
	bit position								**bit position**							
	7	6	5	4	3	2	1	0*	7	6	5	4	3	2	1	0
f1*	f-index = 0			$(RAND_{SZ} - 2)/2$				0/1	$SQN_{SZ} - 5$			$MAC_{SZ} - 1$				
f1	f-index = 1			$(RAND_{SZ} - 2)/2$				0/1	$SQN_{SZ} - 5$			$MAC - 1$				
f2	f-index = 2			$(RAND_{SZ} - 2)/2$				0/1	0	0	0	$RES_{SZ} - 1$				
f3	f-index = 3			$(RAND_{SZ} - 2)/2$				0/1	0	0	0	$CK_{SZ} - 1$				
f4	f-index = 4			$(RAND_{SZ} - 2)/2$				0/1	0	0	0	$IK_{SZ} - 1$				
f5	f-index = 5			$(RAND_{SZ} - 2)/2$				0/1	0	0	0	0	0	$AK_{SZ} - 5$		
f5*	f-index = 6			$(RAND_{SZ} - 2)/2$				0/1	0	0	0	0	0	$AK_{SZ} - 5$		
f5**	f-index =7			$(RAND_{SZ} - 2)/2$				0/1	$MAC_{SZ} - 1$					$AK_{SZ} - 5$		

*) For bit-position 0 of the byte $IN_i[0]$, the value 1 appears if $K_{SZ} == 32$, otherwise a 0 appears.

This approach ensures that the input for every ***f***-function encodes the length-values of all input/output parameters relevant for that specific function. Such encodings provide a simple check for developers to ensure their construct aligns with intended parameter-size usage.

Moreover, the encodings provide additional cryptographic separation between implementations using different values for the parameter sizes: even if two implementations use the same key, the outputs will still differ, if the corresponding input/output length parameters differ.

Also, for all *i* values, the sixteen bytes $IN_i[16 \ldots 31]$ are dependent on the eight 16-byte, operator-selectable (arbitrary) customisation values, $c_0, c_1, \ldots, c_7$:

$IN_i[16 + j] = c_i[j]$, $j \in [0 \ldots 15]$, and $i \in [0 \ldots 7]$.

NOTE 1: Unlike MILENAGE, the security of MILENAGE-256 is not affected if all c_i values are identical. This property holds since the encoding of the *f*-index, $bin_3(fi)$ within $IN_i[0]$, always takes a distinct value for the particular ***f***- function computed. When the PRF is a permutation, this specificity provides cryptographically guaranteed separation between un-truncated output values of all ***f***-functions produced by a given fixed implementation, for a given key and given input values. Operators may nonetheless wish to select non-identical c_i values to further customise the separation among the ***f***- functions.

For $i == 7$, the computation of $IN_7[2 \ldots 31]$ also depends on the **MAC-S** value used during re-synchronisation.

NOTE 2: Observe that if MAC_{SZ} is greater than 29 bytes (240 bits), **MAC-S** will be truncated before inclusion in IN_7.

8.4 Specific example algorithm

Specifying a particular value for the kernel function $\mathrm{PRF}_{\mathbf{K}}$ defines a concrete example set of authentication and key generation algorithms. In this clause, one block cipher based kernel is specified, thereby fully defining one concrete example algorithm set (referred to as MILENAGE-256).

The kernel is required to produce outputs which are qualitatively pseudo-random (i.e., indistinguishable from outputs of a randomly chosen function) [2]. The kernels defined *is* a permutation.

8.4.1 MILENAGE-256: The Rijndael-256-256 PRF kernel

The kernel selected as part of MILENAGE-256 is the block cipher Rijndael-256-256, as defined in clause 11. Recall that AES-(128/192/256) is identical to Rijndael-128- (128/192/256), the algorithm originally proposed (and ultimately successful) as the Advanced Encryption Standard [5]. MILENAGE-256 employs Rijndael-256-256, which uses a 256-bit key and operates on 256-bit blocks. Hence, for MILENAGE-256 the PRF shall be defined as:

$\mathrm{PRF}_{\mathbf{K}}(X)$ = the result of applying the Rijndael-256-256 encryption algorithm to the 256-bit value *X* under a 256-bit key **K**.

NOTE 1: In this specific case, the kernel PRF-function is also a permutation (a one-to- one mapping) which is widely held to be a pseudo-random permutation, PRP. Although it has been indicated above that some properties (e.g. collision probabilities between outputs) do depend on whether the kernel is one-to-one or not, this dependency is not critical for the overall security of MILENAGE-256. The original security proof for the previous MILENAGE algorithm-set [9] was constructed under the assumption of the kernel being a PRP.

A complete specification of Rijndael-256-256 is given in clause 11 of this document.

NOTE 2: The specification in clause 11 is identical to that provided in the original Rijndael submission to NIST as candidate for the AES. Since NIST did not include the 256-bit block option in the AES standard, the complete specification is reproduced in the present document, for self-containment purposes.

9 Implementation considerations

9.1 OP_C computed on or off the USIM

Recall that OP is an Operator Variant Algorithm Configuration Field. It is anticipated that each operator will define a value of OP which may then be used for all its subscribers.

However, individual operators may freely decide how to manage OP. For example, the value of OP could be changed for new batches of USIMs or perhaps a different OP value could be given to each different USIM supplier. (Note that, strictly speaking, the MILENAGE-256 specifications even permit operators to assign different values of OP to every individual subscriber, if desired, but such a fine-grained assignment is not really the intention.)

As shown in clause 8.1, OP_C is computed from OP and **K**, and only OP_C (not OP) is used in subsequent computations. This avails two options for implementing the algorithms on the USIM:

a) **OP_C computed off the USIM:** OP_C is computed as part of the USIM pre- personalisation process, and only OP_C is stored on the USIM. OP itself is not stored on the USIM.

b) **OP_C computed on the USIM:** OP is stored on the USIM (perhaps considered as a hard-coded part of the algorithm, if preferred). OP_C is recomputed each time the algorithms are called.

It is recommended that OP_C be computed off the USIM, where possible, owing to the following benefits:

The complexity of the algorithms run on the USIM is reduced.

It is more likely that OP can be kept secret. If OP is stored on the USIM, it is sufficient for a single USIM to be reverse engineered to allow OP to be discovered and published. However, unlike OP, which may take the same value for all subscribers of a given operator, OP_C is different for each individual subscriber (provided they have distinct keys **K**, as recommended). Moreover, it is a formal cryptographic requirement of the MILENAGE-256 algorithms that it should be very difficult for attackers who discover even a large number of (OP_C, **K**) pairs to deduce OP. Consequently, even if an attacker possesses many (OP_C, **K**) pairs, the OP_C value associated with any other value of **K** is expected to remain unknown, with very high probability. Recall that the algorithms are designed to be secure whether or not OP is known to an attacker. Nonetheless a secret OP value provides an additional hurdle in an attacker's path, which may make it more difficult to successfully mount some kinds of cryptanalytic and forgery attacks.

A difference relative to the former MILENAGE specification is that OP_C in MILENAGE-256 now depends on the key size parameter, K_{SZ}. This dependence decreases the chances that identical OP_C-values are computed for keys of different key sizes.

9.2 Key and parameter sizes

MILENAGE-256 can support a wide range of input/output parameter sizes. Design choices were made to avoid accidental (partial) collisions between security-critical output values (e.g. session keys) generated for different choices of parameter sizes.

EXAMPLE: For a fixed value of the subscriber key **K**, two same-sized **CK** outputs generated from **RAND**-values of different sizes are unlikely to be identical, even if one **RAND** is a proper prefix of another, due to the dependence of ***f3*** on $RAND_{SZ}$. Furthermore, an n_1-bit **CK** output is unlikely (except "by chance") to be a prefix of n_2-bit **CK**, $n_1 < n_2$, even if all other input values are the same, since ***f3*** also depends on CK_{SZ}. If the kernel is not a PRP there will be a small probability for collisions, such as two same-sized **CK** outputs using different **RAND**-values being identical, but this does not reduce security in any substantial way.

Nevertheless, it is generally recommended that implementations are fixed to support one set of parameter sizes. Exceptions could be motivated by security upgradeability, in particular regarding the size of the subscriber key **K** and the **RAND** value, which directly impact the overall security:

MILENAGE-256 supports both 128- and 256-bit keys, though use of 256-bit keys is encouraged. Use of a shorter key could, however, be motivated for flexibility reasons, e.g., while network nodes or other parts of the SIM ordering / personalisation process are limited to 128-bits. If network limitations require a 128-bit key to be used initially, it would be advantageous for USIM manufacturers and operators to include a mechanism allowing upgrade to 256-bit keys once those limitations are removed.

Current 3GPP specifications only support 128-bit **RAND** values, while a 256-bit **RAND** would be necessary for MILENAGE-256 to reach full 256-bit level security with respect to some attacks. MILENAGE-256 can support 256-bit **RAND** values if/when networks are similarly upgraded. Incorporating a mechanism to remotely enable the use of larger **RAND** values could therefore be beneficial.

NOTE: Use of 128-bit key does not provide backward compatibility with the existing MILENAGE [9] algorithm set.

9.3 Further considerations

As mentioned in clause 7.3, individual operators wishing to further customise the algorithms have an additional simple tool at their disposal: namely they may select different values for the constants c_0 to c_7. There are no specific requirements on making secure choices of the constants c_i; all c_i values may be identical if so desired. Furthermore, the algorithms are designed to be secure even if an operator's chosen c_i values are public. Nonetheless operators may see some advantage in keeping their selected c_i values private, since this provides an additional hurdle in an attacker's path.

Distinct approaches for selecting sets of c_i values produce different levels of customisation. Employing the same set of c_i values for all subscribers (equivalently, for all keys **K**) provides additional personalisation of an operator's entire algorithm set, relative to other operators.

Finer-grained customisation is achieved when operators select different sets of c_i values for different sets of subscribers, thereby generating customised separation of the algorithm set among subscribers of a given operator.

EXAMPLE: Operators could select distinct sets of c_i values to customise the separation of the algorithm set for different batches of USIMs or for USIMs received from different suppliers.

The functionality afforded by the choice of *OP* and the choice of c_i values overlaps. Specifically, in MILENAGE-256, both *OP* and the c_i constants offer operator personalisation that may be applied uniformly to all subscribers (by employing one set of values for all subscribers) or non-uniformly to subsets of subscribers (by employing distinct sets of values for different subscribers). Combinations can also be applied, such as fixing *OP* for a batch of USIMS and personalising the ***f***-functions among subscribers within that batch by selecting distinct c_i values. The main distinctions between *OP* and c_i constant functionality are:

OP may be stored off the SIM with *OPc* serving as an encoded proxy on the SIM. There is no in-built functionality in MILENAGE-256 to similarly encode the c_i values.

Both the 256-bit sized *OP* and the 128-bit constants c_i *could* be assigned individual values for all individual subscribers (or subsets of subscribers), if desired, though this is not required, nor is it really the intention.

The personalisation offered by *OP* applies to the algorithm set as a whole whereas the constants c_i permit personalised separation among the distinct functions ***f1***, ***f1***, …, ***f5***, ***f5**** and ***f5*****.

9.4 Resistance to side channel attacks

When these algorithms are implemented on a USIM, consideration should be given to protecting them against side chaNnel attacks such as differential power analysis (DPA); in this regard, multiple implementation considerations are elaborated in the literature [15, 16, 17-29, 21]. In general, countermeasures to side-channel attacks on AES will also be applicable to Rijndael-256-256. Where applicable, protection against emerging artificial intelligence-based or machine learning-based attacks should also be considered, such as the far-field attacks on AES / USIM implementations [12, 22].

10 Figure of the algorithms (informative)

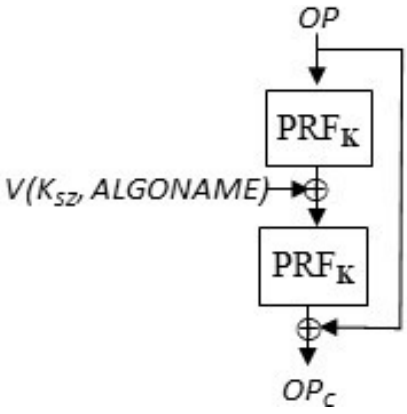

Figure 10-1: Computation of OPC.

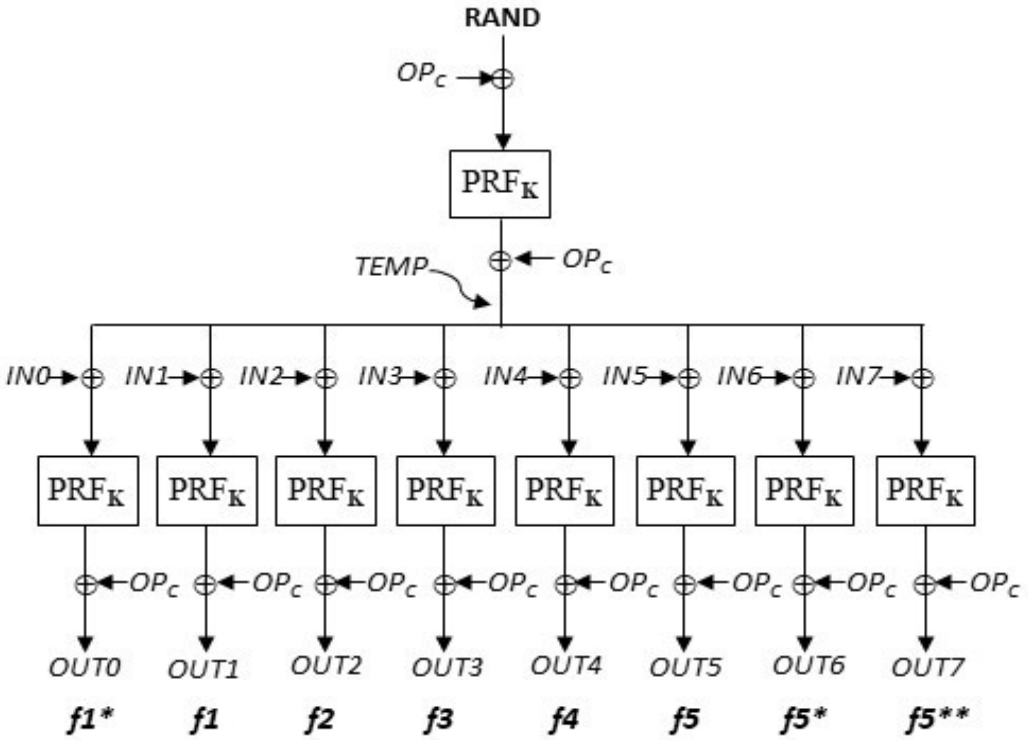

Figure 10-2: Overview of the f-algorithm set.

11 Specification of the Rijndael-256 based kernel function

The present clause contains a specification for the kernel function, which employs the Rijndael-256-256 block cipher. The complete specification of the Rijndael family of block ciphers [8, 9] notes that:

> "Rijndael is an iterated block cipher with a variable block length and a variable key length. The block length and the key length can be independently specified to 128, 192 or 256 bits."

For present 3GPP purposes, Rijndael is used only in encryption mode with the block and key length both set to 256-bits. For the remainder of this clause, it is simply written Rijndael-256, rather than Rijndael-256-256, to refer to the Rijndael block cipher with 256-bit block length and 256-bit key length.

NOTE: Although the presentation in this clause refers to Rijndael-256, multiple aspects of the presentation apply more generally to the full class of Rijndael block ciphers.

The present document describes a simple byte-oriented implementation of the Rijndael-256 encryption mode. Interested readers can find additional details regarding the cipher design and/or implementation speed-ups [8, 13, 23].

11.1 The state and external interfaces of Rijndael-256

Rijndael-256 involves a series of rounds that sequentially transform the initial input into the final output. An intermediate result of this encryption process is called the *State*. The *State* can be viewed as an $\{\mathbb{N}_8\}^{4,8}$ matrix of bytes (giving 256-bits in total). The PRF key is similarly viewed as an element of $\{\mathbb{N}_8\}^{4,8}$.

NOTE: Although Rijndael-256 is a block cipher (a permutation), since the present document uses Rijndael-256 as a PRF, the term PRF key is used in the sequel instead of the otherwise more common term Cipher key.

These representations are illustrated in figure 11.1-1.

$a_{0,0}$	$a_{0,1}$	…	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	…	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	…	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	…	$a_{3,7}$

$k_{0,0}$	$k_{0,1}$	…	$k_{0,7}$
$k_{1,0}$	$k_{1,1}$	…	$k_{1,7}$
$k_{2,0}$	$k_{2,1}$	…	$k_{2,7}$
$k_{3,0}$	$k_{3,1}$	…	$k_{3,7}$

Figure 11.1-1: Example of state and cipher key (PRF key) layout

The Rijndael-256 PRF takes an array of input bytes

$X \in \{\mathbb{N}\}^{32} = \{ X[0], X[1], \ldots, X[31] \}$

and a PRF key array

$\boldsymbol{K} \in \{\mathbb{N}_8\}^{32} = \{ \boldsymbol{K}[0], \boldsymbol{K}[1], \ldots, \boldsymbol{K}[31] \}$,

and generates an array of output bytes

$Y \in \{\mathbb{N}_8\}^{32} = \{ Y[0], Y[1], \ldots, Y[31] \}$.

The input (corresponding to "plaintext", when viewed as a block cipher) bytes are mapped onto the state bytes in the order

$a_{0,0} = X[0]$, $a_{1,0} = X[1]$, $a_{2,0} = X[2]$, $a_{3,0} = X[3]$, $a_{0,1} = X[4]$, $a_{1,1} = X[5]$, $a_{2,1} = X[6]$, $a_{3,1} = X[7]$, ...,

and the PRF key bytes are taken in the order

$k_{0,0} = \boldsymbol{K}[0]$, $k_{1,0} = \boldsymbol{K}[1]$, $k_{2,0} = \boldsymbol{K}[2]$, $k_{3,0} = \boldsymbol{K}[3]$, $k_{0,1} = \boldsymbol{K}[4]$, $k_{1,1} = \boldsymbol{K}[5]$, $k_{2,1} = \boldsymbol{K}[6]$, $k_{3,1} = \boldsymbol{K}[7]$, ...

When the Rijndael-256 based PRF operation concludes, the resulting output is extracted from the *State* and placed in *Y* by taking the *State* bytes in the same order:

$Y[0] = a_{0,0}$, $Y[1] = a_{1,0}$, …

Hence, if the one-dimensional index of a byte within a block or within the key is *n* and the two-dimensional index of the state is (*i*, *j*), we have the following mapping between byte index and the position within the *State*:

$i = n \bmod 4$; $j = \lfloor n/4 \rfloor$; $n = i + 4 * j$.

11.2 Internal structure

Rijndael-256 consists of the following operations (in stated order):

- An initial *Round Key* addition.
- 13 rounds, numbered 1-13, each consisting of:
  - A byte substitution transformation.
  - A shift row transformation.
  - A mix column transformation.
  - A *Round Key* addition.
- A final round (round 14) consisting of:
  - A byte substitution transformation.
  - A shift row transformation.
  - A Round Key addition.

The component transformations and the relationship between the *Round Keys* and the PRF key **K** are specified in the following subclauses.

In the following, when describing an operation on the *State* $\{ a_{i,j} \mid 0 \le i \le 3,\ 0 \le j \le 7 \}$ denotes the *State* before the operation and $\{ b_{i,j} \mid 0 \le i \le 3,\ 0 \le j \le 7 \}$ denotes the resulting *State* after the operation.

11.3 The byte substitution transformation

The byte substitution transformation is a non-linear byte substitution that operates independently on each byte in the *State*. The substitution table (S-box) employed for this transformation is presented in clause 11.8. For every element (i.e. byte) in the *State***,** one applies the transformation:

$b_{i,j}$ = S-box[$a_{i,j}$],

where ai,j denotes the (i,j) element in the current State and bi,j denotes the corresponding output value, which forms the (i,j) element of the new (or transformed) State. S-Box[] denotes the substitution transformation, defined by the table in clause 11.8.

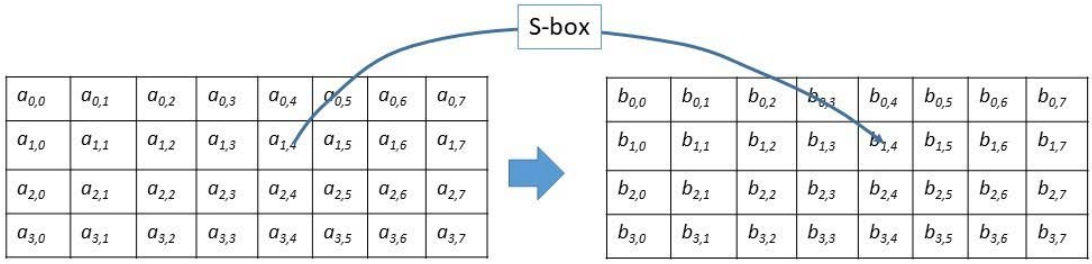

Figure 11.3-1: The State before and after byte substitution, the S-box is applied per entry (though only one application is shown)

11.4 The shift row transformation

The shift row transformation shifts the last three rows of the *State* cyclically to the left, with each shifted row being shifted by a different offset. Specifically, row 0 is not shifted, row 1 is shifted by 1 byte, row 2 by 3 bytes and row 3 by 4 bytes.

EXAMPLE: Bytes $a_{2,7}$ and $a_{3,3}$ in the current *State* are shifted to bytes in position (2,4) and (3,7) in the new *State*.

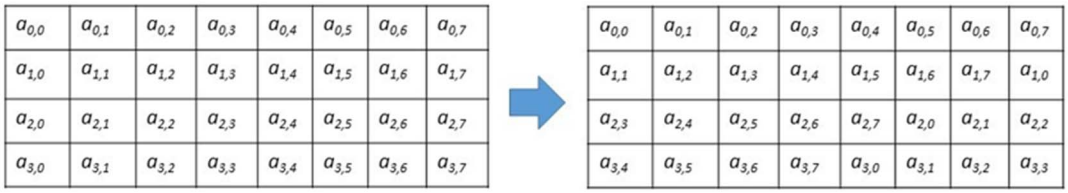

Figure 11.4: The *State* before and after shift row

11.5 The mix column transformation

The mix column transformation operates on each column of the *State* independently. Each column of the *State* is formally treated as a polynomial with coefficients over $GF(2^8)$ and multiplied by the fixed polynomial $c(x) =$ '$0x03$'x^3+'$0x0$'x^2+'$0x01$'$x + '0x02'$, modulo $x^4 + 1$. Here, the constant term coefficient of $c(x)$, '$0x02$' (an element of $GF(2^8)$) is defined to correspond to the polynomial y and the coefficient of x^3, i.e. '$0x03$' corresponds to the polynomial $y + 1$, and so on. Rijndael-256 here defines $GF(2^8)$ through reductions modulo the irreducible polynomial $p(y) = y^8 + y^4 + y^3 + y + 1$. It can be observed that $p(y)$ similarly can be written as '$0x11$b', which corresponds to an integer value of 283.

The action of the mix column transformation, on column j, for $j = 0, 1, 2, 3$ can now be expressed as

$$b_{0,j} = T_{02}(a_{0,j}) \oplus T_{03}(a_{1,j}) \oplus a_{2,j} \oplus a_{3,j}$$
$$b_{1,j} = a_{0,j} \oplus T_{02}(a_{1,j}) \oplus T_{03}(a_{2,j})) \oplus a_{3,j}$$

$$b_{2,j} = a_{0,j} \oplus a_{1,j} \oplus T_{02}(a_{2,j}) \oplus T_{02}(a_{3,j})$$
$$b_{3,j} = T_{03}(a_{0,j}) \oplus a_{1,j} \oplus a_{2,j} \oplus T_{02}(a_{3,j})$$

where T_i corresponds to the multiply-by '*i*' operation in GF(2^8), modulo $p(y)$ as defined above. This multiplication can in turn be converted into operations on integers in $\mathbb{N}_8$ as follows:

$T_{02}(a) = 2a$ if $a < 128$

or $T_{02}(a) = 2a \oplus 283$ if $a >= 128$

and $T_{0,3}(a) = T_{02}(a) \oplus a$.

Here, $a < 128$ means that the element a of $GF(2^8)$, interpreted as an $\mathbb{N}_8$ integer, is smaller than 128.

EXAMPLE:

If $a == 63$ then $T_{02}(63) = 126;\ T_{03}(63) = T_{02}(63) \oplus 63 = 65$

If $a == 143$ then $T_{02}(143) = 5;\ T_{03}(143) = T_{02}(143) \oplus 143 = 138$.

11.6 The round key addition

Rijndael-256 employs 15 *Round Keys*, each of which can be written as 4x8 rectangular matrices, with each element containing a byte. The complete set of *Round Keys* is here denoted as $\{ rk_{r,i,j} \}$, where *r* denotes the current round. Thus, for a fixed value of *r*, this represents a 4x8 array analogous to the representation of the *State* as an array $\{ a_{i,j} \}$. The *Round Key* is derived from the PRF key **K** by means of the key schedule described in clause 11.7. *Round Key* addition involves a simple bitwise exclusive-or operation applied to the current *Round Key* and the current *State*. Hence, for every element of the current *State*, in the r^{th} round one has:

$$b_{i,j} = a_{i,j} \oplus rk_{r,i,j}$$

where:

$a_{i,j}$ is the initial value, namely an element in the current *State* of the r^{th} round.

$b_{i,j}$ is the output value, namely an element in the new *State* of the r^{th} round, and

$rk_{r,i,j}$ is a (byte-sized) element of the *Round Key*.

11.7 Key schedule: 256-bit keys

Rijndael-256 has 15 *Round Keys*, numbered $r \in [0 \ldots 14]$, each of which, by the above, can be expressed as 4x8 rectangular arrays of bytes. *Round Keys* are derived from the PRF key **K** by means of the key schedule. The initial *Round Key* (labelled as the zeroth *Round Key*) is formed directly from the PRF key **K** and is used unaltered for the initial key addition occurring at the begiNning of the first round. The remaining *Round Keys* are used at the end of each of the fourteen rounds, with each new *Round Key* being derived from the previous *Round Key*.

NOTE 1: The key schedule can be executed round by round on an "as needed" basis, so a total of only 32 bytes is required to store the *Round Key*.

Let $rk_{r,i,j}$ be the value of the r^{th} *Round Key* at position (*i*, *j*) in the array and let $\{ k_{i,j} \}$ be the PRF key bytes loaded into a 4x8 array as described above.

Initialisation of the first *Round Key* ($r = 0$): $rk_{0,i,j} = k_{i,j}$ for $i \in [0 \ldots 3]$, and $j \in [0 \ldots 7]$. The other *Round Keys* ($r = 1$ to 14 inclusive) are column-wise calculated in one of two ways:

1) column $j = 0$ is computed from the first and last columns of the previous *Round Key*, and,

2) any other column $j \neq 0$ is computed from the same column *j* of the previous *Round key* and the preceding column, $j - 1$, of the current *Round key*, in a way dependent on *j*.

In detail, the *Round Keys* are constructed as follows:

First the 0^{th} column is constructed from the first and last columns of the previous *Round Key* as:

$rk_{r,0,0} = rk_{r-1,0,0} \oplus$ S-box$[rk_{r-1,7,7}] \oplus$ round_const$[r]$

$rk_{r,1,0} = rk_{r-1,1,0} \oplus$ S-box$[rk_{r-1,2,7}]$

$rk_{r,2,0} = rk_{r-1,2,0} \oplus$ S-box$[rk_{r-1,3,7}]$

$rk_{r,3,0} = rk_{r-1,3,0} \oplus$ S-box$[rk_{r-1.0.7}]$

where round_const[1] = 1 and round_const[r] = T_{02}(round_const[r-1]) where T_{02} is the linear transformation defined in clause 11.5, and S-box is the previously mentioned byte substitution defined in clause 11.8.

NOTE 2: Observe that the row indices used as index into the S-box appear in shifted order.

Then next three columns are constructed in turn from the corresponding column of the previous *Round Key* and the immediately previous column of the current *Round Key*:

$rk_{r,i,j} = rk_{r-1,i,j} \oplus rk_{r,i,j-1}$ for $i \in [0 \ldots 3]$ and $j \in [1, 2, 3]$.

The column $j = 4$ is then constructed using the $j = 4$ column of the preceding *Round Key* and the preceding column ($j - 1 = 3$) of the current *Round Key*, by computing and assigning the following values for j = 4:

$rk_{r,0,j} = rk_{r-1,0,j} \oplus$ S-box$[rk_{r,0,j-1}]$

$rk_{r,1,j} = rk_{r-1,1,j} \oplus$ S-box$[rk_{r,1,j-1}]$

$rk_{r,2,j} = rk_{r-1,2,j} \oplus$ S-box$[rk_{r,2,j-1}]$

$rk_{r,3,j} = rk_{r-1,3,j} \oplus$ S-box$[rk_{r,3,j-1}]$.

NOTE 3: Observe that the row indices used as index into the S-box here do not appear in shifted order.

The final three columns are then generated in a way completely analogous to the columns 1, 2, and 3 as:

$rk_{r,i,j} = rk_{r-1,i,j} \oplus rk_{r,i,j-1}$ for $i \in [0 \ldots 3]$ and $j \in [5, 6, 7]$.

NOTE 4: The fourteen round constants required for 256-bit keys are computed from the equations:

round_const[1] := 1,

round_const[r] := T_{02}(round_const[$r - 1$]), $r \in [2 \ldots 14]$

or equivalently as x^{r-1} mod $(x^8 + x^4 + x^3 + x + 1)$. Given as values in $\mathbb{N}_8$, these values are: 1, 2, 4, 8, 16, 32, 64, 128, 27, 54, 108, 216, 171, 77.

11.8 The Rijndael-256 S-box give ans values in $\mathbb{N}_8$

```
S-box[256] = {
          99, 124,  119, 123,  242,  107,  111, 197,   48,   1,   103,  43,  254,  215, 171,  118,
         202, 130,  201, 125,  250,   89,   71, 240,  173, 212,  162, 175,  156,  164, 114,  192,
         183, 253,  147,  38,   54,   63,  247, 204,   52, 165,  229, 241,  113,  216,  49,   21,
           4, 199,   35, 195,   24,  150,    5, 154,    7,  18,  128, 226,  235,   39, 178,  117,
           9, 131,   44,  26,   27,  110,   90, 160,   82,  59,  214, 179,   41,  227,  47,  132,
          83, 209,    0, 237,   32,  252,  177,  91,  106, 203,  190,  57,   74,   76,  88,  207,
         208, 239,  170, 251,   67,   77,   51, 133,   69, 249,    2, 127,   80,   60, 159,  168,
          81, 163,   64, 143,  146,  157,   56, 245,  188, 182,  218,  33,   16,  255, 243,  210,
         205,  12,   19, 236,   95,  151,   68,  23,  196, 167,  126,  61,  100,   93,  25,  115,
          96, 129,   79, 220,   34,   42,  144, 136,   70, 238,  184,  20,  222,   94,  11,  219,
         224,  50,   58,  10,   73,    6,   36,  92,  194, 211,  172,  98,  145,  149, 228,  121,
         231, 200,   55, 109,  141,  213,   78, 169,  108,  86,  244, 234,  101,  122, 174,   8,
         186, 120,   37,  46,   28,  166,  180, 198,  232, 221,  116,  31,   75,  189, 139,  138,
         112,  62,  181, 102,   72,    3,  246,  14,   97,  53,   87,  185,  134,  193,  29,  158,
         225, 248,  152,  17,  105,  217,  142, 148,  155,  30,  135,  233,  206,   85,  40,  223,
         140, 161,  137,  13,  191,  230,   66, 104,   65, 153,   45,  15,  176,   84,  187,  22}
```

11.9 Other key sizes

The Rijndael-based PRF supports also 128- and 192-bit keys. For MILENAGE-256 usage, only 128 and 256 bits are supported. Usage of 128-bit keys shall be handled by extending the 128-bit (16 bytes) PRF key, **K,** by the 16 zero-bytes, i.e. $\mathbf{K}[i]$ = 0x00 for $i \in [16 \ldots 31]$, and then treating the result as a 256-bit (32 bytes) key.

Annex A (informative): Change history

Change history							
Date	**Meeting**	**TDoc**	**CR**	**Rev**	**Cat**	**Subject/Comment**	**New version**
2024-02	SA3#115	S3-240404				TS skeleton	0.0.0
2024-02	SA3#115	S3-240818				TS skeleton using 3GPP template	0.0.1
2024-02	SA3#115	S3-240408				Addition of introduction	0.1.0
2024-08	SA3#117	S3-243423				Addition of the text based on the selection of Rijndael-based Milenage-256 to specify Milenage-256 algorithm.	0.2.0
2024-11	SA3#119	S3-245103				TR 35.937 replaces TS 35.237, and there is editorial clean up for presentation to TSG-SA.	0.3.0
2024-12	SA#106	SP-241788				Presented for information and approval	1.0.0
2025-01						Upgrade to change control version	19.0.0

History

Document history		
V19.0.0	January 2026	Publication