INTERNATIONAL STANDARD

ISO/IEC 29167-19

First edition
2016-05-15

Information technology — Automatic identification and data capture techniques —

Part 19: Crypto suite RAMON security services for air interface communications

Technologie informative — Identification automatique et technique capturé data —

Partie 19: Air interface pour les services de sécurité suite de crypto RAMON

Reference number
ISO/IEC 29167-19:2016(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT), see the following URL: Foreword — Supplementary information.

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, Subcommittee SC 31, *Automatic identification and data capture*.

ISO/IEC 29167 consists of the following parts, under the general title *Information technology — Automatic identification and data capture techniques*:

— *Part 1: Security services for RFID air interfaces*

— *Part 10: Crypto suite AES-128 security services for air interface communications*

— *Part 11: Crypto suite PRESENT-80 security services for air interface communications*

— *Part 12: Crypto suite ECC-DH security services for air interface communications*

— *Part 13: Crypto suite Grain-128A security services for air interface communications*

— *Part 14: Crypto suite AES OFB security services for air interface communications*

— *Part 16: Crypto suite ECDSA-ECDH security services for air interface communications*

— *Part 17: Crypto suite cryptoGPS security services for air interface communications*

— *Part 19: Crypto suite RAMON security services for air interface communications*

— *Part 20: Crypto suite Algebraic Eraser security services for air interface communications*

The following part is under preparation:

— *Part 15: Crypto suite XOR security services for air interface communications*

Introduction

This part of ISO/IEC 29167 specifies the security services of a Rabin-Montgomery (RAMON) crypto suite. It is important to know that all security services are optional. The crypto suite provides Tag authentication security service.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this International Standard may involve the use of patents concerning radio-frequency identification technology given in the clauses identified below.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have ensured the ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world. In this respect, the statements of the holders of these patent rights are registered with ISO and IEC.

Information on the declared patents may be obtained from:

NXP B.V.

411 East Plumeria, San Jose,
CA 95134-1924 USA

The latest information on IP that may be applicable to this part of ISO/IEC 29167 can be found at www.iso.org/patents.

Information technology — Automatic identification and data capture techniques —

Part 19: Crypto suite RAMON security services for air interface communications

1 Scope

This part of ISO/IEC 29167 defines the Rabin-Montgomery (RAMON) crypto suite for the ISO/IEC 18000 air interfaces standards for radio frequency identification (RFID) devices. Its purpose is to provide a common crypto suite for security for RFID devices that may be referred by ISO committees for air interface standards and application standards.

This part of ISO/IEC 29167 specifies a crypto suite for Rabin-Montgomery (RAMON) for air interface for RFID systems. The crypto suite is defined in alignment with existing air interfaces.

This part of ISO/IEC 29167 defines various authentication methods and methods of use for the cipher. A Tag and an Interrogator may support one, a subset, or all of the specified options, clearly stating what is supported.

2 Conformance

2.1 Claiming conformance

To claim conformance with this part of ISO/IEC 29167, an Interrogator or Tag shall comply with all relevant clauses of this part of ISO/IEC 29167, except those marked as “optional”.

2.2 Interrogator conformance and obligations

To conform to this part of ISO/IEC 29167, an Interrogator shall implement the mandatory commands defined in this part of ISO/IEC 29167, and conform to the relevant part of ISO/IEC 18000.

To conform to this part of ISO/IEC 29167, an Interrogator may implement any subset of the optional commands defined in this part of ISO/IEC 29167.

To conform to this part of ISO/IEC 29167, the Interrogator shall not

— implement any command that conflicts with this part of ISO/IEC 29167, or

— require the use of an optional, proprietary, or custom command to meet the requirements of this part of ISO/IEC 29167.

2.3 Tag conformance and obligations

To conform to this part of ISO/IEC 29167, a Tag shall implement the mandatory commands defined in this part of ISO/IEC 29167 for the supported types, and conform to the relevant part of ISO/IEC 18000.

To conform to this part of ISO/IEC 29167, a Tag may implement any subset of the optional commands defined in this part of ISO/IEC 29167.

To conform to this part of ISO/IEC 29167, a Tag shall not

— implement any command that conflicts with this part of ISO/IEC 29167, or

— require the use of an optional, proprietary, or custom command to meet the requirements of this part of ISO/IEC 29167.

3 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 18000-3, *Information technology — Radio frequency identification for item management — Part 3: Parameters for air interface communications at 13,56 MHz*

ISO/IEC 18000-4, *Information technology — Radio frequency identification for item management — Part 4: Parameters for air interface communications at 2,45 GHz*

ISO/IEC 18000-63, *Information technology — Radio frequency identification for item management — Part 63: Parameters for air interface communications at 860 MHz to 960 MHz Type C*

ISO/IEC 19762 (all parts), *Information technology — Automatic identification and data capture (AIDC) techniques — Harmonized vocabulary*

ISO/IEC 29167-1, *Information technology — Automatic identification and data capture techniques — Part 1: Security services for RFID air interfaces*

4 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 19762 (all parts) and the following apply.

4.1
authentication
service that is used to establish the origin of information

4.2
confidentiality
property whereby information is not disclosed to unauthorized parties

4.3
integrity
property whereby data has not been altered in an unauthorized manner since it was created, transmitted or stored

4.4
non-traceability
protection ensuring that an unauthorized interrogator is not able to track the Tag location by using the information sent in the Tag response

4.5
secure communication
communication between the tag and the interrogator by use of the *Authenticate* command, assuring authenticity, integrity and confidentiality of exchanged messages

5 Symbols and abbreviated terms

5.1 Symbols

xx_2	binary notation
xxh	hexadecimal notation
||	concatenation of syntax elements in the order written

5.2 Abbreviated terms

AES	Advanced Encryption Standard
CBC	Cipher Block Chaining
CH	Challenge
CH_{I1}, CH_{I2}	Interrogator random challenge, 16 bytes
CH_T	Tag random challenge, 16 bytes
CG	Cryptogram
CMAC	Ciphered Message Authentication Code
CRC	Cyclic Redundancy Check
CRC–16	16-bit CRC
CS	Crypto Suite
CSI	Crypto Suite Identifier
DEC(key, data)	AES decryption of enciphered “data” with secret “key”
ENC(key, data)	AES encryption of plain “data” with secret “key”
EPC™	Electronic Product Code
IID	Interrogator Identifier, 8 bytes
IV	Initialization Vector for CBC-encryption, 16 bytes
KDF	Key Derivation Function
K_E	Public key for encryption stored on Tag
K_D	Private decryption key stored on Interrogator
K_V	Public signature verification key stored on Interrogator
K_S	Private signature generation key stored in the tag issuer facility
K_{ENC}	Shared secret message encryption key
K_{MAC}	Shared secret message authentication key
KESel	Key select (determines which KE will be used)

KSel	Key select (determines which pair of KENC, KMAC will be used)
MAC(key, data)	Calculation of a MAC of (enciphered) “data”’ with secret “key”; internal state of the tag’s state machine
MAMx,y	Mutual Authentication Method x.y
MCV	MAC Chaining Value
MIX(CH, RN, SID)	RAMON mix function
PRF	Pseudorandom Function
R	Tag response
RAMON	Rabin-Montgomery
RFU	Reserved for Future Use
RM_ENC(key, data)	RAMON encryption of plain “data” with public “key”
RM_DEC(key, data)	RAMON decryption of enciphered “data” with private “key”
RN	Random Number
RNT	Tag Random Number, 16 bytes
S_{ENC}	Message encryption session key
S_{MAC}	Message authentication session key
SID	Secret IDentifier, 8 bytes, identifying the tag
SSC	Send Sequence Counter for replay protection, 16 bytes
TAMx,y	Tag Authentication Method x.y; internal state of the tag’s state machine
TLV	Tag Length Value
UHF	Ultra High Frequency
UII	Unique Item Identifier
WORM	Write once, read many

5.3 Notation

This crypto suite uses the notation of ISO/IEC 18000-63.

The following notation for key derivation corresponds to Reference [7] and Clause 5.

PRF(*s*,*x*)	A pseudo-random function with seed *s* and input data *x*.
K_I	Key derivation key used as input to the KDF to derive keying material. K_I is used as the block cipher key, and the other input data are used as the message defined in Reference [5].
K_O	Keying material output from a key derivation function, a binary string of the required length, which is derived using a key derivation key.
Label	A string that identifies the purpose for the derived keying material, which is encoded as a binary string.

Context	A binary string containing the information related to the derived keying material. It may include identities of parties who are deriving and/or using the derived keying material and, optionally, a nonce known by the parties who derive the keys.
L	An integer specifying the length (in bits) of the derived keying material K_0. *L* is represented as a binary string when it is an input to a key derivation function. The length of the binary string is specified by the encoding method for the input data.
h	An integer that indicates the length (in bits) of the output of the PRF.
i	A counter that is input to each iteration of the PRF.
r	An integer, smaller or equal to 32, that indicates the length of the binary representation of the counter *i*. in bits.
00h	An all zero octet. An optional data field used to indicate a separation of different variable length data fields.
$\lceil X \rceil$	The smallest integer that is larger than or equal to *X*. The ceiling of *X*.
$\{X\}$	Indicates that data *X* is an optional input to the key derivation function.
$[T]_2$	An integer *T* represented as a binary string (denoted by the "2") with a length specified by the function, an algorithm, or a protocol which uses *T* as an input.
$\emptyset$	The empty binary string.

6 Crypto suite introduction

6.1 Overview

The RAMON Crypto Suite permits two levels of implementation. The first level provides secure identification and tag authentication, while the second level extends the functionality by mutual authentication to securely communicate between Interrogator and Tag, e.g. for secure reading and writing non-volatile memory.

Basic RAMON Tags may provide only the first level of implementation, while more sophisticated Tags also provide the second level. See Figure 1 for the different implementation levels for the RAMON crypto suite.

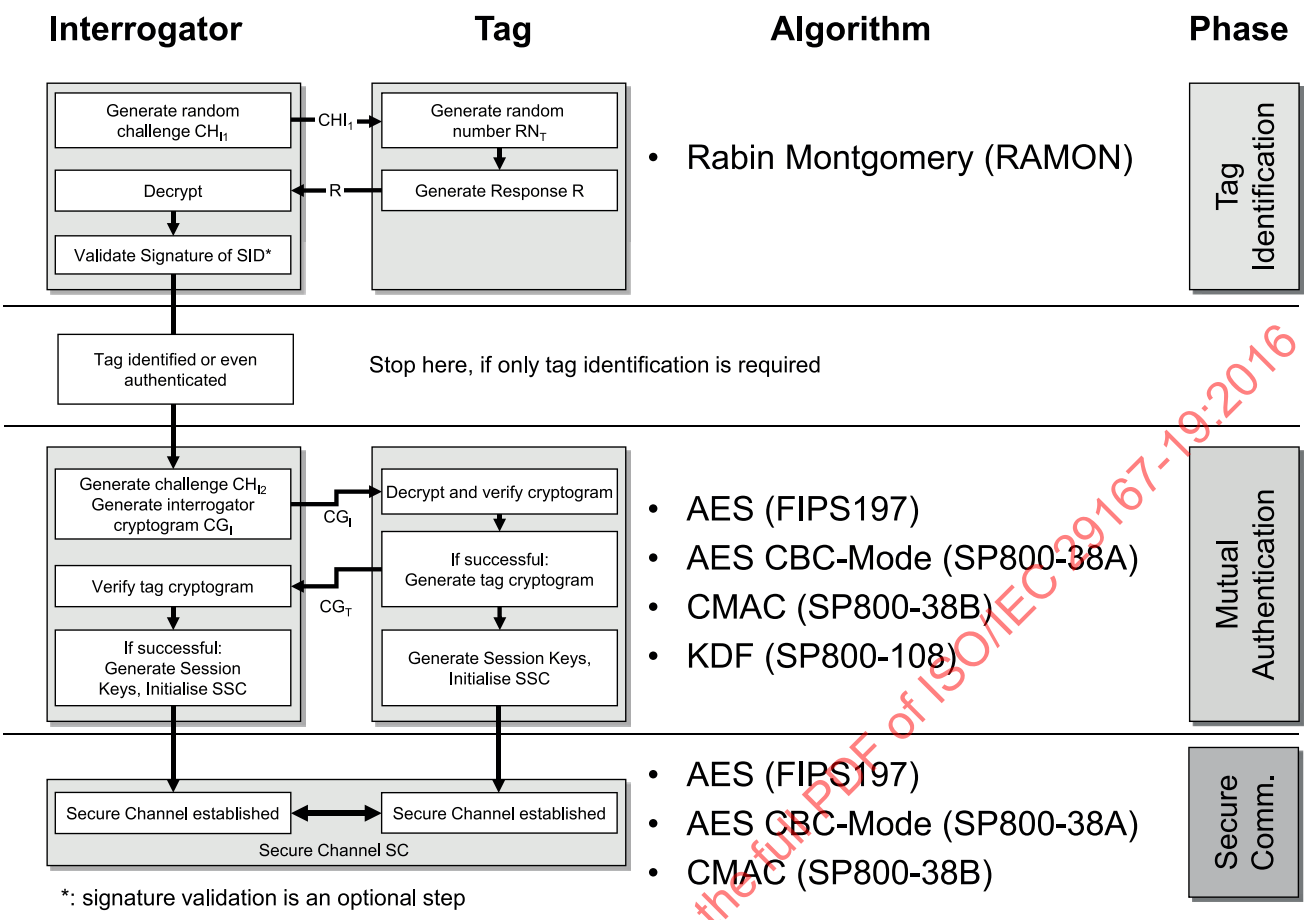


Figure 1 — Overview of the different implementation levels for the RAMON crypto suite

6.2 Authentication protocols

6.2.1 Tag Identification

The Rabin-Montgomery crypto suite provides non-traceable and confidential Tag identification. Confidentiality and privacy for the Tag's identifier are provided without requiring the Tag to store a private key.

The crypto suite is based on the asymmetric cryptosystem developed by Michael O. Rabin[3]. The original algorithm is augmented by a method detected by Peter Montgomery[2], which avoids the division of long numbers in modular arithmetic. Combining Rabin encryption with the concept of Montgomery multiplication advantage is taken of the fact that no "costly" division is required.

The Tag performs only public key operations. The Interrogator performs the "expensive" private key operation. The steps necessary to carry out RAMON are outlined in Table 1. RAMON encryption performed by the Tag and decryption performed by the Interrogator are specified in C.3 and C.4. The cryptographic keys are specified in 6.6.

This specification also includes in C.1 the structure of the clear text record used for authentication of the Tag, comprising the Tag identity record and random data originating in part from the Tag and from the Interrogator for the other part.

Table 1 — Protocol steps for Tag identification

Interrogator (K_D, K_V)		Tag (SID, K_E)
Generate random challenge CH_{I1} and send it to the Tag.	(CH_{I1}) →	
		Generate random number RN_T.
		Generate response cryptogram: R = RM_ENC(K_E, MIX(CH_{I1}, RN_T, TLV record)).
	(R) ←	
Decrypt Tag response and apply the inverse of the MIX function to get the plaintext P: P = MIX^{-1}(RM_DEC (K_D, R)).		
Obtain CH_{I1}, RN_T and SID from plaintext P.		
Compare previously generated Interrogator challenge with the value received from Tag. If successful, Tag is identified.		
If a signature is provided along with the SID, use K_V to validate the signature. If successful, Tag is authenticated.		

6.2.2 Symmetric mutual authentication

This crypto suite allows combining the Rabin-Montgomery scheme for Tag identification with symmetric mutual authentication. The mutual authentication specified by this crypto suite is based on AES, according to Reference [8]. The CBC mode for encryption is specified in Reference [4]. For MAC generation CMAC according to Reference [5] is used. For derivation of secure messaging keys, the KDF in counter mode specified in 5.1 of Reference [7] is used.

The protocol steps for mutual authentication are outlined in Table 2.

Table 2 — Protocol steps for mutual authentication

Phase	Interrogator (IID, Database, K_D, K_V)		Tag (SID, K_E, K_{ENC}, K_{MAC})
(1) Tag Identification	Generate random challenge CH_{I1} and send it to the Tag.	(CH_{I1}) →	Generate random number RN_T. Generate response:
	Decrypt Tag response and apply the	(R) ←	R = RM_ENC(K_E, MIX(CH_{I1}, RN$_T$, TLV record, '00' byte)).
	inverse of the MIX function to get the plaintext P: P = MIX^{-1}[RM_DEC (K_D, R)].		
	Obtain CH_{I1}, RN_T and SID from plaintext P.		
	Compare previously generated Interrogator challenge with the value received from Tag. If successful, Tag is identified.		
	If a signature is provided along with the SID, use K_V to validate the signature. If successful, Tag is authenticated.		
	Set CH_T = RN_T.		
The Interrogator has successfully identified (and authenticated) the Tag. In the following phase, CH_T and SID are used in the mutual authentication.			

Table 2 *(continued)*

Phase	Interrogator (*IID*, Database, K_D, K_V)		Tag (*SID*, K_E, K_{ENC}, K_{MAC})
(2) Mutual Authentication	Generate CH_{I2}. Generate cryptogram: $S = CH_{I2} \| \| IID \| \| CH_T \| \| SID$; C = ENC (K_{ENC}, S); M = MAC (K_{MAC}, C); $CG_I = C \| \| M$.	(CG_I)	Decrypt and verify the cryptogram:
		→	MAC (K_{MAC}, C); DEC (K_{ENC}, C). Verify CH_T and *SID*. If equal, generate Session Keys S_{ENC}, S_{MAC}. Initialize SSC. Generate Tag cryptogram: $S = CH_T \| \| SID \| \| CH_{I2} \| \| IID$; C = ENC (K_{ENC}, S); M = MAC (K_{MAC}, C); $CG_T = C \| \| M$
	Verify the cryptogram: MAC (K_{MAC}, C); DEC (K_{ENC}, C). Verify CH_{I2}, CH_T, *SID* and *IID*. If equal, generate session keys: S_{ENC}, S_{MAC}. Initialize SSC.	(CG_T) ←	
Mutual authentication is now complete and a secure channel is established.			

The Interrogator has access to a list of SIDs (Secret identifiers) with the associated K_{ENC} and K_{MAC} for each Tag. This is represented by the "Database" on Interrogator's site.

After having successfully identified the Tag in Phase 1, the Interrogator is able to find secret keys K_{ENC} and K_{MAC} that it shares with the Tag. K_{ENC} is used in CBC mode. The IV for encryption is set to all zeroes 00h...00h. As the size of S is on both sides a multiple of the AES block size, no padding is applied. K_{MAC} is used to calculate a 16-byte MAC.

CH_T and CH_{I2} are used as challenges in the challenge-response protocol for mutual authentication and for generation of the starting value of the SSC. See 6.3 for details.

The session encryption key, S_{ENC}, is used for confidentiality of data in transit. AES encryption, including an SSC, is illustrated in Figure 18; decryption is illustrated in Figure 19. The session MAC key, S_{MAC}, is used for data and protocol integrity. This crypto suite derives session keys as specified in 6.4.

If the Tag cannot verify the interrogator's MAC, it reports a Crypto Suite error (see Annex B for information) and assumes state **Init**. If the interrogator cannot verify the tag's MAC, the tag is not authenticated.

6.3 Send Sequence Counter

The send sequence counter (SSC) ensures that the Initial Values (IVs) are different for every encryption and the MAC chaining values (MCVs) are different for every MAC generation. To this end, the SSC is incremented (+1) each time before a Secure Communication command or response is processed.

After mutual authentication, the initial value of the send sequence counter SSC is generated as follows:

$SSC = CH_T$ (<algorithm block size/2> least significant bytes) ||

CH_{I2} (<algorithm block size/2> least significant bytes)

After receiving a secure command, the Tag increments SSC, then checks the MAC and then decrypts the command. In turn, before sending a secure response the Tag increments SSC, encrypts the response and generates the MAC. Each particular step is under control of the security flags. Thus, if SSC has the value x at idle time, x+1 is used for processing the next secure command, and x+2 is used for processing the response. SSC may overflow to 0h during the increment without particular action.

6.4 Session key derivation

The derivation of the session keys, S_{ENC} and S_{MAC}, is based on the KDF in counter mode specified in 5.1 of Reference [7]. This method uses CMAC as the PRF with AES as underlying block cipher with full 16 bytes output length. The input to the PRF for this cipher suite is as specified in 6.4.2.

6.4.1 KDF in counter mode

The key derivation function iterates a pseudorandom function n times and concatenates the output until L bits of keying material are generated, where $n := \lceil L / h \rceil$. In each iteration, the fixed input data is the string *Label* || *00h* || *Context* || $[L]_2$. The counter $[i]_2$ is the iteration variable and is represented as a binary string of r bits.

Figure 2 illustrates the process.

The input to the PRF [see step d) of **Process**] is explained in 6.4.2.

For the derivation of session encryption key S_{ENC}, K_I is set to K_{ENC}. For the derivation of session MAC key S_{MAC}, K_I is set to K_{MAC}.

Fixed values

— h – The length of the output of the PRF in bits;

— r – The length of the binary representation of the counter i. in bits.

Input: K_I, *Label*, *Context*, and L.

Process

a) $n := \lceil L / h \rceil$.

b) If $n > 2^r$-1, then indicate a crypto suite error and stop.

c) $result(0) := \emptyset$.

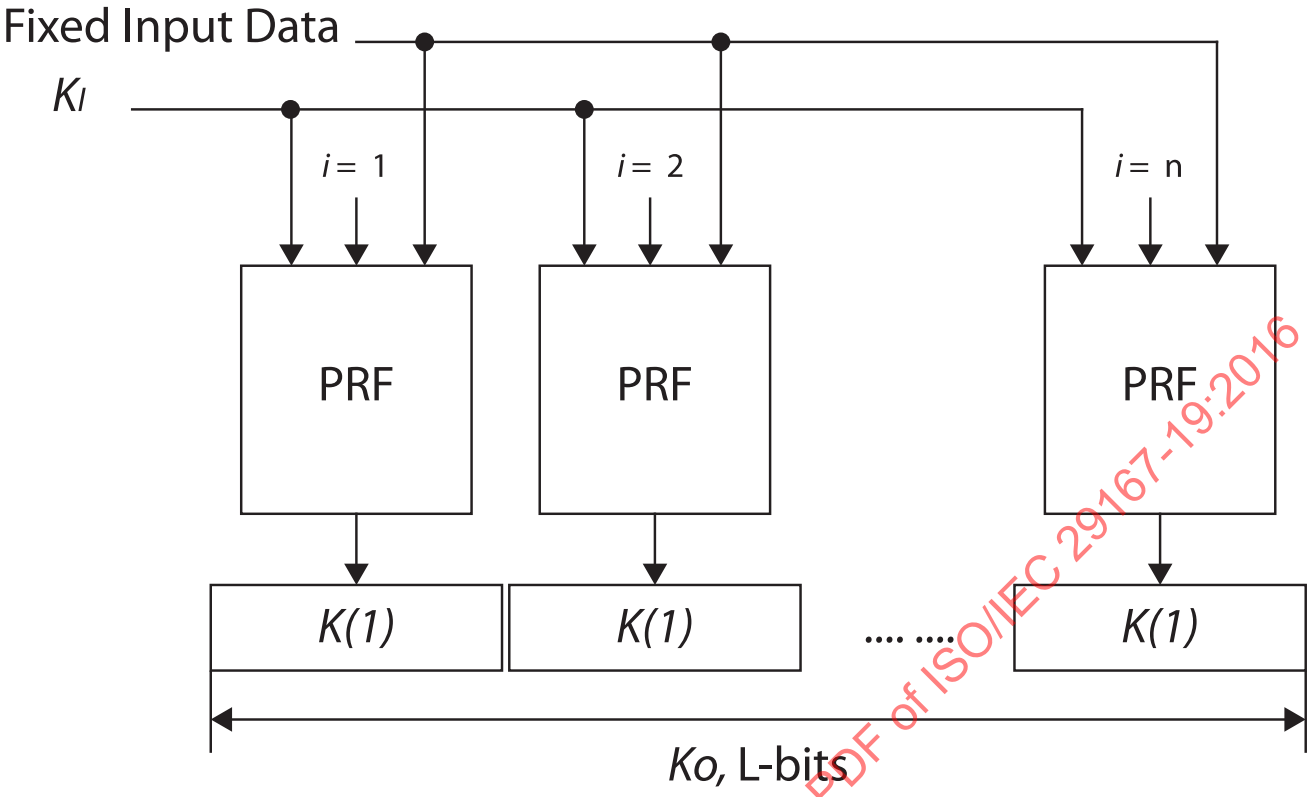
d) For $i = 1$ to n, do

— $K(i) := \text{PRF}\ (K_I, [i]_2 \,||\, Label \,||\, 00h \,||\, Context \,||\, [L]_2)$;

— $result(i) := result(i\text{-}1) \,||\, K(i)$.

e) Return: $K_O :=$ the leftmost L bits of $result\ (n)$.

Output: K_O.



NOTE SOURCE: Reference [7].

Figure 2 — KDF in Counter Mode

6.4.2 Key Derivation Scheme

The following derivation data are used to generate session keys according to the KDF in counter mode as specified in Reference [7].

The one byte counter, *i*, may take the values 01h or 02h. The value 01h is used when *L* takes the value 0080h for derivation of AES-128 keys, which is currently the only case relevant for this part of ISO/IEC 29167 (see NOTE below).

The iteration variable, *i*, is concatenated with the fixed input data. The fixed input data is a concatenation of a *Label*, a separation indicator 00h, the *Context*, and $[L]_2$ as follows:

— *Label*: consists of 11 zeroes (00h) bytes followed by a one byte derivation constant as defined in Table 3;

— One byte separation indicator 00h;

— *Context*: $CH_{I2} \,||\, CH_T$;

— $[L]_2$: the length in bits of the derived data. For derivation of AES-128 keys which is currently the only case relevant for this part of ISO/IEC 29167, *L* takes the value 0080h (see NOTE below).

In each iteration, the fixed input data is the string *Label* || *0x00* || *Context* || $[L]_2$.

NOTE Currently, this part of ISO/IEC 29167 only supports AES-128 keys. In the future, this part of ISO/IEC 29167 may be changed to additional key length.

Table 3 — Encoding of the one byte derivation constant

b8	b7	b6	b5	b4	b3	b2	b1	Description
0	0	0	0	0	0	1	x	Key derivation
0	0	0	0	0	0	x	0	Derivation of session encryption key S_{ENC} with K_I set to K_{ENC}
0	0	0	0	0	0	x	1	Derivation of session MAC key S_{MAC} with K_I set to K_{MAC}
NOTE Any other value is RFU.								

A Tag or an Interrogator using a derivation constant marked as RFU is not compliant to this part of ISO/IEC 29167.

6.5 IID, SID, Used Keys and Their Personalisation

This crypto suite assumes the following keys and information to be available on the Tag:

— the SID, optionally signed with the signature key K_S before it was stored on the Tag;

— the RAMON encryption key K_E;

and, if mutual authentication is provided:

— the shared secret keys K_{ENC} and K_{MAC}.

On the Interrogator:

— the RAMON decryption key K_D;

— optionally, the signature verification key K_V;

— a list of valid SIDs; each SID might have a signature attached to it;

and, if mutual authentication is provided:

— the shared secret keys K_{ENC} and K_{MAC}.

The IID is an 8-byte value, which identifies the interrogator to the Tag. The IID can be chosen freely, but must remain constant during a session.

The SID is a unique 8-byte value, which identifies the Tag to the interrogator. It is set during personalization and remains constant throughout the lifetime of the Tag.

The SID used by this crypto suite is used by the application to securely identify the tag and therefore has nothing in common with any unique identifier defined by an air interface standard. The SID and the optional signature are secret information and should never be readable for an unauthorized reader. The SID used by this crypto suite shall never be sent in plaintext. The SID can be signed to preserve integrity and to provide authenticity. The party that generates the signature, possesses the key pair consisting of the private and the public key (K_S, K_V) for signature generation and verification. It may forward the public key K_V to another party to enable it to verify the signature. F.3 contains the specification of the signature over the SID. Annex F shows the usage of the SID in combination with the non-tractability feature of this crypto suite.

The Tag does not perform signature generation or verification, nor does it store the corresponding keys. It only stores the SID along with its signature (which is optional) and the public key K_E for Tag authentication. If mutual authentication is supported, the Tag also stores the shared secret keys K_{ENC} and K_{MAC}.

The memory locations storing the SID and the secret keys K_E, K_{ENC} and K_{MAC} shall not be readable for any Interrogator after having written these values once during production of the Tag. For that purpose a Tag may have a memory area used for storing the SID and the key K_E configured as WORM or as a fuse. However, production is out of scope of this part of ISO/IEC 29167; this functionality has to be implemented in a proprietary way.

The Interrogator application performing RAMON, shall have access to the private decryption key K_D in order to be able to decrypt the authentication message sent by the Tag. In order to be able to perform mutual authentication, the Interrogator shall have access to the keys K_{ENC} and K_{MAC} associated with a specific SID.

EXAMPLE Figure 3 shows an example system flow with the involved components and keys. System Integrator and Tag Issuer can work in parallel. The steps performed by these parties can be executed independently of the other party. Generation of the signature key pair and signing of SIDs is an optional security function provided by the operational environment.

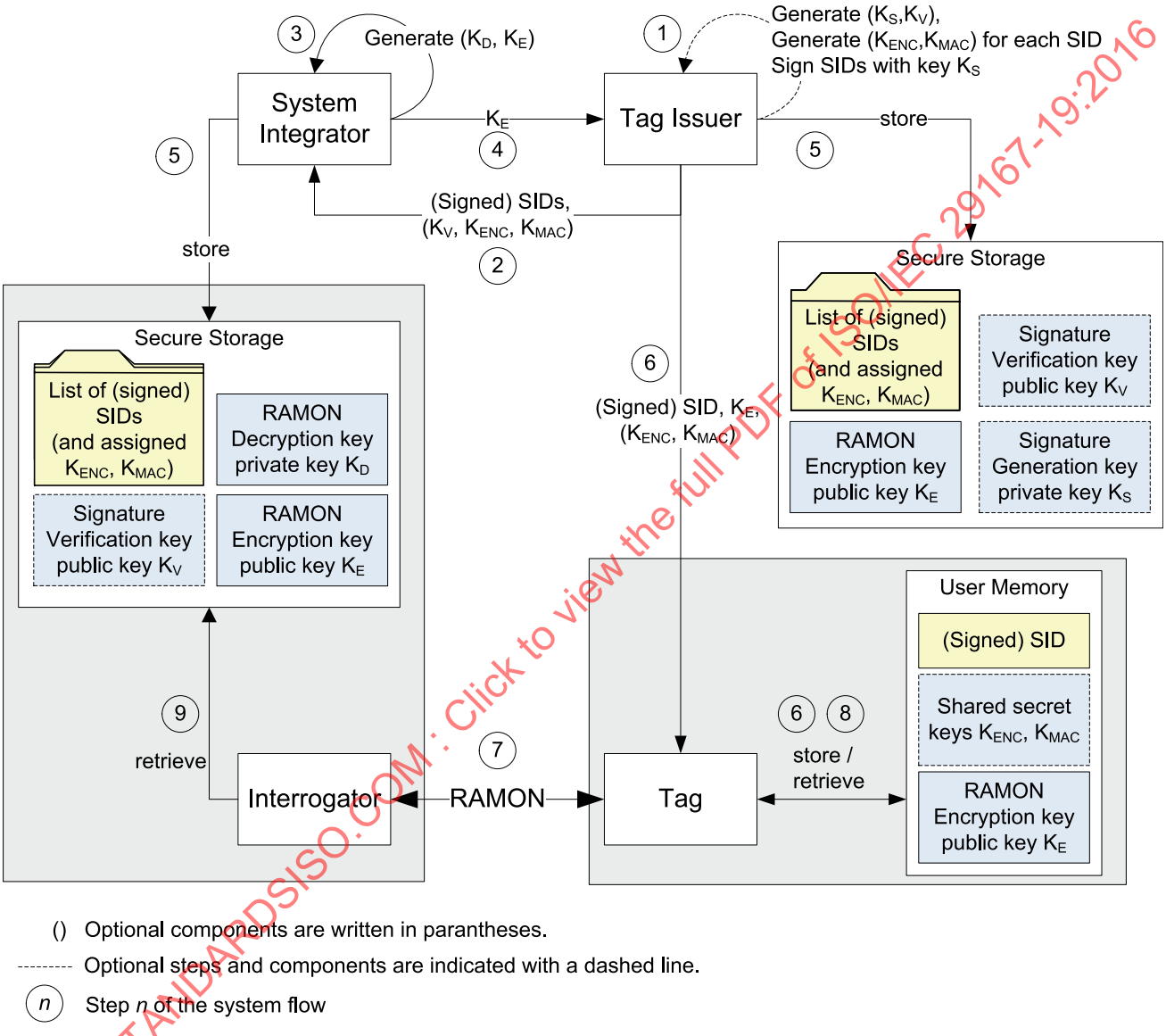


Figure 3 — System flow of an RFID System using the RAMON crypto suite

6.6 Key table

The keys used by this crypto suite are listed in Table 4 and Table 5.

Table 4 — Keys of this cipher suite used for Tag identification

Key	Usage	Length in bits
K_E	Public key for encryption stored on Tag	1024
K_D	Private decryption key stored on Interrogator	1024
K_V	Public signature verification key stored on Interrogator. This is an ECDSA key (see F.3).	160

Table 5 — Keys of this cipher suite used for mutual authentication and secure communication

Key	Usage	Length in bits
K_{ENC}	Shared secret encryption key	128
K_{MAC}	Shared secret message authentication key	128
S_{ENC}	Session encryption key	128
S_{MAC}	Session message authentication key	128

K_{ENC} and K_{MAC} shall be different and shall be available to both the Interrogator and the Tag. The establishment/derivation of these keys is not in the scope of this specification.

Session keys shall be destroyed immediately when they are no longer used.

For the Rabin-Montgomery scheme, the public key, K_E, is an integer which indicates the modulus for the long number arithmetic used for the encryption. The private key, K_D, comprises two prime numbers, p and q, with $p \equiv q \equiv 3 (\mathrm{mod}\, 4)$, where the following relation holds:

$$K_E = p \cdot q$$

The security of the Rabin-Montgomery scheme is given by the fact that a factorization of K_E is computationally hard.

All keys must be stored by the Tag and the Interrogator, such that unauthorized access and modification is prohibited.

The performance of the RAMON encryption can be improved by a factor of about 3/2, if prime numbers p and q are chosen that satisfy the following (optional) additional condition:

$$K_E = p \cdot q = 1 \,(\mathrm{mod} 2^{k/2})$$

where k is the bit length of K_E.

The security of the Rabin-Montgomery scheme is given by the fact that a factorization of K_E is computationally hard.

All keys must be stored by the Tag and the Interrogator, such that unauthorized access and modification is prohibited.

7 Parameter definitions

The parameter definitions of the crypto suite are specified in Table 6.

Table 6 — Definition of Parameters

Parameter	Description
command code	This is a protocol-specific code which indicates that this command belongs to a cipher suite.
KESel [7:0]	Key select, determines which key will be used for RAMON encryption.
KSel [7:0]	Key select, determines which pair of K_{ENC}, K_{MAC} will be used for mutual authentication.
CH_{I1} [127:0] CH_{I2} [127:0]	Interrogator random challenge, 16 bytes.
CH_T [127:0]	Tag random challenge, 16 bytes.
RN_T [127:0]	Tag random number, 16 bytes
SID	Unique identifier of the Tag, 8 bytes. See Table C.2, for additional information.
IID [63:0]	Interrogator identifier, 8 bytes.
IV [127:0]	Initialization vector for CBC-encryption, 16 bytes.
SSC [127:0]	Sequence counter for replay protection, 16 bytes.

8 State diagrams

8.1 General

This crypto suite allows carrying out Tag identification without mutual authentication and secure communication. Mutual authentication may be performed after successful Tag identification and secure communication may be used after successful mutual authentication. Mutual authentication corresponds to AuthMethod 1 and Tag authentication with challenge response corresponds to AuthMethod 3, both defined in 10.3.

A Tag may use one of two authentication protocol modes, the *partial result mode* or the *complete result mode*. Both Tag Identification and Mutual Authentication generate partial results while calculating the cryptogram. A Tag may provide these partial results to the interrogator to allow starting decryption while calculation of the cryptogram is still going on at the Tag. A reader compliant to this part of ISO/IEC 29167 shall support both modes. A Tag shall support at least one mode for each authentication type, depending on its resources and capabilities as well as the features of the selected interface standard (e.g. partial result mode with Tag Identification and complete result mode with Mutual Authentication). Complete result mode may require the capability of the interface standard to handle long time outs or to signal the interrogator that a tag is still processing a command, depending on the Tag's performance See Annex E for detailed information.

In partial result mode a sequence of *Authenticate* commands needs to be sent to the Tag in order to complete the full authentication protocol. In order for the authentication to succeed the entire sequence must be executed successfully. The crypto suite state transitions triggered by the authentication payloads are specified in Clause 10 and in the state transition tables in Annex A.

The crypto suite state transitions and the Tag responses are according to the payloads of the *Authenticate* command sent by the Interrogator and the result mode (partial/complete) embedded in the Tag. The processing of the *Authenticate* command includes generation of an authentication cryptogram that may be returned in the Tag response.

During authentication, both RAMON encryption and AES encryption produce the result in byte order. Partial result mode in a Tag can take advantage of this fact. Completed portions of the result can be fetched by the Interrogator while portions that have not yet been produced can be fetched successively. In every Tag response that carries a portion of the result, the length of the remaining result bytes to be fetched is indicated. The final packet indicates a remaining length of zero bytes in its payload. In case of

any error, an Error Code is transmitted back using the Tag error-reply format defined in the related air interface standard. The error table is given in Annex B.

A Tag receiving a command with incorrect AuthMethod or Step fields, shall respond with a "insufficient privileges" or an "other error" error code. The crypto suite shall transit to the **Init** state.

An interrogator receiving a Tag's response with incorrect AuthMethod or Step fields shall reset the Tag and try to restart the communication.

8.2 State diagram and transitions for Tag identification

All *Authenticate* commands for Tag identification have the AuthMethod field set to 11_2 (AuthMethod 3) as specified in 10.3.

8.2.1 Partial Result Mode

Figure 4 illustrates the state transitions that apply to this crypto suite for Tag identification in partial result mode.

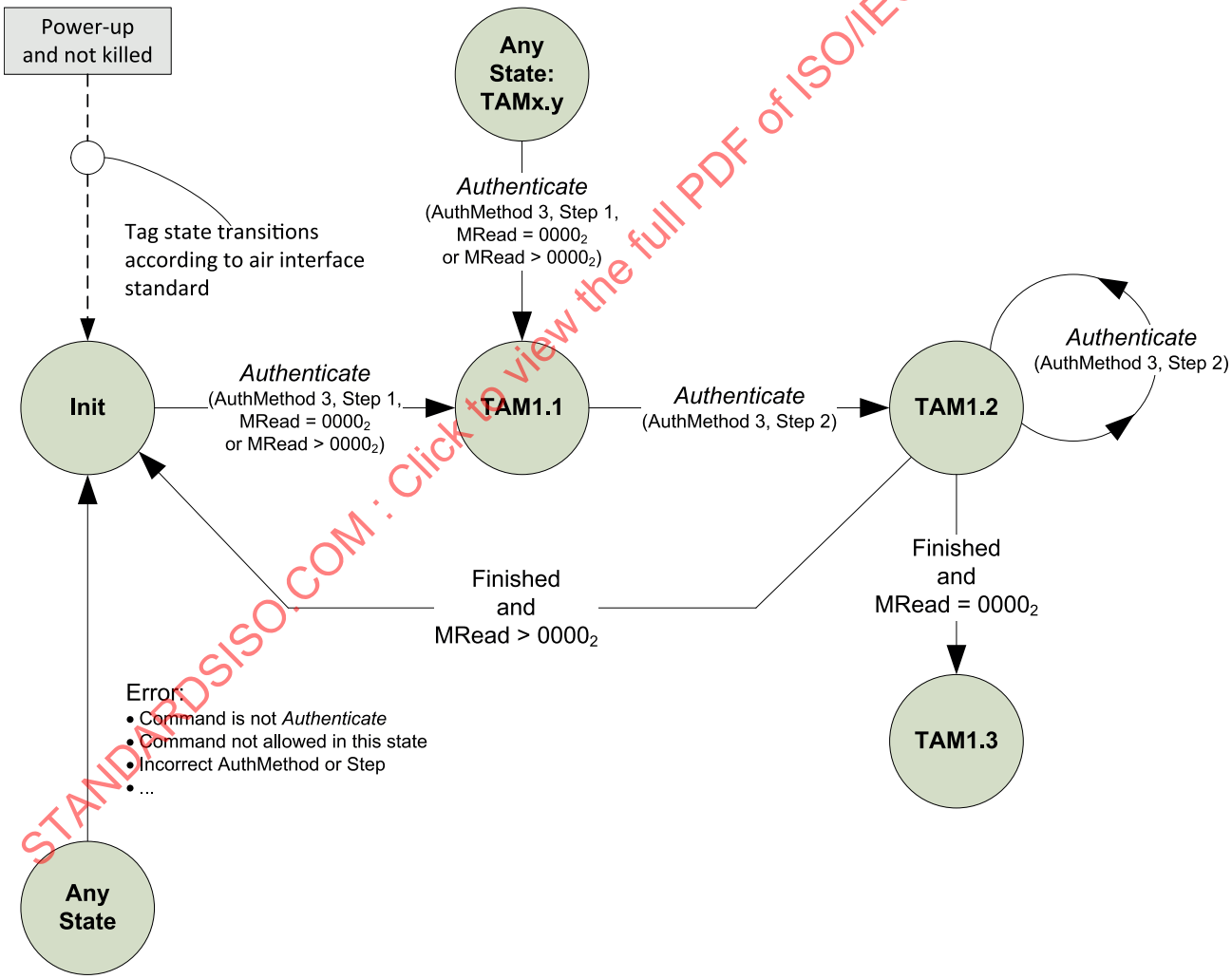


Figure 4 — Crypto suite state transitions for Tag identification in partial result mode

After power up, the crypto suite is in the **Init** state. Once the Tag receives an *Authenticate* command with AuthMethod 3 and payload for Step 1, it processes the command, sends a response confirming the reception of the command and transits to **TAM1.1** expecting an *Authenticate* command with payload for Step 2. When the Tag receives the first *Authenticate* command with AuthMethod 3 and payload for

Step 2, it processes the command, sends a response containing a partial result, transits to **TAM1.2** and remains in this state as long as there are authentication data bytes remaining to be sent. In **TAM1.2** the Interrogator sends as many *Authenticate* commands as required to fetch the entire authentication data produced by the Tag. The Tag indicates in the payload of the response message the number of bytes still available to fetch.

Whenever the Tag receives an *Authenticate* command with AuthMethod 3 and payload for Step 1, it resets all variables, transits to **TAM1.1** and starts processing the command. The crypto suite transits to state **TAM1.3** once it has sent out the last fragment of authentication cryptogram and Tag Identification was not used to read a part of the Tags memory.

In case of failure during one of the steps of the protocol, the crypto suite transits to the **Init** state.

8.2.2 Complete Result Mode

Figure 5 illustrates the state transitions that apply to this crypto suite for Tag identification in complete result mode.

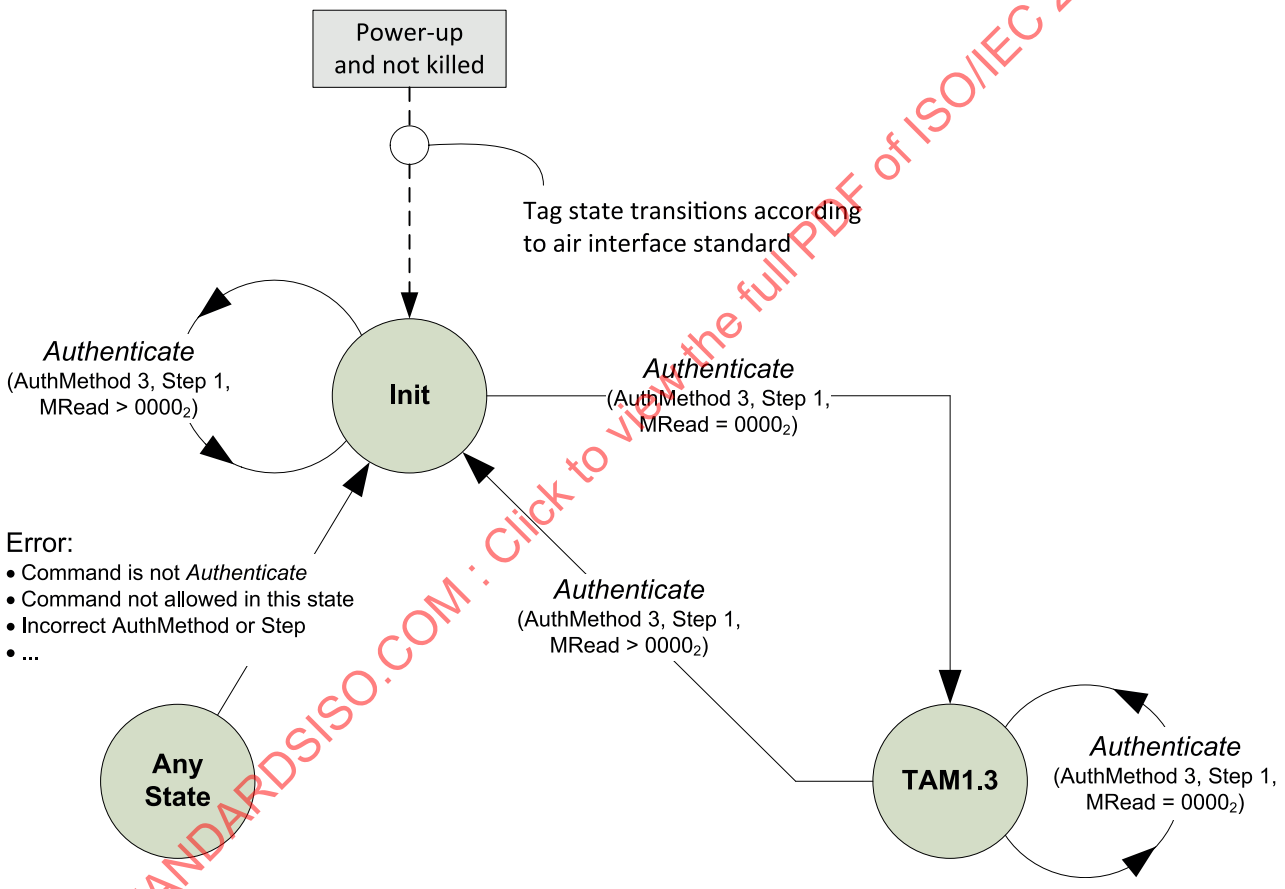


Figure 5 — Crypto suite state transitions for Tag identification in complete result mode

After power up, the crypto suite is in the **Init** state. Once the Tag receives an *Authenticate* command with AuthMethod 3, payload for Step 1 and MRead = 0000_2, it processes the command, sends the complete response and transits to **TAM1.3.**

In case of failure during one of the steps of the protocol, the crypto suite transits to the **Init** state.

8.3 State diagram and transitions for mutual authentication

Mutual authentication can be performed only after the Tag has been successfully identified and therefore is in state **TAM1.3**. Secure communication is possible only after successful mutual authentication that involves generation of the required session keys.

After successful Tag identification, the crypto suite transits to state **TAM1.3**. The Tag is ready to receive and process the *Authenticate* commands for mutual authentication now. All *Authenticate* commands for mutual authentication shall have the AuthMethod field set to 01_2 as specified in 10.3.

8.3.1 Partial Result Mode

Figure 6 illustrates the state transitions that apply to this crypto suite for Tag identification followed by mutual authentication, both in partial result mode, and secure communication. Once in state **TAM1.3** the Tag receives an *Authenticate* command with AuthMethod 1 and payload for Step 1, it processes the command, sends a response confirming the reception of the command and transits to **MAM1.1** expecting an *Authenticate* command with payload for Step 2. When the Tag receives the first *Authenticate* command with AuthMethod 1 and payload for Step 2, it processes the command, sends a response containing a partial result, transits to **MAM1.2** and remains in this state as long as there are authentication data bytes remaining to be sent. In **MAM1.2** the Interrogator sends as many *Authenticate* commands as required to fetch the entire authentication data produced by the Tag. The Tag indicates in the payload of the response message the number of bytes still available to fetch. After having transmitted the last partial response, indicated by setting the remaining number of bytes to zero, the Tag transits into state **SC,** expecting an *Authenticate* command *with AuthMethod 1 Step 3 (= Secure Communication)*. The Tag is ready for secure communication.

In case of failure during one of the steps of the mutual authentication, the crypto suite transits to state **Init**. If the command sequence for mutual authentication is interrupted by any other non-mutual authentication command sent to the Tag, the crypto suite transits to state **Init**.

If in state **SC** the Tag receives any command other than *Authenticate* (AuthMethod 1, Step3), the Tag transits to **Init** state. The secure channel is closed.

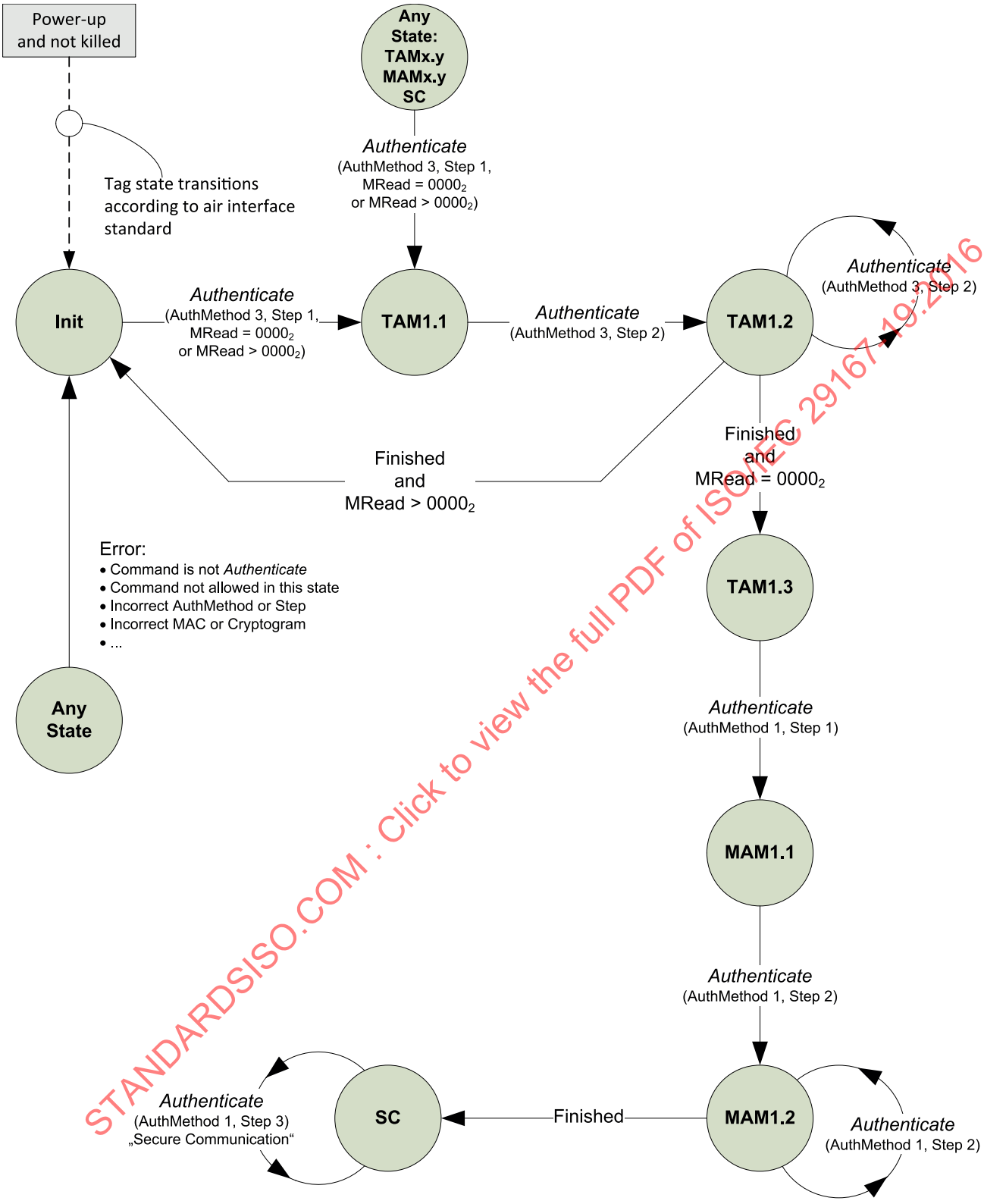


Figure 6 — Crypto suite state transitions for mutual authentication in partial result mode

8.3.2 Complete Result Mode

Figure 7 illustrates the state transitions that apply to this crypto suite for Tag identification followed by mutual authentication, both in partial result mode, and secure communication. Once in state **TAM1.3** the Tag receives an *Authenticate* command with AuthMethod 1 and payload for Step 1, it processes the

command, sends the complete response and transits to **SC** expecting a *Authenticate* command *with AuthMethod 1 Step 3 (= Secure Communication)*. The Tag is ready for secure communication.

In case of failure during the mutual authentication, the crypto suite transits to state **Init**.

If in state **SC** the Tag receives any command other than *Authenticate* (AuthMethod 1, Step3), the Tag transits to **Init** state. The secure channel is closed.

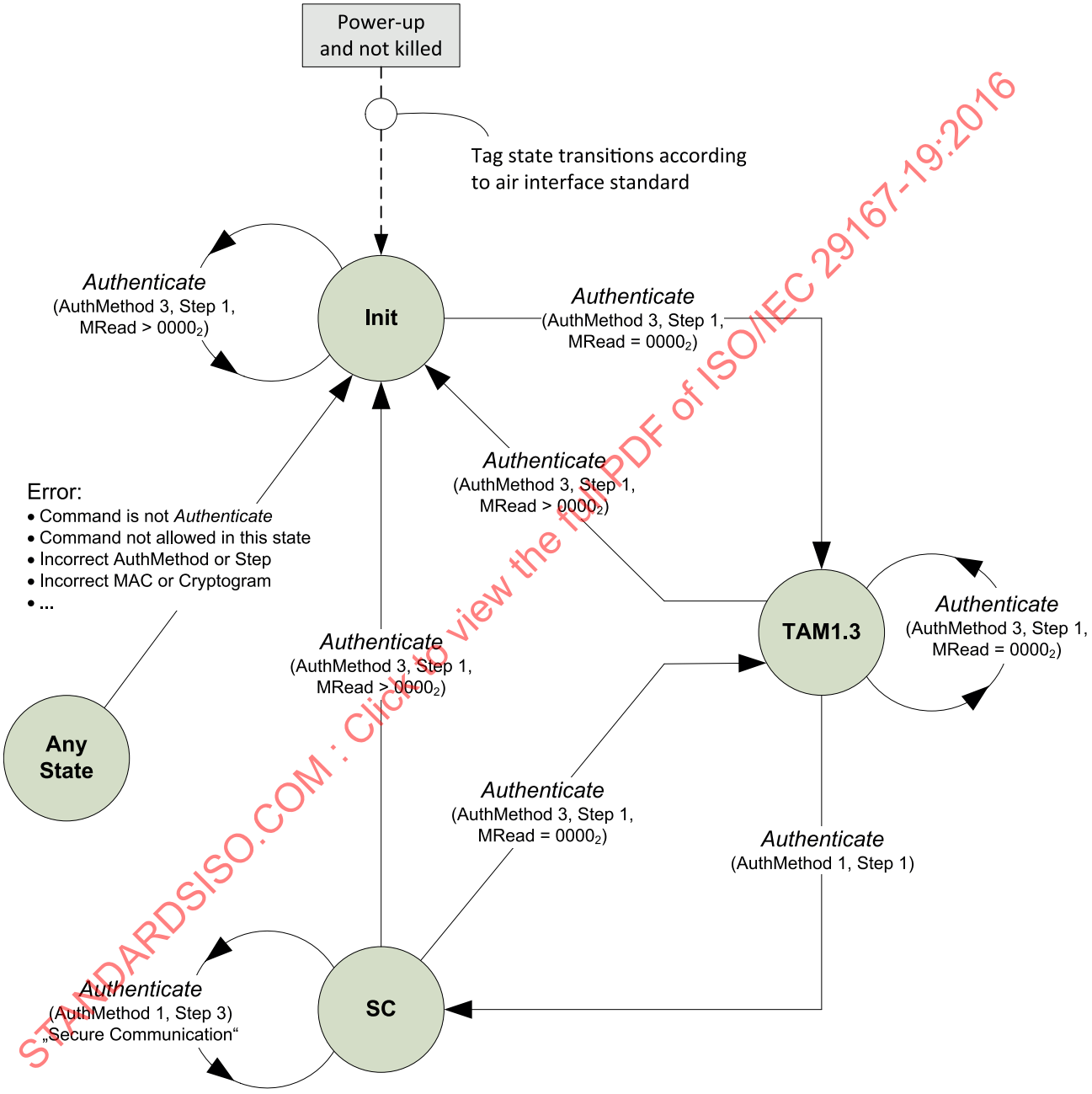


Figure 7 — Crypto suite state transitions for mutual authentication in complete result mode

8.3.3 Combination of complete and partial result mode

Complete result mode and partial result mode may be combined on a tag for the different authentication types, e.g. a tag may perform Tag identification in partial result mode and mutual authentication in complete result mode as a reasonable combination.

9 Initialization and resetting

In order to achieve non traceability, a Tag's unique identifier (e.g. UII, UID) shall be randomized at power on. For a possible implementation see Annex F.

10 Identification and authentication

10.1 Tag identification

The sequence of messages exchanged for Tag identification is depicted in Figure 8 for partial result mode. The sequence of messages exchanged for Tag identification is depicted in Figure 9 for complete result mode.

10.1.1 Partial Result Mode

The first message includes a random challenge generated by the Interrogator and sent to the Tag. In Step 1, the Tag responds with the total length of response that will be sent in. The Tag's response in Step 2 is an encrypted message that only the legitimate Interrogator can decrypt, since it possesses the necessary private key.

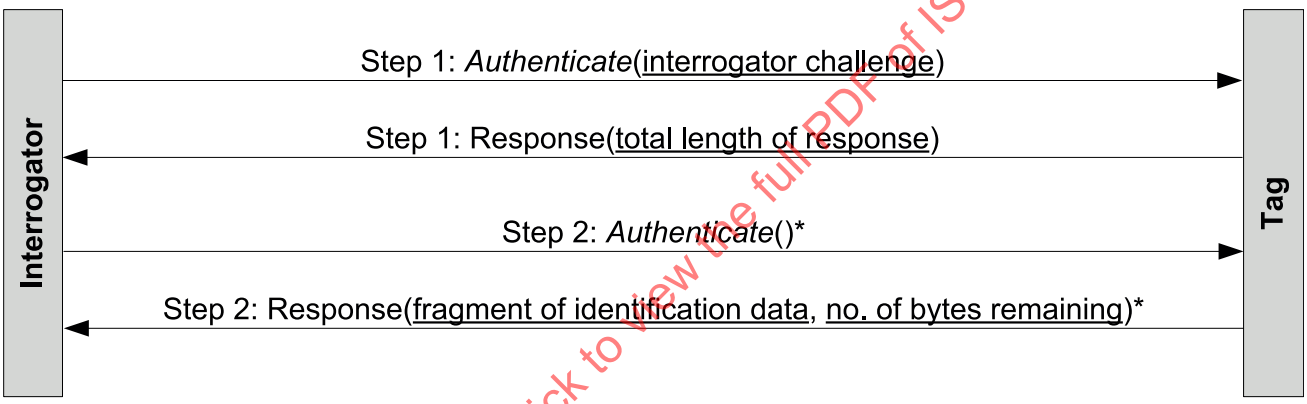


Figure 8 — Message exchange for Tag identification in partial result mode

In Step 1, the Interrogator challenge is delivered to the Tag. This message is used to request the Tag to send its identification data. Upon reception of this message, the Tag starts calculating the response. The Tag's first response is the total length of the identification cryptogram.

In Step 2, the Interrogator retrieves the fragments of the Tag's identification cryptogram by chaining further *Authenticate* commands and responses. Once the Interrogator has fetched the entire identification data it is able to identify the Tag.

10.1.2 Complete Result Mode

The first and only message includes a random challenge generated by the Interrogator and sent to the Tag. The Tag's response is an encrypted message that only the legitimate Interrogator can decrypt, since it possesses the necessary private key.

In Step 1, the Interrogator challenge is delivered to the Tag. The Tag starts calculating the response. If the Tag has finished the calculation completely, it transmits the identification data to the reader, marking this as Step 2 and setting the remaining bytes to zero. Once the Interrogator has fetched the identification data it is able to identify the Tag.

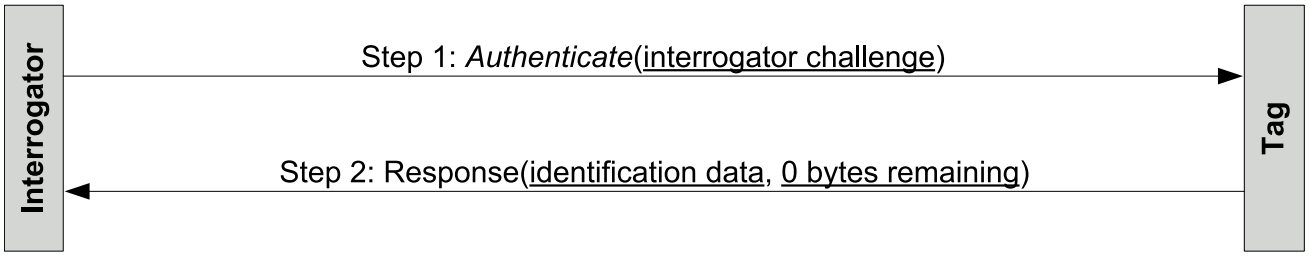


Figure 9 — Message exchange for Tag identification in complete result mode

10.2 Mutual authentication

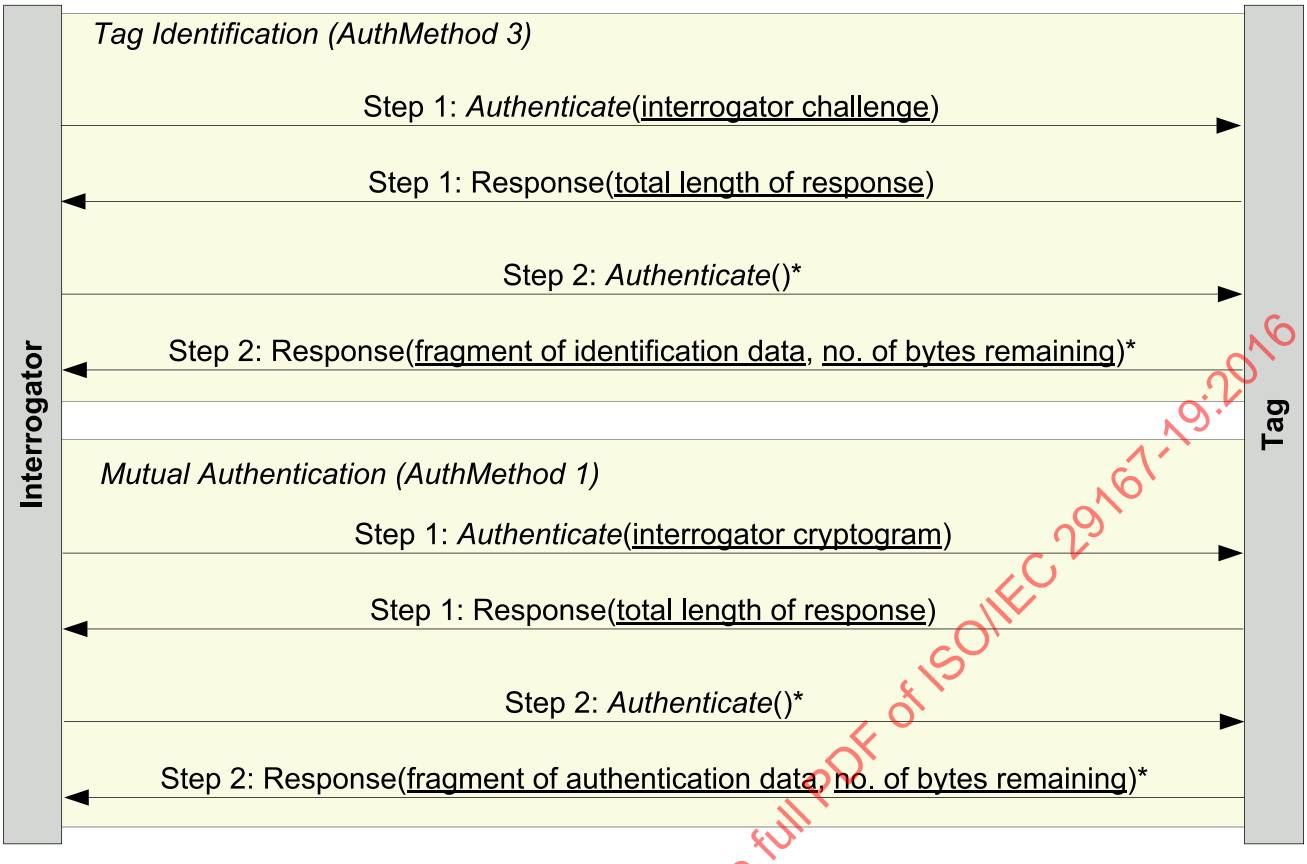
Before mutual authentication can be performed, the Tag has to be identified by the Interrogator. The same messages, as specified in 10.1, need to be exchanged for this purpose. These steps are repeated in Figures 10 and 11 for completeness.

10.2.1 Partial Result Mode

The sequence of messages exchanged for mutual authentication is depicted in Figure 10 for the partial result mode. The first message to the Tag after successful Tag identification includes the Interrogator cryptogram. The Tag is in state **TAM1.3**. The Tag verifies the Interrogator cryptogram; if this is successful, it returns the total length of the authentication cryptogram, transits to **MAM1.1** and starts generating its own cryptogram. If the verification of the Interrogator cryptogram has failed, the Tag returns a Crypto Suite error code and transits to **Init** state.

In Step 2, the Interrogator retrieves the fragments of the Tag's authentication cryptogram by chaining further *Authenticate* commands and responses. Once the Interrogator has fetched the entire authentication data it is able to authenticate the Tag. After the Tag has sent out the first fragment of authentication data, it transits to **MAM1.2**. After the Tag has sent out the last fragment of authentication data, it transits to SC.

If the Tag receives a message that is not formatted as specified in 10.3, it shall respond with a Crypto Suite error code and transit to **Init** state.



* The message is sent multiple times to retrieve all the remaining bytes.

Figure 10 — Message exchange for mutual authentication in partial result mode

10.2.2 Complete Result Mode

The sequence of messages exchanged for mutual authentication is depicted in Figure 11 for the complete result mode. The next message to the Tag after successful Tag identification includes the Interrogator cryptogram. The Tag is in state **TAM1.3**. The Tag verifies the Interrogator cryptogram; if this is successful, it starts generating its own cryptogram. If the verification of the Interrogator cryptogram has failed, the Tag returns a Crypto Suite error code and transits to **Init** State.

If the Tag has finished the calculation completely, it transmits the authentication data to the reader, marking this as Step 2 and setting the remaining bytes to zero. Once the Interrogator has fetched the authentication data it is able to authenticate the Tag. After successfully transmitting the authentication data to the interrogator, the Tag transits to state **SC**.

If the Tag receives a message that is not formatted as specified in 10.3, it shall respond with a Crypto Suite error code and transit to **Init** state.

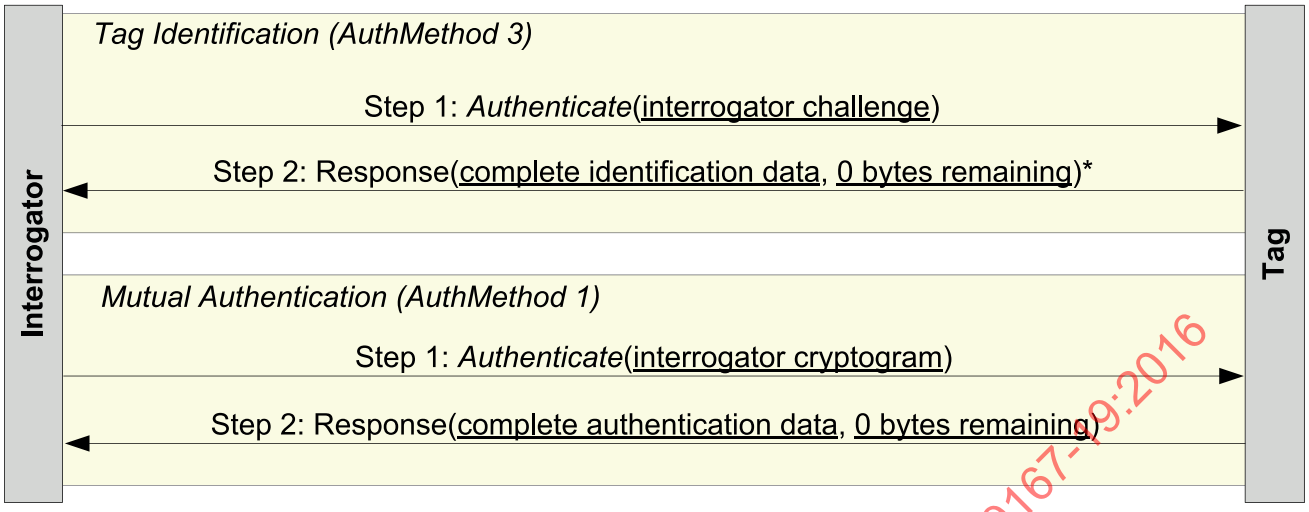


Figure 11 — Message exchange for mutual authentication in complete result mode

10.3 The Authenticate command

Message and *Response* are part of the security commands that are described in the air interface specification. The following subclauses are based on the *Authenticate* command described in the related air interface specification. They describe the formatting and coding of the Message field of an *Authenticate* command.

10.3.1 Message formats for Tag identification

The coding of the Message field for Tag Identification, AuthMethod 3 Step 1 is shown in Table 7. This message transmits the Interrogator Challenge to the Tag. KeySelect allows selecting one K_E out of a number of keys. If only one key is supported, KESel shall be 00h by default. If the interrogator chooses a value for KESel which points to a key K_E not available in the Tag, the Tag shall respond with a "Not supported" error code and transit to Init state. MRead shall be set to 0000_2 for Tag identification.

Table 7 — Message format for Tag Identification, AuthMethod 3 Step 1

	AuthMethod	Step	MRead	RFU	KeySelect	Interrogator Challenge
# of bits	2	2	4	8	8	128
Description	11_2	01_2	0000_2	$0000\ 0000_2$	KESel[7:0]	CH_{I1} [127:0]

An Interrogator shall set all RFU bits of the Message field to "0". An Interrogator not setting all RFU bits to "0" is not compliant with this part of ISO/IEC 29167. A tag receiving a Message field with RFU bits set other than "0" shall respond with a "Not supported" error code and return to **Init state**.

A Tag using partial result mode, requires additional commands to transmit the partial results to the Interrogator while in the state **TAM1.1** or **TAM1.2.** The coding of the Message field in state **TAM1.1** and **TAM1.2** for AuthMethod 3 Step 2 is shown in Table 8. This coding is used to retrieve the partial response bytes calculated by the Tag.

Table 8 — Message format for Tag Identification, AuthMethod 3 Step 2

	AuthMethod	Step	RFU
# of bits	2	2	4
Description	11_2	10_2	0000_2

An Interrogator shall set all RFU bits of the Message field to “0”. An Interrogator not setting all RFU bits to “0” is not compliant with this part of ISO/IEC 29167. A tag receiving a Message field with RFU bits set other than “0” shall respond with a “Not supported” error code and return to **Init state**.

10.3.1.1 Message format for RAMON memory read (optional)

The Tag identification mechanism additionally can be used to read out the Tag’s memory instead of the SID. To read from the Tag’s memory, the MRead field shall be set to a value other then 0000_2.

MRead indicates the memory address to read from and shall be in the range 0001_2 .. 1111_2. The amount of memory bytes transmitted back to the Interrogator is defined by Tag manufacturer and cannot be controlled by this message.

10.3.2 Message formats for Mutual Authentication

The coding of the Message field of the *Authenticate* command for Mutual Authentication (AuthMethod 1, Step 1) is shown in Table 9. This message transmits the Interrogator Cryptogram to the Tag.

KeySelect allows to select one keyset (key pair K_{ENC} + K_{MAC}) out of a number of keysets. If only one keyset is supported, KSel shall be 00h by default. If the interrogator chooses a value for KSel which points to a keyset not available in the Tag, the Tag shall respond with a “Not supported” error code and stay in the current state.

Table 9 — Message format for Mutual Authentication (AuthMethod 1, Step 1)

	AuthMethod	Step	RFU	KeySelect	Interrogator Cryptogram
# of bits	2	2	4	8	512
Description	01_2	01_2	0000_2	KSel[7:0]	CG_I [511:0]

An Interrogator shall set all RFU bits of the Message field to “0”. An Interrogator not setting all RFU bits to “0” is not compliant with this part of ISO/IEC 29167. A tag receiving a Message field with RFU bits set other than “0” shall respond with a “Not supported” error code and return to **Init state**.

The Interrogator Cryptogram is calculated as follows:

a) Select an authentication key pair: K_{ENC} (KSel), K_{MAC} (KSel).

b) Generate random challenge (16 bytes): CH_{I2}.

c) Construct the plaintext message (48 bytes): $S = CH_{I2} \,||\, \text{IID} \,||\, CH_T \,||\, SID$.

d) Encrypt the plaintext (48 bytes, without padding): $C = ENC\,(K_{ENC}, S)$.

e) Compute the *MAC* (16 bytes): $M = MAC\,(K_{MAC}, C)$.

f) The interrogator cryptogram is the concatenation $C \,||\, M$ (64 bytes).

NOTE 1 The Interrogator has obtained the *SID* and CH_T from the previous identification response that was sent by the Tag. In the calculation of the Interrogator Cryptogram, the *SID* is used without any signature, even though a signature might have been included in the Tag authentication response.

NOTE 2 Since the input data to AES encryption is already a multiple of block size, no padding needs to be applied in step d).

A Tag using partial result mode requires additional commands to transmit the partial results to the Interrogator while in the state **MAM1.2.** The coding of the Message field for Mutual Authentication, *AuthMethod 1 Step 2*, is shown in Table 10. This coding is used to retrieve the response bytes of a partial result calculated by the Tag.

Table 10 — Message format Mutual Authentication, AuthMethod 1 Step 2

	AuthMethod	Step	RFU
# of bits	2	2	4
Description	01_2	10_2	0000_2

An Interrogator shall set all RFU bits of the Message field to "0". An Interrogator not setting all RFU bits to "0" is not compliant with this part of ISO/IEC 29167. A tag receiving a Message field with RFU bits set other than "0" shall respond with a "Not supported" error code and return to **Init state**.

10.4 Authentication response

The Tag sends a response message to each *Authenticate* command. *Message* and *Response* are part of the security commands that are described in the air interface specification. The following subclauses of this clause are based on the response described in the related air interface specification. They describe the formatting and coding of the Response field of a response related to an *Authenticate* command.

10.4.1 Response formats for Tag identification

10.4.1.1 Partial Result Mode

The first response shall indicate the overall length of response data and does not carry any bytes of the response data itself. The subsequent response messages transmit fragments of the response data in consecutive order. Each response message indicates the remaining number of bytes to be transmitted. The coding of the Tag Response field for Tag Identification, *AuthMethod 3 Step 1,* is shown in Table 11. In this state the Tag shall only transmit the total length of response. After having transmitted the response frame for AuthMethod 3, Step 1, the Tag shall transit to state **TAM1.1**.

Table 11 — TAM Format of the Tag Response field for Tag Identification, AuthMethod 3 Step 1

	AuthMethod	Step	RFU	Remaining Length
# of bits	2	2	8	12
Description	11_2	01_2	00h	xxxh, "xxx" indicates the total length of response data

A Tag shall set all RFU bits of the Tag Response field in step a) to "0". A Tag not setting all RFU bits to "0" is not compliant with this part of ISO/IEC 29167. An Interrogator receiving an Authenticate Response field with RFU bits set other than "0" shall ignore the RFU bits and try to continue communication with the Tag.

An interrogator receiving a response frame formatted as shown in Table 11 shall continue with *Authenticate* commands for AuthMethod 11_2 with payload for step b).

The coding of the Tag Response field for Tag Identification, *AuthMethod 3 Step 2,* is shown in Table 12. When the Tag receives the first *Authenticate* command for AuthMethod 3, Step 2, it processes the command, sends the response, transits from state **TAM1.1** into state **TAM1.2** and remains in **TAM1.2** as long as there are identification data bytes remaining to be sent and no error occurred. The response data is calculated by the Tag in consecutive order. See Annex C for a detailed description of the clear text input to the authentication cryptogram, transmitted in the response data field. While the calculation on the Tag is ongoing, the Tag can transmit already available fragments of the response data. The Tag shall indicate the remaining number of bytes to be fetched in the Remaining Length field. The Remaining Length encoded to 000h indicates that this is the last fragment.

Table 12 — TAM Format of the Tag Response field for Tag Identification, AuthMethod 3 Step 2

	AuthMethod	Step	RFU	Response Data Fragment	RFU	Remaining Length
# of bits	2	2	4	Variable (n times 8)	4	12
Description	11_2	10_2	0000_2	Fragment from the result of: {RM_ENC (K_E, MIX(CH_{I1}, RN_T, TLV record, '00' byte)} [1023:0]	0000_2	xxxh, "xxx" indicates the remaining length of response data

A Tag shall set all RFU bits of the Tag Response field in step b) to "0". A Tag not setting all RFU bits to "0" is not compliant with this part of ISO/IEC 29167. An Interrogator receiving an Authenticate Response field with RFU bits set other than "0" shall ignore the RFU bits and try to continue communication with the Tag.

10.4.1.2 Complete Result Mode

If the calculation of the response data from the Tag has been finished, the Tag transmits the whole response data in a single response. The coding of the Tag Response field is shown in Table 12. A Tag using Complete Result Mode shall set the Remaining Length field to 000h to indicate that this is the only and complete response. The Response Data Fragment contains the complete RAMON cryptogram, consisting of 128 bytes (1024 bits).

10.4.2 Response formats for mutual authentication

After having received the *Authenticate* command for AuthMethod 1 with payload for step a) and having successfully verified the Interrogator cryptogram, the Tag may start its calculation of the Tag cryptogram. The Tag cryptogram is calculated as follows.

a) Verify and decrypt the Interrogator cryptogram:

   1) Recompute the message authentication code: $MAC(K_{MAC}, C)$.

   2) Compare the $MAC(K_{MAC}, C)$ with received MAC.

   3) If not equal, transmit a crypto suite error code. If equal, continue to 4).

   4) Decrypt the ciphered message: $DEC(K_{ENC}, C)$.

b) Compare received CH_T and SID with stored values. If not equal, transmit a crypto suite error code. If equal, continue with step c).

c) Interrogator authenticated successfully. Tag is ready for Secure Communication.

d) Generate the Tag cryptogram (48 bytes): $S = CH_T \,||\, SID \,||\, CH_{I2} \,||\, IID$.

e) Encrypt the Tag cryptogram (48 bytes, without padding): $C = ENC(K_{ENC}, S)$, IV = 0.

f) Compute the MAC (16 bytes): $M = MAC(K_{MAC}, C)$.

g) Transmit response containing the cryptogram (64 bytes): $CG_T = C \,||\, M$.

NOTE 1 In the calculation of the Tag cryptogram, the *SID* is included without the signature.

NOTE 2 Since the input data to AES encryption is already a multiple of block size, no padding needs to be applied in step d).

When the interrogator has received the response message it proceeds similarly.

a) Verify and decrypt the Tag cryptogram:

   1) Recompute the message authentication code: MAC(K_{MAC}, C).

   2) Compare the MAC(K_{MAC}, C) with received MAC.

   3) If not equal, authentication failed. If equal, continue to step 4).

   4) Decrypt the ciphered message: DEC(K_{ENC}, C).

b) Compare received CH_T, *SID*, CH_{I2} and *IID* with stored values. If not equal, authentication failed. If equal, continue with step c).

c) Tag authenticated successfully. The Interrogator may proceed with Secure Communication,

10.4.2.1 Partial Result Mode

After having received the *Authenticate* command for AuthMethod 1 with payload for step a), the Tag verifies the Interrogator cryptogram. Upon successful verification of the Interrogator cryptogram, the Tag transits from **TAM1.3** to **MAM1.1** and sends the response frame formatted as shown in Table 13. This first response shall indicate the overall length of response data and does not carry any bytes of the response data itself. In state **MAM1.1** the Tag shall only transmit a Remaining Length information, indicating the total length of response. The subsequent response messages transmit fragments of the response data in consecutive order. Each response message indicates the remaining number of bytes to be transmitted.

Table 13 — MAM Format of the Tag Response field for Mutual Authentication, AuthMethod 1 Step 1

	AuthMethod	Step	RFU	Remaining Length
# of bits	2	2	8	12
Description	01_2	01_2	00h	xxxh, “xxx” indicates the total length of response data

A Tag shall set all RFU bits of the Tag Response field in step a) to “0”. A Tag not setting all RFU bits to “0” is not compliant with this part of ISO/IEC 29167. An Interrogator receiving an Authenticate Response field with RFU bits set other than “0” shall ignore the RFU bits and try to continue communication with the Tag.

An interrogator receiving a response frame formatted as shown in Table 13 shall continue with *Authenticate* commands for AuthMethod 1 with payload for step b).

In state **MAM1.1** the Tag accepts *Authenticate* commands for AuthMethod 1 with payload for step b). When the Tag receives the first *Authenticate* command for AuthMethod 1, Step 2, it processes the command, sends the first fragment of the authentication data in response, transits from state **MAM1.1** into state **MAM1.2** and remains in **MAM1.2** as long as there are authentication data bytes remaining to be sent and no error occurred. The response frame in state **MAM1.2** is shown in Table 14.

The processing order in calculating the Tag cryptogram can be arranged in a way that allows partial results to be ready before the cryptogram is complete: After encryption of a data block this part may be transmitted to the interrogator, provided it has been included in the CMAC calculation, and the buffer space can be recovered. While the calculation on the Tag is ongoing, the Tag can transmit already available fragments of the response data. The Tag shall indicate the remaining number of bytes to be fetched in the Remaining Length field.

The last fragment shall be indicated by setting the Remaining Length field to 000h.

Table 14 — MAM – Format of the Tag Response field for Mutual Authentication, AuthMethod 1 Step 2

	AuthMethod	Step	RFU	Response Data Fragment	RFU	Remaining Length
# of bits	2	2	4	Variable	4	12
Description	01_2	10_2	0000_2	Fragment of the Tag cryptogram.	0000_2	xxxh, "xxx" indicates the remaining length of response data

A Tag shall set all RFU bits of the Tag Response field in step b) to "0". A Tag not setting all RFU bits to "0" is not compliant with this part of ISO/IEC 29167. An Interrogator receiving an Authenticate Response field with RFU bits set other than "0" shall ignore the RFU bits and try to continue communication with the Tag.

10.4.2.2 Complete Result Mode

If the calculation of the response data from the Tag has been finished, the Tag transmits the whole response data in a single response. The coding of the Tag Response field is shown in Table 14. A Tag using Complete Result Mode shall set the Remaining Length field to 000h to indicate that this is the only and complete response.

10.4.3 Authentication error response

A Tag that encounters an error during the execution of a cryptographic suite operation shall send an error reply to the Interrogator. The details of these error replies are defined in the respective air interface standards.

Annex B contains a listing of the Error Conditions that may result from the operation of this cryptographic suite.

10.5 Determination of Result Modes

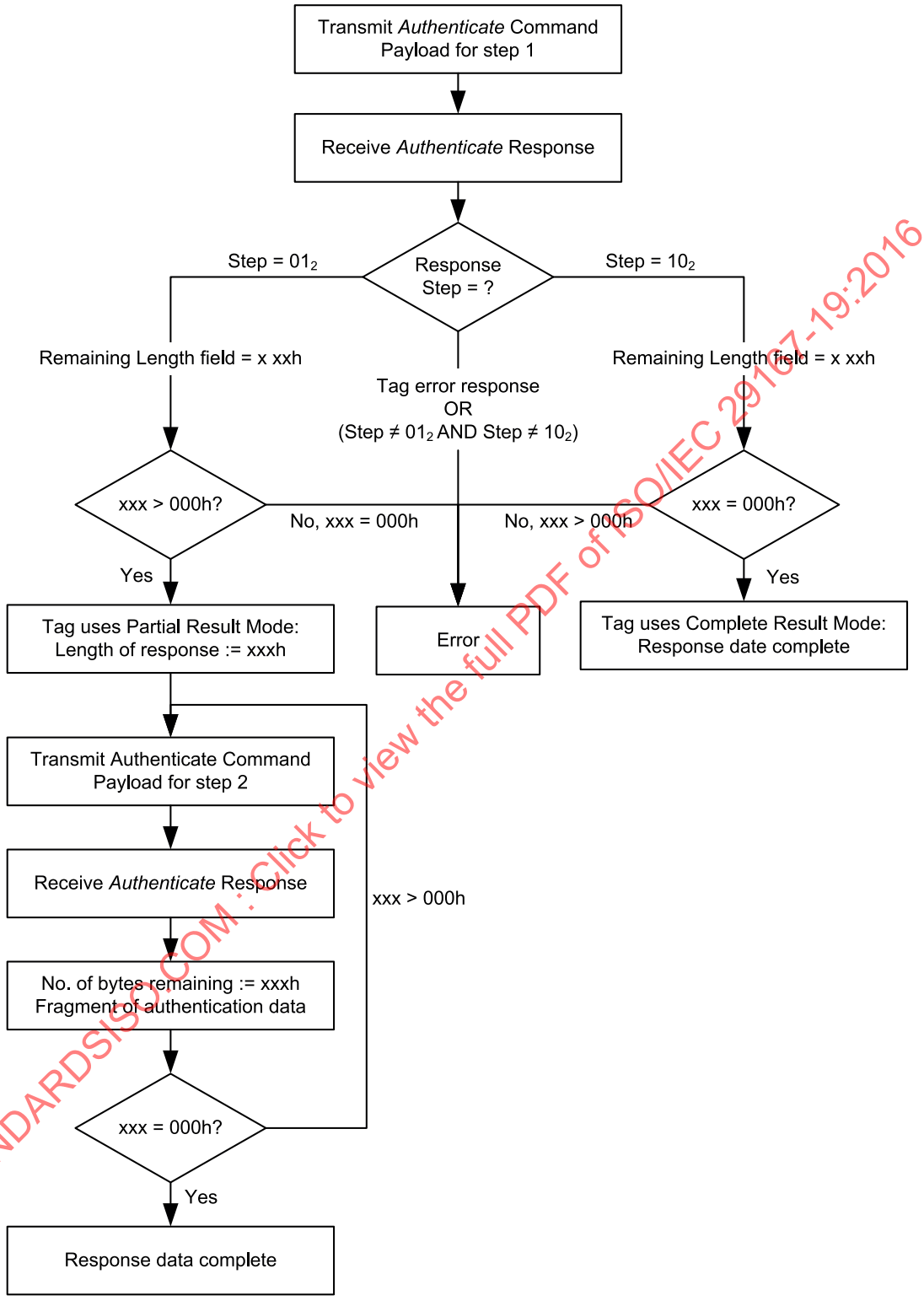


Figure 12 — Determination of result modes from Interrogators point of view.

The determination of the result modes, used from the Tag is shown in Figure 12. An Interrogator shall check the Step and the Remaining Length field in the Tags response to determine between complete- and partial response mode.

11 Secure communication

While in state SC, the Tag is able to process Secure Communication, which supports the transmission of MAC-secured and optionally encrypted data fields. *Message* and *Response* are part of the security commands that are described in the air interface specification.

11.1 Secure communication command

The following subclauses are based on the *Authenticate* command described in the related air interface specification. They describe the formatting and coding of the Message field of the *Authenticate* command, which is used to perform Secure Communication.

The coding of the Message field of the *Authenticate* command is shown in Table 15. The Message field of the *Authenticate* command contains application data, secured with a MAC. The data can be either encrypted or unencrypted. The value of the SCFlags field indicates whether '*encryption and MAC*' or '*MAC only*' is applied. The MAC is generated by the Interrogator using the session key, S_{MAC}. For encryption, the session key, S_{ENC}, is used. The Tag uses the same keys to verify the MAC and decrypt the ciphered data. For secure communication, the use of a MAC is mandated. If encryption is used, the data shall be encrypted first and then the MAC applied to the encrypted data. See 11.4.1 for more details.

Table 15 — Message format for secure communication (AuthMethod 1 Step 3)

	AuthMethod	Step	RFU	SCFlags	Dflags	Command Data	MAC
# of bits	2	2	4	4	4	Variable, n times 8	128
Description	01_2	11_2	0000_2	Security Level	Classification of data field	Plain or ciphered data (depending on SCFlags)	message authentication code

The SCFlags field contains information about the security level of communication for this message exchange. The possible values are listed in Table 16.

Table 16 — SCFlags for secure communication

Value	Description
0101_2	Command and response with MAC, no encryption
0111_2	Command and response with MAC, command encrypted
1101_2	Command and response with MAC, response encrypted.
1111_2	Command and response with MAC, command and response encrypted.
all other values	RFU

The Dflags field contains information about the data encapsulated in this message. The possible values are listed in Table 17.

Table 17 — Dflags for secure communication

Value	Description
0000_2	Proprietary command or data
0001_2	Commands defined in this crypto suite (see 11.3)
0010_2	ISO/IEC 18000-63 commands
0011_2	ISO/IEC 7816-4 APDUs
0101_2 ... 1111_2	RFU

A Tag authentication or mutual authentication shall not be included as payload in the Secure communication commands, described in this clause. For a Tag or mutual authentication, the procedures and commands described in 10.3 and 10.4 shall be used.

11.2 Secure Communication response

Message and *Response* are part of the security commands that are described in the air interface specification. The following subclauses are based on the response described in the related air interface specification. They describe the formatting and coding of the Response field of a response related to an *Authenticate* command.

The Tag Response Field of the *Authenticate* response contains application data, secured with a MAC. The data can be either encrypted or unencrypted. The value of the SCFlags field of the previous command indicates whether encryption and MAC or MAC only is applied. The MAC is generated by the Tag using the session key S_{MAC}. For encryption the session key S_{ENC} is used. The Interrogator uses the same keys to verify the MAC and decrypt the ciphered data. For secure communication, the use of a MAC is mandated. If encryption is used, the data shall be encrypted first and then the MAC applied to the encrypted data. See 11.4.2 for more details.

The coding of the Tag Response Field is shown in Table 18. The Response Information Field may be empty if the Tag has no data to return back to the Interrogator.

Table 18 — Format of the Tag Response field in the Secure Communication response frame

	AuthMethod	Step	RFU	Response Information Field	MAC
# of bits	2	2	4	Variable, n times 8	128
Description	01_2	11_2	0000	Plain or ciphered data (depending on SCFlags in previous command)	Message authentication code

11.2.1 Secure communication error response

A Tag that encounters an error during the execution of a cryptographic suite operation shall send an error reply to the Interrogator. The details of these error replies are defined in the respective air interface standards.

Annex B contains a listing of the Error Conditions that may result from the operation of this cryptographic suite.

11.3 Encoding of Read and Write commands for secure communication

This Crypto Suite supports two commands for record handling: *ReadRecord* and *WriteRecord*. Each record is addressed by a record address and contains 16 bytes of data. Reference to a record which is not contained in the Tag will cause a "Memory overrun" error. Each record number is unique and sequential.

The *ReadRecord* and *WriteRecord* commands shall be encapsulated in the *Command Data* field of the Message field of an *Authenticate* command (see Table 15). The coding of the *ReadRecord* command is shown in Table 19. The coding of a *WriteRecord* command is shown in Table 20.

The Dflag in the Message (see Table 15) field shall be set to 0001_2 to indicate this type of secure command.

The Tag shall only accept read and write commands secured with a MAC. See Table 16 (SCFlags) for more details.

Table 19 — Command Data of the *ReadRecord* command

	Command Identifier	RFU	Record address	Number of records
# of bits	4	4	8	8
Description	0001_2	$0..0_2$	Record address to start reading	Overall number of records to read

Table 20 — Command Data of the *WriteRecord* command

	Command Identifier	RFU	Record address	Data
# of bits	4	4	8	n*128
Description	0010_2	$0..0_2$	Record address to start writing	Data for n complete records

The *ReadRecord* and *WriteRecord* response from the tag shall be encapsulated in the Response Information Field of the Tag Response Field of a Secure Comunication response. The format of the Tag's response field is shown in Table 18. The Response Data field may be empty if the Tag has no data to return back to the Interrogator. The response to the *ReadRecord* command gives the contents of the addressed record or records.

Command identifiers other than 0001_2 and 0010_2 are RFU. A Tag receiving an RFU command identifier shall respond with a "Not supported" error code. An Interrogator using an RFU command identifier is not compliant to this part of ISO/IEC 29167.

An Interrogator shall set all RFU bits of the *Command Data* Field to "0". An Interrogator not setting all RFU bits to "0" is not compliant with this part of ISO/IEC 29167. A tag receiving a Command Data Field with RFU bits set other than "0" shall respond with a "Not supported" error code and transit into **Init** state.

11.4 Application of secure messaging primitives

The following figures demonstrate the application of secure messaging functions to the various message types that may occur. Please note that the commands are always generated by the Interrogator while the responses are generated by the Tag.

Note that different session keys are used for encryption and CMAC calculation: encryption requires session key, S_{ENC}, and CMAC calculation requires S_{MAC}.

In all following cases, the calculation of CMAC, a MAC chaining value (MCV), is required; which is derived from the send sequence counter by the equation MCV = SSC:

11.4.1 Secure Communication command messages

For secure commands, the CMAC calculation always includes the SCFlags and the Dflags, in addition to the data or encapsulated command field; see Figure 13. The MAC is appended to the command which is thereby extended by 16 bytes.

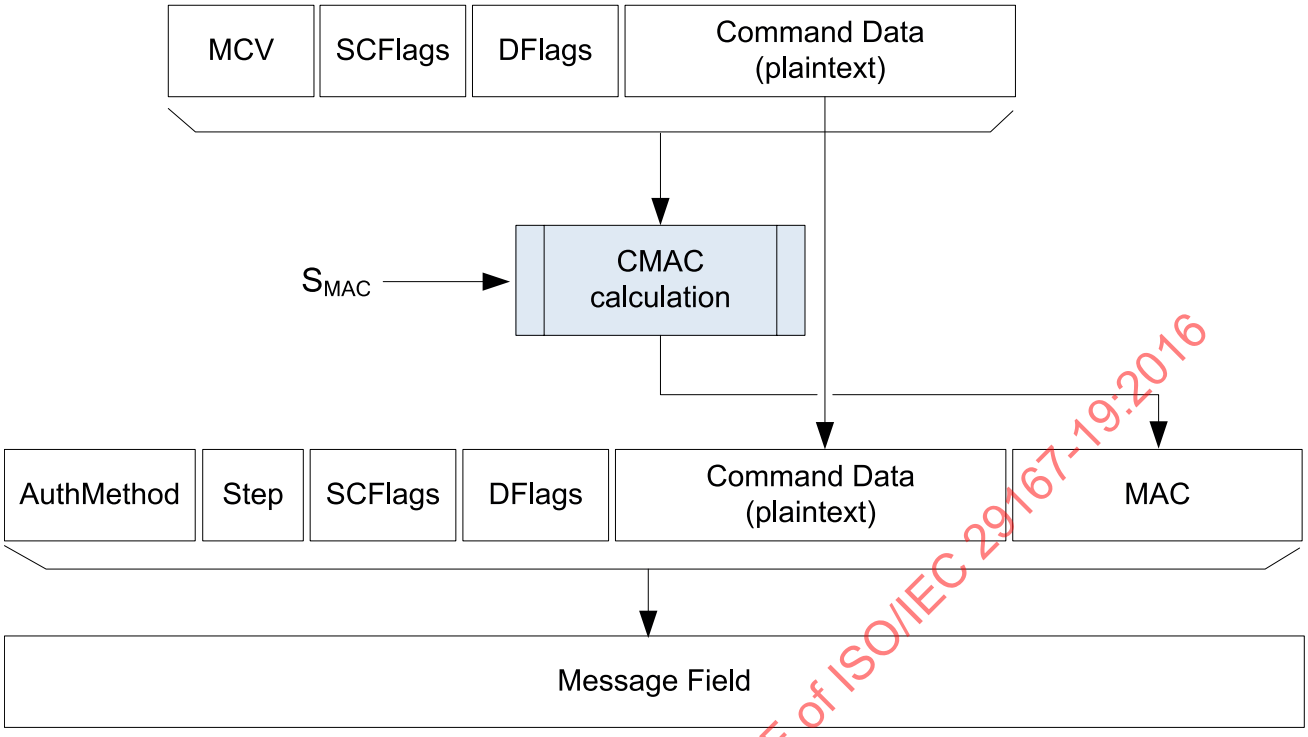


Figure 13 — *Secure Communication* Command MAC

The formation of an encrypted secure command is a bit more complex, as the SCFlags and Dflags have to be interpreted by the Tag and therefore must remain in the clear. Only the data or encapsulated command part is, after padding, encrypted with AES in CBC mode and replaces the previous plaintext field. Then, in a second step, the flags and the encrypted data or encapsulated command are together entered into the CMAC calculation, and the MAC is appended. The formation of an encrypted secure command is shown in Figure 14.

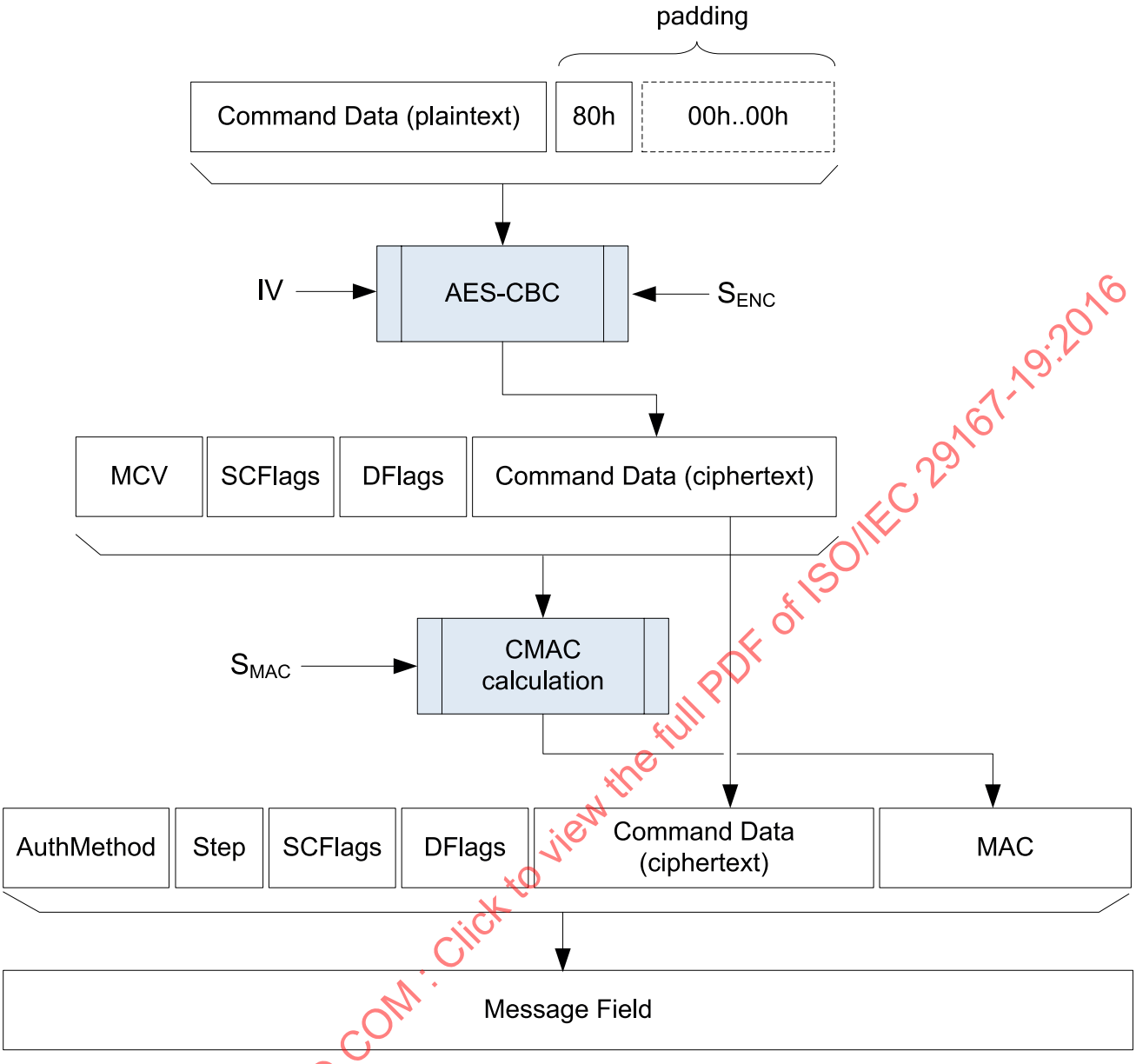


Figure 14 — Secure Communication Command ENC (with padding) and MAC

11.4.2 Secure Communication response messages

In the case of response messages without encryption, there are two cases to distinguish, depending on the presence or absence of response data. In both cases the result of the CMAC calculation is appended to the (unsecure) message, and the size of the response message field is increased by the size of the MAC, 16 bytes.

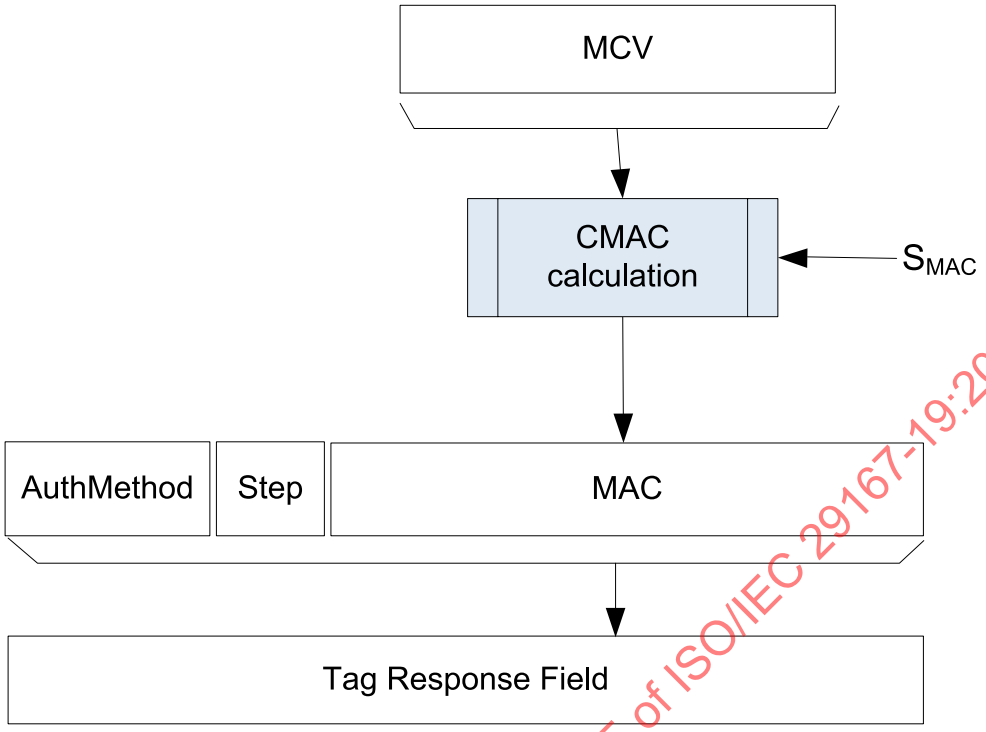


Figure 15 — Response MAC (no data)

Figure 15 above demonstrates the case where a response message comprises no response data. The CMAC function is calculated over the MCV.

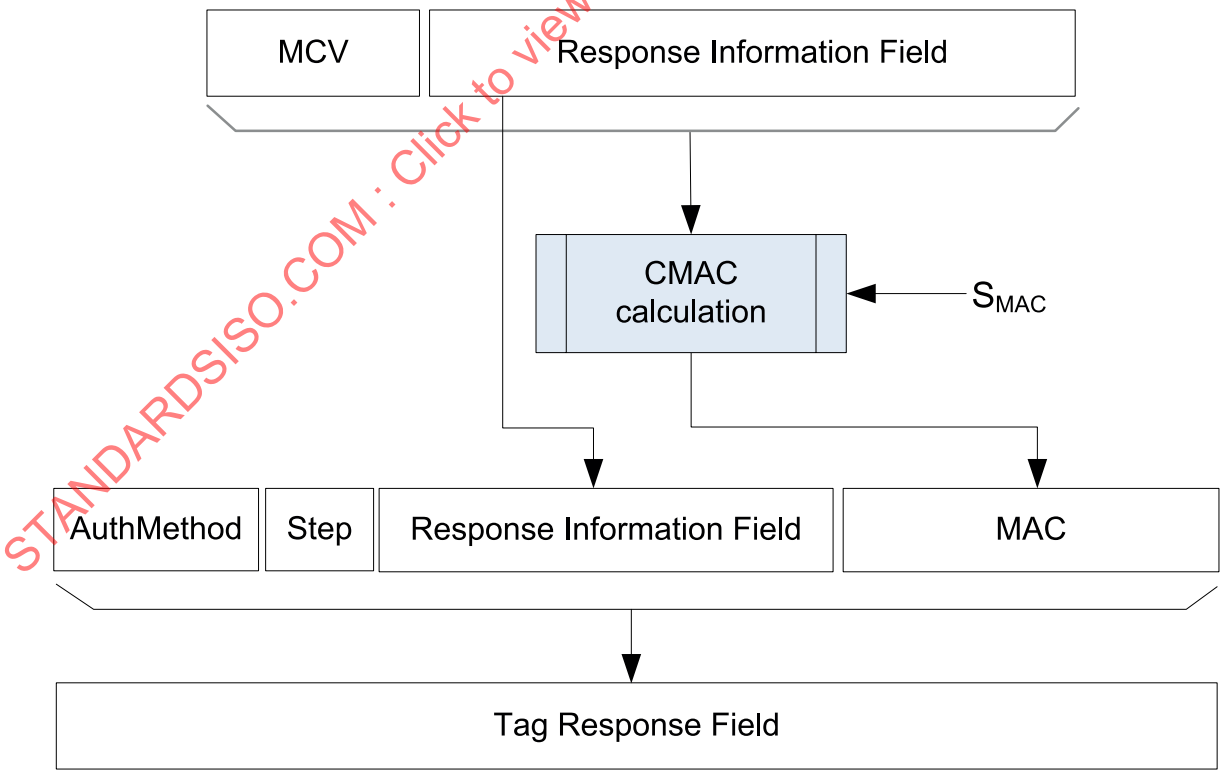


Figure 16 — Response MAC (with data)

In the case with response data (see Figure 16), the CMAC calculation includes the response data. Again, the MAC is appended to the (unsecure) message.

The final description of a secure response message (see the following Figure 17) covers the case where the response field is required to be encrypted and MACed. The encryption encompassed the response information field. Then padding has to be applied in order to match the input length required by the block cipher and to be able to unambiguously recover the original plaintext after decryption. To this end, a byte with content 80h is appended followed by enough zero byte 00h to achieve a total length which is a multiple of 16.

The padded input string is encrypted with AES in CBC mode (to be explained later), and the result is fed into the CMAC calculation. The final response which is sent to the interrogator is the concatenation of the AES-CBC output and the MAC.

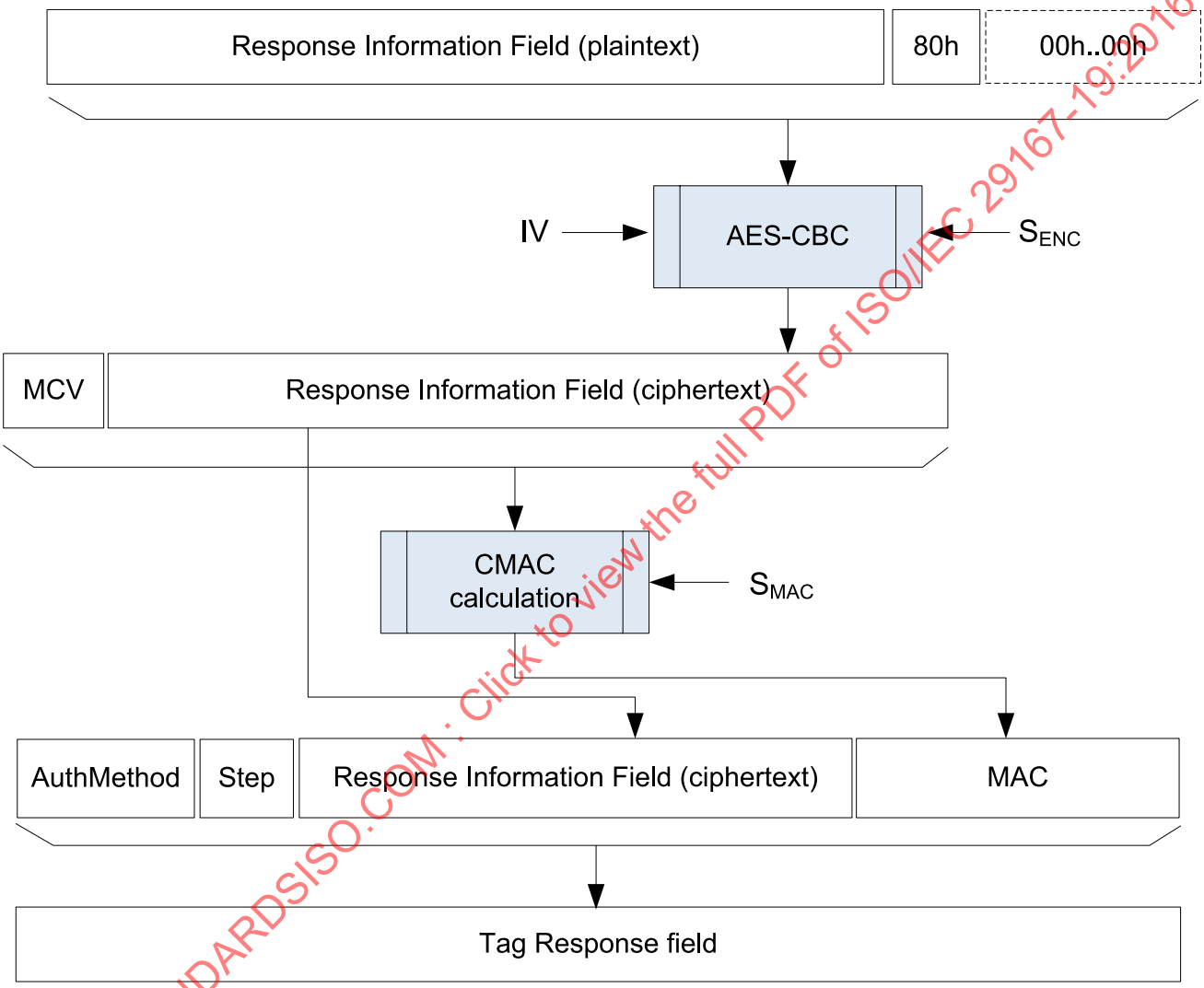
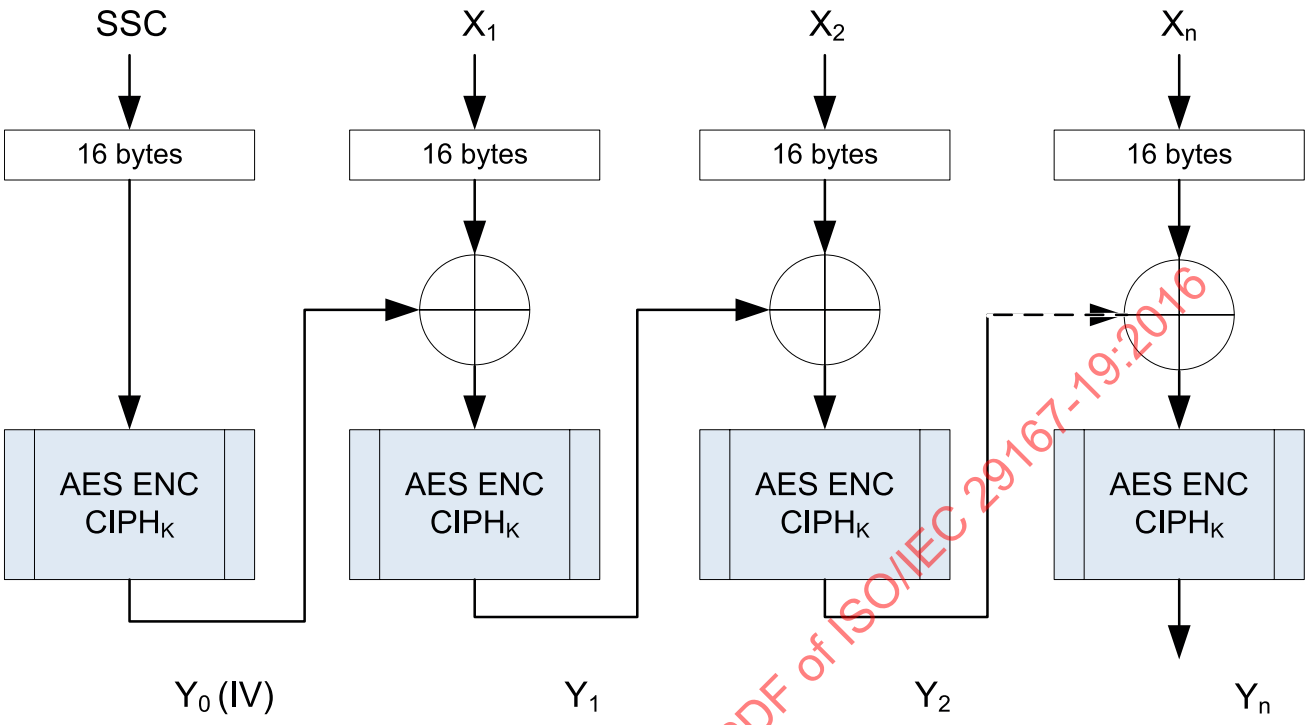


Figure 17 — *SecureComm* Response ENC (with padding) and MAC (with data)

11.4.3 Explanation of cipher block chaining mode



NOTE Xi: input plain text block, Yi: output cipher text block, $CIPH_K$: output of the encryption function of the AES under key K applied to input block.

Figure 18 — Blockwise encryption using AES in CBC-mode with SSC

Figure 18 explains the message encryption with AES in CBC mode using the send sequence counter (SSC). In the first step, SSC is encrypted to the chaining block, Y_0 (or initial chaining vector, IV). Then Y_0 is XORed with X_1, the first plaintext block. The result is encrypted to Y_1 which is the first output block, and at the same time, the next chaining vector. In the following round, $Y_2 = ENC(K, Y_1 \text{ XOR } X_2)$ is calculated, and so forth until the final block, $Y_n = ENC(K, Y_{n-1} \text{ XOR } X_n)$. Note that the key being used for encryption is always the current session key, $K = S_{ENC}$. X_n includes the padding bytes.

For message decryption, the described process has to be reversed and the *ENC* function in the chain has to be replaced with DEC. In particular, $Y_0 = ENC(K, SSC)$, $X_1 = Y_0 \text{ XOR } DEC(K, Y_1)$, …, $X_n = Y_{n-1} \text{ XOR } DEC(K, Y_n)$

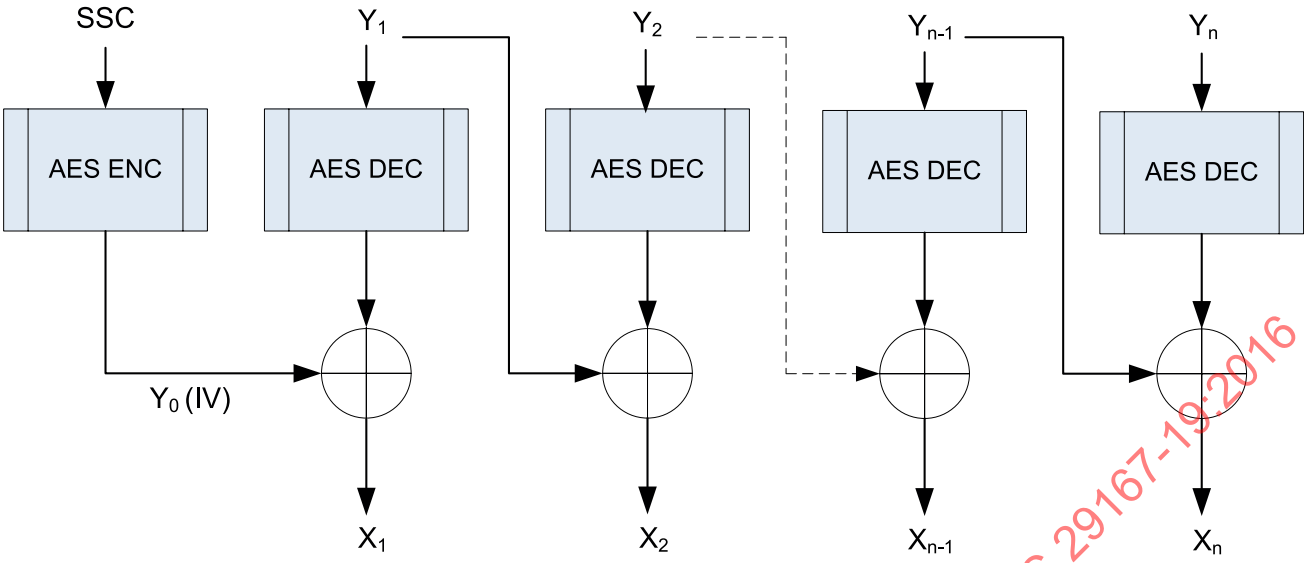


Figure 19 — Blockwise decryption using AES in CBC-mode with SSC

Annex A
(normative)

State transition tables

Table A.1 — Crypto suite state transition table for Tag identification in partial result mode

#	Start State	Command[a]	Next State	Action/Result
1	Init	*Authenticate* Step 1	TAM1.1	Successful processing of the *Authenticate* command; length of identification data is sent to Interrogator in response.
2	Init	*Authenticate* Step 1	Init	An error occurred during the processing of the *Authenticate* command or generating the response: Error code is returned.
3	Init	*Authenticate* Step 2	Init	Command not allowed in this state. Error code is returned.
4	TAM1.1	*Authenticate* Step 1	Init	An error occurred during the processing of the *Authenticate* command or generating response. Error code is returned.
5	TAM1.1	*Authenticate* Step 2	Init	An error occurred during the processing of the *Authenticate* command or generating response. Error code is returned.
6	TAM1.1	*Authenticate* Step 2	TAM1.2	Successful processing of the *Authenticate* command. Total length of identification data sent to Interrogator in response.
7	TAM1.2	*Authenticate* Step 2	TAM1.2	Successful processing of the *Authenticate* command. Fragment of identification data is sent to Interrogator in response. This is not the last fragment, otherwise #11 is valid.
8	TAM1.2	*Authenticate* Step 2	Init	An error occurred while processing the *Authenticate* command. CS transits to **Init** state; an error code is returned.
9	TAM1.1 TAM1.2, TAM1.3	*Authenticate* Step 1	TAM1.1	Reset all variables and start processing *Authenticate* command. Upon successful processing of the *Authenticate* command: transit to **TAM1.1** and send the total length of identification data to Interrogator in response.
10	TAM1.2	*Authenticate* Step 1	Init	In case of an error during command processing, the CS transits to **Init** state; an error code is returned.
11	TAM1.2	*Authenticate* Step 2	TAM1.3	In case of success: Final fragment is returned. CS transits to **TAM1.3** once the final fragment has been returned.
12	TAM1.2	*Authenticate* Step 2	Init	An error occurred while processing the *Authenticate* command. CS transits to **Init** state; an error code is returned.
13	TAM1.3	*Authenticate* Step 2	Init	Command not allowed in this state. Error code is returned. The CS transits to the **Init** state
14	Init, TAM1.1, TAM1.2, TAM1.3	Non-CS command	Init	In case any other non-crypto suite command is received by the Tag after successful Tag identification, the CS remains in or transits to state **Init.**

[a] With AuthMethod field set to 11_2 and MRead = 0000_2.

Table A.2 — Crypto suite state transition table for Tag identification in complete result mode

#	Start State	Command[a]	Next State	Action/Result
1	Init	*Authenticate* Step 1	TAM1.3	Successful processing of the *Authenticate* command; final result data are sent to Interrogator in response.
2	Init	*Authenticate* Step 1	Init	An error occurred during the processing of the *Authenticate* command or generating the response: Error code is returned.
3	Init	*Authenticate* Step 2	Init	Command not allowed in this state and result mode. Error code is returned.
4	TAM1.3	*Authenticate* Step 1	TAM1.3	Reset all variables and start processing *Authenticate* command.
5	Init, TAM1.3	Non-CS command	Init	In case any other non-crypto suite command is received by the Tag after successful Tag identification, the CS transits to or remains in state **Init.**
[a] With AuthMethod field set to 11_2 and MRead = 0000_2.				

State transition tables for Tag identification in partial or complete result mode are shown in Tables A.1 and A.2.

Table A.3 — Crypto suite state transition table for mutual authentication in partial result mode

#	Start State	Command[a]	Next State	Action/Result
1	TAM1.3	*Authenticate* Step 1	MAM1.1	Successful processing of the *Authenticate* command; length of authentication data are sent to Interrogator in response.
2	TAM1.3	*Authenticate* Step 1	Init	An error occurred during the processing of the *Authenticate* command or generating the response: Error code is returned.
3	TAM1.3	*Authenticate* Step 2	Init	Command not allowed in this state. Error code is returned.
4	MAM1.1	*Authenticate* Step 1	Init	Command not allowed in this state. Error code is returned.
5	MAM1.1	*Authenticate* Step 2	Init	An error occurred during the processing of the *Authenticate* command or generating response. Error code is returned.
6	MAM1.1	*Authenticate* Step 2	MAM1.2	Successful processing of the *Authenticate* command. First fragment of authentication data sent to Interrogator in response.
7	MAM1.2	*Authenticate* Step 2	MAM1.2	Successful processing of the *Authenticate* command. Next fragment of authentication data is sent to Interrogator in response. CS remains in **MAM1.2** as long as it has not returned the final fragment.
8	MAM1.2	*Authenticate* Step 2	Init	In case of an error: the CS transits to state **Init**; an error code is returned.
9	MAM1.2	*Authenticate* Step 1	Init	Command not allowed in this state. Error code is returned.
10	MAM1.2	*Authenticate* Step 2	SC	**Successful processing of the** *Authenticate* command: Final fragment and Remaining Length of zero is returned. CS transits to SC once the final fragment has been returned.
11	TAM1.3, MAM1.1, MAM1.2	Non-CS command	Init	In case any other non-crypto suite command is received by the Tag after successful Tag identification and before mutual authentication is completed, the CS transits to state **Init.**
12	SC	Non *SecureComm* command	Init	In case any non-*SecureComm* command is received by the Tag after successful mutual authentication, the CS transits to state **Init**.
[a] With AuthMethod field set to 01_2.				

Table A.4 — Crypto suite state transition table for mutual authentication in complete result mode

#	Start State	Command[a]	Next State	Action/Result
1	TAM1.3	*Authenticate* Step 1	SC	Successful processing of the *Authenticate* command; Final result data are sent to Interrogator in response.
2	TAM1.3	*Authenticate* Step 1	Init	An error occurred during the processing of the *Authenticate* command or generating the response: Error code is returned.
3	TAM1.3	*Authenticate* Step 2	Init	Command not allowed in this state and result mode. Error code is returned.
4	TAM1.3,	Non-CS command	Init	In case any other non-crypto suite command is received by the Tag after successful Tag identification and before mutual authentication is completed, the CS transits to state **Init.**
5	SC	Non-*SecureComm* command	Init	In case any non-*SecureComm* command is received by the Tag after successful mutual authentication, the CS transits to state **Init**.

[a] With AuthMethod field set to 01_2.

State transition tables for Mutual authentication in partial or complete result mode are shown in Table A.3 and A.4.

Annex B
(normative)

Error codes and error handling

A Tag that encounters an error during the execution of a cryptographic suite operation shall send an error reply to the Interrogator. The details of these error replies are defined in the respective air interface standards.

This Annex contains a listing of the Error Conditions that may result from the operation of this cryptographic suite. Annex E defines how to translate this error condition into an error code for the air interface.

Table B.1 — Crypto suite error codes

Crypto Suite Error Condition	Description
Other error	Miscellaneous error
Not supported	The requested functionality is not supported by this Tag or by this CS
Insufficient privileges	The interrogator did not authenticate itself with sufficient privileges for the Tag to perform the operation
Memory overrun	The command attempted to access a non-existent memory location.
Memory locked	The Tag memory location is locked an not writeable
Crypto Suite error	Cryptographic error detected. This triggers a reset

Annex C
(normative)

Cipher description

C.1 Tag Identification Cryptogram Preparation

The preparation of the RAMON authentication cryptogram comprises several distinct steps to be performed consecutively as specified in the following subclauses. The first step is the composition of the authentication record from the components as specified in Table C.1. In this composition the SID, the signature and the random filling bytes are encoded as TLV structures to facilitate the decomposition by the interrogator. If the optional signature is not present, its TLV structure has to be omitted completely. As can be seen in the Table C.2 the length of the random filling field is chosen to bring the whole record to a size of exactly 128 bytes.

Table C.1 — Components of the authentication message

	Interrogator Challenge CH_{I1}	**Tag Random Number** RN_T	**TLV-coded SID**	**TLV-coded signature**	**TLV-coded random filling**	**Zero Padding "00h"**
			"TLV record"			
# of bytes	16	16	10	2+s	2+r = 128-45-s	1
Total # of bytes	128					

TLV encoding of the individual data fields shown in Table C.2 message:

Table C.2 — TLV fields in the authentication message

	T	**L**	**V**
# of bytes	1	1	L
Description	"C1"	8	SID, conditional. At least one of "C1" or "C4" shall be present.
	"C2"	s bytes	Signature over SID, length depends on selected signature scheme and parameters, e.g. in case of using ECDSA with a 320-bit curve, s = 2*40 bytes. Optional field.
	"C4"	8	SID "EPC-coded", conditional. At least one of "C1" or "C4" shall be present. See F.1.3 for more details.
	"C8"	r bytes	Random padding data

As the number of bytes in the content field V is always less than 127 bytes, only a single byte *L* is used to code the length.

A Random filling and a final zero-byte shall be appended to the authentication message to yield a total size of 128 bytes. This length is determined by the modulus n with $2^{1024} - 1 \geq n > 2^{1016}$ in the Rabin-Montgomery scheme.

If only two bytes are left for the TLV-coded random filling, the coding shall be C8h 00h. If only one byte is left for the TLV-coded random filling, the coding shall be 00h.

C.1.1 RAMON Memory Read — Cryptogram Preparation

The RAMON encryption may be used to read out dedicated memory areas of the Tag, when Mutual Authentication and Secure Communication are not implemented on the Tag. To do so, instead of the SID, content of the Tags memory shall be transmitted to the Interrogator.

The preparation of the RAMON cryptogram comprises the same steps to be performed consecutively as specified for the Tag identification. The first step is the composition of the authentication record from the components as specified in Table C.3. In this composition the Memory Content, an optional SHA-256 Hash and the random filling bytes are encoded as TLV structures to facilitate the decomposition by the interrogator. The optional Hash value is calculated from the Memory Content. If the optional Hash is not present, its TLV structure has to be omitted completely. As can be seen in the Table C.4 the length of the random filling field is chosen to bring the whole record to a size of exactly 128 bytes.

Using the optional Hash, a maximum of 59-byte memory can be read by a single command. Not using the Hash, a maximum of 93-byte memory can be read by a single command.

Table C.3 — Components of the authentication message

	Interrogator Challenge CH_{I1}	**Tag Random Number** RNT	**TLV-coded Memory Content**	**TLV-coded Hash**	**TLV-coded random filling**	**Zero Padding "00h"**
			"TLV record"			
# of bytes	16	16	2+m	2+c	2+r = 128-37-c-m	1
Total # of bytes	128					

TLV encoding of the individual data fields shown in Table C.4:

Table C.4 — TLV fields in the memory read message

	T	**L**	**V**
# of bytes	1	1	L
Description	"CA"	m bytes	Memory
	"CC"	c bytes	SHA-256, optional field
	"C8"	r bytes	Random padding data

As the number of bytes in the content field, V, is always less than 127 bytes, only a single byte, *L*, is used to code the length.

If only two bytes are left for the TLV-coded random filling, the coding shall be C8h 00h. If only one byte is left for the TLV-coded random filling, the coding shall be 00h.

C.1.2 Additional Data fields

The RAMON encryption additionally may be used to read out sensor data and/or other dynamic information. This can be done by optionally inserting additional TLV-fields into the authentication message of either the Tag authentication or the RAMON memory read function. The coding of these additional TLV-fields is shown in Table C.5.

Table C.5 — Additional TLV fields for sensor data or other information

	T	L	V
# of bytes	1	1	L
Description	"C5"	p bytes	Session access password, optional field. This field allows providing a session access password (32 bits), to be used to transit a tag into secure state, e.g. to perform a subsequent *Untraceable* command. If this field is used, the size of the random filling shall be reduced by 2+p bytes. See F.1 for additional information.
	"CD"	d bytes	Sensor Data, optional field. If this field is used, the size of the random filling shall be reduced by 2+d bytes for each sensor data field used.
	"CE"	e bytes	General purpose, optional field. If this field is used, the size of the random filling shall be reduced by 2+e bytes for each field used.
	"CF"	f bytes	General purpose, optional field. If this field is used, the size of the random filling shall be reduced by 2+f bytes for each field used.

The usage of these additional TLV fields enables a simple mechanism to transmit sensor data or other possibly dynamic data to the Interrogator in an encrypted way. Table C.6 shows an example for an authentication message, where a single TLV-field to encode sensor data is used.

Table C.6 — Example for an authentication message with a sensor data field

	Interrogator Challenge CH_{I1}	Tag Random Number RNT	TLV-coded TID	TLV-coded Signature	TLV-coded Sensor data	TLV-coded random filling	Zero Padding "00h"'
			"TLV record"				
# of bytes	16	16	10	2+s	2+d	2+r = 128-45-s-2-d	1
Total # of bytes	128						

C.2 The MIX function

Because it interleaves static and dynamic components of the Tag ID the introduction of the MIX function reduces the risk of leaking information. The resulting 128 bytes are input into the Rabin-Montgomery encryption function that is specified in C.3. The following C fragment specifies the MIX functionality as well as padding with random data. If during the execution of MIX the *SID* data is exhausted, padding bytes are taken from the random number generator:

```
uint16_t i,  l, RNSIZE = 16;

for (i=0; i<RNSIZE; ++i) RN_T[i] = rnd08();  // generate tag random number
                                             // (precondition for MIX
                                             // function)
for (i=0; i<16; ++i) {                       // loop for permutation
                                             // rounds
  for (l=0; l<5; ++l) {                      // five bytes from TLV structure
       PERM[i*7+l] = TLV[i*5+l];
  }
  PERM[i*7+5] = CH_I[i];                     // one byte reader challenge
  PERM[i*7+6] = RN_T[i];                     // one byte tag random no.
}
for (i=16*7; i<127; ++i) {                   // add final ID bytes to PERM
     PERM[i] = TLV[i-16*(7-5)];
```

```
}

uint16_t j = 0, k = 1 ;

for (i=0; i< 127; ++i) {                    // loop for masking
     if (i mod 7 == 6 AND i < 112) {
         OUT[i] = PERM[i] ;                  // this is a part of RN_T:
                                             // take it without masking
                                             // do not increment counter j, k
     } else {
        mask = RN_T[j] ^ RN_T[k] ;           // generate mask byte from RN_T
                                             // bytes j and k
        OUT[i] = PERM[i] ^ mask ;
        k += 1 ;                             // increment counter, inner loop
        if (k == 16) {
            j += 1 ;                         // increment counter, outer loop
            k = j + 1 ;                      // avoid equal combinations
        }
     }
}
OUT[127] = 0 ;                               // zero padding
```

The output OUT[] of the MIX function is formed from a couple of components. These are:

CH_I[0..15]	random challenge CH_{I1} received from the interrogator, 16 bytes
RN_T[0..15]	random number RNT generated by the tag, 16 bytes
TLV[0..94]	TLV structure containing SID, an optional digital signature and random padding, (95 bytes). TLV(SID, Sign, RND)

All these data items are input to the MIX(*CH_I*, *RN_T*, *TLV*) function block where they are permuted and masked (XORed) with specific bytes from *RN_T*. *RN_T* itself is not masked. The permutation interleaves static and dynamic data items such as to avoid long runs of possibly known data during the multiplication which could encourage some attacks. The XOR with random data is a countermeasure against SPA/DPA attacks.

The **permutation** is defined by the following sequential procedure: take 5 bytes from *TLV,* then 1 byte from *CH_I* and then 1 byte from *RN_T*. This has to be done 16 times. Store these 16 × 7 bytes = 112 bytes into the message buffer *M*. Finally, take the remaining 15 bytes from *TLV* and store them into the massage buffer *M*.

Next, a **mask** operation is done with the resulting permutation. The **mask** is derived from the *RN_T* by XORing two different bytes. Every possible byte combination shall be used only once, therefore an indexing scheme is used as laid out in the program code and shown in Figure C.1.

The specified indexing scheme allows 120 = i(i-1)/2 combinations, where j and k are different, which is quite sufficient for 111 = 95 + 16 items. The possible combinations of j and k are illustrated in Table C.7.

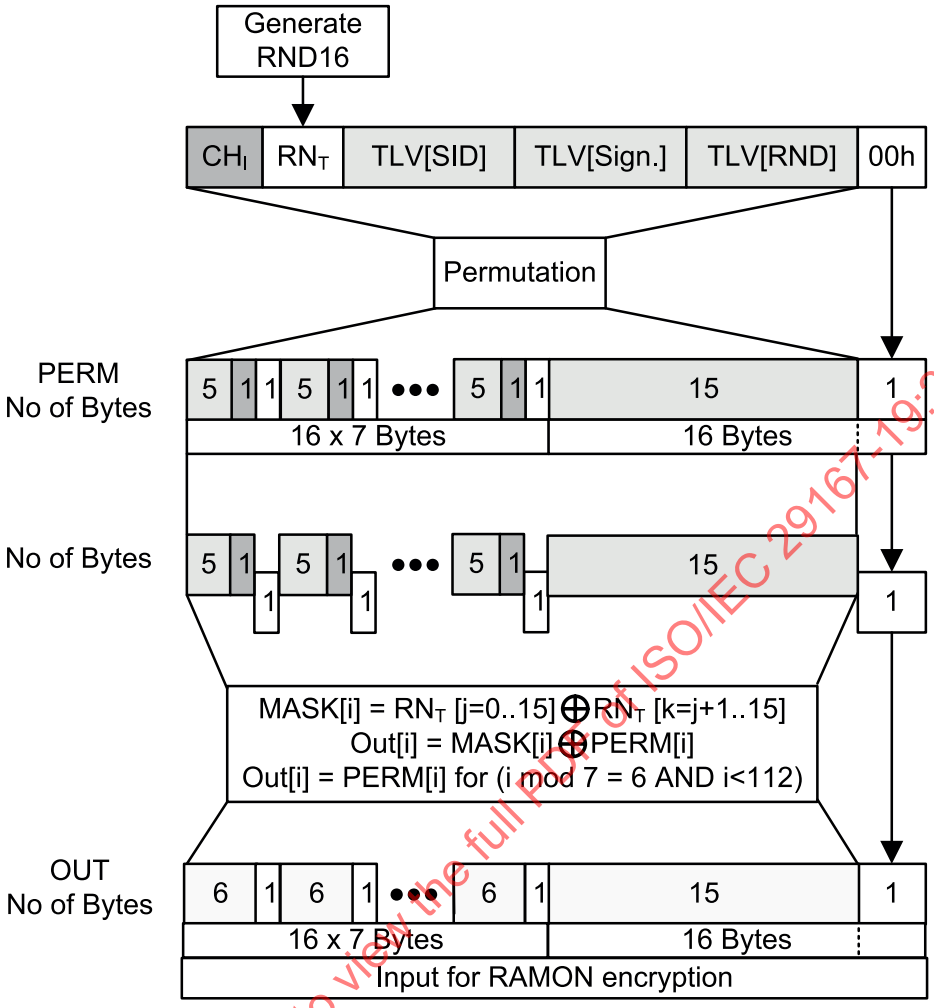


Figure C.1 — Illustration of the mix function

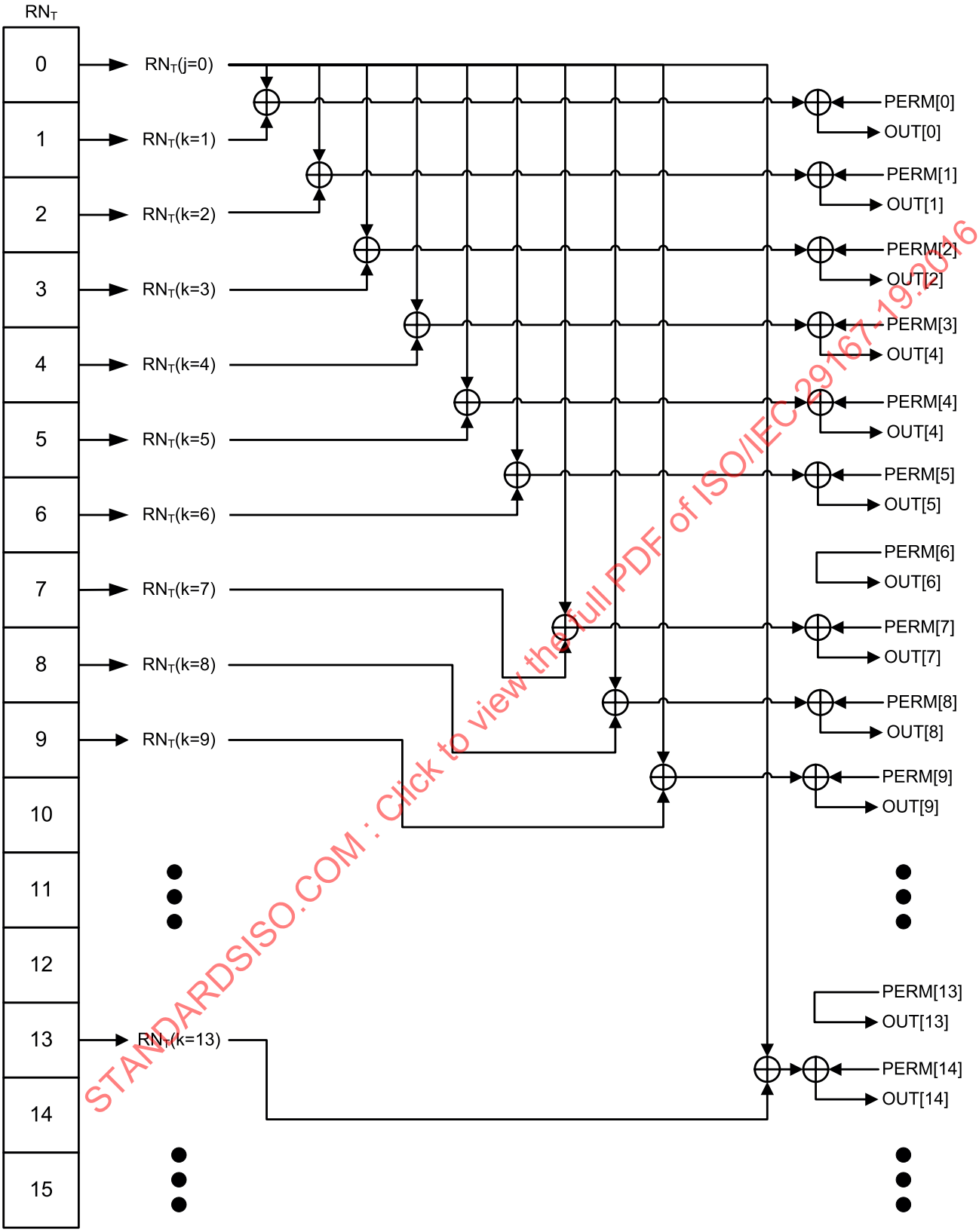


Figure C.2 — Illustration of the application of the mask function

The possible combinations of the counter values j and k are shown in Table C.7.

Table C.7 — Possible combination of j and k, applying the mask of the MIX function

j	k															
0		1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
1			2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
2				3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
3					4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
4						5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
5							6,	7,	8,	9,	10,	11,	12,	13,	14,	15
6								7,	8,	9,	10,	11,	12,	13,	14,	15
7									8,	9,	10,	11,	12,	13,	14,	15
8										9,	10,	11,	12,	13,	14,	15
...																...
13															14,	15
14																15

C.3 Rabin-Montgomery Encryption

For the following specification of the Rabin-Montgomery encryption, the public key, K_E, is replaced in the formulae by the shorthand notation n.

As described in Reference [3], the encryption of a clear text message M is calculated as

$$C = M^2 \bmod n$$

where C is the cipher text. Remember that

$$n = p \cdot q$$

where p and q are primes which satisfy the congruency condition $p \equiv q \equiv 3 \,(\mathrm{mod}4)$.

In order to optimize security, p and q should be of the same order of magnitude, $\log p \approx \log q$, and the message M should not be smaller than the modulus n. Then, the decryption as the calculation of the square root of C (mod n) is unfeasible without knowing p and q.

Taking the remainder of a long number is a computationally expensive operation; therefore, the conventional expression for calculating the modular square of M is replaced by a different method of size reduction which requires only multiplication. This method is known as Montgomery multiplication[2]. To this end, a Montgomery base R, is defined, where R is a power of 2, such that $R \geq 2^{1024}$. Base R, or rather the exponent, is public information and thus a component of the public key. With these definitions it is possible to calculate the cipher text as

$$C^* = M^2 R^{-1} \bmod n$$

The encrypted message C^* is the value sent from the Tag to the Interrogator. The clue with Montgomery multiplication is that the division by R can be implemented as a simple right shift of the result and thus, can be done at almost no computational cost. One should be aware that $C^* \neq C$, which means that the Interrogator has to undo the effect of the base R division and has to do a proper mod n reduction. However, the Interrogator is assumed to have enough computational power to do that without noticeable delay.

As indicated above, the minimum usable value for the particular choice of n is $R = 2^{1024}$. However, it is advantageous to choose a larger value, because that reduces the probability for C^* to exceed the

modulus n. With a proper choice of R, it is not necessary at all to care for the mod-Operation. The proper value of R for this cipher suite is defined in C.5.

The primes p and q used in the Rabin-Montgomery algorithm must satisfy

$$2^{(nBits-1)/2} < p, q < 2^{(nBits)/2}$$

and

$$| \log_2 p - \log_2 q | \leq 0{,}1$$

where nBits is the bit length of the public key.

For additional information refer to Reference [9], 6.4.1.2.1 and Clause 5.

C.4 Rabin-Montgomery Decryption

For the decryption of the cipher text message C^*, a modular square root has to be calculated. As a first step the effect of the Montgomery multiplication has to be unrolled by modular multiplication with the residue R:

$$C = C^*R \bmod n = (M^2R^{-1})R \bmod n$$

Then the clear text message is one of the four roots

$$M = \sqrt{C} \bmod n$$

which can be determined by means of the private exponents p and q and by application of the *Chinese Remainder Theorem*.

In order to determine which one of the four roots is the correct one the Interrogator shall check all of the four roots for correct presence of the previously sent challenge CH_{I1}. To be able to do this, it is necessary to apply the inverse of the MIX function to each of the four roots to get the original plaintext P.

In order to guarantee the security of the Rabin-Montgomery algorithm, the decryption procedure in the Interrogator is not allowed to output any plaintext data corresponding to a root $M = \sqrt{C} \bmod n$ in which the previously sent challenge *CH_I* is not present. It shall delete any such erroneous roots (and also the corresponding plaintexts) from its internal memory before returning to the calling program. If the challenge *CH_I* is not present in any of the four roots, the decryption procedure shall not output any plaintext data at all.

C.5 Definition of the Montgomery Residue

The residue R used in Montgomery multiplication shall be 2^{1088} .

C.6 The inverse MIX Function MIX^{-1}

To get back the original plaintext P finally, the Interrogator has to apply an inverse MIX function MIX^{-1} after encryption of the received message:

```
uint16_t i, RNSIZE = 16;

for (i=0; i<RNSIZE; ++i) RN_T[i] = OUT[7*i+6];// read tag random number from
                            // decrypted message (OUT[] from MIX function)
                            // (precondition for DEMIX function)

uint16_t i_ch = 0, i_tlv = 0, j = 0, k = 1 ;
```

```
  // loop for unmasking (= XOR, using the same mask as used for MIX)
for (i=0; i< 127; ++i) {
                          // do not process the last byte (zero padding byte)
     if (not (i mod 7 == 6 AND i < 112)) {        // skip parts of RN_T
        // generate mask byte from RN_T bytes j and k:
        mask = RN_T[j] ^ RN_T[k] ;
        if ( i mod 7 == 5 AND i < 112) {
                                                  // this is part of the challenge
            CH_I[i_ch] = OUT[i] ^ mask ;
            i_ch += 1 ;
          } else {
                                                  // this is part of TLV
            TLV[i_tlv] = OUT[i] ^ mask ;
            i_tlv += 1 ;
          }

          k += 1 ;       // increment k, j
          if (k == 16) {
              j += 1 ;
              k = j + 1 ;
          }
        }
    }
PADBYTE = OUT[127];
```

The output OUT[] of the inverse MIX function is formed from a couple of components. These are:

CH_I[0..15]	random challenge CH_{I1} received from the interrogator, 16 bytes
RN_T[0..15]	random number RNT generated by the tag, 16 bytes
TLV[0..94]	TLV structure containing SID, an optional digital signature and random padding, (95 bytes). TLV(SID, Sign, RND)

C.7 Padding for Symmetric Encryption

The Command Data of an *Authentication* command with AuthMethod 1 Step 3 (*Secure Communication* command) are transmitted encrypted if indicated in the SCFlags field of this command. The Response Information Field of an Authentication response with AuthMethod 1 Step 3 (Secure Communication response) is transmitted encrypted if indicated in the SCFlags field of the related Authentication command. In these cases, the plaintext to be encrypted shall be padded prior to encryption, using the method specified in this subclause.

After applying this padding method, the input to the encryption is a single or are multiple complete data blocks. Padding shall be applied, even if the total number of bits in the plaintext is already a multiple of the block size of the symmetric cryptographic algorithm (e.g. 128 bits for AES).

Padding is applied as follows:

— Append 80h to the plaintext data block.

— If the resulting block length is a multiple of the algorithm block size (or equals the block size), no further padding is required. If not:

— Append as many zero bytes as necessary to complete the final block (to a length multiple of the algorithm block size).

C.8 Coding Examples

C.8.1 RAMON Encryption

The following C- code is an example of a RAMON encryption./ /

```
// Definitions:
//
//   - the notation is similar to C / Java
//   - Montgomery bit length r:
//     a fixed system parameter (Residue), here: r = 1088
//   - variable n:
//     n is the public key, (known as Ke)
//   - variable x:
//     x is the plaintext to be encrypted
//     (the result of the MIX function)
//   - variable y:
//     y is the ciphertext for the plaintext x
//     (where the least r bits have to be skipped since they are zero)
//   - Function is_odd(y):
//     returns 'True' for odd y && 'False' for even y
//
// Algorithm description:
//
//   Scope:
//     Given a plaintext x && an odd public key n with
//
//         0 <= x < n  <<  2**r
//
//     find an integer y >= 0 which fulfils the equation
//
//         x * x + z * n = y * 2**r
//
//     (for some integer z), such that y < n holds with a
//     very high probability. Here '**' means exponentiation.
//
//     Then y is the cipher text for the plaintext x.
//
//   Step I:
//     "calculate the square y of the plaintext x"
//      y = x * x;
//
//   Step II:
//      "r-Bit Montgomery reduction of y modulo n"
//      for (i = 0; i < r; i = i + 1) {
//          if (is_odd(y)) {
//              y = y + n;
//          }
//          y = y / 2
//      }
//      return y
//
// Sample values:
//
// - r = 1088
//
// - public key n:
//   0xbb24343b4b3e006ce1fa33383e2304081f5c62a3
//     67466e3e987e3717f626b5b40fb9d910a82f595
//     be9b4e281aca0bf80449fc4d3e7a5e35f5665654
//     6c9d47e000000000000000000000000000000000
//     0000000000000000000000000000000000000000
//     0000000000000000000000000000000000000000
//     0000000000000001
//
// - plaintext x:
//   0x7411d164dbd5ab10302a6c8bb17f65280c3cc589
//     f647d653326ca3cb440709884a0ed86d0c8f5309
//     8f1f0068683c3c3c3c3c76763c3c3c3c3ccbcb3c
//     3c3c3c3ccbcb2938342820939365727574617070
//     6e676973200a0a4343453e3e08083e3e3e3e3e1e
//     1e3e3e3e3e3e66663e3e3e30538d8d3c38444967
//     414161743e0849
//
// - expected ciphertext y:
//   0x2203be703fe6f7f23ac51cfadd12b69719a5b02d
//     de2d90b1ca58adf294f93abd98963c12b7e47279
```

```
//      a1e1dfafe1cee6d1b6dc84248aa9f21ba90a3296
//      78c8195cd5359757302e3e35637c684a3e15465b
//      70e3b3635a6cd9c52a48a206eef2471aa121ba50
//      f2eb9d0a9a6eb621163402dcbd0dbcb292d56736
//      3389caa28fceb4d2

//
// Sample in C
//
// Important remark:
//      This code has been written for a better
//      understanding of the Rabin-Montgomery encryption.
//      It is not optimized for run time || memory space.
//
//    - for each long integer x the equation x[i] defines bit number i
//
//
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

uint16_t r = 1088 ;

uint16_t n[1024] ;
uint16_t x[1024] ;
uint16_t y[4096] ;

//
// converts a hex character (0..9,'a'..'f') to a binary value
uint16_t bin_value(char ch) {
   if('0'<=ch && ch<='9')
   {
      return ch - '0';
   }
   else if('a'<=ch && ch<='z')
   {
      return ch - 'a' + 0x0A;
   }
   return 0 ;
}

// set given binary stream to zero
void setzero (uint16_t * bin, uint16_t size) {
   int i ;
   for (i=0; i < size; i++) { bin[i] = 0 ; }
}

// converts a hex string into a string of bits (with LSB = bin[0])
// returns the number of bits
// after converting the function pads the bit stream with zero bits
// until number of bits == size
uint16_t hex2bin (char * hex, uint16_t * bin, uint16_t size) {
   uint16_t len = 0 ;
   char * start = hex ;
   while (*hex != '\0') { hex++ ; } hex-- ;
   while (hex >= start) {
      uint16_t i ;
      uint16_t nibble = bin_value(*hex) ;
      for (i=0; i<4;i++) {
         uint16_t bit = 0 ;
         uint16_t mask = 0x01 << i ;
         if ((nibble&mask) != 0) { bit = 1 ; }
         *bin++ = bit ;
         len+= 1 ;
      }
      hex -- ;
   }
   while(size > len) {
      *bin++ = 0 ;
      len += 1 ;
   }
```

```
   return (len) ;
}

// prints the string of bits as hex
void printbinashex(uint16_t len, uint16_t * bin) {
   while (len > 0) {
      uint16_t nibble = 0 ;
      uint16_t i = 0 ;
      for (i=0; i<4;i++) {
         uint16_t bit = bin[len-4+i] ;
         uint16_t mask = 0x01 << i ;
         nibble += bit * mask ;
      }
      printf("%x",nibble);
      len -= 4;
   }
   printf("\n");
}

// add given bit to binary stream bin
// starting at given position pos
void addbit (uint16_t pos, uint16_t * bin, uint16_t bit) {
   while (bit != 0) {
      uint16_t newbit = bit + bin[pos] ;
      uint16_t carry = 0 ;
      if (newbit > 1) {
         newbit = 0 ;
         carry = 1 ;
      }
      bin[pos] = newbit ;
      pos ++ ;
      bit = carry ;
   }
}

// add x1 to x2 (starting with x2[pos])
// ( equivalent to x2 + x1 * (2 ** pos) )
void add (uint16_t pos, uint16_t * x2, uint16_t * x1, uint16_t size_x1) {
   int i ;
   for (i = 0; i < size_x1; i++) {
      addbit(pos+i,x2,x1[i]);
   }
}

// calculates the square (xq = x1 * x1, where x1 is input && xq is output)
void square (uint16_t * xq, uint16_t * x1, uint16_t size_x1) {
   uint16_t i,j ;
   for (i=0 ; i < size_x1 ; i++) {
      for (j=0 ; j < size_x1 ; j++) {
         uint16_t ij = i + j ;
         uint16_t xij = x1[i] * x1[j] ;
         addbit (ij, xq, xij) ;
      }
   }
}


void run_sample () {
   // set public key n
   hex2bin ("bb24343b439e006ce1fa33383e2304081f5c62a367466e3a9387e3717f626b5b
40fb9d910a82f595be9b4c281aca0bf80449fc4d3e7a5e35f56656546c9d47e0
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001",
n,1024) ;

   // set plain text x
   hex2bin ("7411d164dbd5ab10302a6c8bb17f65280c3cc589f647d653326ca3cb44070988
4a0ed86d0c8f53098f1f0068683c3c3c3c3c76763c3c3c3c3ccbcb3c3c3c3c3c
cbcb2938342820939365727574617070 6e676973200a0a4343453e3e08083e3e
3e3e3e1e1e3e3e3e3e3e66663e3e3e30538d8d3c38444967414161743e0849",
x,1024) ;
```

```
   // calculate y = x * x:
   setzero(y,4096);
   square(y,x,1024);

   // run Montgommery reduction
   int i ;
   for (i=0 ; i < r; i++) {
      if (y[i] != 0) {
         add (i,y,n,1024) ;
      }
   }

   // print encrypted message (trailing zeros not yet skipped)
   printbinashex(4096,y);
}

void main() {
   run_sample() ;
}
```

C.8.2 RAMON Decryption

The following Java- code is an example of a RAMON decryption.

```
//
// sample for Rabin-Montgommery decryption
//

// the sample program requires library java.math.BigInteger
import java.math.*;

class Ramondecrypt{
   static int residue ;       // montgommery residue
   static BigInteger n ;      // the public key (also known as Ke
   static BigInteger p ;            // prime factor of key n
   static BigInteger q ;            // prime factor of key n
                // p and q are the secret keys
   static BigInteger expplain ;    // expected plain message
   static BigInteger crypt ;  // given encrypted message
   static BigInteger euclid_p ;  // euclidian coeeficient
   static BigInteger euclid_q ;  // euclidian coeeficient
                // euclid_p * p + euclid_q * q == 1

   // set sample values by using test vectors of ISO 29167-19
   static void setsample_iso ()
   {
      expplain = new BigInteger
("7411d164dbd5a010302a6c8bb17f65280c3cc589f647d653326ca3cb440709884a0ed86d0c8f530
98f1f0068683c3c3c3c3c76763c3c3c3c3ccbcb3c3c3c3c3ccbcb2938342820939365727574617070
6e676973200a0a4343453e3e08083e3e3e3e3e1e1e3e3e3e3e3e66663e3e3e30538d8d3c384449674
14161743e0849",16) ;

      p = new BigInteger
("c868f88a83d8e9689a44ad154b29d8b6048e5f55cdd9fe5287899ff174168a324e682c127e35118
736af6898e3b62a8a58ed623e409991fb4925056c6a401e57",16) ;

      q = new BigInteger
("ef0d08c4c672f46edc80b908d3e15cea1089d46f90a36a333d22ea59038dcb7cc3c9fad18228da3
710bb633ed5a076224a17e64104631b2ace6a2b4acc05ad67",16) ;

      residue = 1088 ;

      crypt = new BigInteger
("2203be703fe6f7f23ac51cfadd12b69719a5b02dde2d90b1ca58adf294f93abd98963c12b7e4727
9a1e1dfafe1cee6d1b6dc84248aa9f21ba90a329678c8195cd5359757302e3e35637c684a3e15465b
70e3b3635a6cd9c52a48a206eef2471aa121ba50f2eb9d0a9a6eb621163402dcbd0dbcb292d567363
389caa28fceb4d2",16) ;
       }

   // set simple sample values which allows checking via manual computation
```

```
static void setsample_simple ()
{
   expplain = new BigInteger ("9",16) ;
   p = new BigInteger ("13",16) ;
   q = new BigInteger ("0b",16) ;
   residue = 9 ;
   crypt = new BigInteger ("34",16) ;
}

// unroll the effect of the montgommery multiplication
// msg -> ( msg * 2 ^ residue ) mod n
static BigInteger reduceMontgommery (BigInteger msg)
{
   BigInteger msg_expand = msg.shiftLeft (residue) ;
   return (msg_expand.mod (n)) ;
}

// calculate euclid_p and euclid_q from p and q
// by using the extended euclidian algorithm
static void extendedEuclid ()
{
   BigInteger s = BigInteger.ZERO ;
   BigInteger t = BigInteger.ONE ;
   BigInteger r = p ;
   BigInteger old_s = BigInteger.ONE ;
   BigInteger old_t = BigInteger.ZERO ;
   BigInteger old_r = q ;
   while (! r.equals (BigInteger.ZERO)) {
      BigInteger quotient = old_r.divide (r) ;
      BigInteger x ;
         // (old_r, r) := (r, old_r - quotient *r)
      x = r ;
      r = old_r.subtract(quotient.multiply (r)) ;
      old_r = x ;
          //(old_s, s) := (s, old_s - quotient *s)
      x = s ;
      s = old_s.subtract(quotient.multiply (s)) ;
      old_s = x ;
          // (old_t, t) := (t, old_t - quotient *t)
      x = t ;
      t = old_t.subtract(quotient.multiply (t)) ;
      old_t = x ;
   }
   euclid_p = old_t ;
   euclid_q = old_s ;
}

// calculates the root mod n_part (where n_part is q or p)
// this works only for values of p and q which require the equations
// n_part mod 4 == 3
// since this case allows to compute square roots by
// root = (msg ^ ((n_part + 1) / 4)) mod n_part
static BigInteger getRoot (BigInteger msg, BigInteger n_part)
{
   BigInteger exp = n_part ;
   exp = exp.add (BigInteger.ONE) ;
   BigInteger four = new BigInteger ("4") ;
   exp = exp.divide (four) ;
   BigInteger root = msg.modPow (exp,n_part) ;
   return (root) ;
}

// returns ypq * pq * mpq
static BigInteger makeCRTterm (BigInteger ypq, BigInteger pq, BigInteger mpq)
{
   BigInteger help = ypq.multiply (pq) ;
   return (help.multiply (mpq)) ;
}

// decrypt by calculating the four roots as defined
// for Rabin cryptosystem :
```

```
   //  root[0] = ( euclid_p * p * mq + euclid_q * q * mp ) mod n
   //  root[1] = n - root[0]
   //  root[2] = ( euclid_p * p * mq - euclid_q * q * mp ) mod n
   //  root[3] = n - root[2]
   //  where
   //   mp = = (msg ^ ((p + 1) / 4)) mod p
   //   mq = = (msg ^ ((q + 1) / 4)) mod q
   static void calculateRoots (BigInteger msg)
   {
      BigInteger mp = getRoot (msg,p) ;
      BigInteger mq = getRoot (msg,q) ;

      BigInteger term1 = makeCRTterm (euclid_p,p,mq) ;
      BigInteger term2 = makeCRTterm (euclid_q,q,mp) ;

      BigInteger roots[] = new BigInteger[4] ;
      // first root = (term1 + term2) mod n
      BigInteger root = term1.add (term2) ;
      roots[0] = root.mod (n) ;
      // second root = n - first root
      roots[1] = n.subtract (roots[0]) ;
      // third root = (term1 - term2) mod n
      root = term1.subtract (term2) ;
      roots[2] = root.mod (n) ;
      // fourth root = n - third root
      roots[3] = n.subtract (roots[2]) ;

// Warning!
   // The code in the following section is for demonstration only.
   // Annex C.4 requires that a life system must not output any root that
   // does not contain the previously sent challenge CHI1 after running the
   // De-Mix-function. Also, a life system must clean up all internal buffers
   // containing any data related to a root.

      // show all roots :
      System.out.println("One of the following results is our plaintext. Run De-Mix-
function and check for previously sent challenge Chi1 to determine the correct one.");
      for ( BigInteger actroot : roots ) {
         System.out.println( "root = " + actroot.toString(16));
      }
   }

   // run the sample with the given test vectors
   static void runsample ()
   {
      // calculate n = p * q
      n = p.multiply (q) ;
      System.out.println( "Key Ke = n = " + n.toString(16) );

      // apply extended euclidian algorithm to find euclid_p and euclid_q
      // such as euclid_p * p + euclid_q * q == 1
      extendedEuclid () ;

      // unroll the effect of the montgommery multiplication
      BigInteger msg = reduceMontgommery (crypt) ;

      // decrypt by calculating the four roots as defined
      // for Rabin cryptosystem
      calculateRoots (msg)  ;
   }

   // set the test vectors and run the sample
   static public void main( String[] args )
   {
      // setsample_simple () ;
      setsample_iso () ;
      runsample () ;
   }
}
```

Annex D
(informative)

Test Vectors

This Annex contains a test example for the RAMON encryption.

D.1 Notation

In most cases, the messages exchanges, as well as individual message components, are considered as a string of octets or bytes which are presented here in hexadecimal notation. Each byte comprises exactly two hexadecimal digits 0,...,a,...,f. Theses bytes are written from left to right, beginning with the first byte of a message on the **left** and proceeding with the remaining bytes to the **right**. This is the default notation.

As the RAMON cryptosystem operates on long integer numbers, it is sometimes more convenient to interpret the messages as unsigned hexadecimal numbers. These long integers are always preceded with the prefix **0x**. With this notation, endianess is important: we interpret messages as being stored **big** endian, i.e. the first byte of a message is the **most** significant byte of the equivalent integer. This has the effect that a message in numeric notation seems byte reversed as compared to string notation. This difference has to be remembered when following the test example.

D.2 RAMON Keys

Although we do not need the secret keys for the RAMON encryption we give here the prime factors p and q which satisfy the congruency condition $p \equiv q \equiv 3 \pmod 4$ for the modulus (in numeric notation):

```
P = 0xc868f88a83d8e9689a44ad154b29d8b6048e5f55cdd9fe5287899ff174168a32
    4e682c127e35118736af6898e3b62a8c58ed623e409991fb4925056c6a401e57

q = 0xef0d08c4c672f46edc80b908d3e15cea1089d46f90a36a333d22ea59038dcb7c
    c3c9fad18228da3710bb633ed5a076224a17e64104631b2ace6a2b4acc05ad67
```

Now, the modulus n or the public key is given as the product $K_E = n = p \cdot q$

```
KE = n = 0xbb24343b439e006ce1fa33383e2304081f5c62a367466e3a9387e3717f6
         26b5b40fb9d910a82f595be9b4c281aca0bf80449fc4d3e7a5e35f566565
         46c9d47e0000000000000000000000000000000000000000000000000000
         0000000000000000000000000000000000000000000000000000000000000
         0000000000000001
```

D.3 The Authentication Message

This Clause specifies the components from which the authentication message is formed. Here the items are given mostly in message format. The numeric format will be shown in the next Clause. The components are the reader random challenge CH_{I1}, the tag random number RN_T, and the tag TLV record. The content of the sample tag TLV record has no significance; it is merely considered as a test string.

```
CHI1 = c24c6f86f4a4c11e0022bde0b9f22fd7

RNT  = a770a37ab8afd42a0a4a0e1f8d2c1ac1

TLV record = c108878424da7e3b9b44c2502f720d9421e7933702a184c4c8d2d83d9
             5b6a76b34ebe1fa80a8a224a8726e264ee23bc0996c9ac9a30f48a00c
             261256e1e43a4e80ffba17bac4008e9db5d0fde9669c181963d04549e
             ba2d7e7acd7c7c801ab
```

The next step is concatenating these three values and adding the zero padding to generate the complete authentication message:

```
Authentication Message = c24c6f86f4a4c11e0022bde0b9f22fd7 // CHI
                         a770a37ab8afd42a0a4a0e1f8d2c1ac1 // RNT
                         c108878424da7e3b9b44c2502f720d94 // TLV: STID, Sign.
                         21e7933702a184c4c8d2d83d95b6a76b
                         34ebe1fa80a8a224a8726e264ee23bc0
                         996c9ac9a30f48a00c261256e1e43a4e
                         80ffba17bac4008e9db5d0fde9669c18
                         1963d04549eba2d7e7acd7c7c801ab00 // TLV: Random fill, 00
```

The Authentication Message is entered next into the MIX function yielding the message M.

```
M = 160c5a9b2cb1a757d3d632fc667049ed49a107a7a34b85bde90df87a6d5cd8ae
    792db8c9d44a1c1f4daf0ad71a6458a3d4385506f2542e2adc1799702ebb0af5
    57522b9e944a3dfc37ad31c60e25a9c3b3e6c21f625154b05e278d25714e420a
    e72c20eeb98077291acd0226980d50c13f731b011c2cc4876cbd54e5dcce3900
```

This message block has the required size of 128 bytes for the RAMON encryption with a modulus of 1024 bits.

D.4 RAMON Encryption

The plaintext authentication message shown above, is now re-written in integer numeric notation as

```
M = 0x0039cedce554bd6c87c42c1c011b733fc1500d982602cd1a297780b9ee202ce7
    0a424e71258d275eb05451621fc2e6b3c3a9250ec631ad37fc3d4a949e2b5257
    f50abb2e709917dc2a2e54f2065538d4a358641ad70aaf4d1f1c4ad4c9b82d79
    aed85c6d7af80de9bd854ba3a707a149ed497066fc32d6d357a7b12c9b5a0c16
```

In this representation occurs a leading zero byte which may be suppressed by some software. According to the RAMON specification the expression $C^* = M^2R^{-1} \bmod n$ is calculated, with $R = 2^{1088}$, yielding

```
C* = 0x550dd862e4bf04b82bbd929938c7a0a255a598464036bc677f70a903d34d1637
     ffafd3ed7e45fd726792ee057349d5fa3722fe3a8ae8235243dab5d05d451c41
     e274af84964d197054cbd3e9c1045ae867ee0b3ffa09826f4e50d228587e77f6
     d0a8db1b7a561f1f58ac0d4dc3ef252cd2a9df39a90d0c7ff1ae44ee9b9eac93
```

The encrypted message C^* is then sent to the interrogator, LSB first, as the following byte stream

```
TX(C*) = 93ac9e9bee44acf17f0c0da939dfa9d22c25cfc34d0dac581f1f567a1bdba
         8d0f6777e5828d2504e6f8209fa3f0bee67e85a01c1e9d3cb5470194d9684a
         f74e2411c455dd0b5da435223e88a3afe2237fad5497305ee926772fd457ee
         dd3afff37164dd303a9707f67bc36404698a555a2a0c7389992bd2bb804bfe
         462d80d55
```

D.5 Montgomery Reduction and Decryption

The first step in decrypting the received RAMON message is the Montgomery Reduction, i.e. the modular multiplication with the residue $R = 2^{1088} = 16^{272}$. As a consequence of the specific choice of R this is simply the addition of 272 hexadecimal zeroes to the least significant end of the numeric representation. Then it follows a modular reduction mod n which results in the "true" ciphertext $C = M^2 \bmod n$ according to the Rabin cryptosystem:

```
C = 0xac4a30613b1ec7e6d578f960ed8dfe20ddbd3392ecd93c007bd27176434142cb
    435bcc2a82721e0a3654dc6bdf20e62097d56317f4c07a82520e00da6b818a05
    7d64cfb43dabeb37aa62623b68b253cd39c2ba7f81efbea9b6cc944577779da9
    ec3ad3731aa73992c940bcb7e23b846850052eaef2da6b83c40f0624a96278f1
```

Calculation of the roots involves the Chinese Remainder Theorem and the extended Euclidian algorithm which are well known in Number Theory. Resulting are four roots as follows:

```
+r = 0x14b9d27f4571fe5fdb982ea27709aab7d7cfa378c296b3f9f08a7c904b2bbf92
     e0a3189628ff6d13ce7ce588415f2e9ac625dd2a81db2d64231145d3c66a9aeb
     c594cc8b92b649b31253abe2942472cadd0ec627d5f3f2bbd55c7ccbf184808b
```