INTERNATIONAL STANDARD

ISO/IEC 29167-19

Second edition
2019-06

Information technology — Automatic identification and data capture techniques —

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

Technologies de l'information — Techniques automatiques d'identification et de capture de données —

Partie 19: Services de sécurité par suite cryptographique RAMON pour communications par interface radio

Reference number
ISO/IEC 29167-19:2019(E)

COPYRIGHT PROTECTED DOCUMENT

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

Published in Switzerland

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.

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) or the IEC list of patent declarations received (see http://patents.iec.ch).

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

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 31, *Automatic identification and data capture*.

This second edition cancels and replaces the first edition (ISO/IEC 29167-19:2016), which has been technically revised.

The main changes compared to the previous edition are as follows:

— It was thought that the fixed RAMON key length (KE) of 1 024 bits for tag authentication, defined in the first edition of this document, would maybe not be sufficient for all future uses. The method proposed in this edition allows extending the length of the cryptographic RAMON key by discrete steps of 128 bits. Beyond the previously defined key length of 1 024 bits, key lengths of 1 152, 1 280, 1 408, 1 536, 1 664 bits and beyond become feasible. The current method does not limit the possible key length in any way. The key length only is limited by the ability to send the cryptographic authentication response, which is of equal length to the cryptographic key, back to the interrogator. Allowing extended key length makes sure that the RAMON encryption is future-proofed and security can be improved as needed.

— To support different key lengths in a generic approach, the mix function has been revised.

— To improve the readability and consistency of this document, the specification of the cipher and the description of the TLV record have been separated into independent subclauses.

— A new TLV-Structure, supporting data identifiers according to ASC MH 10, was added.

A list of all parts in the ISO 29167 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

This document 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

Impinj, Inc.

400 Fairview Ave N, # 1200
Seattle, WA 98109
USA

Giesecke & Devrient GmbH

Prinzregentenstrasse 159
D-81607 Munich
Germany

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Information technology — Automatic identification and data capture techniques —

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

1 Scope

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

This document 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 document defines various authentication methods and methods of use for the cipher. A Tag and an Interrogator can support one, a subset, or all of the specified options, clearly stating what is supported.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 8825-1, *Information technology — ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) — Part 1*

ISO/IEC 15962:2013, *Information technology — Radio frequency identification (RFID) for item management — Data protocol: data encoding rules and logical memory functions*

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 18000-63:2015, *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, *Information technology — Automatic identification and data capture (AIDC) techniques — Harmonized vocabulary*

3 Terms and definitions

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

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
authentication
service that is used to establish the origin of information

3.2
confidentiality
property whereby information is not disclosed to unauthorized parties

3.3
integrity
property whereby data have not been altered in an unauthorized manner since they were created, transmitted or stored

3.4
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

3.5
secure communication
communication between the tag and the interrogator by use of the *Authenticate* command, assuring authenticity, *integrity* (3.3) and *confidentiality* (3.2) of exchanged messages

4 Conformance

4.1 Claiming conformance

An Interrogator or Tag shall comply with all relevant clauses of this document, except those marked as "optional".

4.2 Interrogator conformance and obligations

An Interrogator shall implement the mandatory commands defined in this document and conform to ISO/IEC 18000-3, ISO/IEC 18000-4 or ISO/IEC 18000-63, as relevant.

An Interrogator may implement any subset of the optional commands defined in this document.

The Interrogator shall not

— implement any command that conflicts with this document, or

— require the use of an optional, proprietary or custom command to meet the requirements of this document.

4.3 Tag conformance and obligations

A Tag shall implement the mandatory commands defined in this document for the supported types and conform to ISO/IEC 18000-3, ISO/IEC 18000-4 or ISO/IEC 18000-63, as relevant

A Tag may implement any subset of the optional commands defined in this document.

A Tag shall not

— implement any command that conflicts with this document, or

— require the use of an optional, proprietary or custom command to meet the requirements of this document.

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	Bit length of public RAMON key K_E and private key K_D k shall be divisible by 128 and ≥1 024.
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 K_E will be used)
KSel	Key select (determines which pair of K_{ENC}, K_{MAC} 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	Random number generated by the tag, 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 document uses the notation of ISO/IEC 18000-63.

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

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 [9].
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 byte. 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 "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 can 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.

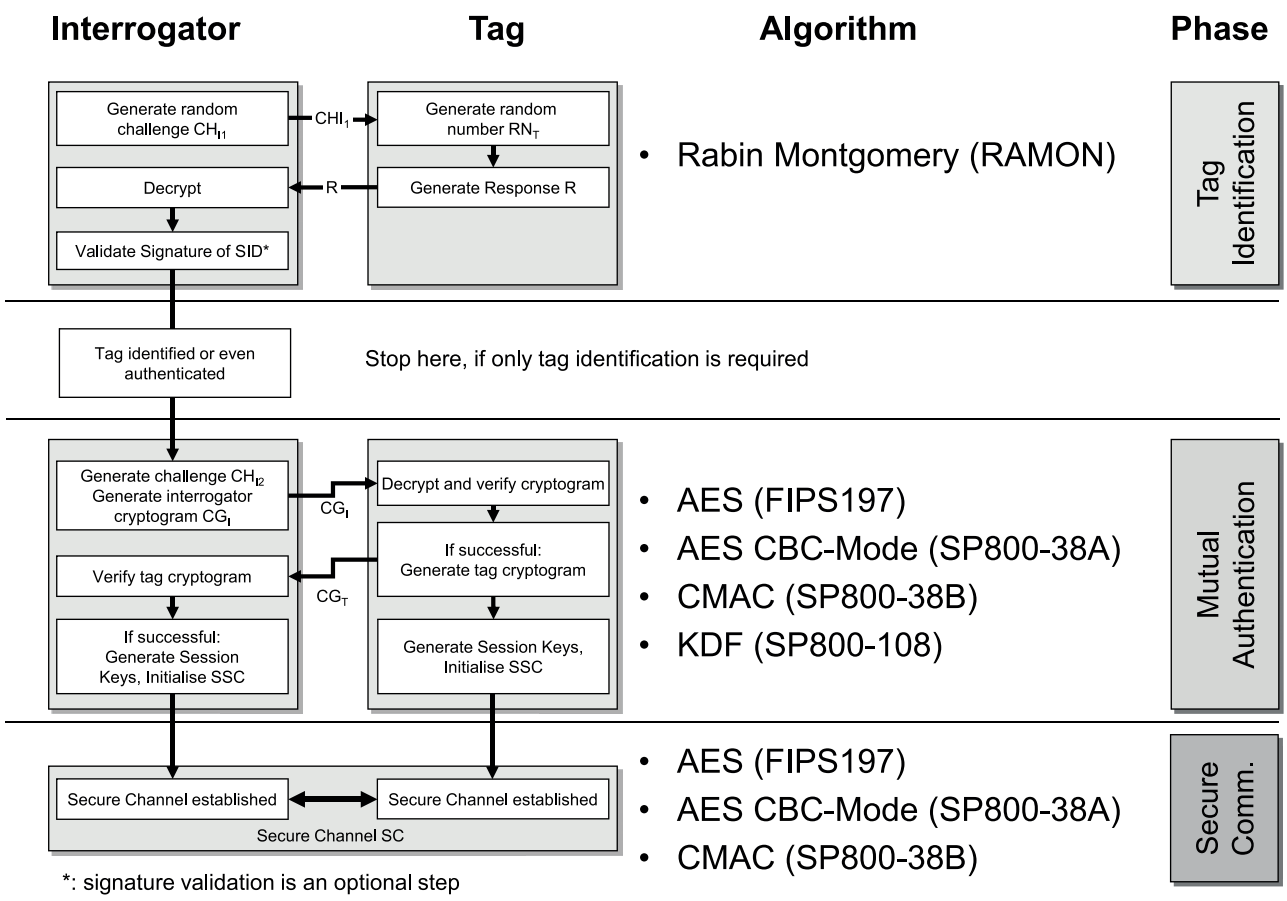


Figure 1 — Overview of the different implementation levels for the RAMON crypto suite

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[14]. The original algorithm is augmented by a method detected by Peter Montgomery[11], which avoids the division of long numbers in modular arithmetic. The advantage of combining Rabin encryption with the concept of Montgomery multiplication is related to 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 shall be performed by the Interrogator as specified in C.3 and C.4. The cryptographic keys are specified in 6.6. A test example for the RAMON encryption with a 1 024-bit public encryption-key K_E is defined in Annex D.

Annex G details the structure of the clear text record, the TLV 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. The TLV record shall be structured in accordance with Annex G.

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 [12]. The CBC mode for encryption is specified in Reference [8]. For MAC generation, CMAC according to Reference [9] 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:
		(R) ←	R = RM_ENC(K_E, MIX(CH_{I1}, RN$_T$, TLV record, '00' byte)).
	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.		
	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 further 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

6.4.1 General

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 6.4.3.

6.4.2 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*) := ∅.

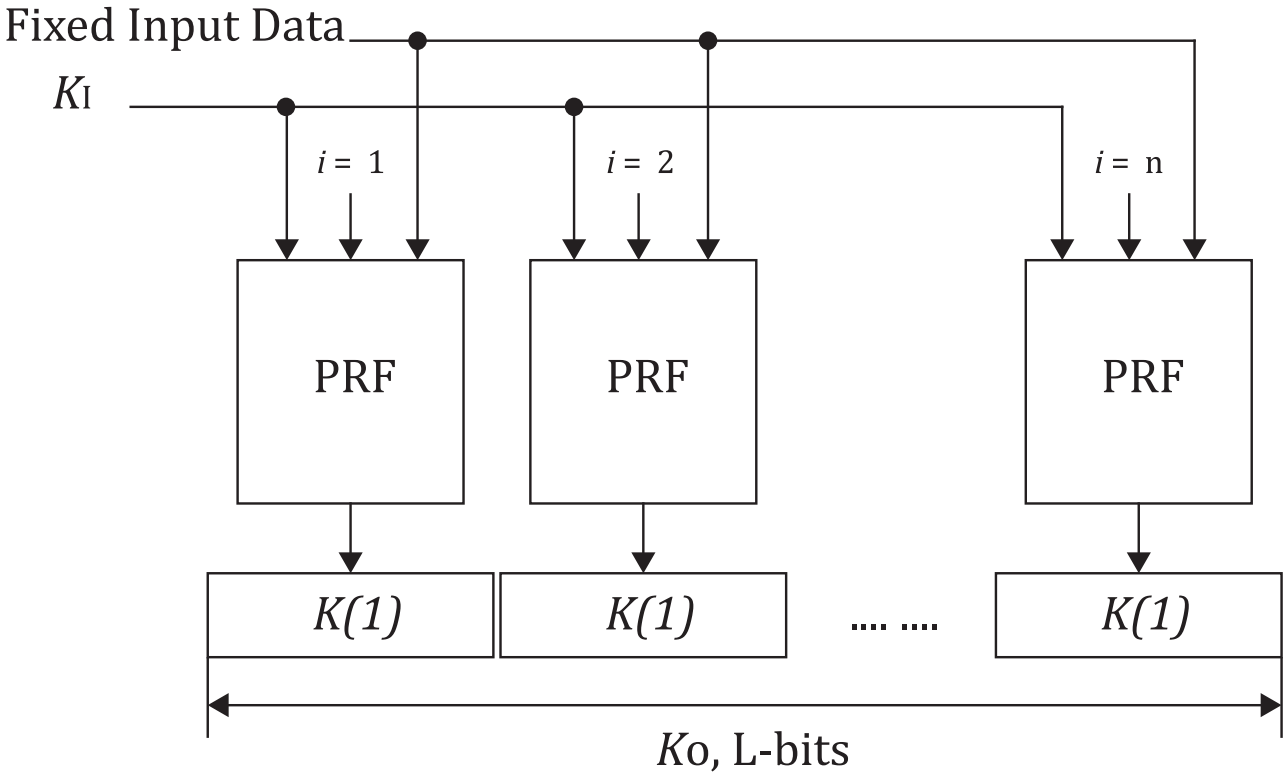
d) For i = 1 to n, do

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

— *result*(i) := *result*(i–1) || $K(i)$.

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

Output: K_O.



SOURCE Reference [7].

Figure 2 — KDF in counter mode

6.4.3 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 document (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 document, *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 document only supports AES-128 keys.

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 shall not use a derivation constant marked as RFU.

6.5 IID, SID, used keys and their personalization

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 is 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 can 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 shall 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 used by this crypto suite shall never be sent in plaintext. The SID can be signed to preserve integrity and to provide authenticity. The SID and the optional signature should never be readable for an unauthorized reader. 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-traceability 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 document; 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.

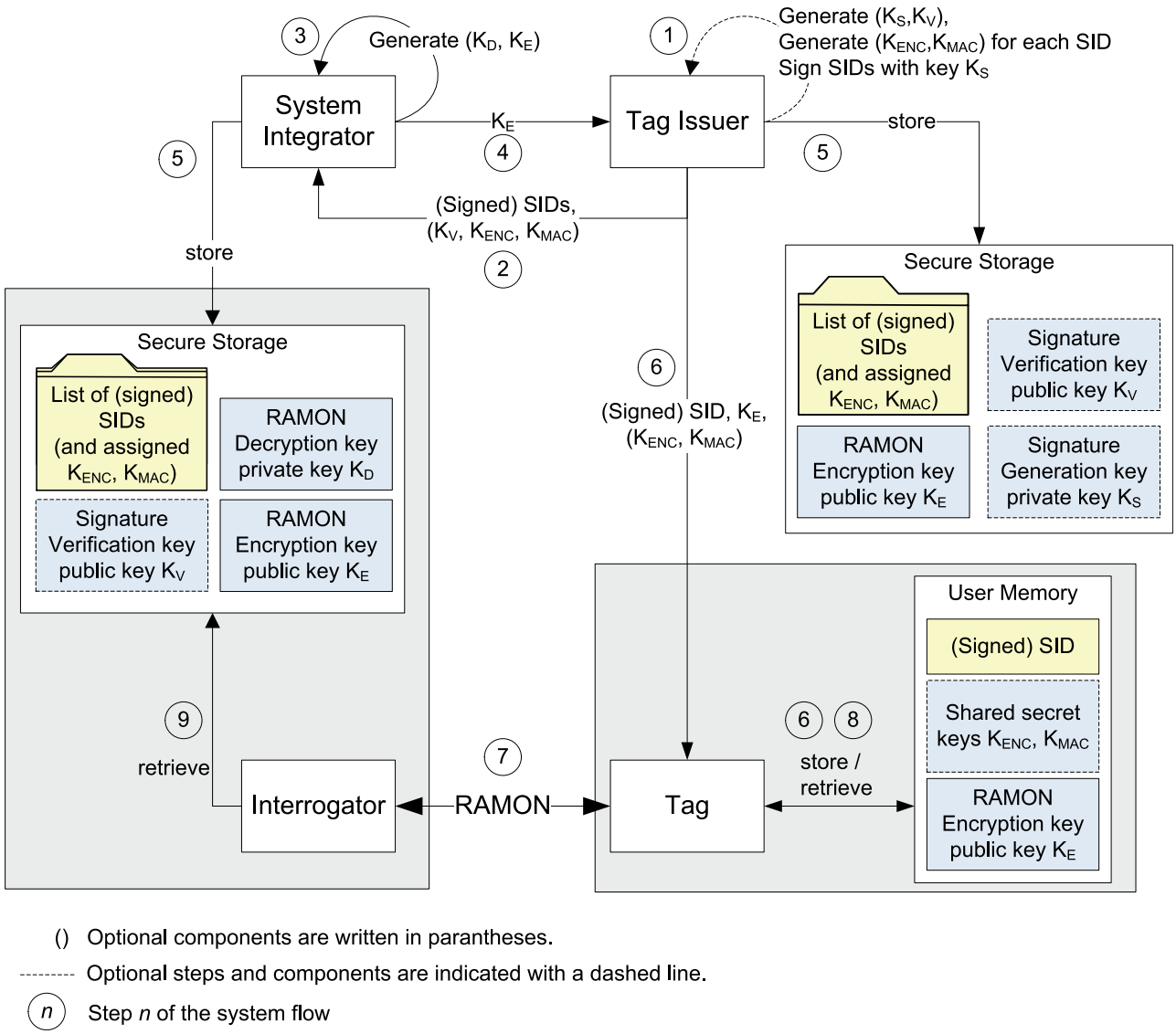


Figure 3 — System flow of an RFID System using the RAMON crypto suite

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	k
K_D	Private decryption key stored on Interrogator	k
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 beyond the scope of this document.

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 $p \equiv q \equiv 3 \pmod 4$, where the following relation holds:

$$K_E = p \times q$$

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

All keys shall be stored by the Tag and the Interrogator, such that unauthorized access and modification are 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:

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

where k is the bit length of K_E. k shall be divisible by 128 and ≥1 024.

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

All keys shall be stored by the Tag and the Interrogator, such that unauthorized access and modification are 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 G.3 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. The reader shall support both modes. The 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 can 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 shall 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 the 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 an "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

8.2.1 General

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

8.2.2 Partial result mode

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

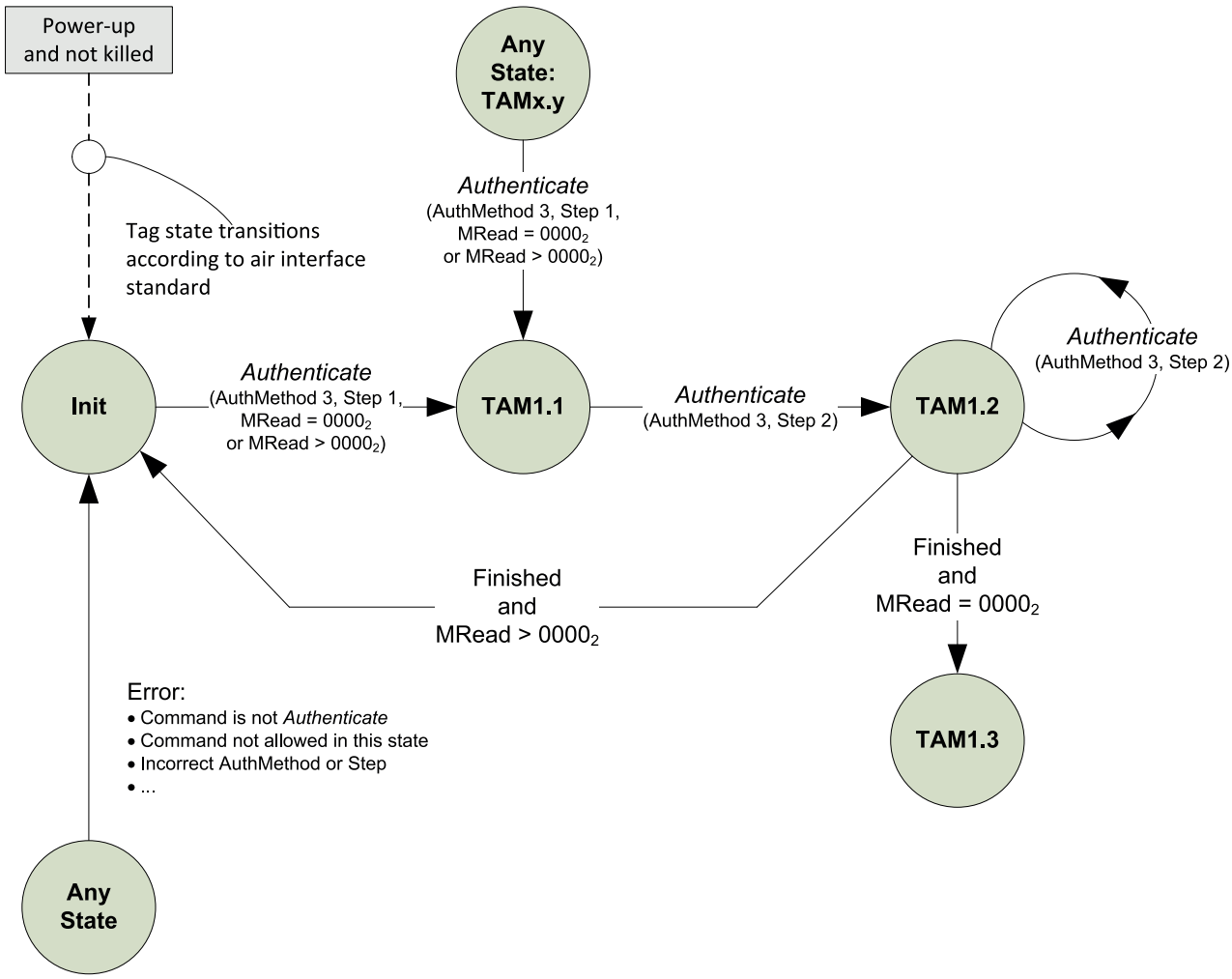


Figure 4 — Crypto suite state transitions for Tag identification in partial result mode

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.

8.2.3 Complete result mode

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

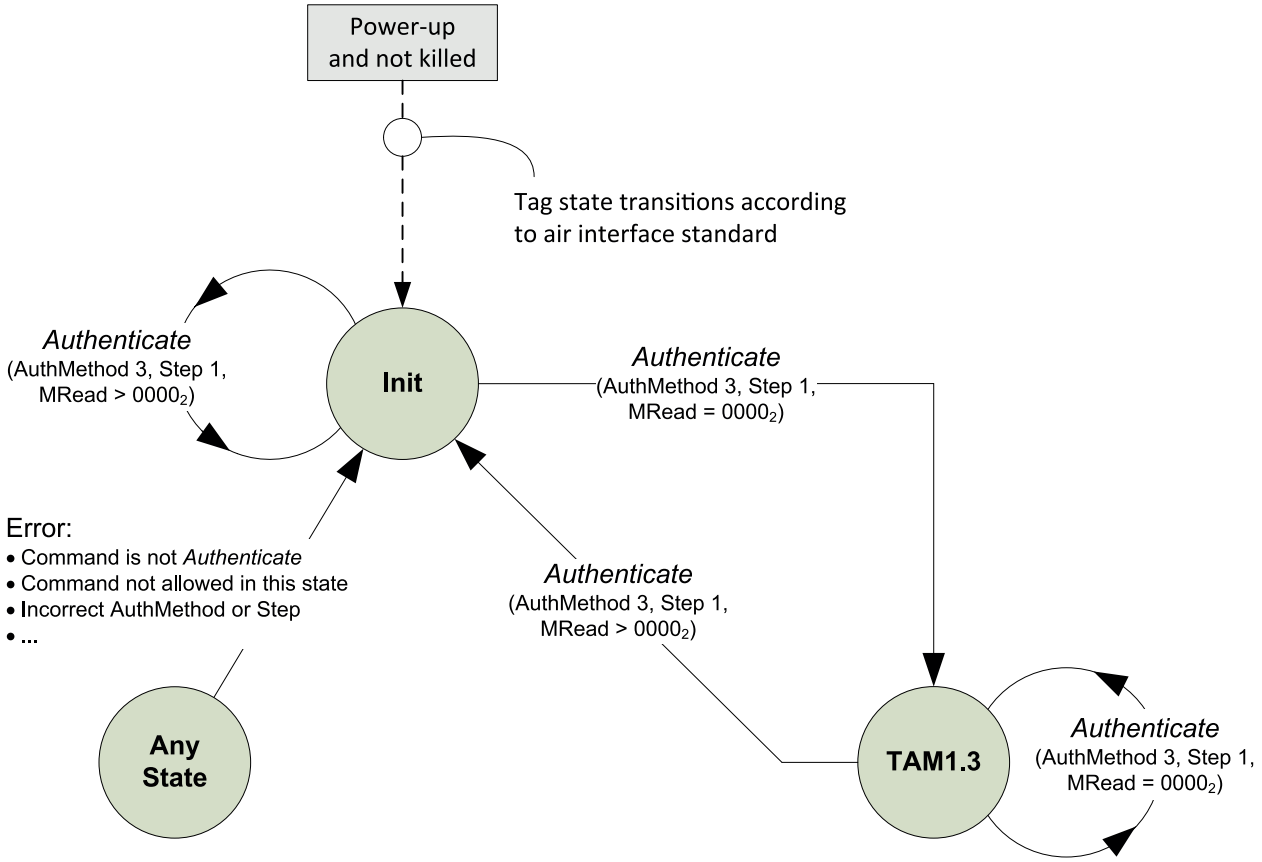


Figure 5 — Crypto suite state transitions for Tag identification in complete result mode

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

8.3.1 General

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.

8.3.2 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.

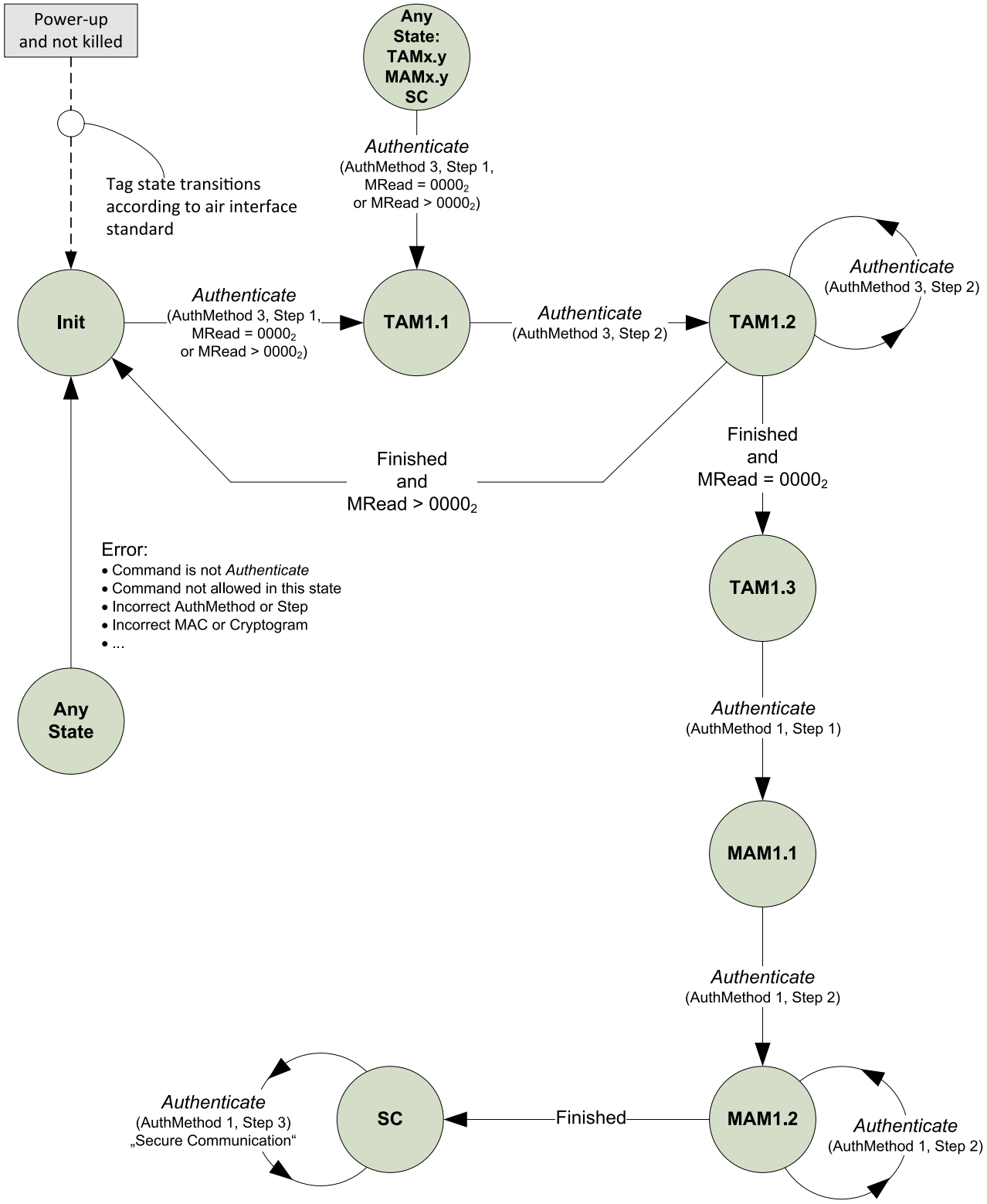


Figure 6 — Crypto suite state transitions for mutual authentication in partial result mode

8.3.3 Complete result mode

Figure 7 illustrates the state transitions that apply to this crypto suite for Tag identification followed by mutual authentication, both in complete 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 an *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.

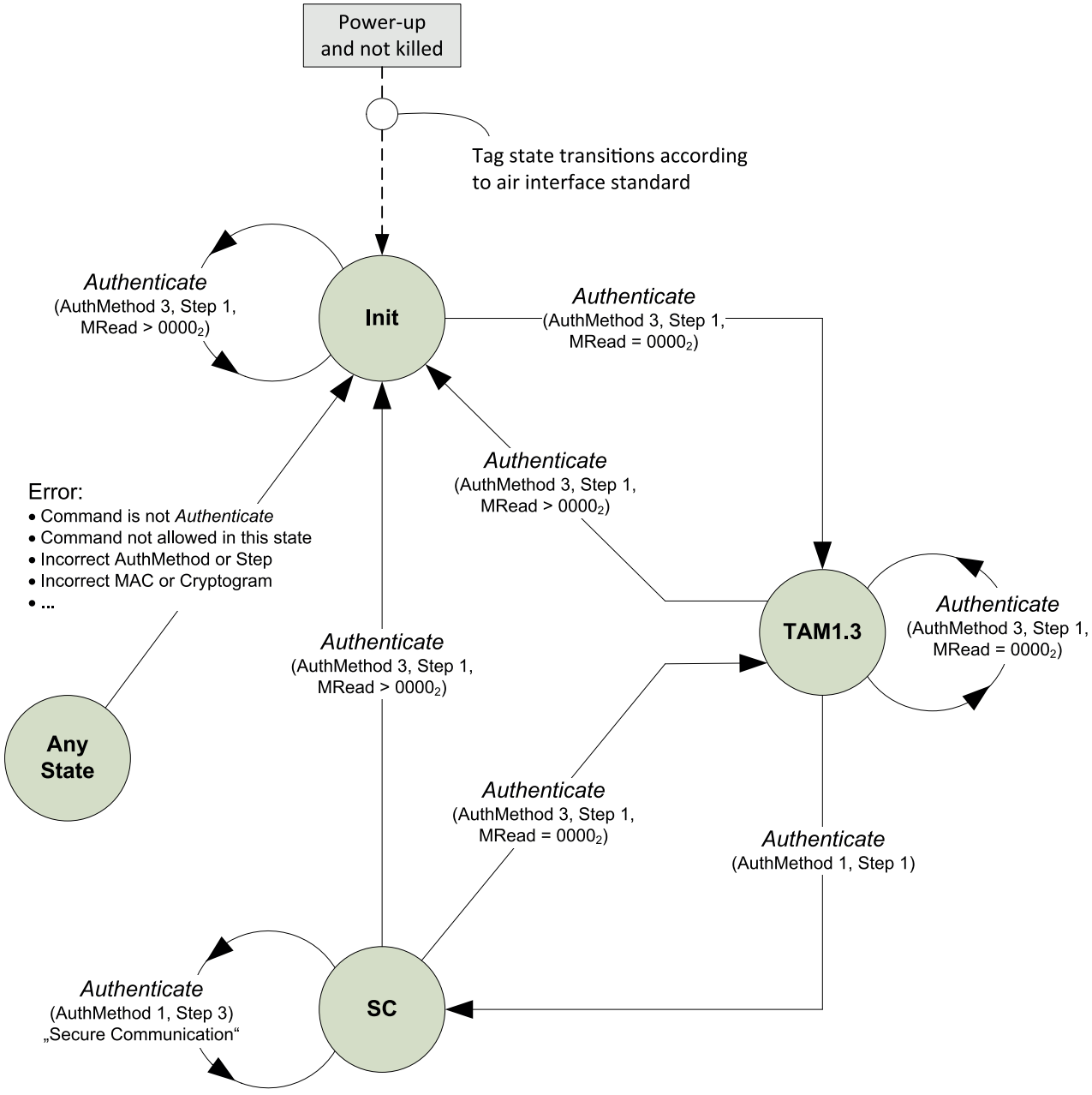


Figure 7 — Crypto suite state transitions for mutual authentication in complete result mode

8.3.4 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

10.1.1 General

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.

10.1.2 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.

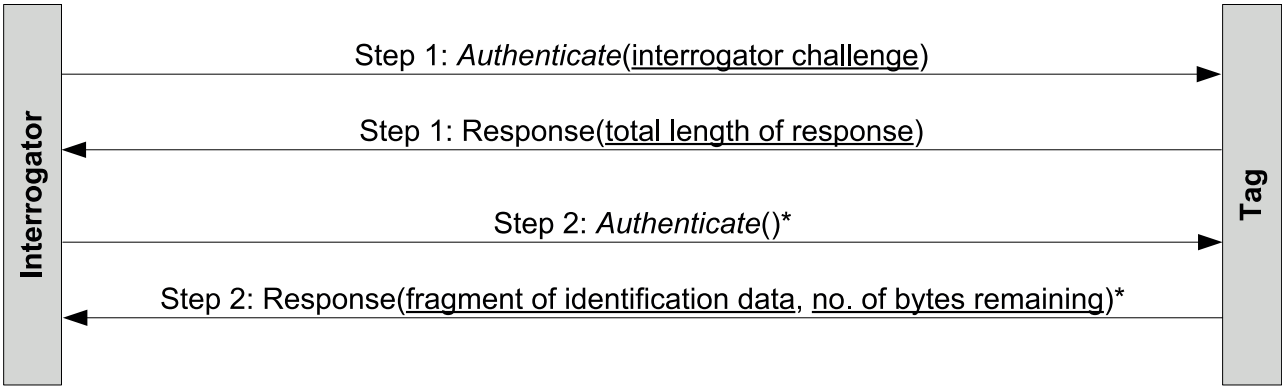


Figure 8 — Message exchange for Tag identification in partial result mode

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.

10.1.3 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.

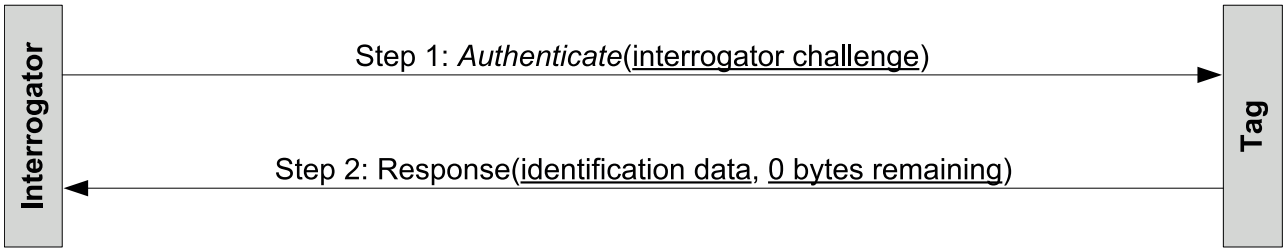


Figure 9 — Message exchange for Tag identification in complete result mode

10.2 Mutual authentication

10.2.1 General

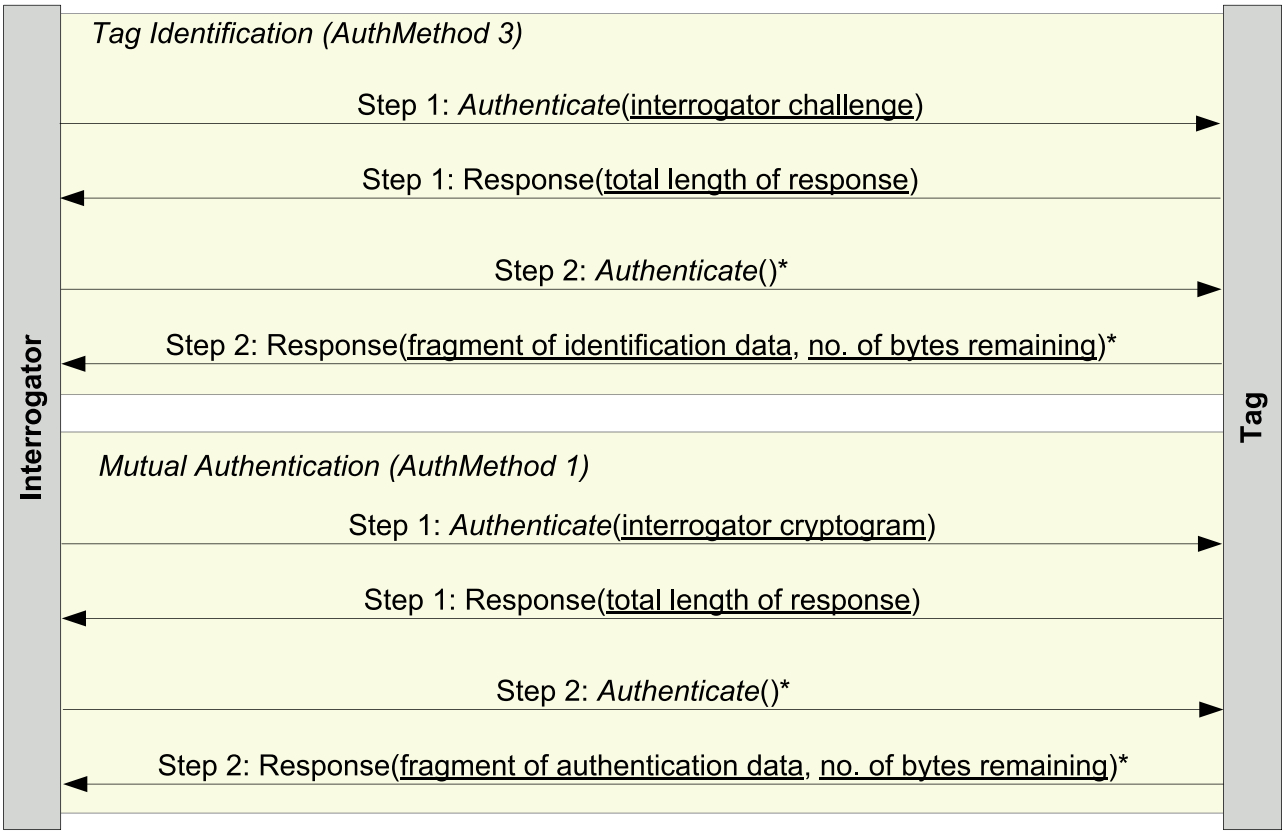
Before mutual authentication can be performed, the Tag has to be identified by the Interrogator. The same messages as specified in 10.1 need to be exchanged for this purpose. These steps are repeated in Figures 10 and 11 for completeness.

10.2.2 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

10.2.3 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.

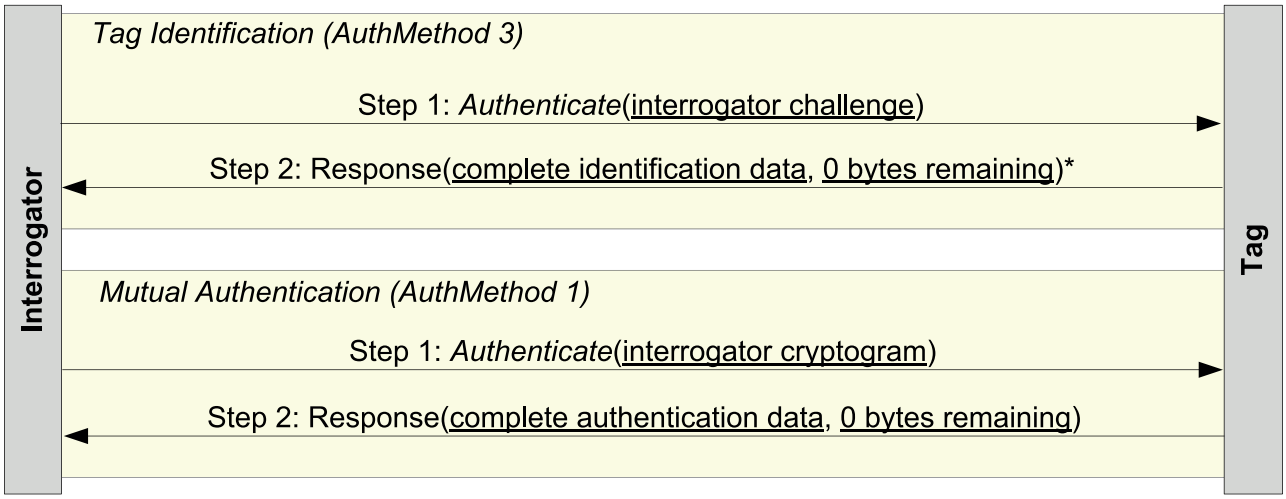


Figure 11 — Message exchange for mutual authentication in complete result mode

10.3 The Authenticate command

10.3.1 General

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.

10.3.2 Message formats for Tag identification

10.3.2.1 General

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". 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". 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**.

10.3.2.2 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 than 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.

10.3.3 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". 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): $S = CH_{I2}$ || 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 can 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". 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

10.4.1 General

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 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.

10.4.2 Response formats for Tag identification

10.4.2.1 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". 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 G for a detailed description of the clear text input (TLV record) 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". 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 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 (1 024 bits). The RAMON cryptogram shall be prepared as defined in C.1.

10.4.3 Response formats for mutual authentication

10.4.3.1 General

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: $\mathrm{DEC}(K_{ENC}, C)$.

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): $C = \mathrm{ENC}(K_{ENC}, S)$, IV = 0.

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

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.

10.4.3.2 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”. 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". 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.

10.4.3.3 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.

10.4.4 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

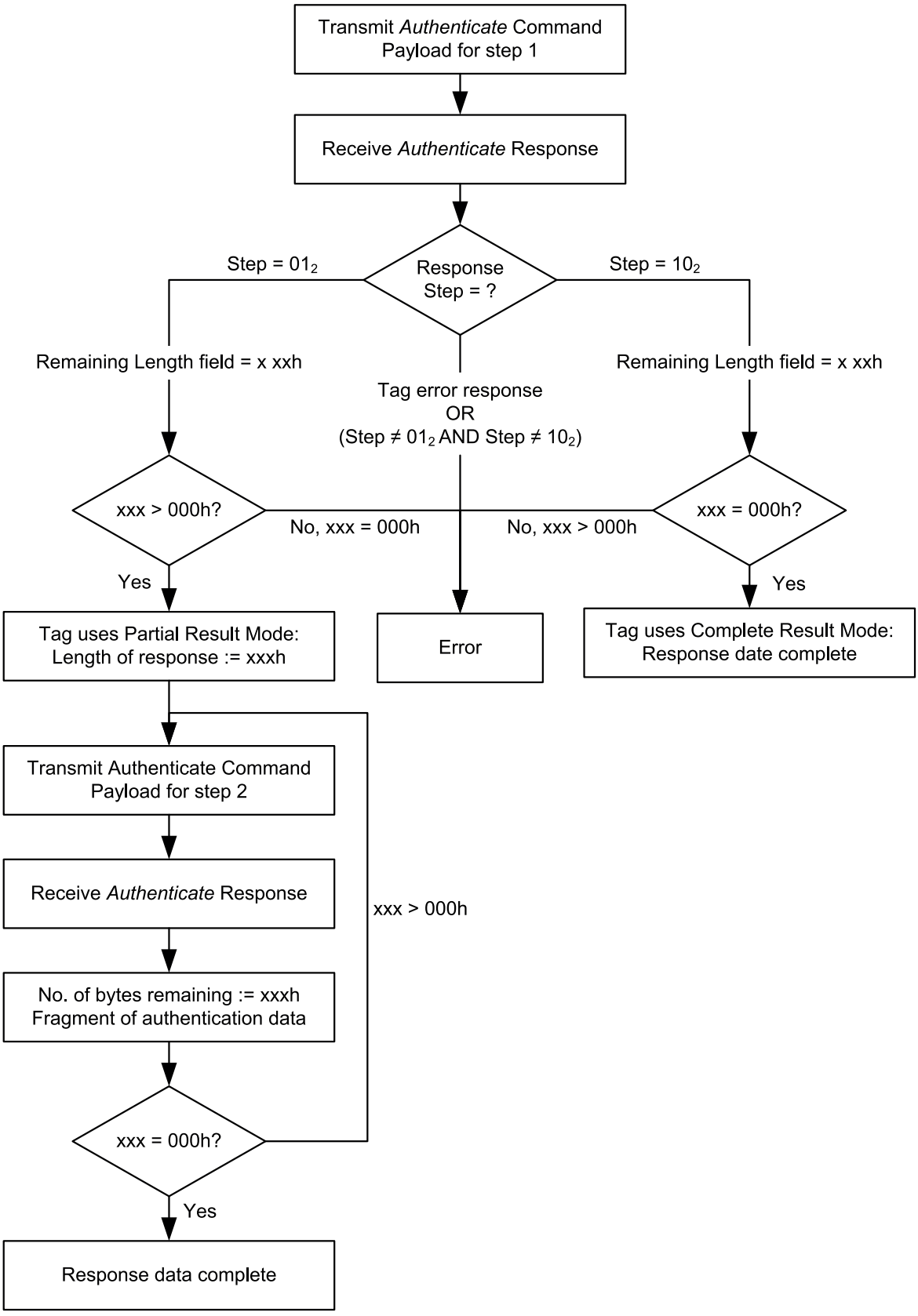


Figure 12 — Determination of result modes from Interrogators point of view

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

11.1 General

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.

11.2 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 11.5.2 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

Table 17 *(continued)*

Value	Description
0001_2	Commands defined in this crypto suite (see 11.4)
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.

11.3 Secure Communication response

11.3.1 General

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 11.5.2 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

11.3.2 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 can result from the operation of this cryptographic suite.

11.4 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.

A memory structure using KSel to select a sector and supporting an 8 bit record address is specified in Annex H.

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. The Interrogator shall not use an RFU command identifier.

An Interrogator shall set all RFU bits of the *Command Data* Field to “0”. 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.

11.5 Application of secure messaging primitives

11.5.1 General

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

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.

11.5.2 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.

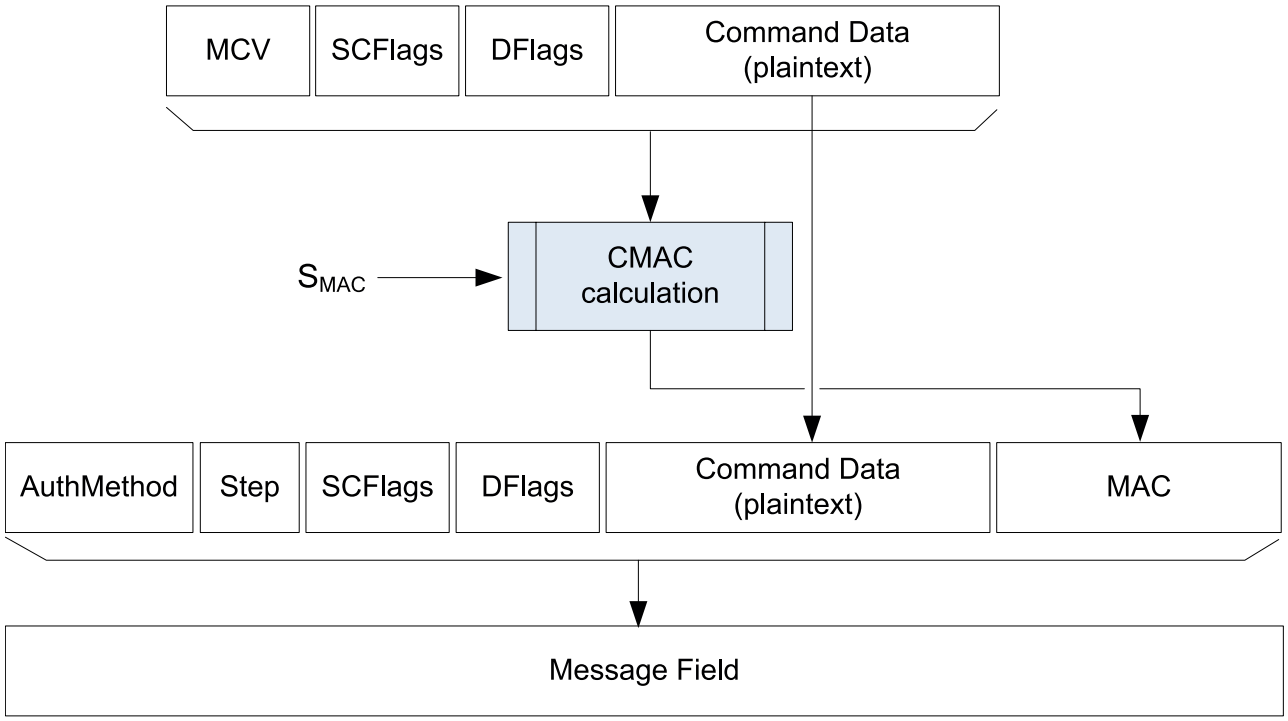


Figure 13 — *Secure Communication* Command MAC

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 have to 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.

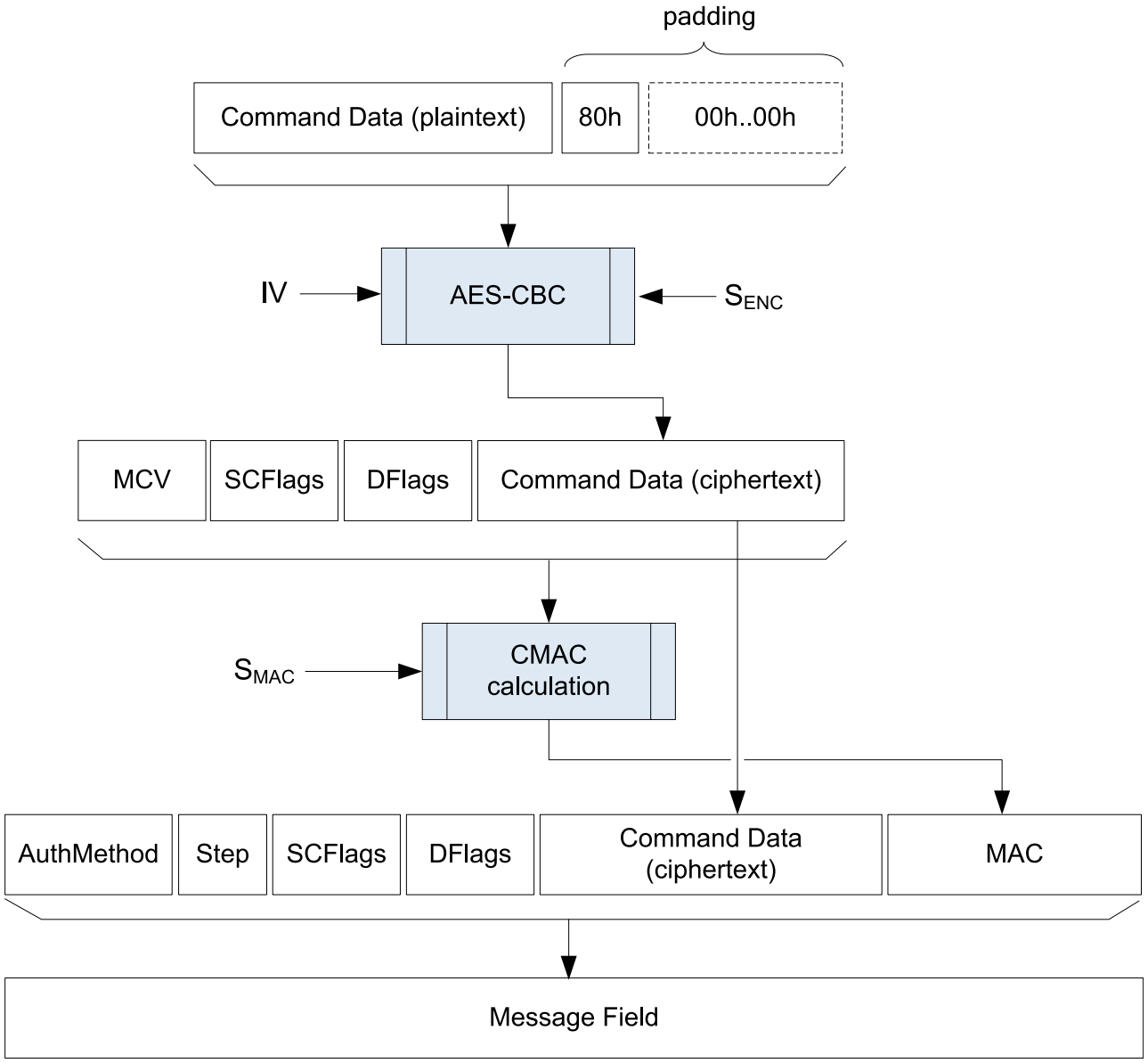


Figure 14 — Secure Communication Command ENC (with padding) and MAC

11.5.3 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.

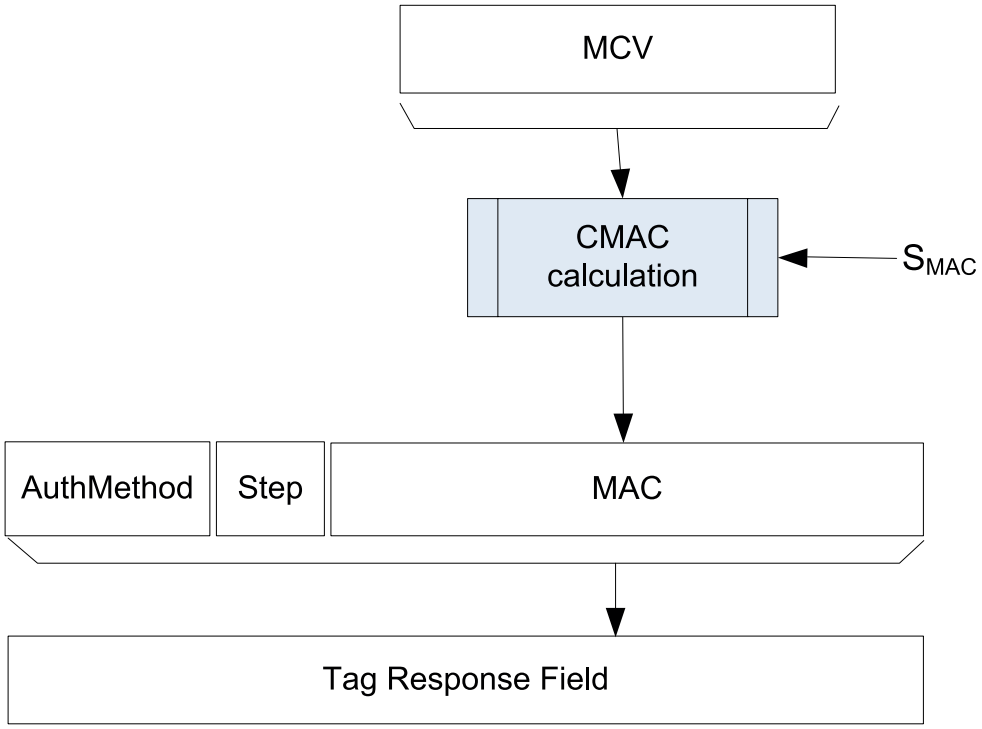


Figure 15 — Response MAC (no data)

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

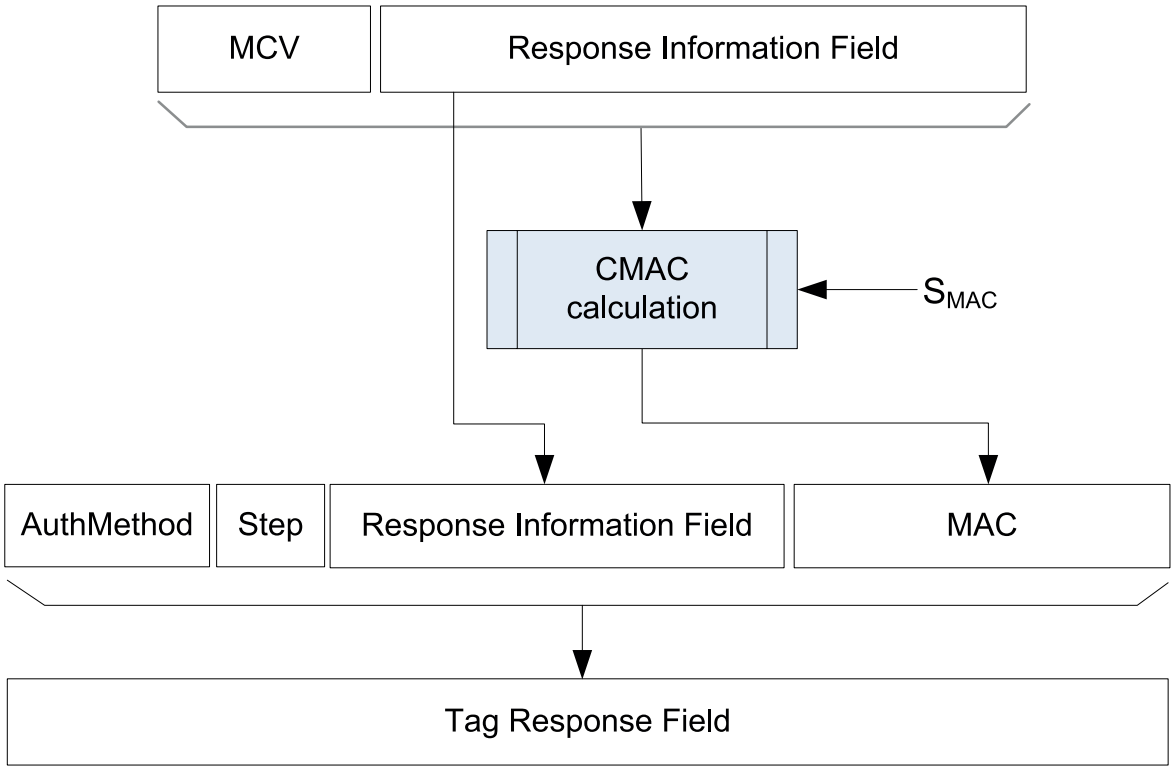


Figure 16 — Response MAC (with data)

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 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, 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.

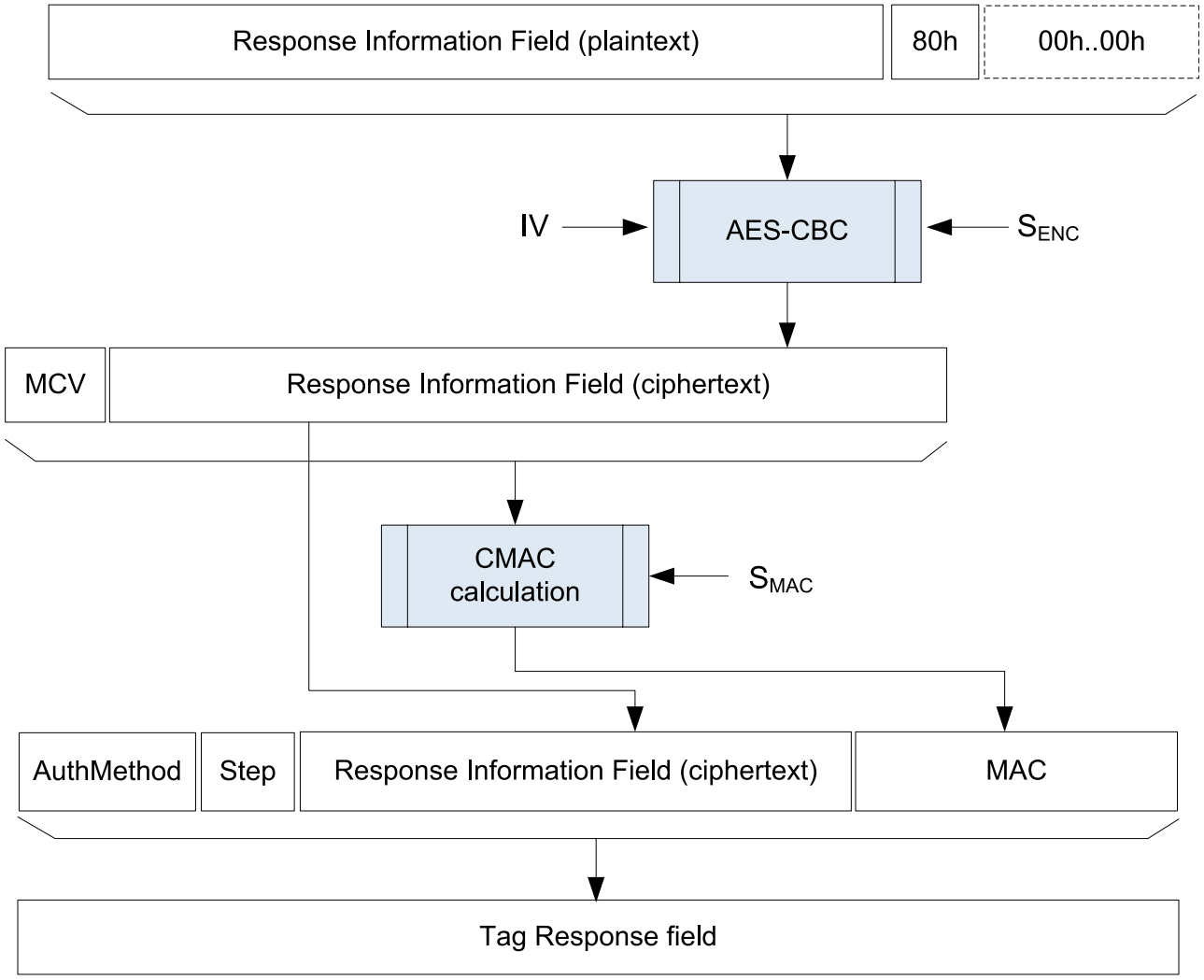
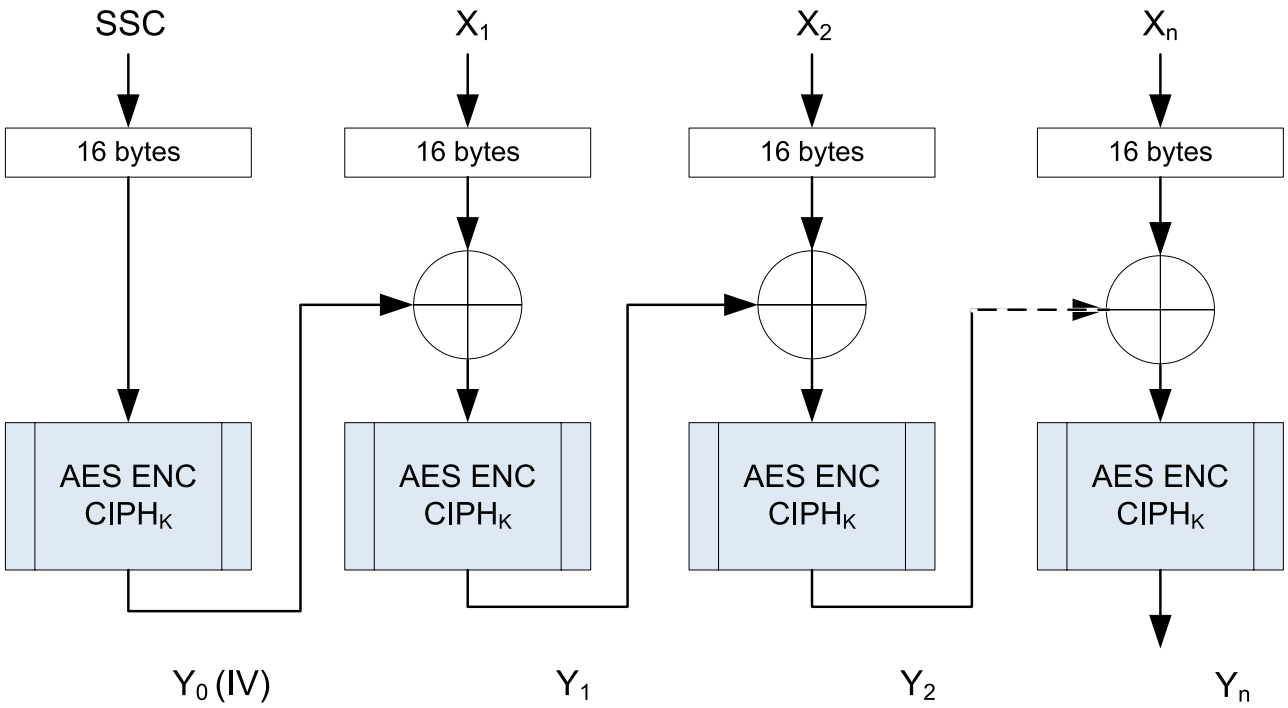


Figure 17 — *SecureComm* Response ENC (with padding) and MAC (with data)

11.5.4 Explanation of cipher block chaining mode



Key

X_i input plain text block

Y_i 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, $Y_n = ENC(K, Y_{n-1} \text{ XOR } X_n)$. 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, $Y_0 = ENC(K, SSC)$, $X_1 = Y_0 \text{ XOR } DEC(K, Y_1)$, …,

$X_n = Y_{n-1} \text{ XOR } DEC(K, Y_n)$

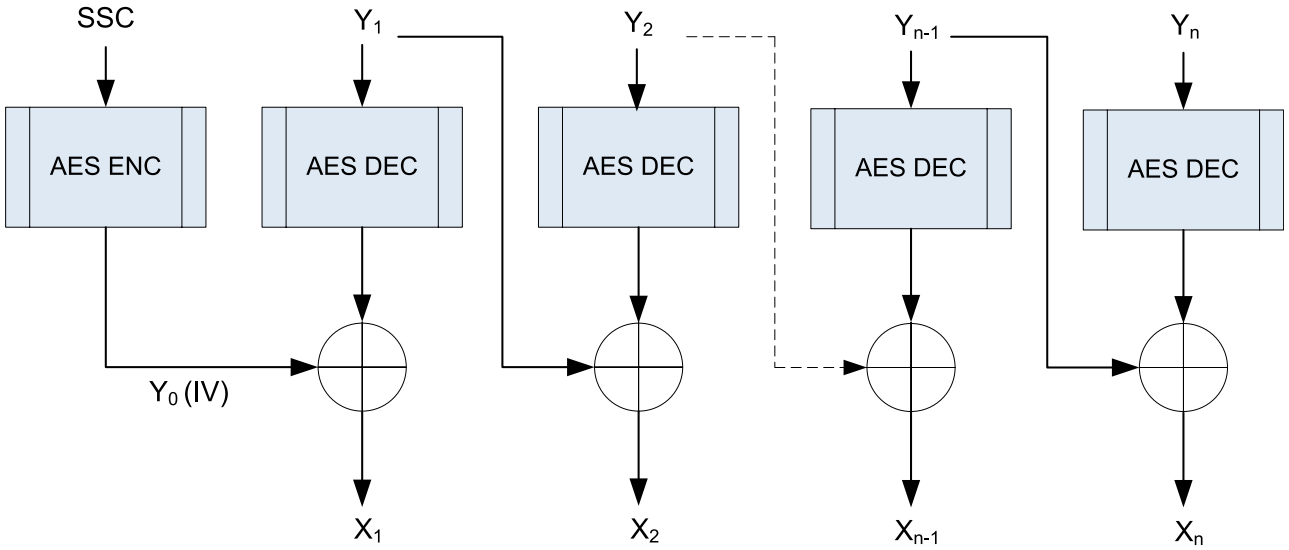


Figure 19 — Blockwise decryption using AES in CBC-mode with SSC

11.6 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).

Annex A
(informative)

State transition tables

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

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.

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

Table A.2 *(continued)*

#	Start State	Command[a]	Next State	Action/Result
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 Mutual authentication in partial or complete result mode are shown in Tables A.3 and A.4.

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 is 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.

Annex B
(informative)

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.

Table B.1 contains a listing of the Error Conditions that can 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 and not writable.
Crypto Suite error	Cryptographic error detected. This triggers a reset.

Annex C
(normative)

Cipher description

C.1 Preparation of the RAMON cryptogram

Depending on the command received by the interrogator, the tag prepares a byte block called the *RAMON cryptogram*. The bit length k of the RAMON cryptogram is equal to the bit length k of the Rabin-Montgomery key. Here *k* shall be ≥1 024 and divisible by 128. To simplify the following description, we define $m = k/64$. Then the RAMON cryptogram has a length of $8 \cdot m$ bytes (= $8 \cdot 8\ m$ bits), corresponding to a bit length of *k*. Since k ≥ 1 024, we always have $m \geq 16$. The Ramon cryptogram is eventually encrypted with the RAMON algorithm by the tag.

The actual payload of the RAMON cryptogram is a sequence of TLV structures. That sequence is called the *TLV-record*. The total length of the TLV record is always exactly $6m$-1 bytes. The internal structure of the TLV record is specified in Annex G.

Apart from the TLV record, the RAMON cryptogram contains the *padded interrogator challenge* PCH_{I1} of *m* bytes length. The first 16 bytes of PCH_{I1} are copied from the interrogator challenge CH_{I1}, which has been obtained from the interrogator. In case $m > 16$ the tag appends $16 - m$ random bytes to CH_{I1} in order to obtain the padded interrogator challenge PCH_{I1}.

The RAMON cryptogram also contains an *m*-byte *tag random number* RN_T generated by the tag. Finally, a zero padding byte is appended to the RAMON cryptogram. The order of the components of the RAMON cryptogram is given in Table C.1.

Table C.1 — Components of the RAMON cryptogram

	Padded Interrogator Challenge CH_{I1}	**Tag Random Number** RN_T	**TLV record**	**Zero Padding "00h"**
# of bytes	16 + (*m*–16)	*m*	$6 \cdot m - 1$	1
Total # of bytes	$8 \cdot m$			

C.2 The MIX function

The MIX function computes an input block for the Rabin-Montgomery encryption function specified in C.3. It is defined for any Rabin-Montgomery key size *k* with $k = 64\ m$, for $m \geq 16$. The Rabin-Montgomery input block is computed from the following input data specified in C.3:

PCH_I[0..m-1]	16-byte padded random challenge PCH_{I1}, *m* bytes, with PCH_I[0..15] received from the interrogator
RN_T[0..m-1]	random number RN_T generated by the tag, *m* bytes
TLV[0..6·m-2]	TLV record, $6m - 1$ bytes; this is the actual payload to be encrypted

Because the MIX function interleaves static and dynamic components of the Tag ID, its introduction reduces the risk of leaking information. The resulting 8 *m* bytes are input into the Rabin-Montgomery encryption function that is specified in C.3. The following C program specifies the MIX function:

```
void MIX(
   int m,              // bit length k of RAMON key divided by 64
```

```
   uint8_t PCH[m],     // input: padded random challenge
   uint8_t RN_T[m],    // input: random number generated by the tag
   uint8_t TLV[6*m-1], // input: TLV record, the payload to be encrypted
   uint8_t PERM[8*m],  // temporary buffer for the permuted RAMON cryptogram
   uint8_t OUT[8*m]    // output: the block to be Rabin-Montgomery encrypted
)
{
   int i, l ;                          // i is the main round counter
   for (i = 0; i < m; ++i) {           // loop for m permutation rounds
      for (l=0; l<5; ++l) {            // five bytes from TLV
         PERM[i*7+l] = TLV[i*5+l] ;
      }
      PERM[i*7+5] = PCH[i] ;           // one byte from PCH
      PERM[i*7+6] = RN_T[i] ;          // one byte from RN_T
   }
   for (i = m*7; i<m*8-1; ++i) {       // append final TLV bytes
      PERM[i] = TLV[i-m*2] ;
   }

   int j1 = 0, j2 = 1, mask ;          // j1, j2 are counters for masking
   for (i = 0; i < 8*m-1; ++i) {       // loop for masking
      if (i % 7 == 6 && i < 7*m) {     // for i < 7*m with i mod 7 = 6:
         OUT[i] = PERM[i] ;            //   byte PERM[i] is part of RN_T,
                                       //   take it without masking,
                                       //   do not increment counters j1, j2,
      } else {                         // otherwise:
         mask = RN_T[j1] ^ RN_T[j2] ;  //   generate mask byte from RN_T,
                                       //   bytes j1 and j2,
         OUT[i] = PERM[i] ^ mask ;     //   mask byte for output.
         j2 += 1 ;                     //   increment counter j2, inner loop
         if (j2 == m) {
            j1 += 1 ;                  //   increment counter j1, outer loop
            j2 = j1 + 1 ;              //   avoid equal combinations
         }
      }
   }
   OUT[8*m-1] = 0 ;                    // set high output byte to zero
}         j2 += 1 ;                    // increment counter j2, inner loop
         if (j2 == m) {
            j1 += 1 ;                  // increment counter j1, outer loop
            j2 = j1 + 1 ;              // avoid equal combinations
         }
      }
   }
   OUT[8*m-1] = 0 ;                    // set high output byte to zero
}
```

Data items *PCH_I*, *RN_T* and *TLV* are input to the MIX function, where they are permuted and masked (i.e. 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 can encourage some attacks. XOR-ing input data with random data unknown to an attacker is a countermeasure against SPA/DPA attacks.

The **permutation** is defined by the following sequential procedure: take 5 bytes from the *TLV* record, then 1 byte from *PCH_I* and then 1 byte from *RN_T*. This has to be done m times. Store these $7 \cdot m$ bytes into the output message buffer. Finally, take the remaining m-1 bytes from the *TLV* record and store them into the output message buffer.

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

Figure C.2 shows a sample application of the mask function for $m = 16$.

There are $m(m\text{-}1)/2$ combinations of different pairs of random bytes from RN_T, which is quite sufficient for masking $7 \cdot m$-1 bytes in case $m \geq 16$. The possible combinations of pairs of bytes are illustrated in Table C.2 for $m = 16$.

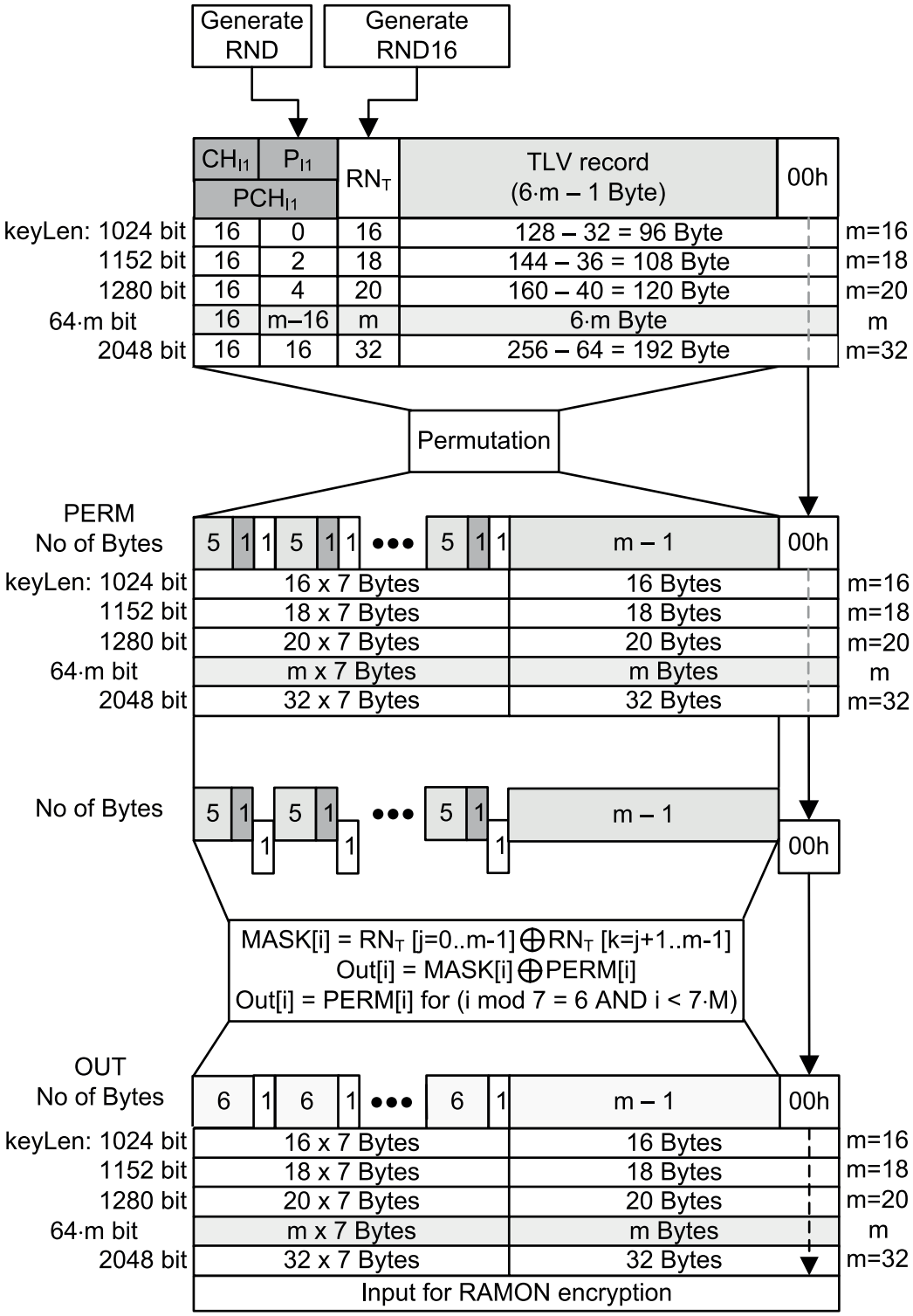


Figure C.1 — Illustration of the mix function

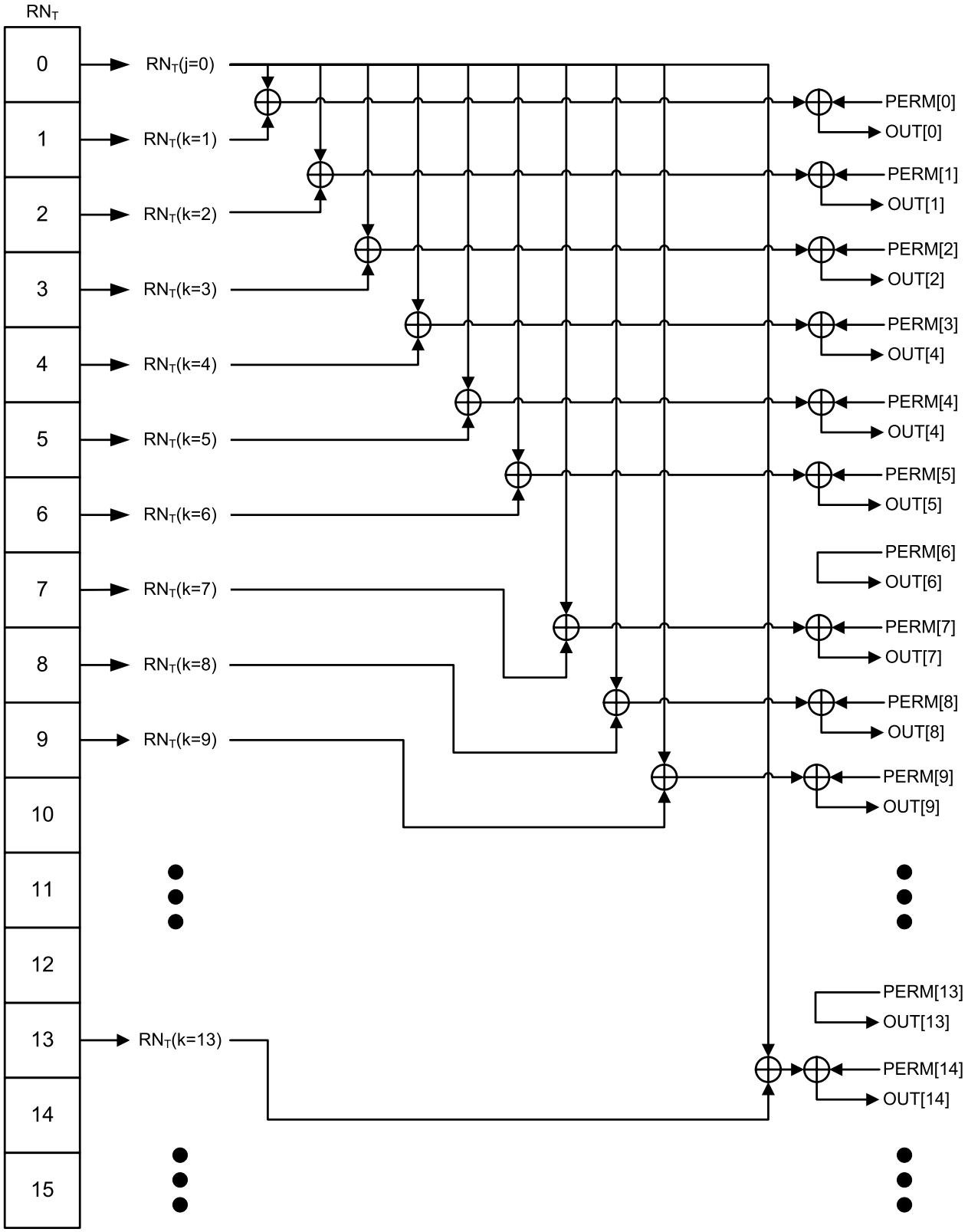


Figure C.2 — Illustration of the application of the mask function for *m* = 16

The possible combinations of the counter values *j*1 and *j*2 are shown in Table C.2 for *m* = 16.

Table C.2 — Possible combination of *j*1 and *j*2, applying the mask of the MIX function for *m* = 16

*j*1	*j*2														
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

The output of the MIX function is a byte string $b = (b_0, \ldots, b_{8m-1})$ of length $8m$ .For Rabin-Montgomery encryption, the string b is converted to an integer M as follows:

$$M = \sum_{i=0}^{8m-1} 256^i \cdot b_i$$

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

As described in Reference [14], the encryption of a clear text message M is calculated as

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

where C is the cipher text. Remember that

$$n = p \times q$$

where p and q are primes which satisfy the congruency condition $p \equiv q \equiv 3 \pmod 4$.

In order to optimize security, p and q should be of the same order of magnitude, $\log p \approx \log q$. The message M should 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[11]. To this end, a Montgomery base R, is defined, where R is a power of 2, such that $R \geq 2^k$. 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 follows:

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

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. $C^* \neq C$, 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 $R = 2^k$. 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 shall satisfy

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

and

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

and

$$p = q = 3 \pmod 4$$

where k is the bit length of the public key. The first two conditions are given for security reasons. The last condition simplifies the decryption as described in C.4.

For additional information, refer to Reference [13], 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

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

Assuming $p = q = 3 \pmod 4$ and p, q prime, the four possible square roots of C modulo $p \cdot q$ can be computed as follows

$$w_p = (q^2 C)^{(p-3)/4} \bmod p$$

$$w_q = (p^2 C)^{(q-3)/4} \bmod q\ ,$$

$$\sqrt{C} = \left(2pq \pm q\left(\left(C\ w_p\right) \bmod p\right) \pm p\left(\left(C\ w_q\right) \bmod q\right)\right) \bmod \left(p \cdot q\right)$$

The four possible combinations of the signs in the last formula yield the four square roots of C modulo p·q, provided that such a square root exists.

This formula is correct, since

$$x^{(p-3)/4} = \pm x^{-1/2} \pmod{p}$$

holds for any quadratic residue x modulo p, with p prime and $p = 3 \pmod 4$, so that we have

$$\sqrt{C} = \pm q \cdot \left(q^2 \cdot C\right)^{-1/2} = \pm q \cdot w_p \pmod{p}$$

Similarly, we have

$$\sqrt{C} = \pm p \cdot w_q \pmod{q},$$

so the formula follows from Chinese remaindering.

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) as well as all other intermediate results, 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^{k+64} for a key length of k bits.

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:

```
void Inverse_MIX(
   int m,              // bit length k of RAMON key divided by 64
   uint8_t RMD[8*m],   // input: the Rabin-Montgomery decrypted block
   uint8_t PCH[m],     // output: padded random challenge
   uint8_t TLV[6*m-1]  // output: plain TLV record
)
{
   int i ;                              // i is the main round counter
   int j1 = 0, j2 = 1, mask ;           // j1, j2 are counters for masking
   int i_pch = 0, i_tlv = 0;            // counter for outputs PCH  and TLV
   for (i = 0; i < 8*m-1; ++i) {        // loop for masking
      if (i % 7 != 6 || i >= 7*m) {     // for all i >= 7*m or i mod 7 != 6:
         mask = RMD[7*j1+6] ^ RMD[7*j2+6]; // mask byte computed from RMD
         if (i % 7 == 5 && i < 7*m) {
            PCH[i_pch] = RMD[i] ^ mask;    // store plain challenge byte
            i_pch += 1;
         } else {
            TLV[i_tlv] = RMD[i] ^ mask;    // store plain TLV record byte
            i_tlv += 1;
         }
         j2 += 1 ;                      // increment counter j2, inner loop
         if (j2 == m) {
            j1 += 1 ;                   // increment counter j1, outer loop
            j2 = j1 + 1 ;               // avoid equal combinations
         }
```

```
      }
   }
}
```

The output OUT[] of the inverse MIX function consists of the following two components:

PCH[0..m-1]	Padded random challenge PCH_{I1} received from the interrogator, m bytes
TLV[0..6·m-2]	TLV record, 6·m-1 bytes

C.7 Coding examples

C.7.1 RAMON encryption

The following C code is an example of a RAMON encryption. Here the input array PLAIN is an array of 8m bytes. The output of the C function MIX() in C.2 may be passed directly to the RAMON encryption function as input array PLAIN.

```
void RAMON_Encrypt(
   int m,               // bit length k of RAMON key divided by 64
   uint8_t PLAIN[8*m],  // input: plain text to be encrypted
   uint8_t N[8*m],      // input: public RAMON KEY
   uint8_t TMP[16*m+8], // temporary buffer for square of input
   uint8_t OUT[8*m]     // output: Rabin-Montgomery encrypted PLAIN
)
{
   int i, j;            // Loop couunters
   uint16_t Inv;        // Montgomery inverse
   int r = 8*m + 8;     // Montgomery residue: 8 * r = k + 64

   // Compute TMP = PLAIN * PLAIN
   for (i = 0; i < 8*m + r; ++i) TMP[i] = 0;  // zero TMP
   for (i = 0; i < 8*m; ++i) {                // TMP = PLAIN * PLAIN
      uint16_t acc = 0;
      for (j = 0; j < 8*m; ++j) {
         acc = acc + PLAIN[i]*PLAIN[j] + TMP[i+j];
         TMP[i+j] = acc & 0xff;
         acc = acc >> 8;
      }
      TMP[i+8*m] = TMP[i+8*m] + acc;
   }

   // Compute Montgomery inverse Inv with Inv * N = -1 (mod 256)
   Inv = 1;                  // Now Inv = -1/N (mod 2). Use Hensel's
   for (i = 0; i < 3; ++i) { // lemma to compute Inv = -1/N (mod 256)
      Inv = (Inv * (2 + N[0] * Inv)) & 0xff;
   }

   // Montgomery reduction of TMP:
   // Add multiple of N to TMP such that TMP = 0 (mod 1 << (k+64))
   for (i = 0; i < r; ++i) {
      uint16_t f = (Inv * TMP[i]) & 0xff; // multiplier for round i
      // Now compute TMP = TMP + f * (1 << (8*i)) * N
      uint16_t acc = 0;
      for (j = i; j < 8*m + i; ++j) {
         acc = acc + f * N[j-i] + TMP[j];
         TMP[j] = acc & 0xff;
         acc = acc >> 8;
      }
      for (j = 8*m + i; j < 8*m + r; ++j) {
         acc = acc + TMP[j];
         TMP[j] = acc & 0xff;
         acc = acc >> 8;
      }
   }

   // Output TMP >> (k+64)
```

```
   for (i = 0; i < 8*m; ++i) {
      OUT[i] = TMP[i+r];
   }
}
```

C.7.2 RAMON decryption

The following Java- code demonstrates the Rabin-Montgomery decryption.

```
//
// Sample code for Rabin-Montgomery decryption
//

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

   // Given a Rabin-encrypted ciphertext c and a Rabin key (p,q),
   // where p and q are secret primes with p = q = 3 (mod 4),
   // function RootTerm() returns  t_p = q * (c * w_p mod p) with
   // w_p =  (q * q * c)^((p-3)/4) mod p as in annex C.2.
   // The corresponding term  t_q =  p * (c * w_q mod 2)
   // will later be calculated by exchanging p with q.
   // Then the four roots of c are +- t_p +- t_q (modulo p*q).
   static BigInteger RootTerm (BigInteger c, BigInteger p, BigInteger q){
      BigInteger exp = p.subtract(new BigInteger("3")) ;
      exp = exp.divide (new BigInteger ("4")) ;
      BigInteger basis = q.multiply(q) ;
      basis = basis.multiply(c) ;
      BigInteger result = basis.modPow (exp,p) ;
      result = result.multiply(c) ;
      result = result.mod(p) ;
      return ( result.multiply(q)) ;
   }
   //
   // Given a Rabin-Montgomery encrypted ciphertext c and a
   // Rabin key (p,q), where p and q are secret primes with
   // p = q = 3 (mod 4), function RamonDecrypt()returns an array
   // of BigInteger containing the four possible plaintext
   // candidates. Here all input and output data are given as
   // big integers. See annex C.3 for the conversion between
   // a big integer and a byte string.
   static BigInteger [] RamonDecrypt(
      BigInteger c,  // Ciphertext to be decrypted
      BigInteger p,    // private key: prime Factor p of n
      BigInteger q,    // private key: prime Factor q of n
      int keysize      // RAMON key size in bits ) {
      // Unroll the effect of the Montgomery multiplication
      // c -> ( c * 2 ^ (keysize + 64) ) mod n, where n = p*q
      BigInteger n = p.multiply(q) ;
      c = c.shiftLeft (keysize + 64) ;
      c = c.mod(n);

      // Calculate the 4 square roots of the modified ciphertext c
      BigInteger term1 = RootTerm (c,p,q) ;
      BigInteger term2 = RootTerm (c,q,p) ;

      BigInteger roots[] = new BigInteger[4] ; // buffer for result
      // 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) ;
      root = root.add (n) ;
      roots[2] = root.mod (n) ;
      // fourth root = n - third root
      roots[3] = n.subtract (roots[2]) ;
      return roots;
```

```
}
```

The inverse MIX function (see C.6) should be applied to each of the four plaintext candidates computed by function RamonDecrypt(). In order to determine which one of the four candidates is the correct one, the Interrogator shall check all candidates for correct presence of the previously sent challenge CH_{I1}. as described in C.4.

WARNING — The code in this subclause is for demonstration only. C.4 requires that a life system do not output any root that does not contain the previously sent challenge CHI1 after running the inverse Mix-function. Also, a life system shall clean up all internal buffers containing any data related to any of the roots.

Annex D
(informative)

Test vectors

D.1 Notation

This annex contains a test example for the RAMON encryption with a 1024-bit public key K_E.

In most cases, the messages exchanged, as well as individual message components, are considered as a string of 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, endianness is important: messages are interpreted 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 the secret keys are not needed for the RAMON encryption, the prime factors p and q which satisfy the congruency condition $p \equiv q \equiv 3 \pmod 4$. for the modulus (in numeric notation) are given:

```
P = 0xc868f88a83d8e9689a44ad154b29d8b6048e5f55cdd9fe5287899ff174168a32
     4e682c127e35118736af6898e3b62a8a58ed623e409991fb4925056c6a401e57

q = 0xef0d08c4c672f46edc80b908d3e15cea1089d46f90a36a333d22ea59038dcb7c
     c3c9fad18228da3710bb633ed5a076224a17e64104631b2ace6a2b4acc05ad67
```

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

```
KE = n = 0xbb24343b439e006ce1fa33383e2304081f5c62a367466e3a9387e3717f6
          26b5b40fb9d910a82f595be9b4c281aca0bf80449fc4d3e7a5e35f566565
          46c9d47e000000000000000000000000000000000000000000000000000000
          000000000000000000000000000000000000000000000000000000000000000
          00000000000000001
```

D.3 Authentication message

This subclause specifies the components from which the authentication message is formed. Here the items are given mostly in message format. The numeric format is shown in D.4. 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 1 024 bits.

D.4 RAMON encryption

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

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

In this representation, a leading zero byte occurs which can 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
     e274af84964d197054cbd3e9c1015ae867ee0b3ffa09826f4e50d228587e77f6
     d0a8db1b7a561f1f58ac0d4dc3cf252cd2a9df39a90d0c7ff1ae44ee9b9eac93
```

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

```
TX(C*) = 93ac9e9bee44aef17f0c0da939dfa9d22c25cfc34d0dac581f1f567a1bdba
         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
      c4df9732cc0a4815d03ac9af47bdf39e95c66d3614574f058732ae4844567a46
```

```
 -r = 0xa66a61bbfe2c020d06620495c7195950478cbf2aa4afba40a2fd66e13436abc8
      605884fae1838881f01e669fd96add5d3e241f22bc9f30d1d2551080a632acf4
      3a6b33746d49b64cedac541d6bdb8d3522f139d82a0c0d442aa383340e7b7f74
      3b2068cd33f5b7ea2fc53650b8420c616a3992c9eba8b0fa78cd51b7bba985bb

 +s = 0x0039cedce554bd6c87c42c1c011b733fc1500d982602cd1a297780b9ee202ce7
      0a424e71258d275eb05451621fc2e6b3c3a9250ec631ad37fc3d4a949e2b5257
      f50abb2e709917dc2a2e54f2065538d4a358641ad70aaf4d1f1c4ad4c9b82d79
      aed85c6d7af80de9bd854ba3a707a149ed497066fc32d6d357a7b12c9b5a0c16

 -s = 0xbaea655e5e4943005a36071c3d0790c85e0c550b4143a1206a1062b791423e74
      36b94f1fe4f5ce370e46fac5fb07254440a0d73e7848b0fdf9290bbfce71f588
      0af544d18f66e823d5d1ab0df9aac72b5ca79be528f550b2e0e3b52b3647d286
      5127a3928507f216427ab45c58f85eb612b68f9903cd292ca8584ed364a5f3eb
```

These roots have to be demixed using the inverse MIX Function MIX^{-1}:

```
MIX^-1(+r) = 56fbae4de351a5dad77db952e28f5da4326dafcc84f2ca1294452a41ffb
             ff9d719e7a8f288a7a3384f674b5e7b311c569dafec601e050d5031da96
             19e1ee763b77d6583a1a923cf355c485e1beb3a50c7a07d4585f50883d4
             371429dc5ee59164dc81d2dd6d3ad1ca82f67585313c6769cc74a18a694
             6267cd95a4d0ac2f0714

MIX^-1(-r) = a98751451c04358f47d984e4adbae464cd9250337b0d35ed6b1022d983
             ab4047e418570d77585c6d47ff5fe7ebcee3a962501335867919e9a1256
             9e61e117ea30bc2e1c5e56dc30c5d5cf9f5074cba3a22cdf72a010ba3ff
             42237209392b5428dff61632c615b0c92e52931705df630d907235f55c9
             7d9565af98d77d90430a6

     CH_I1 = c24c6f86f4a4c11e0022bde0b9f22fd7 //+s is the correct root

MIX^-1(+s) = c24c6f86f4a4c11e0022bde0b9f22fd7a770a37ab8afd42a0a4a0e1f8d2
             c1ac1c108878424da7e3b9b44c2502f720d9421e7933702a184c4c8d2d8
             3d95b6a76b34ebe1fa80a8a224a8726e264ee23bc0996c9ac9a30f48a00
             c261256e1e43a4e80ffba17bac4008e9db5d0fde9669c181963d04549eb
             a2d7e7acd7c7c801ab00

MIX^-1(-s) = 3d3490b60be55e5f9fc603a2ee01810c588f5c8547502bd5f50b3efbf53
             e205e3cf7787bdb25817aaba0d06ab08df26bde186c76e6d996fe572d27
             c26a49978f4cf9db057f575ddb98961634741de467c382954b84d751520
             51b2a8eab8a48f0e9f1cf91ac3d624f5c4c3054c8bfc05b0345ac08ecf9
             03715586825e4005e4ba
```

Extraction of the CH_{I1} bytes from the demixed roots reveals that root +s is the correct one, while the others have to be discarded.

Annex E
(informative)

Protocol specifics

E.1 Supported security services

The RAMON crypto suite provides security services for ISO/IEC 18000-3 mode 1, ISO/IEC 18000-3 mode 3, ISO/IEC 18000-4 mode 4 and ISO/IEC 18000-63 air interface protocols. Details for the specific implementation of these air interface protocols are given in the following subclauses.

For the implementation of the RAMON crypto suite, an air interface protocol shall support the required security commands that this crypto suite has implemented. Security commands contain a message field with parameters for the crypto suite. A reply of a Tag contains a response field with the data returned by the crypto suite.

The crypto suites that are defined by ISO/IEC 29167 can be defined by their Crypto Suite Identifier (CSI). According to ISO/IEC 29167-1, the CSI for this crypto suite shall be defined as the 6-bit value 001001_2 and it is expanded to the 8-bite value 09_h for use by all air interface protocols in this annex.

Table E.1 shows the security services that are supported by this crypto suite.

Table E.1 — Security services

Security service	Method	Mandatory, optional, Prohibited, or not supported
Authentication		
Tag authentication (TA)	RAMON	Mandatory
Interrogator authentication (IA)		—
Mutual authentication (MA)	AES, after successful RAMON TA	Optional
Communication		
Authenticated Tag from TA	Authenticated communication (Tag => Interrogator)	—
	Secure authenticated communication (Tag => Interrogator)	—
Authenticated Interrogator from IA	Authenticated communication (Interrogator => Tag)	—
	Secure authenticated communication (Interrogator => Tag)	—
Authenticated Interrogator and Tag from MA	Authenticated communication (Interrogator <=> Tag)	Optional
	Secure authenticated communication (Interrogator <=> Tag)	Optional

E.2 Security services for ISO/IEC 18000-3 mode 1

A crypto suite supporting ISO/IEC 18000-3 Mode-1 shall fulfil the protocol security command requirements as defined in this annex. The crypto suite shall implement the following statements or equivalents with only allowing the provided choices. If no choice fits to the crypto suite, the crypto suite shall not be used for ISO/IEC 18000-3 mode 1.

Optional choices shall be accepted for 1-to-1 communication.

a) For Tag authentication, the Authenticate command shall be supported and the Challenge command may be supported. For Mutual authentication, the Authenticate command shall be supported, while the Challenge command shall not be supported.

b) The execution time for a Tag authentication shall be below 0,1 s. The execution time for a Mutual authentication shall be below 1 s.

c) The tag shall ignore commands from an interrogator during execution of a cryptographic operation.

d) The tag may support a security timeout following a crypto error. The length of the security timeout shall be defined by the tag manufacturer, depending on the application profile.

e) A Tag in any cryptographic state other than initial (i.e. state after power up) shall reset its cryptographic engine and transition to the ready state upon receiving an invalid command.

NOTE Invalid commands are crypto commands with incorrect UID or CRC error.

f) For each Error Condition defined in the crypto suite, the Tag shall remain in its current state.

g) The Tag shall remain in its current state after a Tag Authentication. The Tag shall transition to the selected secure state after a successful Mutual Authentication.

h) The KeyUpdate command may be supported.

i) The KeyUpdate command, if supported, shall be encapsulated.

In ISO/IEC 18000-3 mode 1, the *Authenticate* command can be initiated from the **Selected** state of the protocol state machine and leaves the Tag in the state it was before.

E.3 Security services for ISO/IEC 18000-63

A crypto suite supporting ISO/IEC 18000-63 shall fulfil the protocol security command requirements as defined in this annex. The crypto suite shall implement the following statements or equivalents with only allowing the provided choices. If no choice fits to the crypto suite, the crypto suite shall not be used for ISO/IEC 18000-63.

Optional choices shall be accepted for 1-to-1 communication. Since the tag is singulated and the TID is known, supported options can be derived from it.

a) For Tag authentication, the Authenticate command shall be supported and the Challenge command may be supported. For Mutual authentication, the Authenticate command shall be supported, while the Challenge command shall not be supported.

b) The execution time for a Tag authentication shall be below 0,1 s. The execution time for a Mutual authentication shall be below 1 s.

c) The tag shall ignore commands from an interrogator during execution of a cryptographic operation.

d) The tag shall support sending the contents of the response buffer in the reply to an ACK command.

e) The tag shall support sending the contents of the read buffer in the reply to a READ_BUFFER command.

f) The tag may support a security timeout following a crypto error. The length of the security timeout shall be defined by the tag manufacturer, depending on the application profile.

g) A Tag in any cryptographic state other than initial (i.e. state after power up) shall reset its cryptographic engine and transition to the open state upon receiving an invalid command.

NOTE Invalid commands are crypto commands with incorrect handle or CRC error.

h) For each Error Condition defined in the crypto suite, the Tag shall remain in its current state.

i) The Tag shall remain in its current state after a Tag Authentication. The Tag shall transition to the secured state after a successful Mutual Authentication.

j) The KeyUpdate command may be supported.

E.4 Communication example

E.4.1 Tag identification sequence in partial result mode

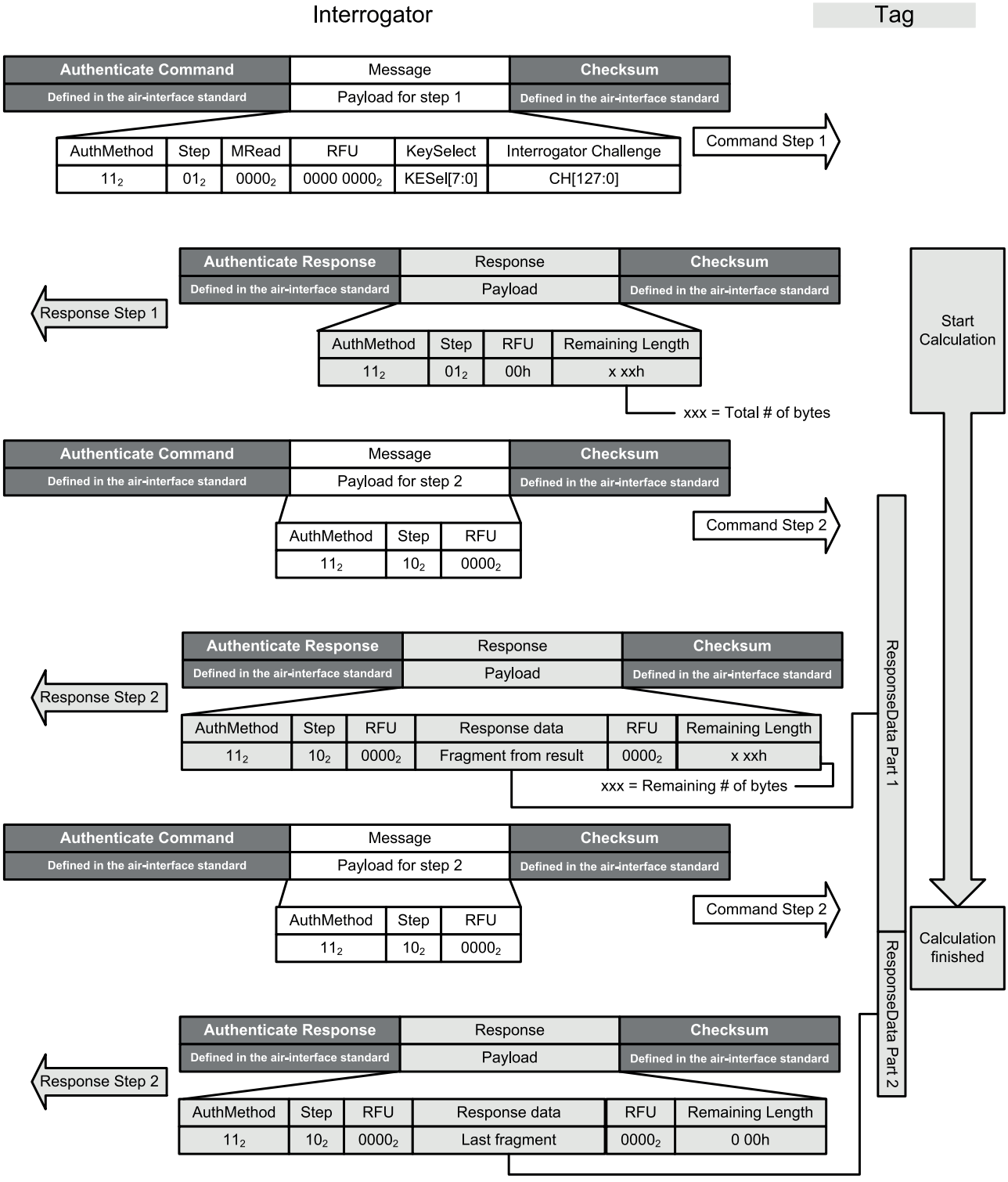


Figure E.1 — Example of a tag identification sequence in partial result mode

An example of a Tag Identification performed is given in Figure E.1. The Tag is using partial result mode to communicate with the Interrogator. The first Tag response data bytes are transmitted while the cryptographic calculation is still going on, followed by the second and final Tag response data bytes when the cryptographic calculation has been finished.

E.4.2 Tag identification sequence in complete result mode

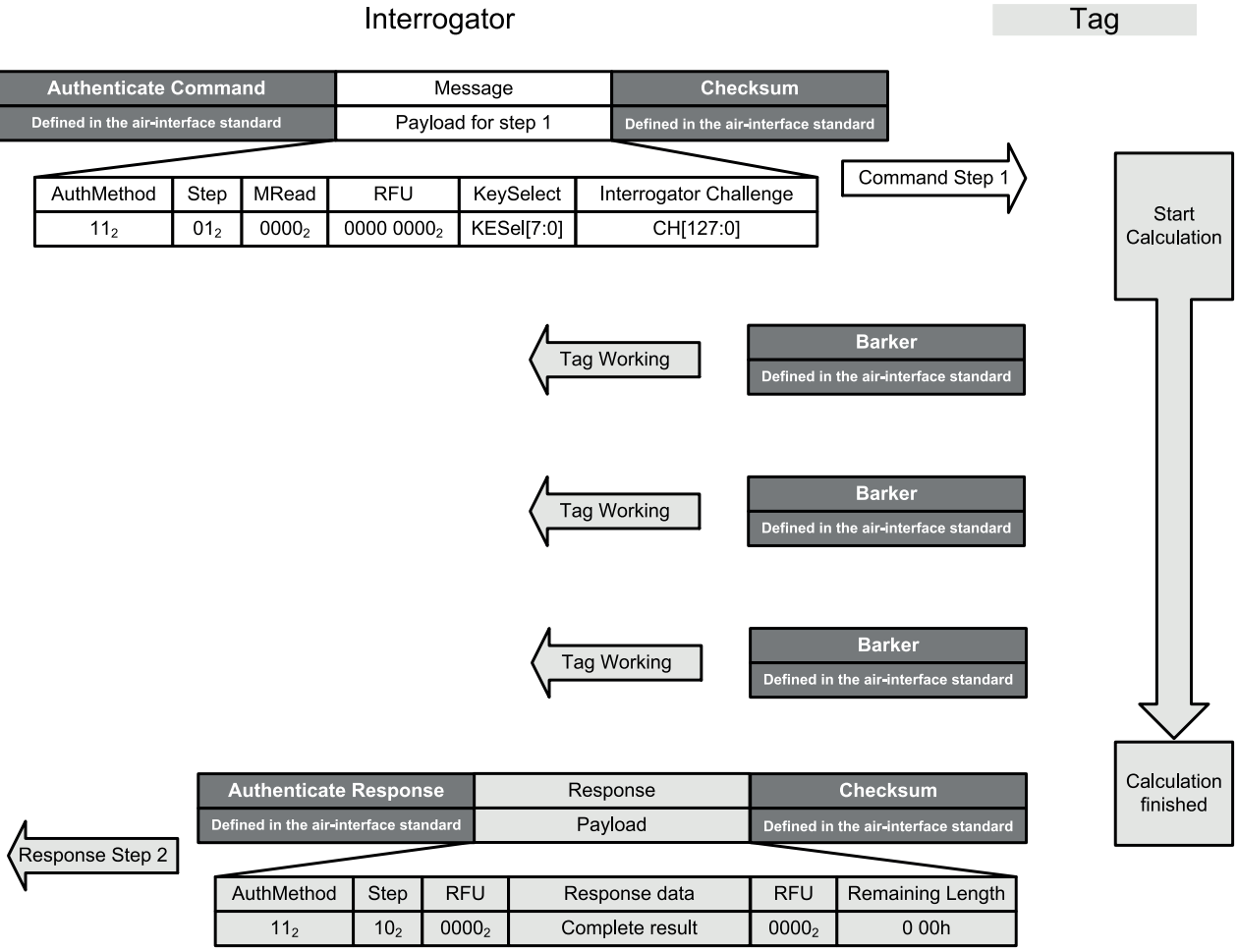


Figure E.2 — Example of a tag identification sequence in complete result mode

An example of a Tag Identification performed is given in Figure E.2. The Tag is using complete result mode to communicate with the Interrogator. The example shows an In-Process Tag reply, using Barkers as defined in ISO/IEC 18000-63. For other air interfaces, different methods than illustrated in this example may apply to indicate that a Tag is still working or to extend the waiting time of the Interrogator.

E.4.3 Mutual authentication sequence

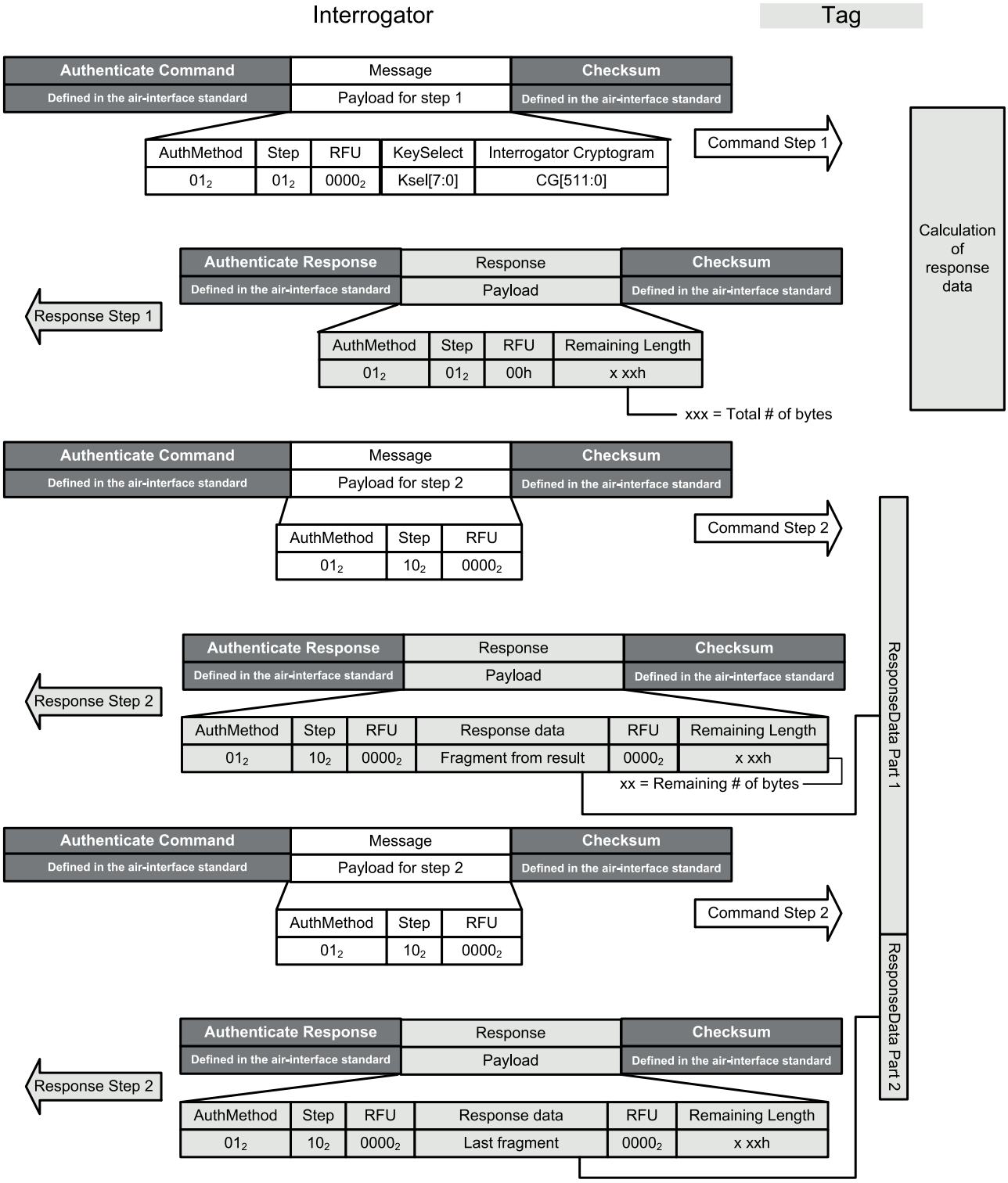


Figure E.3 — Example of a mutual authentication sequence

An example of a mutual authentication sequence performed is given in Figure E.3. The first Tag response data bytes are transmitted while the cryptographic calculation is still going on, followed by the second and final Tag response data bytes when the cryptographic calculation has been finished.

E.4.4 Secure Read and Write communication sequence

A detailed example of a secure Read or Write communication sequence is shown in Figure E.4.

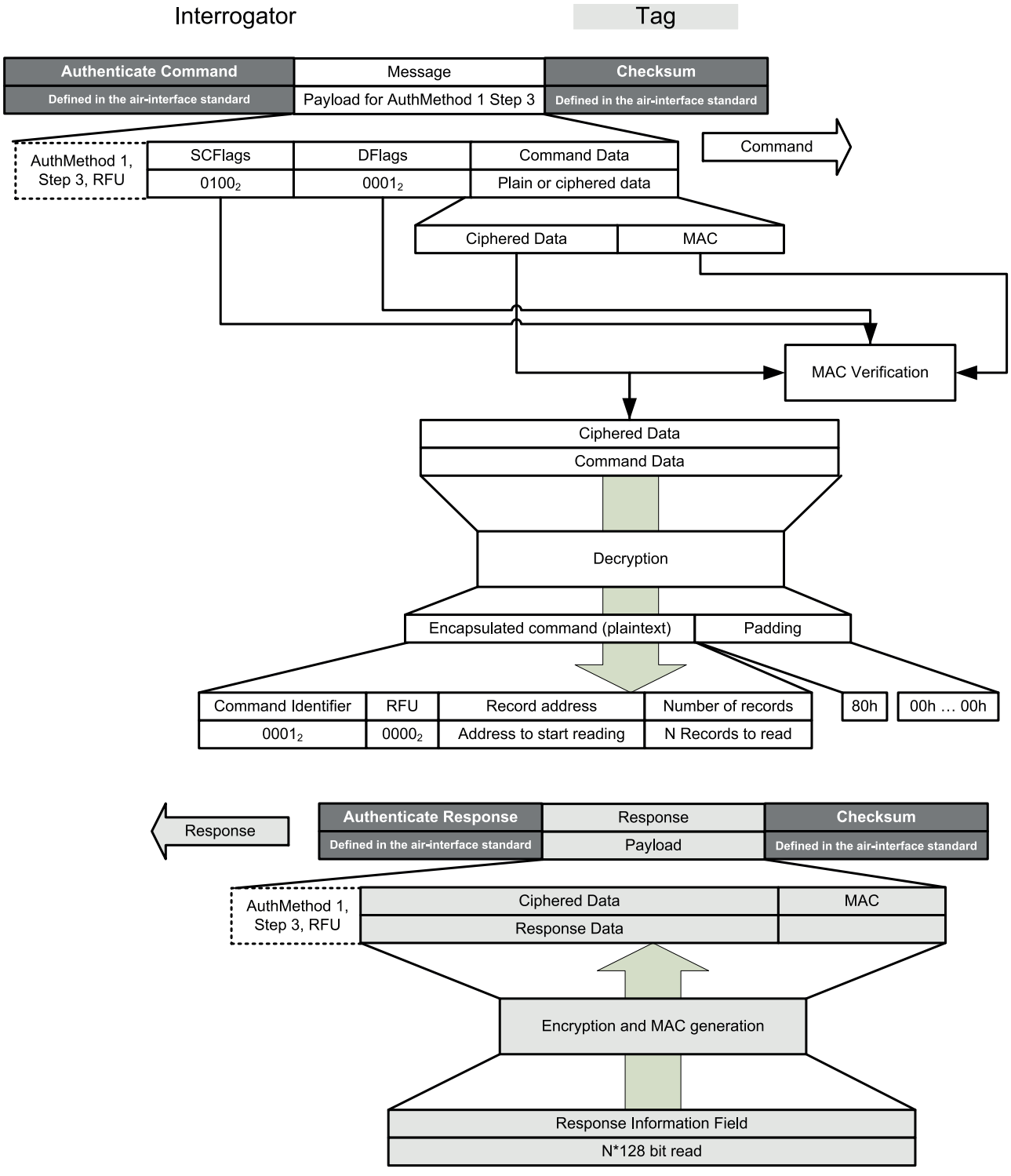


Figure E.4 — Example of a secure communication sequence, seen from the tag

E.5 Implementation of CS error conditions

E.5.1 General

This crypto suite specifies error conditions when any of the commands cannot be completed successfully. The error conditions of the crypto suite shall be returned to the Interrogator as error codes for the air interface.

E.5.2 Implementation of CS error conditions in ISO/IEC 18000-3 Mode-1

Table E.2 shows the conversion of Error Conditions in the crypto suite to ISO/IEC 18000-3 mode 1 error codes if the Tag supports specific error codes.

Table E.2 — Crypto suite error codes

Crypto Suite Error Condition	Description	ISO/IEC 18000-3 Mode-1 '*error code*', Error-Code Name
Other error	Miscellaneous error.	'0F': Error with no information given or a specific error code is not supported.
Not supported	The requested functionality is not supported by this Tag or by this CS.	'03': The command option is not supported.
Insufficient privileges	The interrogator did not authenticate itself with sufficient privileges for the Tag to perform the operation.	'40': Generic cryptographic error.
Memory overrun	The command attempted to access a non-existent memory location.	'10': The specified block is not available (does not exist).
Memory locked	The Tag memory location is locked and not writable.	'12': The specified block is locked and its content cannot be changed.
Crypto Suite error	Cryptographic error detected. This triggers a reset.	'40': Generic cryptographic error.

E.5.3 Implementation of CS error conditions in ISO/IEC 18000-63

Table E.3 shows the conversion of Error Conditions in the crypto suite to ISO/IEC 18000-63 error codes if the Tag supports specific error codes.

Table E.3 — Crypto suite error codes

Crypto Suite Error Condition	Description	ISO/IEC 18000-63 Error-Code Name
Other error	Miscellaneous error.	Other error
Not supported	The requested functionality is not supported by this Tag or by this CS.	Not supported
Insufficient privileges	The interrogator did not authenticate itself with sufficient privileges for the Tag to perform the operation.	Insufficient privileges
Memory overrun	The command attempted to access a non-existent memory location.	Memory overrun
Memory locked	The Tag memory location is locked and not writable.	Memory locked
Crypto Suite error	Cryptographic error detected. This triggers a reset.	Crypto Suite error

Annex F
(informative)

Non-traceable and integrity-protected Tag identification

F.1 Enabling non-traceability for ISO/IEC 18000-63 UHF Tags

F.1.1 General

To provide non-traceability, a tag may provide an *Untraceable* command as defined in ISO/IEC 18000-63:2015, 6.3.2.12.3.16. However, a tag which does only provide Tag Authentication cannot provide a secure channel required from the *Untraceable* command. To avoid using a constant password, which makes the Tag vulnerable to skimming attacks, at least one of the following procedures may be used.

— Use the *Untraceable* command, together with a session access password, provided within the TLV structure received at Tag Authentication. The encrypted response message in this case contains an access password to be used to transit a tag into secure state, which is required to perform an *Untraceable* command when there is no secure channel (requires Interrogator authentication) available. This session password is valid only one time to avoid replay attacks. See Table G.3 for more information.

— Use a partly hidden EPC/TID or a randomized EPC/TID. Randomizing EPC/TID is described in the subsequent subclauses. This method does not require the implementation of an additional *Untraceable* command, and therefore, suites for Tags with low resources. The randomized or hidden part of EPC/TID is provided by the encrypted response message, within the TLV structure received while Tag Authentication. See F.1.4 for additional information on how to recover randomized or hidden information.

F.1.2 Partially randomized EPC Memory

In ISO/IEC 18000-63-based applications, Tags are used to identify and track Tagged objects. UHF Tags carry an Electronic Product Code (EPC). The EPC is a code that identifies the object to which a Tag is affixed. The UHF Tag emits its EPC in plaintext which makes the Tag vulnerable to information leakage (and also cloning/emulation attacks). The EPC can be read without authorization and without being noticed by the person carrying the Tagged object (active access). The EPC can be captured by eavesdropping on the radio interface between Interrogator and Tag (passive access).

To satisfy the non-traceability property for Tags that use this crypto suite, an additional identifier — the SID — is introduced aside from the EPC defined in ISO/IEC 18000-63. In order to defend against tracking of individual Tags, the EPC is randomized at power up. The SID is sent in the RAMON-encrypted Tag authentication message and is instead used by the Interrogator to uniquely identify the Tag.

In order to prevent Tag tracing, a portion of the Tag's EPC is set to a random value that is updated at each power on (the CRC-16 of the EPC is also updated). The Interrogator is still able to inventory the Tag while anti-collision is maintained.

The EPC stored in a Tag according to ISO/IEC 18000-63 is located in the EPC Memory Bank. Table F.1 shows an example where a GTIN (Global Trade Item Number) is used as EPC. In this example, a portion of 32 bits of the Unique Serial Number is randomized at power up.

Table F.1 — A SGTIN-96 as an example of a randomized EPC of 96 bits

EPC Memory Address	MSB 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	LSB F
00-0Fh	Stored CRC (computed by the Tag at power up)															
10-1Fh	Stored PC															
	Length					**UMI**	**XI**	**T**	**RFU or AFI**							
	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
20-2Fh	EPC Header 30h (for SGTIN-96 Encoding Scheme)								Filter			Partition			GTIN [43:42]	
30-3Fh	GTIN [41:26]															
40-4Fh	GTIN [25:10]															
50-5Fh	GTIN [9:0]										Serial Number [37:32]					
60-6Fh	Serial Number [31:16] – **Random Bits**															
70-7Fh	Serial Number [15:0] – **Random Bits**															

F.1.3 Partially randomized TID Memory

In some applications, Extended Tag Identification (XTID) is used to provide more information to end users about the capabilities of tags. The XTID supports serialisation by adding a unique 48 bit IC Serial Number. This IC Serial Number can be read without authorization and without being noticed by the person carrying the tagged object (active access). The IC Serial Number also can be captured by eavesdropping on the radio interface between Interrogator and Tag (passive access). In order to defend against tracking of individual Tags, the IC Serial number is randomized at power up. Table F.2 shows an example where a portion of 32 bits of the Unique Serial Number is randomized at power up.

Table F.2 — Example of a randomized XTID serial number, located in the TID Memory

TID Memory Address	MSB 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	LSB F
00-0Fh	**Allocation Class (E2h)**								**Tag MDID MSB's**							
	1	1	1	0	0	0	1	0	X	S	F					
10-1F	**Tag MDID LSB's**				**Tag Model Number [11:0]**											
20-2F	XTID Header Segment [15:0]															
30-3F	IC Serial Number [47:32]															
40-4F	IC Serial Number [31:16] – **Random Bits**															
50-5F	IC Serial Number [15:0] – **Random Bits**															

F.1.4 Restore randomized or hidden EPC- or XTID Serial Numbers

In some applications, the information of the EPC- or XTID Serial Number hidden behind randomization is required to properly assign the Tag. This information may be provided by the encrypted TLV-Structure, adding dedicated TLV tags. This allows an Interrogator to restore the original EPC- or XTID Serial Number, if randomization was used. See G.3 and G.6.1 for more information.

F.2 Enabling non-traceability for ISO/IEC 18000-3, M1 HF Tags

In ISO/IEC 18000-3 mode 1 based applications, Tags (VICC) are used to identify and track Tagged objects.

The VICCs are uniquely identified by a 64 bits unique identifier (UID). This is used for addressing each VICC uniquely and individually, during the anticollision loop and for one-to-one exchange between a VCD and a VICC.

According to ISO/IEC 18000-3 mode 1, the UID shall be set permanently by the IC manufacturer. The UID comprises

— 8 MSB bits that shall be “E0”,

— IC manufacturer code, on 8 bits according to ISO/IEC 7816-6:1996/Amd.1, and

— a unique serial number on 48 bits assigned by the IC manufacturer.

The HF Tag emits its UID in plaintext, which makes the Tag vulnerable to information leakage (and also cloning/emulation attacks). The UID can be read without authorization and without being noticed by the person carrying the Tagged object (active access). The UID can be captured by eavesdropping on the radio interface between Interrogator and Tag (passive access).

To satisfy the untraceability property for Tags that use this crypto suite, an additional identifier—the SID—is introduced besides the UID defined in ISO/IEC 18000-3 mode 1. The UID is randomized at power up in order to defend against tracking of individual Tags. The SID is sent in the RAMON-encrypted Tag authentication message and is instead used by the Interrogator to identify the Tag.

In order to prevent Tag tracing, a portion of the Tag’s UID is set to a random value that is updated at each power on. The Interrogator is still able to inventory the Tag while anti-collision is maintained.

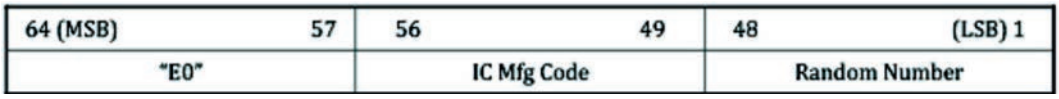


Figure F.1 — Randomized UID format

F.3 Signatures for integrity-protection of the SID

The signature should be generated over the SID, using Elliptic Curve Cryptography. Given the total length of 125 bytes that can be input into the MIX function (see C.1), Elliptic Curve Cryptography is most suited, since it allows for short signatures while maintaining high security strength.

The length of the signature — short enough to fit into the overall SID TLV record — depends on the selected curve. The signature is embedded in a separate TLV record as specified in C.1.

Annex G
(normative)

Description of the TLV record

G.1 General format of the TLV record

Depending on the command received by the interrogator, the tag prepares a byte block called the *TLV record*. The TLV record is embedded into the RAMON cryptogram, and the RAMON cryptogram is encrypted with the Rabin-Montgomery algorithm by the tag as described in Annex C. The byte length of the TLV record depends on the bit length k of the Rabin-Montgomery key used for encryption. Here k shall be ≥1 024 and divisible by 128. To simplify the following description, we define $m = k/64$. Then the TLV record has a length of precisely $6 \cdot m - 1$ bytes.

G.2 Composing the TLV record from TLV structures

The TLV record is a sequence of TLV structures. The coding of a single TLV structure shall be compliant with the Distinguished Encoding Rules (DER) of ASN.1 in ISO/IEC 8825-1[1]. A sequence of TLV structures is coded by simply concatenating the byte strings of the corresponding encodings of the TLV structures. The total byte length of the concatenated byte string may be at most $6\ m - 1$. If the concatenated byte string is shorter than $6\ m - 1$ bytes, a TLV structure encoding a *random filling* of maximum possible length shall be appended to the sequence of TLV structures. Details are given in G.4. The resulting byte string is called the *TLV record.* It is always exactly $6\ m - 1$ bytes long.

According to ISO/IEC 8825-1, a TLV structure is encoded as a sequence of bytes as described in Table G.1:

Table G.1 — Structure of a TLV encoding according to

Identifier bytes T	Length bytes encoding a length L	Content bytes (L bytes)

NOTE To keep the notation in this document consistent, the term "*octet*" used in ISO/IEC 8825-1 has been replaced by the term "*byte*", e.g. the term "*Identifier octets*" has been replaced by "*Identifier bytes*".

The *identifier byte* defines the type of a TLV structure. This standard uses primitive types of class "private" with ISO/IEC 8825-1 tag numbers 1 … 30 only. According to ISO/IEC 8825-1, any such type is encoded with exactly one byte. In this document, the type T of a TLV structure is the value of that single byte. Valid types for this document are given in Table G.3.

The *length bytes* specify the byte length L of the content of the TLV structure. Then the *Content bytes* are given as a byte string of L bytes length. According to ISO/IEC 8825-1, the length bytes are coded as described in Table G.2.

Table G.2 — Encoding of the length information *L* in a TLV structure

Length *L*	No of length bytes	Binary value of the length bytes (1,…,3 bytes)
0 ≤ L ≤ 127	1	`0`*`xxxxxxx`*
128 ≤ L ≤ 255	2	`10000001` *`xxxxxxxx`*
256 ≤ L ≤ 65 535	3	`10000002` *`xxxxxxxx xxxxxxxx`*

Here "*xxxx…xxxx*" is the binary representation of the number L.

G.3 Valid types of TLV structures

Table G.3 defines the types of a TLV structure that may be used in a TLV record. Depending on the type of the TLV record, these fields may be mandatory or optional; see G.6 for details.

Table G.3 — Valid types of TLV structures

Type	Feasible lengths L (Bytes)	Content
C1h	≥8	SID.
C2h	>0	Signature over SID. Its length depends on selected signature scheme and parameters, e.g. in case of using ECDSA with a 320-bit curve, the length is 2·40 bytes.
C3h	≥1	EPC Serial Number: contains at least that part of the EPC Serial Number, which hides behind randomization. Used to restore a (partly) randomized or hidden EPC.
C4h	≥1	XTID Serial Number: contains at least that part of the XTID Serial Number, which hides behind randomization. Used to restore a (partly) randomized or hidden XTID Serial Number.
C5h	4	Session access password. 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. See F.1 for additional information.
C8h	>0	Random filling data.
CAh	>0	Memory (general purpose field: any data, to be provided with the authentication).
CBh	>0	Data Identifier according to ASC MH 10. Data compressed by 6-bit-encoding.
CCh	32	SHA-256 hash value.
CDh	>0	Sensor Data.
CEh	>0	General purpose.
CFh	>0	General purpose.

NOTE The interpretation of the TLV structures (including the signature) is application specific.

G.4 Data Identifier according to ASC MH 10

A TLV structure, starting with CBh, allows appending a data string, coded and structured according to ANSI ASC MH 10.8.2[5]. The data string shall be compressed by using 6-bit-coding. A system identifier and a trailer shall not be contained. The lastbyte is padded with 0-bits.

The applied 6-bit-coding shall be done according to ISO/IEC 15962:2013, E.4.

An example for a TLV structure according to ASC MH 10 is shown below:

A possible, valid data string could be

"25SUN123456789PA12345<GS>4LUS<GS>16D20131108<GS>33LHTTPS://WWW.SECUREUID.COM/ITEMDATA/?ID=12345"

where "25S" is the data identifier and "<GS>" a group separator encoded as 011110_2.

At first, compression by 6-bit-coding according to ISO/IEC 15962 is applied:

2 110010

5 110101

S 010011

•••

5 110101

<GS> 011110

4 110100

•••

4 110100

5 110101

After concatenating the 6-bit blocks and padding with 0000_2, the resulting byte-string consists of 65 bytes. The complete TLV structure for this example is shown in Table G.4.

Table G.4 — Sample of a TLV-structure with 6-bit-coding

T	L	V
CB h	41h	CB 54 D5 3B 1C B3 D3 5D B7 E3 94 01 C7 2C F4 D5 ED 0C 55 37 B1 D8 4C B0 C7 3C 71 C3 87 B3 CC C2 14 51 04 FA BE F5 D7 5E E4 C5 0D 54 85 54 91 2E 0C F3 6F 25 41 4D 10 15 01 BF F2 44 F7 1C B3 D3 50 h

G.5 Appending a random filling TLV structure to the TLV record

A random filling TLV structure of type C8h shall be appended to a TLV record to pad the record to the required byte length of 6 *m* – 1, with *m* as defined in G.2. For simplicity, it is assumed that the total byte length of the sequence of TLV structures (without the random filling TLV structure) is 6 *m* – 1 – l for an integer l. This means that the sequence has to be extended by *l* bytes. Due to the encoding of a TLV structure, as described in G.2, a TLV structure of l byte length cannot be constructed for all relevant values of l. In some cases, an additional zero byte needs to be appended. Details are given in Table G.5:

Table G.5 — Encoding of the random filling TLV Structure

Number *l* of bytes to fill	Random filling TLV structure of type C8h			No of additional appended zero bytes
	Length of T	Length of L	Length of V	
l = 0	(None)			—
l = 1	(None)			1
l = 2	1	1	0	—
3 ≤ l ≤ 129	1	1	l-2	—
l = 130	1	1	l-3	1
131 ≤ l ≤ 258	1	2	l-3	—
l = 259	1	2	l-4	1
260 ≤ l	1	3	l-4	

The value *L* encoded in the length field is the byte length of V.

The tag manufacturer may store a fixed tag-individual random byte sequence on each tag.

G.6 Valid types of TLV records

G.6.1 Tag identification TLV record

The *tag identification* TLV-record shall at least consist of a SID and the random filling bytes, as shown in Table G.6. These components are encoded as TLV structures as described in G.2. If the optional signature is not present, its TLV structure has to be omitted completely. See 10.3.2 for additional information on the command coding.

Additionally to the required TLV structures, any other TLV structure as defined in this document can be used in a tag identification TLV record. See G.6.3 for additional information and G.7.2 for an example.

Table G.6 — Required TLV fields in the tag identification TLV record

Type	Content
"C1"	Mandatory.
"C2"	Signature over SID. Its length s 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.
"C3"	EPC Serial Number, optional field to provide the Serial Number in case randomization is used.
"C4"	XTID Serial Number, optional field to provide the Serial Number in case randomization is used.
"C8"	Random padding data, see G.4.

G.6.2 RAMON Memory Read TLV record

The RAMON encryption may be used to read out dedicated memory areas of the tag. This is useful when Mutual Authentication and Secure Communication are not implemented on the tag. See 10.3.2.2 for additional information on the command coding. To perform a memory read, the tag prepares the *RAMON memory read* TLV record. This TLV record shall at least consist of the Memory Content, and the random filling bytes, as shown in Table G.7. These components are encoded as TLV structures as described in G.2. The optional SHA256 hash value is calculated from the Memory Content. If the optional SHA256 is not present, its TLV structure has to be omitted completely.

Additionally to the required TLV structures, any other TLV structure defined in this document can be used in a memory read TLV record. See G.6.3 for additional information and G.7.3 for an example.

Table G.7 — TLV fields in the memory read message

Type	Content
"CA"	Memory content, mandatory field for memory read.
"CC"	SHA-256, optional field.
"C8"	Random padding data, see G.4

G.6.3 Additional Data fields

The RAMON encryption additionally may be used to read out sensor data and/or other dynamic or static information. This can be done by optionally inserting additional TLV-structures into the Tag identification or the memory read TLV structure. If present, the additional information shall be encoded in TLV structures of type C5h, CBh, CDh, CEh or CFh. These additional TLV-structures are defined as shown in Table G.3.

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.

G.7 Examples TLV records embedded into RAMON cryptograms

G.7.1 General

This section contains examples of TLV records and their embedding into RAMON cryptograms. The variable *m* has the same meaning as in C.1. For the examples in this clause, it is assumed that all TLV structures have length $L \leq 127$, so that encoded length information fits into a single byte.

G.7.2 Example of a RAMON tag identification cryptogram

Table G.8 shows an example of a RAMON cryptogram for an authentication message, where a single additional TLV-field to encode sensor data is used.

Table G.8 — Example of an authentication message with an additional sensor data field

	Padded Interrogator Challenge PCH_{I1}	**Tag Random Number** RNT	**TLV-coded SID**	**TLV-coded Signature**	**TLV-coded Sensor data**	**TLV-coded random filling**	**Zero Padding "00h"'**
			TLV record				
# of bytes	m	m	10	s	d	r = 6 m – 11 – s – d	1
Total # of bytes	8 m						

G.7.3 Example of a RAMON memory read cryptogram

Table G.9 shows an example of a RAMON cryptogram for a memory read message. Here a SHA256 hash value of the memory content is included.

Table G.9 — Components of the memory read message

	Padded Interrogator Challenge PCH_{I1}	**Tag Random Number** RNT	**TLV-coded Memory Content**	**TLV-coded SHA256**	**TLV-coded random filling**	**Zero Padding "00h"**
			TLV record			
# of bytes	m	m	y	34	r = 6 m – 35 – y	1
Total # of bytes	8 m					

Using the optional SHA256, a maximum of 6*m*-37-byte memory can be read by a single command. Not using the SHA256, a maximum of 6*m*-3-byte memory can be read by a single command.

Annex H
(informative)

Memory Organization for Secure UHF Tags

H.1 General

The secure memory of Secure UHF Tags, according to this document, is logically organized as a collection of up to 128 sectors of up to 4 096 bytes each, the substructure of which is explained below. The upper limit for the number of possible sectors is determined by the size of the key selector; however, it is obvious that there should be at least one single sector. Figure H.1 depicts an overview of the secure memory organization.

The public RAMON encryption key K_E is stored independently from the secure storage discussed in the main part of this annex.

H.2 The public key storage buffer

As explained, the public key storage buffer is independent from the Secure Memory. It is organized as an array of individual keys where the appropriate one is selected during the invocation of the RAMON authentication.

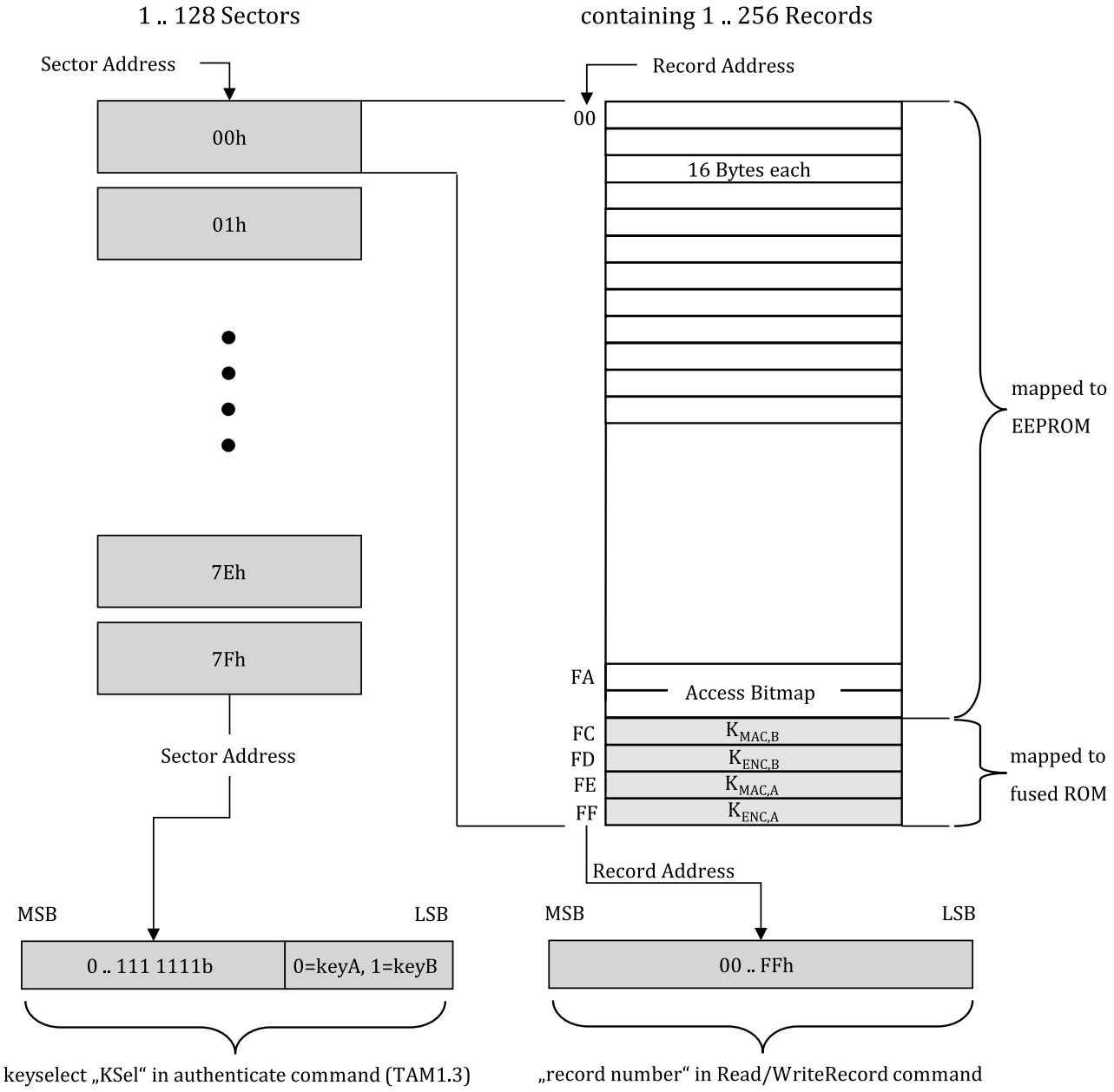


Figure H.1 — Secure memory layout

H.3 Sector selection

A sector is selected during the mutual authentication with a particular key set, identified by the key selector in the mutual authentication command. As each sector hosts two distinct key sets, A and B, the least significant bit is used to determine which one is to be used while the remaining bits of the key selector allow to distinguish up to 128 memory sectors.

Key selectors referring to non-existing sectors are considered RFU. If they are used inadvertently, an error condition results.

H.4 Structure of a secure memory sector

The total address space of a sector is 4 096 bytes or 4 Kbyte. This does not imply that all of this space shall be occupied with real memory.

The sector space is organized as records of 16 bytes each, thus 256 record addresses exist in one sector. A record is the smallest memory portion accessible for reading and/or writing.

As can be seen from Figure H.1, part of the memory is mapped to EEPROM, and another part is mapped to fusible ROM which can be written once during the personalization. Then the memory is protected from any modification after burning the fuses. The data in this write-once part of the sector comprises the two key sets for mutual authentication, $K_{MAC,A}$, $K_{ENC,A}$, $K_{MAC,B}$, and $K_{ENC,B}$.

The difference in these two key sets is in the associated access rights. Key set B grants read and write permission for all records in the current sector, with the exception of the fusible ROM area which is execute-only, while key set A permits to read all records but has write permission only for those records where explicitly granted by the Access Bitmap (explained below). As far as the fusible ROM area is concerned, key set A also grants only execution rights.

Figure H.2 explains the association between the access control bits and the individual records.

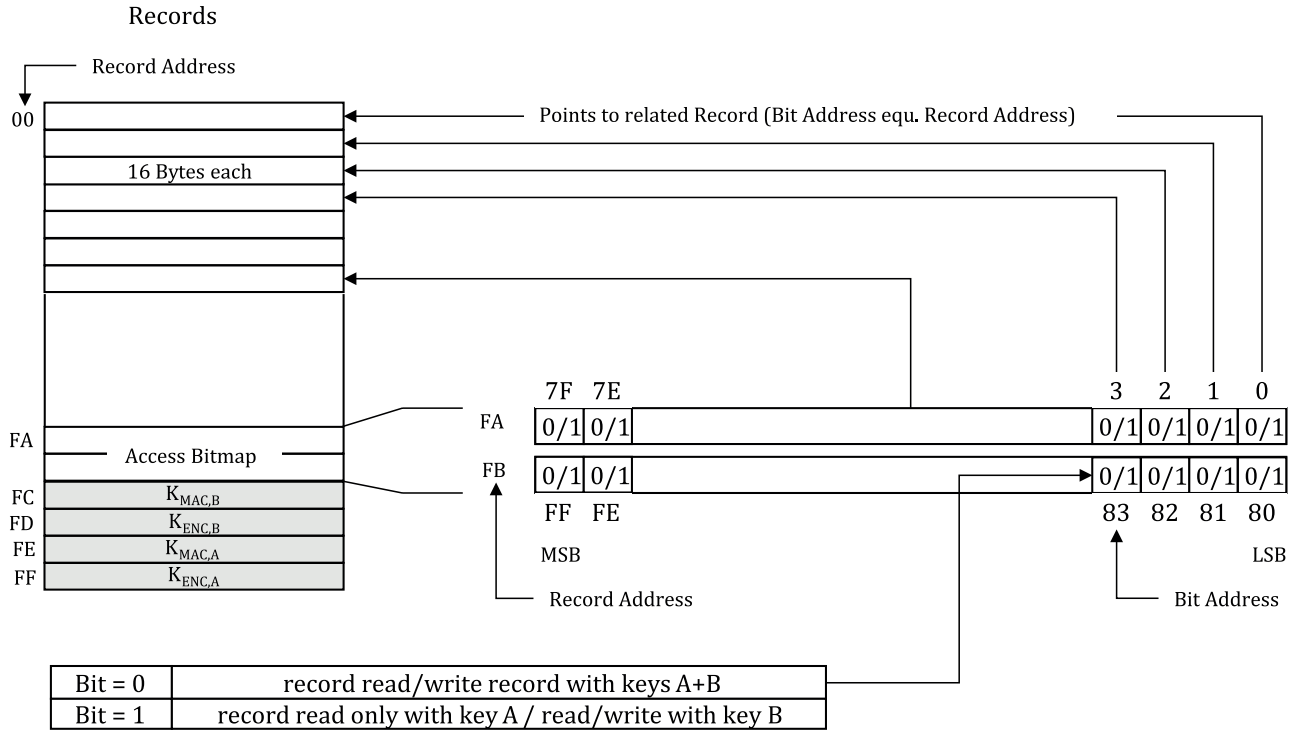


Figure H.2 — Access control bits

The Access Bitmap contains 256 write protection bits, one for each record in the sector, situated in records FA_h and FB_h. Record FA_H corresponds with the first 128 records, and record FB_h with the second 128 records. Within an access control record, the least significant bit (or the "rightmost") corresponds with the first record, and so on.

When an access control bit is set, the corresponding record is protected from any modification after authentication with key set A. Authentication with key set B is required to permit writing of that particular record. The protection mechanism also pertains to the Access Bitmap itself. Once the bit is set, the bitmap can only be changed after authentication with key set B. For the memory area which is mapped to fusible ROM, the Access Bitmap is meaningless, as it is write-protected after burning the fuses.

H.5 Record access and addressing

An individual record in the Secure Storage is accessed through the "leftmost" 8 bits of its absolute address (see Figure H.1). This scheme allows addressing 256 records of 16 bytes each and sums up to the complete sector size of 4 096 bytes per sector.

The memory layout allows holes in the address space as indicated by the dashed lines in Figure H.1. In case a non-existent record is accessed, the logic generates a suitable error code. Alternatively mirrored images of real records can be made visible in the address space. However, this may not lead to a violation of access conditions as stipulated by the access bitmap. The access conditions are always determined by the bits associated with the actual address of a record.

H.6 Access bit mapping

The bits in the access bitmap records are associated with the records beginning with the least significant byte, i.e. bit 0 of the least significant byte in the access bitmap record at FA_h is associated with record 00_h, bit 6 with record 06_h, and so forth. The same rules apply for access bitmap record FB_h: its least significant bit refers to data record 80_h, etc. As already mentioned, the state of the access bits for the access bitmap follow the same rules: When these bits are reset, the access control bitmap may be modified after authentication with any key set. But when these bits are set, only authentication with key set B permits modification.

H.7 Minimum accessible data unit

The minimum addressable data unit in this memory organization is a 16-byte record; therefore, reading or writing is permitted only in **multiples of 16 bytes**. This matches quite well with the block size of AES encryption. Modification of record fractions requires execution of a read-modify-write sequence.

Bibliography

[1] ISO/IEC 7816-4, *Identification cards — Integrated circuit cards — Part 4: Organization, security and commands for interchange*

[2] ISO/IEC 7816-6, *Identification cards — Integrated circuit cards — Part 6: Interindustry data elements for interchange*

[3] ISO/IEC 15946-2, *Information technology — Security techniques — Cryptographic techniques based on elliptic curves — Part 2: Digital signatures*

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

[5] ANSI MH10.8.2, *American National Standard — Data Identifier and Application Identifier Standard*

[6] Barker E. et al. *Recommendation for Key Management — Part 1: General (Revised)*, National Institute of Standards and Technology, NIST Special Publication 800-57, March 2007

[7] Lily Chen, Recommendation for Key Derivation Using Pseudorandom Functions (Revised), NIST: Special Publication (SP) 800-108, October 2009

[8] Morris D., Recommendation for Block, Cipher Modes of Operation, Methods and Techniques, NIST: Special Publication (SP) 800-38A, December 2001

[9] Morris D., Recommendation for Block, Cipher Modes of Operation: The CMAC Mode for Authentication, NIST: Special Publication 800-38B, May 2005

[10] GS1 EPCglobal, EPCTM Radio-Frequency Identity Protocols, Generation-2 UHF RFID, Protocol for Communication at 860 MHz – 960 MHz, Version 2.0.0 Ratified, November 2013

[11] Montgomery Peter L., Modular Multiplication without Trial Division. Math. Comput. 1985, **44** pp. 519–521

[12] NIST, FIPS Publication 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES), November 26, 2001

[13] NIST, SP 800-56B: Recommendation for Pair-Wise Key Establishment Schemes Using Integer Factorization Cryptography August 2009

[14] Rabin Michael O. *Digitalized Signatures and Public-Key Functions as Intractable as Factorization*. MIT-LCS-TR 212, MIT Laboratory for Computer Science, January 1979

ICS 35.040.50

Price based on 76 pages