Guide to Generating Application Load Units

MAO-DOC-TEC-009 v2.9
Copyright

© Copyright 1999 – 2017 MAOSCO Limited. This document contains confidential and proprietary information. No part of this document may be reproduced, published or disclosed in whole or part, by any means: mechanical, electronic, photocopying, recording or otherwise without the prior written permission of MAOSCO Limited.

Trademarks

MULTOS is a registered trademark of MULTOS Limited.
All other trademarks, trade names or company names referenced herein are used for identification only and are the property of their respective owners

Published by

MAOSCO Limited
1st Floor,
GPS House,
215 Great Portland Street,
W1W 5PN,
London,
United Kingdom

General Enquiries

Email: dev.support@multos.com
Web: http://www.multos.com

Document References

All references to other available documentation is followed by the document acronym in square [ ] brackets. Details of the content of these documents can be found on the MULTOS web site (http://www.multos.com).

Data References

All references to MULTOS data can be cross-referenced to the MULTOS Data Dictionary.
4.6.3 Application Provider’s Private Key ................................................................. 41
4.6.4 Application Providers Public Modulus and Exponent ................................. 42
4.6.5 Enciphered Application Signature ................................................................. 42
4.7 Creating the Confidential Application Load Unit ............................................... 43
  4.7.1 Application Load Unit .................................................................................. 44
  4.7.2 Application Load Certificate ........................................................................ 44
  4.7.3 Application Delete Certificate ..................................................................... 45
5 MULTOS Confidential ALUs – Enhanced Cryptography ................................. 45
  5.1 Plaintext Application Load Unit ..................................................................... 45
  5.2 Plaintext Application Unit .............................................................................. 45
  5.3 Enciphering Application Units ....................................................................... 45
  5.4 Retrieval/Authentication of MULTOS Card Public Key ................................. 45
    5.4.1 Recovering the Public Key ...................................................................... 46
    5.4.2 Authenticating the certificate .................................................................. 46
  5.5 Generation of Ciphertext KTU ....................................................................... 46
  5.6 Generation of the Application Signature ....................................................... 47
  5.7 Creating the Confidential Application Load Unit ........................................... 47
1 Introduction

This document has been aimed at MULTOS v4 developers and covers both the Legacy cryptographic scheme (RSA, DES, SHA-1 and AHASH) and the Enhanced cryptographic scheme (RSA, AES and SHA-2).

1.1 Overview

This section gives an overview of how Application Load Units are used within the MULTOS Scheme. To understand this, the whole loading and deleting process must be considered. The following diagram shows the MULTOS Scheme with respect to the loading of applications.

The MEL Application itself and the Application Provider’s Private Key are generated by the Application Provider and must be sent to the Application Load Unit Generator to build the Application Load Units. The Application Provider’s Private Key is required to generate a Protected or Confidential Application Load Unit. Protected and Confidential Application Load Units are described in detail later in this document.

The Application Header for the Application along with the Application Provider’s Public key is sent to the Card Issuer who must send these to the MULTOS CA as part of the Application Registration process. The MULTOS CA will use this information when generating the Application Load and Delete Certificates, which provide the main mechanism by which an application is authenticated during the loading process.

The Application Header contains a brief outline of the application’s main characteristics. The Application ID, Application Code Hash, Code Size and Data Size are the most important parts of this header information. Other characteristics specified in the Application Header relate to the type of application (standard or shell) along with whether the application is able to edit the ATR historical characters, the session data size, DlR record size and the size of File Control
The MCD Public Key and any Personalisation Data which is to be included in the Application Load Unit are sent to the Application Load Unit Generator. The MCD Public Key is required for the generation of Confidential ALU, which must be generated for a specific Target MULTOS Card.

A Confidential ALU is one that has been enciphered and can only be read by the Target MULTOS Card that contains the matching private key.

The Application Load Unit is essentially the MEL Application, supplied by the Application Provider, plus any personalisation data, provided by the Card Issuer. These can be protected with signatures and encryption if necessary. The Application Provider’s Private Key is used to sign a Hash Digest of the application. The MCD Public Key is used to provide the security in Confidential Application Load Units.

Although not directly connected with the actual generation of the Application Load Unit, the Application Load Certificate is closely related to the Application Load Unit and is required to load an application onto a MULTOS Card.

The Application Load Certificate, ALC, is a certified copy of the Application Provider’s Public Key along with the Application Header. The ALC is signed using the MULTOS CA’s Private Key Certifying Key (KCK) and all MULTOS Cards of a specific implementation are able to verify the authenticity of this certificate.

Example

The various entities described - Application Developer, Application Provider, Card Issuer, Application Load Unit Generator and Application Load Facility - may be separate, but in practice some of them will be carried out by the same entity. An example would be a bank, which operates as a Card Issuer. It subcontracts Application Development for a specific application to a software house. A separate card bureau performs the Application Load Unit Generation and the Application Loading. The bank retains the Application Provider function for the application so that it can generate confidential Application Load Units, and so retain control of the encryption process used in the generation. This bank wishes to issue a new card product using MULTOS with two applications on the card. One is an electronic purse supplied by an established payment organisation and the other is an application that the bank has contracted a third party software house to write.

A purse product has secret keys stored within it and these must be protected. In the case of this application the bank does not control the Application Provider key pair as the payment organisation has this responsibility on behalf of the scheme and assumes responsibility for the protection of the secret keys contained within the application.

The purse supplier, i.e. the payment organisation, acts as the Application Provider and generates the Application Load Units which are enciphered to protect the secret keys and to provide integrity throughout the process before they are loaded on the bank’s cards. Each of the Application Load Units will be specific to a single MULTOS Card, since parts of the unit will be encrypted with the cards public key.

The application written by the third party company is supplied to the bank in plaintext format. So the MEL application code is itself not protected from inspection by the bank, or anyone else. In the case
of this application the bank does generate the required Application Provider keys and produces the Application Load Units themselves.

The key point to note here is that the Application Provider keys may be generated by either the Application Developer, or the Card Issuer; the decision depends upon the degree of security required or mandated by each of the parties.

1.2 Loading Applications

This section is a brief overview of the process involved in loading applications onto a MULTOS Card. This process is described in detail in a separate document, Guide to Loading and Deleting Applications.

Load/Delete Certificates

The Application Load Unit (ALU) provides the application code, data, directory file entry (DIR), and File Control Information (FCI) for the application. This may be Unprotected, Protected or Confidential depending upon the security required during the loading process.

For example, within a secure environment an Unprotected Application Load Unit could be used, but in many cases a Protected or Confidential ALU will be required to provide security for the application’s code or data.

The Application Load Certificate (ALC) provides the ability for the MULTOS Card to verify that the Application to be loaded is valid and that the Card Issuer has authorised the loading of the Application.

The MULTOS Card itself must have been enabled for the Card Issuer by loading the MULTOS card enablement data, called MULTOS Security Manager (MSM) Controls Data. It is not possible to load any application until the card has been enabled. When a card is enabled it will be given its MULTOS Carrier Device (MCD) Private and Public Keys, which are used during the Application Load process to decipher any enciphered portions of the ALU.

The load process involves sending the ALU and ALC to the MULTOS Card. Beyond matching up an ALC, ALU and a MULTOS Card there is little processing required here. Any security must already be present within the ALU and ALC.
1.3 Types of Application Load Units

There are three types of Application Load Unit allowable within the MULTOS Scheme.

An Application Load Unit consists of three main components:

- **Application** - the application itself, which is divided into four smaller components
  - Code Record
  - Data Record
  - DIR Record
  - FCI Record
- **Application Signature** - enables the ALU to be protected against changes.
- **Key Transformation Unit** - enables the ALU to be enciphered.

The types are described below in order of generation of the various security protections, but since the creation of the Application Signature is always the last step (if it happens), the details are described, in later sections, in a slightly different order.

1.3.1 Unprotected ALU

An Unprotected ALU is a MEL application packaged into the ALU Structure. The MEL application may be produced using a compiler or assembler. It contains no Application Signature and therefore no protection against tampering or corruption. There is also no Key Transformation Unit so the Application Load Unit is plaintext, not enciphered, and therefore can be easily read.

The generation of an Unprotected Application Load Unit is simply a case of creating the correct file structure.

1.3.2 Protected ALU

A Protected ALU is an Unprotected ALU that has a digital signature, the Application Signature. The Application Signature is created using the Application Provider’s Private Key and therefore can only be generated by the Application Provider. This Application Signature allows any tampering or corruption of the Application Load Unit to be detected by a MULTOS Card. The Application Load Unit is not enciphered, however, and so can be easily read.

Appending the Application Signature of the application to an Unprotected ALU generates a Protected Application Load Unit.

1.3.3 Confidential ALU

A Confidential ALU has one or more sections enciphered. The details of this encryption, including the encryption keys used, are stored in a Key Transformation Unit (KTU). The Key Transformation Unit is enciphered using the MCD Public Key of the Target MULTOS Card. The Enciphered Application Unit is then signed generating an Application Signature.
1.3.4 Additional Notes

Each of the different ALU types are stored using the same file format, the difference between them lies in the presence of the Application Signature and the Key Transformation Unit for Protected and Confidential ALU. In Confidential ALU the data is just stored in enciphered form in the ALU.

Example

An issuer wishes to provide a card that has several applications loaded on it.

One application will always be present on every card and is loaded within a secure environment before cards are first issued. Typically this would be performed at a card-processing bureau. Since enciphering the application would slow down the loading process, the application is not actually enciphered but relies on the secure loading environment. It is the bureau’s operating procedures and security that now protects the ALU.

One application contains no secret data, but must be loaded within an insecure environment. For example, the application may be a loyalty application that is loaded within a supermarket. Although the supermarket is manned, it is not a secure environment. Since the Loyalty application contains no sensitive or secret data Protected ALU are generated. This will ensure that the application is not tampered with or corrupted. The ALU themselves can be easily read, but since they contain no secret information this is not important.

Two applications must be secure since they contain secret information and the customer may choose to dynamically load either onto the card. Confidential ALU are generated for these applications, which enables the customer to load one application within an insecure environment. Also, since the security is present within the ALU it is not possible for the customer or an eavesdropper to read or tamper with the application during the loading process.
2 ALU Generation Overview

The following diagram shows an overview of the process of generating an Application Load Unit.

The Application itself consists of the application code, data, directory file entry and file control Information. The Application Provider will supply this. The Application Data may contain initialisation values for the application, but will not usually contain any data that is specific to a particular customer.

Optional. The Customisation Data consists of data that is used to personalise an Application Load Unit for a particular consumer. In this case there must be one Application Load Unit generated for each target consumer / Target MULTOS card. The Customisation Data may be anything from a set of encryption keys specific to the MULTOS Card, or information about an actual consumer such as account details, or the consumer’s name.

There is no requirement for personalisation to occur during the ALU Generation stage, though it does allow secure loading of such information.

The Application Provider’s Private Key is part of an RSA Public/Private Key set. The Private Key is used during the generation of a Protected or Confidential ALU and therefore gives complete control to the Application Provider of who can read the final ALU.

The Private Key is used to generate the Application Signature. If there is no Application Signature than this key is not required. The presence of an Application Signature ensures that the ALU is protected against changes.
The MCD (MULTOS Carrier Device – smartcard) Public Key is required to generate Confidential Application Load Units. For an Application Load Unit to remain confidential it is important that only the Target MULTOS Card is able to read the ALU. This is achieved by enciphering the ALU and then enciphering the information required when deciphering the ALU with the Target MULTOS Card’s Public Key.

The information required to decipher the ALU is held in a Key Transformation Unit attached to the ALU. This KTU is enciphered and only the Target MULTOS Card has the corresponding private key with which to decipher the KTU and hence decipher the ALU.

The Application Load Unit generated may be either generic (plaintext or protected) and be usable for a large number of cards, or it may be specific to a single MULTOS Card, i.e. a Confidential ALU.

If the ALU is specific to a single MULTOS Card then there must be one ALU per Target Card. There is provision within the recommended file formats to store either a single ALU or many ALU.

### 2.1 Additional Notes

The MCD Public Key (the public key of the card) is available in a certificate returned by the MULTOS CA with the enablement data or by the card itself in response to an Open MEL command. This key must be extracted from the certificate and the corresponding Transport Key Certifying Key (TKCK) Public Key is required to do this. Please contact the MULTOS KMA to order this key.

**Example**

An Issuer requires Application Load Units (ALU) to be generated for use on the issuer’s web site. Cardholders must be able to download, via the Internet, the applications that they want on their cards. If the application has sensitive information stored within it then confidential ALU must be used. The Issuer uses a bureau to generate the Application Load Units.

To generate the ALU required, the Bureau must be given the Application’s compiled code and data itself, the Application Providers Private key, the MCD Public Key of each Target MULTOS Card and any customised data for each of the ALU.

In order for the Consumers to be able to use these ALU there must also be an appropriate ALC available and the consumer must have software capable of receiving the files over the Internet (e.g. a browser). A loader and card reader will also be necessary to load the ALU and Application Load Certificate (ALC) onto the card.
2.2 Public Keys

The security of Protected and Confidential Application Load Units and the integrity of the Application Loading and Deletion processes depend upon the management and handling of keys. Like any other system based upon cryptography, the security ultimately relies upon the keys and this section explains the importance of the keys and how each is used.

The following diagram shows an overview of the keys involved with the process of application loading, deleting and application load units.

The actual sizes of these keys are version or even implementation specific.

2.2.1 MCD Public Key

The MCD Public Key is required for the generation of Confidential Application Load Units. The Key Transformation Unit is enciphered using the MCD Public key of the Target card and only this specific Target MULTOS Card is able to decipher the KTU and therefore decipher any encrypted portions of the Application Load Unit (e.g. code and/or data).

The security of the Confidential Application Load Unit therefore relies upon the authenticity of the MCD Public Key. For this reason the MULTOS CA signs MCD Public Keys using the MULTOS CA’s Private Transport Key Certifying Key and the MCD Public Key is therefore made available in certified format.

For example, it is likely that the Application Load Unit Generator will not receive MCD Public Keys in plain text format. The MCD Public Keys are distributed as Certified Public Keys and these must be deciphered to validate their authenticity so that the Application Provider knows that they correspond to legitimate keys that came from the MULTOS CA.

The MULTOS CA will supply an Issuer with a copy of all MULTOS Public Key Certificates generated for their cards. When an Issuer requests Enablement Data for a new MULTOS Card the CA will return, as
part of the Enablement Data, a copy of all the specific MULTOS Public Key Certificates for all the MULTOS cards. It is the responsibility of the Issuer to retain this information and pass the certificates onto any other entities that may need them.

The Certificates are then deciphered using the MULTOS CA Public Transport Key Certifying Key (tkck_pk) to find out the public key for the card (multos_pk_certificate).

A MULTOS Card will return its Public Key Certificate in response to an Open MEL Application command. This may be more useful for dynamically loaded applications where the card is available to respond. For cases where ALU are being generated en mass then this method is unlikely to be practical and the list of public keys returned by the MULTOS CA with a set on enabled cards can be used instead, to pre-generate the ALU.

Note that the MULTOS KMA keeps a copy of every card specific public key every generated, and hence an Issuer can always request a copy from the CA after the event if they have lost their copy.

2.2.1.1 Recovery
The Messages corresponds to the entire Public Key Certificate and will consist of two portions. The most significant, left hand side, bytes will be plaintext, whilst the least significant bytes will be enciphered. There are always at least the Public Key Length (pkl) bytes in a message.

\[
\text{Plaintext Length} = \text{Certificate Length} - \text{pkl}
\]

2.2.1.2 Key Header Format
The following diagram shows the structure for the Key Header for the MULTOS Public Key Certificate.
Key Type: This is used to indicate the type of key contained within this certified public key. Different key types have different data fields included within the certificate.

Key Identifier: An eight byte identifier of a cryptographic key.

Certification Method ID: This is a single byte value which specifies the way in which the public key has been certified. In practice this will correspond to the key certifying key which has been used.

Hash Method ID: This is a two byte value which specifies the hashing algorithm and hash modulus (where needed) which is used for both the certification of keys and also for the checking of applications during loading.

Public Key Length: This is a two byte field which holds the length of the public key being certified.

Certifying Key Length: This is a two byte field which holds the length of the key certifying key used to produce the public key certificate.

Algorithm ID: This is a single byte field that identifies the algorithm which the certified key is intended to be used with.

Public Exponent Length: This is a double byte field that specifies the length of the Public Exponent.

Public Exponent: This is a four byte field that holds the Public Exponent. The Public Exponent should be aligned in the most significant bytes of this field.

MCD Issuer Product ID: This is a single byte field which specifies the Product ID assigned to the MULTOS Card for which this key is the Public Key.

MCD Issuer ID: This is a four byte field which specifies the Issuer ID of the Card Issuer who issued the MULTOS Card.

Set MSM Controls Data Date: This is a single byte field which specifies the MSM Controls Data Date for the MULTOS Card.

MCD Number: This specifies the eight byte MCD Number for the MULTOS Card.

Note that MULTOS version 4 also has the top bytes of the public key at the end of the header, if the length of the digest + the public key + fixed padding is greater than the TKCK modulus length.

See sections 4.4 and 5.4 for information on how the certificates are formatted, verified and public keys recovered.
2.2.2 MULTOS Transport Key Certifying Keys

The MULTOS Transport Key Certifying Keys are a pair of asymmetric keys, one public and one private. The MULTOS KMA will make the public key available upon request and this is used to recover and verify any MULTOS card certified public keys (multos_pk_certificate) produced by the MULTOS CA. The Private Transport Key Certifying Key remains secret and is known only to the MULTOS CA.

The security of a Confidential ALU relies ultimately upon the authenticity of the MULTOS Transport Key Certifying Key used, since the verification of the MCD Public Key relies on the use of the MULTOS CA Public Key Certifying Key. It is important to ensure that any MULTOS KMA Key Certifying Keys used are received by a trusted route and verified to be authentic.

The MULTOS KMA manages the supply of the MULTOS CA’s Public Transport Key Certifying Key to Application Providers.

The MULTOS KMA may have different Key Certifying Keys since keys may be replaced over time and new keys may be generated for new MULTOS implementations. The MULTOS Data Dictionary has a data item certification_method_id, which lists the id of the key used for all MULTOS implementations.

2.2.3 Application Provider Keys

The Application Provider must generate an asymmetric key pair if protected or confidential ALU are to be used. The private key is used for the generation of the Application Signature and the protection of the application relies on the security of this key. The public key is ultimately passed to the MULTOS Card via the MULTOS CA who embeds it securely in the Application Load Certificate where it is used to verify the application has not been corrupted or tampered with.

Note: The protection of the application does not include confidential information. The confidentiality of the application and its data relies upon the MCD Public Key as described earlier in this section. The Application Provider Keys are used to protect their application from corruption and tampering.

The Application Providers Private Key is used to generate an Application Signature and is required for all Protected and Confidential Application Load Units. The Application Provider’s Private Key will still exist for an Unprotected ALU, but will not be used and therefore can be left blank.

The Application Provider’s Private Key could be generated by either the Application Provider or the Issuer, depending upon who requires a secure ALU. The key must be passed on to the Application Load Unit Generator, who therefore must be a trusted party. It is common for the entity generating the Application Load Unit to be the same as the entity that generated the Application Provider’s Key.

The Application Provider’s public key should be the same length as the MCD Public Key and the MULTOS CA’s Key Certifying Key.

2.3 Summary

The MCD Public Key may be recovered from the MULTOS Public Key Certificate using the MULTOS CA’s Key Certifying Key.

The security of a Confidential Application Load Unit ultimately depends upon the authenticity of the MCD Public Key whose authenticity depends upon the MULTOS CA’s Public Transport Key Certifying key. It is important that all keys are trusted.

The Application Provider can also generate their own keys for signing applications.
2.3.1 Additional Information

There are additional details of the structure for public key certificates in the document File Interface Formats [FIF].
3 Generating Application Load Units

The process of generating an application load unit is summarised in the following diagram.

For simplicity an Unprotected Application Load Unit is regarded as being essentially the same as an Application. Technically an Unprotected ALU has a complete copy of the Code, Data, DIR Record and FCI Record defined within a fixed structure whereas an Application could be held in any proprietary structure without either a DIR Record or FCI Record. For example, an application may be the output from a toolkit.

3.1 Structure of Application Load Units

This section describes the structure of an Application Load Unit. All types of Application Load Unit use the same structure although certain components may be empty. For example, in Unprotected ALUs the Application Signature and Key Transformation Unit components will be empty and have a length of zero.

The Application Load Unit contains the complete application: Code, Data, DIR File entry and File Control Information. The details of the various components are shown in the diagram above. An Application Load Certificate is also required to load an application onto a MULTOS Card, to authenticate the load process. This is stored separately from the actual application normally.

Note: The record length fields in all of the files are stored in a bigendian manner, i.e. have their most significant byte first, and followed by the least significant byte. Unless stated otherwise, this applies to all two byte length fields within this document.

If a record is absent the record length is set to zero and the record data is empty.
3.1.1 MCD Number
An MCD Number is the unique identifier of a specific MULTOS Card. The MCD Number field of an ALU allows the ALU to be locked to only one card. The ALU may contain personalisation data (data specific to a certain user) or the public key of the specific MULTOS card may have been used to encipher portions of the application code and data and create the KTU (confidential ALU).

For Unprotected and Protected ALU, where the load unit will not be limited to one card, the MCD Number should be set to eight zeros.

3.1.2 Code Record
The Code Record contains the MEL Code of the Application. The length of the code is held in the first two bytes followed by the code itself. The code will normally have been produced using either an assembler or a compiler.

3.1.3 Data Record
The application data component holds a snapshot of the Static Segment that the application has available when first loaded. Please see the section on memory in the Application Abstract Machine chapter of the MULTOS Developers Guide for more details on memory segments.

3.1.4 DIR Record
The DIR Record for a file contains information about the name of the application when loaded on the card. At application load time the content of the DIR record is entered into the smart card DIR File by MULTOS. The DIR Record portion of the ALU is formatted using the first two bytes to hold the length, followed by the DIR Record itself.

Each MULTOS Implementation will have a maximum size for a DIR Record and it should be formatted according to the ISO 7816-5 standard.

3.1.5 FCI Record
The File Control Information (FCI) Record contains the information that is returned when a MEL application is selected. MULTOS stores the FCI Record and returns the information if required during a Select File command.

Each MULTOS implementation will have a maximum size for a FCI record and it should be formatted according to the ISO 7816-4 standard.

3.1.6 Application Signature
The Application Signature is a signed hash of the Application Unit and provides the ability to verify that the Application correctly loaded onto the MULTOS Card. It checks that the ALU has not been corrupted or tampered with. The Application Signature is stored within the ALU using the first two bytes to hold the length followed by the Application Signature itself. An Application Signature is present in Protected and Confidential ALU.

3.1.7 KTU
The Key Transformation Unit contains the information needed by the MULTOS Card to decipher any ciphertext within the ALU. The KTU itself is enciphered using the Target MULTOS Card’s Public Key. Only the Target MULTOS Card will have the corresponding private key to decipher the KTU.

Any ALU with a Key Transformation Unit is specific to a single MULTOS Card and only this card is able to decipher the KTU and thereby decipher the encrypted portions of the ALU.
3.2 Structure of the Application Unit

The Application Load Unit is the information that we send to the card. When the input reaches the card it is actually stored in a slightly different format, called the Application Unit.

So the Application Unit represents the structure that a MULTOS Card uses when an application has been loaded. It is way that the application is stored on the card. The following diagram shows the process of converting an Application Load Unit into an MULTOS 4 Application Unit.

![Diagram showing the process of converting an Application Load Unit into an MULTOS 4 Application Unit.]

The format of the Application Unit is unimportant in terms of the load procedure, but is important when we start looking at the generation of the KTU (and signature). When encrypting the code and data, we need to consider the fact that the decryption (done by the MULTOS card) of the loaded code and data is done in continuous blocks and does not consider the original nature of the data as it was loaded.

Therefore we must encrypt the hex block (code and/or data) as though it were on the card already. This means that the hex will be decrypted to valid code/data.

Example:
It is possible to encipher the whole of the code record, and the first portion of the data, in a block of enciphered data. A second block of data could be enciphered using a different key.

3.3 Generation of the Key Transformation Unit

This section details the process involved in generating a Key Transformation Unit, the mechanism for enciphering Application Load Units. If Protected, not Confidential ALU, are going to be used then this section can be ignored.

3.3.1 Overview

A Key Transformation Unit (KTU) contains information on how any enciphered portions of the Application Load Unit have been enciphered. As such, the first task in generating the KTU is to decide how the Application Load Unit is to be enciphered.

A Plaintext KTU is created which holds a list of areas enciphered along with the algorithm identifier and key(s) used to encipher the required area. This Plaintext KTU is then enciphered to generate the Ciphertext KTU.

The key used to encipher the Plaintext Key Transformation Unit is the Target MULTOS Card’s Public Key. This requires that the KTU is created specifically for a single MULTOS Card, the Target Card, and the Public Key of this card must be known. This Public Key is often referred to as the MCD Public Key.
3.3.2 Procedure

The following is an overview of the process involved in generating a Key Transformation Unit.

The secret portions of the Application Unit are enciphered and the regions, algorithm and keys used stored in the plaintext KTU. The Unprotected ALU is converted to an Application Unit which is used to perform enciphering on.

Plaintext KTU is enciphered using MCD Public Key. The enciphered application Unit is copied back into the ALU. The Ciphertext KTU is then copied to the ALU.

The Key Transformation Unit details which areas of the ALU code and data are enciphered and which keys where used. The KTU itself is therefore also enciphered in order to protect the keys contained within it.

Note that the Application Unit is not the same as the Application Load Unit (ALU). Although it contains essentially the same information the ordering of the information is important and they must be considered different. In section 3.2 the differences between the application load unit and the application unit were explained and these are going to be important because we will be enciphering parts of the application unit structure.

3.3.3 Plaintext KTU Overview

The Plaintext Key Transformation Unit is generated from information which defines the areas of the ALU which have been enciphered along with the algorithm identifier and key(s) used for each enciphered area. The following diagram shows the structure for the Plaintext Key Transformation Unit.
The following diagram shows how some of the fields apply to an Application Unit that either has been or will be enciphered. The sub-box within this next diagram shows the structure of each area descriptor.
3.3.3.1 KTU Header
The KTU header consists of the following components:

- **Initial Padding**: This is a single byte, set to ‘55’, which is present to ensure that the most significant bit of the Plaintext Key Transformation Unit is zero.
- **Set MSM Controls Data Date**: This is the byte that represents the date on which the MSM Controls Data for the Target MULTOS Card(s) were created. The byte corresponds to the number of months since January 1998.
- **MCD Number**: This is the identification number for the Target MULTOS Card. Each Key Transformation Unit must be created for a specific MULTOS Card and this identifies the card.
- **Application ID**: The Application ID of the application contained within the ALU (or Application Unit). This is held as three fields: Length of the AID, the actual AID and padding of ‘FF’ to make the length of the whole field equal to 17 bytes.
- **No. Area Descriptors**: The number of blocks that are to be enciphered. The maximum number of blocks depends upon the algorithms used and the size of the card’s Public Key.
- **Area Descriptors**: See below for details of an Area Descriptor structure.
- **Padding**: The Plaintext KTU must be padded such that its size is equal to the size of the MCD Public Key. This padding should be formed from random numbers.

There is a limit on the size of the KTU Ciphertext, which is the size of the MULTOS Card’s Public Key, e.g. 72 bytes. This limits the number of blocks that can be enciphered within the application unit itself.

The Plaintext KTU contains all of the information required to decipher the encrypted blocks of the Application Unit, including the keys used. It clearly must be protected to prevent this information from being seen. Enciphering the Plaintext KTU with the Public Key of the Target MULTOS Card does this.

3.3.3.2 Area Descriptors
An Area Descriptor refers to a block of the Application Unit that has been enciphered. The Area Descriptor stores the location of the block, the manner in which it was enciphered and the key(s) used. The following is a list of the fields in each Area Descriptor:

- **Algorithm ID**: This specifies the algorithm used to encipher the block. Currently only DES CBC (ID = ‘01’), multi-key DES CBC (ID = ‘02’) and AES CBC (ID = ‘06’) are allowed.
- **Area Start**: This is the starting byte of the encrypted area, relative to the start of the Application Unit.
- **Area Length**: This is the number of bytes in the block, and must be in multiples of 8 when doing DES CBC encryption and 16 for AES CBC encryption.
- **Key Data Length**: This holds the number of bytes in the key data. This is ‘08’ for single DES CBC, ‘10’ for 2-key triple DES CBC and ‘10’ or ‘20’ for AES CBC.
- **Key Data**: This holds the key(s) themselves.

3.3.4 Ciphertext Application Unit
The Application Unit must have the secret areas enciphered using the algorithm and keys specified in the Area Descriptors.
Note that the Application Unit is a representation of how MULTOS will see the application as it is loaded, and is therefore the structure which MULTOS will use when deciphering any enciphered areas. The whole process of enciphering the Application Load Unit is designed to reduce the processing required by the MULTOS Card and hence increase load speed.

Once the Ciphertext Application Unit has been generated the Ciphertext is copied back into the original Application Load Unit. The ciphertext will occupy the same space as the original plaintext. An Application Signature will always be required when a KTU is used and the Ciphertext Application Unit should be retained for calculating the Application Signature.

### 3.3.5 Ciphertext KTU

Enciphering the Plaintext KTU with the Public Key of the Target MULTOS Card generates the Ciphertext KTU, as shown in the following diagram.

For the registration of MULTOS version 4 applications a hash of the code must be sent to the MULTOS CA. If a KTU is used to encipher the code area of the application then a hash of this enciphered code area must be sent. This also means that the any area that is enciphered, and contains code, should not run over into the data area. If it does that changing any data will change the enciphered result, and hence change the code hash, requiring a new hash will have to be calculated and registered every time the data changes.

Note: In a perso bureau the target card is often not known at the time the ALU is generated. Typically in this situation the KTU is encrypted by the data preparation system using a symmetric transport key. The KTU is then re-enciphered using an HSM once the MCD public key has been read from the card on the perso machine.
### 3.4 Generation of the Application Signature

The Application Signature is a Signed Hash Digest of the Application Unit. By providing a Hash Digest of the Application it is possible for the MULTOS Card to verify that the application has not been corrupted or tampered with. By providing a Signed Hash Digest it is possible for the MULTOS Card to verify the authenticity of the Hash Digest itself. Please note that an application signature alone does not prevent the examination of the code and data within the Application Load Unit as, in that case, none of code and data are enciphered.

The key used to sign the Hash Digest is the Application Provider’s Private Key, which only the Application Provider knows. The Issuer, as part of the Application Load Certificate request, sends the Application Provider’s Public Key to the MULTOS KMA. This is included within the Application Load Certificate and the MULTOS Card can therefore verify the authenticity of the Application Signature itself.

#### 3.4.1 Procedure – Legacy cryptography

The following diagram is an overview to the process of generating an Application Signature for all MULTOS 4 devices prior to MULTOS 4.4, and later devices that do not implement Enhanced cryptography.

Note that the Application Unit is not the same as the Application Load Unit (ALU). Although it contains essentially the same information the ordering of the information is important and they must be considered different. In section 3.2 the differences between the application load unit and the application unit were explained and these are going to be important because we will be calculating the hash digest over the application unit structure.
3.4.1.1 Calculating the Hash Digest

The following diagram shows the process of calculating the Hash Digest for the Application Signature for a MULTOS 4 card.

The hash digest calculation is done in two steps. The first, using a fixed initial vector, results in a 16-byte hash digest value of the application unit. The second step uses this hash digest value as its initial vector. The input data for the second hash is a block of 0x55 equal to Application Provider’s key modulus length minus the size of the final AHASH digest length and also minus two 8-byte padding blocks; i.e., the length of the input data would be (application provider’s key modulus length – 32). So, in the case of a 128-byte modulus, the input data would be a 96-byte block of ‘55’.

The asymmetric hash algorithm is a chain block cipher that makes use modular exponentiation. So, the data must be handled in blocks. The block size is the hash modulus size minus the AHASH digest size; i.e., hash modulus size – 16. For this example, the modulus size is 72 bytes, which gives a block size of 56 bytes.

In order for the chain block modular exponentiation to work properly, it is necessary that the input data length be an integer multiple of the block size and that the integer representation of each block of the message be smaller than that of the modulus. To ensure that both conditions are met the initial input data block needs to have enough bytes of ‘55’ prepended so that the input data length is divisible by the block size.

In the case of the first step of calculating the hash digest for the AU Signature do not confuse the prepended ‘55’ bytes with the Hash function initial vector, which is a 16-byte block of ‘55’. In the case of the second step, the resulting input block will be one composed entirely of ‘55’, which is then divisible by the block size. To continue with the example of a 128-byte modulus the initial input data to the second step is 96 bytes of ‘55’. As 96 is not divisible by the block size of 56, the data must have enough bytes of ‘55’ prepended so that the input is a multiple of 56. So, by adding 16 bytes of 0x55 the resulting block is 112 bytes of ‘55’.

The Hashing Algorithm is an Asymmetrical Hash, described in the following section. The signing process involves exponentiation of the plaintext using the Application Provider’s Secret Key.
3.4.1.2 Hashing Algorithm
The Application Unit is hashed using the following MULTOS Hash Digest algorithm.

3.4.1.3 Split message into blocks
The length of the Hash Digest produced from the algorithm is fixed at 16 bytes.

The Hash Modulus is the key used to perform the RSA encryption and is set by the MULTOS CA. The 72-byte test hash modulus, with a public exponent of 3 is:

E45D04F70555C571C41262B961FCFE2E
EA1BD440BE6432DCE109E1B271E3E4F9
680B4573321EC95A0B236F4219E40B18
A936EE7502411D75FE8F7B34506DB0B5
563CFDE7CDBB52E1
The length of the blocks into which the message must be split may be calculated by subtracting the hash modulus size from the hash digest size. In this case the hash modulus size is 72 bytes and the hash digest size is 16 bytes. So, the block size is $72 - 16 = 56$ bytes. The following diagram shows the relationship between the length of the Hash Modulus, the 16 byte Hash Digest and the Block Length. The following diagram shows the sizes that must be used for the blocks within the hashing algorithm.

The Block Length is equal to the Length of the Hash modulus minus the size of the Hash Chain.

The size of the data block to be enciphered is equal to the size of the RSA Key: the Hash Modulus.

The size of the final Hash Digest, or hash, is equal to the size of the hash chain.

If the message is not a multiple of the Message Block Length then padding bytes of ‘55’ must be prepended to the message such that the resulting length is divisible by the block size.

The algorithm used to perform the hash may vary in later versions of MULTOS. There is no provision within the ALU to specify the algorithm used and therefore it must be derived from knowledge outside the scope of this document. When generating an ALU, the version of MULTOS for which the ALU is to be used must be known.

3.4.1.4 Encipher blocks

Each block in the message is then enciphered using the Hash Modulus. The block is concatenated to the current Hash Chain value.

Due to the limitation of the RSA calculation that the value of message (m) and modulus (n) should obey the rule: $m < n$, the Most Significant Bit (MSB) of the message is set to 0, since the MSB of the hash modulus is always 1. Therefore during (for the new Hash Chain value), and after enciphering (for the result), the Most Significant Bit in the result must be set to zero.

An Initial Value is used as the first block to be enciphered in the Hash Chain.

The last Hash Chain value generated is the final result of the asymmetrical hash algorithm.
### 3.4.1.5 Signing the hash

The Hash Digest and Padding is signed using the Application Provider’s Private Key using the RSA algorithm as shown below.

![Diagram of the signing process](image)

The Random Padding added to the Hash Digest protects the Application Provider’s Private Key from crypto-analysis. When the Application Signature is verified on the MULTOS Card the length of the Hash Digest is known and the Random Padding is simply discarded. In addition there should be 8 bytes of fixed padding added.

### 3.4.2 Procedure – Enhanced cryptography

The Enhanced method for calculating an Application Signature is significantly simpler for Application Providers to implement due to the use of SHA-256 or SHA-512 rather than the Asymmetric Hash algorithm. Note that the Application Provider key must be at least 72 bytes long and for security preferably more.

**Step 1: Generating the Application Signature plaintext**

<table>
<thead>
<tr>
<th>Application Unit</th>
<th>'55'</th>
<th>'80'</th>
<th>'00' padding as needed</th>
<th>8 random bytes</th>
<th>8 bytes '55'</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHA-256 / SHA-512</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step 2: Signing the plaintext**

![Diagram of the signing process](image)

### 3.4.3 Application Provider Private Key

The key used during the signing of the application signature plaintext is the Application Provider’s Private Key. The Application Provider must generate a pair of asymmetric keys. The Issuer must supply the public key to the MULTOS CA and it is then included within the Application Load Certificate. The
private key is not disclosed and is used to sign the application signature plaintext and so form the Application Signature.

Only a valid Application Load Certificate, with the correct Application Provider’s Public Key, is able to correctly verify an Application Signature, and only the Application Provider, who knows the Private Key, is able to generate an Application Signature.

3.4.4 Application Signature

Once the Application Signature has been generated it is inserted directly into the Application Load Unit.
4 MULTOS Confidential ALUs – Legacy Cryptography

This section contains a complete example of generating a confidential Application Load Unit for a test card with known keys.

It applies to all MULTOS 4 devices prior to MULTOS 4.4, and later devices that do not implement Enhanced cryptography.

In this example the entire data component of the application load unit is enciphered. In addition to this encryption, the application load unit will also have an Application Signature to prevent the application load unit from being corrupted or tampered with before being loaded onto a MULTOS Card.

4.1 Plaintext Application Load Unit

The following is the unprotected application load unit. This is formed from six components, and the application signature and ciphertext KTU components are empty. These shall be added later in the example.

**Code component**

```
3F 01 FF F3 70 00 09 03 05 6E 00 3F 01 FF F4 70
10 09 03 05 6D 00 26 21 02 28 01 5E 00 00 59 00
00 29 0E 40 06 00 40
```

**Data component**

```
4D 55 4C 54 4F 53 20 2D 20 74 68 65 20 63 6F 6D
70 6C 65 74 65 20 73 6F 6C 75 74 69 6F 6E 2E 20
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

**DIR record entry component**

```
61 0E 4F 01 0A 50 09 61 54 65 73 74 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

**FCI record component**

```
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
```

**Application Signature component**

```
{empty}
```

**Ciphertext KTU component**

```
{empty}
```
4.2 Plaintext Application Unit

The first step in the process is to convert the application load unit into an Application Unit. The Application Load Unit is the file format that is used to store the application in a file on a PC. The Application Unit is a snapshot of the application as the MULTOS Card will see it when loaded onto the card. All of the encryption and signature generation is carried out using the application unit and not the application load unit.

The application unit is generated using the binary data from the application load into the following structure:

```
code record
data record
dir record
fci record
```

The following shows the transformation of an application load unit into an application unit.

```
Application Load Unit
```

The Application Unit would consist of the following components.

**DIR record entry component**

```
61 0E 4F 01 0A 50 09 61 54 65 73 74 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

**FCI record component**

```
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
```

**Code component**

```
3F 01 FF F3 70 00 09 03 05 6E 00 3F 01 FF F4 70
10 09 03 05 6D 00 26 21 02 28 01 5E 00 00 59 00
00 29 0E 40 06 00 40
```

**Data component**

```
4D 55 4C 54 4F 53 20 2D 20 74 68 65 20 63 6F 70 6C 65 74 65 20 73 6F 6C 75 74 69 6F 6E 2E 20
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

© 2017 MAOSCO Limited. MAO-DOC-TEC-009 v2.9
MULTOS is a registered trademark of MULTOS Limited.
4.2.1 Combined Application Unit
The following is the original example application load unit once it has been converted to a MULTOS Version 4 Application Unit.

```
61 0E 4F 01 0A 50 09 61 54 65 73 74 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
3F 01 FF F3 70 00 09 03 05 6E 00 3F 01 FF F4 70
10 09 03 05 6D 00 26 21 02 28 01 5E 00 00 59 00
00 29 0E 40 06 00 40 4D 55 4L 54 4F 53 20 2D 20
74 68 65 20 63 6F 6D 70 6C 65 74 65 20 73 6F 6C
75 74 69 6F 6E 2E 20 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

4.3 Enciphering Application Units
For the purposes of this example, the data component of the application load unit will be enciphered. The following is a hex dump of the data component of the sample Application Load Unit we are considering:

```
4D 55 4C 54 4F 53 20 2D 20 74 68 65 20 63 6F 6D
70 6C 65 74 65 20 73 6F 6C 75 74 69 6F 6E 2E 20
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

The algorithm used to encipher this block of data is actually a DES Chain Block Mode decipher. There is some logic in this. Early versions of MULTOS could only encipher, so the decipher algorithm was performed off-card, so that the card need only encipher.

When the application is loaded the MULTOS Card will use a DES Chain Block Mode encipher function to decipher the plaintext. The enciphering may be performed using either single or triple DES chain block mode enciphering. The following diagram shows how a block of plaintext is encrypted (actually using the decipher algorithm) to form part of the ciphertext application unit.

```
4.3.1 The encryption process
The following are more detailed workings for the first three blocks of the chained decipher. The first block is a simple decipher.

```

MAO-DOC-TEC-009 v2.9 © 2017 MAOSCO Limited.
MULTOS is a registered trademark of MULTOS Limited.
Subsequent blocks are then XOR’d with the previous ciphertext block before being deciphered. 

\[ \text{DES Key} = 41AD8223A90BE2A1 \]

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES CBC Decryption of first eight bytes data</td>
<td>\textbf{6BC8592212B0495A}</td>
</tr>
<tr>
<td>2nd plaintext block XOR previous ciphertext block</td>
<td>\textbf{4BBC314732D32637}</td>
</tr>
<tr>
<td>DES CBC Decryption of result</td>
<td>\textbf{7AD431DA01BF7EAE}</td>
</tr>
<tr>
<td>3rd plaintext block XOR previous ciphertext block</td>
<td>\textbf{0AB854AE649F0DD1}</td>
</tr>
<tr>
<td>DES CBC Decryption of result</td>
<td>\textbf{190A89307E40D288E}</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

The generated ciphertext component is shown in \textbf{BOLD} in the table above, and in full here:

```
6B C8 59 22 12 B0 49 5A 7A D4 31 DA 01 BF 7E BE
19 0A 89 30 7E 40 D2 88 C2 00 AF D6 48 6E F6 E2
8B 0E 2E A9 69 C0 B1 46 B5 60 C4 10 C9 36 5C 00
C3 91 52 E6 06 6E EA 36 34 1F 9E 24 5C 0F 44 42
```

The enciphered data is placed back into the application unit. This is required for the generation of the application signature detailed later in this example.

**4.4 Retrieval/Authentication of MULTOS Card Public Key**

Before we can generate the Ciphertext KTU, we must retrieve and authenticate the MULTOS Card Public Key from the certificate returned by the Open MEL command.

The Public key Certificate may be obtained from the MULTOS Controls Data returned from the MULTOS CA to enable a MULTOS Card. Alternatively a MULTOS Card, once enabled, will return its public key certificate in response to an Open MEL Application APDU command. This example assumes that the certificate has already been obtained and demonstrates how the certificate can be authenticated and the key retrieved. This example also assumes that the public portion of the Transport Key Certifying Key (tkck_pk) used by the MULTOS CA to sign the certificate has also been obtained.

The public key certificate will have a Key Header and the Public Key itself. MULTOS has various different public keys (e.g. MULTOS Card PK, Application Providers PK), and the structure of the Key Header is specific to the particular public key type. However it will always contain information that relates to the use of the key. For example, a MULTOS Public Key Certificate will contain information about the MULTOS Card whilst a Application Providers Public Key Certificate will contain information about an application.

The plaintext version of the certificate consists of the length of the public key certificate, followed by the Key Header, followed by the Public Key, and followed by sixteen bytes of padding (eight random, eight fixed). The following is a hex dump of the entire certificate.
Raw MULTOS Public Key Certificate Hex Dump

```
00 A8 50 00 11 22 33 44 55 66 77 01 01 00 04 00
60 00 80 00 00 01 03 00 00 00 FF 11 00 00 01 00
FE F1 80 03 FF F3 FF F1 33 9B B7 7F F1 9E 35 9B
EE 9D 9B 98 64 74 12 1C C1 D0 48 56 A9 39 33 C1
79 33 CD CC 12 4B 30 BE AE 8F 0F C6 D1 9D D1 60
94 D4 DE C9 CD 32 53 27 4C F9 4B 98 7E A4 CF 80
DA 7D 58 49 F2 D0 C6 48 FB 75 BE 7A A3 44 7D D9
5E 36 BA 06 AC 1F FD 6E A7 A9 7B EE AB 0B EB 98
11 E8 33 AE 3C 5D 7D 4F 57 3C E9 A6 46 7C EC 5D
68 68 26 B9 68 D5 95 80 BA BF 4D 09 55 68 B4 0D
FD BC 2C 80 64 C5 2F 25
```

Formatted MULTOS Public Key Certificate Hex Dump

The following is a hex dump of the certificate with the different fields within the certificate separated.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>certified_public_key_length</td>
<td>00 A8</td>
</tr>
<tr>
<td>key_type</td>
<td>50 (1)</td>
</tr>
<tr>
<td>key_identifier</td>
<td>00 11 22 33 44 55 66 77</td>
</tr>
<tr>
<td>certification_method_id</td>
<td>01 01</td>
</tr>
<tr>
<td>hash_method_id</td>
<td>00 04</td>
</tr>
<tr>
<td>public_key_length</td>
<td>00 60</td>
</tr>
<tr>
<td>certifying_key_length</td>
<td>00 80</td>
</tr>
<tr>
<td>algorithm_id</td>
<td>00</td>
</tr>
<tr>
<td>exponent_length</td>
<td>00 01</td>
</tr>
<tr>
<td>public_exponent</td>
<td>03 00 00 00 00</td>
</tr>
<tr>
<td>mcd_issuer_product_id</td>
<td>FF</td>
</tr>
<tr>
<td>mcd_issuer_id</td>
<td>11 00 00 01</td>
</tr>
<tr>
<td>set_msm_controls_data_date</td>
<td>00</td>
</tr>
<tr>
<td>mcd_number</td>
<td>FE F1 80 03 FF F3 FF F1</td>
</tr>
<tr>
<td>(pk_top) + Ciphertext</td>
<td>(2)</td>
</tr>
</tbody>
</table>

(1) The key type value of ‘50’ indicates a MULTOS public key certificate
(2) The PK Top and Ciphertext value is:

```
33 9B B7 7F F1 9E 35 9B EE 9D 9B 98 64 74 12 1C
C1 D0 48 56 A9 39 33 C1 79 33 CD CC 12 4B 30 BE
AE 8F 0F C6 D1 9D D1 60 94 D4 DE C9 CD 32 53 27
4C F9 4B 98 7E A4 CF 80 DA 7D 58 49 F2 D0 C6 48
FB 75 BE 7A A3 44 7D D9 5E 36 BA 06 AC 1F FD 6E
A7 A9 7B EE AB 0B EB 98 11 E8 33 AE 3C 5D 7D 4F
57 3C E9 A6 46 7C EC 5D 68 68 26 B9 68 D5 95 80
BA BF 4D 09 55 68 B4 0D FD BC 2C 80 64 C5 2F 25
```

This is 128 bytes. We are unable to determine what the end of this certificate is at the moment. Until we know the size of the MULTOS KMA Transport Key Certifying Key, we can not determine how much of the end of the certificate is ciphertext and how much is pk_top.
4.4.1 Recovering the Public Key

Most (if not all) of the public key (modulus) is held within the ciphertext at the end of the certificate. To decipher this ciphertext requires the public part of the MULTOS CA Transport Key Certification Key (tkck_pk). The Public Exponent portion of the public key is held in the plaintext portion of the certificate, public_exponent, above.

The value of the tkck_pk used in this example is as follows:

**Public Exponent**

03

**Public Modulus (tkck_pk)**

```
B6  E7  AA  2B  4E  29  96  F1  A9  1E  A7  4F  49  7A  E4  AF
5E  C8  75  C2  88  FA  5F  16  70  26  66  F1  BB  FC  6C  5F
30  9C  1E  17  6A  C1  D0  23  8F  A6  A9  8E  63  42  7E  AA
D6  F5  E6  FF  54  0A  AB  CE  41  2E  74  78  A4  9B  93  AE
CA  E5  EF  E6  31  13  8A  49  45  D7  B2  27  C6  1A  62  20
74  2F  7F  24  12  61  77  FC  9C  15  01  C9  59  C3  34  C8
06  13  86  63  7F  36  DD  49  0C  2E  6E  33  C5  36  EF  9D
EC  CD  73  27  4B  27  13  5D  93  52  F7  1C  37  95  AB  A7
```

This key is the test tkck_pk and not a live tkck_pk.

This is 128 bytes which is used to decrypt the bottom of the ciphertext. The ciphertext portion of the MULTOS certificate is therefore, in fact, all of the ciphertext. The length of pk_top is zero.

Decrypting the Ciphertext we get:

```
14  55  F8  E6  A5  02  BA  4D  2C  1E  1A  E8  B8  1C  47  0C
C2  6C  29  42  65  A6  5C  94  F1  BC  91  16  BA  4A  60  10
EC  0F  91  AE  1F  8B  7E  E5  96  4D  BD  61  A2  22  25  6C
B8  1E  A1  2D  5D  66  07  E4  B7  51  AF  26  C4  AB  4A  D7
1A  19  3F  C1  5C  4A  A3  52  E1  1F  0A  C4  B9  0E  1F  84
AB  6D  20  AA  0E  49  8F  25  07  98  56  1B  46  73  9A  60
D6  18  48  C0  70  9E  B7  4D  C3  53  AC  FF  F0  66  19  A5
E9  E9  FA  0C  70  07  A2  5B  55  55  55  55  55  55  55  55
```
**4.4.2 Interpretation of decrypted certificate ciphertext**

The following is the interpretation of the plaintext. It is a concatenation of a 16 byte Hash Digest, the 96 bytes of the public key (modulus), 8 bytes of random padding and 8 bytes of fixed padding.

\[ \text{Plaintext} = \text{Hash Digest} || \text{Key} || \text{Padding} \]

**Hash Digest**

14 55 F8 E6 A5 02 BA 4D 2C 1E 1A E8 B8 1C 47 0C

**Whole of Card’s Public Modulus**

C2 6C 29 42 65 A6 5C 94 F1 BC 91 16 BA 4A 60 10
EC 0F 91 AE 1F 8B 7E E5 96 4D BD 61 A2 22 25 6C
B8 1E A1 2D 5D 66 07 E4 B7 51 AF 26 C4 AB 4A D7
1A 19 3F C1 5C 4A A3 52 E1 1F 0A C4 B9 0E 1F 84
AB 6D 20 AA 0E 49 8F 25 07 98 56 1B 46 73 9A 60
D6 18 48 C0 70 9E B7 4D C3 53 AC FF F0 66 19 A5

**Random Padding**

E9 E9 FA 0C 70 07 A2 5B

**Padding**

55 55 55 55 55 55 55 55

**4.4.3 Authenticating the certificate**

The Hash Digest is an Asymmetrical Hash of the certificate header. In order to authenticate the certificate the hash digest recovered from the certificate must match the digest of the header. Like the generation of the Application Signature, this is done with two asymmetric hashes but the 2nd Hash is done on public bottom, not a block of ‘55’.

**Plaintext Header**

00 A8 50 00 11 22 33 44 55 66 77 01 01 00 04 00
60 00 80 00 00 01 03 00 00 00 FF 11 00 00 01 00
FE F1 80 03 FF F3 FF F1

Note: There is no top of the Public Key to include here but with different key sizes there might be.

This prepended with bytes of ‘55’ to pad to a multiple of 56 bytes gives one block

55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00 A8 50 00 11 22 33 44 55 66 77 01 01 00 04 00
60 00 80 00 00 01 03 00 00 00 FF 11 00 00 01 00
FE F1 80 03 FF F3 FF F1
Prepending this with an initial value of 16 bytes of ‘55’ gives

55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
00 A8 50 00 11 22 33 44 55 66 77 01 01 00 04 00
60 00 80 00 00 01 03 00 00 00 FF 11 00 00 01 00
FE F1 80 03 FF F3 FF F1

Using the RSA key for the hash modulus function gives

1B 35 DB F5 23 AE 18 68 76 D9 38 77 A7 60 9A 34
D6 CF 53 7F 81 12 77 7A 42 7D 70 6D BD 05 8E B0
56 75 A3 EB 91 F2 FC 90 A9 05 3C 94 1C 85 A0 FD
2F 80 E9 43 60 67 D2 C7 83 2D C0 16 96 83 9A B9
E8 5D C5 8E 99 B8 8D DC

The first sixteen bytes, with MSB set to 0 is 1B35DBF523AE186876D93877A7609A34.

1st A-Hash
The 1st A-Hash of this is calculated to be

1B 35 DB F5 23 AE 18 68 76 D9 38 77 A7 60 9A 34

This value is then used as the initial value of the 2nd asymmetric hash, which is performed on the public modulus. Since the bottom of the public modulus is greater than 56 bytes (a single block) we must split it in to multiple blocks with the first prepended with ‘55’.

Note that this is not the same as Application Signature Generation, where the second hash is performed on a block of ‘55’.

So for this example the bottom of public split in to blocks of 56 prepended with ‘55’ gives us two blocks

55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
C2 6C 29 42 65 A6 5C 94 F1 BC 91 16 BA 4A 60 10
EC 0F 91 AE 1F 8B 7E E5 96 4D BD 61 A2 22 25 6C
B8 1E A1 2D 5D 66 07 E4

and

B7 51 AF 26 C4 AB 4A D7 1A 19 3F C1 5C 4A A3 52
E1 1F 0A C4 B9 0E 1F 84 AB 6D 20 AA 0E 49 8F 25
07 98 56 1B 46 73 9A 60 D6 18 48 C0 70 9E B7 4D
C3 53 AC FF F0 66 19 A5
The concatenated block is (Initial Value (1st A-Hash) + block 1) is

\[
\begin{align*}
&1B\ 35\ DB\ F5\ 23\ AE\ 18\ 68\ 76\ D9\ 38\ 77\ A7\ 60\ 9A\ 34 \\
&55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55\ 55 \\
&C2\ 6C\ 29\ 42\ 65\ A6\ 5C\ 94\ F1\ BC\ 91\ 16\ BA\ 4A\ 60\ 10 \\
&EC\ 0F\ 91\ AE\ 1F\ 8B\ 7E\ E5\ 96\ 4D\ BD\ 61\ A2\ 22\ 25\ 6C \\
&B8\ 1E\ A1\ 2D\ 5D\ 66\ 07\ E4
\end{align*}
\]

Performing RSA on the first block gives

\[
\begin{align*}
&97\ 5D\ 84\ 32\ 35\ 47\ 2B\ B2\ 22\ 4A\ 38\ 78\ 76\ B6\ 8F\ 9B \\
&3D\ 7F\ 33\ 72\ 64\ 81\ A4\ 31\ 81\ 2A\ 75\ 2B\ BD\ 49\ 49\ 46 \\
&A1\ 6B\ 79\ F1\ 7C\ AF\ 92\ 39\ 95\ BD\ 80\ 54\ 09\ 70\ 87\ B3 \\
&2C\ 2A\ 9C\ 13\ E3\ 40\ 5A\ 40\ C2\ C1\ 71\ C6\ A0\ 56\ DF\ D2 \\
&B6\ 67\ 84\ 3C\ 73\ 5F\ A4\ 67
\end{align*}
\]

We must set the MS bit to 0 so second IV is 175D843235472BB2224A387876B68F9B. Second concatenated block is

\[
\begin{align*}
&17\ 5D\ 84\ 32\ 35\ 47\ 2B\ B2\ 22\ 4A\ 38\ 78\ 76\ B6\ 8F\ 9B \\
&B7\ 51\ AF\ 26\ C4\ AB\ 4A\ D7\ 1A\ 19\ 3F\ C1\ 5C\ 4A\ A3\ 52 \\
&E1\ 1F\ 0A\ C4\ B9\ 0E\ 1F\ 84\ AB\ 6D\ 20\ AA\ 0E\ 49\ 8F\ 25 \\
&07\ 98\ 56\ 1B\ 46\ 73\ 9A\ 60\ D6\ 18\ 48\ C0\ 70\ 9E\ B7\ 4D \\
&C3\ 53\ AC\ FF\ F0\ 66\ 19\ A5
\end{align*}
\]

Performing RSA gives

\[
\begin{align*}
&14\ 55\ F8\ E6\ A5\ 02\ BA\ 4D\ 2C\ 1E\ 1A\ E8\ B8\ 1C\ 47\ 0C \\
&A2\ B6\ 3E\ D7\ 95\ DC\ DE\ B8\ 06\ 98\ 4E\ 6F\ 4E\ 9D\ E9\ F3 \\
&94\ A5\ DF\ 76\ 6B\ F4\ E5\ 96\ B8\ A6\ FD\ CE\ D3\ 16\ 56\ B5 \\
&1A\ E8\ 18\ 46\ CB\ B6\ D3\ 30\ 0F\ D4\ C8\ 09\ BF\ 8B\ 81\ C7 \\
&4F\ 8C\ F4\ 9C\ 95\ 10\ D1\ 3E
\end{align*}
\]

The first sixteen bytes, with MSB set to 0 is 1455F8E6A502BA4D2C1E1AE881C470C

2nd A-Hash

The second A-HASH value is

\[
\begin{align*}
&14\ 55\ F8\ E6\ A5\ 02\ BA\ 4D\ 2C\ 1E\ 1A\ E8\ B8\ 1C\ 47\ 0C
\end{align*}
\]

This value matches the Hash Digest retrieved from the Ciphertext Certificate and therefore the certificate is authentic and the extracted key can be used.
4.5 Generation of Ciphertext KTU

The Ciphertext KTU component of the application load unit must now be generated. This holds the information about what parts of the application unit have been enciphered and which algorithm and keys have been used to encipher the application unit.

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Padding</td>
<td>55</td>
</tr>
<tr>
<td>MSM Date</td>
<td>00</td>
</tr>
<tr>
<td>MCD Number</td>
<td>FE F1 80 03 FF F3 FF F1</td>
</tr>
<tr>
<td>Application ID</td>
<td></td>
</tr>
<tr>
<td>AID Length</td>
<td>01</td>
</tr>
<tr>
<td>AID</td>
<td>0A</td>
</tr>
<tr>
<td>AID Padding</td>
<td>FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF</td>
</tr>
<tr>
<td>No Area Descriptors</td>
<td>01</td>
</tr>
<tr>
<td>Area1 Algorithm ID</td>
<td>01</td>
</tr>
<tr>
<td>Area1 Start</td>
<td>00 67</td>
</tr>
<tr>
<td>Area1 Length</td>
<td>00 40</td>
</tr>
<tr>
<td>Area1 Key Data Len</td>
<td>08</td>
</tr>
<tr>
<td>Area1 Key Data</td>
<td>41 AD 82 23 A9 0B E2 A1</td>
</tr>
</tbody>
</table>

Note: When attempting to replicate this process the MSM Date and MCD Number must match those on the card. Not necessarily those values quoted here.

The following shows the above data after being assembled into a single block of data and padded to the 96 bytes length of the MULTOS Cards Public Key.

```
55 00 FE F1 80 03 FF F3 FF F1 01 0A FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
01 01 00 67 00 40 08 41 AD 82 23 A9 0B E2 A1 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Note: The padding must be random for the security of the ALU to be maintained. For the purposes of clarity a padding value of 0x00 has been used in this example.
The plaintext KTU is enciphered using the MULTOS Card’s Public Key (the Public Modulus and exponent from the last section). This key may be obtained from the MULTOS Cards Public Key certificate. For the card used in this example the value of the key is:

**Public Exponent**

03

**Public Modulus (multos_transport_key)**

```
C2 6C 29 42 65 A6 5C 94 F1 BC 91 16 BA 4A 60 10
EC 0F 91 AE 1F 8B 7E E5 96 4D BD 61 A2 22 25 6C
B8 1E A1 2D 5D 66 07 E4 B7 51 AF 26 C4 AB 4A D7
1A 19 3F C1 5C 4A A3 52 E1 1F 0A C4 B9 0E 1F 84
AB 6D 20 AA 0E 49 8F 25 07 98 56 1B 46 73 9A 60
D6 18 48 C0 70 9E B7 4D C3 53 AC FF F0 66 19 A5
```

The resulting ciphertext KTU is shown in the following hex dump. This is the actual component that is appended to the application load unit:

**Ciphertext KTU**

```
09 09 1C C6 7C BE 04 18 6A D6 B6 96 E3 8E C6 6B
18 B3 E3 B7 30 95 11 69 40 25 D5 46 E7 3E E3 39
A3 41 63 6C 2E 14 84 8A D0 A6 BE E2 0E A4 18 8E
20 EA 90 B0 EC EE 2A E4 24 D8 5E 85 E9 31 BE A7
05 4F 84 79 B0 D9 B1 07 36 0D C9 3A 2B C8 48 9A
58 24 36 A0 C6 49 7A 06 D2 8C 44 3F 68 02 EA 44
```

For the registration of MULTOS version 4 applications a hash of the code must be sent to the MULTOS CA. If a KTU is used to encipher the code area of the application then a hash of this enciphered code area must be sent. This also means that the any area that is enciphered, and contains code, should not run over into the data area. If it does that changing any data will change the enciphered result, and hence change the code hash, requiring a new hash will have to be calculated and registered every time the data changes.
4.6 **Generation of the Application Signature**

The encrypted area(s) replace the plaintext in the application unit. The application signature is then generated using the encrypted area and not on the original plaintext application unit.

The following is a hex dump of the application unit after the data component has been enciphered. In this case the last 64 bytes of the application has been ciphered.

```
61 0E 4F 01 0A 50 09 61 54 73 74 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
3F 01 FF F3 70 00 09 03 05 6E 00 3F 01 FF F4 70
10 09 03 05 6D 00 26 21 02 28 01 5E 00 00 59 00
00 29 0E 40 06 00 40 6B C8 59 22 12 B0 49 5A 7A
D4 31 DA 01 BF 7E BE 19 0A 89 30 7E 40 D2 88 C2
00 AF D6 48 6E F6 E2 8B 0E 2E A9 69 C0 B1 46 B5
60 C4 10 C9 36 5C 00 C3 91 52 E6 06 6E EA 36 34
1F 9E 24 5C 0F 44 42
```

The first task is to generate the Asymmetrical Hash of the application unit.

For MULTOS version 4 cards, the Hash Digest is calculated in two steps. The first is to perform an A-Hash on the application unit. The next step is to use the result from this A-Hash as the Initial Value for a second A-Hash. The second A-Hash is applied to a block of ‘55’.

We need to split the above block (Application Unit) into blocks of 56. If we need to pad then they need to be placed at the beginning of the block. The pad byte is ‘55’.
Above we have 167 bytes of application data, which is $3 \times 56 - 1$ bytes. In other words we will need 3 blocks and 1 byte of ‘55’ padding. As such we end up with:

**Block 1**

55 61 0E 4F 01 0A 50 09 61 54 65 73 74 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 01 02 03 04 05 06 07
08 01 02 03 04 05 06 07

**Block 2**

08 01 02 03 04 05 06 07 08 3F 01 FF F3 70 00 09
03 05 6E 00 3F 01 FF F4 70 10 09 03 05 6D 00 26
21 02 28 01 5E 00 00 59 00 00 29 0E 40 06 00 40
6B C8 59 22 12 B0 49 5A

**Block 3**

7A D4 31 DA 01 BF 7E BE 19 0A 89 30 7E 40 D2 88
C2 00 AF D6 48 6E F6 E2 8B 0E 2E A9 69 C0 B1 46
B5 60 C4 10 C9 36 5C 00 C3 91 52 E6 06 6E EA 36
34 1F 9E 24 5C 0F 44 42

### 4.6.1 The Calculation

The encipher part of the Application Hash is done using the MULTOS Cards Hash Modulus (Exponent always 3). This key may be obtained from the MULTOS KMA. For the card used in this example the value of the key is:

**Public Exponent**

03

**Public Hash Modulus**

C2 D3 75 94 54 C6 6B 0B F1 2D 5C C1 EE 3F A8 FF
9D 7E 58 4A 39 0A 12 0E 8E 40 E1 32 9D 6E 40 0E
58 F8 34 94 9D 37 F4 BD 0D 64 37 46 26 E9 72 31
F2 3E 4F 5D 31 19 9F 33 35 25 A2 44 6F AE EC 75
3B 12 B7 39 91 68 C4 99
We have an initial chain block of 16 bytes of ‘55’, which we concatenate with the first block.

**Initial chain block + Block1**

```
55 55 55 55 55 55 55 55 55 55 55 55 55 55 55 55
55 61 0E 4F 01 0A 50 09 61 54 65 73 74 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 01 02 03 04 05 06 07
08 01 02 03 04 05 06 07
```

From this the following ciphertext was generated:

```
22 EE 95 D6 9E 53 5F AF 57 69 35 40 7C 63 21 8E
8A DA 47 76 A7 8B 76 26 FC C4 34 8C 81 D9 11 72
CD 95 08 EE D5 34 FC EE B6 30 49 95 C9 98 87 99
CE 88 B0 C4 A8 5A 94 2F B8 CF 6F F3 26 5F B6 0D
D4 8B 87 F7 3C FF DF A7
```

The first 16 bytes of this ciphertext will make the new chain value, BUT first we must make sure that the most significant bit (msb) is zero. In our example 22 is 00100010 which has the msb set to zero already. Let us suppose however that the first byte were 9D. This is binary 10011101 so we convert it to 00011101, which equals 1D. So our new initial value, when appended to block two, would have the first byte as 1D, not 9D.

**Ciphertext IV + Block 2**

```
22 EE 95 D6 9E 53 5F AF 57 69 35 40 7C 63 21 8E
08 01 02 03 04 05 06 07 08 03 0F 01 FF F3 70 00 09
03 05 6E 00 3F 01 FF F4 70 10 09 03 05 6D 00 26
21 02 28 01 5E 00 00 59 00 00 29 0E 40 06 00 40
6B C8 59 22 12 B0 49 5A
```

This generates ciphertext:

```
35 82 52 3A 10 F7 F7 72 F3 07 07 7B 33 30 A3 8A
BC 0E 9F C6 3B 6C 43 EE AF A7 F6 93 2C A6 BE 7E
31 65 86 62 1A C6 78 E8 21 5E 4D 3E 7A F2 2A 7C
19 4D BD B6 64 1D 94 A4 ED E8 62 DA 15 EE 59 BA
A2 60 97 A8 C0 D0 B1 37
```

The first 16 bytes are used as the IV for the next step in the calculation. The most significant bit is already set to zero. This gives the following value.

**Ciphertext IV + Block 3**

```
35 82 52 3A 10 F7 F7 72 F3 07 07 7B 33 30 A3 8A
7A D4 31 DA 01 BF 7E BE 19 0A 89 30 7E 40 D2 88
C2 00 AF D6 48 6E F6 E2 8B 0E 2E A9 69 C0 B1 46
B5 60 C4 10 C9 36 5C 00 C3 91 52 E6 06 6E EA 36
34 1F 9E 24 5C 0F 44 42
```

This generates ciphertext:
First A-Hash
From this we can determine the Hash Digest of the Application Unit, from the first 16 bytes. We must also make sure that the msb is 0. So the 1st A-Hash is:

\[
\begin{array}{c}
07 \ 6A \ DE \ 39 \ 4A \ 05 \ D9 \ 38 \ 8C \ DB \ 97 \ D2 \ 99 \ 90 \ 7C \ BA \\
FA \ A7 \ 5B \ C9 \ 76 \ AB \ 84 \ 70 \ BF \ 87 \ C8 \ 24 \ 7B \ 1B \ 0F \ DF \\
4A \ B3 \ 09 \ 50 \ 7D \ FB \ 70 \ 47 \ 21 \ 25 \ 10 \ 95 \ DC \ F7 \ BB \ 39 \\
D3 \ B9 \ 17 \ 27 \ E3 \ E8 \ 05 \ 82 \ 11 \ D5 \ 49 \ DC \ 6A \ F1 \ 57 \ 26 \\
17 \ 78 \ 42 \ CF \ 5D \ 96 \ A2 \ B5 \\
\end{array}
\]

To generate the second A-Hash we add 56 bytes of ‘55’ to the 1st A-Hash

1st A-Hash + 56 bytes ‘55’

\[
\begin{array}{c}
07 \ 6A \ DE \ 39 \ 4A \ 05 \ D9 \ 38 \ 8C \ DB \ 97 \ D2 \ 99 \ 90 \ 7C \ BA \\
55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \\
55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \\
55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \ 55 \\
\end{array}
\]

This generates ciphertext:

\[
\begin{array}{c}
A2 \ CA \ C3 \ 58 \ A0 \ A7 \ 4F \ FB \ 4A \ CE \ 9D \ 2F \ 83 \ 43 \ 06 \ 55 \\
12 \ A6 \ 2F \ A9 \ F0 \ 24 \ EA \ AE \ CE \ 9B \ E4 \ 50 \ D1 \ 0A \ 85 \ 79 \\
44 \ 1C \ 1F \ C0 \ DD \ 2A \ E5 \ 23 \ 95 \ 99 \ 09 \ 75 \ FC \ E3 \ B8 \ 67 \\
5D \ 66 \ 1A \ 13 \ 28 \ B4 \ 48 \ 04 \ 5C \ A7 \ 98 \ 02 \ 6B \ 2E \ FC \ 96 \\
AE \ 0D \ 6E \ AD \ E8 \ 00 \ 5C \ EF \\
\end{array}
\]

Second A-Hash
So the 2nd and actual Application Hash (having converted the most significant bit to 0) is:

\[
\begin{array}{c}
22 \ CA \ C3 \ 58 \ A0 \ A7 \ 4F \ FB \ 4A \ CE \ 9D \ 2F \ 83 \ 43 \ 06 \ 55 \\
\end{array}
\]

Pad the hash to 72 bytes to get the plaintext Application Signature
4.6.2 Plaintext Application Signature

Note: For version 4 cards the padding are 40 bytes of 0x55 followed by 8 bytes of random data, followed by 8 bytes of ‘55’.

This is then RSA enciphered using the Application Providers Private Key.

4.6.3 Application Provider’s Private Key

The following is the Application Provider Private Key used to encrypt the plaintext Application Signature.

Application Providers Private Exponent

The private exponent is:

```
92 48 C8 E9 35 7E 1D 3A B8 4E 0B 5A 07 95 04 81
C8 38 7A 0A E1 3E 89 6D 90 3E 86 1B DA FB 44 F0
79 00 17 E7 37 42 CC 35 26 C9 04 13 42 31 D3 58
F5 69 4B CA D9 4A 0A E4 5B F6 EC 32 22 8D 9F AA
E3 20 D2 F7 85 C0 BD FB
```

Expressed in CRT Components:

```
DP
9B F4 A9 1D C4 66 37 D8 70 C3 A8 F7 A6 3A C6 6C
E7 57 01 DE 1B BC 2B 85 33 49 C5 EC 77 74 0D 22
12 E8 B6 A3

DQ
A0 15 3F 35 45 45 28 3B 99 24 A6 FA 1D 6B 44 DC
26 0C 68 42 3E F9 62 49 32 5D D5 A6 F0 32 F3 B1
F5 63 BF 0F
```
4.6.4 Application Providers Public Modulus and Exponent

The APPK modulus is:

```
DB 6D 2D 5D D0 3D 2B D8 14 75 11 07 0B 5F 86 C2
AC 54 B7 10 51 DD CE 24 58 5D C9 29 C8 78 E7 68
B5 80 23 DC AC F3 0E CC 48 AE 96 3A F2 27 34 F0
15 97 02 9D DA 04 2F 87 0D 82 B7 00 CC 4F D8 DD
70 2B BD B1 55 13 CD 83
```

The Public exponent is 3.

With this key we can encipher the plaintext application signature to get the actual Application Signature.

4.6.5 Enciphered Application Signature

The plaintext application signature once encrypted is:

```
27 C7 28 91 0C 2E 78 98 76 E7 AC 0D A1 45 FB 52
A6 31 C5 F0 B7 21 73 3B 3C 2E 69 C4 09 F2 59 A0
C6 0B B4 4E 34 21 58 13 8B E0 80 6D F8 34 C1 FA
17 47 40 83 68 09 8D D6 FE 1E D5 E1 26 A1 C4 98
70 AB FC FD 4C 26 4A B5
```
4.7 Creating the Confidential Application Load Unit

The final stage is to recreate the Application Load Unit. This is a merely a case of copying the Application Unit back into the Application Load Unit and adding the Ciphertext KTU and Application Signature.

Code component (unchanged)

```
3F 01 FF F3 70 00 09 03 05 6E 00 3F 01 FF F4 70
10 09 03 05 6D 00 26 21 02 28 01 5E 00 00 59 00
00 29 0E 40 06 00 40
```

Data component (enciphered)

```
6B C8 59 22 12 B0 49 5A 7A D4 31 DA 01 BF 7E BE
19 0A 89 30 7E 40 D2 88 C2 00 AF D6 48 6E F6 E2
8B 0E 2E A9 69 C0 B1 46 B5 60 C4 10 C9 36 5C 00
C3 91 52 E6 06 6E EA 36 34 1F 9E 24 5C 0F 44 42
```

DIR record entry component (unchanged)

```
61 0E 4F 01 0A 50 09 61 54 65 73 74 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

FCI record component (unchanged)

```
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08
```

Application Signature component (as above)

```
27 C7 28 91 0C 2E 78 98 76 E7 AC 0D A1 45 FB 52
A6 31 C5 F0 B7 21 73 3B 3C 2E 69 C4 09 F2 59 A0
C6 0B 4E 34 21 58 13 8B E0 80 6D F8 34 C1 FA
17 47 40 83 68 09 8D D6 FE 1E D5 E1 26 A1 C4 98
70 AB FC FD 4C 26 4A B5
```

Ciphertext KTU component

This component is enciphered with MULTOS card’s public key.

```
09 09 1C C6 7C BE 04 18 6A D6 B6 96 E3 8E C6 6B
18 B3 E3 B7 30 95 11 69 40 25 D5 46 E7 3E E3 39
A3 41 63 6C 2E 14 84 8A D0 A6 BE E2 0E A4 18 8E
20 EA 90 B0 EC EE 2A E4 24 D8 5E 85 E9 31 BE A7
05 4F 84 79 B0 D9 B1 07 36 0D C9 3A 2B C8 48 9A
58 24 36 A0 C6 49 7A 06 D2 8C 44 3F 68 02 EA 44
```
4.7.1 Application Load Unit

The Application Load Unit once the components are combined in the correct order is as below.

The Application Load Units KTU section may need to be changed to match the MCD Number of the card being used. The Load and Delete certificates will still be valid though.

FE F1 80 03 FF F3 FF F1 00 27 3F 01 FF F3 70 00 09 03 05 6E 00 3F 01 FF
F4 70 10 09 03 05 6D 00 26 21 02 28 01 5E 00 00 05 00 00 29 0E 40 06 00
40 00 6B C8 59 22 12 B0 49 5A 7A D4 31 DA 01 BF 7E BE 19 0A 89 30 7E
40 D2 88 C2 00 AF D6 48 6E F6 E2 8B 0E 2E A9 69 C0 B1 46 B5 60 C4 10 C9
36 5C 00 C3 91 52 E6 06 6E EA 36 34 1F 9E 24 5C 0F 44 42 00 20 61 0E 4F
01 0A 50 09 61 54 65 73 74 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 20 01 02 03 04 05 06 07 08 01 02 03 04 05 06 07 08 01
02 03 04 05 06 07 08 01 02 03 04 05 06 07 08 00 48 27 C7 28 91 0C 2E 78
98 76 E7 AC 0D A1 45 FB 52 A6 31 C5 F0 B7 21 73 3B 3C 2E 69 C4 09 F2 59
A0 C6 0B B4 4E 34 21 58 13 8B E0 80 6D F8 34 C1 FA 17 47 40 83 68 09 8D
D6 FE 1E D5 E1 26 A1 C4 98 70 AB FC FD 4C 26 4A B5 00 60 09 09 1C C6 7C
BE 04 18 6A D6 B6 96 E3 8E C6 6B 18 B3 E3 B7 30 95 11 69 40 25 D5 46 E7
3E E3 39 A3 41 63 6C 2E 14 84 8A D0 A6 BE E2 0E A4 18 8E 20 EA 90 B0 EC
EE 2A E4 24 D8 85 E9 31 BE A7 05 4F 84 79 B0 D9 B1 07 36 0D C9 3A 2B
C8 48 9A 58 24 36 A0 C6 49 7A 06 D2 8C 44 3F 68 02 EA 44

4.7.2 Application Load Certificate

01 22 43 11 11 11 11 11 11 11 11 11 11 11 01 01 01 00 48 00 80 00 00 01 03 00
00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 27 00 40 00 08 00 00 20 00 20 00 20 00 50 50 00 01 00 44 F9 13 21 16 8C
47 4F F7 A3 49 72 F8 D9 FC 7C DF 47 E1 6E F9 99 90 5A 0D 72 15 20 4C 0A
38 D9 06 12 89 D3 D5 0C E5 24 26 9B 68 91 14 EB C6 3A 37 D7 AB 7B FE EF
5A 04 28 1D 3A C6 59 D8 48 A2 43 C9 77 53 1B AF BB 42 D2 D8 AC 9B 01 77
C0 1B FD F5 BB 90 1D 67 F9 51 36 03 87 BF 97 51 AD 8F DA 5D 20 2F DC 17
52 2F 73 7C 75 35 31 78 5A CD 2A D1 AF AA CB 9D 5A CC 11 EE 0F 01 95 8E
1F 0F
4.7.3 Application Delete Certificate

```
01 11 44 11 11 11 11 11 11 11 11 11 01 01 01 00 48 00 80 00 00 01 03 00
00 00 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
01 0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
6C 1A 54 B7 1A 8B 92 95 5B 44 EA D8 AF AC D1 AE D8 76 F4 EB D5 42
8C E6 C1 E9 92 83 17 D4 C0 51 CE C3 E7 AB CC FC 8D AC D8 99 56 ED E1 B9
FB 66 F4 78 89 6D BF CB 32 43 78 6C 10 0A 95 35 70 39 97 4C C6 31 98 6F
01 D4 D8 B6 91 C5 43 8B 5B C4 C9 4B 2F 81 8A FD A1 4B 92 42 73 35 07 61
30 6A 3A 11 F0 06 73 11 15 3D 0A 22 8C 05 A5 96 6D 4B 8B 07 47 64 A5 CB
CA F1 57 24 E8 65 42 80 BB
```

5 MULTOS Confidential ALUs – Enhanced Cryptography

This section mirrors section 4 but is for devices (MULTOS 4.4 and later) implementing Enhanced Cryptography. Calculations are not shown as they are largely repetitive, the import information is to note where the algorithms are different (and much simplified).

5.1 Plaintext Application Load Unit

This is unchanged. See 4.1.

5.2 Plaintext Application Unit

This is unchanged. See 4.2.

5.3 Enciphering Application Units

This is where the first difference occurs compared to the Legacy Cryptography scheme. AES CBC encipher must be used with the Enhanced Cryptography scheme. The block length for AES is 16 bytes so that data must be a multiple of that.

5.4 Retrieval/Authentication of MULTOS Card Public Key

The general process is identical to that in section 4.4 other than the card public key (mkd_pk) certificate is signed differently. The overall process is however much simpler than in the legacy scheme thanks to the use of SHA-256 or SHA-512 as the hashing method, instead of AHASH.
Public Key signing process as applied to the card public key:

* mk_pk_top is only present if there is not enough room for all of the public key in the signed part of the certificate.

The certificate has the same format as the “Formatted MULTOS Public Key Certificate Hex Dump” table in section 4.4.

The hashing method used is indicated by the hash_method_id. A value of ‘08 00’ indicates SHA-256, ‘0C 00’ indicates SHA-512.

The resulting certificate is

<table>
<thead>
<tr>
<th>Length excluding digest (2 bytes)</th>
<th>key_header</th>
<th>mkd_pk_top</th>
<th>Digest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x55</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80</td>
<td>8 bytes 0x55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 random bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x00 padding as needed</td>
<td></td>
</tr>
<tr>
<td>SHA 256 / SHA 512</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4.1 Recovering the Public Key

The field certification_method_id in the key_header is used to identify which tkck public key (tkck_pk) to use. The public keys are available on the KMA website.

Deciphering (RSA exponentiation) Ciphertext with tkck_pk allows mkd_pk_bottom to be retrieved and concatenated to the end of mkd_pk_top (if present). The public exponent value is given in key_header.

5.4.2 Authenticating the certificate

Recalculate the SHA-256 / SHA-512 over the clear data shown in signing process diagram above and compare it to the hash recovered from the Ciphertext.

5.5 Generation of Ciphertext KTU

Again, the process is very similar to that described in section 4.5, but with the following differences:

- The Area Algorithm ID must be ‘06’ (AES)
- The Area Length must be a multiple of 16 bytes (or be padded to a multiple of 16 bytes)
- The Area Key Data Len must be ‘10’ or ‘20’.
5.6 Generation of the Application Signature
This is described in 3.4.2.

5.7 Creating the Confidential Application Load Unit
The process is identical to that described in 4.7 but uses the new format signature and AES encrypted KTU.

------ End of Document ------