Interoperability Specification for ICCs and Personal Computer Systems

Part 8. Recommendations for ICC Security and Privacy Devices

Apple Computer, Inc.

Axalto

Gemplus SA

Infineon Technologies AG

Ingenico SA

Microsoft Corporation

Philips Semiconductors

Toshiba Corporation

Revision 2.01.01

September 2005

Revision History

Revision	Issue Date	Comments
2.00.01	May 28, 2004	Spec 2.0 Final Draft
2.01.00	June 23, 2005	Final Release
2.01.01	September 29, 2005	Changed Schlumberger to Axalto

Contents

1 Scope

This Part of the *Interoperability Specification for ICCs and Personal Computer Systems* specification defines recommendations for ICCs that support "generic" end-user security and privacy requirements. In this context, generic means support for a broad spectrum of applications and existing open systems standards within the networked PC environment. At the same time, this specification does not preclude implementation of functionality required to meet the special needs of specific industries. Every attempt has been made to maximize vendor flexibility in this regard.

This specification describes requirements for the following areas of functionality:

Identification and Authentication. This specification addresses the need to support high-assurance authentication of the user (cardholder) to external entities. Mechanisms appropriate to authentication over public networks, and working through standard protocols (that is, SSL and TLS) as well as mechanisms that can be easily integrated into existing password-based schemes are addressed. Requirements for the user (or applications running on his behalf) to positively authenticate the ICC and for the ICC to authenticate the user are also considered. Cryptographic algorithms, authentication protocols, and secure storage requirements to implement these features are described.

Information integrity, traceability, and confidentiality. Integrity and traceability are addressed by digital signature generation and verification within the ICC using internally stored key material. By implementing these functions within the ICC, the user gains significant advantages in terms of security and portability over software-only solutions. In addition, key exchange functionality supports the need to securely transmit shared key information between two entities to facilitate use of symmetric key encryption.

While symmetric encryption may be implemented on the ICC, it is not required by this specification. The data bandwidth limitations of current ICC devices are not consistent with the needs of the PC environment for high-speed, network-oriented, data-encryption services.

It should also be noted that this specification assumes that the services and functionality provided by the ICC are intended for use by application programs, running on behalf of the cardholder, on the local PC. In this environment, data to be stored on the ICC is at some time in clear text form within the PC environment. Similarly, data retrieved from the ICC is expected to be used within the PC, and hence must be in clear text form. For this reason, secure messaging, as defined by ISO/IEC 7816-4, between the ICC and the PC will generally not add significant value. The ICC vendor, however, may implement secure messaging to meet the needs of specific industries or other domains.

Secure Storage. General-purpose storage and retrieval mechanisms are specified along with a control policy model that allows flexible access for protecting user data. Use of the ICC storage functionality allows these assets to be both protected and readily portable between PC systems.

The focus within this document is on protection of a user's data. The features and functionality described herein assume that data stored in the ICC belongs to the user and should be accessible to the user. (Though in some cases this access will be

indirect, as with private key material used in digital signatures). This is in contrast to mechanisms defined in several existing industry-specific standards that are oriented toward protecting data on behalf of the application definer, or ICC issuer. Such mechanisms can be implemented in conjunction with this specification without violating its intent.

This document is intended to help establish a common base of functionality that end-user cryptographic devices may support in the future. Many PC-based applications would benefit from an ICC supporting authentication mechanisms, general-purpose digital signature capabilities, and secure portable storage facilities. However, such applications must have a base set of capabilities they can count on being available. It is hoped that multiple ICC vendors will deliver products that implement the functionality defined in this document, making it relatively easy for an end user to acquire such a device, and motivating application developers to build products designed to take advantage of such ICCs.

The remainder of this document describes in detail recommendations for compliant ICCs. This includes requirements for:

- Cryptographic services, including algorithms and key management services.
- Authentication protocols used to support a variety of authentication needs.
- Storage services and access control functionality.
- Implementation information describing required files and data structures as well as commands and associated parameters.

Devices that implement this functionality must incorporate tamper-resistant technology and be implemented in a manner that prevents circumvention of the security mechanisms described herein. No specific requirements are mandated, but ICC vendors should provide information on how they have met this requirement for their products.

2 Cryptographic Services

This section describes the recommended cryptography-related services for compliant ICCs. This is not intended to provide a comprehensive technical description of algorithms and data formats required to implement these services. The reader should refer to the relevant industry standards or vendor documentation for this information.

2.1 Required Algorithms

2.1.1 Hashing

Hash algorithms are used in cryptographic systems for a number of purposes, including creation of message digests for digital signatures, authentication protocols, and integrity checking. Their purpose is to create a relatively short, statistically unique representation of an input data sequence. This must be a one-way transform. That is, given an input value, the resulting hash is easily computed, but given a hash value, it is extremely difficult to construct an input sequence that will produce the desired hash.

Compliant ICCs should implement either the SHA-1 algorithm or the MD5 algorithm. The SHA-1 algorithm produces a 160-bit output, while MD5 produces a 128-bit output. Other algorithms may be added in the future.

2.1.2 Digital Signatures

Compliant ICCs shall support digital signature operations. They shall be able to create a digital signature for arbitrary external data using an internal “private” key value. In addition, they should support the ability to verify an externally generated digital signature, given both the signature object and the associated public key.

Compliant ICCs may support signatures based on either RSA or DSA (part of the DSS). Other algorithm options may be added in the future.

In generating a digital signature, a public-private key pair should be used that is bound to a unique identity. This unique identity could belong to the cardholder, ICC manufacturer, ICC issuer, and so on. In general, keys used for signatures should not be used for other purposes.

2.1.2.1 RSA

For RSA digital signatures, a minimum modulus key length of 512 bits should be used; a length of 1024 bits is recommended. The ICC should store the public and private keys internally and provide a means for external applications to retrieve the public key . The private key shall not be retrievable from the ICC and is used only in internal operations.

To generate a digital signature for a message *M*, take the following steps:

- **Data Encoding** - First, the message *M* is digested with a message-digest algorithm. Compliant ICCs can use either MD5 or SHA-1. This digest is then further processed (encoded or padded), to form a value *D* of the same length as the key modulus.

  At this time, there is no single standard for generating a value *D.* As a result,

ICC vendors should provide a means to import an externally generated value of *D*. This will allow maximum application flexibility. If the ICC supports the ability to generate *D*, based on the input of *M*, it is recommended that the de-facto standard described in PKCS #1 be implemented.

- **Signature Generation** - The signature is generated by applying the signer's RSA private key to the data *D*. This is computed by $S = D^{Kpri} \bmod n$, where *Kpri* is the private key exponent and $n = pq$ is the modulus as described in Section 2.3.

To verify a digital signature signed with a known public key, the developer should take the following steps:

- **Signature Recovery** - The signed data *D* is recovered from the signature *S* with the signer's RSA public key. This is computed by $D = S^{Kpub} \bmod n$, where *Kpub* is the public key exponent and $n = pq$ is the modulus.

- **Verification** - The message digest contained in the recovered data *D* is compared with the message digest of the original message *M*. If the computed digest, and the digest encoded in *D* match, the signature is verified. As above, because the data encoding may be application-specific, or because of ICC technical limitations, it may not be desirable to do this within the ICC.

2.1.2.2 Digital Signature Algorithm (DSA)

DSA is an alternative signature algorithm to RSA, which has been standardized by the U.S. government. Compared to RSA, verification using DSA is computationally more expensive. Also, it can only be used for generating digital signatures.

As with RSA, DSA digital signatures use a public-private key pair associated with a unique entity. Usually this is the cardholder, ICC manufacturer, ICC issuer, and so on. The public key should be retrievable from the ICC by external applications. The private key will never be retrievable from the ICC.

For DSA, the public key consists of the following four values (see Section2.3 on key generation):

p - a 512 to 1024 bit prime

q - a 160 bit prime factor of $(p-1)$

$g = h^{(p-1)/q} \bmod p$, where h is less than $(p-1)$ and $h^{(p-1)/q} \bmod p > 1$

$y = g^x \bmod p$ (a p-bit number)

The private key consists of a single value:

$x < q$, a 160-bit number

To generate a digital signature for a message *M*, the developer should take the following steps:

- **Data Encoding** - First, the message *M* is digested with the SHA-1 message-

digest algorithm and a value *H(M)* derived.

- **Signature Generation** - The signature is computed by:
  - Generating a random number *k* such that $k < q$. k should be generated within the ICC.
  - Computing $r = (g^k \bmod p) \bmod q$ and $s = (k^{-1} (H(M) + xr)) \bmod q$.
  - Exporting the values *r* and *s* from the ICC for transmission to an external system. These two values form the digital signature for the message *M*.

To verify a digital signature, signed with a known DSA public key, the developer should take the following steps:

- **Signature Verification** - The signature (*r* & *s*) is verified by computing:
  - $w = s^{-1} \bmod q$
  - $u_1 = (H(M) + w) \bmod q$
  - $u_2 = (rw) \bmod q$
  - $v = ((g^{u_1} * y^{u_2}) \bmod p) \bmod q$
  - if $v = r$ then the signature is verified.

2.1.3 Key Exchange

It is recommended that ICCs support key-exchange operations. These operations are most commonly used to exchange symmetric session keys between two entities. In the case of RSA-based functionality, key-exchange operations can also play a role in supporting key recovery, though this usage is beyond the scope of this specification.

Compliant ICCs can implement key exchange operations using either the RSA or Diffie-Helman algorithms. Other algorithms may be added in the future. To make use of this capability, the ICC will maintain a public-private key pair, which should be used only for key-exchange functionality.

2.1.3.1 RSA Key Exchange

When using RSA for key exchange, a minimum key length of 512 bits is recommended. The ICC will support retrieval of the public key by external applications. The private key will not be retrievable from the ICC and is used only in internal operations. Key exchange requires that an ICC be able to decrypt data using the internal private key, and encrypt data using an externally supplied public key.

To exchange a key between a sender *S,* and a receiver *R,* perform the following operations:

- *S* generates the key to be exchanged. This may be done on the ICC or externally, depending on where it is to be used. Typically, this is a symmetric

algorithm key *k*, where length(*k)* < *n* bits (the RSA key modulus length). Compliant ICCs need not support RSA encryption of values longer than *n* bits.

- The key, *k*, is padded to form an *n*-bit data value *D*. It is recommended that this padding be done in a manner consistent with PKCS #1.

- The data *D* is encrypted on the ICC using the public key of the receiver and the resulting value, *ED*, exported for transmission to the receiver. This requires computing $ED = D^{Kpub} \bmod n$. Where *Kpub* is the receiver public key exponent and *n* is the modulus. Mechanisms for obtaining this public key are beyond the scope of this specification.

- *R* then decrypts the value *ED* within the ICC using the private key, to recover *D*. This requires computing $D = ED^{Kpri} \bmod n$. Where *Kpri* is the receiver private key exponent. The key value *k* can then be recovered, given the known padding scheme used to create *D*.

2.1.3.2 Diffie-Helman Key Exchange

The Diffie-Helman algorithm provides a means to generate a shared key between two entities, though it uses a different approach than the RSA method described earlier.. Both parties generate the key *k* based on clear text information exchanged between parties. The algorithm consists of the following procedure:

First, the sender *S* and receiver *R* are assumed to have already agreed upon constant values *n* and *g,* which are externally loaded to the ICC. The value *n* is a large prime. (It is preferred that *g* be a generator, but this may be impractical because demonstrating that *g* is a generator effectively requires factoring *n*-1. In practice, it is generally sufficient to select a random *g* such that 1<*g*<*n*-1.) For compatibility with existing ICC hardware, *n* should be between 512 and 1024 bits in length, with 1024 bits being preferred.

The algorithm involves the following steps:

- *S* generates a large random value *x* (512 to 1024 bits in length) where 0 < *x* < *n*-1. The value *x* should be generated within the ICC. *S* then computes $X = g^x \bmod n$ and exports *X* from the ICC for transmission to *R*.

- *R* generates a large random value *y* (512 to 1024 bits) where 0 < *y* < *n*-1. The value *y* should be generated within the ICC. *R* then computes $Y = g^y \bmod n$ and exports *Y* from the ICC for transmission to *S*.

- *S* computes $k = Y^x \bmod n$.

- *R* computes $k = X^y \bmod n$.

The key value *k* is the shared key.

2.1.4 Symmetric Ciphers

ICCs compliant with this specification can implement one or more symmetric ciphers such as RC2, RC4, DES, IDEA, SAFER, and so on. Use of such ciphers within an ICC to encrypt and decrypt externally generated data is, however, of questionable utility

within the PC context due to the limited ICC I/O bandwidth. Hence, this specification makes no recommendations in regards to specific ciphers or associated key lengths.

2.2 Random Number Generation

Compliant ICCs should have an independent source of "random" bits. This random source may be used to generate the following:

- Seed values for unpredictable numbers used in challenge-response authentication protocols
- "Seeds" used in deriving cryptographic keys

This source of random bits should not depend in any way on the state of the card, or on any external elements. Two newly manufactured ICCs should be able to be powered up and immediately produce two entirely distinct and uncorrelated bit sequences. Note that it is not critical that the bit sequence output by the source be unbiased (that is,. 50% zeros and 50% ones), only that there be no correlations in the bit sequences generated by different ICCs.

2.2.1 Increasing Entropy

The random numbers generated within the ICC should have *full entropy.* That is, all possible numbers of any chosen length should be equally likely to be generated. Assuming we have a random bit generator that produces uncorrelated bits with a bias of δ (a bit generator that produces 60% zeros and 40% ones is said to have bias δ = 20% = 0.2), the entropy per bit generated is at least $1-\log_2(1+|\delta|)$. Full entropy indicates that there is one bit of entropy per bit of output.

To increase entropy in the random bit sequence generated within the ICC, you can use a high-quality one-way hash function (SHA-1 is recommended). Use of SHA-1 should not reduce entropy in the input bit sequence, unless the input has more than 160 bits of entropy, in which case the output should have full 160-bit entropy. If the input sequence has full entropy, it may be used directly without application of the hash function.

In use, assuming a k-bit output is desired from the random number generator, and the input is obtained from an uncorrelated bit generator with bias δ, then $k/(1-\log_2(1+|\delta|))$ bits of input to a one-way hash should produce an output with suitable entropy characteristics. If the hash output length is $m < k$ bits, then the hash function should be used $\lceil k/m \rceil$ times, with each application of the hash applied to an uncorrelated input sequence of at least $m/(1-\log_2(1+|\delta|))$ bits, to create a k-bit output.

2.3 Key Generation

While not required, it is desirable that ICCs be able to internally generate cryptographic keys. This is especially important for private key material used in digital signature algorithms because it eliminates a critical point of attack. That is, if the keys are externally generated, they may later be exposed outside the tamper-resistant perimeter of the ICC.

This section provides a brief overview of requirements for generating cryptographic keys for several algorithms likely to be employed on ICCs. The topics describe the nature of the key values and important design considerations. Due to the critical role of key generation in the overall security of the system, any implementation should be carefully

designed and reviewed. One of the most critical elements is the source of random numbers used to seed the key generation process. An ICC that supports internal key generation must incorporate a suitable random number source as described in the preceding section.

2.3.1 Determining Prime Numbers

For a number of cryptographic algorithms, such as RSA and DSA, generation of prime numbers is a key step in the creation of keys. It is important that the prime numbers used in key generation not exhibit a skewed distribution. Note: This could potentially be exploited by an attacker to reduce the work required to recover a key value.

Possible algorithms for generating and detecting prime numbers are described following.

2.3.1.1 Generation

Given a suitable source of random numbers, several methods are available for generating primes. Ideally, the ICC will generate (odd) integers of the appropriate size and test them for primality (see below) until an actual prime is found. In theory, if n-bit primes are sought, about 1 in every $\log_e n$ integers (and hence 2 out of every $\log_e n$ odd integers) tested will be prime.

In practice, prime generation using the preceding method may be too slow. If sufficient memory exists, you can use a sieve. . In sieving, a random (odd) integer is selected as a starting point v and a bit array (A[0 .. m-1]) is initialized to zeros. The interpretation of the array is that A[i] is set to one when the integer v+2i is known to be composite (not prime). It is recommended that the value of m be no larger than the number of bits in v (which is also the number of bits in the prime sought). By computing the value v mod 3, you can quickly eliminate (set to one) all entries in the array that correspond to multiples of 3. The same can be done for multiples of 5, 7, 9, 11, and so on. Note that there is no reason to eliminate multiples of composites such as 9, but it may be easier to sieve out all small odd integers rather than maintain a list of small primes. The appropriate time to conclude the sieving (largest odd integer for which multiples are eliminated) depends on many factors and is best optimized heuristically. However, if v has k bits, then sieving out all odd integers less than k serves as a reasonable starting point. After sieving is complete, surviving integers (those whose bit array entries are still zero) can be tested for primality using the method described in the next section. If none of the surviving entries in the sieve prove to be prime, a new random starting point (v) should be selected and sieving should be repeated.

Note that it is important to generate a new random starting point whenever a new sieve is initialized. Using the last value in an exhausted sieve as the starting point in a new sieve can result in an extremely skewed distribution of primes.

2.3.1.2 Primality Testing

To test an odd integer p for primality (whether it is chosen at random or a surviving element from a sieve), one of several methods can be employed. The simplest method is to employ the Fermat test of computing a^{p-1} mod p. If this value is anything other than 1 (or alternatively, if a^p mod p produces any value other than *a*), then p is certain to *not* be prime and should be eliminated.

In practice, if *a*=2 is used then any large integer *p* that is not eliminated with a single execution of the Fermat test is virtually certain to be prime. However, it is recommended

that a candidate value p pass several Fermat tests (either for randomly chosen values of 'a' or for small fixed prime values such as a=2,3,5,7) before it is accepted as prime.

There are some theoretical objections to the Fermat test, and an alternative known as the Miller-Rabin test is both theoretically sound and slightly faster. The principle reason for *not* using the Miller-Rabin test is that it requires greater code space than Fermat testing. To perform Miller-Rabin testing on an odd candidate prime p, first compute the value q of the largest odd integer factor of p-1. This is accomplished simply by shifting p-1 right until its least significant bit is no longer a zero; p-1 can now be written as $p-1=q2^m$, for some (small) integer m. On each iteration of the Miller-Rabin test, one selects a random value *a* with $0<a<p$ and computes the value $x_0 = a^q \bmod p$. If the value of x_0 is either 1 or p-1 (which is -1 mod p), then this iteration is passed. Otherwise, one begins computing $x_i = x_{i-1}^2 \bmod p$ for $0<i<m$ until an x_i is found whose value is either 1 or p-1. If the first such x_i has value p-1, then the iteration is passed; if the first such x_i has value 1, then the iteration is failed (note that this is different from the case of $x_0 = 1$) and p should be discarded as not prime. If none of the x_i have a value of 1 or p-1, then the iteration is failed and p should be discarded as not prime. A candidate p should pass several iterations of this test with different values of *a* before it is accepted as prime. Note that this is more efficient than Fermat testing because computing all of the x_i including the final value x_{m-1} is computationally equivalent to computing the single value $a^{(p-1)/2} \bmod p$ which is one squaring and modular reduction short of a single Fermat test. Additional savings come from the possibility of early termination when an x_i is found that has value 1 or p-1 and subsequent values x_{i+1}, x_{i+2}, and so on, do not need to be computed.

With both the Fermat test and the Miller-Rabin test, failed candidates are almost always detected in the first iteration, and most of the candidates will be failures. Thus, additional iterations on passing candidates add confidence without substantially affecting the total cost of primality testing. Several other primality testing methods are available, but these generally rely on mathematical techniques that are less likely to be supported by hardware available on the ICC (that is, they require other than modular exponentiation).

2.3.1.3 RSA Keys

You generate RSA keys by selecting two primes of the desired length as described earlier, and then multiplying them to form the public modulus. Note that the product of two k-bit integers may have either 2k or 2k-1 significant bits. If it is important that the modulus have 2k significant bits, ensure this by selecting each of the primes such that *both* of the first *two* bits of each prime are set to one. (For example, to guarantee a full 1024-bit modulus, it is sufficient to generate two primes of 512 bits each in which both have the leading two bits set to one.)

If desired, you can embed a small amount of fixed public data (such as name of owner) within an RSA modulus. To accomplish this, select a region within the upper half (not too close to the midpoint) of the bits of the modulus to hold the fixed data. The first prime is selected randomly as described earlier. An approximate value for the second prime is then computed by dividing the desired modulus (with nonfixed data randomly filled in) by the first prime. Either the random search process or sieving can be used to find a second prime near this computed value, and the actual modulus is then formed (as usual) by forming the product of the two primes.

The public exponent for RSA is generally fixed at an agreed upon value (usually either 3 or 65537). The private exponent is computed using the extended Euclidean algorithm to find an inverse of the public exponent modulo (p-1)(q-1), where the modulus n is the product of the two primes p and q. If "Chinese Remaindering" is used to speed private key operations, two private exponents are computed: the inverse of the public exponent

modulo (p-1) and the inverse of the public exponent modulo (q-1).

2.3.1.4 DSA

DSA keys consist of a public component (comprised of four parameters) and a private component. The public key consists of:

p - a 512 to 1024 bit prime (1024 bits are recommended)

q - a 160 bit prime factor of (p-1)

$g = h^{(p-1)/q} \bmod p$, where h is less than (p-1) and $h^{(p-1)/q} \bmod p > 1$

$y = g^{x} \bmod p$ (a p-bit number)

The private key consists of:

$x < q$, a 160-bit number

2.3.1.5 Symmetric Algorithms

Symmetric algorithms include ciphers such as DES, RC4, IDEA, and so on. Their keys are simply random integers (bit sequences) of the appropriate length.

3 Authentication Services

This section describes algorithms that compliant ICCs should implement. These algorithms are used to support the following features:

- Authentication of the ICC product type (as defined by the vendor)
- User Authentication to the ICC
- Authentication of a PC-based application to the ICC
- Authentication of the user (cardholder) to remote entities, generally some type of server

Consistent with the goals of this specification, only authentication modes that deal specifically with the needs of the cardholder or PC user are defined. To meet application-specific needs of third parties such as the ICC issuer or ICC-supported application provider, you can implement additional algorithms, or variations of those described.

3.1 User Authentication to Remote Entities

In many instances, it is best to employ an ICC when authenticating the user to external systems. Using an ICC, mechanisms may be employed which have significantly better resistance to compromise than methods based on passwords entered by users.

3.1.1 Shared Secret Challenge-Response Protocol

This section describes a simple authentication protocol based on shared secrets. You can employ this protocol in situations where the “server,” which is authenticating the user, knows a secret value (similar to a password) that is also stored securely on the ICC. Note that significant advantages result from using the ICC because it is no longer necessary for a person to remember and enter the secret. Hence, the secret value may be an arbitrary binary sequence that is difficult to guess or attack using common dictionary methods.

The server is assumed to authenticate a client by verifying that the client also possesses their shared secret. To prevent interception of the shared secret, it is not transmitted during the authentication process. To maximize resistance to compromise, one of the two entities should choose this secret and transmit it through a secure, “out-of-band" mechanism at initialization time.

For this protocol, it is assumed that the server and client share two pieces of information. First, they share an 80-bit (10-byte) *name*. This value should be unique for each client-server pair, and for the context in which the authentication is being performed. It is not necessary that this be kept secret. A different client-server pair working in the same application context, as well as the same client-server pair using a different application context, should have a different name. Second, *pass* is a secret shared solely between the client and the server. A length of 160 bits (20 bytes) is recommended for *pass*, but it should always be at least 64 bits. Finally, *hash* is a fixed (public) one-way hash function. For compliant ICCs, this will be implemented as SHA-1 or MD5. A USER AUTH command is used to generate the *resp* data within the ICC and a GET CHALLENGE can be used to generate the *chal* for transmission to a remote system.

Using this information, authentication proceeds as follows:

SERVER	CLIENT
Generate a random 160-bit value *chal* ***Send*** *chal* to client.	
	Compute resp=hash(name,chal,pass). ***Send*** *resp* to server.
Verify: resp=hash(name,chal,pass).	

Comments:

- The *pass* value must initially be securely transmitted between the client and server, and should not be used for any purposes other than authentication with a single server.
- This protocol provides only one-way client authentication (that is, authentication of the client to the server). To achieve one-way server authentication, the roles of the client and server can simply be reversed.
- The above protocol is susceptible to a "man in the middle" attack in which an active adversary can assume control of a channel after authentication has been completed and can then pretend to either party to be the other. This threat can be eliminated by "MACing" each subsequent message. To accomplish this, only hashes of messages (not entire messages) need to be imported into the ICC. The hash of the imported value is then hashed (within the ICC) with *pass* and the resulting hash output is exported as the MAC. To verify the integrity of the MAC'd message, the hash of the received message is again imported into the ICC and the same hash of the import with *pass* is computed and exported. This exported value is then compared with the received MAC to verify message integrity.

3.1.2 Enhanced Shared Secret Challenge-Response Protocol

The protocol described above can be enhanced to support two way authentication and to provide the basis for establishing a "secure channel" between the two parties. In what follows, *pass* and *hash* are presumed to be the same as above. The single *name* constant is replaced by a constant for each side - *sConst1*, *cConst1*. These are distinct 80-bit application-specific constants.

To perform a two-way authentication, the following protocol is used. GET CHALLENGE generates the challenge value sent to the other system. USER AUTH generates the response value send to the other system, and EXTERNAL AUTH is used to input the response for verification.

Two-way authentication proceeds as follows:

SERVER	CLIENT
Generate a random 160-bit value s*cha1.* ***Send*** s*chal1* to client.	Generate a random 160-bit value c*chal.* ***Send*** c*chal1* to server.
Compute: *sresp=hash(cConst1,sChal, cChal,*	Compute: *cresp=hash(sConst1,cChal, sChal,*

SERVER	CLIENT
pass) ***Send*** *sresp* to client.	*pass)* ***Send*** *cresp* to server.
Verify*: cresp=hash(sConst1,cChal, sChal, pass)*	Verify: *sresp=hash(cConst1,sChal, cChal, pass)*

While not required by this specification, this protocol can be further enhanced to generate "session" keys to support secure messaging between the client and server. This is optional, because existing ICCs lack the bandwidth to support general-purpose bulk encryption services for typical PC applications. Also, for industry-specific needs there are other secure message implementations in common use.

A vendor who decides to support this extension may either:

- Generate key and hash values as indicated below and provide a means of exporting them to the PC for use in a software symmetric encryption cipher.
- Generate this key material and use it within an internal symmetric encryption cipher. In the latter case, the vendor must also implement appropriate algorithms to track message numbers.

Implementation of this extension requires additional distinct 80-bit constants (similar to *name*). Two additional constants are required for both the client and server: *sConst2*, *cConst2*, *sConst3*, and *cConst3*. Also, each side must generate two additional secrets, *pass1* and *pass2***.** These are derived from *pass* through simple, agreed-upon transformations, such as bitwise reversal of *pass*.

Extensions for a secure channel

SERVER	CLIENT
Authenticate client as above.	Authenticate Server as above.
...	...
Compute: *sCiph=hash(cConst2,sChal,cChal,pass1)* *sHash=hash(cConst3,sChal,cChal,pass2)* *cCiph=hash(sConst2,cChal,sChal,pass1)* *cHash=hash(sConst3,cChal,sChal,pass2)*	Compute: *cCiph=hash(sConst2,cChal,sChal,pass1)* *cHash=hash(sConst3,cChal,sChal,pass2* *sCiph=hash(cConst2,sChal,cChal,pass1)* *sresp=hash(cConst1,sChal1, cChal1, pass)*

SERVER	CLIENT
Messaging	***Messaging***
Message number 'n' is computed as: sEnc = E(sCiph, m) sMac = hash(sHash, n, m) where E = Encrypt; m = message data Send (n, sEnc, sMac) to client.	Message number 'n' is computed as: cEnc = E(cCiph,m) cMac = hash(cHash,n,m) where E = Encrypt; m = message data Send (n, cEnc, cMac) to server.
...	...
Decrypt & Verify incoming message: m = D(cCiph, cEnc) cMac = hash(cHash, n, m) where D = Decrypt	Decrypt & Verify incoming message: m = D(sCiph, sEnc) sMac = hash(sHash, n, m) where D = Decrypt

Comments:

- For simplicity, use the same key may for both directions of the cipher. In this instance, *sConst2* and *cConst2* can be the same.

3.2 Public Key-Based Protocol

This section describes a public key-based authentication protocol that addresses the needs of client and server authentication over public networks. It specifically addresses the requirements of emerging standards for authentication over the World Wide Web (that is, SSL), as well as broader requirements for transport-layer security .

This authentication shall be done using either RSA or DSA digital signatures. Selection between these two approaches should be based on which algorithm is supported within the ICC, because authentication requires generation of a digital signature.

For the client to authenticate to the server, the client must possess a private signature key and certificate (typically using the X.509 standard), which contains the corresponding public signature key. The private key shall be stored within the ICC and the signature operation performed (using RSA or DSS) as described in Section 2. The certificate may be stored in any convenient manner. While storage within the ICC has some usability advantages, it is not necessary.

The authentication protocol begins with generation of a 160-bit challenge value (*schal)* at the server. The Server sends this to the client PC and retains this value for verification of the response from the client. The *schal* is used as the input message to the digital signature algorithm supported by the ICC and as a digital signature generated as described in Section 2. This requires both the creation of a message digest of the *schal*, and a subsequent encoding step performed for RSA. (Note that additional constants, agreed upon by the client and server, can be added to *schal* as desired without affecting this protocol). The digital signature is the client's response to the server's challenge. The client PC then sends its certificate (retrieving it from the ICC if necessary), and the computed response, to the server.

After the client certificate is received, the Server validates it, making sure a known and trusted Certifying Authority (CA) signed it. When this is done, retrieve the client public

key and use it to verify that the client response is a digital signature of the server challenge created using the client private key.

Client Authentication

SERVER	CLIENT
Generate a random 160-bit value s*cha1.* ***Send*** s*chal* to client.	
	Compute: *M = Hash(schal)* *val = Encode(M) (val=M for DSS)* *resp=Sign(Kpri, val)* ***Send*** r*esp* to server.
Verify*: resp* is a properly signed challenge. The method depends on whether RSA or DSS is used.	

The above mechanism authenticates the client to the server. By reversing the roles of the two parties, it may also be used to authenticate the server to the client. To authenticate the server, the ICC should generate and export a 160-bit random challenge value (retaining a copy internally). The server then creates a response by digitally signing the challenge, using the same methods as described for the Client. The server sends the response and its certificate to the client PC.

After the client PC has the server certificate and the server response, it can proceed to authenticate the server. It is possible to perform this authentication completely in PC software, but it is recommended that the facilities of the ICC be used when available.

First, the client must validate the server certificate by verifying it was signed by a known and trusted CA. If the ICC supports the required algorithms, the ICC may be used to verify this signature. The server public-signature key can be extracted from the certificate. It is assumed in this document that certificate-cracking is not performed in the ICC due to the size and complexity of the ASN.1 encoded data structures specified by X.509. Hence, it is expected that the server public key will be retrieved by PC software and externally loaded onto the ICC. Using this key, the ICC should complete the server authentication by checking the supplied server response to insure that it corresponds to the challenge originated by the ICC.

3.3 Authentication to the ICC

Consistent with the scope of this specification, only two mechanisms for external authentication to the ICC are considered in this section. These are:

- Cardholder verification based on a user-entered password (or PIN)
- Application authentication based on a challenge-response protocol

Specific applications may require other cryptogram-based authentication mechanisms (using symmetric or public key algorithms). Implementation of such mechanisms is compatible with this specification, though no attempt is made to define the range of possible options due to the wide variability in the required security properties.

3.3.1 Cardholder Verification

Compliant ICCs should support cardholder verification (CHV). CHV is performed by matching a CHV "code" that an external application supplies to the ICC by using the VERIFY command. (The user is expected to enter this code at the time CHV is required.) This code is matched against a value stored in a distinguished file. If the values do not match, CHV fails.

CHV codes in the range of 4 to 16 bytes are recommended. In general, these will be ASCII-encoded alphanumeric characters. The ICC, however, should interpret the code as a binary array. The CHV file that stores the CHV code may be written to, with proper authentication, to change the CHV code, but only ICC system software should be able to read it.

Encryption of the CHV code is not specifically defined by this specification, but neither is it prohibited. If encryption of the CHV code is desired, the vendor may implement a proprietary secure messaging variant of the CHV command to support specific applications. A nonsecure messaging variant should still be supported to provide access to controlled services and information controlled by the end user from the typical PC, which is unlikely to have an encrypting CHV code-entry device.

3.3.2 Application Verification

Application verification (APP) allows an application to authenticate to the ICC, not necessarily based on user input data. APP is performed using the shared secret authentication algorithm defined in Section 3.1.1. This mechanism allows an application running on the PC to control access to private data for specific users when stored on the ICC. This type of mechanism maps well to evolving notions of user "private storage" facilities such as those described for the Java or Microsoft "wallet." The mechanism enables somewhat finer-grained control to data than that provided by CHV, because CHV implicitly assumes any application running on behalf of the user has equal access to the ICC resources.

In operation, the application is responsible for generating both a unique *name* and *secret* that will be used to control access to the information it has created. This *secret* may be generated algorithmically, or may involve user input (derived from a user-supplied password, for example). The *name* and *secret* should be stored in a protected (not readable) distinguished file within the ICC. Access to the related DF and EF files may then be controlled based on knowledge of this information. The *secret* should be 160 bits (20 bytes) in length and the *name* 80 bits (10 bytes) in length.

To perform an APP, the application requests a 160-bit random (*chal*) value from the ICC using the GET CHALLENGE command (see Section 0). It then computes the response

resp = hash(*name*, *chal*, *secret*)

where hash is either SHA-1 or MD5, depending on which is implemented by the ICC.

The *resp* is input to the ICC using the EXTERNAL AUTHENTICATE command. The ICC performs the same hash computation and compares its result with that supplied by the application. If they do not match, the authentication fails.

3.3.3 Invalid Attempt Limits

It is strongly recommended that ICCs enforce restrictions on the number of invalid authentication attempts that can be made. A Maximum Invalid Attempt Limit and number of Invalid Attempts Detected should be maintained for each CHV code or APP secret within the ICC. When the Invalid Attempts Detected value reaches the Maximum Invalid Attempt Limit value, the corresponding EF should be "blocked," preventing further authentication from occurring. It is recommended the number of invalid attempts allowed be settable in the range of 1 to 255.

After an EF is marked as blocked, it must be unblocked to enable further access to the associated authentication functionality. The method used for unblocking these files is vendor-defined and must be protected by an independent authentication mechanism.

3.4 ICC Authentication

This function is used to allow an application to authenticate the origin or authenticity of the ICC. It is not mandatory, but is strongly recommended. Note that the authentication is intended to validate the ICC type and manufacturer. This can provide valuable information as to the cryptographic support, tamper-resistance features, and overall security of the product. This is not intended to replace mechanisms defined within specific industry segments for validation of the ICC issuer or specific applications that may be present.

This authentication uses either RSA or DSA digital signatures. Base the selection between these two approaches on which algorithm is supported within the ICC, because authentication requires the generation of a digital signature.

To support ICC authentication, the following elements are securely loaded onto the ICC at time of manufacture:

- ICC Manufacturer Private Key
- Public Key Certificate containing the ICC Manufacturer Public Key

After this information is written to the ICC, it must be protected from alteration. The private key should never be retrievable, while the certificate may be retrieved by an external application.

An external application must retrieve the public key encoded in the certificate to authenticate the ICC. Due to the limited storage resources of existing ICCs, it may be desirable to use a lightweight, signed attribute structure. Alternately, you could use a standard encoding such as that defined in X.509.

ICC Identification requires the following steps. The signature algorithm may be either RSA or DSA.

PC	ICC
Generate an n-bit random *chal.* The value n is based on the algorithm and key size determined from the signed attributes. ***Send*** *chal* to the ICC.	
	Compute resp=Signature(K_{pri}, chal). ***Export*** *resp* to PC.

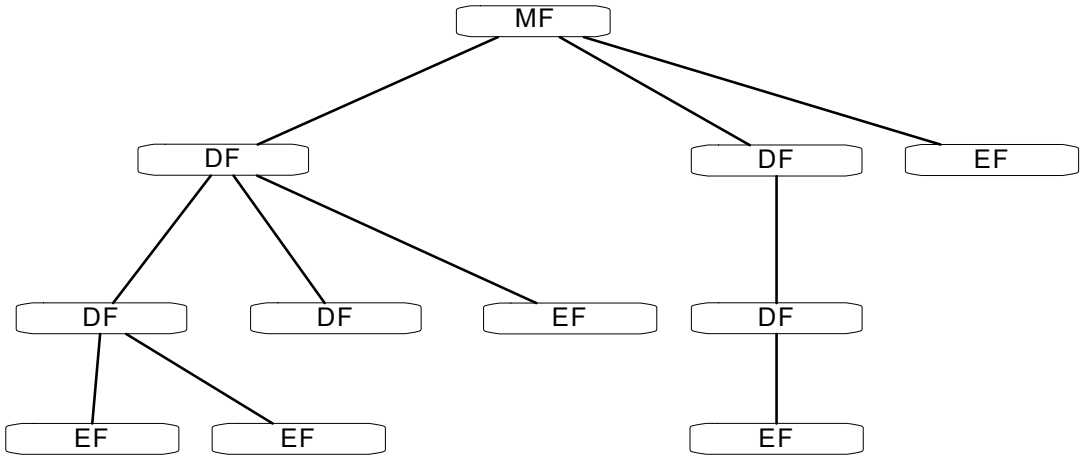
PC	ICC
Verify that the signature is for input data *chal.*	

4 Storage Services

ICCs compliant with this specification shall implement storage functionality compliant with ISO/IEC 7816-4. Required services are described in the following sections. ICC issuers may support additional services as long as the functionality described in this specification is not adversely impacted.

4.1 File Types Supported by the Operating System

The ICC shall support a hierarchical storage system as depicted in the following figure below. The ICC shall not impose restrictions on the breadth or depth of the storage hierarchy, within the limits of available memory, except as specifically noted.



4.1.1 The Master File (MF)

The master file (MF) is a special dedicated file (DF) and defines the root for the storage hierarchy supported by the ICC. The corresponding file ID is 0x3F00. If DF names are supported, the MF name will be set to “Master.File”. The master file will be automatically selected after resetting the card and may be reselected by ID or name, as described in ISO 7816-4.

The MF shall always exist and will generally be created during ICC personalization at the time of manufacture. If the ICC supports re-initialisation of the ICC through an external command, then erasure of any existing storage hierarchy and automatic creation of the MF must be performed by this process. It is not possible to delete the MF.

4.1.2 The Dedicated Files (DF)

A dedicated file (DF) identifies a related group of files consisting of itself and all those files that contain this DF in their parental hierarchy. Each DF has a 2-byte file ID and optional name consisting of 1 to 16 characters. The file ID has the following characteristics:

- It is unique among all siblings having the same DF parent.
- It cannot be any of the following reserved values: 0x3F00, 0x3FFF, or 0xFFFF.

A name, if provided, must be globally unique within the ICC.

The ICC checks for the uniqueness of both the file ID and the name when a DF is created. If the ID and name are not unique, an error will be returned and creation terminated. When a new DF is created, it should be automatically selected as the current DF. The DF may be explicitly selected by its name (when supported), or by using its file ID relative to the parent DF. Selection of file ID 0x3F00 always selects the MF, irrespective of the current DF selection.

4.1.3 The Elementary Files (EF)

The elementary files represent the leaf nodes in the storage hierarchy. EFs may store application or ICC system data. All EFs have a 2- byte file ID that is unique among all siblings having the same DF parent. Certain commands may support a short form of EF addressing in which the EF ID is encoded in 5 bits, with a value between 1 and 30. If short-form addressing is desired, then the vendor specifies how mapping to short IDs is accomplished.

At a minimum, compliant ICCs shall allow selection of an EF based on its file ID relative to its parent DF. This shall include the ability to explicitly select the EF by referencing its file ID

Compliant ICCs should support transparent EFs as defined in ISO/IEC 7816-4. Other EF file types may optionally be supported .

4.1.3.1 Transparent Files

Transparent files consist of a sequence of bytes in a linear array. When addressing transparent files, the fundamental unit of data shall always be defined to be a byte (8 bits). Short-length addressing, as defined for APDUs in ISO/IEC 7816-4 shall be supported.. Hence, the maximum specifiable offset within a transparent EF is 32,767 bytes (15 bits), and a maximum of 256 bytes may be read, or 255 bytes written in a single command. Compliant ICCs shall support a maximum transparent file size of 32,768 bytes.

4.1.4 File References

The ICC should maintain a minimum of two file references at all times. If multiple channels are supported, then the ICC should support two file references per channel. One file reference is the current DF. This reference shall be set to the MF when the ICC is reset. If a valid DF is subsequently selected (using the SELECT command), then the DF reference shall refer to the selected DF. All DF-relative operations are executed in the context of the currently selected DF.

The ICC should maintain a second reference for the currently selected EF. When the ICC is reset, this reference is set to NULL. If a valid EF is subsequently selected, then the EF reference shall refer to the selected EF. Whenever a new DF is selected, then the EF reference shall be reset to NULL. Operations that implicitly operate on the currently selected EF will fail if the EF reference is NULL.

4.1.5 File Control Information

Compliant ICCs will support a file control parameter (FCP) block for each file in the system, as described in ISO/IEC 7816-4. File management data (FMD) and file control information (FCI) blocks may be optionally supported.

4.2 Access Control

Compliant ICCs should implement an access-control mechanism that restricts execution of specific commands for groups of related files based on authentication of the user or calling application. These access conditions apply only to commands generated by application software to the ICC. Since ICC internal system software can bypass the enforcement mechanism, it must be written to prevent alternate command paths that could allow access-control checks to be bypassed.

This section addresses requirements for the two required authentication mechanisms described previously. The ICC vendor may implement other mechanisms. These could include those mechanisms required by specific application environments, such as those defined in EMV or CEN prEN 726.

4.2.1 Access-Control Modes

Compliant ICCs should support the following access control modes:

- ALW (always) - The command can be executed without restriction; in other words, no authentication is required to access this functionality.

- NEV (never) - The command may never be executed by an application; in other words, this command is permanently blocked from executing.

- CHV (cardholder verification) - The command can be executed for the selected DF/EF only if the cardholder (end user) has authenticated himself based on a CHV code associated with the file. CHV authentication is described in Section 4.4.

- APP (application verification) - The command may be executed for the selected DF or EF only if the application has authenticated itself by computing a response based on an ICC-generated random value and a shared secret. APP authentication is described in Section 4.5.

It should be possible to specify one of these conditions for each class of commands such as read, write, update, and delete. Recommended access conditions for certain special files are described in Section 4.4.1. Methods of encoding these access conditions for a given file are at the discretion of the implementor.

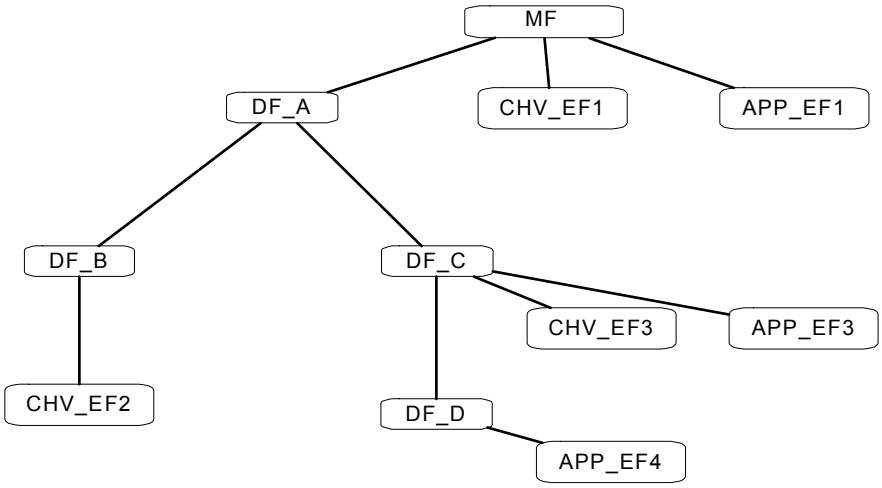
In addition, each EF in the system should maintain a “valid/invalid” flag. After an EF has been marked as invalid, no further operations on that file are allowed except for SELECT, DELETE, and REHABILITATE. A file may be marked as invalid by using the INVALIDATE command to provide administrative control of access to specific files.

4.3 Access Control Enforcement

Access control enforcement decisions are based on the current security state within the

ICC. The security state reflects the currently valid authentication actions (CHV, APP, and so on). The recommended rules for maintaining and updating this security state are described following.

When performing authentication actions, the ICC file system hierarchy determines the CHV code, or APP secret, used. The following figure presents a sample file system hierarchy and depicts the presence of CHV codes and APP secrets associated with specific DFs.



When attempting a CHV authentication (VERIFY command), the CHV code used depends on the currently selected DF. The system first checks for a valid CHV code file for the selected DF. (Note: In making this check, the system ignores CHV code files marked as invalid or disabled. CHV code files marked as blocked are honored, but verification attempts will always fail until the CHV file is unblocked) If the system finds no valid CHV code file, then the DF hierarchy is traversed in reverse order until a valid CHV code file is found, or until the MF is reached. If the system does not find a valid CHV code file in the MF, then CHV authentication fails.

Hence, using the preceding file hierarchy, if DF_D is the selected DF, then CHV_EF3 contains the code value to be checked against the VERIFY command. If DF_B is selected, then CHV_EF2 is used.

For APP authentication, a similar approach is used. In this case, when DF_D is selected, then the APP is validated using the secret stored in APP_EF4. If DF_B is selected, then the secret stored in APP_EF1 is used.

With this approach, it is possible to use a single CHV code and/or APP secret to control access to the ICC files within a given subtree. At the same time, it is possible to restrict functionality to files in a specific DF when necessary.

4.4 CHV Authentication State

The CHV security state determines whether a command with the CHV access mode can be executed for the currently selected file. When an ICC is reset, this security state is set to FALSE for all CHV code files within the system; in other words, all commands requiring CHV are blocked.

In operation, when CHV authentication against a specific CHV code file is successful, then the security state associated with that CHV code file is set to TRUE. From this point on, external applications can execute any of the CHV-controlled commands that rely on that CHV authentication code. This state will be reset to FALSE under any of the following conditions:

- The ICC is reset.
- There is an unsuccessful CHV authentication against the same CHV code file.
- The CHV code file is marked invalid.
- A DF or EF is selected that is not in the subtree controlled by the CHV code file.

This design insures that a cardholder is required to supply a given CHV code only once within the context of a single ICC “session” (in other words, operations against a file in a given subtree). However, the cardholders may be required to perform a subsequent CHV authentication if they select files in a different subtree.

4.5 APP Authentication State

The APP Security State determines whether a command with the APP access mode can be executed for the currently selected file. When an ICC is reset, this security state is set to FALSE for all APP secrets within the system; in other words, all commands requiring APP are blocked.

APP authentication should use quite different usage assumptions than CHV. In particular, it is assumed that APP is used to control access to an application’s private data. In operation, the application will authenticate to the ICC, manipulate its private data in some manner, and then block further access until a subsequent authentication occurs. It is explicitly assumed that other applications running on behalf of the cardholder should not have access to this data.

When APP authentication is successful against a specific APP secret , then the security state associated with that APP secret is set to TRUE. From this point on, the external application can execute any of the APP-controlled commands. This state will be reset to FALSE under any of the following conditions:

- The ICC is reset.
- Another APP authentication is attempted, against the same APP secret file, which is unsuccessful.
- The APP secret file is marked as invalid.
- A DF or EF not in the subtree controlled by the APP secret file is selected.

Typically, an application will gain exclusive access to the ICC (using basic shared access or transaction primitives). It will then select a file containing its private data, authenticate using the APP mechanism, and manipulate this data through a sequence of requests to the ICC. When finished, the application will reset the APP security state to FALSE by selecting the MF and then release to control of the ICC.

5 Special Files

The files described in this section are necessary to support the functionality described previously. The ICC implementor has considerable leeway in how these files are structured, internal data storage conventions, file naming conventions, and so on. However, they should support the basic behavior and access conditions as described in subsequent sections.

5.1 Special Elementary Files

To support both CHV and APP authentication, the ICC must support storage of CHV codes and APP secrets, along with associated usage information such as counters for invalid authentication attempts. Where these files are stored and how they are associated with a specific DF are at the discretion of the implementor.

5.1.1 CHV Code File

The CHV code file should maintain the following information for one or more CHV codes:

- An enabled/disabled flag
- The length of the CHV code (4 to 16 bytes)
- The CHV code
- The invalid attempt limits
- The remaining invalid attempts counter
- Unblocking reference

The following default access control modes should be used for this file to insure protection of the stored codes. This file should not be directly updateable. Rather, a CHANGE CODE function should be used to alter the code or related parameters.

	AC Modes
Create	protected[1]
Read	NEV
Write	NEV
Erase	NEV
Delete	protected
Rehabilitate	protected
Invalidate	protected

Upon creation, the CHV code contents shall be set to NULL and marked as disabled. This indicates an uninitialized state, and the system should ignore this CHV code when making access control decisions.

5.1.2 APP Secret File

The APP secret file should maintain the following information for one or more APP codes:

[1] Implies access is controlled based on CHV, APP or other vendor defined method.

- An enabled/disabled flag
- 10-byte name
- 20-byte secret
- The invalid attempt limits
- The remaining invalid attempts counter
- Unblocking reference

The following default access control modes should be used for this file to insure protection of the stored codes. This file should not be directly updateable. Rather, a CHANGE CODE function should be used to alter the code or related parameters.

	AC Modes
Create	protected
Read	NEV
Write	NEV
Erase	NEV
Delete	protected
Rehabilitate	protected
Invalidate	protected

Upon creation, the APP secret information should be set to the disabled state. This is used to indicate an uninitialized APP secret, and the system will ignore it when making access control decisions.

5.2 Mandatory Files

The following files should exist during normal ICC operation. They will generally be created at the time of manufacture, using a mechanism proprietary to the vendor. If the ICC supports re-initialisation through an external command, then erasure of any existing storage hierarchy and automatic creation of these files must be performed by this process.

5.2.1 Master File

The master file (MF) is the root for the storage hierarchy and must always be present. Consistent with the objectives of this specification in defining an ICC managed by the end-user, the default access control modes for the MF are as follows.

Dedicated File	**ID = 0x3F00; NAME= “Master.File”**	**Default AC Modes**
	Read	ALW
	Update	CHV
	Create File	ALW
	Delete File	CHV
	Rehabilitate	CHV
	Invalidate	CHV

Nothing prevents a card issuer from altering these AC modes before delivering the ICC

to the end user. However, doing this in a manner that precludes all end-user access to the storage facilities of the ICC (for example, restricting Create File based on authentication using an issuer-private key) is not in keeping with the spirit of this specification. An issuer may, of course, create application-specific files not defined in this specification and set the AC modes to any desired value.

5.2.2 MASTER CHV Code

To control access to specific operations beginning with first use of the ICC, associate a master CHV code with the MF. This CHV code should be created when the MF is initialized.

Upon creation, it is recommended that a default CHV code be generated, and that the invalid attempt limit be set to 8. An ICC-specific default code is recommended. However, its selection and method of communicating it to the ICC user are at the discretion of the vendor. It is the responsibility of the end user or intermediate card issuer to change this code, and possibly the invalid attempt limit, on first use of the card. This may be done using the CHANGE CODE command.

5.3 Cryptographic Files

To support the cryptographic functionality described in this document, the ICC must maintain information on available key material. The following sections describe the various types of key material and private data that should be supported, and recommended conventions for referencing that information.

It is strongly recommended that multiple key sets be supported for each algorithm type with key material clearly separated according to intended purpose. To reference multiple key sets, a 1-byte “Key ID” is suggested where the 5 high-order bits indicate the key set, and the 3 low-order bits indicate a version number. This approach allows up to 32 key sets of a particular type to be supported with up to 8 versions of each. The version feature can be used, for example, to keep a historic record of key exchange key sets while periodically updating the active key set.

Tagging the elements that make up the key sets of various types is also recommended. This allows individual components to be referenced for loading and retrieval. Within the context of current ICCs, this represents a good approach to resolving the issues of loading key sets when the size of a key set far exceeds the data field limits of individual T=0 and T=1 command packets. Recommended tag values are provided for each key type, to allow individual components to be transferred using a tag-length-value structure.

5.3.1 Cryptographic DF

The cryptographic functionality described in this specification requires storage of multiple key sets. It is recommended that files storing this information be collected in a single subtree of the storage hierarchy under a cryptographic DF This simplifies establishing a well-defined security perimeter and consistent protection of this information for unauthorized modification.

It is recommended that access modes on this DF be set as follows:

	AC Mode

Read	ALW
Update	Protected
Create File	Protected
Delete File	Protected
Rehabilitate	Protected
Invalidate	Protected

5.3.2 Cryptographic Algorithm EF

It is recommended that the ICC vendor provide an EF that contains information on the algorithms and key sizes that the ICC supports. . This can assist cryptographic service provider developers in providing support for a family of ICC products that may vary in their cryptographic support. The information is encoded at the discretion of the ICC vendor. If provided, this information should be created at the time of manufacture, and further modification should be blocked.

	AC Mode
Read	ALW
Write	CHV
Erase	CHV
RFU	NEV
Rehabilitate	CHV
Invalidate	CHV

5.3.3 RSA Private Signing Keys

If RSA digital signatures operations are supported, then the ICC will provide a protected means of storing related private key information. Vendors can define their own file format to optimize storage or processing efficiency. This key material should never be exported out of the ICC.
It is expected that applications will interact with this key storage only indirectly to load key material.
Since an RSA-private key consists of a number of potentially large components, it is recommended that each component be independently loadable. The components should be identified by a key ID and the following component tags.

Tag	**Description**
0x02	Prime #1 (p)
0x04	Prime #2 (q)
0x06	Coefficient ($q^{-1} \bmod p$).
0x08	Exponent #1 (Kpri mod (p-1)).
0x0A	Exponent #2 (Kpri mod (q-1)).
0x0C	Private key exponent (Kpri)

5.3.4 RSA Public Signing Keys

The ICC should store an RSA public signing key corresponding to each private signing key used within the ICC. Vendors can define their own file format to optimize storage or processing efficiency. Public keys should be retrievable by external applications using the GET PUBLIC KEY command.

Because an RSA public key consists of a number of potentially large components, it is recommended that each component be independently loadable. The components should be identified by a key ID and the following component tags.

Tag	Description
0x12	public key exponent (Kpub)
0x14	public modulus (n)
0x1A	H
0x1C	J0

J0 and H are optional Montgomery constants often used to increase performance..

5.3.5 DSA Private Signing Keys

If DSA digital signature operations are supported, then the ICC will provide a protected means of storing related private key information. Vendors can define their own file format to optimize storage or processing efficiency. This key material should never be exported out of the ICC.
It is expected that applications will interact with this key storage only indirectly to load key material.

Because a DSA private key consists of a number of potentially large components, it is recommended that each component be independently loadable. The components should be identified by a key ID and the following component tags.

Tag	Description
0x42	Public prime (p)
0x44	Private value (x)
0x46	public prime factor (q)
0x48	g

5.3.6 DSA Public Signing Keys

The ICC should store a DSA public signing key corresponding to each DSA private signing key used within the ICC. Vendors can define their own file format in order to optimize storage or processing efficiency. Public keys should be retrievable by external applications using the GET PUBLIC KEY command.

Because a DSA public key consists of a number of potentially large components, it is recommended that each component be independently loadable. The components should be identified by a key ID and the following component tags.

Tag	Description
0x46	public prime factor (q)
0x42	public prime (p)
0x48	g
0x52	y

5.3.7 RSA Private Key Exchange Keys

If the ICC supports RSA key exchange operations, it will need to store RSA private key exchange keys. Vendors can define their own file format to optimize storage or processing efficiency. This key material should never be exported out of the ICC. It is expected that applications will interact with this key storage only indirectly to load key material.

When loading key material, the tag values identified in Section7.2.3.7 should be used.

5.3.8 RSA Public Key Exchange Keys

The ICC should store an RSA public key exchange corresponding to each RSA private key exchange key used within the ICC. Vendors may define their own file format to optimize storage or processing efficiency. Public keys should be retrievable by external applications using the GET PUBLIC KEY command.

When loading key material, the tag values identified in Section 0 should be used.

5.3.9 Diffie-Helman Constants

If the ICC supports Diffie-Helman key exchange, then it will need to store the required key material. A separate set of keys (or pre-arranged constants) is required for each party with whom the user wants to exchange keys. Vendors can define their own file format to optimize storage or processing efficiency. This key material should never be exported out of the ICC. It is expected that applications will interact with this key storage only indirectly to load key material.
Because a Diffie-Helman key exchange requires storing two large constants, it is recommended that each constant be independently loadable. The components should be identified by a key ID and the following component tags.

Tag	Description
0x72	Constant prime factor (n)
0x74	Constant (g)

5.3.10 User Secrets

If the ICC supports one or more of the user authentication protocols defined in Section 7.2.2.7, it will need to store externally supplied secrets (similar to passwords). For each secret, the ICC should store the following information:

- Resource name: An application-defined or user-defined name associated

with this secret
- *name* – A 10-byte unique name associated with a specific service.
- *pass* – A 20-byte shared secret.

These secrets are accessed by using the USER AUTH command and should be referenceable by a 1-byte ID value or the resource name. Storage for up to 256 secrets, to support authentication to multiple servers, is recommended. Vendors can define their own file format to optimize storage or processing efficiency. External applications interact with this store only indirectly to load secret material or perform authentication using the USER AUTH command .

For consistency and future flexibility, it is recommended that this information be loadable as a set of independent components. Each component would be identified by a key ID and the following component tags.

Tag	Description
0xA2	resource name
0xA4	*name*
0xA6	*pass*

5.4 ICC Identification Files

The following files are optional. If the ICC vendor implements ICC authentication as described in Section 3.43.4, the relevant information should be stored in the manner described following.

5.4.1 ICC Identification DF

This DF would be the parent for the EFs holding data required to support the ICC Authentication function. Use of an ID=0x3F15 and Name=”ICC.ID” is suggested to aid in locating this information. Note that if supported, this DF and associated EFs must be created at the time of manufacture and must not be alterable.

The DF should have the following access control conditions.

ID = 3F15; Name=”ICC.ID”	Modes
Read	ALW
Update	NEV
Create File	NEV
Delete File	NEV
Rehabilitate	NEV
Invalidate	NEV

5.4.2 ICC Identification EF

The ICC Identification EF is a Transparent EF that contains the ICC manufacturer’s public key certificate. This EF must be created at the time of manufacture and must not be alterable. Recommended access modes are listed in the following table.

	Relevant
Read	ALW
Write	NEV
Erase	NEV
RFU	NEV
Rehabilitate	NEV
Invalidate	NEV

Note that this file would contain the certificate that includes the digital signature block used to authenticate the information. For efficiency, it may be desirable to store the public key material in a separate location. If this is done, it is critical to precisely state the method of including the public key with the remaining certificate information to create the digital signature.

5.4.3 ICC Identification Private Key

The ICC manufacturer private key should be stored and protected in a manner consistent with storage of RSA or DSA private signing key material. It must be installed at the time of manufacture and be neither retrievable nor modifiable.

5.4.4 ICC Identification Public Key

The ICC manufacturer's public key should be stored and protected in a manner consistent with storage of RSA or DSA public signing key material. It must be installed at the time of manufacture and be retrievable by external applications. It must not be modifiable.

6 ATR Requirements

Compliant ICCs support asynchronous response to a reset as described in Part 2 of this specification. They shall supply a valid ATR sequence following a reset operation that complies with the format indicated below. Standardization of the vendor specific information in the ATR historical byte field, consistent with methods described in ISO/IEC 7816-4, will allow efficient mapping of ICCs to supporting service providers within the PC environment.

ATR Byte	Value (in HEX)
TS	Direct or inverse convention flag
T0	xF – where 'x' is a value indicating the valid interface characters (which follow), and F indicates that 15 historical bytes are present.
TA_i - TD_i	Up to 14 interface characters describing supported protocols and parameters.
T1	80 – indicates that compact TLV data objects follow.
T2	25 – tag indicating that a 5-byte issuer ID number follows. This identifier shall be a registered application identifier (RID) for the ICC issuer as defined in ISO/IEC 7816-5.
T3	RID_1
T4	RID_2
T5	RID_3
T6	RID_4
T7	RID_5
T8	56 – tag indicating that 6 bytes of ICC issuer data follow. Compliant ICCs include a vendor model/revision identifier. A 5-byte model identifier followed by a 1-byte major revision and 1-byte minor revision number is recommended.
T9	$Model_1$
T10	$Model_2$
T11	$Model_3$
T12	$Model_4$
T13	$Model_5$
T14	Major
T15	Minor
TCK	Checksum byte

7 Recommended Commands and Functional Behavior

This section describes commands required to implement functionality described in this document. All commands described herein shall be interpreted in a manner consistent with the short addressing APDUs as defined in ISO/IEC 7816-4, and shall be mapped onto the T=0 or T=1 protocol as described in Annexes A and B of that specification.

ICC vendors can support additional commands, providing they do not adversely impact the functionality described in this specification.

7.1 Command Summary

The following tables summarize the recommended command set. Class and instruction bytes for each command should be selected for compatibility with ISO/IEC 7816-4. If the ICC vendor chooses to implement channels or secure messaging, use of these features will be indicated by the value encoded in this low-order nibble in accordance with ISO/IEC 7816-4 Table 9. A description of the recommended functional behavior for each command, along with a description of suggested input parameters and error codes, is provided in Section 7.2.1.

Table 7-1. Security Relevant Instructions

Command
VERIFY
CHANGE CODE
UNBLOCK
GET CHALLENGE
INTERNAL AUTH
EXTERNAL AUTH
USER AUTH
INVALIDATE
REHABILITATE

Table 7-2 Cryptographic Instructions

Command
LOAD PUB KEY
LOAD PRIV KEY
GENERATE KEY
GET PUBLIC KEY
DELETE KEY
LOAD DATA
SIGN DATA
LOAD VERIFY KEY
VERIFY SIGNATURE
LOAD EXPORT_KEY
EXPORT KEY

Command
IMPORT KEY
HASH DATA

Table 7-3. Operational Command Summary

Command
CREATE FILE
DELETE FILE
SELECT
READ BINARY
GET RESPONSE
GET DATA
WRITE BINARY
UPDATE BINARY
PUT DATA

Note: Only the subset of the ISO/IEC 7816-4 defined command set required to implement functionality described in this document is included herein. Vendors who implement additional 7816-4 functions are fully compliant with the intent of this specification.

7.2 Functional Description

7.2.1 Common Status Codes

The following general status codes are recommended to indicate errors in executing the commands defined in this document. Additional Status codes specific to a command are noted with each command.

Value (in hex)	Meaning
9000	Success
61xx	xx indicates the number of response bytes still available.
6281	Part of returned data may be corrupted.
6282	End of file reached on read operation.
6283	Selected file marked invalid.
6381	File space exhausted.
6581	Memory error.
6700	Error in data length (P3 invalid).
6982	Access conditions not satisfied.
6986	Command is not allowed.
6A81	Function is not supported.

Value (in hex)	Meaning
6A82	File not found.
6A84	Insufficient space in the file.
6985	Instruction sequence not performed as required.
6A86	Parameter errors in P1-P2.
6A88	Referenced data not found.

7.2.2 Security Relevant Instructions

7.2.2.1 VERIFY

The VERIFY function allows the cardholder to supply a password for verification against CHV code values associated with a DF. A failed authentication attempt will cause the associated invalid attempt counter to be incremented.

INPUT DATA

Name	Length	Description
token ID	1 byte	Optional; used to distinguish between multiple CHV tokens associated with a single DF.
password	4-16 bytes	Byte array containing the value.

ERROR CODES

Value (in hex)	Meaning
6300	Authentication failed

7.2.2.2 CHANGE CODE

The CHANGE CODE function allows the cardholder to change the values associated with either a CHV code or APP secret . This function attempts to operate on CHV or APP value(s) associated with the currently selected DF. If no such value exists, the command fails.

This command should accept a valid current authentication token (code or secret) followed by the new token to be set. If the current token fails the validation attempt, the command fails and the invalid attempt counter is incremented. If the CHV or APP is disabled, then a zero length authentication token, followed by a new token, is used to enable the mechanism. The following table shows the input data for CHV.

Name	Length	Description
CHV ID	1 byte	CHV algorithm identifier (vendor defined).
token ID	1 byte	Optional; used to distinguish between multiple CHV tokens associated with a single DF.
token length	1 byte	Length of current password.
current password	4-16 bytes	Byte array containing current password.

Name	Length	Description
new length	1 byte	Length of a new password; specifying a length of 0x00 will disable the CHV.
new password	4-16 bytes	Byte array containing a new password.

The following table shows input data for APP.

Name	Length	Description
APP ID	1 byte	APP algorithm identifier (vendor defined)
token ID	1 byte	Optional, used to distinguish between multiple APP tokens associated with a single DF
token length	1 bytes	Length of current password
token	16 or 20 bytes	Hash(name, chal, pass). The chal value must have been obtained using GET CHALLENGE immediate preceding this command
new name	10 bytes	New value for name parameter
new pass length	1 bytes	Length of new password
new password	8 to 20 bytes	New APP password value

The following table shows error codes.

Value (in hex)	Meaning
6300	Authentication failed.
6581	Update is not allowed.

7.2.2.3 UNBLOCK

The UNBLOCK function allows a blocked CHV code or APP secret to be unblocked. This function attempts to operate on the appropriate authentication mechanism associated with the currently selected DF. If no CHV or APP is defined for the DF , then the command fails. The following table shows input data.

Name	Length	Description
token length	varies	Vendor defined unblock token
token ID	1 byte	Optional, used to distinguish between multiple tokens associated with a single DF

The following table shows error codes.

Value (in hex)	Meaning
6300	Authentication failed.
6581	Update is not allowed.

7.2.2.4 GET CHALLENGE

This command provides one-time challenge data that can be used in a subsequent EXTERNAL AUTH command to the ICC. Note that the data supplied is guaranteed to be valid only in the context of the next command to the ICC, and hence should be immediately followed by the EXTERNAL AUTH command. The following table shows input data.

Name	Length	Description
challenge size	varies	Typically an explicit size or algorithm identifier

7.2.2.5 INTERNAL AUTH

The INTERNAL AUTH function supports authentication of the ICC to an external application, as described in Section 3.3.2 of this specification.

INPUT DATA

Name	Length	Description
challenge	varies	If RSA, a value of size RSA modulus length. If DSA, a 160-bit value.

7.2.2.6 EXTERNAL AUTH

The EXTERNAL AUTH function supports the ability of an external application to authenticate to the ICC based on a shared secret. This is used in situations involving a challenge-response authentication protocol.

Can be used to support one-way or two-way client/server authentication, as defined in Section 3.1. When using two-way authentication, the USER AUTH function must be executed before EXTERNAL AUTH. This insures that the external challenge data required for authenticating the external system is available. In operation, the following sequence should be executed in succession to avoid possible destruction of intermediate results:

1. GET CHALLENGE
2. USER AUTH
3. EXTERNAL AUTH

For the digital signature-based algorithms, it is assumed that the signed data represents a hash of the last challenge generated using the GET CHALLENGE command. In the case of RSA, the resulting message digest should be encoded as described in PKCS #1.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Algorithm identifier
response	varies	Depends on algorithm type

ERROR CODES

Value (in hex)	Meaning
6300	Authentication failed

7.2.2.7 USER AUTH

The USER AUTH function supports generation of user (cardholder) authentication data for use by an external server using a shared secret algorithm. It requires input of a reference to a shared secret..

Note that the corresponding authentication of the server to the PC, based on a shared secret, may be verified using the EXTERNAL AUTH command.

INPUT DATA

Name	Length	Description
secret ID	1 byte	Reference to shared secret data
challenge	20 bytes	Random challenge data from an external system
resource name	varies	Optional, a string constant between 1 and 32 bytes in length

ERROR CODES

Value (in hex)	Meaning
6300	Authentication failed

7.2.2.8 INVALIDATE

This function will mark the currently selected EF as invalid. If the current EF selection is NULL, this function returns an error.

7.2.2.9 REHABILITATE

The REHABILITATE function allows an invalid EF to be reset to the valid state. If no EF is currently selected, or if the selected EF is valid, this function returns an error.

7.2.3 Cryptographic Instructions

7.2.3.1 LOAD PUB KEY, LOAD PRI KEY

The LOAD PUB KEY and LOAD PRI KEY functions support loading key material into key storage. These functions are considered privileged operations and should be restricted using APP or another appropriate authentication mechanism.

For RSA and DSA operations, these functions load private key material used in ICC internal computations, along with associated public key material. Care must be taken to insure that public and private keys, within a complimentary key pair, are loaded with the identical key ID values.

For Diffie-Helman, key exchange only the LOAD PUB KEY function is used. This is used to load the required public constants.

For user-secret information used in authenticating the user to external entities, only the

LOAD PRI KEY function is used to load the resource name, and the *name* and *pass* parameters.

When loading a key value, care must be taken to insure the use of a key length compatible with the ICCs internal operation. Because of the potential size of key material to be loaded for algorithms specified in this specification, loading a single key will generally require multiple independent invocations of this command to load all associated components. It is the responsibility of the calling external application to know which key components are required by a given ICC implementation and to load all required values. Failure to do so will result in an unusable key. All key components should be loaded using a Tag-Length-Value (TLV) representation. The Tag values are provided in Section 7.2.3.7 and the length specifies the length of the key component. ICC implementations of this command must be able to determine keys for which not all components are loaded, and return an error if an application attempts to use them. Also, the invocation that attempts to load the first key component initializes a new Key ID value for the specified Alg ID in the appropriate key storage. Finally, attempts to overwrite an already existing key component using this command should return an error. The following table shows input data.

Name	Length	Description
Alg ID	1 byte	Algorithm ID
Key ID	1 byte	Key identifier
tag	1 byte	Key component Tag
length	1 byte	Length of the following component value
value	varies	Key component value

7.2.3.2 GENERATE KEY

Supports internal generation of keys into key storage associated with the algorithm identified by the Alg ID. This function can be used to generate either RSA or DSA keys, as described in this specification. This command is considered privileged and will succeed only if required authentication has been successful.

For RSA and DSA, both private and public keys are generated and loaded with the same key ID.

The vendor can also define extended functionality for this command to include generation of symmetric key values.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Algorithm ID.
Key ID	1 byte	Key identifier.
seed value	variable	An external seed value to the ICC key generation algorithm, if required. If the ICC generates internal key seeds, this field is ignored.

ERROR CODES

Value (in hex)	Meaning
698E	Key generation did not find an appropriate prime within designated limits; re-initiate to try again.

7.2.3.3 GET PUBLIC KEY

Returns the requested public key component for a specific Alg ID and Key ID. Since a complete public key may contain multiple components, multiple invocations of this command may be required to retrieve a complete public key value.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Algorithm ID
Key ID	1 byte	Key identifier
tag	1 byte	Key component tag

7.2.3.4 DELETE KEY

The DELETE KEY command supports deletion of keys previously created using the LOAD PUB KEY, LOAD PRI KEY, or GENERATE KEY commands. The command will fail unless required authentication has been successful. When a key is deleted, both the public and private portions of the key with the specified ID for the designated algorithm are deleted. If no key with the specified ID exists, an error is returned.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Algorithm ID
Key ID	1 byte	Key identifier

7.2.3.5 LOAD DATA

The LOAD DATA command loads data to be signed using the SIGN DATA or VERIFY SIGNATURE command. The length of the supplied data must match that required for RSA or DSA, as implemented within the ICC. No transformation or encoding of this data is made before signing. Note that this data is guaranteed to be available only for execution of the next command; hence, a LOAD DATA command should be immediately followed by SIGN DATA or VERIFY SIGNATURE.

The HASH DATA command described below may provide an alternate means of preparing data to be signed.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Algorithm identifier
data length	1 byte	Length of valid data.

Name	Length	Description
data	variable	For RSA, the length of the data should equal the RSA modulus length. For DSA, it should be 160 bits.

7.2.3.6 SIGN DATA

The SIGN DATA command creates a digital signature using the specified algorithm and key. For algorithms defined in this specification, the key is always located in the private signing key storage. The data to be signed should be loaded using the LOAD DATA or HASH DATA commands. Use of the latter functionality is optional. If implemented, the ICC system software must perform any data encoding or transformation necessary to prepare the message digest for signing.

It is recommended that this be treated as a privileged operation and CHV authentication be required before completing a signature operation.

The SIGN DATA command returns an error if no data to be signed is found.

DATA

Name	Length	Description
Alg ID	1 byte	Signature algorithm to be used
Key ID	1 byte	Key identifier

7.2.3.7 LOAD VERIFY KEY

The LOAD VERIFY KEY loads an external public key to be used in a subsequent signature verification. The key length must be compatible with that supported by the ICC. Because of the potential size of key material to be loaded for algorithms specified herein, this command must be independently invoked multiple times to load a complete key. It is the responsibility of the calling external application to know which key components are required by a given ICC implementation and to load all required values. Failure to do so will result in an unusable key.

Note that these keys are only guaranteed to be available for execution of the next command; hence, a VERIFY SIGNATURE command should immediately follow the LOAD VERIFY KEY command.

DATA

Name	Length	Description
Alg ID	1 byte	Algorithm identifier for this key
Tag	1 byte	Key component tag
Length	1 byte	Length of the following component value
Value	varies	Key component value

7.2.3.8 VERIFY SIGNATURE

The VERIFY SIGNATURE command verifies an externally generated digital signature. The key used in the verification may be an external public key loaded using the LOAD VERIFY KEY function, or optionally retrieved from an internal file. If the prior command

was LOAD VERIFY KEY, then that key should be used.

The value to be compared against the input signature (encoded message digest) should be input using the LOAD DATA command before executing this command.

INPUT DATA

Name	Length	Description
AlgID	1 byte	Signature algorithm to be used
key ref	variable	A key reference unless implicit through use of LOAD VERIFY KEY command

ERROR CODES

Value (in hex)	Meaning
6300	Verification failed

7.2.3.9 LOAD EXPORT KEY

The LOAD EXPORT KEY loads an external RSA public key to be used in a subsequent key export operation. This command is not required for Diffie-Helman key exchange. The RSA key length must be compatible with that supported by the ICC. Because of the potential size of key material to be loaded for algorithms specified herein, this command must be independently invoked multiple times to load a complete key. It is the responsibility of the calling external application to know which key components are required by a given ICC implementation and to load all required values. Failure to do so will result in an unusable key.

Note that this key is guaranteed to be available only for the execution of the next command; hence, an EXPORT KEY command should immediately follow the LOAD EXPORT KEY command(s).

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Key exchange algorithm
tag	1 byte	Key component tag
length	1 byte	Length of the following component value
value	varies	Key component value

7.2.3.10 EXPORT KEY

The EXPORT KEY command creates data necessary to exchange a key using the ICC key exchange algorithm. For RSA, the key exchange key is assumed to be the public key of the other party. This key may be an external public key loaded using the LOAD EXPORT KEY function, or may be retrieved from an internal file. If the prior command was a LOAD EXPORT KEY, then that key is used and the Key ID field is ignored.

For Diffie-Helman key exchange, the Diffie-Helman key data is used and only a key reference is required.

The output of this function is an encrypted key blob for RSA (PKCS #1 format), or X or Y

values for Diffie-Helman key exchange.

DATA

Name	Length	Description
Alg ID	1 byte	Key exchange algorithm.
key ref	variable	For Diffie-Helman key exchange, a key identifier is input. For RSA, it is a key length followed by the key value.

7.2.3.11 IMPORT KEY

The IMPORT KEY command decodes a key blob using the ICC key exchange algorithm and an internal private key.

The treatment of the unwrapped key depends on the ICC implementation. For ICCs that do not support internal symmetric cryptographic algorithms, this function should return the key value as a 1-byte length followed by the key data. For ICCs that support symmetric ciphers, the key should be placed in a key file, and only a Key ID value returned.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Key exchange algorithm.
Key ID	1 byte	Identifier for the key exchange key to be used.
key blob length	1 byte	
key blob	variable	For RSA, this is an encrypted key blob. For Diffie-Helman key exchange, this data is input from the other party (X or Y).

7.2.3.12 HASH DATA

The HASH DATA command allows a hash, or message digest, to be computed for arbitrary input data. The initial call to HASH DATA should re-initialize internal registers and begin a new hash. The data supplied in this command and in and subsequent HASH DATA commands are then interpreted as the message data to be hashed.

After all desired data is hashed, a SIGN DATA command can be issued to digitally sign the hash. A GET RESPONSE command can be issued to retrieve the hash value. . After a SIGN DATA or GET RESPONSE command is executed, HASH DATA must be re-initialized.

INPUT DATA

Name	Length	Description
Alg ID	1 byte	Hash algorithm
data length	1 byte	Length of valid data to add to hash
data	variable	An arbitrary byte sequence representing data to be hashed

7.2.4 Operational Instructions

7.2.4.1 CREATE FILE

The CREATE FILE operation creates a DF or an EF. The new file is created as a child of the currently selected DF. The newly created file is then automatically selected.
The following is typical of data that must be supplied. Specific implementations may require additional data.
For a DF:

Name	Length	Description
File ID	2 bytes	The file identifier must be unique within the ICC storage system, or creation fails.
name length	1 byte	Length of file name.
name	varies	A 1- to 16-byte unique DF name; if a duplicate exists, creation fails.
access control	varies	Initial access control settings.

For an EF:

Name	Length	Description
File ID	2 bytes	The file identifier must be unique within the ICC storage system, or creation fails.
access control	varies	Initial access control settings.

7.2.4.2 DELETE FILE

The DELETE FILE command deletes a DF or EF that is an immediate child of the currently selected DF. The space allocated by a deleted file is fully reclaimed by the system and should always be re-initialized to insure that any residue from the file is not accessible based on a subsequent file allocation.

The application must have permission to delete the selected file, as defined by the current security state. If an attempt is made to delete a DF that has children, this function will return an error.

DATA

Name	Length	Description
File ID	2 bytes	Identifier for an EF or DF that is an immediate child of the selected DF

7.2.4.3 SELECT

The SELECT command allows an EF or DF to be selected, and the implementation will comply with ISO/IEC 7816-4.

DATA

Name	Length	Description
selection control	2 bytes	Per 7816-4
file information	variable	Either a 2-byte file identifier or a DF name

7.2.4.4 READ BINARY

The READ BINARY command reads the specified data from a transparent EF. If the selected EF is NULL, or if the EF is not transparent, an error is returned. Attempting to read past the end of a file returns an error.

INPUT DATA

Name	Length	Description
offset	1 byte	Offset into file of first byte to read
length	1 byte	Length of data to read

7.2.4.5 GET RESPONSE

The GET RESPONSE command returns the result of the last command, as described in ISO/IEC 7816-4.

7.2.4.6 GET DATA

The GET DATA command supports reading of primitive data objects (TLV encoded) within the currently selected DF.

INPUT DATA

Name	Length	Description
tag	1 byte	Tag of data element
length	1 byte	Length of data to retrieve

7.2.4.7 WRITE BINARY

the WRITE BINARY command allows data to be written to a transparent EF. The WRITE command should be used to append new information to a file. To alter existing data, use the UPDATE command. It is recommended that the default behavior of this command have a one-time write of the supplied data bytes..

INPUT DATA

Name	Length	Description
data	variable	Data to write

7.2.4.8 UPDATE BINARY

The UPDATE BINARY command allows existing data within a transparent EF to be altered. If the command references beyond the existing end of file, an error occurs and no data is updated.

INPUT DATA

Name	Length	Description
offset	1 byte	Offset into file to begin update
data	variable	Data to update

7.2.4.9 PUT DATA

The PUT command supports writing a primitive data object (TLV encoded) within the currently selected DF.

INPUT DATA

Name	Length	Description
tag	1 byte	Tag of data element
length	1 byte	Length of data
value	variable	Data to be written