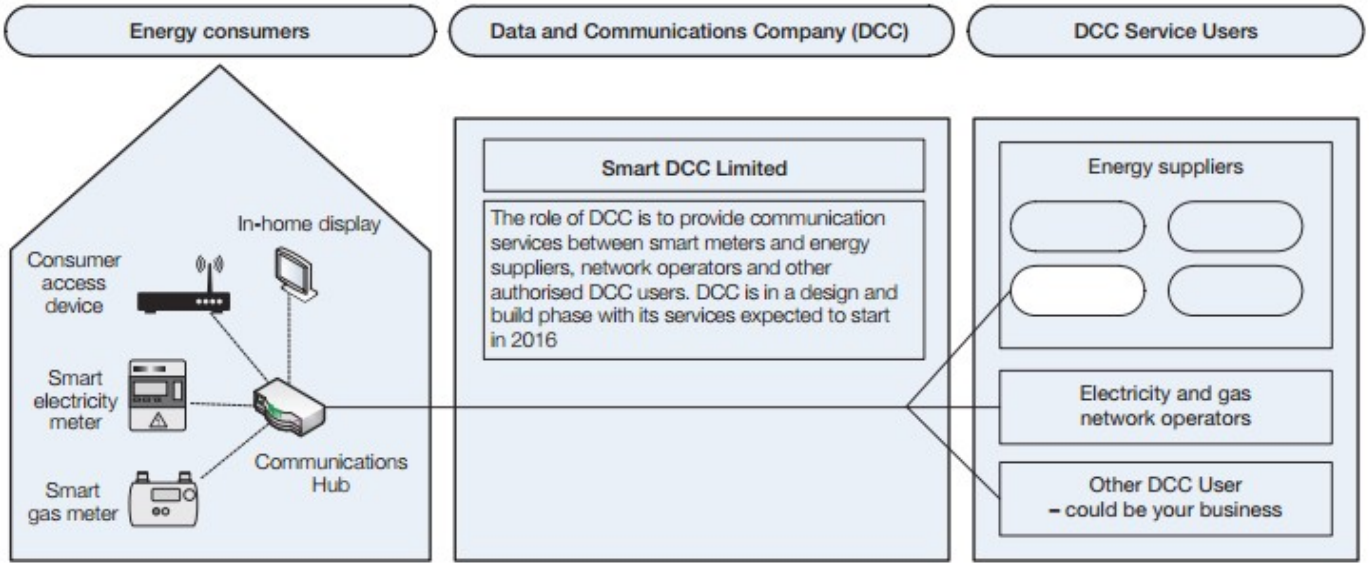
Smart Energy Metering Concept:



Data Communication Company(DCC):
https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/397291/2903086_DECC_cad_leaflet.pdf

The DCC will put in place communications across Great Britain to send and receive information from smart meters to energy suppliers, energy network operators and energy service companies.The DCC will be operated by Capita PLC under a licence regulated by Ofgem. The DCC will manage three main subcontractors. CGI IT UK Limited is the Data Services Provider, which controls the movement of messages to and from smart meters. Arqiva Limited and Telefónica UK Limited are the Communications Service Providers who will put in place the Wide Area Network.

Arqiva will provide the network for Scotland and the north of England using long-range radio communications. Such infrastructure and technology is already used for other important national communications networks, such as those for digital television and emergency services.

Telefónica's network will cover the rest of England and Wales using cellular radio communications (technology typically used in mobile phone systems) plus "mesh" radio technology to supplement connectivity in a small number of hard to reach locations (such mesh systems have been used in smart meter installations in Sweden, Norway and Finland).

Protection of Remote Party Messages achieved as follows:

https://www.smartenergycodecompany.co.uk/docs/default-source/secdocuments/guidance/sec-guidance---smart-cos-process-diagram.pdf?sfvrsn=10

A Command that is sent from a Remote Party to a Device is constructed by the Remote Party and sent to the ACB.
The ACB adds integrity and authenticity protection to the message by applying a MAC. The message is sent to the Device which will validate and check the message, including verifying the ACB's MAC.
If the checks are successful the Device will execute the Command. The Device will construct a

Response and apply a MAC that can be verified by the Remote Party, then send the Response to the ACB.
The ACB will pass the Response back to the Remote Party which will verify the MAC.

If the Command is a Critical Command the Remote Party will sign the Command to provide non-repudiation, before sending it to the ACB.
In this case the Device checks will include verifying the Remote Party's signature as well as the ACB's MAC.
If the checks are successful the Device will execute the Command.
The Device will construct a Response and sign it, then send the Response to the ACB. The ACB will pass the Response back to the Remote Party which will verify the signature.
Similarly, an Alert that is sent by a Device has a MAC applied that can be verified by the Remote Party. If it is a Critical Alert, it will have a signature rather than a MAC.

Where data items require confidentiality protection within a message, the AES GCM primitives (see below) are used to encrypt the data.

Each Device on the HAN (apart from Type 2 Devices) has its own Public-Private Key Pair, and a Device Security Credential to make its Public Key available, enabling it to be identified and authenticated. It is capable of securely holding a set of Security Credentials for Remote Parties with which it will need to communicate. It also maintains a Device Log in which it holds the Device Security Credentials of other Devices on the HAN with which it is authorised to communicate.

To communicate on the HAN, a Device must establish a secure ZigBee connection with the Communications Hub. The Communications Hub Function maintains its own Device Log that acts as a whitelist for those Devices allowed to communicate on the HAN. Device Security Credentials are added to a Device's Device Log by a command from an appropriate Remote Party, see [d, 13] for details.

Some messages require anti-replay protection as described in [d, 4.3]. Some messages may be futuredated as described in [d, 9.2].

The ZigBee HAN encryption uses the AES-128 cipher in CCM* mode with MMO as the hash function. Key establishment is achieved using Certificate-Based Key Establishment (CBKE), between a device and the Communications Hub which acts as a ZigBee Trust Center. Further details can be found in [f, 5.4] and [f, c.4].

Counters and their Use:

1) Remote Party Originator Counter:
The KRP or the ACB's Originator Counter. Originator Counters are always strictly numerically greater than any previous Originator Counter from that Message originator to the targeted Device. Originator Counters shall not use the UTRN reserved range unless as part of a Prepayment Top Up Command. Remote Parties may choose to increment a UTRN Originator Counter separately from other Originator Counters.

Purpose**:** The Originator Counter provides a unique Message identity (in combination with CRA Flag, sender id and recipient id). The Originator Counter is also used as an input value for symmetric Key Derivation Functions(KDF). The Originator Counter is used for
Protection Against Replay protection.

Impact on Device:The highest accepted value is stored as the Execution Counter or in the UTRN Counter cache as appropriate.

2) Device Originator Counter:
A Device's Originator Counter. This must be strictly numerically greater than any previous Originator Counter from that Device.

Purpose: The Originator Counter provides a unique Message Identity (in combination with CRA Flag, sender id and recipient id) . The Originator Counter is also used as an input value for symmetric Key Derivation Functions(KDF).

Impact on Device: The Device shall ensure that the value it generates (e.g. for Alerts) is strictly numerically greater than any previous Originator Counter value or Supplementary Originator Counter value it has placed in any previous Message it has generated

3) Supplementary Remote Party Counter:
The Originator Counter (or reference) of an Unknown Remote Party requesting the service from the ACB.

Purpose: The Supplementary Remote Party Counter supports Message identification of Responses by the URP as the originator of the service request associated to the Command.

Impact on Device: The Supplementary Remote Party Counter is incorporated into the corresponding Response by the Device. The Response also contains the Originator Counter of the ACB.

4) Supplememntary Originator Counter:
The Supplementary Originator Counter is a Device generated number which is strictly numerically greater than any previous Supplementary Originator Counter or Originator Counter placed in previous Messages by the Device). This is used in response to Commands.

Purpose: The Supplementary Originator Counter is used in a Response to a Command from an URP for the generation of symmetric keys for use in MAC creation and Encryption of sensitive values.

Impact on Device: The Device shall ensure that the value itgenerates (e.g. for Alerts) is strictly numerically greater than any previous Originator Counter or Supplementary Originator Counter values it has used in any previous Message. The Supplementary Originator Counter may be the same as Originator Counter in any given Message but this is an implementation decision).

5) Execution Counter:
The Execution Counter is the last accepted Originator Counter value for commands requiring Protection Against Replay and which cannot be future dated. It is stored by the Device for each Remote Party/Command combination. Note that only the Supplier (or for CHF the WAN Provider) can send Commands that require Protection Against Replay with the exception of the Update Security Credentials Command which can be sent by multiple roles.

Purpose: The Execution Counter is used to support Protection Against Replay of Commands for immediate execution. Where Commands are protected from Protection Against Replay then Devices will reject Commands where the Originator Counter in the Command not greater than the existing value of the Execution Counter stored on the Device.

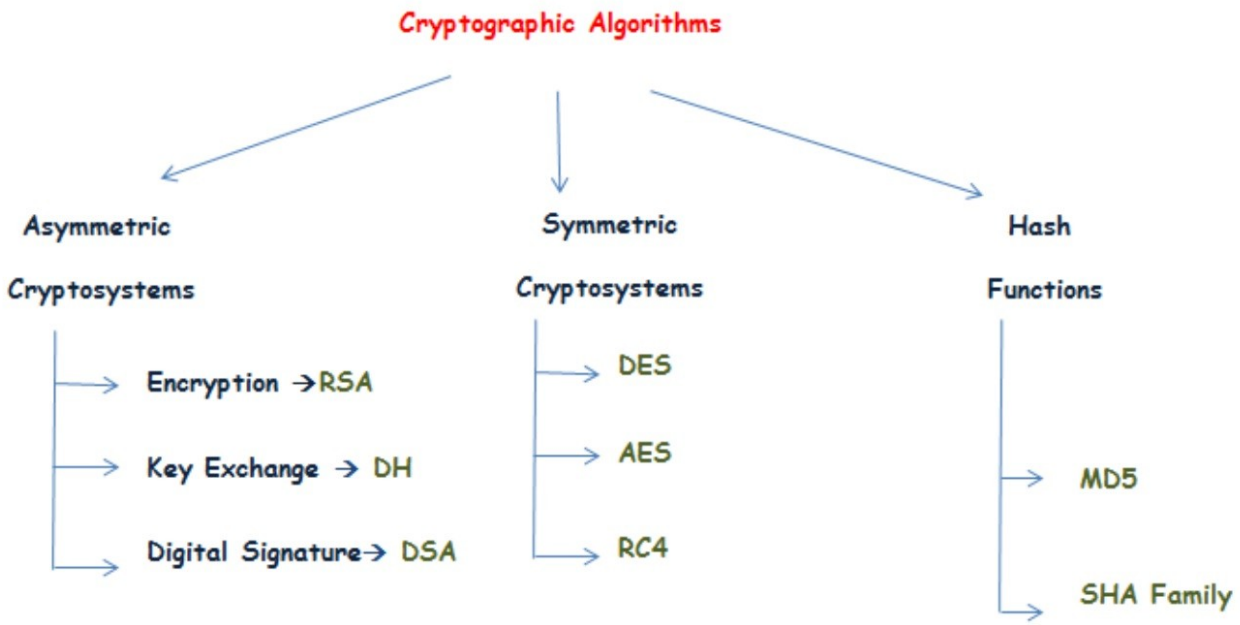
Impact on Device: Each Device will store an Execution Counter value for each KRP/Commandtype combination.

6) Invocation Counter:
The invocation field of the initialization vector. It is an integer counter which increments upon each invocation of the authenticated encryption function using the same key. When a new key is established the related invocation counter shall be reset to 0.

What is Cryptography?

Cryptography or **cryptology** is origin from word 'Kryptos' from Greek meaning "hidden or secret writing", is the practice and study of techniques for secure communication in the presence of third parties (called adversaries). It provides various aspect of information securety such as data confidentiality, data integrity, authentication, and non-repudiation.

Authentication: Authentication is a service used to provide the identity of an entity.
Confidentiality: Confidentiality is a service used to guarantee information it is accessible only to authorized entities and is inaccessible to others.
Integrity: Integrity is a service used to guarantee that the information remains unchanged from the source entity to the destination entity.
Non-repudiation: Non-repudiation is a service used to confirm the involvement of an entity in a certain form communication, and prevents any party from denying the sent message.



About Public Key Cryptography:(https://www.cs.rutgers.edu/~pxk/rutgers/notes/content/13-crypto.pdf**)**

Public key cryptography is the one-way function. This is a function where it is relatively easy to compute f(x) but extremely difficult to compute x, given f(x) (that is, the inverse function f -1 (x)). By extremely difficult, we mean a complexity that would take millions of years if all the computers

in the world were assigned to the problem. One often-cited way of thinking about a one-way function is to think of breaking a glass, or a plate. It's infinitely easier to break it than it is to put it back together.

So what good are these one-way functions? We can't use them for encryption (nobody would be able to decrypt the message). One particular form of a one-way function is the one-way hash function. This is also known as a message digest, fingerprint, cryptographic checksum, integrity check, manipulation detection code (MDC). The function takes variable-length input (the message) and computes a generally smaller, but fixed length, output which is the hash value. This value indicates whether the pre-image (the original message) is likely to be the same as the real message. While it is easy to compute the hash from a pre-image, it is (nearly) impossible to generate a pre-image that results in a given hash. The hash itself is a public function. No secrecy is needed. Its onewayness is its security. One way hashes are used for fingerprinting files. A variant of the one-way hash is the encrypted hash, known as a Message Authentication Code (MAC) or a Data Authentication Code (DAC). This is the same as the hash but is also a function of a secret key so that only the possessor of the key can verify the integrity of the message. This means that if the message is intercepted and altered, the encrypted hash cannot be computed by the perpetrator who does not possess the key.

Example of Transposition cipher(involves permuting the latters):

Suppose we have a message "If she weighs the same as a duck, she's made of wood" and a key "31415927", we can arrange the plaintext message under the key, wrapping around as needed:

3 1 4 1 5 9 2 7
I F S H E W E I
G H S T H E S A
M E A S A D U C
K S H E S M A D
O F W O O D X X

To transmit the coded message, we read out the text column-first, sorting the columns by the elements in the key (numbers in this case), obtaining:

FHESFHTSEOESUAXIGMKOSSAHWEHASOIACDXWEDMD

A recipient would simply arrange these in column-first order to fill eight columns and then move each column into its unsorted position. Both the number of columns and the positions are functions of the secret key.The problem with using a good transposition cipher is that these ciphers generally require a lot of memory and may require that messages be of certain lengths. If a cipher requires that a message be a multiple of a certain size, it is known as a **block cipher** and encryption is performed a block at a time. If the message is performed character by character and there is no requirement that a message be a specific size, the cipher is a **stream cipher.**

Coomunication:
We can engage in secure communication using symmetric cryptography, public key cryptography, or a hybrid system .
Communication with Symmetric key cryptography:
To communicate using symmetric cryptography, both parties have to agree on a secret key. After that, each message is encrypted with that key, transmitted, and decrypted with the same key.

Key distribution must be secret. If it is compromised, messages can be decrypted and users can be

impersonated. However, if a separate key is used for each pair of users, the total number of keys increases rapidly as the number of users increases. With n users, we would need [n(n-1)]/2 keys. Secure key distribution is the biggest problem in using symmetric cryptography.

Communication with public key cryptography:
Public key cryptography, by using a different key for decrypting than encrypting solves problems of key distribution. If Alice and Bob wish to communicate, Alice sends Bob her public key and Bob gives his public key to Alice. Alice then encrypts her message to Bob with Bob's public key, knowing that only Bob, the possessor of Bob's private key, can decrypt the message. Likewise, Bob encrypts his messages to Alice with Alice's public key. Public keys may be stored in a database or some well-known repository so that the keys do not have to be transmitted. Not only does public key cryptography solve key distribution, it also solves the problem of having [n(n-1)]/2 keys for n users. Now we only need 2n keys (n public and n private).

Communication with hybrid cryptosystems:
A common use of public key cryptography is to encrypt symmetric keys to solve the key distribution problem. It also enables a communicating party to pick a random key that will be valid for only one communication session. Suppose Alice and Bob wish to communicate. Alice sends Bob her public key. Bob then generates a random session key, encrypts it with Alice's public key, and sends it to Alice. Alice is now the only one who can decrypt the session key since only she has her private key, which is needed to decrypt the session key. After that, messages can be encrypted with the randomly generated session key. This type of cryptosystem, which relies on both public key and symmetric algorithms, is known as a hybrid cryptosystem.

Digital Signature:
We use signatures in today's society to identify ourselves uniquely. A signature has several important properties. A signature is considered to be:
- Authentic – the recipient knows that the signer deliberately signed the document.
- Unforgeable – the signature is proof that the signer signed the document.
- Not reusable – the signature is part of the document and cannot be copied onto another document.
- Unalterable – once a document is signed, it cannot be changed.
- Nonrepudiatable – the signer cannot claim that he/she didn't sign the document.

It is these properties that make signatures so indispensable in society. Unfortunately, all of these properties are completely untrue. Luckily, forging signatures and altering documents is often quite a bit of trouble. In the digital domain, however, copying and modifying files is trivially easy. We'd like the properties of signatures without the problems. For example, Alice should be able to "sign" a document so others would know that it was really Alice and that the document was not altered after she signed it.

Arbitrated protocol:
We can turn to a trusted third party (or arbiter) to authenticate our messages. In this case, a third party, Trent, has the symmetric key of every user. Trent is a trusted party – he will not forge messages or give away keys. If Alice wishes to send a message to Trent, she composes a message, including the destination (Bob), and encrypts it for herself: EA(M). She then sends this message to Trent. Since Trent has all the keys, he can decrypt the message and know that it could have originated only from Alice (since Alice is the only other party that has Alice's key). Trent then generates a statement of receipt, adds it to the message, and encrypts the message for Bob: EB(M). He may also choose to log a record of this message along with a hash of the message. When Bob receives the message, he uses his own key to open it, knowing that it could have come only from him (it didn't) or from Trent (nobody else has Bob's key). Since Trent is a trusted party, Bob trusts Trent's attestation that the message really did originate from Alice.

Going one step further, if David wants to forward the message to Charles, he encrypts the message (along with Trent's attestation to the origin of the message from Alice) along with a destination using his own key. The encrypted message is sent to Trent, who can then look up the hash of the message in his database to ensure that it was not altered by Bob. Trent then adds his attestation that the message was also "signed" by Bob, encrypts it for Charles and sends it to him.

Digital signatures and public key cryptography:
With public key cryptography, encrypting a message with one's private key is the same as signing the message! Anybody with access to the public key can decrypt the message but will know that the document could have been encrypted by the possessor of the private key. No third party is needed.

We can generate a stand-alone fixed-length signature for a message by creating a hash of the document H(M) and then encrypting the hash with our private key. If a recipient wishes to verify the signature, it produces a hash of the document and decrypts the hash we sent by using our public key. If the hashes match, the document has not been altered. This scheme makes it easy to attach multiple signatures: each party computes a hash of the message, encrypts it with its private key, and attaches it to the message. Another advantage is speed: we do not have to encrypt the entire message using public key cryptography.

If nonrepudiation is desired, we will need to turn a third party. The originator will add a header to the message containing identifying information (name, timestamp), sign the message, and send it to the third party. The third party will then add its own timestamp and ID to the message, log the transaction, and send it to both the sender and recipient (so the sender will know if someone is trying to impersonate her).

If secrecy of the message is desired, encryption can be combined with the digital signature, providing privacy as well as proof of authorship. To do this, we can pick a random key, K, with which to encrypt the message (using a symmetric algorithm). This key will then be encrypted with the public key of each recipient of the message. A recipient will be able to decrypt K with his private key, then decrypt the message, compute the hash, decrypt the hash attached to the message (decrypted with the sender's public key), and verify the origin and authenticity of the message.

Let's Look at this again,Alice has a message, M, to send to Bob. She computes its hash, H(M) and encrypts it with her own private key: Ea(H(M)) . This is her signature. Secrecy of the message is important in this example, so she will encrypt the signed message with a symmetric algorithm using a randomly generated key, K. The encrypted signed message is EK({M,Ea(H(M))}). Now she has to enable only Bob to be able to decrypt this message, so she encrypts the key, K, with Bob's public key: EB(K). Finally, she sends out the complete message: { EK({M,Ea(H(M))}), EB(K) }.

When Bob gets this message. He first decrypts the key, K, using his private key. Now, using K, he can decrypt the entire message with signature. Having done this, he computes a hash of M, H(M). He then decrypts Alice's signature using Alice's public key and compares the two hashes to validate the message.

Combined authentication and key exchange
If we can combine authentication with key exchange, then two parties across a network can exchange keys and be sure that they're communicating with each other.

1) Wide_Mouth Frog:

A protocol that accomplishes key exchange and authentication using symmetric cryptography is the

Wide-Mouth Frog algorithm. It uses an arbitrated protocol where one party encrypts a message for itself containing the key and sends it to the trusted third party. This third party decrypts the message and re-encrypts it for the recipient. The problem of having n(n-1)/2 keys is avoided because the secret keys are only for the third party. To prevent replay attacks (somebody snooping on the message and sending it at a later time), a timestamp is added to each message.
If Alice wants to talk to Bob, she sends a message to Trent (the third party) encrypted with her key (A).

Alice to Trent: { "Alice", EA(TA, "Bob", K) }

Trent receives the message and sees that it's from Alice. He looks up her key in his database and decrypts the rest of the message. He verifies the timestamp TA to determine whether to accept the message. Seeing that it's for Bob, he looks up Bob's key and composes a new message (using A new timestamp TT):

Trent to Bob: { EB(TT, "Alice", K) }

2) Diffie-Hellman exponential key exchange(Used in Key Agreement):
Diffie-Hellman is the first public key algorithm. Its use is different from RSA public-key cryptography in that it is only suitable for key exchange, not encryption. The publicly readable data is not really a key that will be used for encryption or decryption. The algorithm is based on the difficulty of calculating discrete logarithms in a finite field compared to the ease of calculating exponentiation. Exponential key exchange allows us to negotiate a secret session key without the fear of eavesdroppers.
To perform this algorithm, all arithmetic operations are performed in the field of integers modulo some large number (modulo means that we divide the results by and keep the remainder). Both parties then agree on some large prime number, p, and a number α, where $\alpha < p$ and α is a primitive root of p.
Each party then generates a public/private key pair. The private key for user i is Xi ,
which is just a random number less than q.
The corresponding public key, Yi , is computed as: **$Y_i = \alpha_B^{X_i} \bmod p$.**

Now, suppose that Alice and Bob wish to talk. Alice has a secret key XA and a public key YA and Bob has a secret key XB and a public key YB.
1. Alice sends Bob her public key, YA.
2. Bob sends Alice his public key, YB.
3. Alice computes:

$$K = (Y_B)^{X_A} \bmod p$$

4. Bob computes:

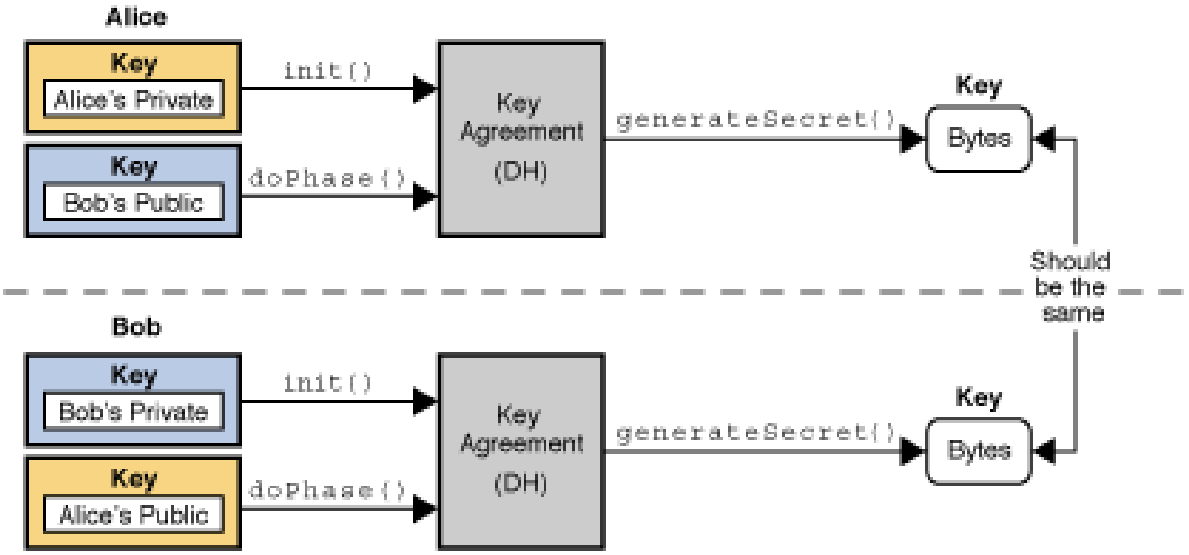
$$K = (Y_A)^{X_B} \bmod p$$

5. Alice and Bob can now use symmetric encryption using the same key. This key is called **shared key K.**

The essential point is that both Alice and Bob could generate a common shared key using their private key and the other's public key but nobody else could do so. The keys are equivalent because:

$$K = (Y_A)^{X_B} \bmod p = (\alpha^{X_A} \bmod p)^{X_B} \bmod p = \alpha^{X_A X_B} \bmod p$$

Now that two parties can derive a common conversation key (that only they can derive) , one of them can pick a random token and send it to the other for encryption as was done in the SKID/2 or SKID/3 protocol.



Key Derivation Function(KDF) (http://csrc.nist.gov/publications/nistpubs/800-56C/SP-800-56C.pdf): In cryptography, a **key derivation function** (or **KDF**) derives one or more secret keys from a secret value such as a master key or other known information such as a password or passphrase using a pseudo-random function.

Key derivation functions are often used in conjunction with non-secret parameters to derive one or more keys from a common secret value (which is sometimes also referred to as "key diversification"). Such use may prevent an attacker who obtains a derived key from learning useful information about either the input secret value or any of the other derived keys. A KDF may also be used to ensure that derived keys have other desirable properties, such as avoiding "weak keys" in some specific encryption systems.

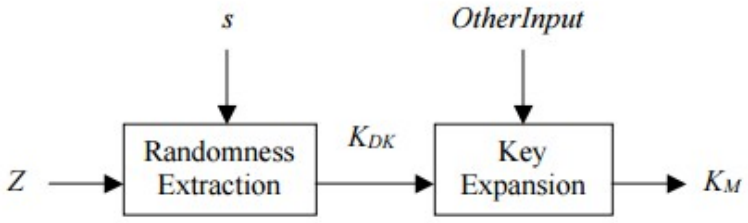


Figure 1: Extraction-then-Expansion Procedure

The extraction-then-expansion key derivation procedure specified in this Recommendation begins with a **shared secret Z** that is a byte string established during an execution of a public-key-based key establishment scheme specified in NIST SP 800- 56A or NIST SP 800-56B.
The **randomness extraction** step uses HMAC as defined in FIPS 198-1 or AES CMAC as defined in NIST SP 800-38B, with a byte string s (called the salt) as the "key", and the shared secret Z as the "message". The output of the randomness extraction step is a key derivation key KDK.
The **key expansion** step uses the key derivation key KDK and other information exchanged and/or pre-shared, such as identifiers for the involved parties, protocol identifiers, and the labels of the derived keys, as the input to an approved key derivation function specified in SP 800-108 to produce secret keying material KM of a desired length L.

In the extraction step, the following notations are used,
MAC – The MAC function used during the extraction step,MAC is either HMAC-hash, an (untruncated) instantiation of the HMAC function employing the hash function hash, or AES-CMAC, an (untruncated) instantiation of the CMAC function employing the AES block cipher.

s – Salt- a (public or secret) byte string used as the "key" during the execution of the randomness extraction step. The length of s is determined by the MAC function used in the extraction step as shown below,

-> **HMAC-hash algorithms** can accommodate keys of any length up to the maximum bit

length permitted for input to the hash function hash. If the bit length of an HMAC key is greater than the input block length for hash, that key is replaced by its hash value.

-> **AES-CMAC** requires key lengths to be 128, 192, or 256 bits. The bit length of the salt s shall be the same length as the AES key (i.e., 128, 192, or 256 bits).

The salt could be, for example, a value computed from nonces exchanged as part of a key establishment protocol, a value already shared by the protocol participants, or a value that is pre-determined by the protocol. If there is no means to select a salt and to share it with all participants, then the salt shall be the all-zero byte string. If HMAC-hash is used, the bit length of the all-zero byte string shall equal that of the input block for hash. If AES-CMAC is used, the bit length of the all-zero byte string shall equal the length of the AES key used.

Z - A shared secret established during an execution of an approved public keybased key establishment scheme. It is represented as a byte string and used as the "message" in a MAC execution in the randomness extraction step. Each call to the randomness extraction step requires a freshly-computed shared secret Z, and this shared secret shall be zeroized immediately following its use in the extraction process.

$\mathbf{K_{DK}}$ - The output of the randomness extraction step. When HMAC-hash is used, it is a binary string of length h, where h is the output length in bits of the hash function hash. When AES-CMAC is used, it is a binary string of length 128 bits.

Popular Algorithm

MD5: MD5 is a message digest (hash) algorithm created by Ron Rivest of MIT (the 'R' in RSA). Its input is a message of arbitrary length and its output is a 128 bit message digest.

SHA: SHA stands for Secure Hash Algorithm. It is a hash algorithm based on MD4, a precursor to MD5, created by the National Institute of Standards and Technology in 1993. It produces a 160-bit message digest. Because of the longer output, it is harder to produce another message that yields the same digest. On the other hand, it requires more computational steps than MD5 (80 vs. 60), making it approximately 25% slower.

SHA-256: SHA-256 (secure hash algorithm, FIPS 182-2) is a cryptographic hash function with digest length of 256 bits. It is a keyless hash function; that is, an MDC (Manipulation Detection Code). A message is processed by blocks of 512 = 16 × 32 bits, each block requiring 64 rounds.

The SHA-256 compression function operates on a 512-bit message block and a 256- bit intermediate hash value. It is essentially a 256-bit block cipher algorithm which encrypts the intermediate hash value using the message block as key. Hence there are two main components to describe: (1) the SHA-256 compression function, and (2) the SHA-256 message schedule.

Message Size: 2^64 bit
Block Size: 512bit
Word Size: 32bit
Message Digest Size: 256

ISO authentication framework:
Public key systems have great appeal for key exchange, but if we want to use them as a basis for authentication, we need something that will bind one's identity to the public key. If you ask for Fred's public key, how can you be sure that it really belongs to Fred and not some imposter? How

can you find key identifying information about Fred and his public key without contacting him? One option is to maintain a centralized database of public keys. The problem with this is that one must always turn to this trusted source for the distribution of keys: you cannot trust a key that was passed on to you by an untrusted party.

The International Standards Organization (ISO) introduced a set of protocols known as **X.509** protocols to provide standards for authentication across networks. While no standard algorithms are specified, RSA public key encryption is recommended.

The most important part of the ISO framework is the structure for public key certificates. Each user has a unique name (called a distinguished name) that is a collection of several attributes including the user's real name, organization, locality, and country. A trusted certification authority (CA) issues a signed certificate that contains the distinguished name and the user's public key. This certificate is signed by the CA. A certificate looks like this:

version	serial #	algorithm, params	issuer	validity: from, to	distinguished name	Pub. Key: alg, params, key	signature of CA

If Marge wants to talk to Homer and they have a common CA, she can get his certificate from some database (or some source). She can then verify the signature of the CA (by hashing the contents of the certificate and comparing them with the CA's signature decrypted with the CA's public key). This gives Marge assurance that Homer's certificate was indeed generated by the same certification authority.

A more complicated problem is when a different certification authority certified Homer than Marge. Certification authorities are designed to fit into a hierarchical structure. Each CA has a certificate signed by the CA above it and by the CA below it. If you have a certificate signed by an unknown CA, you can ask the CA for its key and see which higher-level signed it. If that is also unknown, you can repeat the process until a CA is reached (a common root). This is known as certificate chaining. If a common root isn't reached then you may choose not to honor it.

The process of certificate discovery is not consistently implemented at this time. One possible mechanism is for Marge to move up her chain of certification authorities and look at the graph of CAs below, searching for the CA which certified Homer. A more reasonable approach may be for Marge to know her entire chain of CAs and follow Homer's chain until a common CA is located and Homer's identity may be trusted.

Certification Path Validation:

The **certification path validation algorithm** is the algorithm which verifies that a given **certificate path** is valid under a given public key infrastructure (PKI). A path starts with the Subject certificate and proceeds through a number of intermediate certificates up to a trusted root certificate, typically issued by a trusted Certification Authority (CA).

Public Key Infrastructure (PKI) supports a number of security-related services, including data confidentiality, data integrity, and end-entity authentication. Fundamentally, these services are based on the proper use of public/private key pairs. The public component of this key pair is issued in the form of a public key certificate and, in association with the appropriate algorithm(s), it may be used to verify a digital signature, encrypt data, or both.

Before a certificate can be used, it must be validated. In order to validate such a certificate, a chain of certificates or a certification path between the certificate and an established point of trust must be established, and every certificate within that path must be checked. This process is referred

to as certification path processing. In general, certification path processing consists of two phases: 1) Path construction and 2) Path validation described as follows: 1) **Path construction** involves "building" one or more candidate certification paths. Note that we use "candidate" here to indicate that although the certificates may chain together properly, the path itself may not be valid for other reasons such as path length, name, or certificate policy constraints/restrictions. 2) Path validation includes making sure that each certificate in the path is within its established validity period, has not been revoked, has integrity, et cetera; and any constraints levied on part or all of the certification path are honored (e.g., path length constraints, name constraints, policy constraints). However, some aspects that might be associated with path validation are sometimes taken into consideration during the path construction process in order to maximize the chances of finding an acceptable certification path sooner rather than later.

Certificates can also be revoked by a CA and each CA is responsible for maintaining a **Certificate Revocation List(CRL).**

Certificate Revocation List (CRL):

A CRL is a Certificate Revocation List. When any certificate is issued, it has a validity period which is defined by the Certification Authority. Usually this is one or two years. Any time a certificate is presented as part of an authentication dialog, the current time should be checked against the validity period. If the certificate is past that period, or expired, then the authentication should fail. However, sometimes certificates should not be honored even during their validity period. For example, if the private key associated with a certificate is lost or exposed, then any authentication using that certificate should be denied. When their certificates are replaced, the old certificates have to be marked somehow as “no longer accepted”. **The purpose of the CRL is to list certificates which are valid, but are revoked.** A CRL is generated and published periodically, often at a defined interval. A CRL can also be published immediately after a certificate has been revoked. The CRL is always issued by the CA which issues the corresponding certificates. All CRLs have a lifetime during which they are valid; this timeframe is often 24 hours or less. During a CRL's validity period, it may be consulted by a PKI-enabled application to verify a certificate prior to use. To prevent spoofing or denial-of-service attacks, CRLs usually carry a digital signature associated with the CA by which they are published. To validate a specific CRL prior to relying on it, the certificate of its corresponding CA is needed, which can usually be found in a public directory (e.g., preinstalled in web browsers).The certificates for which a CRL should be maintained are often X.509/public key certificates, as this format is commonly used by PKI schemes.

Certificate-based authentication is either one-way, two-way, or three-way:

one-way: Marge authenticates herself to Homer
two-way: Mutual authentication: Marge authenticates herself to Homer and gets a reply from Homer to establish his identity
three-way: Another message is added from Marge to Homer to avoid the need for timestamps (and authenticated, synchronized time).

The authentication from Marge to Homer works as follows:

1. Marge generates a invoke counter, IC. She constructs a message: M = {Tm, IC, Ih, d } where, Tm a timestamp, Ih is an identifier for Homer, and d is arbitrary data.
2. Marge sends her certificate, M encrypted for Homer and sighed by Marge.
3. Homer verifies Marge's certificate and obtains her public key. He then decrypts M using his private key and verifies Marge's signature.
4. Homer checks Ih for accuracy to ensure that Marge really intended to address him and has the

right identifier for him. He then checks the timestamp to see whether it's a current message. Optionally, Homer may check IC in a database of previously used counter value if he's concerned about a replay attack and doesn't trust the timestamp.

To support two-way authentication, Homer would send back the original IC sent by Marge. For three-way authentication, Marge would encrypt the IC sent by Homer and send it back to him.

GCM: This Recommendation specifies an algorithm called **Galois/Counter Mode (GCM)** for authenticated encryption with associated data. GCM is constructed from an approved symmetric key block cipher with a block size of 128 bits, such as the Advanced Encryption Standard (AES) algorithm that is specified in Federal Information Processing Standard (FIPS) Pub. 197. Thus, GCM is a mode of operation of the AES algorithm.

GCM provides assurance of the confidentiality of data using a variation of the Counter mode of operation for encryption. GCM provides assurance of the authenticity of the confidential data (up to about 64 gigabytes per invocation) using a universal hash function that is defined over a binary Galois (i.e., finite) field. GCM can also provide authentication assurance for additional data (of practically unlimited length per invocation) that is not encrypted. If the GCM input is restricted to data that is not to be encrypted, the resulting specialization of GCM, called GMAC, is simply an authentication mode on the input data. **Galois Message Authentication Code(GMAC)** is an authentication-only variant of the GCM which can be used as an incremental message authentication code. Both GCM and GMAC can accept initialization vectors of arbitrary length.

Elliptic Curve Digital Signature Algorithm(http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf):

The Elliptic Curve Digital Signature Algorithm (ECDSA) is the elliptic curve analogue of the DSA. ECDSA was first proposed in 1992 by Scott Vanstone in response to NIST's (National Institute of Standards and Technology) request for public comments on their first proposal for DSS. It was accepted in 1998 as an ISO (International Standards Organization) standard (ISO 14888-3), accepted in 1999 as an ANSI (American National Standards Institute) standard (ANSI X9.62), and accepted in 2000 as an IEEE (Institute of Electrical and Electronics Engineers) standard (IEEE 1363-2000) and a FIPS standard (FIPS 186-2). It is also under consideration for inclusion in some other ISO standards. In this paper, we describe the ANSI X9.62 ECDSA, present rationale for some of the design decisions, and discuss related security, implementation, and interoperability issues.

ECDSA is concerned with asymmetric digital signatures schemes with appendix. "Asymmetric" means that each entity selects a key pair consisting of a private key and a related public key. The entity maintains the secrecy of the private key which it uses for signing messages, and makes authentic copies of its public key available to other entities which use it to verify signatures. "Appendix" means that a cryptographic hash function is used to create a message digest of the message, and the signing transformation is applied to the message digest rather than to the message itself.

MAC & Encryption Understanding:

Encrypt-then-MAC: Provides integrity of Ciphertext. Assuming the MAC shared secret has not been compromised, we ought to be able to deduce whether a given ciphertext is indeed authentic or has been forged; for example, in public key cryptography anyone can send you messages. EtM ensures you only read valid messages. EtM is the most ideal scenario. Any modifications to the ciphertext that do not also have a valid MAC code can be filtered out before decryption, protecting

against any attacks on the implementation.

Encryption provides confidentiality, a MAC provides integrity. Using encryption alone makes your messages vulnerable to a ciphertext only attack. An example will make it more clear. Say you send a message that says:

M = "transfer 100$ to account 591064"

The sender, with the symmetric key, can encrypt the message and send E(M). No one should be able to send a valid message other than the holder of the key. You have confidentiality covered. But an attacker could alter the ciphertext to make it say something else when decrypted. Obviously, the larger the message and the more structure it has, the harder it gets to carry out in practice. Now if you use a MAC along with encryption, you will be able to detect changes to the cipher text because the MAC will not compute. In our example, if you use the same key for encryption and MAC, then you can change your message to:

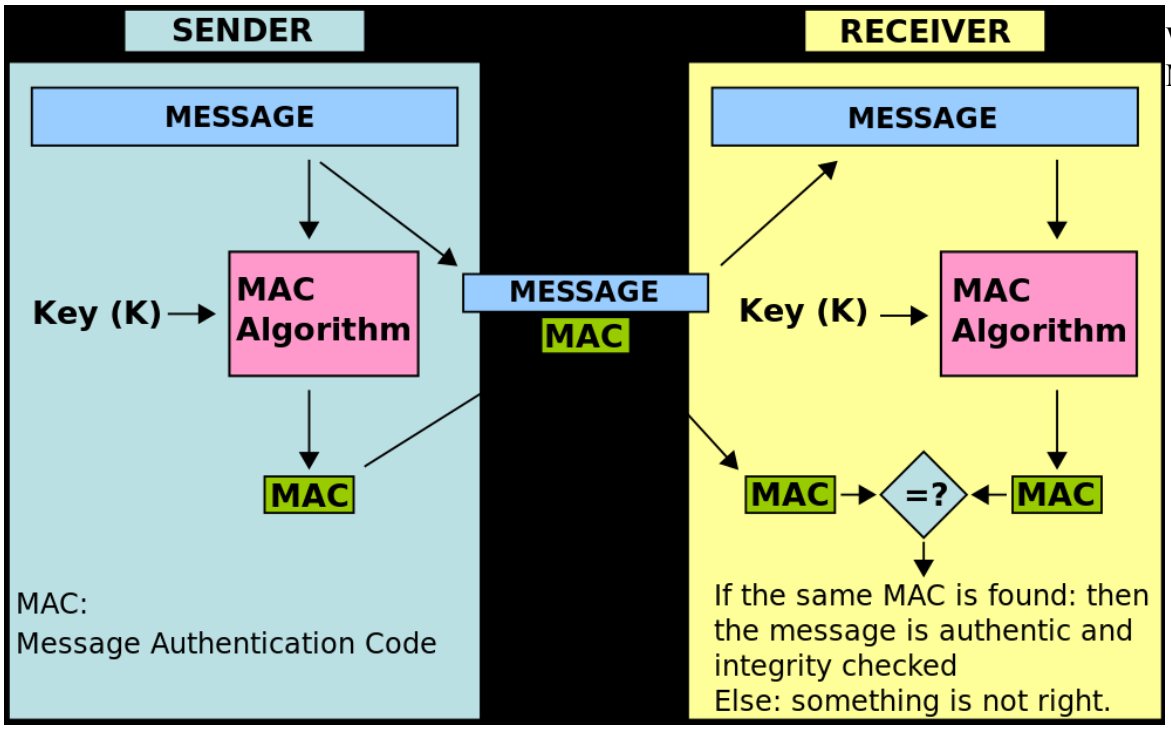
M = "transfer 100$ to account 591064 | a46c0db15acdd36b4e92a82e5dc6c14f"

and encrypt it, again sendingE(M). The hash is encrypted (that's your MAC), the message is encrypted (for confidentiality). That way, you make it computationally impossible to alter the cipher text and come up with a valid message, even if your message is a single, random byte.

In conclusion:

- Encryption does not provide integrity by itself
- MAC (integrity) does not provide confidentiality by itself

You often have to combine cryptographic primitives to achieve many security properties.

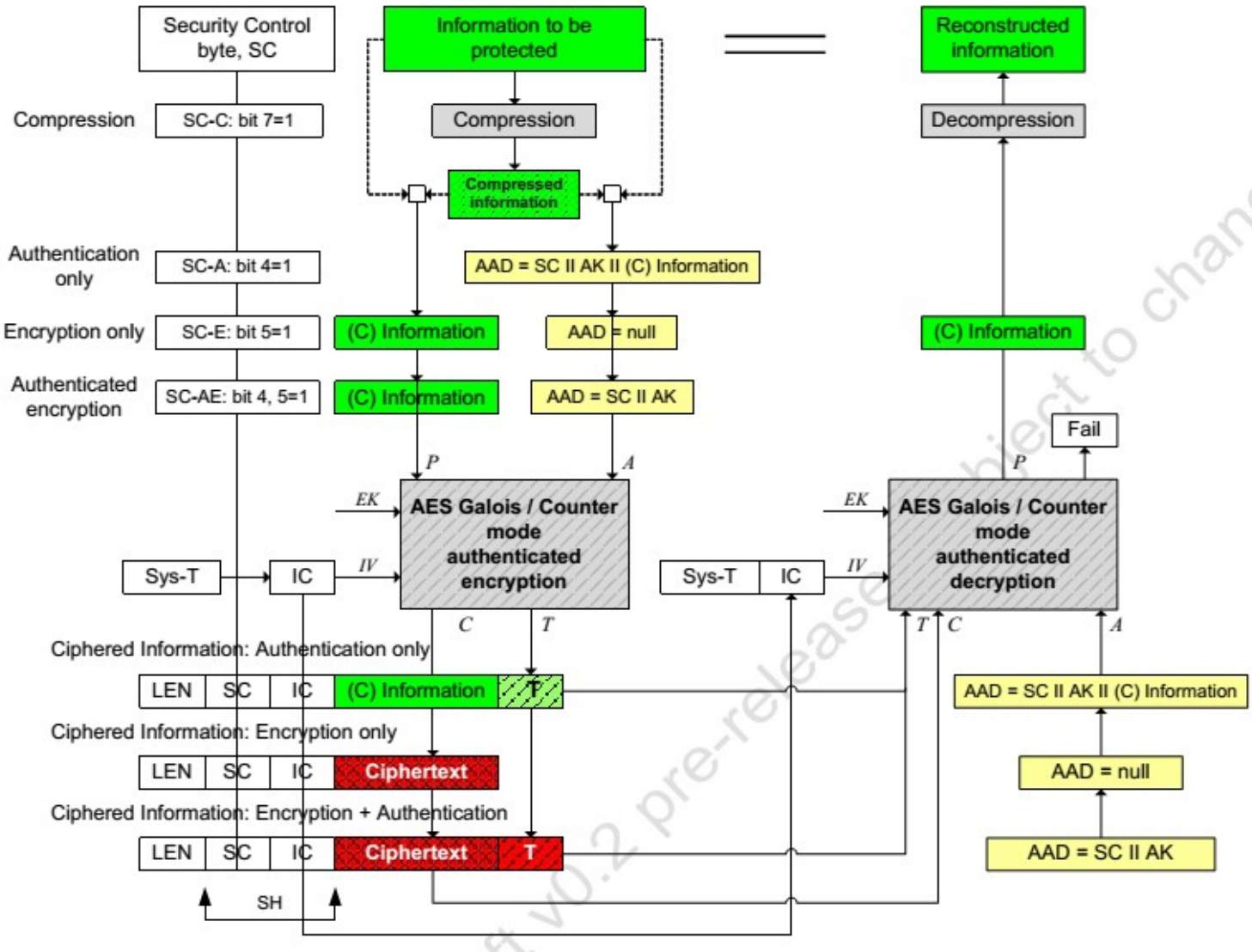


While MAC functions are similar to cryptographic hash functions, they possess different security requirements. To be considered secure, a MAC function must resist existential forgery under chosen-plaintext attacks. This means that even if an attacker has access to an oracle which possesses the secret key and generates MACs for messages of the attacker's choosing, the attacker cannot guess the MAC for other messages (which were not used to query the oracle) without performing infeasible amounts of computation.

MACs differ from digital signatures as MAC values are both generated and verified using the same

secret key. This implies that the sender and receiver of a message must agree on the same key before initiating communications, as is the case with symmetric encryption. For the same reason, MACs do not provide the property of non-repudiation offered by signatures specifically in the case of a network-wide shared secret key: any user who can verify a MAC is also capable of generating MACs for other messages. In contrast, a digital signature is generated using the private key of a key pair, which is public-key cryptography. Since this private key is only accessible to its holder, a digital signature proves that a document was signed by none other than that holder. Thus, digital signatures do offer non-repudiation. However, non-repudiation can be provided by systems that securely bind key usage information to the MAC key; the same key is in possession of two people, but one has a copy of the key that can be used for MAC generation while the other has a copy of the key in a hardware security module that only permits MAC verification.

Encryption, authentication and compression Process in DLMS(as given in green book):



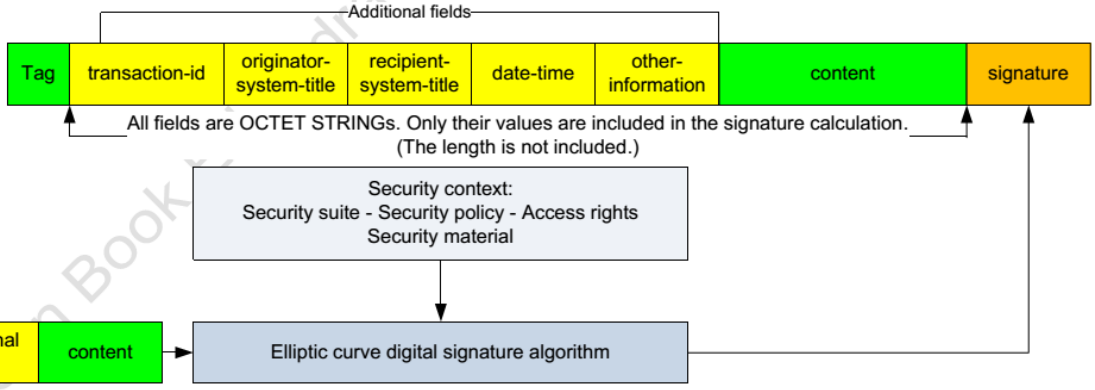
Security Control(SC):

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3..0
Compression	Key_Set	E	A	Security_Suite_Id
The Key_Set bit is not relevant and shall be set to 0 when the service specific dedicated ciphering, the general-dedicated-ciphering or the general-ciphering APDUs are used.				

Security control, *SC*		**Protection**	***P***	***A, Additional Authenticated Data***	
E field	**A field**			**service specific-glo-ciphering** **service specific-ded-ciphering** **general-glo-ciphering** **general-ded-ciphering**	**general-ciphering** [1]
0	0	None	–	–	–
0	1	Authenticated only	–	*SC* ll *AK* ll *(C) I*	*SC* ll *AK* ll transaction-id ll originator-system-title ll recipient-system-title ll date-time ll other-information ll *(C) I*
1	0	Encrypted only	*(C) I*	–	–
1	1	Encrypted and authenticated	*(C) I*	*SC* ll *AK*	*SC* ll *AK* ll transaction-id ll originator-system-title ll recipient-system-title ll date-time ll other-information
1) For the elements contributing to AAD only the values of the octet strings are included. The length of the octet strings is protected by the nature of the algorithm that is used to calculate the message authentication code (MAC). This is because it is computationally infeasible to find any two distinct AADs that would have the same message authentication code.					

– global-get-request APDU :

Digital



	X	Contents			*LEN (X)* bytes	*Len (X)* bits
Security						
unicast key)		10	20	30		
Security header	*SH*	*SH = SC-A* ‖ IC	*SH = SC-E* ‖ IC	*SH = SC-AE* ‖ IC		
		1001234567	2001234567	3001234567	5	40
Inputs		**Authentication**	**Encryption**	**Authenticated encryption**		
xDLMS APDU to be protected	*APDU*	C001000008000001 0000FF0200 (Get-request, attribute 2 of the Clock object)			13	104
Plaintext	*P*	Null	C001000008000001 0000FF0200	C001000008000001 0000FF0200	13	104
Associated data	*A*	*SC* ‖ *AK* ‖ *APDU*	–	*SC* ‖ *AK*		
Associated Data – Authentication	*A-A*	10D0D1D2D3D4D5D6 D7D8D9DADBDCDDDE DFC0010000080000 010000FF0200	–	–	30	240
Associated Data – Encryption	*A-E*	–	–	–	0	0
Associated Data – Authenticated encryption	*A-AE*	–	–	30D0D1D2D3D4D5D6 D7D8D9DADBDCDDDE DF	17	136
Outputs		**Authentication**	**Encryption**	**Authenticated encryption**		
Ciphertext	*C*	NULL	411312FF935A4756 6827C467BC	411312FF935A4756 6827C467BC	13	104
Authentication tag	*T*	06725D910F9221D2 63877516	–	7D825C3BE4A77C3F CC056B6B	12	96
The complete Ciphered APDU		*TAG* ‖ *LEN* ‖ *SH* ‖ *APDU* ‖ *T*	*TAG* ‖ *LEN* ‖ *SH* ‖ *C*	*TAG* ‖ *LEN* ‖ *SH* ‖ *C* ‖ *T*	–	–
Authenticated APDU		C81E1001234567C0 0100000800000100 00FF020006725D91 0F9221D263877516	–	–	32	256
Encrypted APDU		–	C812200123456741 1312FF935A475668 27C467BC	–	20	160

Length of Various Parameter mentioned in GBCS document

//.....//

Transation-ID= 9 bytes =0x04 || Encryption Originator Counter (0x04 ensures this value is not used in other KDF invocation)

Initialization Vector(IV)= 96 bits which contains, FixedField || InvocationField
where, Fixed Field= Entity Identyfier(64bit)
Invocation Field=0x00000000(32bit)

OtherInfo used execute KDF contains=
AlgorithmID || value of originator-system-title || length of transaction-id || transaction-id || value of recipient-system-title from Group. Header

where, AlgorithmID= 0x60857406080300 (for AES GCM-128) (7 bytes)
value of originator/recipient-system-title=same as entity identifie r=64 bits=8 bytes
so **total size= 7 + 8 + 1 + 9 + 8 = 33 bytes**

Size of MAC= 96 bits = 12 bytes

Digital Signing Private Key= 32 bytes
Digital Signing Public Key= 64 bytes
Key Agreement Private Key= 32 bytes
Key Agreement Public Key= 64 bytes
The Share Secret Key= 32 bytes

Grouping Header Total Size= 34 bytes
MAC Header Total Size= 13 to 15 bytes (depending on the encoded length of the whole remaining APDU)

//.....//

From RPMSECURITY_Services.c

Trust Anchor Info:

```
typedef struct {
   ASN1_RemotePartyRole role;
   ASN1_KeyUsage keyUsage;
   ASN1_CellUsage cellUsage;
   int32_t credRole;
} TrustAnchorInfo;
```

```
-> CosemAlgo_AES_GCM128_ID[Algorithm_ID_Size=7]={ 0x60, 0x85, 0x74, 0x06, 0x08, 0x03, 0x00 };
```

1)To create other info data used for KDF for a msg,

```
static void rpmsecurity_CreateOtherinfo(const grouping_header_t *grouping_header, uint8_t otherinfo[OTHERINFO_SIZE]);
```

```
OTHERINFO_SIZE=ALGORITHM_ID_SIZE+BUSINESS_ID_SIZE+TRANSACTION_ID_LEN_SIZE+TRANSACTION_ID_SIZE+BUSINESS_ID_SIZE
```

```
static void rpmsecurity_CreateOtherinfo(const grouping_header_t *grouping_header, uint8_t otherinfo[OTHERINFO_SIZE])
```

```
{
   uint8_t *p_otherinfo;

   p_otherinfo = otherinfo;
   (void)memcpy(p_otherinfo, cosemAlgoID_aesGcm128, sizeof cosemAlgoID_aesGcm128);
   p_otherinfo += sizeof cosemAlgoID_aesGcm128;
         (void)memcpy(p_otherinfo,   grouping_header->gh_BusinessOriginatorID,   sizeof
grouping_header->gh_BusinessOriginatorID);
   p_otherinfo += sizeof grouping_header->gh_BusinessOriginatorID;
         (void)memcpy(p_otherinfo,   &grouping_header->gh_TransactionIdLength,   sizeof
grouping_header->gh_TransactionIdLength);
   p_otherinfo += sizeof grouping_header->gh_TransactionIdLength;
    (void)memcpy(p_otherinfo, grouping_header->gh_TransactionId, sizeof grouping_header-
>gh_TransactionId);
   p_otherinfo += sizeof grouping_header->gh_TransactionId;
            (void)memcpy(p_otherinfo,    grouping_header->gh_BusinessTargetID,    sizeof
grouping_header->gh_BusinessTargetID);
   p_otherinfo += sizeof grouping_header->gh_BusinessTargetID;
   if (p_otherinfo != otherinfo + OTHERINFO_SIZE)
   {
      LOG_FATAL("Logic error");
   }
}
```

2) To create other info data used in KDF for encryption,

```
static void rpmsecurity_CreateOtherinfoForEncryption(const uint8_t
originatorId[BUSINESS_ID_SIZE],const uint8_t targetId[BUSINESS_ID_SIZE], uint64_t
originatorCounter, uint8_t otherinfo[OTHERINFO_SIZE]);
```

```
static void rpmsecurity_CreateOtherinfoForEncryption(const uint8_t
originatorId[BUSINESS_ID_SIZE],
      const uint8_t targetId[BUSINESS_ID_SIZE], uint64_t originatorCounter, uint8_t
otherinfo[OTHERINFO_SIZE])
{
   uint8_t *p_otherinfo;
   uint8_t counterBytes[REMOTE_PARTY_COUNTER_SIZE];
   uint32_t i;

   for(i = 0; i < 8; ++i)
   {
      counterBytes[i] = (originatorCounter >> (56 - i * 8)) & 0xFF;
   }

   p_otherinfo = otherinfo;
   (void)memcpy(p_otherinfo, cosemAlgoID_aesGcm128, sizeof cosemAlgoID_aesGcm128);
   p_otherinfo += sizeof cosemAlgoID_aesGcm128;
   (void)memcpy(p_otherinfo, originatorId, BUSINESS_ID_SIZE);
   p_otherinfo += BUSINESS_ID_SIZE;
   *p_otherinfo++ = TRANSACTION_ID_SIZE;
   *p_otherinfo++ = 0x04; /* Normally CRA flag, different value for encryption */
   (void)memcpy(p_otherinfo, counterBytes, sizeof counterBytes);
   p_otherinfo += sizeof counterBytes;
   (void)memcpy(p_otherinfo, targetId, BUSINESS_ID_SIZE);
   p_otherinfo += BUSINESS_ID_SIZE;
   if (p_otherinfo != otherinfo + OTHERINFO_SIZE)
   {
      LOG_FATAL("Logic error");
   }
}
```

3) Allocate and fill in the actual bytes to sign, taken from (parts of) the grouping header and the payload. Used both for signing and validating signatures. The caller must BUFFER_Release the returned value if not NULL

```
static uint8_t *rpmsecurity_AllocCreateDataToSign(const grouping_header_t
*grouping_header, const uint8_t *payload, uint32_t payload_len, size_t
*to_sign_len_out);
```

```
static uint8_t *rpmsecurity_AllocCreateDataToSign(const grouping_header_t
*grouping_header, const uint8_t *payload, uint32_t payload_len, size_t
*to_sign_len_out)
{
   const uint8_t *p;
   uint8_t *to_sign, *p_to_sign;

   *to_sign_len_out = sizeof grouping_header->gh_TransactionId + sizeof
grouping_header->gh_BusinessOriginatorID +
         sizeof grouping_header->gh_BusinessTargetID + MESSAGE_ID_SIZE + payload_len;
   to_sign = BUFFER_Get(*to_sign_len_out, TASKID_GetRunningTaskEnum());
   if (to_sign == NULL)
   {
      return NULL;
   }
```

```
   p = &grouping_header->gh_OtherInformationLength + grouping_header-
>gh_DateTimeLength; /* Start of OtherInformationLength */
   p += 1 + (((*p & 0x80) != 0) ? (*p & 0x7F) : 0); /* Start of OtherInformation */

   p_to_sign = to_sign;
   (void)memcpy(p_to_sign, grouping_header->gh_TransactionId, sizeof grouping_header-
>gh_TransactionId);
   p_to_sign += sizeof grouping_header->gh_TransactionId;
   (void)memcpy(p_to_sign, grouping_header->gh_BusinessOriginatorID, sizeof
grouping_header->gh_BusinessOriginatorID);
   p_to_sign += sizeof grouping_header->gh_BusinessOriginatorID;
   (void)memcpy(p_to_sign, grouping_header->gh_BusinessTargetID, sizeof
grouping_header->gh_BusinessTargetID);
   p_to_sign += sizeof grouping_header->gh_BusinessTargetID;
   (void)memcpy(p_to_sign, p, MESSAGE_ID_SIZE);
   p_to_sign += MESSAGE_ID_SIZE;
   (void)memcpy(p_to_sign, payload, payload_len);
   p_to_sign += payload_len;
   if (p_to_sign != to_sign + *to_sign_len_out)
   {
      LOG_FATAL("Logic error");
   }

   return to_sign;
}
```