This is an unfinished draft. If you have comments or corrections, please mark this document up and send it to jorg@cs.virginia.edu. If you send your in plain text, please include the date (see upper left corner), the page number and the paragraph number. If you find discrepancies between this document and the most recent version of the HyperCast software, please give a detailed description of the problem.

Thank you,

Jörg Liebeherr

CHAPTER 5 Security Architecture

ABSTRACT

This document summarizes the design of the security architecture in Hypercast3.0, including key management, authentication, and integrity and confidentiality of messages.

5.1 OVERVIEW

The security architecture in HyperCast attempts to satisfy integrity and confidentiality requirements of information processing in overlay networks. The architecture realizes practical security solutions for potentially very large and very dynamic overlay networks that do not require or assume permanent availability of a network infrastructure. The security goals are an assurance of backward secrecy (a new member of the network cannot access data transmitted before the member joined) and forward secrecy (a member cannot access data that is transmitted after it left the network) for application data. The building blocks of the architecture are as follows:

- **Authentication:** Authentication is managed through certificates signed by a trusted third party or designated certificate authority. An exchange of certificates is required when an overlay socket receives a protocol message from another overlay socket for the first time. HyperCast assumes that certificates are formatted following the X.509 specification. In HyperCast, all authenticated overlay sockets are trusted.

- **Key Management:** This refers to creation and exchange of secret keys at overlay sockets. HyperCast supports two key management. In one scheme, overlay sockets share a single symmetric group key for encrypting and signing messages. This is commonly done in secure group communications for overlay networks or network-layer multicast. To ensure forward and backward secrecy, group keys must be updated and distributed each time the group membership changes. This is referred to as re-keying. HyperCast does not provide a protocol for re-keying and leaves this task to the application. HyperCast has an alternative to group keys, called neighborhood key method, where each overlay socket has its own secret key, called the neighborhood key, which is shared only with authenticated neighbors in the overlay network. The neighborhood key method avoids network wide re-keying operations.

- **Message Keys:** In a scheme such as the neighborhood key method, where secrets are exchanged only between neighbors of the overlay network, encrypted message payloads cannot be deciphered by non-neighbors. This creates a problem when a message is forwarded. Clearly, decrypting and re-encrypting a message at each hop is very time-consuming and not practical in large networks. To reduce the overhead incurred at each overlay socket HyperCast employs separate keys for each message. Here, when an overlay socket wants to transmit a message, it generates a new symmetric key for this message, called a *message key*, and encrypts or signs the payload of the message with the message key. Then, the message key is encrypted with the neighborhood key and appended to the message. When overlay sockets share their neighborhood keys with their neighbors in the overlay network, only the message key must be decrypted and re-encrypted at each hop, without modifying the encrypted message payload.

- **Integrity:** Protection against unauthorized manipulation of protocol and overlay messages is achieved by adding signed hashes to a message. There are three types of signed hashes in HyperCast: one for a protocol message, one for the header of an overlay message, and one for the payload of an overlay message.

- **Confidentiality:** When confidentiality is desired, the entire payload of the message is encrypted with a group key or a message key. Headers of overlay messages and protocol messages are never encrypted. When message keys are used, they are

encrypted with a neighborhood key. When an encrypted overlay message is received by an overlay socket, it only decrypts the message if it is a destination of the message. Messages are decrypted just before they are delivered to the application program, and after they have been forwarded to the next hop other specified receivers. When the confidentiality is selected, all signed hashes are computed as well to ensure integrity.

5.2 SECURITY LEVELS

HyperCast has three security levels: *plaintext*, *protocol integrity*, *integrity*, and *confidentiality*. Plaintext means that no security features are activated. Here protocol messages and overlay messages are transmitted in plaintext and the sender of a message is not authenticated. All other levels require an authentication before overlay sockets exchange messages. With protocol integrity, all protocol messages are digitally signed with a message authentication code (MAC), and application messages are transmitted as plaintext messages without computing a MAC. With integrity, both protocol messages and overlay messages are digitally signed. Signed hashes for headers of overlay messages are verified and computed at each hop. Signed hashes for the payload of an overlay message are computed only by the source of the message, and verified only at the destination of the message. At the confidentiality level, the payload of each overlay messages is encrypted. The encryption is done at the source of a message and the decryption occurs at the destination(s) of the message. The confidentiality level also computes the same hashes as is done in the integrity level.

Table 1 summarizes the security operations for overlay messages and protocol messages that are performed at the available security levels.

Security Level	**Authentication**	**Operations applied to protocol messages**	**Operations applied to overlay messages**
Plaintext	No	None	None
Protocol Integrity	Yes	Signed hash	None
Integrity	Yes	Signed hash	Signed hash for header Signed hash for payload
Confidentiality	Yes	Signed hash	Encrypted payload Signed hash for header Signed hash for payload

Table 1. Security levels.

Note: There is an orthogonal mechanism available to ensure security, where overlay messages and protocol messages are transmitted over Secure Socket Layers (SSL) connections. An SSL connection is a secure communication channel that provides message confidentiality by encrypting all information exchanged using a session key, that is negotiated with the public key of the requestor of the SSL tunnel. The configuration of SSL security is different from the configuration of the previously discussed security methods, and is done entirely by configuring adapters that transmit messages over SSL tunnels. SLL tunnels for protocol messages and overlay messages are configured

independently. SSL security is established for overlay messages by selecting the socket adapter in the configuration file to be of type *SocketAdptSLL*, and for protocol messages by selecting the node adapter to be of type *NodeAdptSSL*. If an overlay socket transmits data over one of these adapters, it first establishes an SSL tunnel and then transmits the data over that tunnel.

5.3 TRUST ESTABLISHMENT

The establishment of trust is a key characteristic of a security architecture. Trust is the enabling of confidence that something will or will not occur in a predictable or expected manner, and is supported by mechanisms for identification, authentication, encryption, authorization, and availability [Andert02]. A key characteristic of peer-to-peer systems is that peer applications help with forwarding or storing information on behalf of other peer applications. This makes the peer network vulnerable to malicious or non-cooperative peers in several ways. A non-cooperative peer that receives an application message that is destined to some other peer, may, instead of forwarding the message to a neighbor drop the message. A malicious user may alter the content of messages, may disrupt the overlay network topology by sending false protocol messages, or may stage a denial of service attack. For these reasons, trust establishment in peer networks is probably more important than in an infrastructure network.

Trust establishment in an infrastructure network such as the Internet can be obtained through a PKI where a certificate authority (CA) initiates all trust relationships. Digital certificates issued by a CA that authenticate identities can be passed to a key management and encryption schemes, e.g., as described in the previous section. The difficulty of building trust increases dramatically without access to trusted third parties or intermediaries.

A variety of approaches have been tried for the establishment of trust in peer networks., including advance dissemination of private keys for all overlay socket pairs, threshold cryptography approaches, and many more, each offering a particular trade-off with respect to overhead, scalability, availability, and the ability to perform trust revocation. Some peer networks adopt a concept as introduced by PGP [Gar94], where a peer in the network knows the public keys of some other peers (with which it has a trust relationship) and relies on them to certify the public keys of other peers. By representing all trust relationship as a graph, one obtains what is called a web-of-trust [Datta03][Chen00]. A peer accepts a signed public key of a peer if it can find a path in the web that leads to this peer. The drawback of the web-of-trust is that trustworthiness is determined by the weakest trust relationship in the web. As a result, the trust established between peers becomes weaker when the size of the web grows. Another set of trust schemes in peer networks determines the trustworthiness of a peer by evaluation of its past behavior, e.g., feedback from other peers [Aberer01][Cornelli02][Kamwar03] Here, the different measures of trustworthiness are heuristically mapped into a cost metric. In a completely different approach, some peer networks attempt to mitigate the damage inflicted by malicious or non-cooperative peers that have joined an overlay network [Castro02][Wallach02]. Measures against non-cooperative, malicious, or faulty users are discussed in [Wallach02], and incentive systems that reward cooperative behavior are discussed in [Buragohain03][Chun04][Feldman04][Kamvar03]. While research in this area has provided many insights and continues to be important, the large variety of non-cooperative or malicious behaviors may not yield a general solution that detects and isolates undesirable behaviors after a peer has gained access to the peer network. A promising approach for trust establishment in a peer network is a distributed authentication scheme that is stronger than the web-of-trust. Distributed authentication

has been extensively studied in the context of distributed systems [Reiter96][Zhou??] and ad-hoc networks [Asokan00][Hubaux03] [Luo05][Wang03] [Yi03][Yi02][Zhou99]. A distributed CA can be constructed using threshold cryptography [Luo05][Yi03][Yi02][Zhou99]. In (K, N) threshold cryptography [Shamir79], a secret number D is added to a randomly selected polynomial of degree K-1. The resulting polynomial is evaluated at N positions. Then, D can be computed by obtaining K out of N values. To build a distributed CA, the private key of the CA is distributed with (K, N) threshold cryptography, yielding N *partial CA's*. A new user is authenticated when K out of N partial CA's have signed the certificate of the new user. Possibly, authentication scheme based on threshold cryptography will evolve into practical protocols that can be adapted to unpredictable information needs and access to other peers.

HyperCast views non-cooperative and malicious peers as a matter of access control to the overlay network. HyperCast provides authentication through the use of digital certificates that relies on the exchange of certificates. Authentication is required for all overlay sockets whenever the security requirements are set to a level stronger than plaintext. Once a HyperCast overlay socket has passed the authentication by its peers, the application is trusted to be cooperative and not malicious.

5.4 AUTHENTICATION

Before overlay sockets can establish a secure association, they must authenticate each other. HyperCast employs an authentication method based on public key certificates. Each overlay socket must have a certificate that has been previously signed by a trusted third party, and certificates of trusted third parties. Without online access to certificate authorities, trust revocation is not resolved by this method, unless it is enhanced by a distributed authentication protocol. An exchange and verification of certificates between neighbors in the overlay occurs only when needed in an on-demand fashion. Each overlay socket accepts protocol and overlay messages only from authenticated overlay sockets. When an overlay socket receives an overlay protocol message from another overlay socket for the first time it requests a signed certificate from this overlay socket and includes its own certificate in the request. With the neighborhood scheme method, once certificates are exchanged, the overlay sockets send each other their neighborhood which are used to encrypt or sign messages.

The exchange of certificates is illustrated in Figure 1 for two overlay sockets A and B. When B receives a protocol message from A and the certificate of A is unknown (Step 1), overlay socket B discards the message, and sends a certificate request (*CertRequest*) message to A (Step 2), which includes B's certificate. When A receives the request, it verifies the signature of B's certificate and, if valid, stores the certificate. Verification of the signature requires that the private key that signed the certificate in question be the private counterpart of the public key known to belong to a trusted third party. Next, in Step 3, A sends a certification reply (*CertReply*) message containing its own certificate. In Figure 1, B's authentication at A is completed in Step 2, and A's authentication at B is completed in Step 3. Once authenticated, the overlay sockets can process each others protocol and overlay messages.

Until the authentication is completed for a remote overly socket, all protocol messages (and also overlay message) received from that socket are dropped. If the authentication of the certificate fails, the certificate received from the remote socket is dropped and no further action is taken.

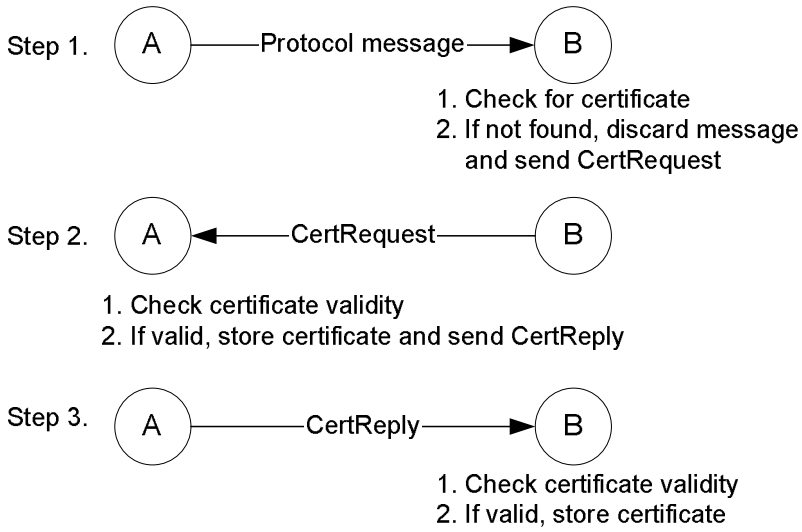


Figure 1. Authentication.

5.5 KEY MANAGEMENT

The purpose of key management in HyperCast is usage of secret keys and the distribution of keys to other overlay sockets. Encryption of data and the signing of hashes are done with symmetric keys. HyperCast offers two methods for managing keys: group keys and neighborhood keys, where the neighborhood key method has three variants.

In the group key methods, there is a single symmetric key, called the group key, that is shared by all overlay sockets, and which is used for signing and encrypting methods. The session key can be dynamically modified while the socket is running. With shared group key security, all tasks that require a key, such as MAC computation and message encryption, are performed with the specified group key. To ensure forward and backward secrecy, group keys must be updated each time the group membership changes. The update of group keys, also known as re-keying, is not handled by the overlay socket and must be implemented by the application programs.

The remainder of this section describes the neighborhood key method, which has been developed in the context of the HyperCast project. The neighborhood key method avoids network wide re-keying operations, without requiring that message payloads be re-encrypted at each hop. Each overlay socket maintains a single symmetric key with all authenticated overlay sockets. We call this key a *neighborhood key*. An overlay socket authenticates each overlay socket from which it receives a protocol message. This includes the current neighbors in the overlay topology, but potentially also many other overlay sockets. For example, overlay sockets that play a role in the rendezvous process (rendezvous server, buddy list member), overlay sockets that are probed to become potential neighbors, or the receivers of a broadcast overlay message, etc. Whenever the set of (current or potential) authenticated overlay sockets changes, i.e., a new neighbor appears or an existing neighbor disappears, the overlay socket computes a new neighborhood key and sends this new key to all authenticated overlay sockets. In compassion to shared group keys where all overlay sockets in the overlay network must update (re-key) the shared key whenever the membership in the overlay network changes, updating keys in the neighborhood method is a local operations, i.e., each overlay socket updates keys only with current neighbors in the overlay network.

Neighborhood keys are securely exchanged in a *KeyUpdate* message, by encrypting the key with the public key of the receiver using the RSA algorithm. The public key is obtained from the certificate that was exchanged during the authentication. Whenever an overlay socket receives a *KeyUpdate* for an overlay socket for which a key value already exists, it merely replaces the key entry to the new value. The transmission of a *KeyUpdate* message to authenticated neighbors is triggered when (1) a new authenticated neighbor has appeared; (2) an authenticated neighbor leaves the neighborhood or has not sent a message for a long time; (3) an authenticated neighbor requests the neighborhood key; (4) the overlay socket has reached the maximum sequence number; or (5) the current neighborhood key has exceeded a specified maximum lifetime. We discuss these situations now in more detail.

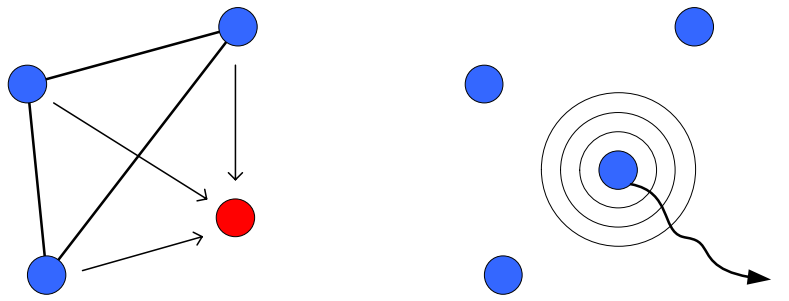
1. New authenticated overlay socket has appeared: To ensure backward secrecy, an overlay socket must generate and disseminate a new key. In Figure 1, the *Key Update* messages with the new keys are sent immediately after the authentication is completed. A sends a *KeyUpdate* immediately following the *CertReply* message, and A sends a *KeyUpdate* after it has verified the certificate contained in the *CertReply*.
2. Neighbor leaves the neighborhood or authenticated socket is quiescent: When an authenticated neighbor leaves the neighborhood or has not sent a message for a long time, an overlay socket must generate a new key and transmit it in a *KeyUpdate* message. Overlay sockets that are not neighbors in the overlay topology also trigger the creation and dissemination of a new key if no communication has been received from these sockets for a longer time.
3. Authenticated overlay socket requests the neighborhood key: A *KeyRequest* message is transmitted when an integrity check fails on a message. Here, the overlay socket assumes that it does not have an updated neighborhood key. The receiver of a *KeyRequest* checks if the requestor is authenticated and then sends a *KeyUpdate* message with the current value of the neighborhood key. No new key is generated in this situation. To prevent a malicious adversary from staging a DoS attack by sending forged messages that never pass an integrity test, the frequency of transmitted *KeyRequest* messages is limited.
4. Wrapping of sequence numbers: Every overlay socket maintains a sequence number for outgoing protocol and overlay messages, which is recorded at the receiver of a message. A receiver only accepts messages with increasing sequence numbers. The sequence number is reset when a new key is generated. When the sequence number wraps around, a new key must be generated. In this event, the overlay socket resets its sequence numbers to 0 (the first message sent will have value 1) if it updates its neighborhood key and sends out a new *KeyUpdate* message to all of its neighbors.
5. Expiration of neighborhood keys: An overlay socket must update its neighborhood key if the key as exceeded its specified maximum lifetime.

Other situations when messages are sent:

- The timestamp permits the receiver of the message to determine if it has a current key. When the timestamp does not match the timestamp that is locally stored, the overlay socket knows that it does not have a current key. Then, the overlay socket sends a *KeyRequest* message, to request a more recent key. In a secure protocol message, the DST header field is set to zeros.

We have seen that, whenever the set of authenticated neighbors changes, an overlay socket updates its neighborhood key. If messages are encrypted or signed with neighborhood keys, only authenticated overlay sockets in the overlay network can decrypt or verify transmitted messages. Since the neighborhood key is updated each time the neighborhood in the overlay topology changes, a newly joined overlay socket is unable to read messages sent before the overlay socket joined, and a departing overlay socket cannot read messages that are transmitted after it leaves. In this fashion, the neighborhood key method realizes backward and forward secrecy.

The workload due to updating neighborhood keys can be high. For example, when a new overlay socket joins the overlay network it may contact many other overlay sockets before it converges to its final position in the overlay network. Since each change to the neighborhood requires that the sockets generates and distributes a new neighborhood key, the security features may delay the convergence of the overlay protocol. The problem is exacerbated during failures in the substrate network when the overlay topology must be reconstructed and many sockets join and leave the overlay network at the same time. When the time interval between changes to the neighborhood is smaller than the time required updating a neighborhood key, the overlay protocol may no longer converge to a stable topology. HyperCast provides several variations of the Neighborhood scheme method that attempt to reduce the overhead due to key updates in the neighborhood key method. These variations are discussed below. It is also possible to relax the requirement of generating new keys each time the neighborhood of a sockets changes the overlay topology is unstable, at the cost of weakening forward and backward secrecy.



(a) Rendezvous node is contacted by many overlay sockets.

(b) Mobile node that broadcasts protocol messages.

Figure 3. Scenarios with frequent updates of neighborhood keys.

In Figure 3, we illustrate two scenarios that will incur frequent key exchange operations in the neighborhood key method. In Figure 3(a) we depict an overlay network with three nodes the neighborhood in the overlay topology is indicated by thick lines, and a rendezvous server. We assume that all nodes in the network periodically contact the rendezvous server, indicated by arrows, to verify that the network is not partitioned (Having all nodes in an overlay network contact the same rendezvous server is not practical for large networks. The depicted scenario is shown to illustrate the problem, and not derived from an actual protocol solution.) In the neighborhood key method, the rendezvous server exchanges messages with all nodes in the overlay network. Thus, whenever the node membership changes, the rendezvous server needs to update and distribute its neighborhood key to all current members in the overlay network. This not only leads to a prohibitively high load at the rendezvous server. Also, each node in the

overlay network must perform protocol operations when the membership of the overlay network changes.

In Figure 3(b) we show a problem that may appear in overlay networks in a mobile ad hoc environment. The figure depicts a mobile node (the path is indicated by a thick line) that broadcasts protocol messages (indicated by circles around the transmitting node) that announce the presence of the mobile node to other nodes in its vicinity. With the neighborhood key method, whenever a node receives broadcast messages from a mobile node for the first time it must update its neighborhood key. When the number of mobile nodes is large and the mobility of nodes is high, the load due to updating keys in terms of traffic and computations may be significant.

Common to both examples above is that nodes may spent a lot of effort to update keys with its neighbors due to nodes that are not neighbors in the overlay network topology, e.g., a rendezvous server or nodes receiving broadcast messages. Since the primary role of the neighborhood key is the encryption or protection of application data, and only neighbors in the overlay network exchange application data, the effort spent with updating neighborhood keys may be reduced by distinguishing that are neighbors in the overlay network from those that are not neighbors in the overlay topology and have separate key management for neighbors and non-neighbors. This leads to the following variations of the neighborhood key method.

1. **Separate Neighborhood Keys for each Non-neighbor:** Here, each overlay socket has a neighborhood key for all neighbors in the overlay topology, and a separate key for each non-neighbor with which it communicates. The advantage of this method becomes evident in the scenario in Figure 3(a). An overlay socket does not need to communicate with the rendezvous server when its neighborhood changes in the overlay topology. The rendezvous server would need a separate key for each overlay socket in the overlay network. However, when an overlay socket joins or leaves, the rendezvous server need not update keys with any other node in the network. The drawback of this variation is that maintaining separate security associations with single overlay sockets precludes the use of broadcast operations in the substrate network.

2. **Shared Key for all Non-neighbor (Double-check: It is also possible to send *all* protocol messages with a shared key):** In this method, each overlay socket has a neighborhood key for all neighbors in the overlay topology, and a single shared key for protocol messages sent to non-neighbors in the overlay topology. The shared key must be known to all overlay sockets in the overlay network. This method addresses the scenario of Figure 3(b) when protocol messages are transmitted in the substrate network with a broadcast transmission. When the broadcast message is signed with the shared key, the authenticity of the broadcast message can be verified without requiring the neighborhood key from the mobile node. The drawback of this method is that it has the same re-keying requirements as the shared group key scheme. With this variation of the neighborhood scheme, however, the shared key scheme is only extended to (some) protocol messages. Application payload is still protected with a neighborhood key, that is exchanged whenever the set of neighbors in the overlay network changes. In other words, forward and backward secrecy is maintained for application data.

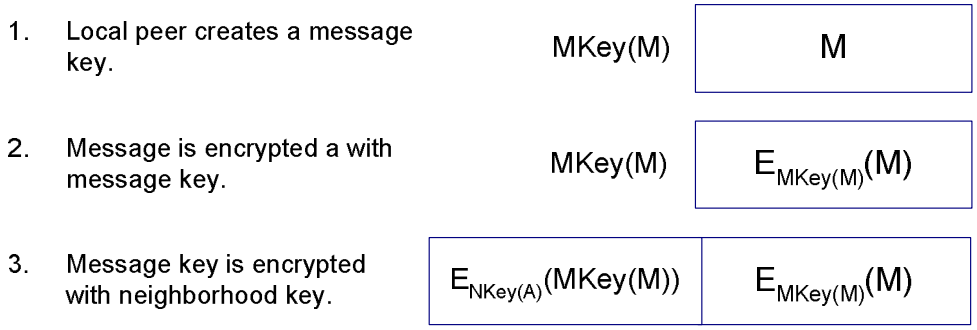
The above schemes do not relax the requirement of a mutual authentication. Overlay sockets always drop messages received from unauthenticated sockets and initiate a certificate exchange. Also, the variations do not affect the transmission of overlay

messages, since only neighbors exchange overlay messages and the variations do not change the security scheme for overlay messages.

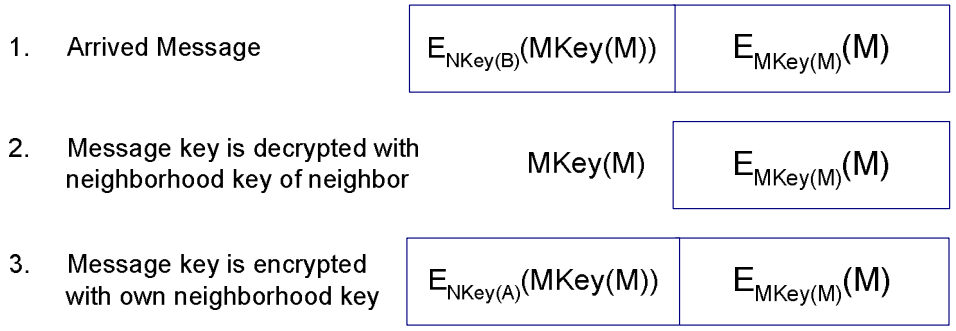
5.6 DATA CONFIDENTIALITY

In HyperCast, confidentiality is provided only for application data, i.e., the payload field in overlay message. Encryption of the payload is performed at the source of a message and decryption is performed at the destination(s) of the message. Data is encrypted using a symmetric key algorithm (*AES*, *Blowfish*, *DES*, *DESede*) with a specified key length (between 0 and 1024 bits). With shared group keys, encryption and decryption is straightforward using the shared group key. Payload encryption with the neighborhood key method is more complex and discussed in the remainder of this section.

With the neighborhood key method, when an encrypted message is forwarded in the overlay network, the message must be decrypted and re-encrypted at each hop. Clearly, this is very time-consuming and not practical in large networks. To reduce the overhead incurred at each overlay socket, we employ separate keys for each message. Here, when an overlay socket wants to transmit a message, it generates a new symmetric key for this message, called a message key, and encrypts the payload of the message with the message key. Then, the message key is encrypted with the neighborhood key and appended to the message. When an overlay socket receives an encrypted message it first decrypts the message key. (Recall that each overlay socket has the neighborhood keys of all authenticated neighbors.) If the message must be forwarded to another overlay socket, it re-encrypts the message key with its own neighborhood key.



(a) Transmission.



(b) Forwarding.

Figure 4. Processing an encrypted application message (*M* is the message, *MKey(M)* is the message key for message *M*, *NKey(A)* and *NKey(B)* are the neighborhood keys of overlay sockets *A* and *B*, $E_{MKey(M)}(M)$ is the message encrypted with the message key, $E_{NKey(B)}(MKey(M))$ is the message key encrypted with the neighborhood key of *B*.

In Figure 2(a) we show the encryption of a message that is transmitted by a node A with neighborhood key *NKey(A)*. The node generates a message key *MKey(M)* for a message *M*, encrypts the message with the message key, encrypts the message key with its neighborhood key, appends the encrypted message key to the message, and, finally forwards the message to a neighbor. In Figure 2(b), we show how node A forwards an encrypted message received from a neighbor B. First, A decrypts the message with B's neighborhood key, re-encrypts the message key with its own neighborhood key, and then forwards the message. Note that the encrypted message payload is not modified in this process. Merely, the encrypted message key must be processed. Since a message key is short (128 bits with current best practices), the delay incurred by decrypting and re-encrypting the message key is limited.

5.7 DATA INTEGRITY

Integrity is provided by requiring the sender of a message to include a signed checksum in the message, called message authentication code (MACs)[1]. The receiver verifies the MAC of the message by computes a checksum over the message and comparing it to the checksum that was included in the message. If the checksums are identical the receiver assumes that there has not been an unauthorized manipulation of the message. If checksums are not identical, i.e., the integrity test fails, the message is assume do be compromised and the message is dropped.

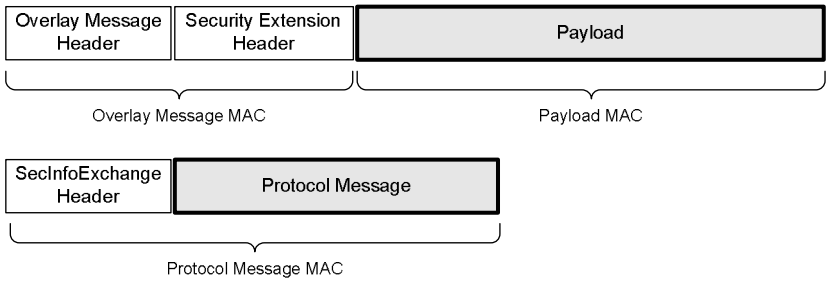


Figure 5. MAC computation. (When the security level is stronger than plaintext, each overlay message has a security extension header and each protocol message has an *SecInfoExchange* header.)

As illustrated in Figure 5, HyperCasts uses three types of MACs: one for protocol messages, one for headers of an overlay message, and one for the payload of an overlay message. The MACs play the following role:

1. **Protocol message MAC:** The role of the protocol message MAC is to protect against protocol messages that are sent or manipulated by unauthorized users. With the protocol message MAC only authenticated overlay sockets can transmit overlay message and participate in an overlay topology. The sender of a protocol message computes a MAC over the entire protocol message, and adds it to the message. The MAC is verified by the receiver of the message.[2] [3]

[1] Precisely, we use a keyed-hash message authentication code (HMAC), which involves a cryptographic hash function in combination with a secret key.

[2] Even with a plaintext security level, most protocol messages have an *overlay hash* field that is computed over specified attributes in the configuration files. The overlay has provides a weak integrity check since it can be used to distinguish protocol messages from different overlay networks.

2. **Overlay message MAC:** The overlay MAC protects, among others, against unauthorized changes of the route of a packet and the destination address of a message. When an overlay socket sends or forwards an overlay message, it computes a MAC for the header of the message, including all extension headers, but with exception of the payload. The header MAC is verified and recomputed at each *hop* of the message.
3. **Payload MAC:** The payload MAC permits to verify the origin of an overlay message and can detect unauthorized changes of application data while a message is transmitted in the overlay network. The payload MAC is computed at the source of an overlay message over the payload field in the message. The MAC is verified at the destination(s) of the message, and is neither inspected nor modified at intermediate hops that forward the message. This protects against unauthorized changes of application data while a message is transmitted in the overlay network.

The algorithms employed for the hash algorithms are specified in the configuration file (*HmacMD5, HmacSHA1*). The steps for computing and verifying signed hashes are similar to the encryption and decryption of a message, and vary dependent on the choice of the key management method. With group keys, all MACs are computed with the shared group key. We next describe the computation of MACs with the neighborhood key method.

With (all variations of) the neighborhood method, neighborhood keys and message keys are used to sign and verify a message. The source of a message first builds a message key and computes a MAC of the message payload with the message key. When the payload is encrypted, the same message key is used for encryption and for the payload MAC. The message key is encrypted with the neighborhood key, and then added to the message. When a message is forwarded at an intermediate hop, the message key is decrypted and re-encrypted (as shown in Figure 4), but the payload MAC is not modified. The overlay message MAC is computed over the entire message with exception of the payload field and the field that stores the overlay message MAC. The MAC is computed with the neighborhood key, and is verified and recomputed at each hop that forwards the message. Both MACs, together with the encrypted message key, are transmitted as an extension header of the overlay message. The computation of the MAC for protocol messages depends on the version of the neighborhood key method. In the basic scheme, the MAC is computed over the entire protocol message with the neighborhood key. In other versions, the MAC may be computed with a shared key.

In our implementation, with the assumption that data confidentiality implies a need for integrity, the MACs for the payload and header of overlay messages, and the MAC for protocol messages are always computed, when encryption of application payload is requested.

Using the neighborhood key method an integrity check may fail for two reasons: (1) the message or header has been altered, or (2) the neighborhood key has changed and the verifying node does not have a recent copy of the key. When an overlay socket is configured with the neighborhood key method, an overlay socket, upon a failed integrity check, it discards the message, but assumes that the failed test is due to an outdated version of the neighborhood key. Here, it sends a message to the neighbor from which the message was received requesting its neighborhood key. By limiting the rate at which these requests are sent, e.g., no more than one request for a key can be outstanding at

[3] Protocol messages are never forwarded.

any time, the key requests sent to neighbors even though a message has been altered without authorization can be ignored.

5.8 CONFIGURING AN OVERLAY SOCKET FOR SECURITY

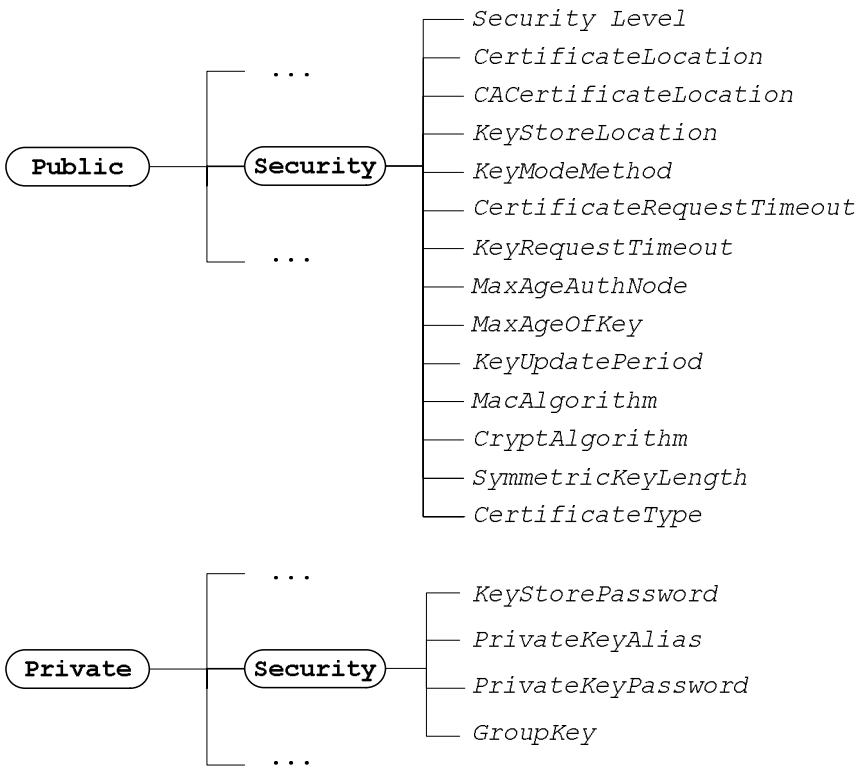


Figure 6. Attributes for security architecture.

The parameters of the security configuration of an overlay socket are determined from configuration attributes. Recall from Chapter *Advance API*.3.5 that there is a distinction between public and private attributes. Public attributes make up the majority of configuration parameters. These attributes are stored in the configuration file of a socket. Private attributes contain confidential information, such as a password to access a private key or a certificate. They must be configured by the application program that creates an overlay socket. All public security attributes have the prefix */Public/Security* and all private security attributes have the prefix */Private/Security*. The type and format of public and private attributes are specified in separate XML schema files. The schema file for public attributes can assign default values to attributes, but private attributes do not have default values. The public and private attributes for the security configuration of an overlay socket are shown in Figure 6.

The private key corresponding to the local certificate is obtained from the specified *keystore* file. This is a file that can be created by Java programs or the *keytool* command line utility (see Chapter Advance API 3.5).

There are separate schema files that specify the type and format of public and private attributes. The schema for public attributes can have default values for attributes,

Each application program is responsible for maintaining its own certificate, and the associated private and public keys. The certificate is managed through a *keystore*, an

encrypted database of private keys and X.509 certificates authenticating the public keys. Overlay sockets access private keys and certificates from the *keystore*. The configuration file declares the location of the certificates and the *keystore*. The following attributes are used for accessing and processing certificates:

The following are the public security attributes

SecurityLevel

An overlay socket can be configured with three different security levels: *plaintext*, *integrity*, and *confidentiality*. The default value is *plaintext*.

CertificateLocation

Contains the file that contains the X.509 certificate of the application program. The certificate is obtained from the information in keystore file. The default value is *testcert.cer*. (Since the certificate can be extracted from the keystore file, this attribute can be viewed as being redundant.)

CACertificateLocation

Specifies the file that contains the X.509 certificate of the Certificate Authority (CA) that granted the certificates. The default values of the file is *testcert.cer*. The *CACertificateLocation* can be used to obtain Certificate Revocation Lists (CRLs) so that sockets can stay up-to-date on the validation of certificates. Certificate revocation is not supported in the current implementation.

KeystoreLocation

The attribute stores the location of the keystore file (default is .keystore). The certificate must be in a binary or text DER (Distinguished Encoding Rules)-encoded format, such as PKCS#7 or Base-64.

KeyModeMethod

This attributes selects the method for constructing and managing the keys that hash or encrypt information in an overlay socket. The value *NeighborhoodKey*. The value *GroupKeys* is based on shared group keys. Valid values are:

- *GroupKeys* – All overlay sockets are assumed to have the same shared group key. Whenever the group key is accessed, e.g., for signing, verifying, encrypting, or decrypting messages, the overlay socket checks if the value of the attribute has changed.

- *NeighborhoodKey1 (formerly: UniformDynamicKey)* – This is the default value. Each overlay socket has one neighborhood key that it exchanges with all authenticated overlay sockets.

- *NeighborhoodKey2 (formerly: StaticNonNeighborKey)* – Each overlay socket has one neighborhood key for all overlay sockets that are neighbors in the overlay network topologies, and one neighborhood key for each overlay socket that has been authenticated, but that is not a neighbor in the overlay topology.

- NeighborhoodKey3 *(formerly: SharedProtocolKey)* – This scheme is a hybrid of the *GroupKeys* and the *NeighborhoodKey1* method. Each overlay socket has a shared group key that is known to all overlay sockets. In addition, each overlay socket has a neighborhood key that is exchanged with all authenticated overlay sockets and that is updated whenever the set of authenticated overlay sockets changes (Or is it: "whenever the neighbors in the overlay topology change"? Determine if *NeighborHoodKey3* makes a distinction between neighbors and non-neighbors.). All protocol messages are signed with the group key. All overlay messages are signed and encrypted with the neighborhood key. The advantage of this scheme is that an overlay socket can verify incoming signed protocol messages even when it does not have an updated neighborhood key.

CertificateRequestTimeout

Minimum time that must elapse between transmissions of two *CertRequest* messagess. The default value is set to 30 seconds.

KeyRequestTimeout

Maximum waiting time for the reply to a *KeyRequest* message before a *KeyRequest* message is retransmitted. The default value is set to 10 seconds.

MaxAgeOfAuthNode

The maximum age of an entry in the key vault about an authenticated overlay socket after the last protocol message received from this overlay socket.

MaxAgeOfKey

The maximum age of a neighborhood key before it is updated with a *KeyUpdate* Message.

~~*KeyUpdatePeriod (←Deleted, or?What is the difference to the MaxAgeofKey)*~~

~~The time interval between the transmission of KeyUpdate messages.~~

MacAlgorithm

Specifies the algorithm used to compute message authentication codes. The values of the attribute and the corresponding algorithms are given in Table 2.

CryptAlgorithm

Specifies the symmetric cryptographic algorithm for encrypting and decrypting the payload of overlay messages. The values of the attribute and the corresponding algorithms are given in Table 3. The selection of the algorithm restricts the value of the *SymmetricKeyLength*.

SymmetricKeyLength

The length of the key for the cryptographic algorithm. The value must be in the range from 0 to 1024, with 128 as the default value. As given in Table 3, each cryptographic algorithm only permits a certain range of values. (What is the role of Table 4? Is this Java specific?)

CertificateType

The certificate specification that is used for authentication of overlay sockets. The only permitted value is X.509.

Table 1. Security policy. (The default value is underlined.)

Security Level	Level of overlay messages	Level of protocol messages
Plaintext	Plaintext	Plaintext
Protocol Integrity	Plaintext	Integrity
Integrity	Integrity	Integrity
Confidentiality	Confidentiality	Integrity

Table 2. Values for the attribute *MacAlgorithm.* (The default value is underlined.)

HmacMD5	The HMAC-MD5 keyed-hashing algorithm as defined in RFC 2104: "HMAC: Keyed-Hashing for Message Authentication".
HmacSHA1	The HMAC-SHA1 keyed-hashing algorithm as defined in RFC 2104: "HMAC: Keyed-Hashing for Message Authentication".

Table 3. Values for the attributes *CryptAlgorithm*. (The default value is underlined.)

AES	Advanced Encryption Standard as specified by NIST in a draft FIPS. Based on the Rijndael algorithm by Joan Daemen and Vincent Rijmen. *SymmetricKeyLength* must be set to 128, 192, or 256 bits.
Blowfish	The block cipher designed by Bruce Schneier. *SymmetricKeyLength* must be a multiple of 8, and can only range from 32 to 448.
DES	The Digital Encryption Standard as described in FIPS PUB 46-2. *SymmetricKeyLength* must be equal to 56.
DESede	Triple DES Encryption (DES-EDE). *SymmetricKeyLength* must be equal to 112 or 168.

Table 4. Key size parameters and keys sizes for various algorithms. (JL: How come that "key size parameter" and "size of created keys" have different values. Where does the table come from?)

Key algorithm	**Key size parameter (in bits)**	**Size of the created key (in bits)**
AES	128	128
Blowish	128	128
DES	56	64
DESede	112 or 168	192

The following are the private attributes for the security configuration of an overlay socket. All attributes have the prefix */Private/Security*.

KeyStorePassword

Password to access the keystore file as specified by the *KeystoreLocation* attribute.

PrivateKeyAlias

The private key is accessed with an alias and protected with a password. The alias and password are set when a new public/private key pair is generated.

PrivateKeyPassword

The password for the private key that is accessed with the value of the *PrivateKeyAlias* attribute.

GroupKey

Specifies the symmetric shared group key of an overlay network. All overlay sockets must have the same value of the attribute. The attribute **Note:** Each time the group key is accessed, the value of the *GroupKey* attribute must be read, since it may have changed since the last access. (This is an example of a special class of attributes, which should not be stored internally. Most attributes are read only during configuration of the overlay socket. Maybe there should be a separate class of attributes, identified by an XML attribute, that specifies if an attribute should be handled in this fashion.)

Since private attributes do not have default values and are not stored in configuration files, they must be explicitly set by the application program. In the Java implementation of HyperCast, this is done by invoking:

```
config.setPrivateTextAttribute(XPath xpathOfPrivateAttribute, String value)
```

Here *config* is a configuration object of type *HypercastConfig* of the overlay socket, *xpathOfPrivateAttribute* is an XPath expression that is created from the private security attributes, and *value* is the value assigned to the attribute as a string.

For example, the group key can be generated from a string with the following line of code:

```
//Set the group key into private attribute document
ConfObj.setPrivateTextAttribute(XmlUtil.createXPath("/Private/GroupKey"), “MyKey”);

(Check: what if the string MyKey is very long? Is there truncation? Check out the following
groupKey = new SecretKeySpec(groupKeyString.getBytes(), confidentialityAlgorithm);
what does SecretKeySpec do if getBytes returns many bytes? )
```

5.9 SECURE PROTOCOL MESSAGES

In an overlay socket with elevated security levels, i.e., levels that are different from plaintext, all protocol messages created in the overlay socket are encapsulated as security information exchange (*SecInfoExchange*) messages. In addition to wrapping protocol messages of overlay protocols, *SecInfoExchange* messages are also employed for the exchange of certificates (*CertRequest*, *CertReply*) and keys (*KeyRequest, KeyUpdate*). Messages of type *KeyRequest* and *KeyUpdate* are used only in the context

of the neighborhood key method. In the following we describe the message format of *SecInfoExchange* messages and provide additional information on the processing of these messages.

5.9.1 MESSAGE FORMAT

In this section we discuss the format of *SecInfoExchange* messages. *SecInfoExchange* messages are defined as a distinct overlay protocol. In particular, the protocol number 0xf0 designates *SecInfoExchange* messages. The messages follow the convention of all protocol messages that (1) the first byte identifies the protocol, (2) the next two bytes define the total length of the message, and (3) the next byte specifies the protocol specific message type. All *SecInfoExchange* message have a common header fields as shown in Figure 7. These fields are referred to as the *SecInfoExchange* header. The remainder of the message is different dependent on the message type.

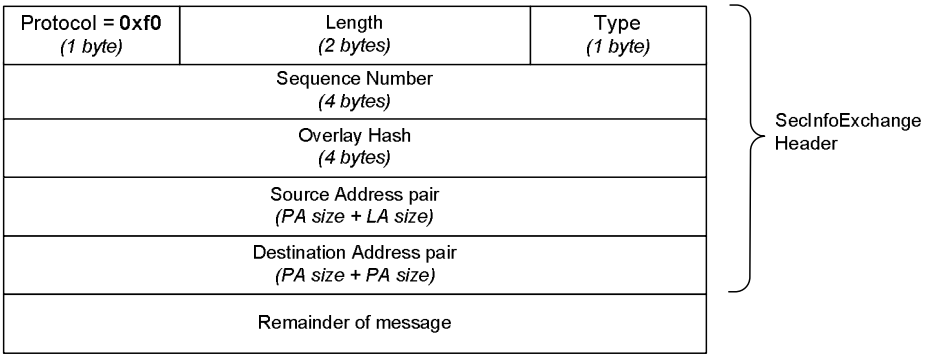


Figure 7. Format of *SecInfoExchange* message.

Protocol (1 byte):

The protocol number of all *SecInfoExchange* messages is set to 0xf0.

Length (2 bytes):

The length of message following the length field, i.e., the actual length of the message larger by 3 bytes.

Type (1 byte):

There are five different message types:

0x01 Certification Request (*CertRequest*)
0x02 Certification Reply (*CertReply*)
0x03 Key Request (*KeyRequest*)
0x04 Key Update (*KeyUpdate*)
0x05 Encapsulated Protocol Message (*ProtoMsg*)

Sequence Number (4 bytes):

The sequence numbers is set to 1 in the first message and incremented each time before a message is transmitted. Messages from the same overlay socket that do not have increasing sequence numbers are discarded. With the neighborhood key method, sequence numbers are prevented from wrapping. If a sequence number has reached the maximum value, a new neighborhood key must be created and sent to

neighbors in a *KeyUpdate* message. Sending *KeyUpdate* triggers a reset of the sequence number counters.

Overlay Hash (4 bytes):

A hash value that is computed from the *HashAttributes*, which is a list of attributes written as XPath expressions. The *HashAttributes* may contain the overlay identifier *(/Public/OverlayID)*, the name of the overlay protocol *(/Public/Node)*, or other attributes. To compute the overlay hash, the values of the listed attributes are obtained from the configuration file,[4] concatenated using the UTF-8 character encoding scheme, converted into a byte array, and a hash function is applied. The hash function, which can operate on variable-length byte arrays, is defined as follows:

Input: Byte array A[ ], containing UTF-8 encoded hash attributes
Output: a 4-byte unsigned integer result

```
procedure OverlayIDHash (byte A[] )
begin
  Result := 0;
  for ( int i := 0 ; i < length of A[ ] ; i++ ) {
     byte upperByte := (byte) ( (result >> 24) & 0xFF );
     int  leftShiftValue := ((upperByte ^ A[i]) & 0x07) + 1;
     result := ((result << leftShiftValue) ^ ((upperByte ^ A[i]) & 0xFF));
  }
  return result;
end
```

Source Address Pair (*LA Size* + PA Size bytes):

The logical address and the physical address of the sender of this message. LA Size and PA Size are, respectively, the size of the logical address and the physical address. The size of the addresses are known from the configuration file.

Destination Address Pair (*LA Size* + PA Size bytes):

The logical address and the physical address of the destination of this message. For messages of type *ProtoMsg*, the fields of the destination address are set to zero.

The format of the additional fields for the *SecInfoExchange* message types shown in Figure 8.

[4] If the XPath expression identifies a simple XML element, the lookup returns the value of the attribute. Otherwise, the lookup returns the name of the first element contained in this element.

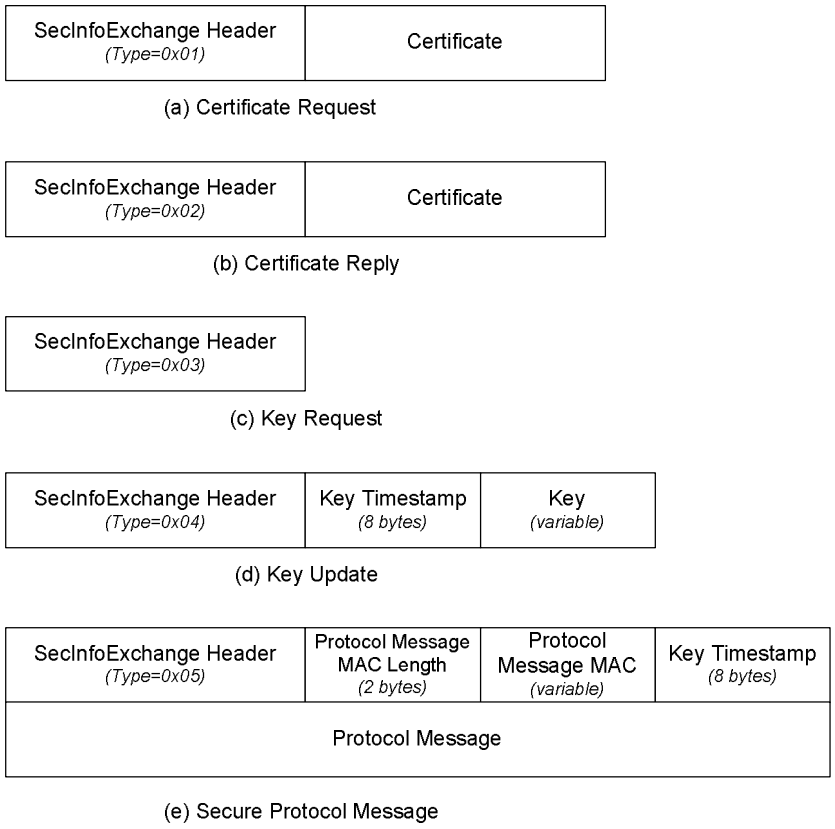


Figure 8. Format of *SecInfoExchange* messages.

The sender of a *CertRequest* and *CertReply* message include their own certificate following the *SecInfoExchange* header. The *KeyRequest* message only consists of the header. The *KeyUpdate* message is only transmitted when the neighborhood key method is employed. Here, the sender of the message sends its neighborhood key in encrypted form, and also a timestamp when key was created.

A secure protocol message contains the protocol message MAC, its length, a key timestamp. The timestamp permits the receiver of the message to determine if it has a current key. When the timestamp does not match the timestamp that is locally stored, the overlay socket knows that it does not have a current key. Then, the overlay socket sends a *KeyRequest* message, to request a more recent key.

Below we explain the fields appearing in the *SecInfoExchange*.

Certificate (variable):

The encoded form of a X.509 certificate in this message. X.509 certificates are encoded as ASN.1 DER. The length of the field is derived from the length field of the *SecInfoExchange* header.

KeyTimeStamp (8 bytes):

The time when the neighborhood key was created at the sender. The time is the difference, measured in milliseconds, between the timestamp and midnight, January 1, 1970 UTC.

Key (variable):

The encrypted neighborhood key contained in this message. The local key

is encrypted with the RSA algorithm using the private key that corresponds to the public in the local certificate. The length of the field is derived from the length field of the *SecInfoExchange* header.

Protocol Message (variable):

A valid protocol message with a protocol message header.

5.9.2 PROCESSING SECINFOEXCHANGE MESSAGES

Next we describe remaining details of the protocol that governs the transmission and processing of *SecInfoExchange* messages. Most of the operations of the protocol were discussed in Sections 5.3 and 5.5. Figure 7 shows the processing of an incoming *SecInfoExchange* message that contains a protocol message. First, there is lookup for the certificate of the sender. If the certificate is not available, an authentication process is initiated and the message is dropped. Initializing a certificate authentication includes the transmission of a *CertRequest* message to the remote overlay socket, and the addition of an entry for the remote overlay socket in the neighbor table. Authentication can only be triggered by the receipt of a *SecInfoExchange* message that encapsulates a protocol message. When an overlay socket receivers a *CertRequest* message, it verifies if there is a pending *CertRequest* that it sent to the sender of the message. In this case, there are two authentication processes ongoing between the same pair of overlay sockets. To prevent this from happening, the overlay socket discards the incoming *CertRequest* message when the logical address of the sender is larger than its own.

There is no retransmission provided for certificate requests. Each time a protocol message is received from an unauthorized neighbor, a *CertRequest* is sent. However, a minimum time given by *CertRequestTimeout* must elapse between two requests to the same destination. **(JL: 6/11, Is** *CertRequestTimeout* **provided?)**

In the next step, the overlay socket accesses the key used to perform the integrity check. If the key is not available, the message is dropped. If the *NeighborhoodKey* method is running, the current key is requested in a KeyRequest message. If, on the other hand, the key is available, the socket performs an integrity check. If the check fails, the message is dropped. In the context of the Neighborhood method, the node also requests a recent version of the key from the sender of the message. After a successful integrity check, the protocol message following the *SecInfoExchange* header is passed to the overlay node.

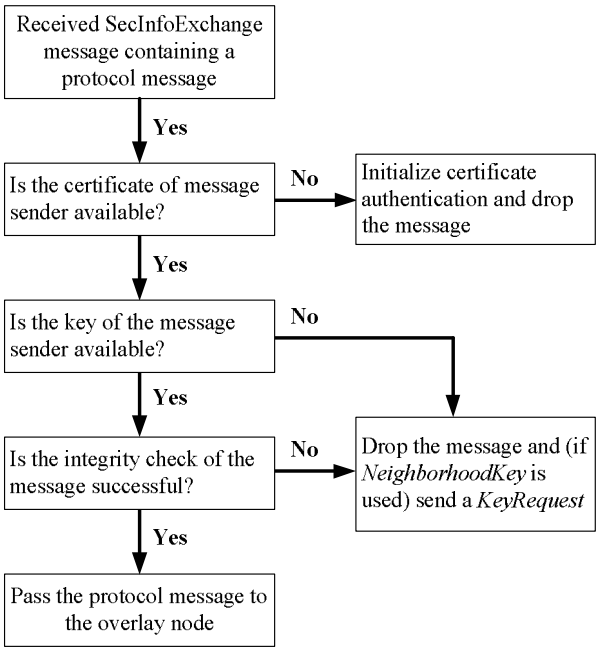


Figure 9. Processing an incoming *SecInfoExchange*.

An overlay socket removes entries, including keys and certificates, about authenticated overlay sockets if it has not received a message from this overlay socket for an extended period of time. An overlay socket maintains the time expired since the last message was received from an authenticated overlay socket, and resets the time to zero whenever a new message arrives. When the age exceeds a maximum value, given by the attribute *MaxAgeAuthNode*, the entry is removed. Periodically, an overlay socket checks all its authenticated neighbors if they need to be removed. The period is set equal to the maximum age.

With the neighborhood key method, a key can be used at most a time given by the attribute *MaxAgeOfKey*. The maximum lifetime of a key is enforced by a timer, calledthe *RekeyTimer* that is associated with the neighborhood key(s), called the *RekeyTimer*. When the timer expires, the socket generates a new key and sends it to other sockets in an *UpdateKey* message.

The transmission of a *CertRequest* message is triggered by the arrival of a protocol message from an unauthenticated overlay socket, i.e., an overlay socket for which no certificate is available. If a protocol message arrives and a certification process is ongoing, the overlay socket will send another CertRequest message. However, the minimum time, given by the attribute *CertRequestTimeout*, must elapse between two *CertRequest* transmissions to the same destination. (JL, 6/11: *CertRequestTimeout* is not provided, but I think it was there earlier. If it was deleted, what was the reason?).

Overlay sockets that run the neighborhood key method transmit *KeyRequest* messages when one of the following events occurs: (1) A protocol message is received from an overlay socket, for which an the certificate is available, but not the key; (2) A protocol or overlay message is received from an overlay socket that fails the integrity check; or (3) A timeout occurs because a *KeyRequest* has outstanding for more than *KeyRequestTimeout* milliseconds. The third event is triggered by a timer, called the *KeyRequestRetransmitTimer*. When a *KeyRequest* message is retransmitted in this fashion, the timer is restarted.

To reduce the number of *KeyRequest* message transmissions, an overlay socket does not remove the key for another socket when a message from this socket fails the integrity check. Instead the key entry is marked as invalid. An invalid key entry is still used to perform integrity checks for incoming messages. When an incoming message passes the integrity check for an invalid key, the *invalid* marking of the entry is removed. The timeouts in the third event above are ignored when the corresponding key entry is not marked as *invalid*. In this fashion, a DOS attack is avoided where a malicious user sends faked messages with a spoofed source address, and the receiver of the messages permanently removes the key for the overlay socket with the spoofed address. Retransmissions to an overlay socket are repeated only as long as an entry about that socket exists.

JL (11/6): Is there a maximum times of retransmissions for *CertRequest* and *KeyRequest* after which an overlay socket gives up?

In Figure 9 we show the steps for computing the protocol message MAC for an outgoing protocol message. The MAC is computed either with a group key or a neighborhood key. The MAC is calculated over the entire *SecInfoExchange* message, where the field of the MAC is removed. Once the MAC is computed, it is written into the field reserved for the protocol message MAC.

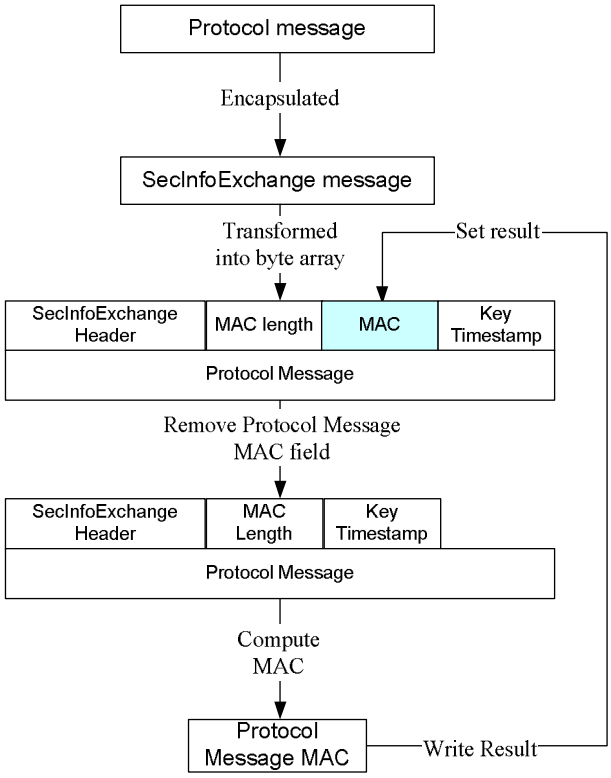


Figure 10. MAC calculation for a secure protocol message.

5.10 SECURE OVERLAY MESSAGES

All overlay messages in an overlay socket with the security level set to protocol integrity, integrity, or confidentiality contain a security extension, also called *security header*. Overlay messages with a security extension header are also called secure overlay

messages. The security extension is specified in the preceding header by a next header field with value 0x21. The security extension contains, among others, the MACs for the header and the payload and, if the neighborhood key method is enable, an encrypted message key. The security extension header is shown in Figure 11. **(What happens if there is no payload extension?)**

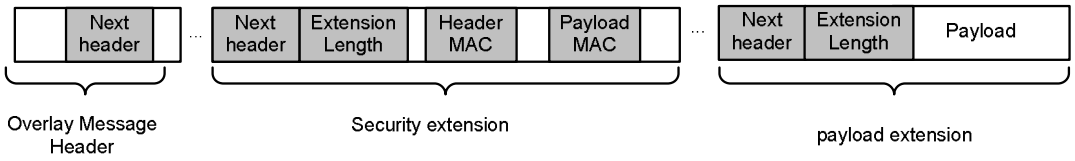


Figure 11. Security extension header in a secure overlay message.

The format of a security header is shown in the Figure 12. The contents of the fields is as follows:

Next Header *(1 byte)*	Length *(2 bytes)*	Sequence Number *(4 bytes)*
Encrypted Message Key *(0-32 bytes)*		
SPI *(4 bytes)*		
Length of LA *(1 byte)*	LA of Sender *(LA size bytes)*	
Header MAC Length *(1 byte)*	Header MAC *(variable, >= 16 bytes)*	
Payload MAC Length *(1 byte)*	Payload MAC *(variable, >= 16 bytes)*	

Figure 12. Format of the security extension.

Next Header (1 byte):

Specifies the type of extension following this header.

Length (4 bytes):

The length of the security header in bytes following the Length field, i.e., not include the Next Header and Length fields.

Sequence Number (4 bytes):

Specifies the sequence number of the message. The field is used in the same was as described for the *SecInfoExchange* header. The sender of a message increments the sequence number before each message transmission. With the neighborhood method, when the number of messages, both protocol and overlay messages, exceeds the maximum allowed number, then a new neighborhood key is generated and transmitted in a *KeyUpdate* message, and the sequence number is set to zero.

Encrypted Message Key (0-32 bytes):

Contains the encrypted message key. This field is not used when the group key method is executed, i.e., the length of the field is zero. The

length of the encryption key is determined from the configuration attributes. **(JL (6/11): Verify that message key is not included in message for group key method.)**

SPI (4 bytes):

The field SPI (Security Parameter Index) can contain a security association identifier, which is a random value identifying the security association for this message. The field is currently not used.

LA length (1 byte):

The length of the logical address of the overlay socket.

LA of Sender (variable):

The logical address of the neighbor that forwarded this message. The field is used to look up certificates and keys for the neighbor.

Header MAC Length (1 byte):

Length of the MAC for the header of the overlay message in bytes.

Header MAC (≥ 16 bytes):

Contains the MAC for the overlay headers. Precisely, the MAC is computed over the byte array of the entire message, with the Header MAC field and the payload field removed.

Payload MAC Length (1 byte):

Indicates the length of Payload MAC field in bytes.

Payload MAC (≥ 16 bytes):

Contains the MAC for the payload. If the payload is encrypted, this is the MAC of the encrypted payload. **(What happens to the field if there is no payload?)**

An overlay socket must compute the extension header in the following order: (1) Encrypt the payload field of the payload header; (2) Compute the payload MAC; and (3) Compute the Header MAC. When encryption is required, the payload MAC is computed over the encrypted payload field. The header MAC is computed over the entire message, with exception of the MAC header field and the payload field in the payload extension header.

With the group key method the group key is used for the payload encryption and the computation of both MACs. Here, the message key field is not used. With neighborhood keys, the message key is used for payload encryption and the payload MAC. The neighborhood key is used to encrypt the message key and to compute the header MAC.

For an incoming secure overlay message, an overlay socket verifies the sequence number and the header MAC. If either of these checks fails, the message is dropped without further processing. Otherwise, with the neighborhood key scheme, the message key is decrypted. When an overlay socket forwards a secure overlay message, it recomputes the header MAC and the sequence number, re-encrypts the message key in the security header (if present), and updates the LA Sender field with its own logical address. The header MAC is recomputed either with the group key or the neighborhood key of the local overlay socket. The encrypted message payload, the payload MAC, and the message key are not modified when the message is forwarded. The forwarding of

messages with security headers can be viewed as an operation that replaces a security header.

When a secure message arrives at an overlay socket that is a destination of the message, a message is processed like any incoming message. Then the payload MAC is verified either with the group key or the message key. If not successful, the message is dropped. Otherwise, the payload is decrypted with the group key or the message, and delivered to the application. Incoming multicast and flood messages are delivered to the application, but may also be forwarded to other overlay sockets. Here, it is advisable that the message is forwarded before the payload is decrypted, otherwise the processing time for decrypting the payload at intermediate hops increases the latency of a packet.

When messages are transmitted with an enhanced delivery semantics (see Chapter *MessageStore*) they may be stored in the message store of the overlay socket, and may be forwarded at a later time, e.g., to retransmit a message when no acknowledgment has been received. When a message is stored in the message key, it is important that the decrypted message key is stored together with the message.

5.11 SOFTWARE DESIGN

The majority of the security architecture is realized by two components of the overlay socket: a key vault and a security processor. These components are instantiated only if the security level is set to protocol integrity, integrity, or confidentiality. Hooks to access the key vault and the security processor are added to various functions in the overlay socket. Most components in the overlay socket do not know whether the security components are activated. Specifically, there is no security-specific API defined for the overlay socket. All security features of an overlay socket are activated by attributes in the configuration file. Refer to Chapter “Overlay Socket API (Advanced)” for the information on security configuration.

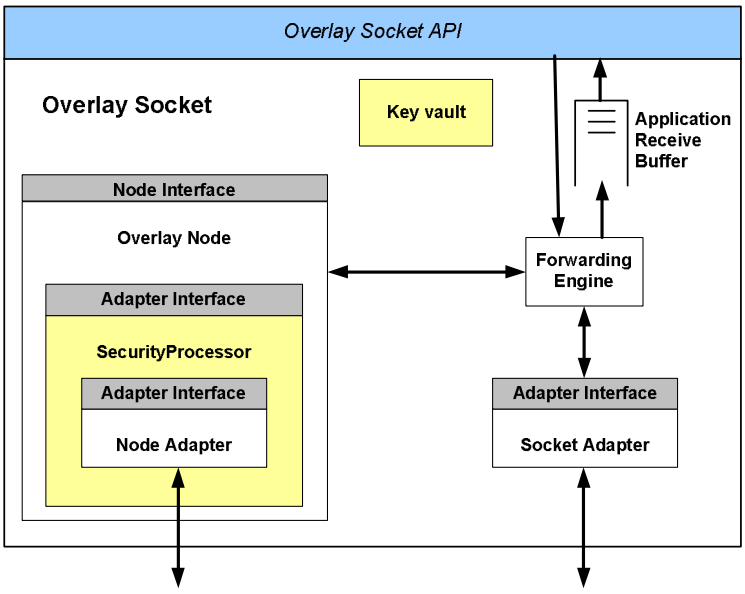


Figure 13. Overlay socket with key vault and security processor.

The *key vault* manages certificates and keys of the local overlay socket and remote overlay sockets that communicate with the local socket, and also stores the information about the currently used cryptographic algorithms. The key vault must be accessed by all overlay socket components that need to access keys or certificates. For example, the socket adapter that performs an integrity check the key vault is accessed to check if the sender has been authenticated, to access the algorithm and the key needed to perform an integrity check.

The security processor is a wrapper for the overlay socket adapter that provides a layer between the overlay node and the node adapter. The security processor is responsible for adding and removing the security encapsulation headers. All protocol messages created by the overlay node are passed to the security processor where they are converted into *SecInfoExchange* messages and sent out by the overlay socket adapter. *SecInfoExchange* messages encapsulate regular protocol message. In an overlay socket with a security processor, all incoming protocol messages are received as *SecInfoExchange* messages, reconstructed as regular (plaintext) protocol messages by the security processor, and then passed to the overlay node for processing.

5.11.1 KEY VAULT

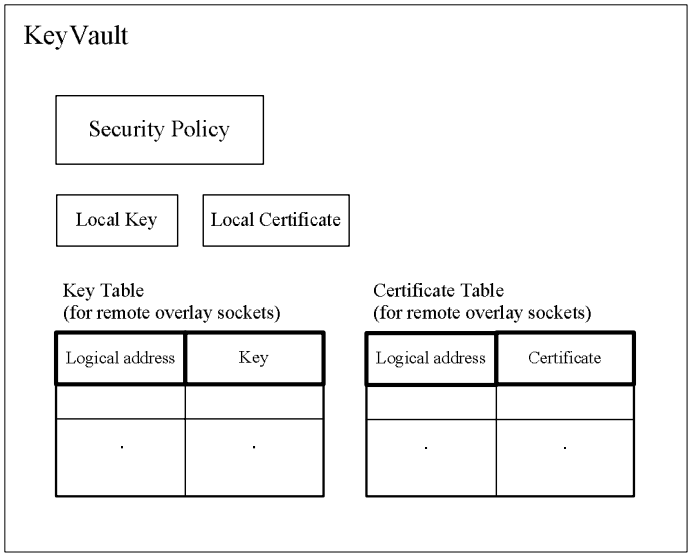


Figure 14. The key vault.

The key vault is a component created by and contained in the overlay socket to manage various security properties. It is the central place for managing certificates and keys. An overlay socket needs to have a key vault only when the security level is set to protocol integrity, integrity, or confidentiality. The key vault is configured with security attributes defined in the configuration file of the overlay socket and with attributes that must be provided by the application program.

The key vault maintains all security information of the overlay socket, including the security policy, the algorithms for encryption and hashing, all local keys and their sizes, as well as the local certificate, and the private key associate with the certificate. Information about remote overlay sockets is stored in two lookup tables, the key table and the certificate table. Both tables are indexed by their logical addresses. The tables store, respectively, keys and certificates of remote overlay sockets. When the group key method is enabled, the key table is not used.

The certificate table and key table store information about remote overlay sockets with which the local socket exchanges protocol messages. These remote overlay sockets fall into two groups: (1) Sockets that are current neighbors in the overlay topology, and (2) overlay sockets that are non-neighbors. An example of a non-neighbor is a newly joining overlay socket that announces itself to all overlay sockets, but does not become a neighbor of all sockets that receive the message. Another example are overlay sockets that are involved in the rendezvous process, e.g., buddies or rendezvous servers. For some security configurations, the overlay socket employs different authentication methods for neighbors and non-neighbors. For this reason, the key vault maintains separate sub-tables for certificates and keys of neighbors and non-neighbors.

Periodically, the key vault tables are searched for expired entries that need to be deleted. This period is given by the attribute *KeyVaultCleanUp*. A timer controlled by the security processor initiates the clean up of expired entries.

The key vault is used extensively by the security processor and functions that process protocol and overlay messages. The security processor accesses the key vault to check and update the local certificate and key, and to verify the existence and validity of certificates and keys of remote overlay sockets. If the desired certificate or key is not available in the key vault, the security processor initiates a certificate or key exchange with the remote overlay socket. The functions in the overlay socket that process protocol and overlay messages rely on the key vault for information on the security policy, access to local and remote keys and certificates, and to access and creation of new keys.

5.11.2 SECURITY PROCESSOR

The security processor provides authentication and key management, as well as handling of secure protocol messages. For any incoming secure protocol message, the security processor checks the presence and validity of the certificate and key for the sender of the message. If necessary, and as discussed earlier in this chapter, the security processor initiates a certificates or key exchange. All *SecInfoExchange* messages for authentication and key exchange (*CertRequest*, *CertReply*, *KeyRequest*, *KeyUpdate*) are created and handled by security processor independent of the activated overlay protocol.

The security processor hides the presence of security features form the overlay node. An overlay node is not aware of security and exchanges plaintext protocol messages with other overlay nodes. When security is specified, all protocol messages are protected by an signed hash, the *Protocol Message MAC*, which is computed in the security processor.

As indicated in Figure 13, the security processor supports the same interfaces as the node adapter. In fact, to an overlay node, the security processor serves as the node adapter, and to the node adapter, the security processor works as the overlay node. Neither the overlay node nor the node adapter is aware of the existence of the security processor. Protocol messages are wrapped as *SecInfoExchange* messages by the security processor before transmission. At the receiver side, the *SecInfoExchange* messages are converted into plaintext protocol messages by the security processor and passed to the overlay node to process.

Operations in the security processor are triggered by the transmission of protocol messages by the overlay node, changes to the neighborhood table in the overlay node, the arrival of *SecInfoExchange* messages from the node adapter, and timeouts of timers

that were previously set by the security processor. The handling of changes to the neighborhood tables requires special considerations. Recall that upon a change to the neighborhood in an overlay node the security processor may need to update and distribute its neighborhood key. Since the change to the neighborhood table is internal to the overlay node and does not necessarily result in the transmission of protocol messages, the security processor catches the *NeighborhoodChanged* event which is defined through the HyperCast event notification system (see Chapter "Overlay Socket API (Advanced)") by providing an event handling routine for the *NeighborhoodChanged* event.

Figure 16 shows the structure of the class *SecurityProcessor* that implements the security processor.The security processor can be viewed as a layer that bridges the overlay node and the node adapter. To an overlay node, the security processor works as the node adapter which implements the *I_MulticastAdapter* interface. To the node adapter, the security processor serves as the overlay node that implements the *I_AdapterCallback* interface. Neither the overlay node nor the node adapter are aware of the existence of the security processor. Figure 15 shows the structure of class *SecurityProcessor*.

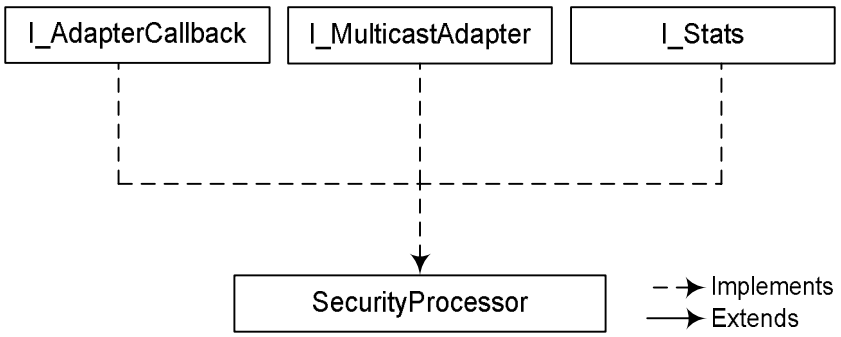


Figure 15. The class structure of *SecurityProcessor*

The class *SecInfoExchange_Message* implements the *I_Message* interface with the standard methods to process protocol and overlay messages. Figure 16 shows the structure of *SecInfoExchange*_Message class.

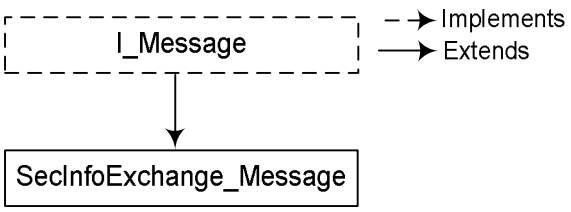


Figure 16. *SecInfoExchange_Message* Class.

Figure 17 and 18 show the flow of outgoing and incoming secure protocol messages. An outgoing protocol message created in the overlay node is wrapped by the security processor into a *SecInfoExchange* message, and sent out through the node adapter. When the security processor receives a protocol message from the overlay node, it creates a *SecInfoExchange* header. Before the message is transmitted, the

SecInfoExchange header is transformed into a byte array and concatenated with the byte array of the protocol message. The result is sent by the node adapter.

An incoming secure protocol message is received in the node adapter as a byte array, which is reconstructed into a *SecInfoExchange* message by the security processor ((1) in Figure 18). This is done by calling the *restoreMessage* method of an *SecInfoExchange* message. This method restores the *SecInfoExchange* header information and verifies the Protocol Message MAC. The reconstructed *SecInfoExchange* message is passed to the *messageArrivedFromAdapter* method of the security processor where the byte array is restored into a plaintext protocol message via the *restoreMessage* method of the protocol message ((3) in Figure 18). Finally, the method *messageArrivedFromAdapter* of the overlay node is invoked to process the message ((4) in Figure 18).

Similarly, when the security processor needs to send a *CertRequest* or *KeyRequest* message, it creates a *SecInfoExchange* message and passes it to the node adapter. When a *CertReply* or *KeyUpdate* message is received by the node adapter, the node adapter invokes the *restoreMessage* method, which in turn calls the *restoreMessage* of the *SecInfoExchange* message. The reconstructed *SecInfoExchange* message is passed to the method *messageArrivedFromAdapter* in of the security processor where the *SecInfoExchange* message is processed.

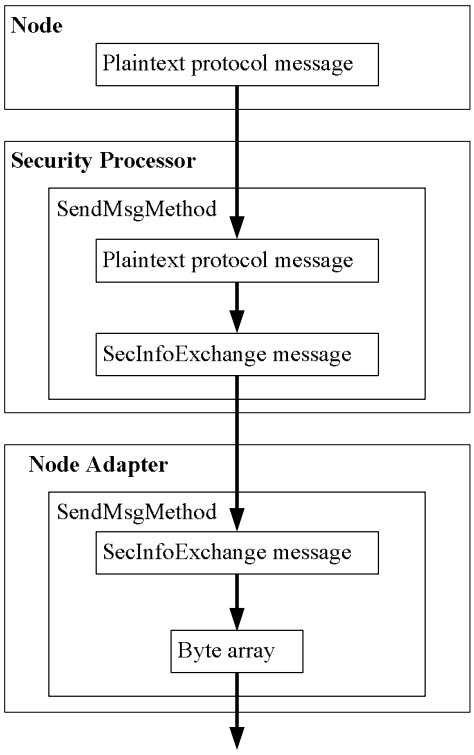


Figure 17. Processing an outgoing secure protocol message.

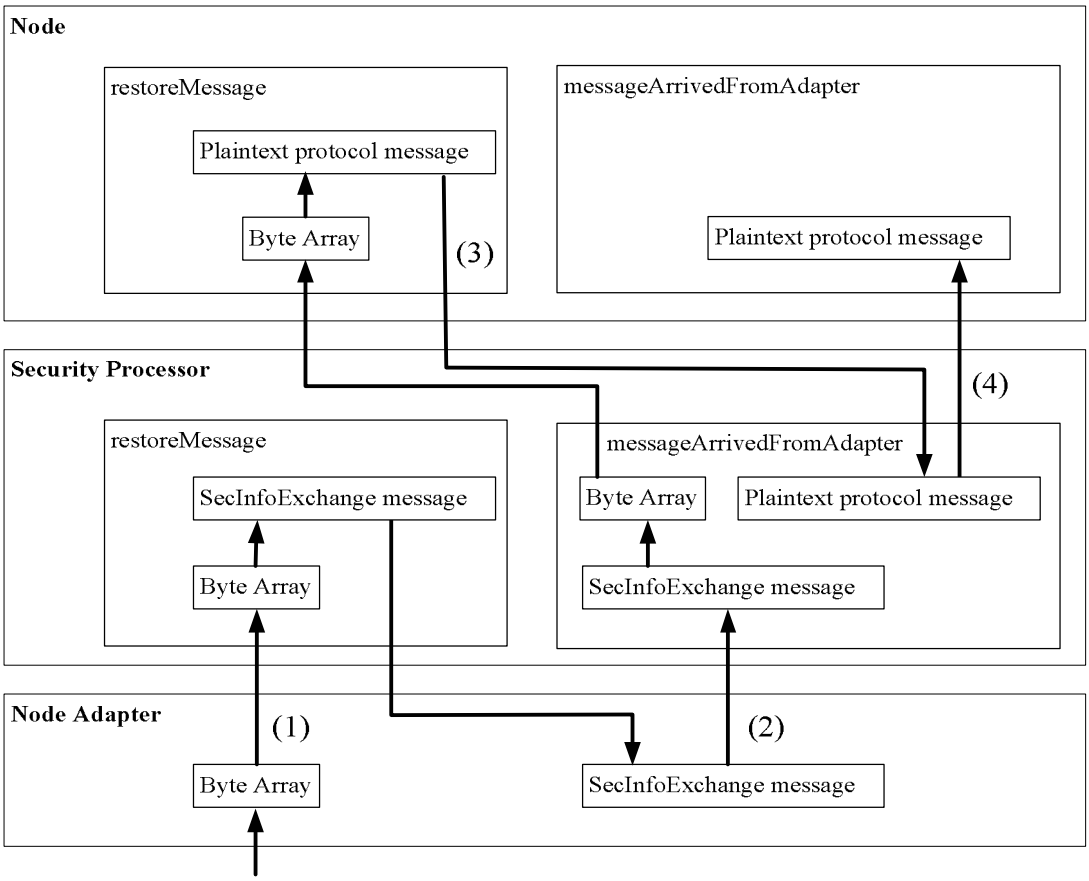


Figure 18. Processing an incoming secure protocol message.

5.11.3 SECURE OVERLAY MESSAGES

We now turn to the implementation aspects of secure overlay messages. Unencrypted overlay message payloads are implemented by the *PayloadExtension* class. When the security level is set to confidentiality, the encryption and decryption of the payload is handled by an instance of the *EncryptedPayloadExtension* class. Figure 18 shows the inheritance structure of the class.

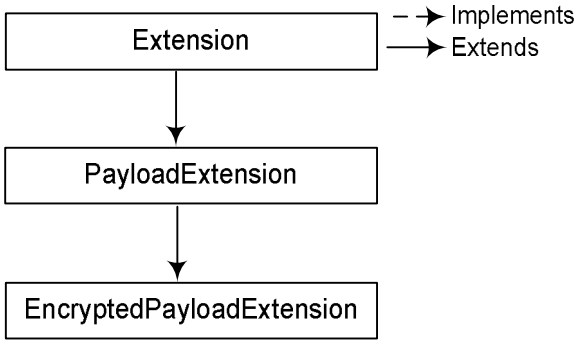


Figure 19. *EncryptedPayloadExtension* Class.

Encryption and decryption of message payloads is transparent to the class that implements the overlay message, and, thereby, to the application. An overlay message treats an *EncryptedPayloadExtension* as a normal *Extension* instance. When the methods *toByteArray()* or *getPayload()* are called for this type of extension, the encryption or decryption is started. To reduce the time spent on encrypting or decrypting data, payload encryption and decryption is only done when an overlay message is converted into a byte array, the encryption on the payload is executed, and decryption is done only when the plaintext payload is needed.

Another optimization is that the overlay socket avoids encryption or decryption the payload of a message more than once, even if a message is retransmitted multiple times. This is done by storing a plaintext copy and an encrypted copy of an encrypted message in the *EncryptedPayloadExtension* object. Encryption of a message is performed only when the encrypted copy does not exist and decryption is done only when the plaintext copy of the payload is not available.

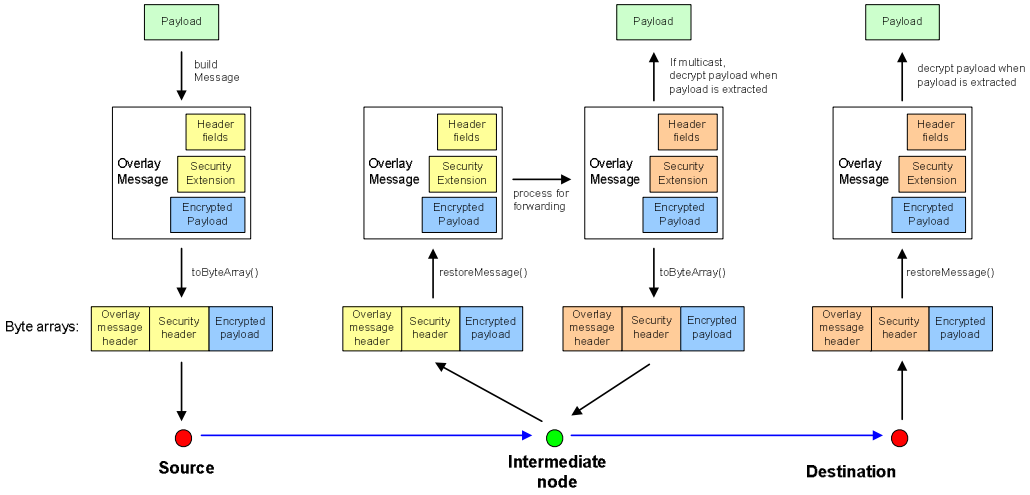


Figure 20. Processing an encrypted overlay message.

5.12 BIBLIOGRAPHY

[1] [AES01] Announcing the Advanced Encryption Standard Federal Information Processing Standards Publication, National Institute of Standards and Technology, 2001.

[2] [DSS00] Digital Signature Standard. Daley, W.M. ed. Federal Information Processing Standards Publication Series, National Institute of Standards and Technology, 2000.

[3] [SHS95] Secure Hash Standard. Brown, R.H. ed., National Institute of Standards and Technology, 1995.

[4] [RFC2104] H. Krawczyk, M.B., R. Canetti. HMAC: Keyed-Hashing for Message Authentication, Internet Engineering Task Force, RFC 2104, 1997.

[5] [Gar94] S. Garfinkel. PGP: Pretty Good Privacy. O'Reilly, November 1994.

[6] [Datta03] A. Datta, M. Hauswirth, and K. Aberer. Beyond `Web of Trust': Enabling P2P E-commerce. In Proceedings of IEEE Conference on E-Commerce (CEC'03), June 2003.

[7] [Chen00] R. Chen and W. Yeager. Poblano: A Distributed Trust Model for Peer-to-Peer Networks. Sun Microsystems Technical Paper, 2000.

[8] [Aberer01] K. Aberer and Z. Despotovic. Managing Trust in a Peer-2-Peer Information System. In Proceedings of the 10th International Conference on Information and Knowledge Management (ACM CIKM), New York, USA, 2001.

[9] [Cornelli02] F. Cornelli, E. Damiani, S. D. C. D. Vimercati, S. Paraboschi, and S. Samarati. Choosing Reputable Servents in a P2P Network. In Proceedings of the 11th World Wide Web Conference, Hawaii, USA, May 2002.

[10] [Kamvar03] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust algorithm for reputation management in P2P networks. In Proceedings of the 12th International Conference World Wide Web, Budapest, Pages 640 – 651, May 2003.

[11] [Castro02] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Security for Peer-to-Peer Routing Overlays. In Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02), December 2002.

[12] [Wallach02] D.S. Wallach. A Survey of Peer-to-Peer Security Issues. In Proceedings of the International Symposium on Software Security, Tokyo, Japan, November 2002.

[13] [Buragohain03] C. Buragohain, D. Agrawal, and S. Suri. A Game Theoretic Framework for Incentives in P2P Systems. In Proceedings of the 3rd International Conference on Peer-to-Peer Computing (P2P '03), 2003.

[14] [Chun04] B.-G. Chun, R. Fonseca, I. Stoica, and J. Kubiatowicz. Characterizing selfishly constructed overlay networks. In Proceedings of IEEE INFOCOM'04, Hong Kong, March 2004.

[15] [Feldman04] M. Feldman, K. Lai, I. Stoica, and J. Chuang. Robust incentive techniques for peer-to-peer networks. In Proceedings of the Fifth ACM Conference on Electronic Commerce (EC'04), New York, NY, June 2004, pp. 102–111.

[16] [Kamvar03] D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. Incentives for Combatting Freeriding on P2P Networks. In Proceedings of EURO-PAR, 2003. International Conference on Parallel and Distributed Computing, August 2003.

[17] [Reiter96] M. K. Reiter, M. K. Franklin, J. B. Lacy, and R. N. Wright. The Omega key management service. *Journal of Computer Security* 4(4):267–287, IOS Press, 1996.

[18] [Zhou??] L. Zhou, F. B. Schneider, R. Van Renesse. COCA: A Secure Distributed On-line Certification Authority. ACM Transactions on Computer Systems, 20(4):329-368.

[19] [Asokan00] N. Asokan, P. Ginzboorg. Key Agreement in Ad Hoc Networks. Computer Communications, 23:1627-1637, 2000.

[20] [Hubaux03] J.-P. Hubaux, L. Buttyan, S. Capkun. Self-Organized Public-Key Management for Mobile Ad Hoc Networks. IEEE Transaction on Mobile Computing, 2(1): 52-[26]64, January/March 2003.

[21] [Luo05] H. Luo, J. Kong, P. Zerfos, S. Lu, and L. Zhang, URSA: Ubiquitous and Robust Access Control for Mobile Ad Hoc Networks, ACM/IEEE Transactions on Networking, 2005 (to appear).

[22] [Wang03] W. Wang, Y. Zhu, and B. Li. Self-Managed Heterogeneous Certification in Mobile Ad Hoc Networks. In Proceedings of IEEE Vehicular Technology Conference (VTC 2003), October 2003.

[23] [Yi03] S. Yi and R. Kravets, MOCA: Mobile Certificate Authority for Wireless Ad Hoc Networks, In Proceedings of 2nd Annual PKI Research Workshop Program (PKI 03), Gaithersburg, April 2003.

[24] [Yi02] S. Yi and R. Kravets. Practical PKI for Ad Hoc Wireless Networks. University of Illinois, Computer Science, Report. No. UIUCDCS-R-2002-2273, May 2002.

[25] [Zhou99] L. Zhou and Z. J. Haas. Securing Ad Hoc Networks. IEEE Network, 13(6):24-30, 1999.

[26] [Shamir79] A. Shamir. How to share a secret. Communications of the ACM, 22(11):612-613, 1979.

[27] [Andert02] D. Andert, R. Wakefield, and J. Weise. Trust Modeling for Security Architecture Development. Sun Microsystems Sun BluePrint, December 2002.

[28] [RFC ???] M. J. Moyer, J.R.R., P. Rohatgi. Maintaining Balanced Key Trees for Secure Multicast, Internet Engineering Task Force, 1999.

[29] [RFC ????] S. Kent, R.A. IP Authentication Header, Network Working Group, Internet Engineering Task Force, 1998.

[30] [RFC ????] S. Kent, R.A. IP Encapsulating Security Payload (ESP), Network Working Group, 1998.