

# ***Chapter 3***

## **The Kernel Implementation**

*Wherein the implementation details of the cryptlib security kernel are examined.*



## 1. Kernel Message Processing

The cryptlib kernel acts as a filtering mechanism for all messages which pass through it, applying a configurable set of filtering rules to each message. These rules are defined in terms of pre- and post-dispatch actions which are performed for each message. In terms of the separation of mechanism and policy requirement given in a previous chapter, the filter rules provide the policy and the kernel provides the mechanism. The advantage of using a rule-based policy is that it allows the system to be configured to match user needs and to be upgraded to meet future threats which hadn't been taken into account when the original policy for the system was formulated. In a conventional approach where the policy is hard-coded into the kernel, a change in policy may require the redesign of the entire kernel. Another advantage of a rule-based policy of this type is that it can be made fairly flexible and dynamic to account for the requirements of particular situations, for example allowing the use of a corporate signing key only during normal business hours, or locking down access or system functionality during a time of heightened risk. A final advantage is that an implementation of this type can be rather easier to verify than more traditional implementations, an issue which is covered in more detail in a later chapter.

### 1.1. Rule-based Policy Enforcement

The advantage of a kernel which is based on a configurable ruleset is that it's possible to respond to changes in requirements without having to redesign the entire kernel. Each rule functions as a check on a given operation, specifying which conditions must hold in order for the operation to execute without breaching the security of the system. When the kernel is presented with a request to perform a given operation, it looks up the associated rule and either allows or denies the operation. The cryptlib kernel also applies rules to the result of processing the request, although it appears to be fairly unique in this regard.

```

static constraint Simple_Security_Policy
begin
  -- for all subjects and objects it must be true that
  for all sub : Subjects; ob : Objects |
    -- current read or write access between a subject and an object implies that
    ( read in current_access( sub, ob ) or write in current_access( sub, ob ) ) -->
      -- the current security label of the subject dominates the object
      current_security_label( sub ) >= security_label( ob );
end Simple_Security_Policy;

```

**Figure 1: Bell-LaPadula simple security policy expressed as SMDE rule**

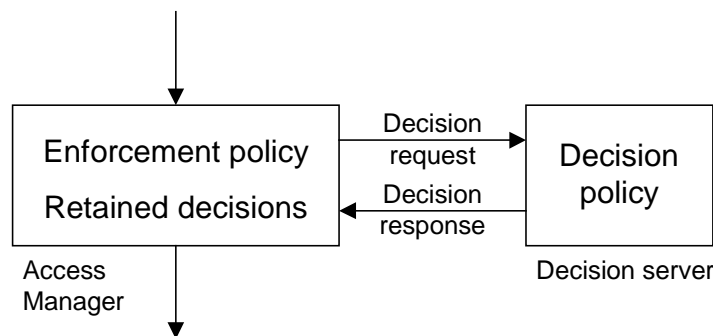
The use of a fixed kernel implementing a configurable rule-based policy provides a powerful mechanism which can be adapted to meet a wide variety of security requirements. One implementation of this concept, the Security Model Development Environment (SMDE), uses a rule-based kernel to implement various security models such as the Bell-LaPadula model, the military message system (MMS) model which is based on mandatory controls on information flow, and the MAC portion of the SeaView relational database model. These policies are enforced by expressing each one in a common notation of which an example is shown in Figure 1, which is then parsed by a model translator tool and fed to a rule generator which creates rules for use by the kernel based on the parsed policy information. Finally, the kernel itself acts as an interpreter for the rule generator [1]. Another, more generalised approach, the Generalised Framework for Access Control (GFAC) proposed the use of a TCB-resident rule base which is queried by an access decision facility (ADF) with the decision results enforced by an access enforcement facility (AEF). The GFAC implements both MAC and DAC controls which can be configured to match a particular organisation's requirements [2][3]. Closely related work in this area is the ISO access control framework (from which the ADF/AEF terminology originates) [4][5], although this was presented in a very abstract sense which was intended to be suitable for a wide variety of situations such as network access control. These approaches may be contrasted with the standard policy enforcement mechanism which relies on the policy being hardcoded into the kernel implementation.

### 1.2. The DTOS/Flask Approach

A slightly different approach is taken by the Distributed Trusted Operating System (DTOS) which provides security features based on the Mach microkernel [6][7]. The DTOS policy enforcement mechanism is based

on an enforcement manager which enforces security decisions made by a decision server as shown in Figure 2. This approach was used because of perceived shortcomings in the original trusted Mach approach (which was described in a previous chapter) in which access control decisions were based on port rights, so that someone who gained a capability for a port had full access to all capabilities on the associated object. Because trusted Mach provides no object-service-specific security mechanisms, it provides no direct control over object services. The potential solution of binding groups of object services to ports has severe scalability and flexibility problems as the number of groups is increased to provide a more fine-grained level of control, and isn't really practical.

The solution to the problem was to develop a mechanism which could ensure that each type of request made of the DTOS kernel is associated with a decision which has to be made by the decision server before the request can be processed by the kernel. The enforcement manager represents the fixed portion of the system which identifies where in the processing a security decision is needed and what type of decision is needed, and the decision server represents the variable portion of the system which can be configured as required to support particular user needs. A final component of the system is a cache of retained decisions which have been made by the decision server, which is required for efficiency reasons in order to speed access in the distributed Mach system [8].



**Figure 2: DTOS security policy management architecture**

As Figure 2 indicates, this architecture bears some general resemblance to the cryptlib kernel message processing mechanism, although in cryptlib security decisions are made directly by the kernel based on a built-in ruleset rather than by an external decision component. Another difference between this and the cryptlib implementation is that DTOS doesn't send the parameters of each request to the decision server which rather limits its decision-making abilities. In contrast in the cryptlib kernel all parameters are available for review, and it is an expected function of the kernel that it subject them to close scrutiny.

One feature of DTOS which arose from the observation that most people either can't or won't read a full formal specification of the security policy is the use of a simple, table-based policy specification approach. This was used in DTOS to implement a fairly conventional MLS policy and the Clark-Wilson policy (as far as it's possible), with enforcement of other policies such as ORCON being investigated.

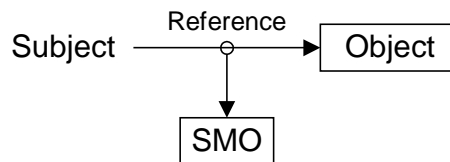
A later refinement of DTOS was Flask which, like cryptlib, has a reference monitor which interposes atomically on each operation performed by the system to enforce security policy [9]. Flask was developed in order to correct some shortcomings in DTOS, mostly to do with dynamic policy changes. Although the overall structure is similar to its ancestor DTOS, Flask includes a considerable amount of extra complexity which is required in order to handle sudden policy changes (which can involve undoing the results of previous policy decisions and aren't made any easier by the presence of the retained decision cache which no longer reflects the new policy) and a second level of security controls which are required to control access to the policies for the first level of security controls. Since the cryptlib policy is fixed when the system is built and very specifically can't be changed after this point, there's no need for a similar level of complexity in cryptlib.

An even more extreme version of this approach which is used in specialised systems where the subjects and their interactions with objects is known at system build time compiles not only the rules but also the access control decisions themselves into the system. An example of such a situation occurs in integrated avionics

environments where, due to the embedded and fixed nature of the application, the roles and interactions of all subjects and objects is known *a priori* so that all access mediation information can be assembled at build time and loaded into the target system in preset form [10]. Taking this approach has little benefit for cryptlib since its main advantage is to allow for faster startup and initialisation, which in the application mentioned above leads to “faster turnaround and takeoff” which isn’t generally a consideration for the situations where cryptlib is used.

### 1.3. Meta-Objects for Access Control

Another access control mechanism which has some similarity to the one implemented in the cryptlib kernel is that of security meta-objects (SMOs), meta-objects which are attached to object references to control access to the corresponding object and which can be used to implement arbitrary and user-defined policies. SMOs are objects which are attached to an object reference (in cryptlib terms an object’s handle) which control access to the target object via this reference. An example of an object with an SMO attached to its reference is shown in Figure 3. The meta object has the ability to determine whether a requested type of access via the reference is permissible or not, and can perform any other types of special-case processing which may be required [11][12].



**Figure 3: Security meta-object attached to an object reference**

If a subject tries to access an object via the protected reference, the SMO is implicitly invoked and can perform access checking based on the subject identity and the parameters being passed across in the access to the protected object. If the SMO allows the access, everything continues as normal. If it denies the access, the invocation is terminated with an error result.

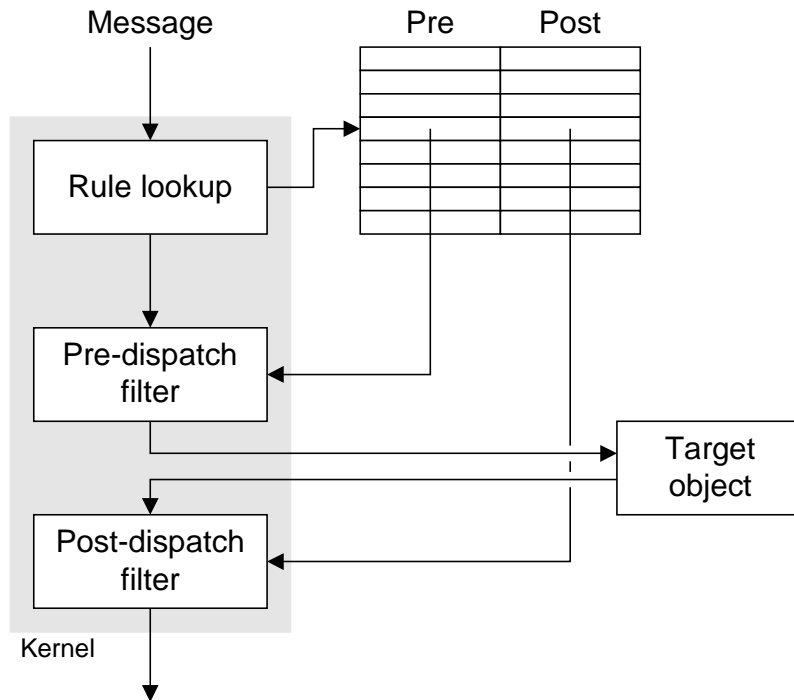
The filter rules used in the cryptlib kernel differ from the SMOs discussed above in several ways, the main one being that while SMOs are associated with references to an object, kernel filter rules are associated with messages and are always invoked. In contrast SMOs are invoked on a per-reference basis so that one reference to an object may have an SMO attached while a second reference is free of SMOs. In addition the kernel contains filter rules for both pre- and post-access states while SMOs only apply for the pre-access state (although this would be fairly easy to change if required). A major feature of SMOs is that they provide an extended form of capability-based security, fixing many of the problems of capability-based systems such as revocation of capabilities (implemented by having the SMO disallow access when the capability is revoked) and control over who has a given capability (implemented by having the SMO copied across to any new reference which is created, thus propagating its security policy across to the new reference) [13]. Because of these mechanisms, it’s not possible for a subject to obtain an unprotected reference to an object.

### 1.4. Access Control via Message Filter Rules

The principle interface to the kernel is the `krnlSendMessage` function, which provides the means through which subjects interact with objects. When a message arrives through a call to `krnlSendMessage` the kernel looks up the appropriate pre- and post-processing rules and information based on the message type and applies the pre-dispatch filtering rule to the message before dispatching it to the target object. When the message is returned from the object it applies the post-dispatch filtering rule and returns the result to the caller. This message-filtering process is shown in Figure 4.

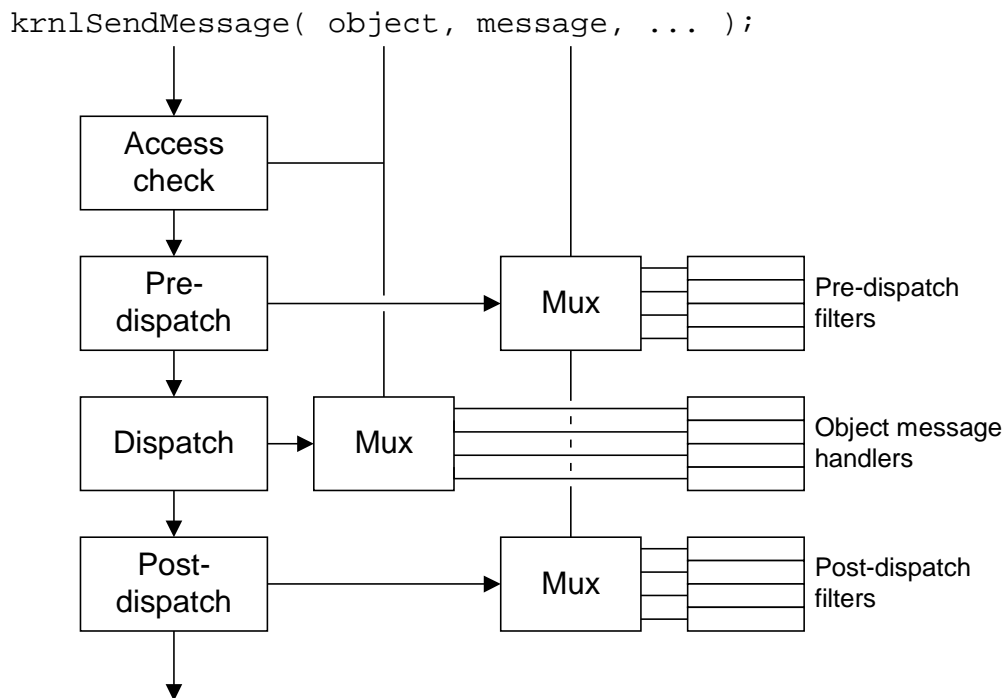
The processing which is being performed by the kernel is driven entirely by the filter rules and doesn’t require that the kernel have any built-in knowledge of object types, attributes, or object properties. This means that although the following sections describe the processing in terms of checks on objects, access and usage permissions, reference and usage counts, and the various other controls which are enforced by the kernel, this checking is performed entirely under the control of the filter rules and the kernel itself doesn’t contain any

built-in knowledge of (say) an object's usage count or what it signifies. This means that new functionality can be added at any point by adding new filter rules or by amending or extending existing ones. An example of this type of change is given later when the rules are amended to enforce the FIPS 140 security requirements, but they could just as easily be applied to enforce a completely different, non-cryptographic policy.



**Figure 4: Kernel message filtering**

The general similarities of this arrangement and the one used by DTOS/Flask are fairly obvious, in both cases a fixed kernel consults an external rule base to determine how to react to a message. As has been pointed out earlier, cryptlib provides somewhat more complete mediation by examining the message parameters and not just the message itself and by providing post-dispatch filtering as well as the pre-dispatch filtering used in DTOS/Flask.



**Figure 5: Filter rule application**

The manner in which the filter rules are applied to each message being processed is shown in Figure 5. The first check which is applied is a general access check on the object to which the message is addressed, the details of which were given in the previous chapter. Once this check has been passed, the pre-dispatch filter rule, selected by the type of the message being processed, is applied. The message is then dispatched to the appropriate object handler, after which the post-dispatch filter, again selected by message type, is applied. Finally, the result of the processing is returned to the caller.

## 2. Filter Rule Structure

Each filter rule begins with an indication of the message type which it applies to. This information isn't required for the implementation itself since the kernel performs the rule lookup via a simple table lookup based on the message type, but is used as part of the internal consistency checks which are performed by the kernel when it is initialised.

The next series of entries contain routing information for the message. If the message has an implicit target type (for example a generate key message is always routed to an encryption context) then the type is specified here. If the message has special-case routing requirements then a handler which performs this routing is specified here. As was mentioned earlier, the filtering code has no explicit knowledge of object types, it just applies the routing mechanism described in a previous chapter to ensure sure that whatever type is given in the rule matches the target object type.

The next entry is used for type checking, and contains the object subtypes for which this message is valid, for example the generate key message mentioned previously would only be valid for conventional and public-key encryption and MAC contexts. As with object types used for routing, the kernel has no explicit knowledge of object subtypes but just checks to make sure that the subtype for the object which the message is eventually routed to matches one of the subtypes given in the filter rule.

The next entry defines the type of assertion checking which is performed on message parameters. This is used for assertion-based testing of the implementation and is discussed in a later chapter.

The final series of entries contain information about the message handlers. These handlers perform any additional checking and processing which may be necessary before and after a message is dispatched to an object. In addition some message types are handled internally by the kernel (for example a message which increments or decrements an object's reference count), in which case the handlers are kernel-internal mechanisms.

## 2.1. Filter Rules

The message filtering policy definitions are best illustrated with examples of actual filtering rules. The simplest rule is for messages which are handled internally by the kernel without being forwarded to the target object. These include messages to increment and decrement an object's reference count and to manipulate dependent objects. The rules for changing an object's reference count are shown in Figure 6.

```
{ MESSAGE_INCREFCOUNT,          /* Increment object ref.count */
  ROUTE_NONE,
  SUBTYPE_ANY,
  PARAMTYPE_NONE_NONE,
  HANDLE_INTERNAL( incRefCount ) },
{ MESSAGE_DECREFCOUNT,        /* Decrement object ref.count */
  ROUTE_NONE,
  SUBTYPE_ANY,
  PARAMTYPE_NONE_NONE,
  HANDLE_INTERNAL( decRefCount ) }
```

**Figure 6: Rules for messages handled by the kernel**

The first entry in each rule contains the message type which is used for internal consistency checking by the kernel at startup. Following this is the routing information, in this case `ROUTE_NONE` which indicates that this message is addressed directly to its final destination. The next entry contains the object subtypes for which this message is valid, in this case the messages are valid for all object subtypes. The next entry is used for assertion-based testing of the implementation and specifies that the message has no parameters. Finally, the last entry specifies the use of an internal handler which increments or decrements the objects reference count.

When the kernel receives one of these messages, it performs the appropriate checks specified by the filtering rules (in this case none apart from the standard object validity and accessibility checks which are always performed), bypasses the routing stage since the rules indicate that the messages aren't routed, and passes control over to the appropriate internal handler, from which it returns to the caller.

A slightly more complex rule which results in a message being passed on to a destination object is the destroy object message, which is usually not invoked directly but results from an object having its reference count decremented to zero. The rule for the destroy object message is shown in Figure 7, and is almost identical to the ones in Figure 6 except that the use of a pre-dispatch handler is specified which signals any dependent objects that their controlling object is about to have its reference count decremented and places the object in the signalled state to ensure that no further messages will be dispatched to it.

```
{ MESSAGE_DESTROY,            /* Destroy the object */
  ROUTE_NONE,
  SUBTYPE_ANY,
  PARAMTYPE_NONE_NONE,
  PRE_DISPATCH( SignalDependentObjects ) }
```

**Figure 7: Destroy object filter rule**

The messages so far have been ones which are sent directly to their target object, however many messages are routed based on the message type. An example of this type of message is shown in Figure 8. The encrypt data and decrypt data messages are routed to encryption context objects with an object subtype of conventional or public-key encryption object, as was mentioned earlier the kernel doesn't need to know about the exact semantics of the objects involved (the message could just as easily be routed to objects of type cat with subtypes siamese and persian), all it needs to do is correctly apply the rule definitions.

```

{ MESSAGE_CTX_ENCRYPT,          /* Context: Action = encrypt */
  ROUTE( OBJECT_TYPE_CONTEXT ),
  SUBTYPE_CTX_CONV | SUBTYPE_CTX_PKC,
  PARAMTYPE_DATA_LENGTH,
  PRE_POST_DISPATCH( CheckActionAccess, UpdateUsageCount ) },
{ MESSAGE_CTX_DECRYPT,        /* Context: Action = decrypt */
  ROUTE( OBJECT_TYPE_CONTEXT ),
  SUBTYPE_CTX_CONV | SUBTYPE_CTX_PKC,
  PARAMTYPE_DATA_LENGTH,
  PRE_POST_DISPATCH( CheckActionAccess, UpdateUsageCount ) }

```

**Figure 8: Rules for messages routed by object type**

These rules also contain extra functionality in areas other than message routing. Since the encrypt data message requires as parameters the data to be encrypted and its length, the entry for the assertion-based verification specifies this instead of the null parameters used for the previous messages. In addition the pre- and post-dispatch filtering for these messages is more complex than it was for the earlier ones. In each case the pre-dispatch rule applies the access checks which were described in a previous chapter, and the post-dispatch rule updates the object’s usage count if the object returns an indication that the message was processed successfully.

Some messages can change an object’s state, resulting in a transition from the low to the high state if the object reports that they were successfully processed. Examples of two such messages are shown in Figure 9, with the first one being a message which generates a key in a conventional, public-key, or MAC context and the second one being a message which signs a certificate or some variant thereof (a certification request, certificate chain, or attribute certificate) or a CRL.

```

{ MESSAGE_CTX_GENKEY,          /* Context: Generate a key */
  ROUTE( OBJECT_TYPE_CONTEXT ),
  SUBTYPE_CTX_CONV | SUBTYPE_CTX_PKC | SUBTYPE_CTX_MAC,
  PARAMTYPE_DATA_BOOLEAN,
  PRE_POST_DISPATCH( CheckState, ChangeState ) },
{ MESSAGE_CERT_SIGN,         /* Cert: Action = sign cert */
  ROUTE( OBJECT_TYPE_CERTIFICATE ),
  SUBTYPE_CERT_ANY_CERT | SUBTYPE_CERT_CRL,
  PARAMTYPE_NONE_ANY,
  PRE_POST_DISPATCH( CheckStateParamHandle, ChangeState ) }

```

**Figure 9: Rules for messages which change an objects state**

These messages are again automatically routed to the appropriate object type, before being dispatched a filter rule is applied which checks to ensure that the object isn’t already in the high state and (in the case of the certificate) also checks that the signing key parameter is valid for this type of operation. If the target object reports the successful processing of the message, the kernel applies a post-dispatch filter which moves the object into the high state.

Some messages aren’t routed (in the same way as if they had a routing entry of ROUTE\_NONE) but don’t apply to all object types, being specific to only one or occasionally two object types. Examples of two such messages which create an object in a device and get a key from a keyset or device are shown in Figure 10. The first rule specifies that the message to create an object must be targeted specifically at a device and that the assertion-based verification will require a parameter indicating the object type which is to be created, the second rule specifies that the message to get a key (which results in the instantiation of encryption contexts and/or certificates) must be targeted at a device or a keyset.

```

{ MESSAGE_DEV_CREATEOBJECT,   /* Device: Create object */
  ROUTE_FIXED( OBJECT_TYPE_DEVICE ),
  SUBTYPE_ANY,
  PARAMTYPE_DATA_OBJTYPE },
{ MESSAGE_KEY_GETKEY,        /* Keyset: Instantiate ctx/cert */
  ROUTE_FIXED_ALT( OBJECT_TYPE_KEYSET, OBJECT_TYPE_DEVICE ),
  SUBTYPE_ANY,
  PARAMTYPE_DATA_NONE }

```

**Figure 10: Rules for messages with fixed routing and alternative targets**

In theory we could allow routing of such messages, for example a “get key” message sent to a certificate could be interpreted to mean “get another key from the same location that this one came from” and, with the

appropriate rule changes, the kernel would indeed perform this action, however this type of functionality is probably stretching the orthogonality of the message-based implementation a bit too far, and would probably only cause confusion among users.

Some message types are internal to cryptlib and are used to invoke mechanisms and actions which can never be directly accessed by the user. Examples of rules for two such messages are shown in Figure 11. These rules apply to messages which are used to wrap one key in another (for example a session key in a public key) and to perform the corresponding unwrapping action. The rules are fairly straightforward, requiring that a valid wrapping or unwrapping mechanism is used as part of the message and checking that the supplied object types and parameters are appropriate for the mechanism.

```
{ MESSAGE_DEV_EXPORT,          /* Device: Action = export key */
  ROUTE( OBJECT_TYPE_DEVICE ),
  SUBTYPE_ANY,
  PARAMTYPE_DATA_MECHTYPE,
  PRE_DISPATCH( CheckMechanismWrapAccess ) },
{ MESSAGE_DEV_IMPORT,        /* Device: Action = import key */
  ROUTE( OBJECT_TYPE_DEVICE ),
  SUBTTYPE_ANY,
  PARAMTYPE_DATA_MECHTYPE,
  PRE_DISPATCH( CheckMechanismWrapAccess ) }
```

**Figure 11: Rules for messages which invoke internal mechanisms**

Examples of a final class of processing rules, which apply to messages which manipulate object attributes, are given in Figure 12. These messages are routed implicitly by attribute type, so that for example a message which manipulates an encryption key attribute would be implicitly routed to an encryption context and a message which manipulates a signature creation time attribute would be implicitly routed to a certificate object.

```
{ MESSAGE_GETATTRIBUTE,      /* Get numeric object attribute */
  ROUTE_IMPLICIT,
  SUBTYPE_ANY,
  PARAMTYPE_DATA_ANY,
  PRE_DISPATCH( CheckAttributeAccess ) },
{ MESSAGE_SETATTRIBUTE,     /* Set numeric object attribute */
  ROUTE_IMPLICIT,
  SUBTYPE_ANY,
  PARAMTYPE_DATA_ANY,
  PRE_POST_DISPATCH( CheckAttributeAccess, ChangeStateOpt ) },
{ MESSAGE_DELETEATTRIBUTE,  /* Delete object attribute */
  ROUTE_IMPLICIT,
  SUBTYPE_CTX_ANY | SUBTYPE_CERT_ANY,
  PARAMTYPE_NONE_ANY,
  PRE_DISPATCH( CheckAttributeAccess ) }
```

**Figure 12: Rules for attribute-manipulation messages**

In each case the pre-dispatch filter rule which is applied is one which checks the attribute data and ensures that the access is valid. For the set attribute message the attribute being set may result in the object being transitioned into the high state (for example this would happen if the attribute was a key being set for an encryption context), so a post-dispatch rule is applied which performs the state change if required.

### 3. Attribute ACL Structure

As with the message filter rules, each attribute ACL begins with an indication of the attribute type which it applies to which is used as part of the internal consistency checking performed by the kernel when it is initialised.

The next series of entries are used for type checking and specify the type of the attribute (whether it's a boolean, a numeric value, an object, a string, or various other types) and the object subtype for which the attribute is valid. As with the type information for messages, the kernel has no explicit knowledge of object subtypes but just checks to make sure that the subtype for the object for which the attribute is being manipulated matches one of the subtypes given in the ACL.

The next series of entries contain the access restrictions for the attribute and a series of flags which define additional handling restrictions and conditions for the attribute. The access restrictions are a standard bitmap of read/write/delete (RWD) permissions for internal and external access with a one bit indicating that this type of access is allowed. There are two sets of permissions, one for the object when it's in the low state and one when it's in the high state. If an attribute is accessible both internally and externally then the RWD permissions are identical for internal and external access, if the attribute is only visible internally then the RWD permissions for external access are set to all zeroes. Some example permissions and the attributes they might apply to are given in Table 1. The RWD permissions are divided into two groups, with the first group applying when the object is in the low state and the second group applying when the object is in the high state.

Permission	Description
ACCESS_xxx_xxx	No access from anywhere in any state. This is used for placeholder attributes which represent functionality which will be added at a later date.
ACCESS_xxx_Rxx	Read-only access in the low state, no access in the high state. This is used for status information when the object is in the low state which has no meaning any more once it's been moved into the high state, for example the details of a key which is required in order to move the object into the high state.
ACCESS_Rxx_xxx	Read-only access in the high state, no access in the low state. This is used for information which is created when the object changes states, for example a certificate fingerprint (hash of the encoded certificate) which only exists once the certificate has been signed and is in the high state.
ACCESS_xxx_RWx	Read/write access in the low state, no access in the high state. This is a variant of ACCESS_xxx_Rxx and is used for information which has no meaning in the high state but is required in the low state.
ACCESS_Rxx_RWD	Full access in the low state, read-only access in the high state. This is used for information which can be manipulated freely in the low state but which becomes immutable once the object has been moved into the high state, typical examples being certificate attributes.
ACCESS_RWD_xxx	Full access in the high state, no access in the low state. This is used for information pertaining to fully initialised objects (for example signed certificates) which doesn't apply when the object is in the low state where the details of the object are still subject to change.
ACCESS_INT_xxx_Rxx	Internal read-only access in the low state, no external access or access in the high state. This is identical to ACCESS_xxx_Rxx except that it's used for attributes which are only visible internally.
ACCESS_INT_Rxx_RWx	Internal read/write access in the low state, internal read-only access in the high state, no external access. This is mostly identical to ACCESS_Rxx_RWD (except for the lack of delete access) but is used for attributes which are only visible internally.

**Table 1: Examples of attribute access permissions**

The flags which accompany the access permissions indicate any additional handling which must be performed by the kernel. There are only two of these, the first one being `ATTRIBUTE_FLAG_PROPERTY` which indicates that the attribute applies to the object itself rather than being an attribute of the object. Examples of attribute properties include the object type, whether the object is externally visible, whether the object is in

the low or high state, and so on (all of these properties are internal attributes, so that the corresponding access permissions are `ACCESS_INT_xxx`). The second flag is `ATTRIBUTE_FLAG_TRIGGER`, which indicates that setting this attribute triggers a change from the low to the high state. As with messages which initiate this change, if the object reports that a message which sets an attribute with the `ATTRIBUTE_FLAG_TRIGGER` flag set was processed successfully, the kernel will move the object into the high state. Examples of trigger attributes are ones which contain key components such as public keys, user passwords, or conventional encryption keys.

The next series of entries contain routing information for the message which affects the attribute. If the message has an implicit target type which is given via the attribute type then the target type is specified here. If the message has special-case routing requirements then a handler which performs this routing is specified here. As with the message-routing code, the kernel has no explicit knowledge of object types, it just applies the routing mechanism described in a previous chapter to ensure sure that whatever type is given in the ACL entry matches the target object type.

The final series of entries is used for type checking and contains range information for the attribute data (for example a range of 192...192 bits for triple DES keys or 1...64 characters for many X.509 certificate strings) and any additional checking information which may be required. This includes things like sequences of allowed values for the attribute, limits on sub-ranges rather than a single continuous range, an indication that the attribute value must correspond to a valid object, and so on.

### 3.1. Attribute ACLs

As with the message filtering rules, the attribute ACLs are best illustrated through examples. One of the simplest of these is a basic boolean flag indicating the status of a certain condition. The ACL for the `CRYPT_CERTINFO_SELGSIGNED` attribute, which indicates whether a certificate is self-signed (that is, whether the public key contained in it can be used to verify the signature on it) is shown in Figure 13. This ACL indicates that the attribute is a boolean flag which is valid for any type of certificate, that it can be read or written when the certificate is in the low (unsigned) state but only read when it's in the high (signed) state, and that the message which manipulates it is routed to certificate objects.

```

MKACL_B(                                     /* Cert is self-signed */
  CRYPT_CERTINFO_SELFSIGNED,
  SUBTYPE_CERT_ANY_CERT,
  ACCESS_Rxx_RWx,
  ROUTE( OBJECT_TYPE_CERTIFICATE ) )

```

**Figure 13: ACL for boolean attribute**

Two slightly more complex entries which apply for attributes with numeric values are shown in Figure 14. Both are for encryption contexts, and both are read-only, since the attribute value is set implicitly when the object is created. The first ACL is for the encryption algorithm which is used by the context, the allowable range is defined in terms of the predefined constants `CRYPT_ALGO_NONE` and `CRYPT_ALGO_LAST`. The attribute is allowed to take any value which is within these two limits. The second ACL is for the block size of the algorithm used by the context, the allowable range is defined in terms of the largest block size used by any algorithm, which in this case is the size of the hash value produced by a hash algorithm.

```

MKACL_N(                                     /* Algorithm */
  CRYPT_CTXINFO_ALGO,
  SUBTYPE_CTX_ANY,
  ACCESS_Rxx_Rxx,
  ROUTE( OBJECT_TYPE_CONTEXT ),
  RANGE( CRYPT_ALGO_NONE + 1, CRYPT_ALGO_LAST - 1 ) ),
MKACL_N(                                     /* Block size in bytes */
  CRYPT_CTXINFO_BLOCKSIZE,
  SUBTYPE_CTX_ANY,
  ACCESS_Rxx_Rxx,
  ROUTE( OBJECT_TYPE_CONTEXT ),
  RANGE( 1, CRYPT_MAX_HASHSIZE ) )

```

**Figure 14: ACL for numeric attributes**

The two examples shown above illustrate the way in which the kernel is kept separate from any low-level object implementation details. If it knew every nuance of every object's implementation it would know that (for example) a DES object can only have a `CRYPT_CTXINFO_ALGO` attribute value of `CRYPT_ALGO_DES` and a `CRYPT_CTXINFO_BLOCKSIZE` value of 8, however the kernel shouldn't be required to be aware of these details, since all it's enforcing is a general set of rules with any object-specific details being handled by the objects themselves (going back to the cat analogy from earlier on, the rules could just as well be specifying cat fur colours and lengths as encryption algorithms and key sizes). What the kernel guarantees to subjects and objects in terms of message parameters is that the messages it allows through have parameters within the ranges which are permitted for the object as defined by the filter rules it enforces.

An example of ACLs for general-purpose string attributes is shown in Figure 15. The first entry is for the IV for an encryption context, which is a general-purpose string attribute with no restrictions on access so that it can be read or written when the object is in the low or high state. Since only conventional encryption algorithms have IVs, the permitted object subtype range is conventional encryption contexts only. As with the algorithm block size in Figure 14, the allowed size is given in terms of the predefined constant `CRYPT_MAX_IVSIZE`, with the object itself taking care of the exact details (in practice this means it pads short IVs out as required and truncates long ones, the semantics of mismatched IVs are undefined in any crypto standards which provide for the use of variable-length IVs so in practice cryptlib is generous with what it accepts).

```

MKACL_S(                                     /* IV */
    CRYPT_CTXINFO_IV,
    SUBTYPE_CTX_CONV,
    ACCESS_RWx_RWx,
    ROUTE( OBJECT_TYPE_CONTEXT ),
    RANGE( 8, CRYPT_MAX_IVSIZE ) ),
MKACL_S(                                     /* Label for private key */
    CRYPT_CTXINFO_LABEL,
    SUBTYPE_CTX_PKC,
    ACCESS_Rxx_RWD,
    ROUTE( OBJECT_TYPE_CONTEXT ),
    RANGE( 1, CRYPT_MAX_TEXTSIZE ) )

```

**Figure 15: ACL for a string attribute**

The second entry is for the label for a private key, with an object subtype allowing its use only with private key contexts. This attribute contains a unique label which is used to identify a key when it's stored to disk or to a crypto token such as a smart card, typical labels being "My encryption key" or "My signature key". cryptlib enforces the uniqueness requirement by sending a message to the keyset or device which the object will be held in inquiring whether something with this label already exists. If the keyset or device indicates that an object with the given label is already present, a duplicate value error is returned to the user. Because the user could bypass this check by changing the label after the object is stored in or associated with the keyset or device, the label is made read-only once the object is in the high state.

```

MKACL_S(                                     /* Ctx: Key ID */
    CRYPT_IATTRIBUTE_KEYID,
    SUBTYPE_CTX_PKC,
    ACCESS_INT_Rxx_Rxx,
    ROUTE( OBJECT_TYPE_CONTEXT ),
    RANGE( 20, 20 ) )

```

**Figure 16: ACL for internal attribute**

Having looked at some of the more generic attribute ACLs we can now look at the more special-case ones. The first of these is shown in Figure 16, and constitutes the ACL for the key identifier for a public or private key object. The key identifier (also known under a variety of other names such as thumbprint, key hash, `subjectPublicKeyIdentifier`, and various other terms) is an SHA-1 hash of the public key components and is used to uniquely identify a public key both within cryptlib and externally when used with data formats such as X.509 and S/MIME version 3. Since this value isn't something which is of any use to the user, its ACL specifies it as being accessible only within cryptlib. If an outside user tries to access it, they'll be returned an error code indicating that this attribute doesn't exist. Note that this is in contrast to many systems where the error would be permission denied, in cryptlib's case it's not even possible to determine the existence of an internal attribute from the outside.

```

MKACL_S_EX(                                     /* Key */
    CRYPT_CTXINFO_KEY,
    SUBTYPE_CTX_CONV | SUBTYPE_CTX_MAC,
    ACCESS_xxx_xWx,
    ATTRIBUTE_FLAG_TRIGGER,
    ROUTE( OBJECT_TYPE_CONTEXT ),
    RANGE( bitsToBytes( MIN_KEYSIZE_BITS ), CRYPT_MAX_KEYSIZE ) )

```

**Figure 17: ACL for an attribute which triggers an object state change**

Figure 17 indicates another special-case attribute, this time one which, when set, triggers a change in the object's state from the low to the high state. This attribute, the encryption key, is valid for conventional and MAC encryption contexts (public-key contexts have public-key parameters which are somewhat different than standard keys) and when set causes the kernel to transition the object into the high state. An attempt to set it if the object is already in the high state is disallowed, thus enforcing the write-once semantics for encryption keys.

Some security standards don't allow plaintext keys to pass over an external interface, a rule which can be enforced through the ACL change shown in Figure 18. Previously the attribute could be set from inside and outside the architecture, with this change it can only be set from within the architecture. In order to load a key into a context it's now necessary to send in an encrypted key from the outside which can be unwrapped internally and loaded into the context from there, but plaintext keys can't be loaded any more. This example illustrates the flexibility of the rule-based policy enforcement, which allows an alternative security policy to be employed by a simple change to an ACL entry which then takes effect across the entire architecture.

```

MKACL_S_EX(                                     /* Key */
    CRYPT_CTXINFO_KEY,
    SUBTYPE_CTX_CONV | SUBTYPE_CTX_MAC,
    ACCESS_INT_xxx_xWx,
    ATTRIBUTE_FLAG_TRIGGER,
    ROUTE( OBJECT_TYPE_CONTEXT ),
    RANGE( bitsToBytes( MIN_KEYSIZE_BITS ), CRYPT_MAX_KEYSIZE ) )

```

**Figure 18: Modified trigger attribute ACL which disallows plaintext key loads**

## 4. Message Filter Implementation

The previous sections have covered the filter rules which are applied to messages and, at a more fine-grained level, the attributes which are manipulated by messages. This section covers the implementations of some of the filters which are applied by the kernel filtering rules.

### 4.1. Pre-dispatch Filters

One of the simplest filters is the one which is invoked before dispatching a destroy object message, whose implementation is shown in Figure 19. This decrements the reference count for any dependent objects which may exist and moves the object being destroyed into the signalled state which indicates to the kernel that it shouldn't dispatch any further messages to it. Once these actions have been taken, the message is dispatched on to the object for processing.

```

preDispatchSignalDependentObjects ::=
    if( objectInfoPtr->dependentDevice != CRYPT_ERROR )
        decRefCount( objectInfoPtr->dependentDevice, 0, NULL );
    if( objectInfoPtr->dependentObject != CRYPT_ERROR )
        decRefCount( objectInfoPtr->dependentObject, 0, NULL );
    objectInfoPtr->flags |= OBJECT_FLAG_SIGNALED;

```

**Figure 19: Destroy object message filter**

When the object finishes processing the message, the kernel dequeues all further messages for it and clears the object table entry. This is the one message which has an implicit rather than explicit post-dispatch action, since the act of dequeuing messages is logically part of the kernel dispatcher rather than an external filter rule.

```
preDispatchCheckState ::=
    if( isInHighState( objectHandle ) )
        return( CRYPT_ERROR_PERMISSION );
```

**Figure 20: Check object state filter**

The pre-dispatch filter which checks an object's state in response to a message which would transition it into the high state is shown in Figure 20. This is an extremely simple rule which should be self-explanatory.

One of the more complex pre-dispatch filters, which checks that an action which is being requested for an object is permitted, is shown in Figure 21. This begins by ensuring that the object is in the high state (if it isn't, it can't perform any action) and that if the requested action is one which caused a transition into the high state that it can't be applied a second time. In addition it ensures that if the object has a usage count set and it's gone to zero, it can't be used any more.

```
preDispatchCheckActionAccess ::=
    /* If the object is in the low state, it can't be used for any action */
    if( !isInHighState( objectHandle ) )
        return( CRYPT_ERROR_NOTINITED );

    /* If the object is in the high state, it can't receive another message of the kind
       which causes the state change */
    if( message == RESOURCE_MESSAGE_CTX_GENKEY )
        return( CRYPT_ERROR_INITED );

    /* If there's a usage count set for the object and it's gone to zero, it can't be
       used any more */
    if( objectInfoPtr->usageCount != CRYPT_UNUSED && objectInfoPtr->usageCount <= 0 )
        return( CRYPT_ERROR_PERMISSION );

    /* Determine the required level for access. Like protection rings, the lower the
       value, the higher the privilege level. Level 3 is all-access, level 2 is
       internal-access only, level 1 is no access, and level 0 is not-available (eg
       encryption for hash contexts) */
    requiredLevel = \
        objectInfoPtr->actionFlags & MK_ACTION_PERM( message, ACTION_PERM_MASK );

    /* Make sure the action is enabled at the required level */
    if( message & RESOURCE_MESSAGE_INTERNAL )
        /* It's an internal message, the minimal permissions will do */
        actualLevel = MK_ACTION_PERM( message, ACTION_PERM_NONE_EXTERNAL );
    else
        /* It's an external message, we need full permissions for access */
        actualLevel = MK_ACTION_PERM( message, ACTION_PERM_ALL );
    if( requiredLevel < actualLevel )
    {
        /* The required level is less than the actual level (eg level 2 access attempted
           from level 3), return more detailed information about the problem */
        return( ( ( requiredLevel >> ACTION_PERM_SHIFT( message ) ) == ACTION_PERM_NONE )
            ? \
                CRYPT_ERROR_NOTAVAIL : CRYPT_ERROR_PERMISSION );
    }
```

**Figure 21: Check requested action permission filter**

Once the basic security checks have been performed, it then checks whether the requested action is permitted at the object's current security setting. This is a simple comparison between the permission level of the message (in other words the permission level of the subject which sent it) and the permission level set for the object. If the message's permission level is insufficient, the request is denied. Since there are two different ways of saying no, ACTION\_PERM\_NOTAVAIL (it's not there) and ACTION\_PERM\_NONE (it's there but you can't use it), the filter performs a check for why the request was denied and returns the appropriate error code to the caller.

## 4.2. Post-dispatch Filters

The post-dispatch filters are all very simple, mostly performing housekeeping and cleanup tasks after a message has been processed by an object. The one implicit filter, which is invoked after an object has processed a destroy object message, has already been covered. Another post-dispatch filter is the one which updates an object's usage count if it has one set. This filter is shown in Figure 22, and simply decrements the

object's usage count if this is being used. Although it would appear that this filter can decrement the usage count past zero, this can never occur because the pre-dispatch filter shown earlier will prevent further messages from being dispatched to it once the usage count reaches zero. Not shown in the code snippet presented here are the assertion-based testing rules which ensure that this is indeed the case. The testing and verification of the filter rules (and the kernel as a whole) are covered in a later chapter.

```
postDispatchUpdateUsageCount ::=
/* If there's an active usage count present, update it */
if( objectInfoPtr->usageCount != CRYPT_UNUSED )
    objectInfoPtr->usageCount--;
```

**Figure 22: Decrement object usage count filter**

Another filter, which moves an object into the high state, is shown in Figure 23. This rule should need no further comment.

```
postDispatchChangeState ::=
/* The state change message was successfully processed, the object is now in the high
state */
objectTable[ objectHandle ].flags |= OBJECT_FLAG_HIGH;
```

**Figure 23: Transition object into high state filter**

In practice this filter is used as part of the `PRE_POST_DISPATCH( CheckState, ChangeState )` rule shown in earlier examples.

## 5. Customising the Rule-based Policy

As was mentioned in an earlier section, one of the advantages of the rule-based policy used in cryptlib is that it can be easily adapted to meet a particular set of requirements without requiring the redesign, rebuilding, and revalidation of the entire security kernel upon which the system is based. This section looks at the changes which would be required in order to make cryptlib comply with policies such as the FIPS 140 crypto module security requirements [14]. This task is made relatively easy by the fact that both cryptlib and FIPS 140 represent a commonsense cryptographic security policy containing requirements such as the one that “plaintext keys shall not be accessible from outside the cryptographic module” (section 4.7.5), so that the native cryptlib policy already complies with most of FIPS 140. Other requirements such as the ones that “if a cryptographic module supports concurrent operators then the module shall internally maintain the separation of the roles and services performed by each operator” (section 4.3) and “the output data path shall be logically disconnected from the circuitry and processes while performed key generation, manual key entry, or key zeroization” (section 4.2) are met through the use of the separation kernel. The reason for the disconnection requirement in FIPS 140 is to ensure that there's no chance that the currently active keying material could be interfered with through the arrival of new keying material on shared circuits, the cryptlib kernel actually goes much further than the mere isolation of key handling by isolating all operations which take place.

In addition to the design requirements, several of the FIPS 140 documentation and specification requirements are already addressed through the use of the rule-based policy, for example the “precise specification of the security rules under which a cryptographic module shall operate, including the security rules derived from the requirements of this standard and the additional security rules imposed by the vendor” (appendix C.1) is provided by the kernel filter rules, and the ability to “provide answers to the following questions: what access does operator X, performing service Y while in role Z, have to data item W?” (appendix C.1) is provided by the expert-system nature of the kernel which was discussed in the previous chapter.

The FIPS 140 requirements which remain to be addressed by cryptlib are relatively few and relate to the separation of I/O ports for data and cryptovariables (critical security parameters or CSPs in FIPS-140-speak) and the use of role-based authentication for users. Both of these requirements, which are present at the higher FIPS 140 security levels, are meant for hardware-based crypto modules and aren't addressed in the current cryptlib implementation because it is used almost exclusively in its software-only form. Updating the current implementation to meet the FIPS 140 requirements requires three sets of changes, two fairly simple ones to kernel filter rules and ACLs and one slightly more complex one to the access check performed for object attributes.

The first and simplest change arises from the requirement that “all encrypted secret and private keys entered into or output from the cryptographic module and used in an approved mode of operation shall be encrypted using an approved algorithm” (section 4.7.4). Currently cryptlib allows keys to be loaded in plaintext form since this is what’s usually done in software-only implementations, meeting the above requirement involves changing the key attribute ACLs from `ACCESS_XXX` to `ACCESS_INT_XXX`, which removes the ability to load plaintext keys into the module exactly as required. Because the new ACL is enforced centrally by the kernel, this change immediately takes effect throughout the entire architecture rather than having to be implemented in every location where a key load might take place. This again demonstrates the advantage of having standardised, rule-based controls enforced by a security kernel, since in a more conventional design a single security check omitted from any of the many functions which typically manage key import and export would result in the FIPS 140 requirement not being met (incredibly, one vendor even provides detailed step-by-step instructions complete with sample code telling users how to bypass the security of their cryptographic API and extract plaintext keys [15]).

The second change arises from the requirement that “a cryptographic module shall support the following authorized roles for operators: User role, the role assumed to obtain security services and to perform cryptographic operations or other authorised functions. Crypto officer role, the role assumed to perform a set of cryptographic initialisation or management functions” (section 4.3.1). Again, the use of roles doesn’t make much sense in a software-only implementation where cryptlib is being controlled by a single user who takes all roles, however it can be added fairly easily through a simple modification to the filter rules and ACL structure. In addition to the internal and external access bits, each ACL can be extended to include an indication of whether it applies to the user or crypto officer, for example the encryption key attributes would be marked as being accessible only by the crypto officer while the encrypt/decrypt/sign/verify object usage would be marked as being usable only by the user. Interpreting the extra ACL bits would be handled through a simple change to the filter rule implementation.

The final change, which is specific to hardware implementations, is that “the data input and output physical port(s) used for plaintext cryptographic key components, plaintext authentication data, and other unprotected CSPs shall be physically separated from all other ports of the cryptographic module” (section 4.2). Since this requirement is very specific to the underlying hardware implementation, there’s no general-purpose solution to the problem, although a general approach would be to use the standard filter rule to ensure that CSP-related attributes can only be set through a safe I/O channel or trusted I/O path. An example of this type of mechanism is presented in a later chapter which uses a trusted I/O path with an implementation of cryptlib which is running in embedded cryptographic hardware. Another approach which eliminates most of the problem is to disallow most forms of unprotected CSP load (which the ACL change described earlier has the effect of doing), although some form of I/O channel over which the user or crypto officer can authenticate themselves to the crypto module will still be required.

A set of requirements which predate the FIPS 140 ones is the British Telecom cryptographic equipment security code of practice [16], which suggests measures such as checking for attempts to scan for all legal commands and options (a standard technique for finding interesting things in ISO 7816-4 smart cards), detection of commands issued outside normal operating conditions (for example an attempt to create a contract signature at 3am), and detection of a mismatch in the number of commands submitted vs. the number of commands authorised. cryptlib already performs the last check, the first two can be implemented without too much trouble through the use of filter rules for appropriate commands such as object usage actions in combination with a retry counter and a mechanism for recording the conditions (for example the time of day) under which an action is permitted.

The ease with which cryptlib can be adapted to meet the FIPS 140 and BT code of practice requirements demonstrates the flexibility of the rule-based policy and kernel implementation, which allow the policy change to be handled through a few minor changes in a centralised location which are immediately reflected throughout the entire cryptlib architecture. In contrast a more conventional security kernel with hardcoded policies would require at least a partial kernel redesign, and a conventional crypto toolkit implementation would require a potentially huge number of changes scattered throughout the code, with both accompanying verification and assurance difficulties.

## 6. Miscellaneous Implementation Issues

Making each object thread-safe across multiple operating systems is somewhat tricky. The locking capabilities in cryptlib are implemented as a collection of preprocessor macros which are designed to allow them to be mapped to appropriate OS-specific user- and system-level thread synchronisation and locking functions. Great care has been taken to ensure that this locking mechanism is as fine-grained as possible, with locks typically covering no more than a dozen or so lines of code before they are relinquished, and the code executed while the lock is active being carefully scrutinised to ensure it can never become the cause of a bottleneck, for example by executing a long-running loop while the lock is active. The following sections cover specific implementation details of the object and thread locking.

### 6.1. Object Locking Implementation

The basic macros which are used for object locking are `DECLARE_RESOURCE_LOCKING_VARS` (to declare the variables needed to handle object locking) and `initResourceLock`, `deleteResourceLock`, `lockResource` and `unlockResource` (to create and destroy a lock, and lock and unlock it). The details of the macros depend on the underlying OS.

Under Windows, the locking is handled by critical sections, which aren't really critical sections at all but a form of fast mutex. `DECLARE_RESOURCE_LOCKING_VARS` reserves storage for a `CRITICAL_SECTION` data structure and some extra information used for accounting purposes, and the four macros which manipulate the lock expand to `InitializeCriticalSection`, `DeleteCriticalSection`, `EnterCriticalSection`, and `LeaveCriticalSection` respectively. As was mentioned above, these so-called critical sections aren't really critical sections at all but really fast mutexes which only allow a single thread to enter them at a time. In order to reduce confusion with real critical sections they are referred to here as pseudocritical sections. If a thread enters a pseudocritical section, all other threads continue running normally unless one of them tries to enter the same pseudocritical section, at which point it is suspended until the first thread exits the pseudocritical section.

For the Windows kernel-mode version, the locking variables have somewhat more accurate names and are implemented as a kernel mutex, `KMUTEX`. The four macros to manipulate them expand to `KeInitializeMutex`, nothing (the mutex is never destroyed since the kernel driver always remains resident), `KeWaitForMutexObject`, and `KeReleaseMutex`. Otherwise their behaviour is the same as the user-level pseudocritical sections.

Under Unix, the implementation is somewhat more complex since there are a number of threading implementations available. The most common is the Posix pthreads one, but the mechanism used by cryptlib allows any vaguely similar threading mechanism to be employed. One problem shared by many Unix threading implementations is the fact that mutexes aren't reentrant, which means that trying to lock an object more than once results in deadlock (multiple locking can occur in a number of instances, for example when an object posts a message to itself the cryptlib dispatcher will try to lock the object to send the message). Although there is a mechanism provided in some pthreads implementations to turn off the deadlock problem with:

```
pthread_mutexattr_settype( attr, PTHREAD_MUTEX_RECURSIVE );
```

this isn't very widespread. To fix the problem, cryptlib takes advantage of the existence of a `trylock` function which attempts to lock a mutex but doesn't do so if it's already locked. By combining this with a record of who has currently locked the mutex, we can implement reentrant mutexes as shown in Figure 24. This only performs a blocking lock if the mutex is locked by someone else; if we already own it, it drops through and continues without trying to re-lock it.

```
/* Try and lock the mutex */
if( mutex_trylock( mutex ) == error )
{
    /* The mutex is already locked, if someone else has it locked, we block until it
       becomes available */
    if( thread_self() != mutex_owner )
        mutex_lock( mutex );
}
mutex_owner = thread_self();
```

**Figure 24: Reentrant mutex implementation under Unix**

Using Solaris threads, `DECLARE_RESOURCE_LOCKING_VARS` reserves storage for a `mutex_t` (for the mutex) and a `thread_t` (to record the mutex owner), and the four macros which manipulate the mutex expand to `mutex_init`, `mutex_destroy`, `mutex_lock`, and `mutex_unlock` respectively. DECThreads and LinuxThreads use a `pthread_mutex_t` and `pthread_t`, and the macros expand to `pthread_mutex_init`, `pthread_mutex_destroy`, `pthread_mutex_lock`, and `pthread_mutex_unlock` respectively (these are Posix threads functions which are supported by most Unix versions which support threads). Mach threads use a `mutex_t` and `cthread_t`, and the macros expand to `mutex_init`, `mutex_clear`, `mutex_lock`, and `mutex_unlock` respectively.

Under OS/2 the locking is handled by mutexes in a manner similar to the Unix version, except that there is no need for the calisthenics necessary under Unix since OS/2 mutexes are reentrant. Under OS/2 `DECLARE_RESOURCE_LOCKING_VARS` reserves storage for a `HMTX`, and the four macros which manipulate the mutex expand to `DosCreateMutexSem`, `DosCloseMutexSem`, `DosRequestMutexSem`, and `DosReleaseMutexSem` respectively.

Now that object locking is handled, we need a way to manage the ACL's which tie an object to a thread. This is again built on top of preprocessor macros which map to the appropriate OS-specific data structures and functions. The macros are `DECLARE_OWNERSHIP_VARS` (to declare the variables needed to handle object ownership) and `getCurrentIdentity` (to return the identity of the current thread). If the ownership variable is set to the predefined constant `CRYPT_ERROR` (a value equivalent to the floating-point NaN constant) then the object is not owned by any particular thread.

Under Windows, `DECLARE_OWNERSHIP_VARS` reserves storage for a `HANDLE`, and `getCurrentIdentity` expands to `GetCurrentThreadId` (there is a companion function `GetCurrentThread` which returns a process-specific handle, `GetCurrentThreadId` returns a systemwide unique handle). Windows kernel objects are all owned by the kernel, so in this environment the macros expand to nothing.

Under Unix, `DECLARE_OWNERSHIP_VARS` reserves storage for a `thread_t`, `pthread_t`, or `cthread_t` and `getCurrentIdentity` expands to `thr_self`, `pthread_self`, or `cthread_self` as appropriate. Under OS/2, `DECLARE_OWNERSHIP_VARS` reserves storage for a `TID` and `getCurrentIdentity` expands to `DosGetThreadID`.

The `getCurrentIdentity` macro is used to check object ownership: if the objects owner is `CRYPT_ERROR` or is the same as `getCurrentIdentity` then the object is accessible. If the object is unowned then setting the owner to `getCurrentIdentity` binds it to the current thread. The object can also be bound to another thread by setting the owner to the given thread ID (provided the objects ACL allows the thread which is trying to set the new owner to do so).

## 7. Performance

There are a number of factors which make an assessment of the overall performance impact of the cryptlib kernel implementation rather difficult. Firstly, the access controls and parameter checking which are performed by the kernel take the place of the parameter checking which is usually performed by functions used in conventional implementations (at least in properly-implemented ones), so that much of the apparent overhead imposed by the kernel would also exist in more conventional implementations.

A second factor which makes the performance impact difficult to assess is the fact that although the kernel appears to contain mechanisms such as the message queue and message routing code which could add some amount of overhead to each message which is processed, the stunt box eliminates any use of the queue except under very heavy loads, and the message routing for most messages sent to objects only takes one or two compares and a branch, again having almost no overhead.

A final factor which makes performance assessment difficult is the fact that the nature of the cryptlib implementation changes the way in which code is written. Whereas normal code might require a variety of checks around a function call to ensure that everything is as required and to handle special-case conditions by the caller, with cryptlib it's quite safe to fire off a message since the kernel will ensure that no inappropriate outcome arises.

Although the kernel would appear to impose a certain amount of extra overhead on all operations which it manages, its overall effect is probably more or less neutral when compared to a more conventional implementation (for example the kernel greatly simplifies a number of areas such as checks on key usage which would otherwise need to be performed explicitly either by the caller or by the called code). Without rewriting most of cryptlib in a more conventional manner for use in a performance comparison, the best performance assessment which can be made is the one described earlier for Blacker in which users couldn't detect the presence of the security mechanisms (in this case the cryptlib kernel) when they were activated.

## 8. References

- [1] "Evaluation of Security Model Rule Bases", John Page, Jody Heaney, Marc Adkins, and Gary Dolsen, *Proceedings of the 12<sup>th</sup> National Computer Security Conference*, October 1989, p.98.
- [2] "A Generalized Framework for Access Control: An Informal Description", Marshall Abrams, Leonard LaPadula, Kenneth Eggers, and Ingrid Olson, *Proceedings of the 13<sup>th</sup> National Computer Security Conference*, October 1990, p.135.
- [3] "Generalized Framework for Access Control: Towards Prototyping the ORGCON Policy", Marshall Abrams, Jody Heaney, Osborne King, Leonard LaPadula, Manette Lazear, and Ingrid Olson, *Proceedings of the 14<sup>th</sup> National Computer Security Conference*, October 1991, p.257.
- [4] "Mediation and Separation in Contemporary Information Technology Systems", Marshall Abrams, Jody Heaney, and Michael Joyce, *Proceedings of the 15<sup>th</sup> National Computer Security Conference*, October 1992, p.359.
- [5] "Information Retrieval, Transfer and Management for OSI: Access Control Framework", ISO 10181-3, 1993.
- [6] "Providing Policy Control Over Object Operations in a Mach Based System", Spencer Minear, *Proceedings of the 5<sup>th</sup> Usenix Security Symposium*, June 1995, p.141.
- [7] "A Comparison of Methods for Implementing Adaptive Security Policies", Michael Carney and Brian Loe, *Proceedings of the 7<sup>th</sup> Usenix Security Symposium*, January 1998, p.1.
- [8] "Developing and Using a 'Policy Neutral' Access Control Policy", Duane Olawsky, Todd Fine, Edward Schneider, and Ray Spencer, *Proceedings of the 1996 ACM New Security Paradigms Workshop*, September 1996, p.60.
- [9] "The Flask Security Architecture: System Support for Diverse Security Policies", Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Pepreau, *Proceedings of the 8<sup>th</sup> Usenix Security Symposium*, August 1999, p.123.
- [10] "The Privilege Control Table Toolkit: An Implementation of the System Build Approach", Thomas Woodall and Roberta Gotfried, *Proceedings of the 19<sup>th</sup> National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1996, p.389.
- [11] "Meta Objects for Access Control: Extending Capability-Based Security", Thomas Riechmann and Franz Hauck, *Proceedings of the 1997 ACM New Security Paradigms Workshop*, September 1997, p.17.
- [12] "Meta Objects for Access Control: Role-Based Principals", Thomas Riechmann and Jürgen Kleinöder, *Proceedings of the 3<sup>rd</sup> Australasian Conference on Information Security and Privacy (ACISP'98)*, Springer-Verlag Lecture Notes in Computer Science No.1438, July 1998, p.296.
- [13] "Meta Objects for Access Control: A Formal Model for Role-Based Principals", Thomas Riechmann and Franz Hauck, *Proceedings of the 1998 ACM New Security Paradigms Workshop*, September 1998, p.30.
- [14] "Security Requirements for Cryptographic Modules", FIPS PUB 140-2, National Institute of Standards and Technology, 1999.

- [15] “HOWTO: Export/Import Plain Text Session Key Using CryptoAPI”, Microsoft Knowledge Base Article Q228786, 11 January 2000.
- [16] “Cryptographic Equipment Security: A Code of Practice”, Stephen Serpell, *Computers and Security*, **Vol.4, No.1** (March 1985), p.47.