

1 The Software Architecture

1.1 Introduction

Traditional security toolkits have been implemented using a “collection of functions” design in which each encryption capability is wrapped up in its own set of functions. For example there might be a “load a DES key” function, an “encrypt with DES in CBC mode” function, a “decrypt with DES in CFB mode” function, and so on [1][2]. More sophisticated toolkits hide the plethora of algorithm-specific functions under a single set of umbrella interface functions with often complex algorithm-selection criteria, in some cases requiring the setting of up to a dozen parameters to select the mode of operation [3][4][5][6]. Either approach requires that developers tightly couple the application to the underlying encryption implementation, requiring a high degree of cryptographic awareness from developers and forcing each new algorithm and application to be treated as a distinct development. In addition, there is the danger — in fact almost a certainty due to the tricky nature of cryptographic applications and the subtle problems arising from them — that the implementation will be misused by developers who aren’t cryptography experts, when it could be argued that it is the task of the toolkit to protect developers from making these mistakes [7].

Alternative approaches concentrate on providing functionality for a particular type of service such as authentication, integrity, or confidentiality. Some examples of this type of design are the GSS-API [8][9][10], which is session-oriented and is used to control session-style communications with other entities (an example implementation consists of a set of GSS-API wrapper functions for Kerberos), the OSF DCE security API [11], which is based around access control lists and secure RPC, and IBM’s CCA, which provides security services for the financial industry [12]. Further examples include the SESAME API [13], which is based around a Kerberos derivative with various enhancements such as X.509 certificate support, and the COE SS API [14], which provides GSS-API-like functionality using a wrapper for the Netscape SSL API and is intended to be used in the Defence Information Infrastructure (DII) Common Operating Environment (COE).

This type of design typically includes features specific to the required functionality. In the case of the session-oriented interfaces mentioned above this is the security context that contains details of a relationship between peers based on credentials established between the peers. A non-session-based variant is the IDUP-GSS-API [15], which attempts to stretch the GSS-API to cover store-and-forward use (this would typically be used for a service such as email protection). Although these high-level APIs require relatively little cryptographic awareness from developers, the fact that they operate only at a very abstract level makes it difficult to guarantee interoperability across different security services. For example, the

DCE and SESAME security APIs, which act as a programming interface to a single type of security service, work reasonably well in this role, but the GSS-API, which is a generic interface, has seen a continuing proliferation of “management functions” and “support calls” that allow the application developer to dive down into the lower layers of the code in a somewhat haphazard manner [16]. Since individual vendors can use this to extend the functionality in a vendor-specific manner, the end result is that one vendor’s GSS-API implementation can be incompatible with a similar implementation from another vendor.

Both of these approaches represent an outside-in approach that begins with a particular programming interface and then bolts on whatever is required to implement the functionality in the interface. This work presents an alternative inside-out design that first builds a general crypto/security architecture and then wraps a language-independent interface around it to make particular portions of the architecture available to the user. In this case, it is important to distinguish between the architecture and the API used to interface to it. With most approaches the API *is* the architecture, whereas the approach presented in this work concentrates on the internal architecture only. Apart from the very generic APKI [17] and CISS [18][19][20][21] requirements, only CDSA [22][23] appears to provide a general architecture design, and even this is presented at a rather abstract level and defined mostly in terms of the API used to access it.

In contrast to these approaches, the design presented here begins by establishing a software architectural model that is used to encapsulate various types of functionality such as encryption and certificate management. The overall design goals for the architecture, as well as the details of each object class, are presented in this chapter. Since the entire architecture has very stringent security requirements, the object model requires an underlying security kernel capable of supporting it — one that includes a means of mediating access to objects, controlling the way this access is performed (for example, the manner in which object attributes may be manipulated), and ensuring strict isolation of objects (that is, ensuring that one object can’t influence the operation of another object in an uncontrolled manner). The security aspects of the architecture are covered in the following chapters, although there is occasional reference to them earlier where this is unavoidable.

1.2 An Introduction to Software Architecture

The field of software architecture is concerned with the study of large-grained software components, their properties and relationships, and their patterns of combination. By analysing properties shared across different application areas, it’s possible to identify commonalities among them that may be candidates for the application of a generic solution architecture [24][25].

A software architecture can be defined as a collection of components and a description of the interaction and constraints on interaction between these components, typically represented visually as a graph in which the components are the graph nodes and the connections that handle interactions between components are the arcs [26][27]. The connections can take a variety of forms, including procedure calls, event broadcast, pipes, and assorted message-passing mechanisms.

Software architecture descriptions provide a means for system designers to document existing, well-proven design experience and to communicate information about the behaviour of a system to people working with it, to “distil and provide a means to reuse the design knowledge gained by experienced practitioners” [28]. For example, by describing a particular architecture as a pipe-and-filter model (see Section 1.2.1), the designer is communicating the fact that the system is based on stream transformations and that the overall behaviour of the system arises from the composition of the constituent filter components. Although the actual vocabulary used can be informal, it can convey considerable semantic content to the user, removing the need to provide a lengthy and complicated description of the solution [29]. When architecting a system, the designer can rely on knowledge of how systems designed to perform similar tasks have been designed in the past. The resulting architecture is the embodiment of a set of design decisions, each one admitting one set of subsequent possibilities and discarding others in response to various constraints imposed by the problem space, so that a particular software architecture can be viewed as the architect’s response to the operative constraints [30]. The architectural model created by the architect serves to document their vision for the overall software system and provides guidance to others to help them avoid violating the vision if they need to extend and modify the original architecture at a later date. The importance of architectural issues in the design process has been recognised by organisations such as the US DoD, who are starting to require contractors to address architectural considerations as part of the software acquisition process [31].

This section contains an overview of the various software architecture models employed in the cryptlib architecture.

1.2.1 The Pipe-and-Filter Model

The architectural abstraction most familiar to Unix¹ users is the pipe and filter model, in which a component reads a data stream on its input and produces a data stream on its output, typically transforming the data in some manner in the process (another analogy that has been used for this architectural model is that of a multi-phase compiler [32]). This architecture, illustrated in Figure 1.1, has the property that components don’t share any state with other components, and aren’t even aware of the identities of any upstream or downstream neighbours.



Figure 1.1. Pipe-and-filter model.

¹ Unix is or has been at various times a trademark of AT&T Bell Laboratories, Western Electric, Novell, Unix System Laboratories, the X/Open Consortium, the Open Group, the Trilateral Commission, and the Bavarian Illuminati.

Since all components in a pipe-and-filter model are independent, a complete system can be built through the composition of arbitrarily connected individual components, and any of them can be replaced at any time with another component that provides equivalent functionality. In the example in Figure 1.1, `tr` might be replaced with `sed`, or the `sort` component with a more efficient version, without affecting the functioning of the overall architecture.

The flexibility of the pipe-and-filter model has some accompanying disadvantages, however. The “pipe” part of the architecture restricts operations to batch-sequential processing, and the “filter” part restricts operations to those of a transformational nature. Finally, the generic nature of each filter component may add additional work as each one has to parse and interpret its data, leading to a loss in efficiency as well as increased implementation complexity of individual components.

1.2.2 The Object-Oriented Model

This architectural model encapsulates data and the operations performed on it inside an object abstract data type that interacts with other objects through function or method invocations or, at a slightly more abstract level, message passing. In this model, shown in Figure 1.2, each object is responsible for preserving the integrity of its internal representation, and the representation itself is hidden from outsiders.

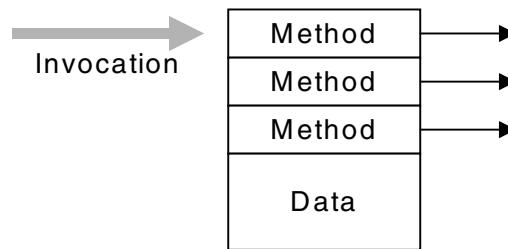


Figure 1.2. Object-oriented model.

Object-oriented systems have a variety of useful properties such as providing data abstraction (providing to the user essential details while hiding inessential ones), information hiding (hiding details that don’t contribute to its essential characteristics such as its internal structure and the implementation of its methods, so that the module is used via its specification rather than its implementation), and so on. Inheritance, often associated with object-oriented models, is an organisational principle that has no direct architectural function [33] and won’t be discussed here.

The most significant disadvantage of an object-oriented model is that each object must be aware of the identity of any other objects with which it wishes to interact, in contrast to the pipe-and-filter model in which each component is logically independent from every other

component. The effect of this is that each object may need to keep track of a number of other objects with which it needs to communicate in order to perform its task, and a change in an object needs to be communicated to all objects that reference it.

1.2.3 The Event-Based Model

An event-based architectural model uses a form of implicit invocation in which components interact through event broadcasts that are processed as appropriate by other components, which either register an interest in a particular event or class of events, or listen in on all events and act on those which apply to the component. An example of an event-based model as employed in a graphical windowing system is shown in Figure 1.3, in which a mouse click event is forwarded to those components for which it is appropriate.

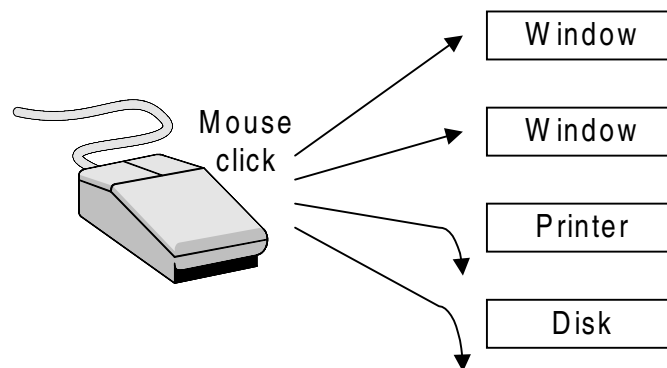


Figure 1.3. Event-based model.

The main feature of this type of architecture is that, unlike the object-oriented model, components don't need to be aware of the identities of other components that will be affected by the events. This advantage over the object-oriented model is, however, also a disadvantage since a component can never really know which other components will react to an event, and in which way they will react. An effect of this, which is seen in the most visible event-based architecture, graphical windowing systems, is the problem of multiple components reacting to the same event in different and conflicting ways under the assumption that they have exclusive rights to the event. This problem leads to the creation of complex processing rules and requirements for events and event handlers, which are often both difficult to implement and work with, and don't quite function as intended.

The problem is further exacerbated by some of the inherent shortcomings of event-based models, which include nondeterministic processing of events (a component has no idea which other components will react to an event, the manner in which they will react, or when they will have finished reacting), and data-handling issues (data too large to be passed around as

part of the event notification must be held in some form of shared repository, leading to problems with resource management if multiple event handlers try to manipulate it).

1.2.4 The Layered Model

The layered architecture model is based on a hierarchy of layers, with each layer providing service to the layer above it and acting as a client to the layer below it. A typical layered system is shown in Figure 1.4. Layered systems support designs based on increasing levels of abstraction, allowing a complex problem to be broken down into a series of simple steps and attacked using top-down or bottom-up design principles. Because each layer (in theory) interacts only with the layers above and below it, changes in one layer affect at most two other layers. As with abstract data types and filters, implementations of one layer can be swapped with different implementations provided they export the same interface to the surrounding layers.

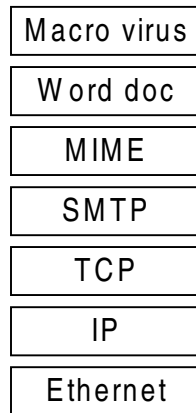


Figure 1.4. Typical seven-layer model.

Unfortunately, decomposition of a system into discrete layers isn't quite this simple, since even if a system can somehow be abstracted into logically separate layers, performance and implementation considerations often necessitate tight coupling between layers, or implementations that span several layers. The ISO reference model (ISORM) provides a good case study of all of the problems that can beset layered architectures [34].

1.2.5 The Repository Model

The repository model is composed of two different components: a central scoreboard-style data structure which represents the current state of the repository, and one or more

components that interact with the scoreboard on behalf of external sources. A typical example of this type of model is a relational database.

1.2.6 The Distributed Process Model

Also known as a client-server architecture, the distributed process model employs a server process that provides services to other, client processes. Clients know the identity of the server (which is typically accessed through local or remote procedure calls), but the server usually doesn't know the identities of the clients in advance. Typical examples include database, mail, and web servers, and significant portions of Microsoft Windows (via COM and DCOM).

1.2.7 The Forwarder-Receiver Model

The forwarder-receiver model provides transparent interprocess communications (typically implemented using TCP/IP or Unix domain sockets, named pipes, or message queues) between peered software systems. The peer may be located on the same machine or on a different machine reached over a network. On the local machine, the forwarder component takes data and control information from the caller, marshals it, and forwards it to the receiver component. The receiver unmarshals it and passes it on to the remote software system, which returns results back to the caller in the same manner. This process is shown in Figure 1.5.

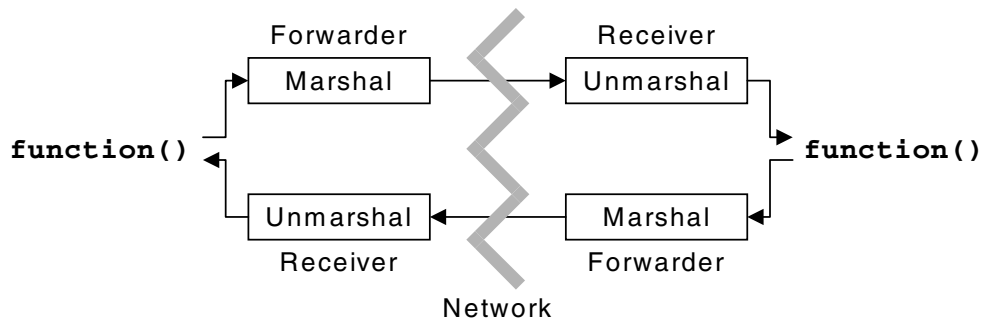


Figure 1.5. Forwarder-and-receiver model.

The forwarder-receiver model provides a means for structuring communications between components in a peer-to-peer fashion, at the expense of some loss in efficiency due to the overhead and delay of the marshalling and interprocess communication.

1.3 Architecture Design Goals

An earlier work [35] gives the design requirements for a general-purpose security service API, including algorithm, application, and cryptomodule independence, safe programming (protection against programmer mistakes), a security perimeter to prevent sensitive data from leaking out into untrusted applications, and legacy support. Most of these requirements are pure API issues and won't be covered in any more detail here. The software architecture presented here is built on the following design principles:

- Independent objects. Each object is responsible for managing its own resource requirements such as memory allocation and use of other required objects, and the interface to other objects is handled in an object-independent manner. For example a signature object would know that it is (usually) associated with a hash object, but wouldn't need to know any details of its implementation, such as function names or parameters, in order to communicate with it. In addition, each object has associated with it various security properties such as mandatory and discretionary access control lists (ACLs), most of which are controlled for the object by the architecture's security kernel, and a few object-specific properties that are controlled by the object itself.
- Intelligent objects. The architecture should know what to do with data and control information passed to objects, including the ability to hand it off to other objects where required. For example if a certificate object (which contains only certificate-related attributes but has no inherent encryption or signature capabilities) is asked to verify a signature using the key contained in the certificate, the architecture will hand the task off to the appropriate signature-checking object without the user having to be aware that this is occurring. This leads to a very natural interface in which the user knows that an object will Do The Right Thing with any data or control information sent to it without requiring it to be accessed or used in a particular manner.
- Platform-independent design. The entire architecture should be easily portable to a wide variety of hardware types and operating systems without any significant loss of functionality. A counterexample to this design requirement is CryptoAPI 2.x [36], which is so heavily tied into features of the very newest versions of Win32 that it would be almost impossible to move to other platforms. In contrast, the architecture described here was designed from the outset to be extremely portable and has been implemented on everything from 16-bit microcontrollers with no file system or I/O capabilities to supercomputers, as well as unconventional designs such as multiprocessor Tandem machines and IBM VM/ESA mainframes and AS/400 minicomputers.
- Full isolation of architecture internals from external code. The architecture internals are fully decoupled from access by external code, so that the implementation may reside in its own address space (or even physically separate hardware) without the user being aware of this. The reason for this requirement is that it very clearly defines the boundaries of the architecture's trusted computing base (TCB), allowing the architecture to be defined and analysed in terms of traditional computer security models.
- Layered design. The architecture represents a true object-based multilayer design, with each layer of functionality being built on its predecessor. The purpose of each layer is to

provide certain services to the layer above it, shielding that layer from the details of how the service is actually implemented. Between each layer is an interface that allows data and control information to pass across layers in a controlled manner. In this way each layer provides a set of well-defined and understood functions that both minimise the amount of information that flows from one layer to another and make it easy to replace the implementation of one layer with a completely different one (for example, migrating a software implementation into secure hardware), because all that a new layer implementation requires is that it offer the same service interface as the one it replaces.

In addition to the layer-based separation, the architecture separates individual objects within the layer into discrete, self-contained objects that are independent of other objects both within their layer and in other layers. For example, in the lowest layer, the basic objects typically represent an instantiation of a single encryption, digital signature, key exchange, hash, or MAC algorithm. Each object can represent a software implementation, a hardware implementation, a hybrid of the two, or some other implementation.

These principles cover the software side of the architecture. Accompanying this are a set of security mechanisms, which are addressed in the next chapter.

1.4 The Object Model

The architecture implements two types of objects, container objects and action objects. A container object is an object that contains one or more items such as data, keys, certificates, security state information, and security attributes. The container types can be broken down roughly into three types: data containers (referred to as envelope or session objects), key and certificate containers (keyset objects), and security attribute containers (certificate objects). An action object is an object that is used to perform an action such as encrypting, hashing, or signing data (referred to using the generic label of encryption action object, which is very similar to the GCS-API concept of a cryptographic context [4]). In addition to these standard object types, there is also a device object type that constitutes a meta-object used to work with external encryption devices such as smart cards or Fortezza cards, that may require extra functions such as activation with a user PIN before they can be used. Once they are initialised as required, they can be used like any of the other object types whose functionality they provide. For example, an RSA action object could be created through the device object for a smart card with RSA capabilities, or a certificate object could be stored in a device object for a Fortezza card as if it were a keyset.

Each object is referenced through its handle, a small integer value unrelated to the object itself, which is used to pass control information and data to and from the object. Since each object is referred to through an abstract handle, the interface to the object is a message-based one in which messages are sent to and received from the object. cryptlib's object handles are equivalent to the "unique name" or "object number" portion of the { unique name, type, representation } tuple used in hardware-based object-oriented systems such as the Intel 432 [37] and its derivative BiiN [38], Recursiv [39], and AS/400 [40]. This provides a single systemwide-unique identifier by which all objects can be identified and that can be

mapped to appropriate type and representation information by the system. Figure 1.6 illustrates a DES encryption action object and a certificate attribute container object contained inside the architecture's security perimeter and referenced through their handles. Although the external programming interface can be implemented to look like the traditional "collection of functions" one, this is simply the message-passing interface wrapped up to look like a more traditional functional interface.

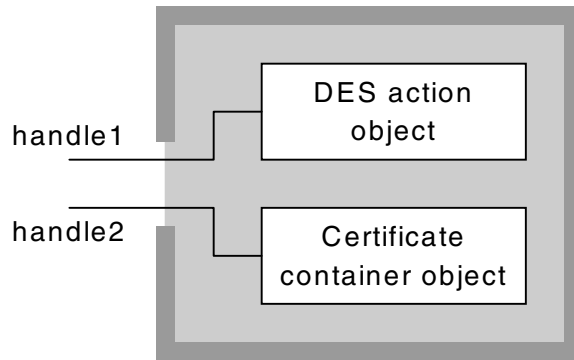


Figure 1.6. Software architecture objects.

A distinction should be made between cryptlib's message passing and the "message passing" that occurs in many object-oriented methodologies. In most widely used object-oriented environments such as C++ and Java, the term "message" is applied to describe a method invocation, which in turn is just a function call in an expensive suit. In cryptlib a message really is a message, with no direct communication or flow-of-control transfer between the source and destination except for the data contained in the message.

1.4.1 User ↔ Object Interaction

All interactions with objects, both those arising from the user and those arising from other objects, are performed indirectly via message passing. All messages sent to objects and all responses to messages are processed through a reference monitor, the cryptlib kernel, which is actually a full Orange Book-style security kernel and is discussed in more detail in the next chapter. The kernel is responsible for access control and security checking, ensuring that messages are routed to appropriate objects, and a range of object and security management functions. The message-passing mechanism connects the objects indirectly, replacing pointers and direct function calls, and is the fundamental mechanism used to implement the complete isolation of architecture internals from the outside world. Figure 1.7 shows a user application interacting with a number of objects via the cryptlib kernel.

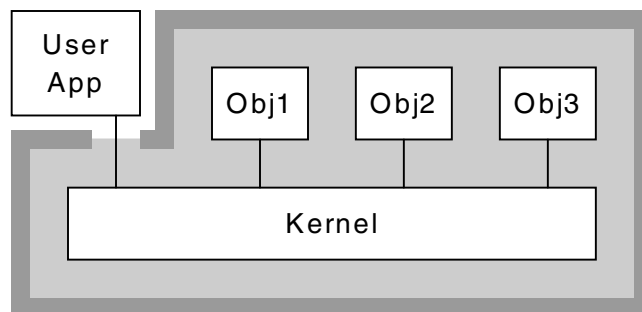


Figure 1.7. Objects accessed via the cryptlib kernel.

When the user calls a cryptlib function, the conventional function call is converted into a message by the cryptlib front-end wrapper code and passed through to the kernel. The kernel performs any necessary checking and processing and passes the message on to the object. Any returned data from the object is handled in the same manner, with the return status and optional data in the returned message being converted back into the function return data. This type of interaction with an object is shown in Figure 1.8, with a user application calling a function (for example, `cryptEncrypt()`), which results in the appropriate message (in this case, `MESSAGE_ENCRYPT`) being sent to the target object and the result being returned to the caller.

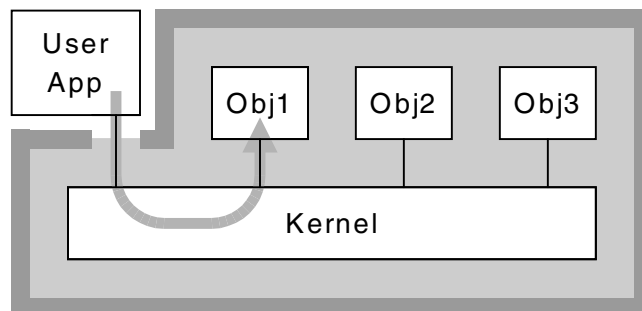


Figure 1.8. User ↔ object interaction via message passing.

Internally, objects communicate with other objects via direct (but still kernel-mediated) message passing, without the veneer of the function-based interface.

Although cryptlib is typically employed as a library linked statically or dynamically into an application, it can also be used in the forwarder-receiver model with the function-based interface acting as a forwarder that passes the messages on to the cryptlib implementation

running as a separate process or even in physically separate hardware. An example of an implementation that uses cryptlib as the control firmware for embedded PC hardware is given in Chapter 7.

1.4.2 Action Objects

Action objects are a fairly straightforward implementation of the object-oriented architectural model and encapsulate the functionality of a security algorithm such as DES or RSA, with the implementation details of a software-based DES action object shown in Figure 1.9. These objects function mainly as building blocks used by the more complex object types. The implementation of each object is completely hidden from the user so that the only way the object can be accessed is by sending information to it across a carefully controlled channel.

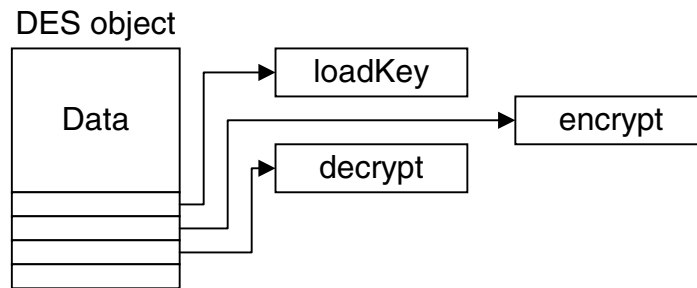


Figure 1.9. Action object internal details.

Action objects are usually attached to other objects such as data or attribute containers, although the existence of the action object is invisible to the user, who sees only the controlling container object. To the user, it appears as though they are using an envelope to encrypt data even though the work is actually being performed by the attached encryption object under the control of the envelope, or using a certificate to verify a signature even though the work is being performed by the attached public-key encryption object. The example given earlier that illustrated a certificate and encryption action object would actually be encountered in the combination shown in Figure 1.10, with the RSA public-key action object performing the encryption or signature-checking work for a controlling certificate object.

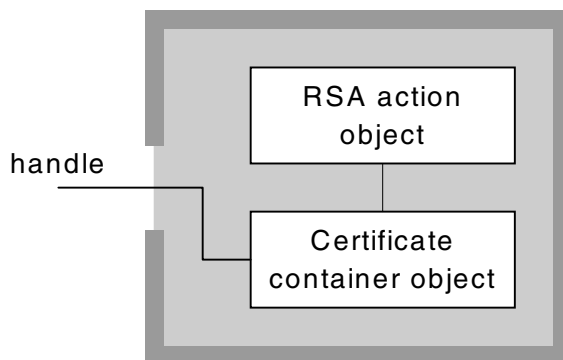


Figure 1.10. Object with dependent object.

This encryption action object can't be directly accessed by the user but can be used in the carefully controlled manner provided by the certificate object. For example, if the certificate object contains an attribute specifying that the attached public-key action object may only be used for digital signature (but not encryption) purposes then any attempt to use the object for encryption purposes would be flagged as an error. These controls are enforced directly by the kernel, as explained in later Chapters 2 and 3.

1.4.3 Data Containers

Data containers (envelope and session objects) act as a form of programmable filter object whose behaviour is modified by the control information that is pushed into it. To use an envelope, the user pushes in control information in the form of container or action objects or general attributes that control the behaviour of the container. Any data that is pushed into the envelope is then modified according to the behaviour established by the control information. For example if a digital signature action object was added to the data container as control information, then data pushed into the container would be digitally signed. If a password attribute was pushed into the container, then data pushed in would be encrypted. Data containers therefore represent the pipe-and-filter model presented in Section 1.2.1. An example of a pipe-and-filter envelope construction that might be used to implement PGP or S/MIME messaging is shown in Figure 1.11 (PGP actually compresses the data after signing rather than before, since the PGP designers felt that it was desirable to sign data directly before any additional processing had been applied [41]).

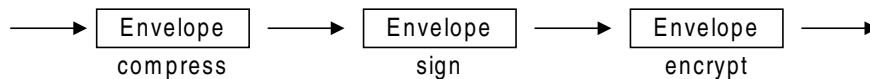


Figure 1.11. Pipe-and-filter construction using envelopes.

Session objects function in a similar manner, but the action object(s) used by the session object are usually established by exchanging information with a peered system, and the session objects can process multiple data items (for example network packets) rather than the single data item processed by envelopes — session objects are envelope objects with state. Session objects act as one-stage filters, with the filter destination being a peered system on a network. In real-world terms, envelope objects are used for functions like S/MIME and PGP, whereas session objects are used for functions such as SSL, TLS, and ssh.

This type of object can be regarded as an intelligent container that knows how to handle data provided to it based on control information that it receives. For example, if the user pushes in a password attribute followed by data, the object knows that the presence of this attribute implies a requirement to encrypt data and will therefore create an encryption action object, turn the password into the appropriate key type for the object (typically through the use of a hash action object), generate an initialisation vector, pad the data out to the cipher block size if necessary, encrypt the data, and return the encrypted result to the user. Session objects function in an almost identical manner except that the other end of the filter is located on a peered system on a network.

Data containers, although appearing relatively simple, are by far the most complex objects present in the architecture.

1.4.4 Key and Certificate Containers

Key and certificate containers (keyset objects) are simple objects that employ the repository architectural model presented in Section 1.2.5 and contain one or more public or private keys or certificates, and may contain additional information such as certificate revocation data (CRLs). To the user, they appear as an (often large) collection of encryption or certificate objects. Two typical container objects of this type are shown in Figure 1.12. Although the diagram implies the presence of huge numbers of objects, these are only instantiated when required by the user. Keyset objects are tied to whatever underlying storage mechanism is used to hold keys, typically PKCS #12 and PKCS #15 files, PGP keyrings, relational databases containing certificates and CRLs, LDAP directories, HTTP links to certificates published on web pages, and crypto tokens such as PKCS #11 devices and Fortezza cards that can act as keysets alongside their usual crypto functionality.

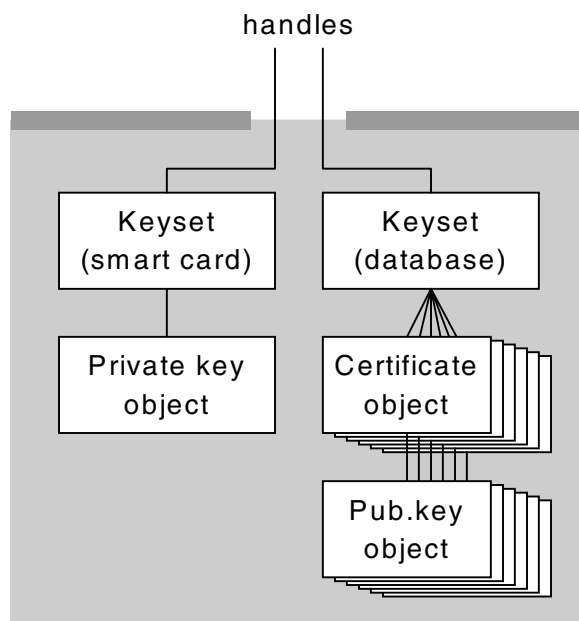


Figure 1.12. Key container objects.

1.4.5 Security Attribute Containers

Security attribute containers (certificate objects), like keyset objects, are built on the repository architectural model and contain a collection of attributes that are attached to a public/private key or to other information. For example signed data often comes with accompanying attributes such as the signing time and information concerning the signer of the data and the conditions under which the signature was generated. The most common type of security attribute container is the public-key certificate, which contains attribute information for a public (and by extension private) key. Other attribute containers are certificate chains (ordered sequences of certificates), certificate revocation lists (CRLs), certification requests, and assorted other certificate-related objects.

1.4.6 The Overall Architectural and Object Model

A representation of some of the software architectural models discussed earlier mapped onto cryptlib's architecture is shown in Figure 1.13. At the upper levels of the layered model (Section 1.2.4) are the envelopes, implementing the pipe-and-filter model (Section 1.2.1) and communicating through the distributed process model (Section 1.2.6). Below the envelopes

are the action objects (one of them implemented through a smart card) that perform the processing of the data in the envelopes.

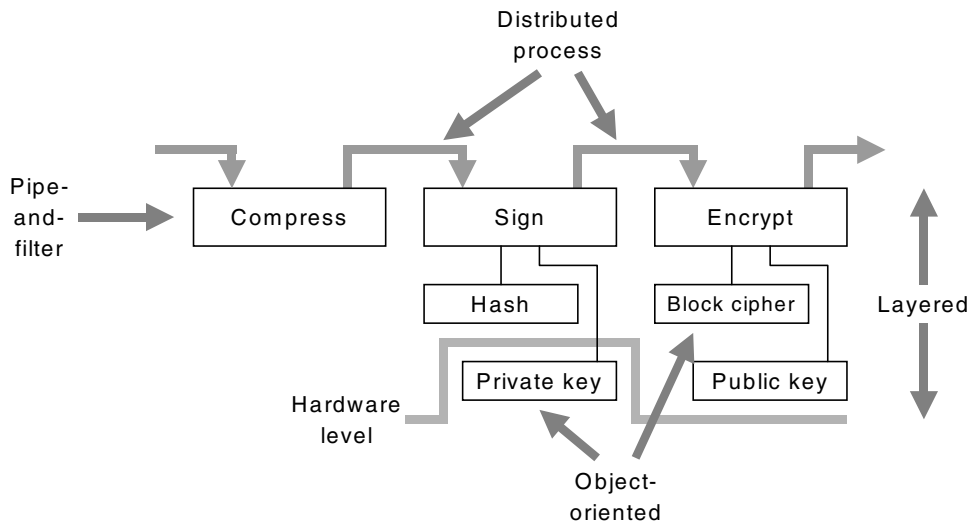


Figure 1.13. Overall software architectural model.

Not shown in this diagram are some of the other architectural models used, which include the event-based model (Section 1.2.3) used for general interobject communications, the repository model (Section 1.2.5) used for the keyset that supplied the public key that is used in the third envelope, and the forwarder-receiver model (Section 1.2.7) which is used to manage communications between cryptlib and the outside world.

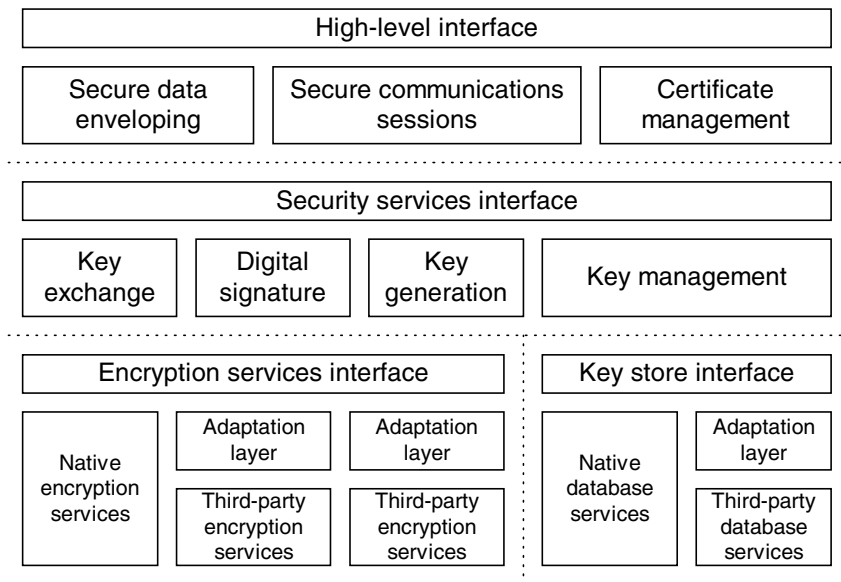


Figure 1.14. Architecture implementation.

Figure 1.13 gave an example of the architecture at a conceptual level, and the actual implementation is shown in Figure 1.14, which illustrates the layering of one level of service over one or more lower-level services.

1.5 Object Internals

Creating or instantiating a new object involves obtaining a new handle, allocating and initialising an internal data structure that stores information on the object, setting security access control lists (ACLs, covered in the next chapter), connecting the object to any underlying hardware or software if necessary (for example, establishing a session with a smart card reader or database backend), and finally returning the object's handle to the user. Although the user sees a single object type that is consistent across all computer systems and implementations, the exact (internal) representation of the object can vary considerably. In the simplest case, an object consists of a thin mapping layer that translates calls from the architecture's internal API to the API used by a hardware implementation. Since encryption action objects, which represent the lowest level in the architecture, have been designed to map directly onto the functionality provided by common hardware crypto accelerators, these can be used directly when appropriate hardware is present in the system.

If the encryption hardware consists of a crypto device with a higher level of functionality or even a general-purpose secure coprocessor rather than just a simple crypto accelerator,

more of the functionality can be offloaded onto the device or secure coprocessor. For example, although a straight crypto accelerator may support functionality equivalent to basic DES and RSA operations on data blocks, a crypto device such as a PKCS #11 token would provide extended functionality including the necessary data formatting and padding operations required to perform secure and portable key exchange and signature operations. More sophisticated secure coprocessors which are effectively scaled-down PCs [42] can take on board architecture functionality at an even higher level. Figure 1.15 shows the levels at which external hardware functionality can be integrated, with the lowest level corresponding to the functionality embodied in an encryption action object and the higher levels corresponding to functionality in envelope, session, and certificate objects. This represents a very flexible use of the layered architectural model in which the hardware implementation level can move up or down the layers as performance and security requirements allow.

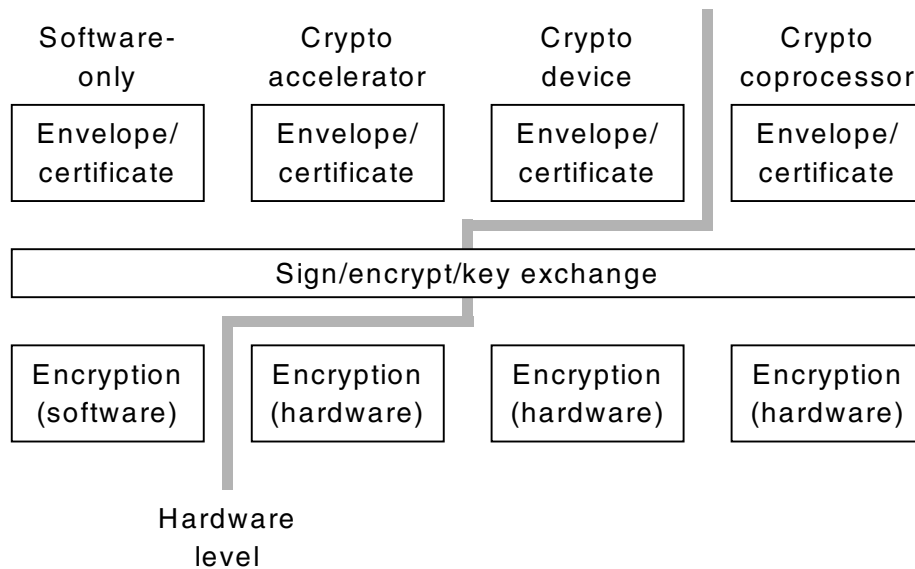


Figure 1.15. Mapping of cryptlib functionality levels to crypto/security hardware.

1.5.1 Object Internal Details

Although each type of object differs considerably in its internal design, they all share a number of common features, which will be covered here. Each object consists of three main parts:

1. State information, stored either in secure or general-purpose memory, depending on its sensitivity.

2. The object's message handler.
3. A set of function pointers for the methods used by the object.

The actual functionality of the object is implemented through the function pointers, which are initialised when the object is instantiated to refer to the appropriate methods for the object. Using an instantiation of a DES encryption action object with an underlying software implementation and an RSA encryption action object with an underlying hardware implementation, we have the encryption object structures shown in Figure 1.16.

When the two objects are created, the DES action object is plugged into the software DES implementation and the RSA action object is plugged into a hardware RSA accelerator. Although the low-level implementations are very different, both are accessed through the same methods, typically `object.loadKey()`, `object.encrypt()`, and `object.decrypt()`. Substituting a different implementation of an encryption algorithm (or adding an entirely new algorithm) requires little more than creating the appropriate interface methods to allow an action object to be plugged into the underlying implementation. As an example of how simple this can be, when the Skipjack algorithm was declassified [43], it took only a few minutes to plug in an implementation of the algorithm. This change provided full support for Skipjack throughout the entire architecture and to all applications that employed the architecture's standard capability query mechanism, which automatically establishes the available capabilities of the architecture on startup.

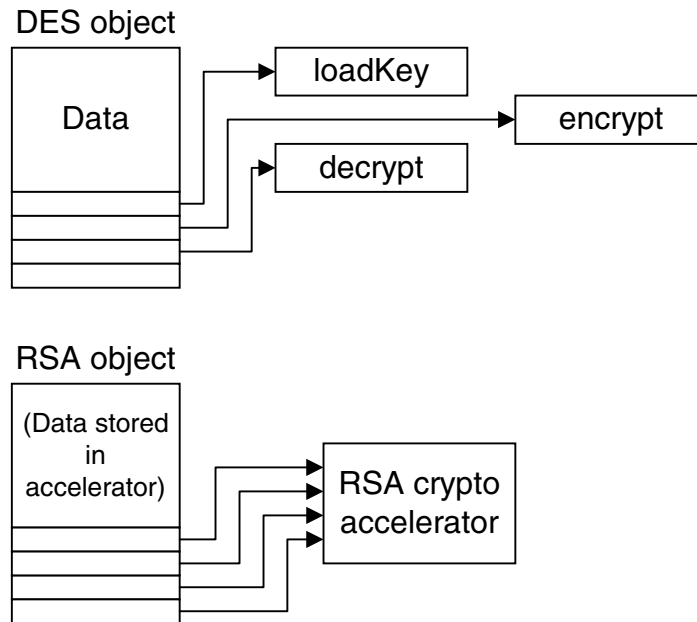


Figure 1.16. Encryption action object internal structure.

Similar implementations are used for the other cryptlib objects. Data containers (envelope and session objects) contain a general data area and a series of method pointers that are set to point to format-specific methods when the object is created. An example of two envelope objects that produce as output S/MIME and PGP messages is shown in Figure 1.17. As with the action objects presented above, changing to a new format involves substitution of different method pointers to code that implements the new format. The same mechanism is used for session objects to implement different protocols such as SSL, TLS, and ssh.

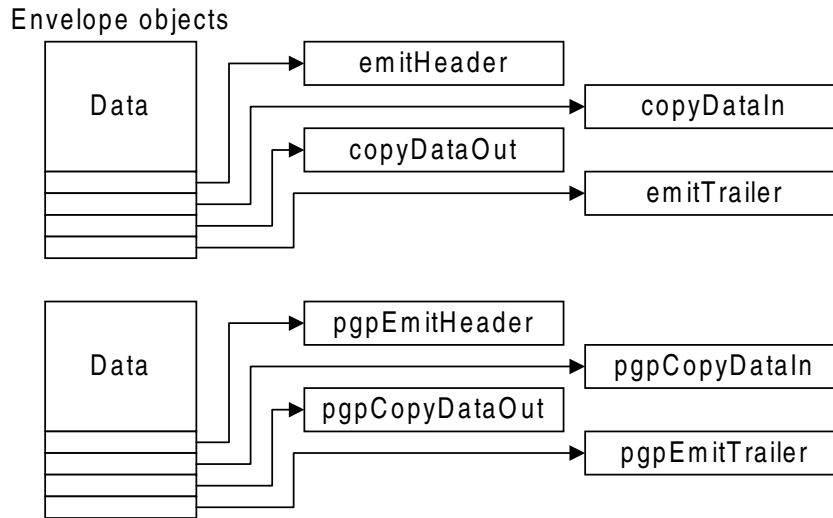


Figure 1.17. Data container object internal structure.

Keyset objects again follow this architectural style, containing method pointers to functions to initialise a keyset, and get, put, and delete keys from the keyset. By switching method pointers, it is possible to switch the underlying data store between HTTP, LDAP, PGP, PKCS #12, PKCS #15, and relational database key stores while providing an identical interface for all keyset types.

1.5.2 Data Formats

Since each object represents an abstract security concept, none of them are tied to a particular underlying data format or type. For example, an envelope could output the result of its processing in the data format used by CMS/S/MIME, PGP, PEM, MSP, or any other format required. As with the other object types, when the envelope object is created, its function pointers are set to encoding or decoding methods that handle the appropriate data formats. In addition to the variable, data-format-specific processing functions, envelope and certificate objects employ data-recognition routines that will automatically determine the format of input

data (for example whether data is in CMS/S/MIME or PGP format, or whether a certificate is a certificate request, certificate, PKCS #7 certificate chain, CRL, OCSP request or response, CRMF/CMP message, or some other type of data) and set up the correct processing methods as appropriate.

1.6 Interobject Communications

Objects communicate internally via a message-passing mechanism, although this is typically hidden from the user by a more conventional functional interface. The message-passing mechanism connects the objects indirectly, replacing pointers and direct function calls, and is the fundamental mechanism used to implement the complete isolation of architecture internals from the outside world. Since the mechanism is anonymous, it reveals nothing about an object's implementation, its interface, or even its existence.

The message-passing mechanism has three parts:

1. The source object
2. The destination object
3. The message dispatcher

In order to send a message from a source to a destination, the source object needs to know the target object's handle, but the target object has no knowledge of where a message came from unless the source explicitly informs it of this. All data communicated between the two is held in the message itself. In addition to general-purpose messages, objects can also send construct and destruct messages to request the creation and destruction of an instantiation of a particular object, although in practice the destroy object message is almost never used, being replaced by a decrement reference count message that allows the kernel to manage object destruction.

In a conventional object-oriented architecture the local client will send a message to the logical server requesting a particular service. The specification of the server acts as a contract between the client and the server, with the client responsible for sending correct messages with the correct contents and the server responsible for checking each message being sent to it, ensuring that the message goes to the correct method or operation, and returning any result data to the client or returning an appropriate error code if the operation could not be performed [44]. In cryptlib's case, the cryptlib kernel acts as a proxy for the logical server, enforcing the required checks on behalf of the destination object. This means that if an object receives a message, it knows that it is of a type that is appropriate for it, that the message contents are within appropriate bounds (for example, that they contain data of a valid length or a reference to a valid object), and that the object is in a state in which processing of the message in the requested manner is an appropriate action.

To handle interobject messaging, the kernel contains a message dispatcher that maintains an internal message queue that is used to forward messages to the appropriate object or objects. Some messages are directed at a particular object (identified by the object's handle), others to an entire class of object or even to all objects. For example, if an encryption action object is instantiated from a smart card and the card is then withdrawn from the reader, the

event handler for the keyset object associated with the reader may broadcast a card-withdrawal message identifying the card that was removed to all active objects, as illustrated in Figure 1.18. In practice this particular event doesn't occur because very few card reader drivers support card-removal notification even if the reader itself does. cryptlib provides a brute-force solution to this problem using a background polling thread, but many readers can't even report a card removal or change properly (one solution to this problem is examined in Section 1.10.2). Other implementations simply don't support card removal handling at all so that, for example, an MSIE SSL session that was established using smart card-based client authentication will remain active until the browser is shut down, even if the smart card has long since been removed.

The mechanism used by cryptlib is an implementation of the event-based architectural model, which is required in order to notify the encryption action object that it may need to take action based on the card withdrawal, and also to notify further objects such as envelope objects and certificates that have been created or acted upon by the encryption action object. Since the sender is completely disconnected from the receiver, it needs to broadcast the message to all objects to ensure that everything that might have an interest is notified. The message handler has been designed so that processing a message of this type has almost zero overhead compared to the complexity of tracking which message might apply to which objects, so it makes more sense to handle the notification as a broadcast rather than maintaining per-object lists of messages in which the object is interested.

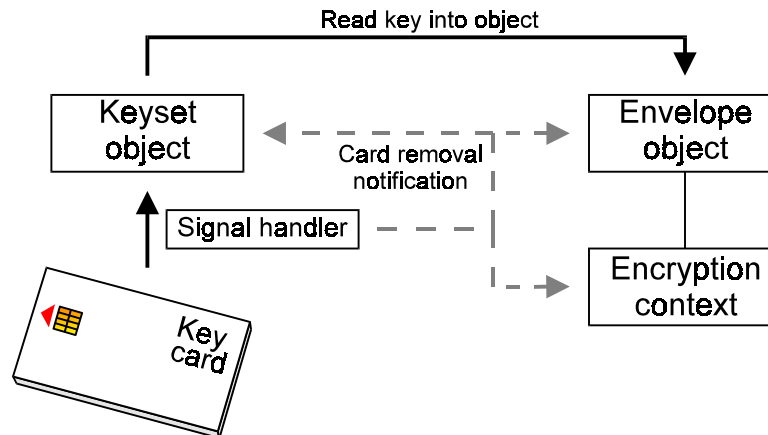


Figure 1.18. Interobject messaging example.

Each object has the ability to intelligently handle external events in a controlled manner, processing them as appropriate. Because an object controls how it handles these events, there is no need for any other object or control routine to know about the internal details or function of the object — it simply posts a notification of an event and goes about its business.

In the case of the card-withdrawal notification illustrated in Figure 1.18, the affected objects that do not choose to ignore it would typically erase any security-related information, close active OS services such as open file handles, free allocated memory, and place themselves in a signalled state in which no further use of the object is possible apart from destroying it. Message queueing and dispatching are handled by the kernel's message dispatcher and the message handlers built into each object, which remove from the user the need to check for various special-case conditions such as smart card withdrawals. In practice, the only object that would process the message is the encryption action object. Other objects that might contain the action object (for example, an envelope or certificate object) will only notice the card withdrawal if they try to use the action object, at which point it will inform them that it has been signalled externally and is no longer usable.

Since the objects act independently, the fact that one object has changed state doesn't affect any of the other objects. This object independence is an important feature since it doesn't tie the functioning of one object to every component object it contains or uses — a smart card-based private key might only be needed to decrypt a session key at the start of a communications session, after which its presence is irrelevant. Since each object manages its own state, the fact that the encryption action object created from the key on the card has become signalled doesn't matter to the object using it after it has recovered the session key.

1.6.1 Message Routing

The kernel is also responsible for message forwarding or routing, in which a message is forwarded to the particular object for which it is appropriate. For example, if an “encrypt data” message is sent to a certificate object, the kernel knows that this type of message is inappropriate for a certificate (which is a security attribute container object) and instead forwards it on to the encryption action object attached to the certificate. This intelligent forwarding is performed entirely within the kernel, so that the end effect is one of sending the message directly to the encryption action object even though, as far as the user was concerned, it was sent to the certificate object.

This forwarding operation is extremely simple and lightweight, taking only a few instructions to perform. Alternative methods are far more complex and require the involvement of each object in the chain of command from the logical target object to the actual target. In the simplest case, the objects themselves would be responsible for the forwarding, so that a message such as a key-size query (which is handled by an encryption action object) to a certificate would proceed as in Figure 1.19. This has the disadvantage of requiring a message to be passed through each object in turn, which has both a high overhead (compared to in-kernel forwarding) and requires that every object in the chain be available to process the message. If one of the objects is otherwise engaged, the message is stalled until the object becomes available to process it. In addition, processing the message ties up every object it passes through, greatly increasing the chances of deadlock when large numbers of objects are unavailable for further work.

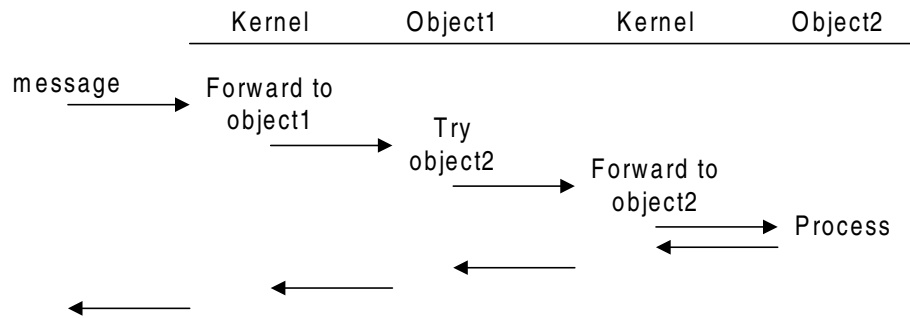


Figure 1.19. Message forwarding by objects.

A slight variation is shown in Figure 1.20, where the object doesn't forward the message itself but instead returns a "Not at this address, try here instead" status to the kernel. This method is slightly better than the previous alternative since it only ties up one object at a time, but it still has the overhead of unnecessarily passing the message through each object.

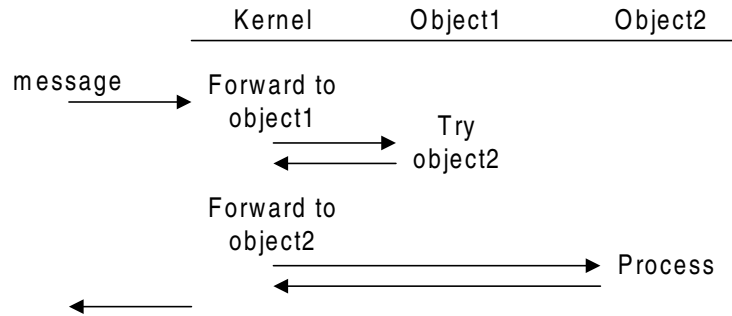


Figure 1.20. Message redirection by objects.

In contrast the in-kernel forwarding scheme shown in Figure 1.21, which is the one actually used, never ties up other objects unnecessarily and has almost zero overhead due to the use of the extremely efficient pointer-chasing algorithm used for the routing.

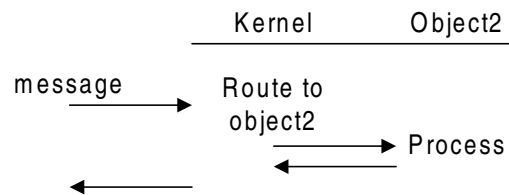


Figure 1.21. Kernel message routing.

1.6.2 Message Routing Implementation

Each message sent towards an object has an implicit target type that is used to route the message to its ultimate destination. For example, a “create signature” message has an implicit target type of “encryption action object”, so if the message were sent to a certificate object, the kernel would route it towards the action object that was associated with the certificate in the manner described earlier. cryptlib’s routing algorithm is shown in Figure 1.22. Although messages are almost always sent directly to their ultimate target, in the cases where they aren’t this algorithm will route them towards their intended target type, either the associated object for most messages or the associated crypto device for messages targeted at devices.

```

/* Route the request through any dependent objects as required until we
   reach the required target object type */
while( object != ε && object.type != target.type )
{
    if( target.type == OBJECT_TYPE_DEVICE )
        object = object.associated device;
    else
        object = object.associated object;
}
  
```

Figure 1.22. Kernel message-routing algorithm.

Eventually the message will either reach its ultimate destination or the associated object or device handle will be empty, indicating that there is no appropriate target object present. This algorithm usually terminates immediately (the message is being sent directly to its intended target) or after a single iteration (the intended target object is directly attached to the initial target). A more formal treatment of the routing algorithm is given in Chapter 5.

Not directly shown in the pseudocode in Figure 1.22 is the fact that the algorithm also includes provisions for messages having alternate targets (in other words `target.type` can be multi-valued). An example of this is a “get key” message that instantiates a public- or private-key object from stored keying data, which is usually sent to a keyset object but may also be intended for a device acting as a keyset. For example, a Fortezza card usually stores

an entire chain of certificates from a trusted root certificate down to that of the card owner, so a “get key” message would be used to read the certificate chain from the card as if it were a keyset object. There can never be a routing conflict for messages with alternate targets because either the main or the alternate target(s), but never more than one, can be present in any sequence of connected objects.

One potential problem that can occur when routing messages between objects is the so-called yo-yo problem, in which a message wanders up and down various object hierarchies until an appropriate target is found [45]. Since the longest object chain that can occur has a length of three (a high-level object such as a data or attribute container linked to an encryption action object linked to a device object) and because the algorithm presented above will always either route a message directly to its target or fail immediately if no target exists, the yo-yo problem can’t occur.

In addition to the routable messages, there are also unroutable messages that must be sent directly to their intended targets. For example a “destroy object” message should never be routed to a target other than the one to which it is directly addressed. Other, similar messages that fall into the class of object control messages (that is, messages which are handled directly by the kernel and are never passed on to the object, an example being the increment reference count message shown in Figure 1.31) are never routed either.

1.6.3 Alternative Routing Strategies

The standard means of handling packet-switched messages is to route them individually, which has a fixed per-message overhead and may lead to blocking problems if multiple messages are being routed over a shared channel, in this case the cryptlib kernel. An alternative routing technique, wormhole routing, groups similar messages into a collection of flits, the smallest units into which messages can be decomposed, with the first flit containing routing information and the remaining flits containing data. In this way the routing overhead only applies to the header flit, and all of the other flits get a free ride in the slipstream [46][47]. By creating a virtual channel from source to destination, the routing overhead for n messages intended for the same target is reduced from n to 1. This is particularly critical in high-speed networks such as those used in multiprocessor/multicomputer systems, where switching overhead has a considerable impact on message throughput [48].

Unfortunately, such a simple solution doesn’t work for the cryptlib kernel. Whereas standard packet switching is only concerned with getting a message from source to destination as quickly as possible, the cryptlib kernel must also apply extensive security checks (covered in the next chapter) to each message, and the outcome of processing one message can affect the processing of subsequent messages. Consider the effects of processing the messages shown in Figure 1.23. In this message sequence, there are several dependencies: The encryption mode must be set before the IV can be set (ECB mode has no IV, so if a mode that requires an IV isn’t selected, the attempt to set an IV will fail), the mode can’t be set after the key has been loaded (the kernel switches the object to the key-loaded state, which disables most further operations on it), and the object can only be used for encryption once the previous three attributes have been set.

Message	Attribute	Value
set attribute	Encryption mode	CBC
set attribute	IV	27FA170D
set attribute	Key	0F37EB2C
encrypt	—	“Secret message”

Figure 1.23. Message sequence with dependencies.

Because of these dependencies, the kernel can’t arrange the messages into a sequence of flits and wormhole-route them to the destination as a single block of messages because each message affects the destination in a manner that also affects the processing of further messages. For example if a sequence of two consecutive messages { *set attribute*, *key*, *value* } were wormhole-routed to an object, the second key would overwrite the first since the kernel would only transition the object into the key-loaded state once processing of the second message had completed. In contrast in the normal routing situation the second key load would fail since the object would already be in the key-loaded state from the first key load. The use of wormhole routing would therefore void the contract between the kernel and the cryptlib objects.

If full wormhole routing isn’t possible, is it possible to employ some form of partial wormhole routing, for example by caching the destination of the previous message? It turns out that, due to the design of the cryptlib object dependency hierarchy, the routes are so short (typically zero hops, more rarely a single hop) that the overhead of performing the caching is significantly higher than simply routing each message through. In addition, the complexity of the route caching code is vastly greater than the direct pointer-chasing used to perform the routing, creating the risk of misrouted messages due to implementation bugs, again voiding the contract between the kernel and cryptlib’s objects. For these reasons, cryptlib individually routes each message and doesn’t attempt to use techniques such as wormhole routing.

1.7 The Message Dispatcher

The message dispatcher maintains a queue of all pending messages due to be sent to target objects, which are dispatched in order of arrival. If an object isn’t busy processing an existing message, a new message intended for it is immediately dispatched to it without being enqueued, which prevents the single message queue from becoming a bottleneck. For group messages (messages sent to all objects of a given type) or broadcast messages (messages sent to all objects), the message is sent to every applicable object in turn.

Recursive messages (ones that result in further messages being generated and sent to the source object) are handled by having the dispatcher enqueue messages intended for an object that is already processing a message or that has a message present in the queue and return immediately to the caller. This ensures that the new message isn’t processed until the earlier message(s) for the object have been processed. If the message is for a different object, it is

either processed immediately if the object isn't already processing a message or it is prepended to the queue and processed before other messages, so that messages sent by objects to associated subordinate objects are processed before messages for the objects themselves. An object won't have a new message dispatched to it until the current one has been processed. This processing order ensures that messages to the same object are processed in the order sent, and messages to different objects arising from the message to the original object are processed before the message for the original object is completed.

The dispatcher distinguishes between two message types: one-shot messages (which inform an object that an event has occurred; for example, a destroy object message), and repeatable messages (which modify an object in a certain way; for example, a message to increment an object's reference count). The main distinction between the two is that duplicate one-shot messages can be deleted whereas duplicate repeatable messages can't. Figure 1.24 shows the message processing algorithm.

```

/* Don't enqueue one-shot messages a second time */
if( message is one-shot and already present in queue )
    return;

/* Dispatch further messages to an object later */
if( message to this object is already present in queue )
{
    insert message at existing queue position + 1;
    return;
}

/* Insert the message for this object and dispatch all messages for this
   object */
insert message at queue start;
while( queue nonempty && message at queue start is for current object )
{
    call the object's message handler with the message data;
    dequeue the message;
}

```

Figure 1.24. Message-dispatching algorithm.

Since an earlier message can result in an object being destroyed, the dispatcher also checks to see whether the object still exists in an active state. If not, it dequeues all further messages without calling the object's message handler.

The operation of the dispatcher is best illustrated with an example. Assume that we have three objects A, B, and C and that something sends a message to A, which results in a message from A to B, which in turn results in B sending in a second message to A, a second message to B, and a message to C. The processing order is shown in Figure 1.25. This processing order ensures that the current object can queue a series of events for processing and guarantee execution in the order in which the events are posted.

Source	Action	Action by Kernel	Queue
User	Send message to A	Enqueue A ₁ Call A's handler	A ₁
A	Send message to B	Enqueue B ₁ Call B's handler	B ₁ , A ₁
B	Send message to A	Enqueue A ₂	B ₁ , A ₁ , A ₂
B	Send message to B	Enqueue B ₂	B ₁ , B ₂ , A ₁ , A ₂
B	Send message to C	Enqueue C Call C's handler	C, B ₁ , B ₂ , A ₁ , A ₂
C	Processing completes	Dequeue C	B ₁ , B ₂ , A ₁ , A ₂
B	Processing completes	Dequeue B ₁ Call B's handler	B ₂ , A ₁ , A ₂
B	Processing completes	Dequeue B ₂	A ₁ , A ₂
A	Processing completes	Dequeue A ₁ Call A's handler	A ₂
A	Processing completes	Dequeue A ₂	

Figure 1.25. Complex message-queueing example.

An examination of the algorithm in Figure 1.24 will reveal that the head of the queue has the potential to become a serious hot spot since, as with stack-based CPU architectures, the top element is continually being enqueued and dequeued. In order to reduce the hot spot problem, the message dispatcher implements a stunt box [49] that allows messages targeted at objects that aren't already processing a message (which by extension means that they also don't have any messages enqueued for them) to be dispatched immediately without having to go through the no-op step of being enqueued and then immediately dequeued. Once an object is processing a message, further messages to it are enqueued as described earlier. Because of the order of the message processing, this simple shortcut is equivalent to the full queue-based algorithm without the overhead of involving the queue.

In practice, almost no messages are ever enqueued, the few that are being recursive messages, although under high-load conditions with all objects occupied in processing messages the queue could see more utilisation. In order to guard against the problems that arise in message queue implementations when the queue is filled more quickly than it can be emptied (the most publicly visible sign of which is the "This Windows application is not responding to messages" dialog), once more than a given number of messages are enqueued no further messages except control messages (those that are processed directly by the kernel, such as ones to destroy an object) are accepted. This means that one or more objects that are stalled processing a message can't poison the queue or cause deadlock problems. At worst the object handle will be unavailable for further use, with the object marked as unavailable by the kernel, but no other objects (and certainly not the kernel itself) will be affected.

1.7.1 Asynchronous versus Synchronous Message Dispatching

When processing messages, the dispatcher can handle them in one of two ways, either asynchronously, returning control to the caller immediately while processing the object in a separate thread, or synchronously, suspending the caller while the message is processed. Asynchronous message channels can require potentially unbounded capacity since the sending object isn't blocked, whereas synchronous channels are somewhat more structured since communication and synchronisation are tightly coupled so that operations in the sending object are suspended until the receiving object has finished processing the message [50]. An example of a synchronous message is shown in Figure 1.26.

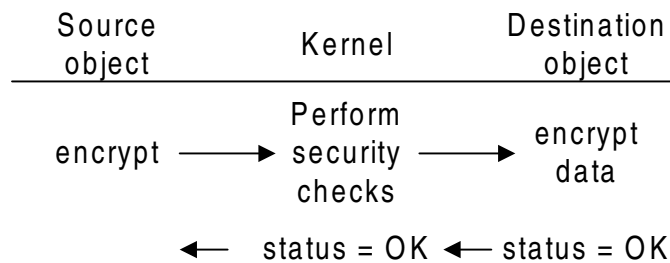


Figure 1.26. Synchronous message processing.

There are two types of messages that can be sent to an object: simple notifications and data communications that are processed immediately, and more complex, generally object-specific messages that can take some time to process, an example being “generate a key”, which can take a while for many public-key algorithms. This would in theory require both synchronous and asynchronous message dispatching. However, this greatly increases the difficulty involved in verifying the kernel, so the cryptlib architecture makes each object responsible for its own handling of asynchronous processing. In practice, this means that (on systems that support it) the object has one or more threads attached to it which perform asynchronous processing. On the few remaining non-threaded systems, or if there is concern over the security implications of using multiple threads, there's no choice but to use synchronous messaging.

When a source object sends a message to a destination that may take some time to generate a result, the destination object initiates asynchronous processing and returns its status to the caller. If the asynchronous processing was initiated successfully, the kernel sets the status of the object to “busy” and enqueues any normal messages sent to it for as long as the object is in the busy state (with the aforementioned protection against excessive numbers of messages building up). Once the object leaves the busy state (either by completing the asynchronous operation or by receiving a control message from the kernel), the remaining enqueued messages are dispatched to it for processing, as shown in Figure 1.27. In this way, the kernel enforces strict serialisation of all messages sent to an object, guaranteeing a fixed order of execution even for asynchronous operations on an object. Since the objects are

inherently thread-safe, the messaging mechanism is also safe when asynchronous processing is taking place.

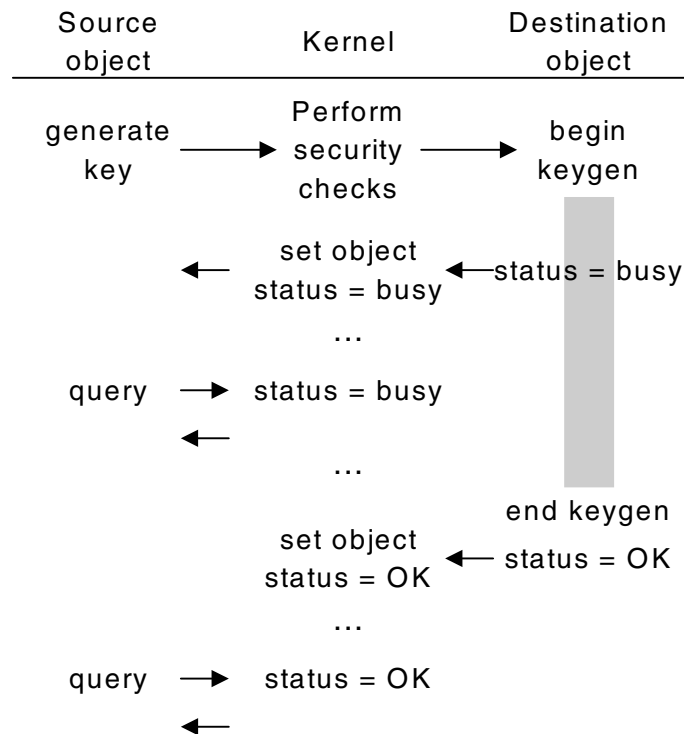


Figure 1.27. Asynchronous message processing.

1.8 Object Reuse

Since object handles are detached from the objects with which they are associated, a single object can (provided its ACLs allow this) be used by multiple processes or threads at once. This flexibility is particularly important with objects used in connection with container objects, since replicating every object pushed into a container creates both unnecessary overhead and increases the chances of compromise of sensitive information if keys and other data are copied across to each newly created object.

Instead of copying each object whenever it is reused, the architecture maintains a reference count for it and only copies it when necessary. In practice the copying is only needed for bulk data encryption action objects that employ a copy-on-write mechanism to

ensure that the object isn't replicated unnecessarily. Other objects that cannot easily be replicated, or that do not need to be replicated, have their reference count incremented when they are reused and decremented when they are freed. When the object's reference count drops to zero, it is destroyed. The use of garbage collection greatly simplifies the object management process as well as eliminating security holes that arise when sensitive data is left in memory, either because the programmer forgot to add code to overwrite it after use or because the object was never cleared and freed even if zeroisation code was present [51].

The decision to use automatic handling of object cleanup was motivated by the problems inherent in alternative approaches that require explicit, programmer-controlled allocation and de-allocation of resources. These typically suffer from memory leaks (storage is allocated but never freed) and dangling pointer problems (memory is freed from one location while a second reference to it is kept active elsewhere) [52][53][54]. Since the object hierarchy maintained by the kernel is a pure tree (strictly speaking, a forest of trees), the many problems encountered with garbage collectors that work with object hierarchies that contain loops are avoided [55][56].

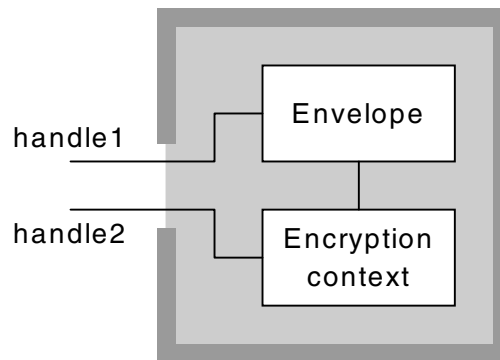


Figure 1.28. Objects with multiple references.

To see how this works, let us assume that the user creates an encryption action object and pushes it into an envelope object. This results in an action object with a reference count of 2, with one external reference (by the user) and one internal reference (by the envelope object), as shown in Figure 1.28. Typically, the user would then destroy the encryption action object while continuing to use the envelope with which it is now associated. The reference with the external access ACL would be destroyed and the reference count decremented by one, leaving the object as shown in Figure 1.29 with a reference count of 1 and an internal access ACL.

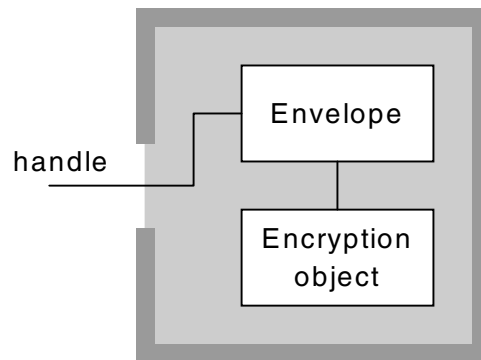


Figure 1.29. Objects with multiple references after the external reference is destroyed.

To the user, the object has indeed been destroyed since it is now accessible only to the envelope object. When the envelope object is destroyed, the encryption action object's reference count is again decremented through a message sent from the envelope, leaving it at zero, whereupon the cryptlib kernel sends it a "destroy object" message to notify it to shut itself down. The only time objects are explicitly destroyed is through an external signal such as a smart card withdrawal or when the kernel broadcasts destroy object messages when it is closing down. At any other time, only their reference count is decremented.

The use of the reference-counting implementation allows objects to be treated in a far more flexible manner than would otherwise be the case. For example, the paradigm of pushing attributes and objects into envelopes (which could otherwise be prohibitively expensive due to the overhead of making a new copy of the object for each envelope) is rendered feasible since in general only a single copy of each object exists. Similarly, a single (heavyweight) connection to a key database can be shared across multiple threads or processes, an important factor in a number of client/server databases where a single client connection can consume a megabyte or more of memory.

Another example of how this object management technique works is provided by the case where a signing key is reused to sign two messages via envelope objects. Initially, the private-key object that is used for the signing operation is created (typically by being read from a private-key file or instantiated via a crypto token such as a smart card) and pushed into both envelopes. At this point, there are three references to it: one internal reference from each envelope and the original external reference that was created when the object was first created. This situation is shown in Figure 1.30.

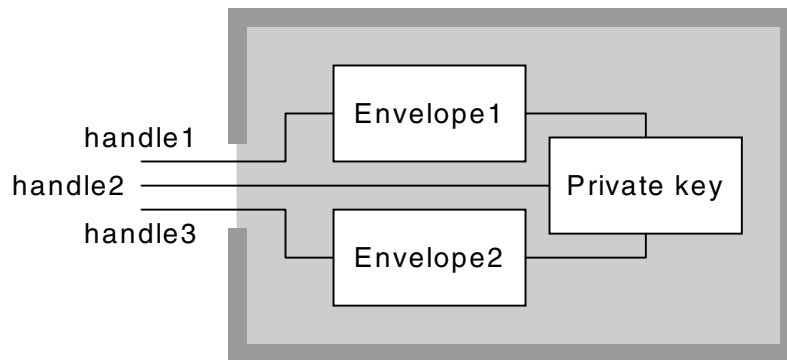


Figure 1.30. Objects with internal and external references.

The user no longer needs the reference to the private-key object and deletes the external reference to it, which decrements its reference count and has the effect that, to the user, the object disappears from view since the external reference to it has been destroyed. Since both envelopes still have references to it, the object remains active although hidden from the outside world.

The user now pushes data through the first envelope, which uses the attached private-key object to generate a signature on the data. Once the data has been signed, the user destroys the envelope, which again decrements the reference count for the attached private-key object, but still leaves it active because of the one remaining reference from the second envelope. Finally, when this envelope's job is also done and it is destroyed by the user, the private-key object's reference count drops to zero and it is destroyed along with the envelope. All of this is performed automatically by the cryptlib kernel without any explicit action required from either the user or the envelope objects.

1.8.1 Object Dependencies

Section 1.4.2 introduced the concept of dependent objects which are associated with other objects, the most common example being a public-key action object that is tied to a certificate object. Dependent objects can be established in one of two ways, the first of which involves taking an existing object and attaching it to another object. An example of where this occurs is when a public-key action object is added to an envelope, which increments the reference count since there is now one reference by the original owner of the action object and a second reference by the envelope.

The second way to establish a dependent object is by creating a completely new object and attaching it to another object. This doesn't increment the reference count since it is only referred to by the controlling object. An example of where this occurs is when a certificate object is instantiated from stored certificate data in a keyset object. This creates two

independent objects, a certificate object and a public-key action object. When the two are combined by attaching the action object to the certificate, the action object's reference count isn't incremented because the only reference to it is from the certificate. In effect, the keyset object that is being used to create the action object and certificate objects is handing over its reference to the action object to the certificate object.

1.9 Object Management Message Flow

We can now combine the information presented in the previous three sections to examine the object management process in terms of interobject message flow. This is illustrated using a variation of the message sequence chart (MSC) format, a standard format for representing protocols in concurrently operating entities such as processes or hardware elements [57][58][59]. A portion of the process involved in signing a message using an envelope is shown in Figure 1.31. This diagram introduces a new object, the system object, which is used to encapsulate the state of a particular instantiation of cryptlib. The system object is the first object created by the kernel and the last object destroyed, and controls actions such as the creation of other objects, random number management, and the access privileges and rights of the currently logged-on user when cryptlib is being used as the control system for a piece of crypto hardware. The system object is the equivalent of the user object present in other message-based object-oriented architectures [60] except that its existence doesn't necessarily correspond to the presence of a logged-in user but instead represents the state of the instantiation of the system as a whole (which may or may not correspond to a logged-in user). In Figure 1.31, messages are sent to the system object to request the creation of a new object (the hash object that is used to hash the data in the envelope) and to request the application of various crypto mechanisms (typically key wrapping and unwrapping or signature creation and verification) to collections of objects, in this case the PKCS #1 signature mechanism applied using the private-key and hash objects.

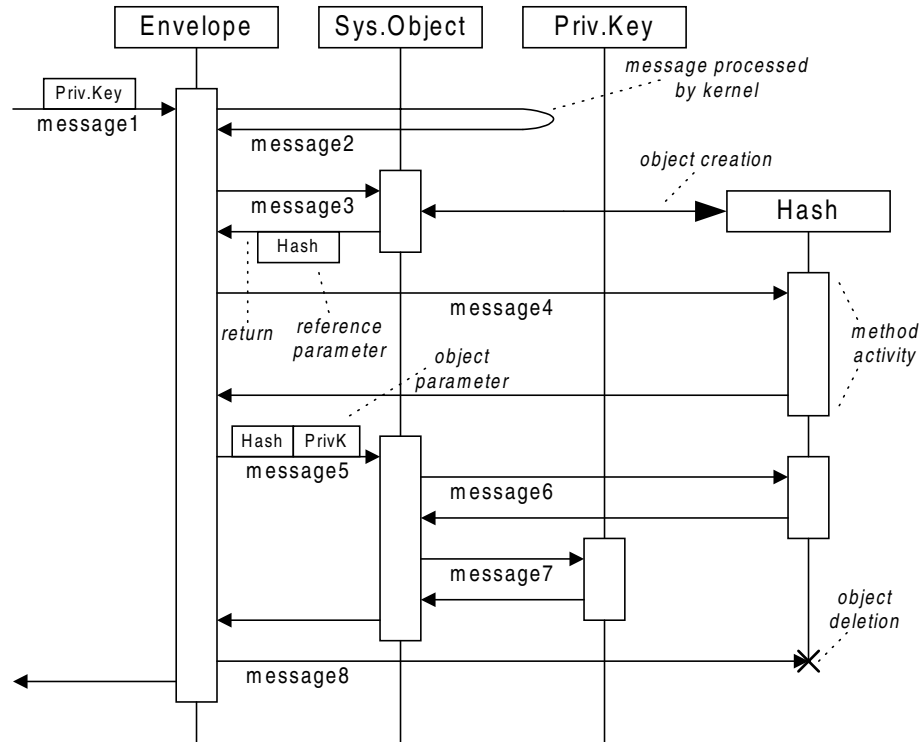


Figure 1.31. Partial data-signing message flow.

With message1, the user adds the private signature key to the envelope, which records its handle and sends it message2, an increment reference count message. This is a control message that is handled directly by the kernel, so the object itself never sees it. The envelope now sends message3 to the system object, requesting the creation of a hash object to hash its data. The system object instantiates a hash object and returns a reference to it to the envelope, which sends it message4, telling it to hash the data contained in the envelope. The private key and hash objects are now ready for signature creation, handled by the envelope sending message5 to the system object, requesting the creation of a PKCS #1 signature using the private-key and hash objects. The system object sends message6 to the hash object to read the hash value and message7 to the private-key object to generate a signature on the hash. Finally, the envelope is done with the hash object and sends it a decrement reference count message, message8, which results in its deletion by the kernel.

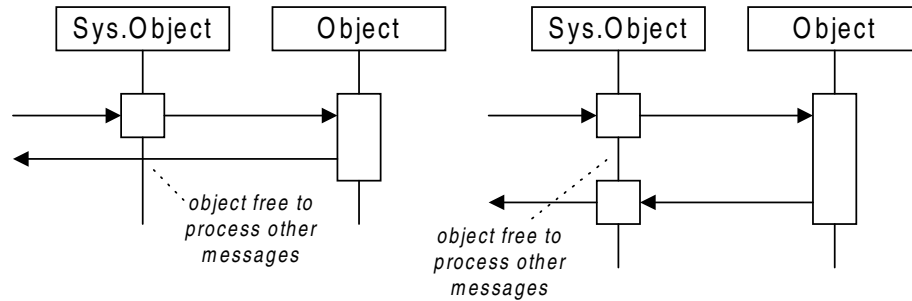


Figure 1.32. System object message processing with direct return (left) and indirect return (right).

Figure 1.31 would appear to indicate that the system object remains busy for the duration of any message processing it performs, but in fact cryptlib's fine-grained internal locking allows the system object to be unlocked while the message processing is performed, ensuring that it doesn't become a bottleneck. The standard MSC format doesn't easily allow this type of operation to be represented. An excerpt from Figure 1.31 that shows the handling of messages by the system object is shown in Figure 1.32. The system object either hands the incoming message over to the appropriate handler which returns directly to the sender (via the kernel), or in more rare cases the return value is passed through the system object on its way back to the kernel/sender. In this way, the system object can never become a bottleneck, which would be particularly troublesome if it remained busy while handling messages that took a long time to process.

The use of such fine-grained locking permeates cryptlib, avoiding the problems associated with traditional kernel locks of which the most notorious was Win16Lock, the Win16 mutex that could stop the entire system if it was acquired but never released by a process. Win16Lock was in fact renamed to Win16Mutex in Windows 95 to give it a less drastically descriptive name [61]. The effect of Win16Mutex was that most processes running on the system (both Win16 and Windows 95, which ended up calling down to 16-bit code eventually) could be stopped by Win16Mutex [62]. Since cryptlib uses very fine-grained locking and never holds a kernel lock over more than a small amount of loop-free code (that is, code that is guaranteed to terminate in a fixed, short time interval), this type of problem cannot occur.

1.10 Other Kernel Mechanisms

In order to work with the objects described thus far, the architecture requires a number of other mechanisms to handle synchronisation, background processing, and the reporting of events within the architecture to the user. These mechanisms are described below.

1.10.1 Semaphores

In the message-passing example given earlier, the source object may want to wait until the data that it requested becomes available. In general, since each object can potentially operate asynchronously, cryptlib requires some form of synchronisation mechanism that allows an object to wait for a certain event before it continues processing. The synchronisation is implemented using lightweight internal semaphores, which are used in most cases (in which no actual waiting is necessary) before falling back to the often heavyweight OS semaphores.

cryptlib provides two types of semaphores: system semaphores (that is, predefined semaphore handles corresponding to fixed resources or operations such as binding to various types of drivers, which takes place on startup) and user semaphores, which are allocated by an object as required. System semaphores have architecture-wide unique handles akin to the stdio library's predefined stdin, stdout, and stderr handles. Before performing an operation with certain types of external software or hardware such as crypto devices and key databases, cryptlib will wait on the appropriate system semaphore to ensure that the device or database has completed its initialisation.

1.10.2 Threads

The independent, asynchronous nature of the objects in the architecture means that, in the worst case, there can be dozens of threads all whirring away inside cryptlib, some of which may be blocked waiting on external events. Since this acts as a drain on system resources, can negatively affect performance (some operating systems can take some time to instantiate a new thread), and adds extra implementation detail for handling each thread, cryptlib provides an internal service thread that can be used by objects to perform basic housekeeping tasks. Each object can register service functions with this thread, which are called in a round-robin fashion, after which the thread goes to sleep for a preset time interval, behaving much like a fiber or lightweight, user-scheduled thread. This means that simple tasks such as basic status checks can be performed by a single architecture-wide thread instead of requiring one thread per object. This service thread also performs general tasks such as touching each allocated memory page that is marked as containing sensitive data whenever it runs in order to reduce the chances of the page being swapped out.

Consider an example of a smart card device object that needs to check the card status every now and then to determine whether the card has been removed from the reader. Most serial-port based readers don't provide any useful notification mechanism, but only report a "card removed" status on the next attempt to access it. Some can't even do that, requiring that the caller track the ID of the card in the reader, with the appearance of a different ID indicating a card change. This isn't terribly useful to cryptlib, which expects to be able to destroy objects that depend on the card as soon as it is removed.

In order to check for card removal, the device object registers a service function with the service thread. The registration returns a unique service ID that can be used later to deregister it. Deregistration can also occur automatically when the object that registered the service function is destroyed.

Once a service function is registered, it is called whenever the service thread runs. In the case of the device object it would query the reader to determine whether the card was still present. If the card is removed, it sends a message to the device object (running in a different thread), after which it returns, and the next service function is processed. In the meantime the device object notifies all dependent objects and destroys itself, in the process deregistering the service function. As with the message processing, since the objects involved are all thread-safe, there are no problems with synchronisation (for example, the service function being called can deregister itself without any problems).

1.10.3 Event Notification

A common method for notifying the user of events is to use one or more callback functions. These functions are registered with a program and are called when certain events occur. Typical implementations use either event-specific callbacks (so the user can register functions only for events in which they are specifically interested) or umbrella callbacks which get all events passed to them, with the user determining whether they want to act on them or not.

Callbacks have two main problems. The first of these is that they are inherently language and often OS-specific, often occurring across process boundaries and always requiring special handling to set up the appropriate stack frames, ensure that arguments are passed in a consistent manner, and so on. Language-specific alternatives to callbacks, such as Visual Basic event handlers, are even more problematic. The second problem with callbacks is that the called user code is given the full privileges of the calling code unless special steps are taken [63]. One possible workaround is to perform callbacks from a special no-privileges thread, but this means that the called code is given too few privileges rather than too many.

A better solution which avoids both the portability and security problems of callbacks is to avoid them altogether in favour of an object polling mechanism. Since all functionality is provided in terms of objects, object status checking is provided automatically by the kernel — if any object has an abnormal status associated with it (for example it might be busy performing a long-running operation such as a key generation), any attempt to use it will result in the status being returned without any action being taken.

Because of the object-based approach that is used for all security functionality, the object status mechanism works transparently across arbitrarily linked objects. For example, if the encryption object in which the key is being generated is pushed into an envelope, any attempt to use it before the key generation has completed will result in an “object busy” status being passed back up to the user. Since it is the encryption object that is busy (rather than the envelope), it is still possible to use the envelope for non-encryption functions while the key generation is occurring in the encryption object.

1.11 References

- [1] libdes, <http://www.cryptsoft.com/ssleay/faq.html>, 1996.

- [2] “Fortezza Cryptologic Programmers Guide”, Version 1.52, National Security Agency Workstation Security Products, National Security Agency, 30 January 1996.
- [3] “BSAFE Library Reference Manual”, Version 4.0, RSA Data Security, 1998.
- [4] “Generic Cryptographic Service API (GCS-API)”, Open Group Preliminary Specification, June 1996.
- [5] “Microsoft CryptoAPI Application Programmers Guide”, Version 1, Microsoft Corporation, 16 August 1996.
- [6] “PKCS #11 Cryptographic Token Interface Standard”, Version 2.10, RSA Laboratories, December 1999.
- [7] “Lessons Learned in Implementing and Deploying Crypto Software”, Peter Gutmann, *Proceedings of the 11th Usenix Security Symposium*, August 2002.
- [8] “Generic Security Service Application Programming Interface”, RFC 2078 (formerly RFC 1508), John Linn, January 1997.
- [9] “Generic Interface to Security Services”, John Linn, *Journal of Computer Communications*, **Vol.17, No.7** (July 1994), p.483.
- [10] “Practical Intranet Security”, Paul Ashley and Mark Vandenwauver, Kluwer Academic Publishers, 1999.
- [11] “DCE Security Programming”, Wei Hu, O’Reilly and Associates, July 1995.
- [12] “Common Cryptographic Architecture Cryptographic Application Programming Interface”, D.Johnson, G.Dolan, M.Kelly, A.Le, and S.Matyas, *IBM Systems Journal*, **Vol.30, No.2** (1991), p.130.
- [13] “SESAME Technology Version 4”, December 1995 (newer versions exist but are no longer publicly available).
- [14] “Security Services Application Programming Interface (SS API) Developer’s Security Guidance”, Amgad Fayad and Don Faatz, MITRE Technical Report MTR 99W0000027, MITRE Corporation, March 2000.
- [15] “Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API)”, RFC 2479, Carlisle Adams, December 1998.
- [16] “Cryptographic APIs”, Dieter Gollman, *Cryptography: Policy and Algorithms*, Springer-Verlag Lecture Notes in Computer Science, No.1029, July 1995, p.290.
- [17] “Architecture for Public-key Infrastructure (APKI), Draft 3”, The Open Group, 27 March 1998.
- [18] “CISS: Generalised Security Libraries”, Sead Muftic and Edina Hatunic, *Computers and Security*, **Vol.11, No.7** (November 1992), p.653.
- [19] “Security Architecture for Open Distributed Systems”, Sead Muftic, Ahmed Patel, Peter Sanders, and Rafael Colon, John Wiley and Sons, 1993.
- [20] “Implementation of the Comprehensive Integrated Security System for computer networks”, Sead Muftic, *Computer Networks and ISDN Systems*, **Vol.25, No.5** (1992), p.469.

- [21] "Practical Intranet Security: Overview of the State of the Art and Available Technologies", Paul Ashley and Mark Vandenwauver, Kluwer Academic Publishing, 1999.
- [22] "Common Data Security Architecture (CDSA) Version 2.0", The Open Group, May 1999.
- [23] "A Comparison of CDSA to Cryptoki", Ruth Taylor, *Proceedings of the 22nd National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1999, CDROM distribution.
- [24] "Domain Models and Software Architectures", Rubén Prieto-Díaz, *ACM SIGSOFT Software Engineering Notes*, **Vol.20, No.3** (July 1995), p.71.
- [25] "Pattern-Oriented Software Architecture: A System of Patterns", Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, John Wiley and Sons, 1996.
- [26] "An Introduction to Software Architecture", David Garlan and Mary Shaw, *Advances in Software Engineering and Knowledge Engineering*, **Vol.1**, 1993.
- [27] "Proceedings of the First International Workshop on Architectures for Software Systems", Seattle, Washington, April 1995.
- [28] "Design Patterns : Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch, Addison-Wesley, 1995.
- [29] "Formulations and Formalisms in Software Architecture", Mary Shaw and David Garlan, *Computer Science Today: Recent Trends and Developments*, Springer-Verlag Lecture Notes in Computer Science, No.1000, 1996, p.307.
- [30] "Succedings of the Second International Software Architecture Workshop (ISAW-2)", Alexander Wolf, *ACM SIGSOFT Software Engineering Notes*, **Vol.22, No.1** (January 1997), p.42
- [31] "Test and Analysis of Software Architectures", Will Tracz, *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA'96)*, ACM, January 1996, p.1.
- [32] "Foundations for the Study of Software Architecture", Dewayne Perry and Alexander Wolf, *ACM SIGSOFT Software Engineering Notes*, **Vol.17, No.4** (October 1992), p.40.
- [33] "Essays on Object-Oriented Software Engineering", Edward Bernard, Simon and Schuster, 1993.
- [34] "The Elements of Networking Style and other Essays and Animadversions on the Art of Intercomputer Networking", Mike Padlipsky, Prentice-Hall, 1985.
- [35] "Security Service API: Cryptographic API Recommendation, Updated and Abridged Edition", NSA Cross Organization CAPI Team, National Security Agency, 25 July 1997.
- [36] "Microsoft Cryptographic Application Programming Interface (CryptoAPI)", Version 2, Microsoft Corporation, 22 December 1998.

- [37] "A programmer's view of the Intel 432 system", Elliott Organick, McGraw-Hill, 1985.
- [38] "An Architecture Supporting Security and Persistent Object Stores", M.Reitenspieß, *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information (Security and Persistence '90)*, Springer-Verlag, 1990, p.202.
- [39] "Rekursiv: Object-Oriented Computer Architecture", David Harland, Ellis Horwood/Halstead Press, 1988.
- [40] "AS/400 Architecture and Application: The Database Machine", Jill Lawrence, QED Publishing Group, 1993.
- [41] "OpenPGP Message Format", Jon Callas, Lutz Donnerhacke, Hal Finney, and Rodney Thayer, RFC 2440, November 1998.
- [42] "Building a High-Performance Programmable, Secure Coprocessor", Sean Smith and Steve Weingart, *Computer Networks and ISDN Systems*, **Vol.31, No.4** (April 1999), p.831.
- [43] "SKIPJACK and KEA Algorithm Specification", Version 2.0, National Security Agency, 29 May 1998.
- [44] "Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach", Donald Firesmith, John Wiley and Sons, 1993.
- [45] "Problems in Object-Oriented Software Reuse", David Taenzer, Murthy Ganti, and Sunil Podar, *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP'89)*, Cambridge University Press, July 1989, p.25.
- [46] "Virtual Cut-Through: A New Computer Communication Switching Technique", Parviz Kermani and Leonard Kleinrock, *Computer Networks*, **Vol.3, No.4** (September 1979), p.267.
- [47] "A Survey of Wormhole Routing Techniques in Direct Networks", Lionel Ni and Philip McKinley, *IEEE Computer*, **Vol.26, No.2** (February 1993), p.62.
- [48] "Wormhole routing techniques for directly connected multicomputer systems", Prasant Mohapatra, *ACM Computing Surveys*, **Vol.30, No.3** (September 1998), p.374.
- [49] "Design of a Computer: The Control Data 6600", J.E.Thornton, Scott, Foresman and Co., 1970.
- [50] "Paradigms for Process Interaction in Distributed Programs", Gregory Andrews, *ACM Computing Surveys* **Vol.23, No.1** (March 1991), p.49.
- [51] "Conducting an Object Reuse Study", David Wichers, *Proceedings of the 13th National Computer Security Conference*, October 1990, p.738.
- [52] "The Art of Computer Programming, Vol.1: Fundamental Algorithms", Donald Knuth, Addison-Wesley, 1998.
- [53] "Garbage collection of linked data structures", Jacques Cohen, *ACM Computing Surveys*, **Vol.13, No.3** (September 1981), p.341.

- [54] “Uniprocessor Garbage Collection”, Paul Wilson, *Proceedings of the International Workshop on Memory Management (IWMM 92)*, Springer-Verlag Lecture Notes in Computer Science, No.637, 1992, p.1.
- [55] “Reference Counting Can Manage the Circular Environments of Mutual Recursion”, Daniel Friedman and David Wise, *Information Processing Letters*, **Vol.8, No.1** (2 January 1979), p.41.
- [56] “Garbage Collection: Algorithms for Automatic Dynamic Memory Management”, Richard Jones and Rafael Lins, John Wiley and Sons, 1996
- [57] “Message Sequence Chart (MSC)”, ITU-T Recommendation Z.120, International Telecommunication Union, March 1993.
- [58] “The Standardization of Message Sequence Charts”, Jens Grabowski, Peter Graubmann, and Ekkart Rudolph, *Proceedings of the IEEE Software Engineering Standards Symposium (SESS’93)*, September 1993.
- [59] “Tutorial on Message Sequence Charts”, Ekkart Rudolph, Peter Graubmann, and Jens Grabowski, *Computer Networks and ISDN Systems*, **Vol.28, No.12** (December 1996), p.1629.
- [60] “Integrating an Object-Oriented Data Model with Multilevel Security”, Sushil Jajodia and Boris Kogan, *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1990, p.76.
- [61] “Inside Windows 95”, Adrian King, Microsoft Press, 1994.
- [62] “Unauthorised Windows 95”, Andrew Schulman, IDG Books, 1994.
- [63] “Java Security Architecture”, JDK 1.2, Sun Microsystems Corporation, 1997.



<http://www.springer.com/978-0-387-95387-8>

Cryptographic Security Architecture

Design and Verification

Gutmann, P.

2004, XVIII, 320 p. 56 illus., Hardcover

ISBN: 978-0-387-95387-8