A Java Beans Component Architecture for Cryptographic Protocols

Pekka Nikander

pekka.nikander@hut.fi
Helsinki University of Technology

Arto Karila

arto.karila@hut.fi
Helsinki University of Technology

Abstract

Global networking has brought with it both new opportunities and new security threats on a worldwide scale. Since the Internet is inherently insecure, secure cryptographic protocols and a public key infrastructure are needed. In this paper we introduce a protocol component architecture that is well suited for the implementation of telecommunications protocols in general and cryptographic protocols in particular. Our implementation framework is based on the Java programming language and the Conduits+ protocol framework. It complies with the Beans architecture and security API of JDK 1.1, allowing its users to implement application specific secure protocols with relative ease. Furthermore, these protocols can be safely downloaded through the Internet and run on virtually any workstation equipped with a Java capable browser[*]*. The framework has been implemented and tested in practice with a variety of cryptographic protocols. The framework is relatively independent of the actual cryptosystems used and relies on the Java 1.1 public key security API. Future work will include Java 1.2 support, and utilization of a graphical Beans editor to further ease the work of the protocol composer.*

1 Introduction

Designing and implementing telecommunications protocols has proven to be a very demanding task. Building secure cryptographic protocols is even harder, because in this case we have to be prepared for not just random errors in the network and end-systems but also premeditated attackers trying to take advantage of any weaknesses in the design or implementation [3] [29]. During the last ten years or so, much attention has been focused on the formal modelling and verification of cryptographic protocols [21] [27]. However, the question how to apply these results to real design and implementation has received considerably less attention [17]. Recent results in the area of formalizing architecture level software composition and integrating it with object oriented modelling and design seem to bridge one section of the gap between the formal theory and everyday practice [2] [16] [31].

In this paper we present a practical architecture and an implementation framework for building secure communications protocols that have the following properties:

- The architecture is made to the needs of today's applications based on the global infrastructure that is already forming (Internet, WWW, Java).
- The implementation framework allows us to construct systems out of our own trusted protocol components and others taken from the network. These systems can be securely executed in a "protocol sand box", where they, for example, cannot leak encryption keys or other secret information.
- Together they allow us to relatively easily implement application specific secure protocols, securely download the protocol software over the Internet and use it without any prior arrangements or software installation.

We have implemented the main parts of the architecture as an object oriented protocol component framework called Java Conduits. It was built using JDK 1.1 and is currently being tested on the Solaris operating system. The framework itself is pure Java and runs on any Java 1.1 compatible virtual machine.

Our goal is to provide a sound practical basis for protocol development, with the desire to create higher level design patterns and architectural styles that could be formally combined with protocol modelling and analysis. The current focus lies in utilizing the "gang of four" object level design patterns [10] to create a highly stylistic way of building both cryptographic and non-cryptographic communications protocols. Our implementation experience has shown that this approach leads to a number of higher level design patterns, i.e., protocol

[*] In order to achieve real sandbox security, either JDK 1.2 or a specially tailored SecurityManager is needed [12].

patterns, that describe how protocols should be composed from lower level components in general.

The rest of this paper is organized as follows. In Section 2 we introduce our architecture and its relationship to existing work. In Section 3 we present the component framework developed. Section 4 dwells into implementational details and experience gained while building prototypes of real protocols. At the end we present a summary (Section 5) and outline some future work (Section 6).

2 The architecture

In our view, the world to which we are building applications consists of the following main components: the *Internet*, the *World Wide Web* (WWW), the *Java programming language and execution environment* and an *initial security context* (based on predefined trusted keys). Our architecture is based on these four corner stones. In addition, there are three more components that are not indispensable but "nice-to-have": a *Public Key Infrastructure* (PKI), the *Internet Security Association and Key Management Protocol* (ISAKMP) and the *Internet Protocol Security Architecture* (IPSEC).

2.1 The essential components

The world-wide Internet has established itself as the dominating network architecture that even the public switched telephone network has to adapt to. The new Internet Protocol IPv6 will solve the main problem of address space, and together with new techniques, such as resource reservation and IP switching, provide support for new types of applications, such as multimedia on a global scale. As we see it, the only significant threats to the Internet are political, not technical or economic. We regard the Internet, as well as the less open extranet and intranet, as an inherently untrusted network.

The World Wide Web (WWW) has been the fastest growing and most widely used application of the Internet. In fact, the WWW is an application platform which is increasingly being used as an user interface to a multitude of applications. Hyper Text Markup Language (HTML) forms and the Common Gateway Interface (CGI) make it possible to create simple applications with the WWW as the user interface. More recently, we have seen the proliferation of executable content.

The Java programming language extends the capabilities of the WWW by allowing us to download executable programs, Java applets, with WWW pages. A Java virtual machine has already become an essential part of a modern web browser and we see the proliferation of Java as being inevitable. We are basing our work on Java and the signed applets security feature of Java 1.1.

In order to communicate securely, we always need to start with an initial security context. In our architecture, the minimal initial security context contains the trusted keys of our web browser, which we can use to check the signatures of the downloaded applets and other Java Beans.

2.2 The optional components

While our architecture does not depend on the existence of the following three components, they are "nice to have", as they will make the architecture more efficient and scalable.

A public key infrastructure (PKI) allows us to associate a public key with a person, company, service, authorization, or such with a reasonable assurance level. It also allows us to prove the authenticity of a digital signature in a court of law. A global PKI is a prerequisite for many new application areas for the Internet. Until recently, most of the work in this area has focused on X.509 type certificates and a hierarchical tree of certification authorities (CAs). While this approach works for some application areas, e.g., in relations between governments, it is not suitable for others, since trust is inherently intransitive. The Simple Public Key Infrastructure (SPKI) [9] appears to us as a more widely applicable PKI.

The Internet Security Association and Key Management Protocol (ISAKMP) [19] provides us with a standard way of securely generating keys and setting up security contexts. We expect a number of application-specific security protocols to be built on top of ISAKMP. The authentication information needed for securing a connection can easily be augmented with capabilities such as authorization information. This allows future access control policies to be based on signed authority in addition to explicit identity.

The Internet Protocol Security Architecture (IPSEC) [6] [30] is an extension to IPv4 and an essential part of IPv6. It provides us with authenticated, integral and confidential channels for transparent exchange of information between any two hosts, users or programs on the Internet. Designed to be used everywhere, it will be implemented on most host and workstation operating systems in the near future. The flexible authentication schemes provided by ISAKMP make it possible to individually secure single TCP connections and UDP packet streams.

IPSEC is not yet ubiquitously available, so, for now, its functionality can be substituted with an transport layer protocol such as SSL. The current JDK architecture does not allow IPSEC to be implemented in Java

without resorting to native interfaces that allow access to the underlying protocol stack or media.

2.3 Implementational requirements

Future protocols will be drastically different from what most of us believed only a few years ago. The role of security cannot be over-emphasized. Unfortunately, most of the tools and frameworks developed so far either tend to ignore security or do not facilitate integrating protocol security with that of the underlying operating system or the supported applications. This is unacceptable, since security should be designed and built in to the protocols and the system as a whole from the very beginning.

The earlier protocol frameworks were typically based on a virtual operating environment that was clearly separated from the underlying operating system. From the modularization point of view this was good. However, this made it hard to build application level programs that were able to use the protocols running within the framework. Java Conduits is clearly different in this respect. For example, under the JavaOS, the protocols and the applications all run within a single virtual environment, making seamless integration straightforward.

The use of an object oriented implementation language allows us to extensively use object-level design patterns. This makes the framework itself more generic and extensible, and creates a highly stylistic way for writing actual protocol implementations. With a suitable object oriented design tool, the outline for the classes needed to implement a new protocol can be created in minutes. The actual implementation code for the protocol actions typically takes a little longer, depending on the complexity of the protocol.

Performance will always be an issue with communications protocols. Even though processing power is constantly increasing, the new applications need ever-increasing bandwidth and reasonable transfer delay. The new protocols require large transfer capacity, short and fixed delay, and lots of cryptography, among other things.

There are two facets to performance. First, the processing power available should be used as efficiently as possible. The importance of this will gradually decrease as processing power increases. Second, and more important, there should not be any design limitations which set a theoretical limit to the performance of the protocols, no matter how much processing power we have. We want to allow as much parallelism as possible and build the protocol implementations such that they can be efficiently divided between a number of processors. Java, with its built-in threads and synchronization, allows parallelism to be utilized with relative ease.

2.4 Related work

Our implementation framework is heavily based on the ideas first presented with the x–Kernel [15] [18] [22] and the Conduits [32] and Conduits+ [14] frameworks. Some of the ideas, especially the microprotocol approach, have also been used in other frameworks, including Isis [8], Horus/Ensemble [24], and Bast [11]. However, Isis and Horus concentrate more on building efficient and reliable multiparty protocols, while Bast objects are larger than ours, yielding a white box oriented framework instead of a black box one.

Compared to x–Kernel, Isis and Horus, our main novelty is in the use and recognition of design patterns at various levels. Furthermore, our object model is more fine-grained. These properties come hand-in-hand — using design patterns tends to lead to collections of smaller, highly regular objects.

The Horus/Ensemble security architecture is based on Kerberos and Fortezza. Instead, we base our architecture on the Internet IPSEC architecture. Kerberos does not scale well and requires a lot of trusted functionality. Fortezza is developed mainly for U.S. Government use, and not expected to be generally available. On the other hand, we expect the IPSEC architecture to be ubiquitously available in the same way as the Domain Name System (DNS) is today.

Most important, our framework is seamlessly integrated into the Java security model. It utilizes both the language level security features (packages, visibility) and the new Java 1.1 security functionality. A further difference is facilitated by the Java run time model. Java supports code and object mobility. This allows application specific protocols to be loaded or used on demand.

Another novelty lies in the way we use the Java Beans architecture. This allows modern component based software tools to be used to compose protocols. The introduction of the Protocol class, or the metaconduit (see Section 3.2), which allows composed subgraphs to be used as components within larger protocols, is especially important. The approach also allows the resulting protocol stacks to be combined with applications.

3 The implementation framework

Java Conduits provides a fine grained object oriented protocol component framework. The supported way of building protocols is very patterned, on several levels. The framework itself utilizes heavily the "gang of four" object design patterns [10]. A number of higher level patterns for constructing individual protocols are emerging. At the highest level, we envision a number of architectural patterns to surface as users will be able to

construct protocol stacks that are matched to application needs.

Our goal is to allow application-specific secure protocols to be built from components. The protocols themselves can be constructed from lower level components, called conduits. The protocol components, in turn, can be combined into complete protocol stacks. To achieve this, we have to solve a number of generic problems faced by component based software.

3.1 Component based software engineering

Recently, attention has shifted from basic object oriented (OO) paradigms and object oriented frameworks towards combining the benefits of OO design and programming with the broad scale architectural viewpoints [2] [20]. Component based software architectures and programming environments play a crucial role in this trend.

For a long time, it was assumed that object oriented programming alone would lead to software reusability. However, experience has shown this assumption false [20]. On the other hand, non object oriented software architectures, such as Microsoft OLE/COM and IBM/Apple OpenDoc, have shown modest success in creating real markets for reusable software components. Early industry response seems to indicate that the Java Beans architecture may prove more successful.

The Java Beans component model we are using defines the basic facets of component based software to be *components*, *containers* and *scripting*. That is, component based software consists of component objects that can be combined into larger components using containers. The interaction between the components can be controlled by scripts that should be easy to produce, allowing less sophisticated programmers and users to create them. This is achieved through *runtime interface discovery*, *event handling*, *object persistence*, and *application builder support*. [33]

Java as a language provides natural support for run–time interface discovery. A binary Java class file contains explicit information about the names, visibility and signatures of the class and its fields and methods. Originally provided to enable late loading and to ease the fragile superclass problem, the runtime environment also offers this information for other purposes, e.g., to application builders. Java 1.1 provides a reflection API as a standard facility, allowing any authorized class to dynamically find out and access the class information.

The Java Beans architecture introduced a new event model for Java 1.1. The model consists of *event listeners*, *event objects* and *event sources*. The mechanism is very lean, allowing basically any object to act as a event source, event listener, or even the event itself. Most of this is achieved through class and method naming conventions, with some extra support through manifestational interfaces.

Compared to other established component software architectures, i.e., OLE/COM, CORBA and OpenDoc, the Java Beans architecture is relatively light-weight. Under Java 1.1, nearly any object can be turned into a Java Bean. If an object's class supports serialization* and the object does not contain any references to its environment, the object can be considered to be a Bean without any changes at all. When Bean properties are provided by naming access functions appropriately, event support added with a few lines of code, and any references to the enclosing environment marked transient, almost any class can be easily turned into a Bean.

On the other hand, the Java Beans architecture, as it is currently defined, does not address some of the biggest problems of component based software architectures any better than its competitors. These include the mixing and matching problem that faces anyone trying to build larger units from the components. Basically, each component supports a number of interfaces. However, the semantics of these interfaces are often not immediately apparent, nor can they be formally specified within the component framework. When the components are specifically designed to co-operate, this is not a problem. However, if the user tries to combine components from different sources, the interfaces must be adapted. This may, in turn, yield constructs that cannot stand but collapse due to semantic mismatches.

In the protocol world, the mixing and matching problem is reflected in two distinct ways. First, the data transfer semantics differ. Second, and more importantly, the information content needed to address the intended recipient(s) of a message greatly differ. In our framework, the recipient information is always implicitly available in the topology of the conduit graph. Thus, the protocols have no need to explicitly address peers once an appropriate conduit stream has been created.

It has been shown that secure cryptographic protocols, when combined, may result in insecure protocols [13]. This problem cannot be easily addressed within the current Java Beans architecture. We hope that future research, paying more attention to the formal semantics, will alleviate this problem.

* A Java class supports serialization by manifesting implementation of the `java.lang.Serializable` interface. Most Java classes can do this. However, there are classes that are inherently impossible to be serialized as such, e.g., `java.lang.Thread`.

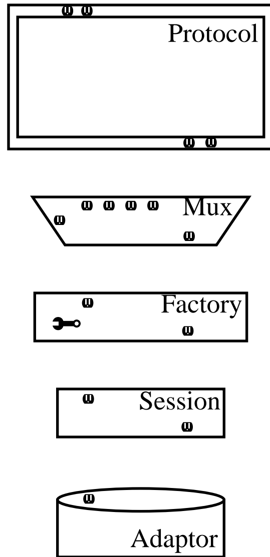

Figure 1: The five types of conduits

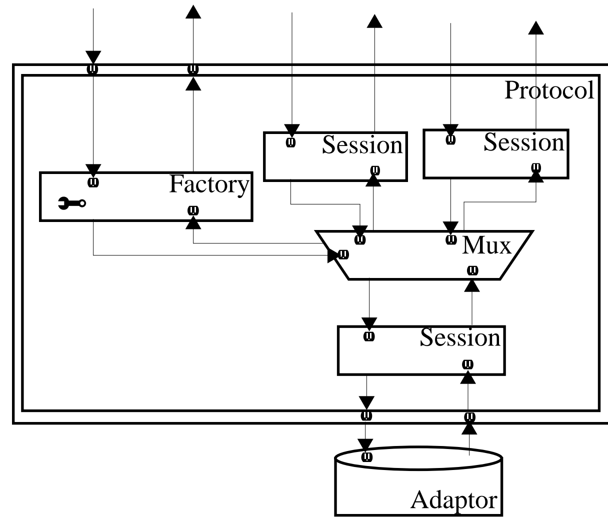

Figure 2: Aa example of a simple partial protocol graph

3.2 Basic Conduits architecture

The basic architecture of Java Conduits is based on that of Conduits+ by Hueni, Johnson and Engel [14]. The basic kinds of objects used are *conduits* and *messages*. Messages represent information that flows through a protocol stack. A conduit, on the other hand, is a software component representing some aspect of protocol functionality. To build an actual protocol, a number of conduits are connected into a graph. Protocols, moreover, are conduits themselves, and may be combined with other protocols and basic conduits into larger protocol graphs, representing protocol stacks.

There are five kinds of conduits: *Session, Mux, ConduitFactory, Adaptor* and *Protocol*. Each conduit has two sides: side A and side B. A given conduit can connect to either side A or side B of another conduit.

Sessions are the basic functional units of the framework. A session implements the finite state machine of a protocol, or some aspects of it. The session remembers the state of the communication and obtains timers and storage for partial messages from the framework. The session itself does not implement the behaviour of the protocol but delegates this to a number of State objects, using the State design pattern.

The Mux conduits are used to multiplex and demultiplex protocol messages. In practical terms, a Mux conduit has one side A that may be connected to any other conduit. The side B[0] of the Mux is typically connected to a ConduitFactory. In addition, the Mux has a number of additional side B[i] conduits. Protocol messages arriving from these conduits are multiplexed to the side A conduit, and vice versa.

If the Mux determines, during demultiplexing, that there is no suitable side B[i] conduit to which a message may be routed, the Mux routes the message to the ConduitFactory attached to side B[0]. The ConduitFactory creates a new Session (or Protocol) that will be able to handle the message, installs the newly created Session to the graph, and routes the message back to the Mux.

Adaptors are used to connect the conduit graph to the outside world. In conduit terms, adapters have only side A. The other side, side B, or the communication with the outside world, is beyond the scope of the framework, and can be implemented in whatever means feasible. For example, a conduit providing the TCP service may implement the Java socket abstraction.

A protocol is a kind of metaconduit that encapsulates several other conduits. A protocol has sides A and B. However, typically these are conduit connections that are mainly used for the delivery of various kinds of interprotocol control messages. Typically the actual data connections directly stretch between the conduits that are located inside some protocols. In practice, a protocol is little more than a conduit that happens to delegate its sides, i.e., side A and side B independently, to other conduits. The only complexity lies in the building of the initial conduit graph for the protocol. Once the graph is built, it is easy to frame it within a protocol object. The protocol object can then be used as a component in building other, more complex protocols or protocol stacks.

3.3 Using Java to build protocol components

Java 1.1 provides a number of features that facilitate component based software development. These include *inner classes*, Bean *properties*, *serialization* and Bean *events*. These all play an important role in making development of protocols easier.

A basic protocol component, i.e., a conduit, has (at least) two sides. Whenever a message arrives at the protocol component, it is important to know where the message came from, in order to be able to act on the message. On the other hand, it is desirable to view each conduit as a separate unit, having its own identity. Java inner classes and the way the Java Beans architecture uses them, provides a neat solution for this problem.

Each conduit is considered a single Java Bean. Internally the component is constructed from a number of objects: the conduit itself, sides A and B, and typically also some other objects depending on the exact sort of the conduit. The Conduit class itself is a normal Java class, specialized as a Session, Mux, ConduitFactory or such. On the other hand, the side objects, A and B, are implemented as inner classes of the Conduit class. In most respects, these objects are invisible to the rest of the object world. They implement the Conduit interface, delegating most of the methods back to the conduit itself. However, their being separate objects makes the source of a message arriving at a conduit immediately apparent.

Since the conduits are attached to each other, when constructing the conduit graph, the internal side objects are actually passed to the neighbour conduits. Now, when the neighbouring conduit passes a message, it will arrive at the receiving conduit through some side object. This side object uniquely identifies the source of the message, thereby allowing the receiving conduit to act appropriately.

The Java Bean properties play a different role. Using the properties, the individual conduits may publish run time attributes that a protocol designer may use through a visual design tool. For example, the Session conduits allow the designer to set the initial state as well as the set of allowed states using the properties. Similarly, the Accessor object connected to a Mux may be set up using the Beans property mechanism.

Java 1.1 provides a generic event facility that allows Beans and other objects to broadcast and receive event notifications. In addition to the few predefined notification types, the Beans are assumed to define new ones. Given this, it is natural to map conduit messages onto Java events.

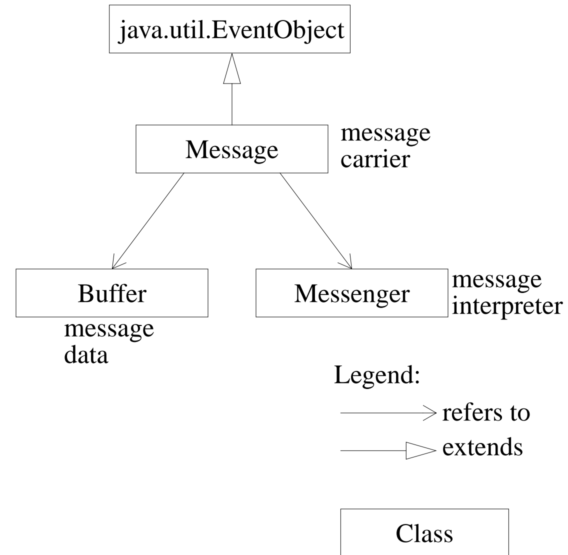

Figure 3: Structure of conduits messages

In Java Conduits, a protocol message is composed of three objects: a *message carrier*, a *message body* and a *message interpreter*. The message carrier extends the `java.util.EventObject` class, thereby declaring itself as a Bean event. The carrier includes references to the message body that holds the actual message data, and a message interpreter that provides protocol specific interpretation of the message data. The message interpreters are called Messengers, and they act in the role of a command according to the Command pattern [10].

Messages are passed from one conduit to the next one using the Java event delivery mechanism. The next conduit registers its internal side object as an event listener that will receive events generated by the previous conduit.

The actual message delivery is synchronous. In practice, the sending conduit indirectly invokes the receiving conduit's accept method, passing the message carrier as a parameter. The receiving conduit, depending on its type and purpose, may apply the Messenger to the current protocol state, yielding an action in the protocol state machine, replace the Messenger with another one, giving new interpretation to the message, or act on the message independent on the Messenger. Typically, the same event object is used to pass the message from conduit to conduit until the message is delayed or consumed.

Java Conduits use the provider / engine mechanism offered by the JDK 1.1 security API. Since neither the encryption / decryption functionality nor its interface specification was not available outside the United States,

we created a new engine class `java.security.Cipher` along the model of `java.security.Signature` and `java.security.MessageDigest` classes.

The protocols use the cryptographic algorithms directly through the security API. The data carried in the message body is typically encrypted or decrypted in situ. When the data is encrypted or decrypted, the associated Messenger is typically replaced to yield new interpretation for the data.

3.4 Usage of language level security features

Java offers a number of language level security features that allow a class library or a framework to be secure and open at the same time. The basic facility behind these features is the ability to control access to fields and methods. In Java, classes are organized in packages. A well designed package has a carefully crafted external interface that controls access to both black box and white box classes. Certain behaviour may be enforced by making classes or methods final and by restricting access to the internal features used to implement the behaviour. Furthermore, modern virtual machines divide classes into security domains based on their classloader. There are numerous examples of these approaches in the JDK itself. For example, the `java.net.Socket` class uses a separate implementation object, belonging to a subclass of the `java.net.SocketImpl` class, to provide network services. The internal SocketImpl object is not available to the users or subclasses* of the socket class. The `java.net.SocketImpl` class, on the other hand, implements all functionality as protected methods, thereby allowing it to be used as a white box.

The Java Conduits framework adheres to these conventions. The framework itself is constructed as a single package. The classes that are meant to be used as black boxes are made `final`. White box classes are usually `abstract`. Their behaviour is carefully divided into user extensible features and fixed functionality.

The combination of black box classes, fixed behaviour, and internal, invisible classes allows us to give the protocol implementor just the right amount of freedom. New protocols can be created, but the framework conventions cannot be broken. Nonetheless, liberal usage of explicit interfaces makes it possible to *extend* the framework, but again without the possibility of breaking the conventions used by the classes provided by the framework itself.

* Actually, other classes within the same package can access the SocketImpl object. Classes outside the package can't.

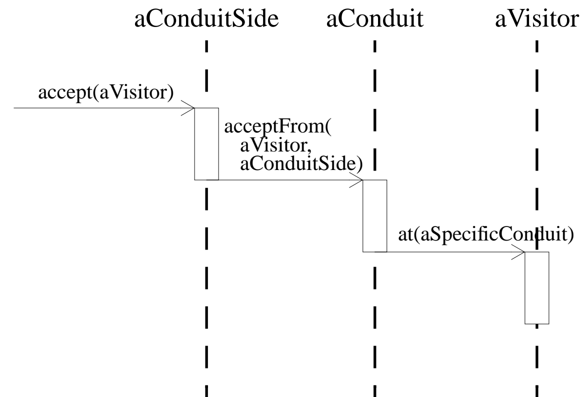

Figure 4: A visitor arrives at a Conduit

All this makes it possible to create *trusted protocols*, and to combine them with untrusted, application specific ones. This is especially important with cryptographic protocols. The cryptographic protocols need access to the user's cryptographic keys. Even though the actual encryption and other cryptographic functions are performed by a separate cryptoengine, the current Java 1.1 security API does not enforce key privacy. However, it is easy to create, e.g., an encryption / decryption microprotocol that encrypts or decrypts a buffer, but does not allow access to the keys themselves.

3.5 Object level design patterns used in the resulting architecture

The Conduits architecture is centred around the idea of a conduit graph that is traversed by protocol messages. The graph is the local representation of a protocol stack. The messages represent the protocol messages exchanged by the peer protocol implementations. This aspect of a graph and graph traversal is abstracted into a Visitor pattern [10]. The pattern is generalized in order to allow also other kinds of visitors to be introduced on demand. These may be needed, e.g., to pass interprotocol control messages or to visualize protocol behaviour.

In this pattern, a protocol message or other visitor arrives as a Java event at an internal side object of a conduit. The side object passes the message to the conduit itself. The conduit invokes the appropriate overloaded at(ConduitType) method of the message carrier, allowing the message decide how to act, according to the Visitor pattern.

As a more complex example of the usage of the gang of four patterns, let us consider the situation when a protocol message arrives at a Session that performs cryptographic functions (see Figure 5). The execution proceeds in steps, utilizing a number of design patterns.

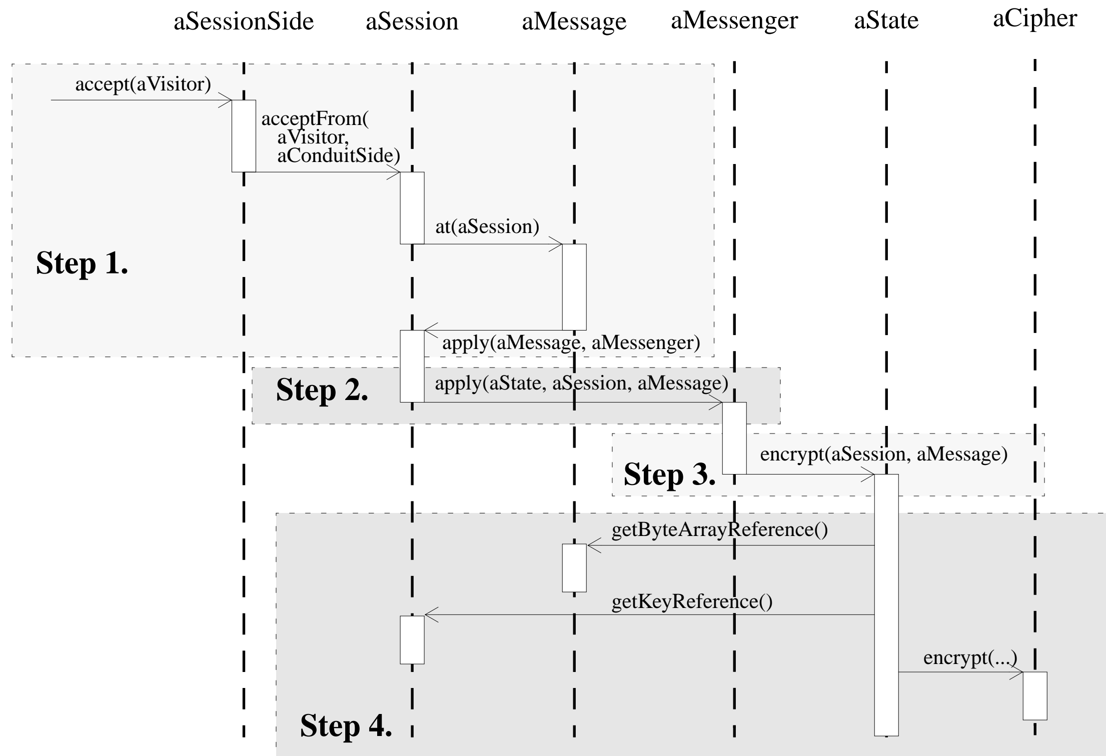

Figure 5: A Message arrives at a cryptographic Session

1. *The message arrives at the Session according to the Visitor pattern.*
   The message is passed to the Session's internal side as a Java Beans visitor event. The event is passed to the session, which invokes the message's at(Session) method. Since the visitor in hand is a message, it calls back the Session's apply(Message) method.
2. *The Session gets the message, and applies it according to the command pattern.*
   The Session uses the Messenger command object, and asks it to be applied on itself, using the current state and message.
3. *The Messenger command object acts on the session, state and message (second half of the Command pattern).*
   This behaviour is internal to the protocol. Typically all states of the protocol implement an interface that contains a number of command methods. The Messenger calls one of these, depending on the message's type. In the example situation where a message arrives and should be sent encrypted, the Messenger invokes the protocol state's encrypt(Session, Message) method.
4. *The current State object acts on the Session and Message.*
   This, again, depends on the protocol. The State may replace the current state at the Session with another State (according to the State pattern), modify the actual data carried by the message, or replace its interpretation by changing the Messenger associated with the Message. In our example, the State encrypts the message data. A reference to a Cipher has been obtained during the State initialization through the Java 1.1 security API. The key objects are stored at the Session conduit.

As examples of other kinds of usage of patterns, the following are worth mentioning:

- The actual encoding/decoding aspect of the Muxes is delegated to separate Accessor objects using the Strategy pattern.
- The State objects are designed to be shared between the Sessions of the same protocol. In order to encourage this behaviour, the base State class implements the basic details needed for the Singleton pattern.
- The ConduitFactories are used as black boxes in the framework. Each ConduitFactory has a reference to a Conduit that acts as its prototype, following the Prototype pattern.
- Obviously, the Adaptor conduits act according to the Adapter pattern with respect to the world outside the conduits framework.
- With respect to the Visitor pattern, the Protocol conduits act according to the Proxy pattern, delegating actual processing to the conduits encapsulated into the protocol.

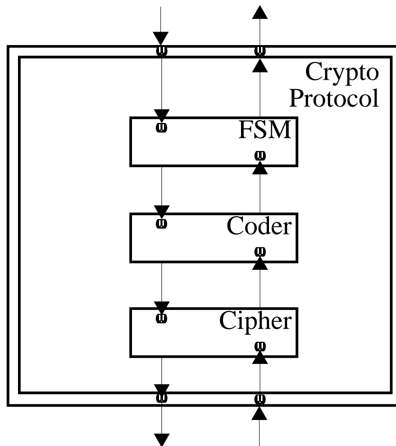

Figure 6: Cryptographic protocol pattern

3.6 Protocol design patterns

Our experience with the framework has shown that protocol independent implementation patterns do arise. That is, there seems to be certain common ways how the different conduits are connected to each other when building protocols. Here we show how the use of encryption tends to be reflected as a conduit topology pattern.

A cryptographic protocol handles pieces of information that are binary encoded and cryptographically protected. Usually the whole message is signed*, encrypted, or both. This yields a highly regular conduits structure where three sessions are stacked on top of each other (see Figure 6). The uppermost session (FSM) receives messages from upper protocols or applications, and maintains the protocol state machine, if any. Directly below lies a session that takes care of the binary encoding and decoding of the message data (Coder). The lowermost session within the protocol takes care of the actual cryptographic functions (Cipher).

According to the conduits architecture, the actual cryptographic keys are stored into the cryptosession. Thus, the information about what key to use is implicitly available from the conduit graph topology. However, this is not always feasible.

In the case of IPSEC AH protocol we resorted to storing the keying information as additional, out of band information within the outgoing protocol message. Similarly, the incoming messages are decorated with information about the security associations that actually were used to decrypt or the check the message integrity. These are then checked further up in the protocol stack to ensure security policy.

* Signed or otherwise integrity protected

4 Implementation experiences

Our current prototype is the third one in a series. The first working prototype was successfully implemented in December 1996. The second one was a complete rewrite, based on the experiences with the first one. The main difference between the second and third prototypes is Java Beans support. The only major change needed was to the message delivery mechanism, due to the added Java event support. Other than that, compliance with the Beans architecture required method naming changes and other minor changes needed to properly support serialization. The protocols themselves were transferred from the second framework prototype to the third with almost no changes. Our next step is to further enhance Java Beans support to facilitate visual protocol composition.

4.1 The framework

The elements used to build protocols are relatively small. This leads to a very piecemeal protocol development. According to our experience, once one is familiar with the model, the actual implementation of protocols is usually very straightforward and fast.

The small component approach seems to be very well suited for building microprotocols. For example, it is easy to represent the individual IPv6 header handlers as separate protocols, and create runtime structures to mix and match them appropriately.

Event delivery and scheduling. The basic Java event delivery mechanism is synchronous. The event source invokes, directly or indirectly, an appropriate method at every registered listener. However, nothing in the architecture mandates this approach. Since events are represented as objects, their delivery may well be queued and delayed. In fact, the listeners themselves may easily create an event queue if desired.

Our current goal is to achieve better performance on a uniprocessor implementation. Earlier experience with a UNIX STREAMS based IPSEC prototype [1] has shown that scheduling should be avoided on a uniprocessor environment. Therefore we have tried to minimize the number of threads and synchronized methods in the current prototype. This may change later when multiprocessing is taken care of.

The framework has one main thread that takes care of carrying a message through the conduit graph. It handles one message a time, passing it from conduit to conduit. If a conduit cannot pass the message, e.g., because it is a partial message and the other fragments are needed, the message stops at the conduit. The carrier thread then

handles the next message in queue, or waits if there are no messages currently waiting in a message/event queue.

A separate thread takes care of timers. Timer events are delivered to the conduits by the same thread as the message and other visitor events. A conduit may register a timer event to be scheduled at a particular time, after some delay, or periodically. Whenever the timer expires, a timer event is added to the message/event queue. After the carrier thread has handled a message, it takes the next message or timer event from the queue, and delivers it.

The adapters protect the conduits framework from other threads. The adapter methods are fully synchronized, and may be called by whatever threads. When a message arrives at an adapter from outside, the adapter wraps the message data into a conduit message carrier, attaches some interpretation to it, and places the message into the message/event queue. The carrier thread will select it at first appropriate opportunity.

Since Java I/O is inherently synchronous, the adapters communicating with the world external to the virtual machine typically contain their own internal threads. This allows the conduit processing to continue independent on delays on external I/O.

Memory management. The framework discourages explicit object creation and garbage collection. Typically, the constructors are either private (for black box classes) or protected (for white box classes). Most classes provide a public static instantiation method. This allows objects to be recycled by the class rather than being created and garbage collected for every occasion.

Footprint. The current framework prototype consists of 43 public classes, or about 3800 lines of Java source code (including comments). Only about 760 lines were written by hand; the rest were generated using an UML based case tool.

Of the 43 public classes, 23 are actual user visible classes. The rest are various exceptions (5), housekeeping classes (10) or other classes (5). The relationships of the user visible classes are displayed as an UML class diagram in Appendix A.

4.2 IPSEC

Our IPSEC prototype is designed to work with both IPv4 and IPv6. So far, it has been tested only with IPv6. It is designed to be policy neutral, allowing different kinds of security policies to be enforced.

A basic IP protocol stack, including IPSEC, is shown in Figure 7. In this configuration, the IPSEC is located as a separate protocol above IP. IP functions as usual, forwarding packets and fragments and passing upwards only the packets that are addressed to the current host. IPSEC receives complete packets from IP. The example configuration initially accepts packets that have either no protection, or are protected with AH, or with AH *and* ESP. It does not accept packets that are protected with ESP only or with e.g. double AH. This is one expression of policy. Furthermore, the conduit graph effectively prevents denial of service attacks with multiply encrypted packets.

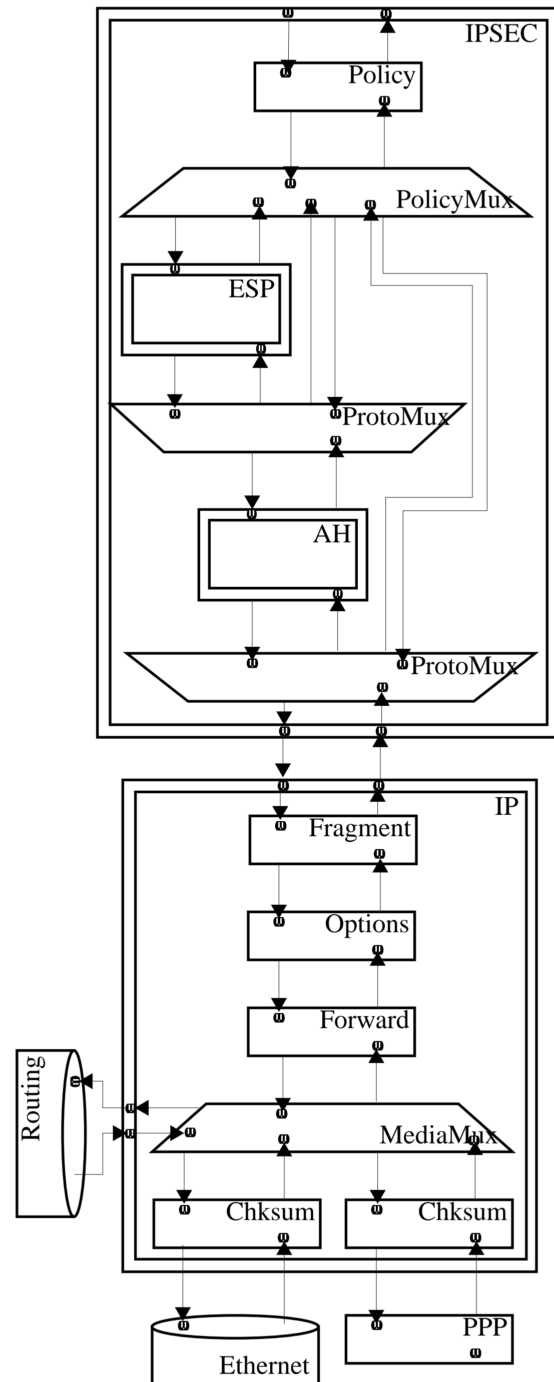

Figure 7: Host IPSEC conduit graph (simplified)

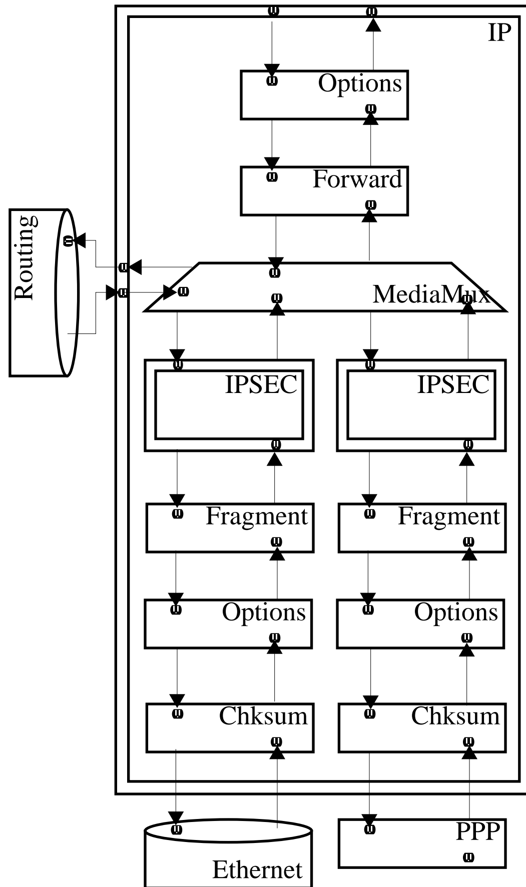

Figure 8: Security GW IPSEC conduit graph (simplified)

During input processing, the AH and ESP protocols decorate the packet with information about performed decryptions and checks. Later, at the policy session, this information is checked to ensure that the packet was protected according to the desired policy. We have also experimented with an alternative configuration, where the policy is checked immediately after every successful decryption or AH check. This seems to be more efficient, since faulty packets are typically dropped earlier. However, the resulting conduits graph is considerably more complex.

During output processing, the policy session and the policy mux together select the right level of protection for the outgoing packet. This information may be derived from the TCP/UDP port information or from tags attached to the message earlier in the protocol stack.

A different IPSEC configuration, suitable for a security gateway, is shown in Figure 8. In this case, instead of being on top of IP, IPSEC is integrated as a module within the IP protocol. Since the desired functionality is that of a security gateway, we want to run all packets through IPSEC and filter them appropriately. Since IPSEC is always applied to complete packets, all incoming packets must be reassembled. This is performed by the Fragment session, which takes care of fragmentation and reassembly.

Once a packet has travelled through IPSEC, passing the policy decisions is applies, it is routed normally. Packets destined to the local host are passed to the upper layers. Forwarded packets are run again through IPSEC, and a separate outgoing policy is applied to them. In this case, it is easier to base the outgoing policy on packet inspection rather than on separate tagging.

Our current IPSEC prototype runs on top of our IPv6 implementation, also built with Java Conduits, on Solaris. We use a separate Ethernet adapter, which is implemented as a native class on top of the Solaris DLPI interface. We have not yet applied JIT compiler technology, and therefore the current performance results are modest.

4.3 ISAKMP

The structure of our ISAKMP implementation is shown in Figure 9. The implementation is attached to the Java UDP implementation through a UDP adapter. Alternatively, it could be attached directly on top of our own UDP implementation. On top of the ISAKMP implementation lies a security policy manager, which forms the “political layer” of our protocol stack.

ISAKMP packets received through the UDP adaptor are directed either to an ISAKMP factory or to some established ISAKMP session, depending on the ISAKMP cookies. If the packet initiates a new ISAKMP association (i.e., is the first main mode packet), the ISAKMP factory consults the upper layer to determine whether the association should be established. The same applies for proposals for new AH or ESP associations. If a new AH or ESP association is accepted by the policy, the AH/ESP factory creates a new AH or ESP protocol instance. The protocol instance takes care of running the ISAKMP quick mode to create the new association.

When a new AH or ESP association has been established, the negotiated parameters are passed to the policy layer. The policy manager takes care of creating the new association to the IP stack, either through PF_KEY interface (if a non-conduits IPSEC is used), or by modifying the IP/IPSEC conduit graph appropriately.

The main novelty in our approach is the separation of the ISAKMP daemon and the policy manager. Currently the policy manager is implemented as a separate conduits protocol. However, it would be possible to implement the policy manager outside the conduits framework as well, and use Java events to communicate between the conduit world and the policy manager.

The current implementation is slightly out of date, due to changes recently made to the ISAKMP and Oakley Internet drafts [19].

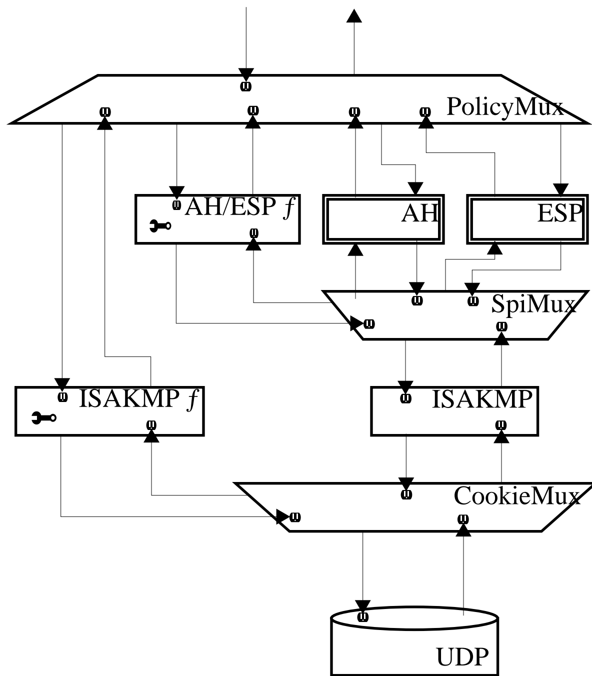

Figure 9: ISAKMP conduit graph (simplified)

4.4 Non-cryptographic protocols

In addition to the cryptographic protocols, we have implemented partial but functional prototypes of the IPv4, IPv6, ARP, ICMP (IPv4 version only), UDP and TCP protocols. Integration of these, along with the IPSEC implementation, into a complete TCP/IP protocol stack is under way.

4.5 Availability

The current framework prototype is available at `http://www.tcm.hut.fi/~pnr/jacob/`. The actual protocol prototypes and the protocol sandbox prototype are available directly from the authors. An integrated, JDK 1.2 based release is expected to be published in late May or early June.

5 Summary

We define an architecture and an object oriented implementation framework for cryptographic protocols. The architecture is based on the Internet, WWW, Java and an initial security context, and optionally augmented with a PKI and the ISAKMP and IPSEC protocols. The implementation framework is based on a fully object oriented language, so it benefits greatly from design patterns, making it easy to use and extensible at the same time. Furthermore, the use of object level design patterns leads to a highly stylistic way of implementing protocols, thereby allowing creation of new, higher level *protocol patterns*.

The implementation framework was developed with JDK 1.1 using the Java Beans and the security API of Java 1.1. In the framework, protocols are built from lower level component called conduits. The protocols are conduits themselves, allowing incremental building of higher level protocols from lower level ones.

The Java execution environment allows the resulting protocols to be seamlessly integrated into the operating system and applications alike. This is especially important for security protocols, since this allows the security systems at various levels to be integrated. We have taken advantage of the Java language level security features (packages, visibility, classloaders). The framework is implemented as a single Java package. Special attention has been paid to dividing the functionality into fixed and user customizable feature sets.

So far we have implemented functional prototypes of IPv4, IPv6, ARP, ICMP, UDP, TCP, IPSEC and ISAKMP protocols. We expect to implement prototypes of further protocols in the near future.

6 Future work

There are a number of future projects that we are planning to start. Due to our limited resources we have not been able to work on all the fronts simultaneously.

Even though a PKI is not an absolute prerequisite for using our architecture, it is in practice essential for most wide-spread real-life applications. We are currently implementing SPKI type certificates that will be integrated into our framework.

The use of security services and features is usually mandated by security policies. The management of security policies in global networks has become a major challenge. We have recently started a project to design and implement an Internet Security Policy Management Architecture (ISPMA) based on trusted Security Policy Managers (SPM). When a user contacts a service, they need to be authorized. Authorization may be based on the identity or credentials of the user. Having obtained the necessary information from the user, the server asks the SPM if the user can be granted the kind of access that they have requested. Naturally all communications between the parties need to be secured.

A graphical Java Beans editor could make the work of the implementor much more efficient than it currently is. This would also make it easier to train new, on the average only average, programmers to develop secure appli-

cations. In a graphical editor, the building blocks of our architecture would show as graphical objects that can be freely combined into a multitude of applications. The amount of programming work in developing such an editor is quite large and there certainly are lots of ongoing projects in the area of graphical Java Beans editors. Our plan is to take an existing editor and integrate it into our environment.

So far our work has been focused on the design and implementation of secure application specific protocols. Our long term goal is to create an integrated development environment for entire secure applications. This environment would also include tools for creating the user interface and database parts of the applications.

References

[1] Timo P. Aalto and Pekka Nikander, "A Modular, STREAMS Based IPSEC for Solaris 2.x Systems", In *Proceedings of Nordic Workshop on Secure Computer Systems,* Goethenburg, Sweden, November 1996.

[2] Robert Allen and David Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, 6(3), July 1997.

[3] Ross J. Anderson, "Programming Satan's Computer", In *Computer Science Today — Recent Trends and Developments*, LNCS 1000, pp. 426–440, Springer-Verlag, 1995.

[4] Ross J. Anderson and Roger Needham, "Robustness principles for public key protocols", *Advances in Cryptology—CRYPTO'95 Proceedings*, Springer-Verlag, 1995.

[5] Ken Arnold and James Gosling, *The Java Programming Languag*e, Addison-Wesley, 1996.

[6] Randal Atkinson, *Security Architecture for the Internet Protocol,* RFC1825, Internet Engineering Task Force, August 1995.

[7] Kent Beck and Ralph Johnson, "Patterns Generate Architectures", In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'94)*, Bologna, Italy, pp. 139–149, Springer-Verlag, 1994.

[8] Kenneth Birman and Robert Cooper, "The ISIS Project: Real Experience with a Fault Tolerant Programming System", *Operating Systems Review*, pp. 103–107, April 1991.

[9] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas and Tatu Ylönen, *Simple Public Key Certificate*, Internet-Draft `draft-ietf-spki-cert-structure-02.txt`, work in progress, Internet Engineering Task Force, July 1997.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] Benoit Garbinato, Rachid Guerraoui, "Using the Strategy Design Pattern to Compose Reliable Distributed Protocols", The *Third Conference on Object-Oriented Technologies and Systems (COOTS) Proceedings*, Portland, Oregon, June 16-20, 1997, pp. 221–232.

[12] Li Gong, *Java Security Architecture (JDK1.2) DRAFT DOCUMENT (Version 0.7)*, Sun Microsystems, October 1, 1997, `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.htm`

[13] Nevin Heintze and J. D. Tygar, "A model for secure protocols and their compositions", In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy,* pp. 2–13, IEEE Computer Society Press, May 1994.

[14] Herman Hueni, Ralph Johnson, R. Angel, "A framework for network protocol software", *Object Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*, ACM Press 1995.

[15] N. C. Hutchinson and L. L. Peterson, "The x–Kernel: An architecture for implementing network protocols." *IEEE Transactions on Software Engineering,* 17(1):64–76, January 1991.

[16] Darrell Kindred, Jaennette M. Wing, "Fast, Automatic Checking of Cryptographic Protocols", In *Proceedings of the Second USENIX Workshop on Electronic Commerce,* November 18-21, 1996, Oakland, California.

[17] Wenbo Mao and Colin A. Boyd, "Development of authentication protocols: some misconceptions and a new approach", *Proceedings of IEEE Computer Security Foundations Workshop VII*, IEEE Computer Society Press, 1994, pp. 178-186.

[18] S. W. O'Malley, L. L. Peterson, "A Dynamic Network Architecture", *ACM Transactions on Computer Systems* 10(2):110–143, May 1992.

[19] Douglas Maughan, Mark Schertler, Mark Schneider and Jeff Turner, *Internet Security Association and Key Management Protocol (ISAKMP),* Internet-Draft `draft-ietf-ipsec-isakmp-08.txt`, work in progress, Internet Engineering Task Force, July 1997.

[20] Bertrand Meyer, "The Next Software Breakthrough", *Computer*, 30(7): 113–114, IEEE Computer Society, July 1997.

[21] Pekka Nikander, *Modelling of Cryptographic Protocols,* Licenciate's Thesis, Helsinki University of Technology, December 1997.

[22] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. "Paving the road to network security, or the value of small cobblestones". In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, February 1994.

[23] Michael K. Reiter, Kenneth P. Birman and Robbert Van Renesse, *A Security Architecture for Fault-Tolerant Systems*, Cornell University Technical Report, TR93-1354, June, 1993.

[24] Robbert van Renesse, Kenneth P. Birman and Silvano Maffeis, "Horus, a flexible Group Communication System," *Communications of the ACM*, April 1996.

[25] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr, "A Framework for Protocol Composition in Horus", In *Proceedings of Principles of Distributed Computing*, August, 1995.

[26] Jorma Rinkinen, *Java DES Speed Test*, `http://www.tcm.hut.fi/~jrin/des/` July 1997.

[27] Aviel D. Rubin and Peter Honeyman, *Formal methods for the analysis of authentication protocols*, Technical Report 93–7, Center for Information Technology Integration, Department of Electrical Engineering and Computer Science, University of Michigan, 8. November 1993.

[28] Douglas C. Schmidt, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software", *Communications of the ACM*, 38(10):65–74, October 1995.

[29] Gustavus J. Simmons, "Cryptanalysis and protocol failures", *Communications of th*e ACM, 37(11):56–65, November 1994.

[30] R. Thayer, N. Doraswamy and R. Glenn, IP Security Document Roadmap, Internet-Draft `draft-ietf-ipsec-doc-roadmap-01.txt`, work in progress, Internet Engineering Task Force, July 1997.

[31] Amy Moormann Zremski and Jeannette M. Wing, "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.

[32] Jonathan M. Zweig and Ralph E. Johnson, "The Conduit: A Communication Abstraction in C++", In *Usenix C++ Conference Proceedings,* San Francisco, CA, April 9–11, 1990, pp. 191–204. The Usenix Association 1990.

[33] Joanne Wu (Editor), *Component-Based Software with Java Beans and ActiveX*, White paper, Sun Microsystems, `http://www.sun.com/javastation/whitepapers/javabeans/javabean_ch1.html`, August 1997.

Appendix A UML class diagram