An Object-Oriented ISAKMP Implementation

Sanna Liimatainen
Sanna.Liimatainen@hut.fi
Helsinki University of Technology *

October 7, 1999

Abstract

When two parties want to use a secure communication path, they first have to agree on how the connection will be established. They have to authenticate the other party and agree on protocols and keys. The Internet Security Association and Key Management Protocol (ISAKMP) [4] is a protocol framework that has been developed for exchanging the keys and authentication information protocol independently.

In this paper I present an object-oriented implementation of the ISAKMP protocol. I have implemented the ISAKMP protocol using the Jacob framework [5]. The Jacob framework is developed in Helsinki University of Technology for implementing cryptographic protocols.

Keywords: Authentication, Key Management, ISAKMP, Object-Oriented Framework, Conduits+, Jacob.

1 Introduction

Design Patterns and software frameworks are important tools in object-oriented software development. Design Patterns are reusable solutions to some common problem. Frameworks are extendible program skeletons that are developed for some specific purpose. Object-oriented frameworks are usually heavily grounded on Design Patterns.

The Internet Security Association and Key Management Protocol (ISAKMP) is a protocol framework that is used for negotiation of secure communication. ISAKMP can be used for authentication and key exchange and it supports several algorithms [4].

In this paper, I introduce my object-oriented implementation of the ISAKMP protocol. I also discuss the developing process and how object-oriented tools have helped me, as a less experienced programmer.

The rest of this paper is organized as follows. Section 2 introduces the object-oriented design tools used in this implementation. The third section describes an authentication and key management protocol and the fourth section describes the design process. The fifth chapter introduces the object-oriented architecture of my ISAKMP implementation and the sixth chapter presents the implementation. Finally, the seventh chapter is my conclusions.

*This work was funded by the TeSSA research project at Helsinki University of Technology under a grant from Tekes.

2 Object-Oriented Protocol Design

Object-oriented programming is based on a different way of thinking than the traditional functional programming. A programming problem must be divided into classes that represents the entities involved. The program consists of these classes and their interactions. Ideally, these classes are easily reusable. When the classes are well documented and proven to be usable in many applications, they become known as design patterns. The best known book about reusable components is Design Patterns written by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides [2].

Communication protocols have usually a lot of similar functionalities. Using a framework allows the program developer to concentrate on problems that are specific to the protocol being implemented. Frameworks are developed to solve problems of some specific area. For example, the Conduit+ framework [3] is built for implementing communication protocols and the Jacob framework [5] is built for implementing especially cryptographic communication protocols. Implementing cryptographic protocols is harder than common communication protocols because the programmer has to consider also intentional errors caused by a malicous attacker.

The next section presents the Conduits+ and Jacob frameworks.

2.1 Protocol Frameworks

2.1.1 Conduits+

In their paper, Herman Hueni, Ralph Johnson and Robert Engel introduced the Conduits+ framework for developing network protocol software [3]. The Conduits+ framework contains information chunks and four kind of conduits. Conduits are software components that usually have two sides: sideA and sideB. Information chunks travel through the conduits from one side to opposite side. Figure 1 shows the four kinds of conduits.

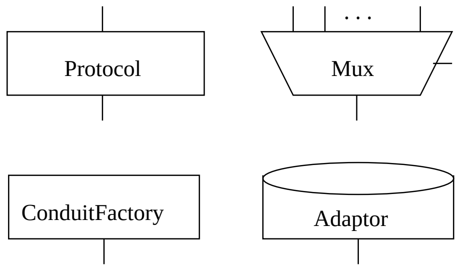

Figure 1: Four kinds of conduits

A Protocol conduit implements a finite state machine that describes a communication protocol. It remembers the current state of the communication. The Protocol may have counters, timers and storages that are needed for handling information chunks. This conduit has two sides, both connected to exactly one neighbor conduit.

A Mux can have several conduits connected to its sideB. The Mux multiplexes information chunks that arrives from some sideB conduit to the conduit connected to its sideA. An information chunk that arrives to the Mux from sideA is demultiplexed to one sideB conduit based on the session key of the information chunk. New conduits can be added to and removed from the Mux's sideB dynamically.

When the Mux receives an information chunk from sideA with no suitable conduit on sideB, the Mux routes the chunk to its default conduit. This default conduit is usually *a ConduitFactory*. The ConduitFactory then creates the required session protocol to sideB of the Mux. Now the Mux is ready to handle this information chunk.

An Adapter conduit is an interface to some software or hardware that is outside of the protocol. The Adapter is connected from its sideA to the protocol.

All conduits can be connected to any other kind of conduit. This allows a programmer to construct a complex graph that implements a communication protocol.

2.1.2 Jacob

The Java Conduits Beans (Jacob) framework is based on the Conduits+ framework [5]. The goal of Jacob is to allow application-specific secure protocols to be built from components.

In Jacob, the Conduits+ Protocol conduit is divided to two different conduits: *Session* and *Protocol*. The Session conduit implements a basic functional unit of a protocol and it has basically same function as the protocol conduit of the Conduits+ framework. The Protocol conduit of the Jacob framework is a kind of meta-conduit that can contain several other conduits. The Protocol conduit makes the structure of conduit graphs simpler. A protocol can be encapsulated in a Protocol conduit that can then be used just like other components to built more complex graphs.

Conduits+ has been implemented with C++ programmin language and Jacob framework is based on the Java programming language. Java has several benefits compared to C++. Each conduit is a Java Bean. The sides of a conduits are inner classes of the Conduit class. When conduits are connected, the internal side object is passed to the neighbor conduit and a message arrives to conduit from a known side. The internal side of the neighbor conduit is an event listener that receives messages from the previous conduit.

The messages are an invisible part of Conduits+ graph. In the Jacob framework a message consists of three parts: message content, Messenger and Out-of-Band. The message contents may be any kind of data, the Messenger knows how to handle the message content and the Out-of-Band part consists of information that may be needed elsewhere in the Conduits+ graph. Usually in Jacob, each Session conduit demands two Messengers: one for messages that travel from sideA to sideB and one for the other direction. The Session conduit changes the Messenger for the next session that is reached when the message travels through the graph. In other words, the Session must know where the message is going and what kind of Messenger it needs at the destination.

3 Authentication and Key Management

Protocols provide rules for communication between computers. Protocols define message formats and specify how to handle arriving messages and errors. A security protocol performs a security service (or services) for networked communication.

Authentication protocols are developed to authenticate the other peer's identity or authority or origin of a message. Some protocols combine authentication and key management. These protocols solve a general problem: How communicating parties can exchange a secret key and at the same time be sure that they are communicating together and not with some other person.

There are two kind of ciphers: symmetric ciphers and public key ciphers. To use these ciphers, we need key management to distribute the keys. Symmetric cipher need a secret key between all parties that want to communicate with each other. In public key cryptography, every party has two keys: a public and a secret keys. Communication is encrypted with other peer's public key. Only this other peer can decrypt the message with it's secret key. Both of these ciphers need key generation, key distribution and key storage systems that belong to key management.

3.1 Internet Security Association and Key Management Protocol

The Internet Engineering Task Force (IETF) has standardized the Internet Security Association and Key Management Protocol (ISAKMP). It is a protocol that combines the security concepts of authentication, key management, and security associations to establish the required security for government, commercial, and private communications on the Internet. ISAKMP defines procedures and packet formats to establish, negotiate, modify and delete Security Associations (SA) [4].

ISAKMP is a *framework*. It does not enforce the use of any specific cryptographic algorithm, key generation technique or security mechanism. A *Security Association* (SA) defines the set of services and mechanisms that are required to protect communication. In the ISAKMP, Security Associations are negotiated in two *phases*. In the first phase, two ISAKMP servers agree on how the forthcoming negotiation communication is to be protected, and a generic ISAKMP SA is established. In the second phase, negotiated security associations for some other security protocols, like IPSEC, are established. These security associations can be used to protect message/data exchanges of different security protocols.

Every ISAKMP message begins with an ISAKMP Header. This header contains information to maintain state, process payloads, and prevent possible denial of service or replay attacks. There is an Initiator Cookie and a Responder Cookie field to identify communication peers as well as to which Security Association this message belongs.

ISAKMP introduces 13 *payloads* that can be used to exchange information. These payloads are Security Association Payload, Proposal Payload, Transform Payload, Key Exchange Payload, Identification Payload, Hash Payload, Nonce Payload, Certificate Payload, Certificate Request Payload, Signature Payload, Notification Payload, Delete Payload and Vendor ID Payload. Every payload starts with a Generic Payload Header that tells what the next payload of the ISAKMP message is and how long this payload is.

ISAKMP defines five default *exchange types*: Base Exchange, Identity Protection Exchange, Authentication Only Exchange, Aggressive Exchange and Informational Exchange. The different exchanges provide different protection for the exchanged information. The Base Exchange and the Aggressive Exchange provide key and authentication exchange. In the Identity Protection Exchange, the key is exchanged first and identification and authentication information is protected by encrypting communication with this key. The Authentication Only Exchange changes only authentication information and the Informational Exchange informs the other party about deletion of a security association or that an error has occurred.

ISAKMP is an extendible protocol. A designer can define own payloads and exchanges.

4 Development Process

Martin Fowler describes four phases in a software development process: interception, elaboration, construction and transition [1]. During the interception phase, the project is sold to its customers and its scope is decided. In elaboration, more and detailed information is collected about requirements. The developers decide with the customers what and how they are building and what technology should be used in the development. Risks of the project is also scanned in the elaboration phase. The result of the elaboration is a plan for the construction phase. The construction phase is iterative. Each iteration consists of analysis, coding, testing and integration. Also more detailed planning can be made within each iteration. The last phase is transition, which includes beta testing, performance tuning and user training.

In my developing process, there were two rounds of three phases that actually consists of only the construction phase of development. First, I made a draft for a conduit model of ISAKMP. Based on this conduit model, I made the first class diagrams that contained mainly the conduit classes and their subclasses. The messages and their handling were only vaguely modeled in these first diagrams. Then I made a crude sequence diagram about handling cookies in an ISAKMP message exchange. After this first development round I had quite a clear picture of what I was doing.

In the second round, these three phases were performed again, and some modifications and specifications were made, especially for message handling, to the conduit model and the class diagrams.

For an inexperienced programmer, the use of a case tool can make programming easier. In my work I used the ClassDesigner of Cayenne Software. In Telecommunication Software and Multimedia Laboratory at Helsinki University of Technology, a Java code generator was added to the ClassDesigner [7]. I used this code generator to get the class skeletons. Then I implemented the contents to classes. Every time I wanted to change, for example, a name or a type of a variable I had to change it first into my model and then generate the code again. This way the documentation of my implementation is in good condition and up to date all the time.

5 ISAKMP Architecture in Object-Oriented Terms

My ISAKMP implementation was made in three phases. First, the handling of cookies and the basic ISAKMP was planned and implemented. The second phase consisted of a plan for the Base Exchange and its implementation. In the last phase, all other exchanges were planned and implemented. The following sections describe the architecture of the ISAKMP.

5.1 Conduits+-model

Figure 2 shows the Conduits+ graph of the ISAKMP. The ISAKMP is presented as a Protocol conduit that is connected to two Adaptor conduits: the PolicyAdaptor and the NetworkAdaptor. The PolicyAdaptor is used by the Security Policy Daemon (SPD) that decides which connections are allowed and which not and what kind of connections can be accepted. The NetworkAdaptor represents the network, including all underlaying protocols (like UDP and IP) and hardware. The ISAKMPSubProtocol conduit is dynamically added to the graph when an application wants to send a message to someone else or the first message of new SA negotiation is coming from the network side.

The base of the ISAKMP protocol consists of ISAKMPCheckingSession, ISAKMPReturnMux, ISAKMPCookieMux, ISAKMPInitiator-

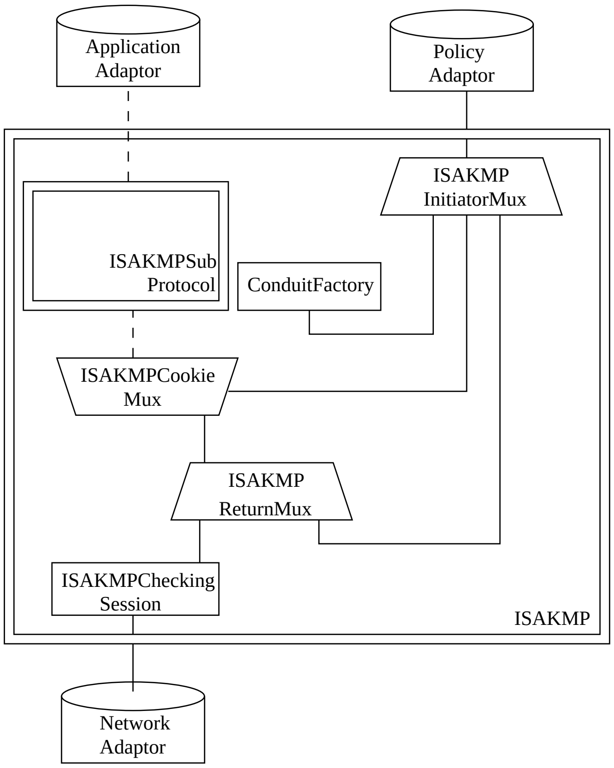

Figure 2: Conduits+ graph of ISAKMP

Mux and ConduitFactory. Each ISAKMP protocol has its own ISAKMPCookieStorage for storaging the cookies. Cookies consists of the initiator cookie and the responder cookie from the ISAKMP message header. The function of the ISAKMPCheckingSession is to check that the version numbers and length of message are correct in incoming messages header. The ISAKMPCheckingSession sets these values in the header of the outgoing message. The ISAKMPCookieMux checks the cookies from incoming message. If the cookies are unknown, it passes the message to the PolicyAdaptor. If the cookies have been stored in the ISAKMPCookieStorage, there is a corresponding ISAKMPSubProtocol and the message is directed to this ISAKMPSubProtocol. In the other direction, the ISAKMPCookieMux adds the cookies to the outgoing message.

In the case of an unknown cookie, the PolicyAdaptor gets the message. If the policy decides to allow the connection, it sends a message to the ConduitFactory, that inserts a new ISAKMPSubProtocol to the ISAKMPCookieMux. The original message is send back to the CookieMux through the ISAKMPInitiatorMux and the ISAKMPReturnMux. Now the ISAKMPCookieMux can send the message to the new ISAKMPSubProtocol. Similarly, when the ISAKMP protocol is sending the first message (because some application has asked a new connection), a new ISAKMPSubProtocol is connected to the ISAKMPCookieMux.

In my implementation, every ISAKMP exchange has its own ISAKMPSubProtocol. For example, figure 3 shows the contents of ISAKMPBaseExchangeSubProtocol. Every ISAKMPSubProtocol has three sessions: ISAKMPCodeSession, ISAKMPMessageSession and corresponding ExchangeSession. The ISAKMPCodeSession encrypts outgoing and decrypts incoming messages that have the encryption bit set in the ISAKMP header. The ISAKMPMessageSession converts the messages from bit stream to objects and vice versa. The third session in the ISAKMPBaseExchangeSubProtocol is an ISAKMPBaseExchangeSession that implements the actual finite state machine of the base exchange. Other subprotocols have their own exchange sessions.

6 Implementation

Sessions, Protocols and Adaptors from the Conduits+ graph are easy to implement as Java classes. Every Session has at least one state class, ExchangeSessions have several state classes. Protocols and Adaptors have a

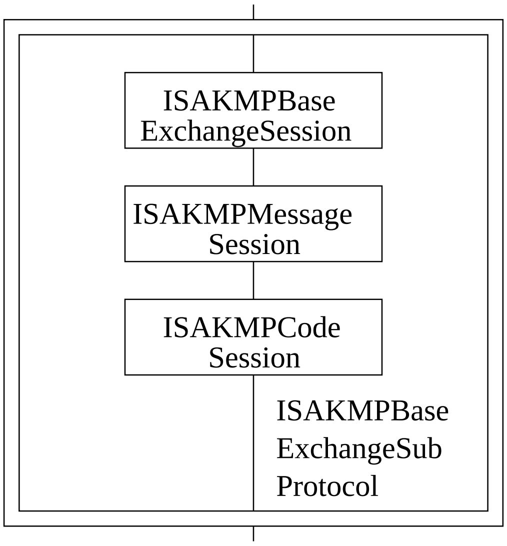

Figure 3: Conduits+ graph of ISAKMPBase-ExchangeSubProtocol

separate class for each conduit.

In Jacob, there is actually only one Mux and only one ConduitFactory. The Mux has an Accessor that knows how a key can be found in the message. Accessors have gotten their names from Muxes in the Conduits+ graph. A prototype of a class is given to the ConduitFactory and then the ConduitFactory can dynamically connect new conduits to a Mux. For example, when the first message coming from the network has reached the PolicyAdaptor, the type of the ISAKMP exchange is extracted from the ISAKMP Header. Next, an order to insert a new ISAKMPSubProtocol to the ISAKMPCookieMux is sent to the ConduitFactory. This order contains a prototype of the correct kind of ISAKMPSubProtocol according to the exchange type.

Every Session conduit has two messengers to handle incoming and outgoing messages.The implementation of those messates are described below.

6.1 Messages

In the Jacob framework, a message consists of a Messenger, an Out-of-Band object and the actual contents of the message. The Messenger part know how to handle the message contents. Figure 4 shows the UML model of messengers in my ISAKMP implementation. The Messenger is an interface. The BufferMessenger allows programmers to handle binary format messages while the InstallMessenger is used by the ConduitFactory for connecting new sides to a Mux. These Messengers are all part of the conduits model.

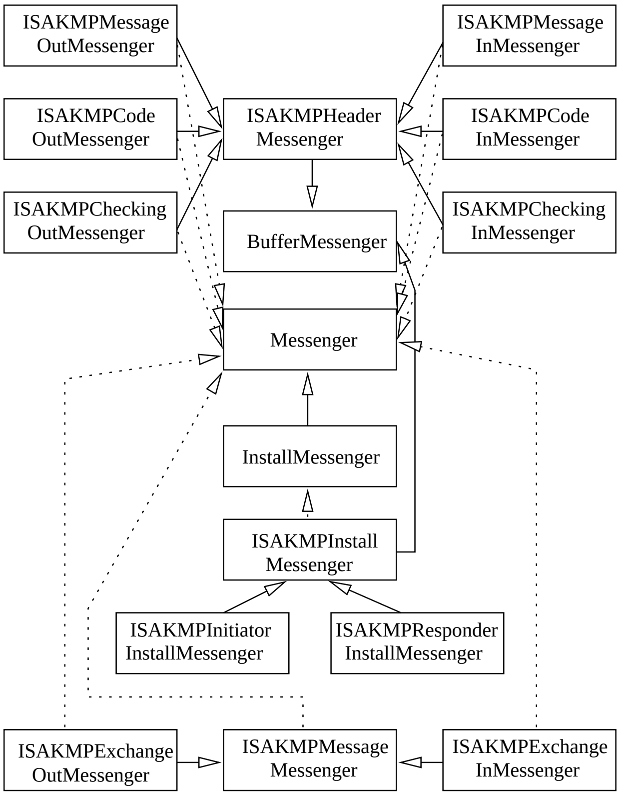

Figure 4: ISAKMP Messengers

The ISAKMPHeaderMessenger knows how to handle binary ISAKMP messages and where information is stored in the ISAKMP Header. ISAKMPCheckingInMessenger and ISAKMPCheckingOutMessenger are used by ISAKMPCheckingSession for handling and checking a message, especially the ISAKMP Header. A message coming from network through ISAKMPCheckingSession to the ISAKMPSubProtocol contain an ISAKMPCodeInMessenger. It is used in the ISAKMPCodeSession. The ISAKMPCodeSession encrypts and decrypts messages if the encryption bit is set in the ISAKMP Header. ISAKMPCodeSession changes the messenger to ISAKMPMessageInMessenger that is used in ISAKMPMessageSession and knows how to change the message from bit stream format to payload objects. The corresponding messenger for outgoing messages is ISAKMPMessageOutMessenger. ISAKMPExchangeInMessenger and ISAKMPExchangeOutMessenger are used to handle messages in the finite state machines of all ExchangeSessions.

When a new connection has to be added to the ISAKMPCookieMux, the PolicyAdaptor sends a message that contains ISAKMPInitiatorInstallMessenger or ISAKMPResponderInstallMessenger to the ConduitFactory. The ISAKMPInitiatorInstallMessenger is used when first message needs a new ISAKMPSubProtocol. Similarly the ISAKMPResponderInstallMessenger is used when the first message is received from the network side. The ConduitFactory responds with a Notifier message when it has successfully added new ISAKMPSubProtocol to the ISAKMPCookieMux. The original message is in the Notifier. If the original message was coming from the Network the message is sent through ISAKMPInitiatorMux, ISAKMPReturnMux and ISAKMPCookieMux to the newly created ISAKMPSubProtocol. If it was an application that asked for a new connection, the first message of appropriate exchange is sent by the ISAKMPSubProtocol.

6.2 Sessions and States of Base Exchange

There is a finite state machine in ISAKMPBaseExchangeSession that knows what message should be received or sent next. PolicyAdaptor adds policy information to this session when it has been created. According to the policy, the finite state machine knows what to put into messages. Figure 5 shows the finite state machine of base exchange.

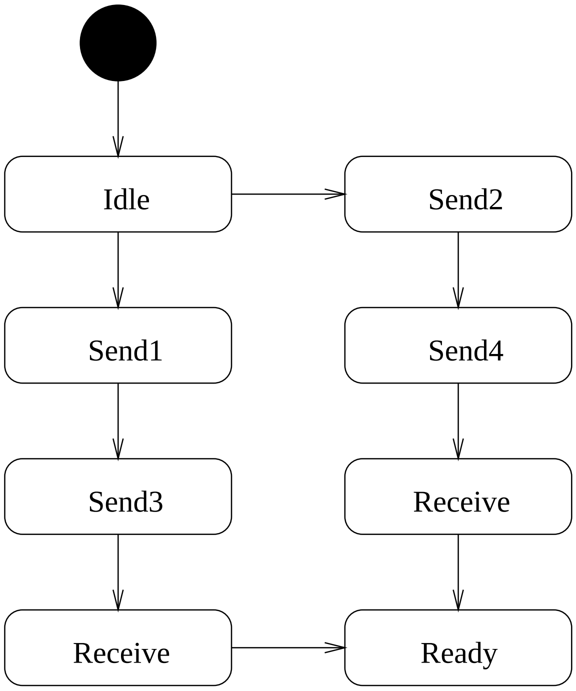

Figure 5: Finite State Machine of Base Exchange

When a message reaches ISAKMPBaseExchangeSession, it is addressed to Idle state. This state knows whether the message is coming from network or from application and changes the state of session to correct state accordingly. If the message is coming from application, the new state is Send1 and if the message is from network, finite state machines next state is Send2. States are named according to the message that they send next. Finally, the last message is sent and the finite state machine waits if some mistake that occurred during the exchange. If not, the state machine moves to the Ready state. Message exchange is over and new Security Association has been established.

7 Conclusions

Frameworks are very useful for developing communication protocols. Cryptographic protocols are more difficult than common communication protocols because their requirements are more demanding. A communication protocol has to work after some random error but cryptographic protocol has to prevent malicious persons from making damages.

Knowing how to use a framework made by someone else is not very easy. Frameworks are rather a new technology in computer science and, in many cases, need developing. Understanding a framework depends on how good the documentation of the framework is.

The Jacob framework consists of two parts: conduits and messages that travel through conduits. The messages and their functions are more difficult to understand than the conduits and their connections because the messages are less concrete. Programmer has to decide if she put some functionality to messenger or to state.

Use of case tool and code generator saves me from "fear of white paper" and helps me to start programming and understand the connections between objects. Use of these tools also forces me to keep the class diagrams up to date. One improvement to the case tool could be, that it automaticaly insert set and get methods to public class variables.

8 Future Work

The ISAKMP framework does not enforce to use some specific key exchange algorithm. The Internet Key Exchange Protocol is developed by IETF to be one possible key exchange algorithm [6]. This protocol will be implemented next.

In the future, an new exchange type for SPKI certificates is added to our implementation. The purpose of this exchange is to allow secure online verifications of SPKI certificates.

References

[1] M. Fowler: UML Distilled, Addison-Wesley, 1997

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995

[3] H. Huni, R. Johnson, R. Engel, A Framework for Network Protocol Software. Object-Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95), ACM Press, 1995.

[4] D. Maughan, M. Schertler, M. Schneider, J. Turner. Internet Security Association and Key Management Protocol (ISAKMP). RFC 2408, November 1998.

[5] P.Nikander, A.Karila, A Java Beans Component Architecture for Cryptographic Protocols. Proceedings of the 7th USENIX Security Symposium, San Antonio, Texas. Usenix Association, 1998.

[6] D. Harkins, D. Carrel. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[7] E. Heimonen: TCM's Java Code Generator for CayenneSoft's ClassDesigner, 1998 URL: http://www.tcm.hut.fi/Research/TeSSA/Jacob/codegen.html