UNIVERSIDADE NOVA DE LISBOA

Faculdade de Ciências e Tecnologia

Departamento de Engenharia Electrotécnica e de Computadores

Supporting NAT Traversal and Secure Communications in a Protocol Implementation Framework

Por:

Pedro Arruda Pereira

Dissertação apresentada na Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa para obtenção do Grau
de Mestre em Engenharia Electrotécnica e de Computadores.

Orientador: Prof. Doutor Paulo da Costa Luís Fonseca Pinto

Lisboa
2011

Supporting NAT Traversal and Secure Communications in a Protocol Implementation Framework

To my family

Preface

I would like to thank all those who in some way contributed and supported me during the completion of my degree and this dissertation.

To Prof. Paulo Pinto, for giving me the honour of his advice, the availability and patience to answer all my questions and the assertive guidance towards the completion of this dissertation. It was very kind of him to accept being my supervisor under my exchange studies period.

To Prof. Jarmo Harju for believing in my capabilities and giving me the opportunity to work in the communications research group and for supporting with all the technical material needed. I am very grateful for the support.

To Researcher Bilhanan Silverajan for the valuable help during the development of this thesis. I would like to thank him for the patience in all this learning process and for all the hours spent helping and guiding me towards the completion of this dissertation.

In my office at the Department of Communications Engineering at TUT I was surrounded by knowledgeable and friendly people who helped me daily. I would like to express my gratitude to my office mates Jani Peltotalo and Joona Kannisto for being so nice and helpful.

To all my Portuguese friends in Finland especially Luis Sousa and Alberto Miranda who treated me as family and supported me from very closely.

I would like to thank all of my colleagues and close friends from FCT-UNL that gave me support since the beginning of my studies: Tiago Gaspar, Fábio Silva, Bruno Alves, Pedro Neves, David Gonçalves, Filipe Correia, João Melo and Diogo Figueiredo.

I would like to show the most kind and special gratitude to my family especially my parents Armando and Conceição, as well as my brother Ricardo, and my grandmother Maria Teresa, who have been a constant source of support during my graduation years and have always supported me during the most difficult times of my life. This thesis would certainly not have existed without them.

Last but not least, a special gratitude goes to my beloved girlfriend Emma for her patience, understanding and support, who shared time towards the completion of this thesis, and who always trusted in my abilities and encouraged me to follow my dreams without disappointment and fatigue.

Abstract

The DOORS framework is a versatile, lightweight message-based framework developed in ANSI C++. It builds upon research experience and subsequent knowledge garnered from the use and development of CVOPS and OVOPS, two well known protocol development frameworks that have obtained widespread acceptance and use in both the Finnish industry and academia. It conceptually resides between the operating system and the application, and provides a uniform development environment shielding the developer from operating system specific issues. It can be used for developing network services, ranging from simple socket-based systems, to protocol implementations, to CORBA-based applications and object-based gateways.

Originally, DOORS was conceived as a natural extension from the OVOPS framework to support generic event-based, distributed and client-server network applications. However, DOORS since then has evolved as a platform-level middleware solution for researching the provision of converged services to both packet-based and telecommunications networks, enterprise-level integration and interoperability in future networks, as well as studying application development, multi-casting and service discovery protocols in heterogeneous IPv6 networks.

In this thesis, two aspects of development work with DOORS take place. The first is the investigation of the *Network Address Translation* (NAT) traversal problem to give support to applications in the DOORS framework that are residing in private IP networks to interwork with those in public IP networks. For this matter this first part focuses on the development of a client in the DOORS framework for the *Session Traversal Utilities for NAT* (STUN) protocol, to be used for IP communications behind a NAT. The second

aspect involves secure communications. Application protocols in communication networks are easily intercepted and need security in various layers. For this matter the second part focuses on the investigation and development of a technique in the DOORS framework to support the *Transport Layer Security* (TLS) protocol, giving the ability to application protocols to rely on secure transport layer services.

Keywords: DOORS, framework, NAT, NAT traversal, STUN, security, secure communications, SSL, TLS.

Resumo

DOORS é um framework versátil e baseado em mensagens que foi desenvolvido em C++. É fruto da experiência de investigação e conhecimento obtida pelas plataformas de desenvolvimento CVOPS e OVOPS. Estas plataformas têm sido bastante aceites tanto na indústria como no campo de investigação Finlandesa. O DOORS reside entre o sistema operativo e a aplicação, fornecendo um ambiente de desenvolvimento uniforme protegendo o programador de questões específicas do sistema operativo. Pode ser usado tanto para o desenvolvimento de serviços de rede, que vão desde sistemas simples com sockets, implementação de protocolos, como para aplicações baseadas em CORBA e gateways baseados em objectos.

Originalmente o DOORS foi concebido como uma extensão natural da plataforma OVOPS para suportar aplicações de redes genéricas baseadas em eventos, distribuídas e cliente-servidor. No entanto, desde então, o DOORS tem evoluído como uma solução de middleware para investigar a prestação de serviços convergentes para as redes de pacotes e de telecomunicações, para a integração a nível empresarial e de interoperabilidade em redes futuras. Também se pretende que sirva para estudar o desenvolvimento de aplicações multi-casting e protocolos de descoberta de serviços em redes heterogéneas IPv6.

Esta tese cobre dois aspectos de desenvolvimento no framework DOORS. O primeiro é a investigação do problema da penetração em routers *Network Address Translation* (NAT), de modo a dar-se suporte às aplicações de redes IP privadas (para que possam interagir com as redes IP públicas). Para tal, desenvolveu-se um cliente no framework DOORS que suporta o protocolo *Session Traversal Utilities for NAT* (STUN), que pode ser utilizado para comunicações IP com máquinas protegidas por um NAT. O segundo aspecto envolve

comunicações seguras. Protocolos de aplicação em redes de comunicação são facilmente interceptados e precisam de segurança em várias camadas. Nesta segunda parte investigou-se e desenvolveu-se uma técnica para suportar o protocolo *Transport Layer Security* (TLS) no framework DOORS, permitindo que os protocolos de aplicação usem segurança na camada de transporte.

Palavras-chave: DOORS, framework, NAT, penetração em NATs, STUN, segurança, comunicações seguras, SSL, TLS.

Contents

List of Acronyms

AES	Advanced Encryption Standard
ANSI	American National Standards Institute
API	Application Programming Interface
CBC	CipherBlock Chaining
CFB	Cipher FeedBack
DES	Data Encryption Standard
DH	Diffie-Hellman
DH_anon	Anonymous Diffie-Hellman
DHCP	Dynamic Host Configuration Protocol
DHE	Ephemeral Diffie-Hellman
DOORS	Distributed Object OpeRationS
DoS	Denial of Service
DDoS	Distributed Denial of Service
DTLS	Datagram Transport Layer Security
ECB	Electronic Code Book
ECC	Elliptic Curve Cryptography
ECDHE	Elliptic curve Diffie-Hellman
FTP	File Transfer Protocol
FSM	Finite-State Machine
GCM	Galois/Counter Mode
GNU	GNU's Not Unix
HMAC	Hash-based Message Authentication Code
HTTP	Hypertext Transfer Protocol

ICE	Interactive Connectivity Establishment
IDEA	International Data Encryption Algorithm
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IP	Internet Protocol
IPsec	Internet Protocol Security
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
ISP	Internet Service Provider
LAN	Local Area Network
L2TP	Layer 2 Tunneling Protocol
MAC	Media Access Control
	Message Authentication Code
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OFB	Output FeedBack
P2P	Peer to Peer
PDU	Protocol Data Unit
PGP	Pretty Good Privacy
PKI	Public Key Infrastructure
PPTP	Point-to-Point Tunneling Protocol
RC	Rivest Cipher
RFC	Request for Comments
SAP	Service Access Point
SDU	Service Data Unit
SHA	Secure Hash Algorithm
SHS	Secure Hash Standard
SHTTP	Secure HyperText Transfer Protocol

SIP	Session Initiation Protocol
SSL	Secure Socket Layer
STL	Standard Template Library
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TUT	Tampere University of Technology
UDP	User Datagram Protocol
VoIP	Voice over IP
VPN	Virtual Private Network
XML	Extensible Markup Language
WAN	Wide Area Network
WCDMA	Wideband Code Division Multiple Access
WEP	Wired Equivalent Privacy
WiMAX	Worldwide Interoperability for Microwave Access
WLAN	Wireless Local Area Network
WWW	World Wide Web

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement and Motivation

The rapid development of computers and communications technology is allowing more and more devices to interact with each other. One of the most profound changes today is the increase in mobility of portable yet powerful wireless devices capable of communicating via several different kinds of wireless radio networks of varying link-level characteristics. Such wireless networks include 802.11-based Wi-Fi *Local Area Networks* (LANs), *Wideband Code Division Multiple Access* (WCDMA) cellular networks, and Bluetooth or Infrared based short range communications. Low-power energy efficient radio networks such as ZigBee are widely expected to play dominant roles in the medium to long-term future wireless mobility.

Accordingly to Silverajan [1], such increase of mobile networks raises the importance of taking into account the resilience and sustainability of the Internet of the future, and give special effort on the way applications and protocols today are being designed. Designing applications and protocols in the future requires a clean layered design while raising the level of abstraction to represent multiple applications in a device, multiple devices for a user and multiple users in a network.

Taking into account these needs, *Distributed Object OpeRationS* (DOORS) [2] is presented as a highly interoperable and lightweight event-based framework capable of serving these

needs while modelling, monitoring and handling events intrinsically present in communication architectures. Technically, DOORS is a C++ network programming framework developed for implementing protocols and network applications. The framework can also be used for developing network services, ranging from simple socket-based systems, to distributed applications, object-based gateways as well as general event-based client-server applications.

Therefore, developing protocols at the network and application layers must also be considered. In this thesis, two different aspects are discussed. The first is the investigation of the *Network Address Translation* (NAT) traversal problem to give support to applications in the DOORS framework that are residing in private IP networks to interwork with others in public IP networks. For this matter this first part focuses on developing client support in the DOORS framework for the *Session Traversal Utilities for NAT* (STUN) [3] protocol, to be used for IP communications behind a NAT. The second aspect involves secure communications. Application protocols in communication networks are easily intercepted and might need security in various layers. For this matter the second part focuses on the investigation and development of a technique in the DOORS framework to support the Transport Layer Security (TLS) [4] protocol, giving the ability to application protocols to rely on secure transport layer services.

1.2 Objectives and Contributions

This thesis focuses on two different aspects, both important for developing protocols in the DOORS framework.

The first aspect focuses on NAT traversal solutions. The presented solution is the implementation of the STUN protocol in the DOORS framework, a tool to be used for other protocols in the context of NAT traversal solutions.

The second aspect focuses on security in the network. Many applications interworking in a public network might need a security layer in order to ensure a secure communication. The goal is to study the vulnerabilities in a network (especially in the Transport Layer),

and give support to applications in the DOORS framework that need to ensure secure transactions. The presented solution is the integration of the TLS protocol in the DOORS framework, a protocol to be combined with other application protocols that might need to provide communication security over the Internet.

The main contributions of this dissertation are the implementations themselves, that are fully integrated in the DOORS framework and can be used in the future by other developers. The proposed models are meant to be easy to use and to be integrated with other protocols. The implementations are tested in order to study their operation and their interoperability.

1.3 Thesis Outline

The thesis is structured in seven chapters, including the introduction and the conclusions chapter.

Chapter 2 introduces a brief state-of-the-art description of the DOORS framework. It presents its architecture and some of its functionalities.

Chapter 3 introduces a brief state-of-the-art description for NAT. It presents the usage of NAT and the problems of its usage. Later on, it presents a useful protocol (STUN), a tool to provide NAT traversal solutions.

Chapter 4 introduces a brief state-of-the-art description for Security in the Network. It starts by presenting the most typical problems in different layers of a network, and later on presents useful solutions to provide security. Lastly, it presents the TLS protocol, that combines all the algorithms explained before to provide a sense of security for the problems described before.

Chapter 5 adds some considerations and introduces the design of models and the implementation for the following two protocols: STUN and TLS.

Chapter 6 shows the results and presents the analysis and validation of the proposed models.

Finally, chapter 7 presents the final conclusions and remarks of this thesis, as well as some future work perspectives.

Appendix A presents some technical information for the STUN implementation, and Appendix B presents technical information for the TLS implementation.

Chapter 2

The DOORS Framework

2.1 Introduction

Distributed Object OpeRationS (DOORS) [1] is an object-based software framework for designing distributed systems in heterogeneous network environments, especially on the Internet. It is a C++ network programming framework developed for implementing protocols and network applications. It gives the possibility to the programmer to write callback functions and invoke them by the event handlers. Besides, the framework gives a platform to the programmer and all the user events are mapped into DOORS messages. In terms of portability, DOORS is both portable and lightweight. The developer is not hindered from exercising any of the advanced C++ language-level features such as templates and exception handling, nor is prevented from using other frameworks and other libraries as part of the developed application. The DOORS is a single-threaded framework that can be compiled and used in the most of UNIX and Linux variants [5].

2.2 Requirements

Since DOORS is written in *American National Standards Institute* (ANSI) C++ language, first it requires an ANSI-compliant compiler supporting features of the language including *Standard Template Library* (STL), namespaces and template programming. Usually,

development platforms such as Solaris, FreeBSD, OpenBSD and Linux use GNU C++ compiler version 2.95.2 or later. Sun C++ compiler version 5 and later are used on Solaris version 8 and 9 operating systems.

The operating system must also support many functions that are typically used in UNIX systems such as *pipe()*, *select()*, *malloc()* and *socket()*. It must also support UNIX style *tty devices*.

DOORS makes use of GNU *autoconf* and *automake* to determine the current compilation environment in use. They are used during the build phase to create and configure a UNIX shell script which is then included in the released versions of DOORS and used in the target system to make several tests on the system. These tests consist of checking if the required C or C++ header files are available, compiling various C source files to discover characteristics of the compiler in use and to verify that the required external libraries exist in the system. After a successful check, the configure script creates *Makefile* files that are used by *make* command to build the DOORS system by calling the compiler with each source file and finally linking them to one library or an executable application [6].

2.3 Framework Architecture

The DOORS framework is an extensible object-oriented framework for network enabled application development. It is not just a set of libraries which can be utilised to implement interconnectivity, but also a complete application development environment featuring advanced Scheduling, I/O Handling, and automatic code generation.

DOORS conceptually resides between the operating system (Unix based) and the application developed by the programmer. The developer is shielded from the operation system specific issues and is provided a uniform development environment. All the user events are mapped into DOORS messages. When a developer wants to implement an application, DOORS reduces this event-driven application into a set of interacting Tasks, Ports and DOORS Messages. Tasks communicate with each other using Ports and passing messages through the connected Ports. Each Message passed to the Task such as user input, incom-

ing protocol packet or time-outs represents a event. The result of any application and its interaction with DOORS is largely asynchronous and message-based. Figure 2.1 depicts how the various modules, subsystems and tools of the framework, together with the machine's operating system and external libraries, can be architecturally represented.

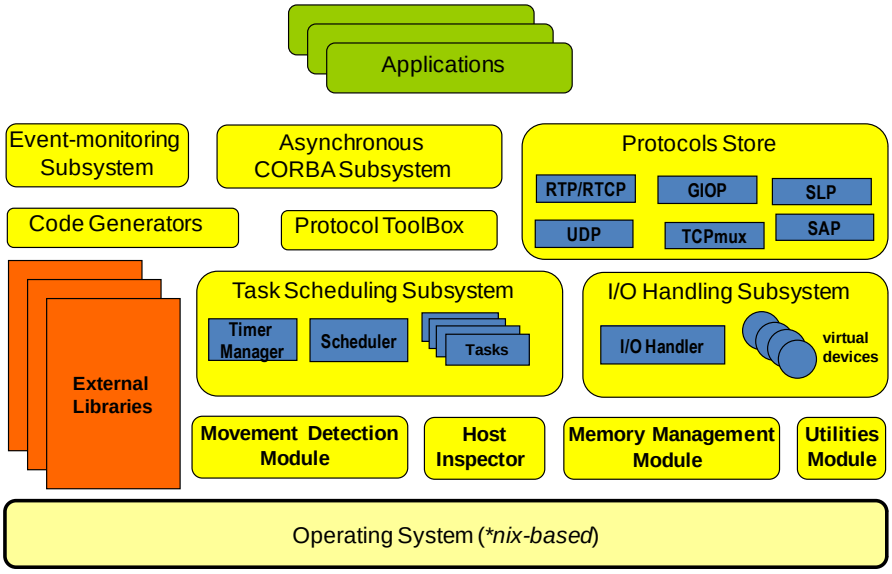


Figure 2.1: DOORS architecture[1]

The Protocols Store contains several existing protocols already implemented, and can be directly used by the developer to accompany new applications, such as UDP, TCPmux, RTP and SLP.

The Protocol Toolbox contains specialized Protocol Tasks for aiding protocol development, finite state machine implementations, Service Access Points, and encoding/decoding of Protocol Data Units, bi-directional Ports used by Tasks for message passing and multiplexers that allow M:N communications between Tasks.

The Event-monitoring Subsystem consists of tools used at runtime to monitor various parts of a running system such as the state of tasks, internal variables, the number of messages processed, their types, timers as well as the scheduler load. The Subsystem contains a local event monitor capable of inspecting intra-process events in a locally running application, a distributed event monitor capable of inspecting events simultaneously in multiple DOORS based applications running in a network, tools for logging events into files as well as

[1]Reprinted from Silverajan et. al [1]

a hierarchically organized symbol handling interface through which the event monitors communicate with the application.

In addition, the Utilities and Memory Management Modules provide basic support for implementation and runtime memory management. The Utilities Module offers basic datatypes, structures and class templates. Frame and cell classes provide flexible ways in storing and manipulating large octet arrays. The Memory Management Module offers advanced memory management featuring basic, statistical and block memory allocations.

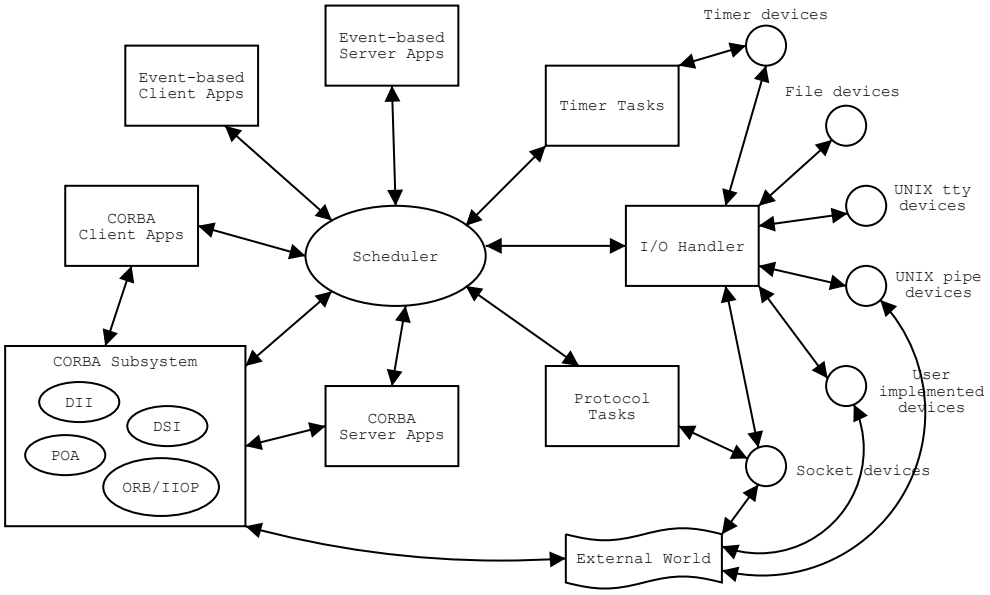


Figure 2.2: DOORS communication model[2]

Consequently, using this architecture it is possible to develop systems using DOORS to communicate at multiple levels simultaneously, possessing the capacity of harnessing interprocess communication, virtual devices, protocols and the asynchronous CORBA system to interact with many types of external programs. Figure 2.2 illustrates the communication model of DOORS [5]. A DOORS-process runs in a single UNIX-process with its own scheduler and tasks that are controlled by a scheduler. A DOORS-process is a collection of DOORS tasks and a scheduler running on one UNIX-process. A DOORS application refers to a distributed application that may contain more than one DOORS-processes in communication to each other. Some interesting modules of these subsystems and their

[2]Reprinted from Karvinen [6]

connections are described in detail in the following subsections.

DOORS contains several libraries and applications. Figure 2.3 shows the inheritance of some of the C++ base classes in DOORS. A task in DOORS is an object that is represented by the *Otask* base class and its derivations. *Otask* defines an abstract method run() which is inherited and implemented by its descendants. That function is called by a scheduler for execution of the task's own routines. *Address* is a base class for all the address type classes in DOORS. It is just shown in the figure as an example of DOORS types. It is specialised to *InetAddr* which is an address handler for *Internet Protocol version 4* (IPv4) addresses and *InetAddr6* for *Internet Protocol version 6* (IPv6) [6].

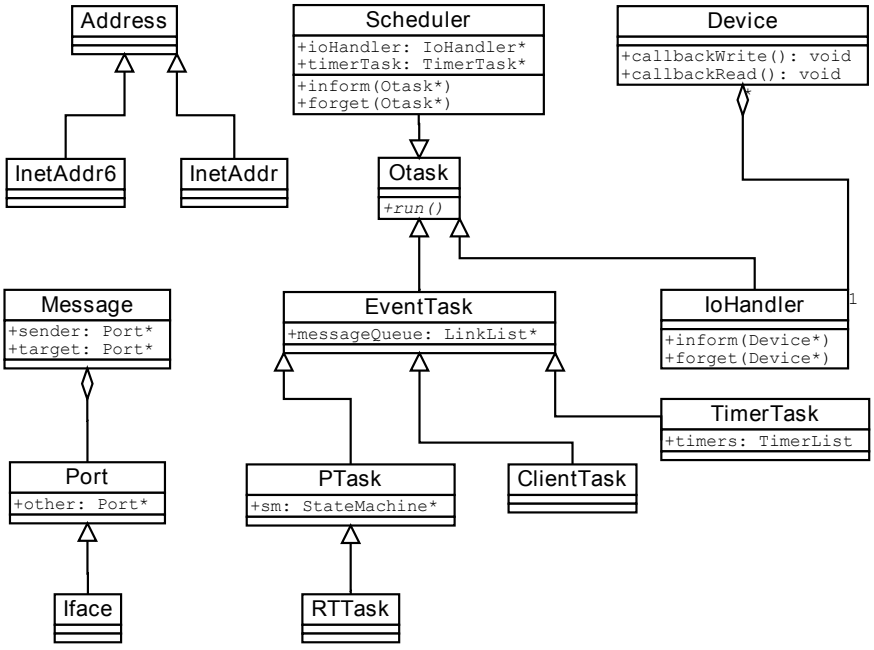


Figure 2.3: Examples of some of the DOORS classes[3]

2.3.1 Scheduler

The *Scheduler* is part of the Scheduling subsystem and is a task which is derived from *Otask* but it is specialised in allocating execution turns to other tasks. Various tasks might have calls to execute at the same time. In order to execute them in DOORS, a Task needs to ask for a execution turn to the Scheduler. In a typical DOORS application, it is called in a loop from the *main()* function. The Scheduler class is an implementation of Round-Robin

[3]Reprinted from Karvinen [6]

Scheduler and it is currently the only scheduler algorithm available within DOORS. The Scheduler maintains a list of the tasks that are being scheduled. The methods *inform()* and *forget()* are used to add a task for scheduling and to remove it, respectively [6].

2.3.2 I/O Handler and devices

It is essential for a protocol implementation framework to provide access to the device's interfaces and hardware configurations, as well as to have a flexible yet efficient input/output handling system. The responsibility of such service is handled by the I/O Handling subsystem.

The I/O Handling subsystem is directly responsible for allowing event-driven tasks and applications to communicate with external entities by introducing the concept of a *Virtual device*. Such virtual devices currently include IPv4 and IPv6 network devices that perform both unicast and multicast communication, and stream I/O facilities such as files and pipes. Virtual devices possesses identifiers represented by file descriptors and perform a uniform interface to tasks using functions such as *open()*, *close()*, *read()* and *write()*. However data transfer is asynchronous. Hence when writing to a device, the data is stored in a buffer until the execution turn is obtained from the *Scheduler*, and a callback function is invoked to perform the actual bytewise transmission.

Therefore, the Handling Subsystem relies on a *I/O Handler*. The I/O Handler is derived from *Otask* and it is responsible to manage the devices required for communication. It is responsible to obtain execution turns from the Scheduler and ensure callback routines from the devices to handle events[1].

2.3.3 Tasks, Messages and Ports

Ports provide a way to connect DOORS Tasks to each other. It has the methods *putMessage()* and *getMessage()* for putting and getting a Message object in a port for transition. Figure 2.4 illustrates how to send a message between DOORS tasks.

In practice, the message object is instantiated from a specialised *Message* class. That

specialised class contains attributes for the data being delivered through the ports. For simplicity, this is not shown in the Figure. *EventTask* is derived from *Task* and it supports message queues and connecting to another *EventTask* or its descendant via a *Port*.

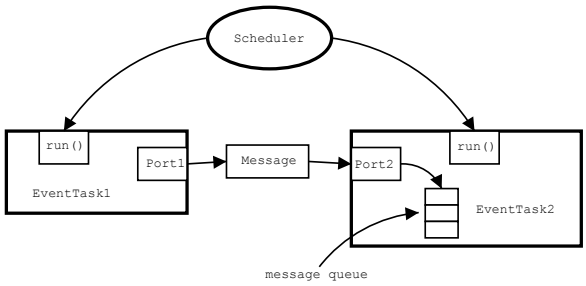


Figure 2.4: Sending and asynchronous message in DOORS[4]

Tasks in general communicate with each other using *Ports* and passing messages through the connected *Ports*. The connection is established using *connect()* method provided by *Port*. The *Scheduler* calls the run() method of a *EventTask* to make possible the execution of each *Task*. *PTask* is a descendant of *EventTask* and it is specialised to tasks featuring a finite state machine [6].

2.3.4 XML code generators

Implementing a protocol with a computer language typically follows the same pattern of writing encoders and decoders for messages and because even the simplest protocols feature a state machine, a programmer must also write a state machine in his protocol implementation. Using a code generator which reads generalised representation of the protocol messages and its state machine and creates skeleton code for the programmer to use in his implementation speeds up the coding work.

The Code Generators are used to compile XML code parsed by the programmer. DOORS provides three code generators:

- *dpeerg*: Responsible for defining *Protocol Data Unit* PDU's between protocol peers;
- *dsapg*: Responsible for defining *Service Access Point* (SAP);
- *dsmg*: Responsible for defining finite state machines.

[4]Reprinted from Karvinen [6]

XML specifications are then parsed by generators in DOORS at compile time to produce corresponding framework specific C++ classes such as Messages, Service Access Points, Service Primitives as well as extended finite state machines. The example of the figure 2.5 provides a rough idea of a XML specification for the Protocol Data Units (PDUs) of the *Real-time Transport Protocol* (RTP) in DOORS.

```
<Peer Name="RtpPeer">
  <Message Name="DATA">
        <Field> InetAddr srcIP </Field>
        <Field> InetAddr destIP </Field>
        <Field> Uint8 version </Field>
        <Field> Uint8 padding </Field>
        <Field> Uint8 extension </Field>
        <Field> Uint8 csrc_count </Field>
        <Field> Uint8 marker </Field>
        <Field> Uint8 payload_type </Field>
        <Field> Uint16 sequence_number </Field>
        <Field> Uint32 timestamp </Field>
        <Field> Uint32 ssrc </Field>
        <Field> Frame payload </Field>
  </Message>
</Peer>
```

Figure 2.5: PDU definiton example

2.3.5 Local Event Monitor (LEMon)

As an example on how to monitor a DOORS-process, this section describes the *Local Event Monitor* (LEMon). It was originally developed for OVOPS in Lappeenranta University of Technology as the *Textual Protocol Tracer* (TPT) but was renamed to follow the naming conventions in DOORS. Initially, it was not included in DOORS but was later adopted to DOORS because at the time there were no monitoring tools available in DOORS. LEMon is also useful as a protocol programming teaching and debugging tool.

LEMon provides a text-based debugging environment for tracing events and also for sending user-defined events to a process. The user is not required to make changes in his code during testing but makes it possible to try various execution scenarios using the user interface and textual representation of the messages [6].

Chapter 3

Network Address Translation (NAT)

3.1 Introduction

Internet as a global and huge network has been growing exponentially. As millions of new users and devices have been connecting every day, lack of Internet addresses has becoming an increasingly demanding challenge. Traditionally, home and business customers connect to the Internet with dial-up connections using an *Internet Service Provider* (ISP) that dynamically assign *Internet Protocol* (IP) addresses that should be unique on the Internet. Originally IP addresses were defined by IPv4 protocol. However, the number of addresses provided by IPv4 is limited. That was one of the reasons why a complete redesign of the protocol was started, known as IPv6. However, in order to respond to the shortage of IPv4 addresses in the short-term, *Network Address Translation* (NAT) was proposed [7, pp. 444–448], which is described in RFC 3022 [8].

The basic idea is to let a NAT-based router rewrite the address information in outgoing and incoming messages. By changing the source private IP address and port number to public ones for outgoing connection requests, NAT creates a mapping to allow the return packets to reach the original initiator, and in this case, NAT is "transparent"to both endpoints.

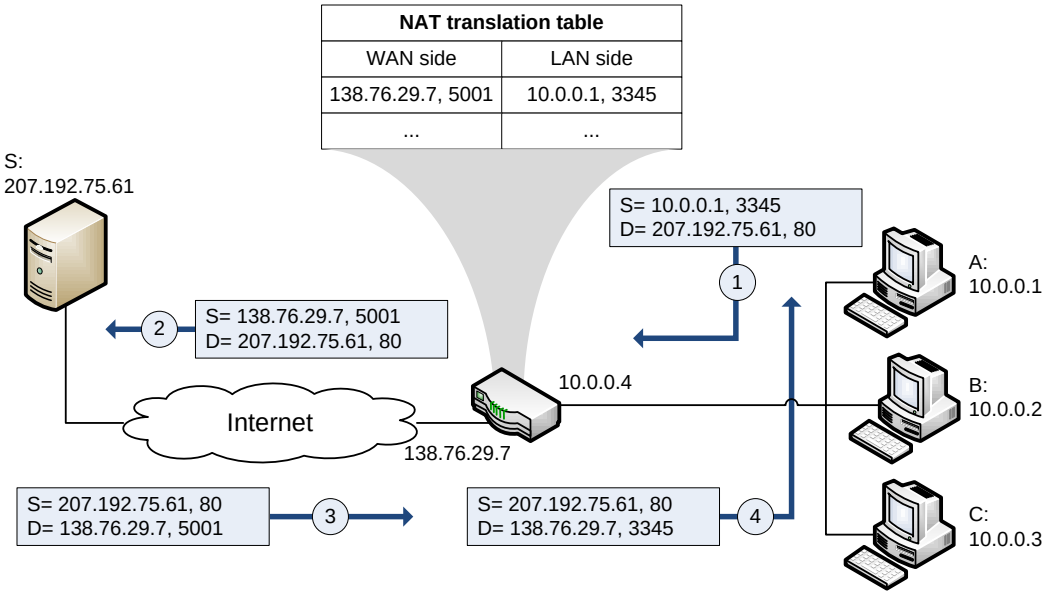


Figure 3.1: Network address translation[1]

A NAT router residing at home, can be used as a gateway connection to the Internet, having at least one network interface connected to the internal network and at least one network interface connected to the Internet (possessing a routable IP address), in order to connect all the machines to the Internet. Therefore, to multiple hosts in the same network, NAT provides different private IP addresses, and a single public IP address.

Figure 3.1 is one example, and shows the operation of a NAT-enabled router. Hosts A, B and C are in the same home network and connected to a router. The router is a NAT-enabled router, and provides for all the computers different private IP addresses, respectively 10.0.0.1, 10.0.0.2, 10.0.0.3, and provides a single public IP address 138.76.29.7. Therefore, the NAT-enabled router behaves to the outside world as a single device with single public IP address, and hides the details of the home network. This is possible due to the fact that the NAT-enabled router maintains a NAT translation table with IP addresses in the table entries including port numbers.

Usually the NAT-enabled routers use *Dynamic Host Configuration Protocol* (DHCP) to provide addresses in the home network. Considering the example of the figure 3.1, the host A: 10.0.0.1 wishes to reach Internet, and opens a browser to read the news in the

[1]Adapted from Ross et. al [9]

Server S: 207.192.75.61. The host A: 10.0.0.1 assigns the (arbitrary) source port 3345 locally and sends the packet into the LAN (transaction 1). The NAT router receives the IP packet, checks the source port and destination, and generates a new source port 5001 for the packet, creating a new entry in the NAT translation table. Then, the NAT router replaces the source IP address and port with its WAN-side address 138.76.29.7 and the generated source port 5001 and sends the packet into the Internet to reach the destination server (transaction 2). The Web server completely unaware of these changes, receives the packet and reads the source IP address and port, sending a response back to the router (transaction 3). The NAT router gets the response, and checks its NAT translation table to compare the destination port number 5001 of the packet to the entries in the table. Obtained the appropriate IP address 10.0.0.1 and destination port number 3345, the NAT router replaces the destination port and IP address and route the packet to the host A (transaction 4) [9].

3.2 NAT Traversal

Problems arise when a peer wants to initiate a new communication with another peer that is connected to the Internet via NAT. Without further arrangements, the NAT-based router will deny this communication request and will drop incoming messages. Thus, the peer is unreachable from other peers [10]. For example, services that require *Transmission Control Protocol* (TCP) connections initiated from outside of the NAT (public IP addresses), or stateless protocols such as those using *User Datagram Protocol* (UDP), can be disrupted and the functionality of the protocol may be compromised. Developed services following the *Peer-to-peer* (P2P) paradigm such as file sharing, instant messaging and *Voice over IP* (VoIP) are examples of applications that suffer from the existence of NAT.

One possible solution for this problem is to let users configure the router manually, changing the NAT tables. However, this is very inconvenient for the user and requires administration rights. Then, to allow incoming messages to reach the peer, a so-called NAT traversal approach is required.

NAT traversal techniques are typically required for clients that need to start a protocol and the communication is affected by the NAT-based routers. Since NAT behaviour is not standardized, many techniques of NAT traversal exist, but just one can be applied [11]. In the next section 3.3, the *Session Traversal Utilities for NAT* (STUN) protocol is described as a tool to be used in the context of NAT Traversal solutions.

3.3 Session Traversal Utilities for NAT (STUN)

As explained in section 3.2, NAT traversal techniques are required for clients that need to start a specific protocol that is affected by the NAT-based routers. In the original specification *Simple Traversal of User Datagram Protocol (UDP) through Network Address Translators (NATs)* (STUN) described in the RFC 3489 [12], sometimes called "classic STUN", was proposed as a complete NAT traversal technique. However, experience since the publication of RFC 3489 has found that classic STUN simply does not work sufficiently well to be a deployable solution. Therefore, a specification of an updated set of methods published in RFC 5389 [3], *Session Traversal Utilities for NAT* (STUN) was proposed, retaining the same acronym but with some differences. STUN is not a NAT traversal solution by itself. Rather, just a tool to be used in the context of NAT traversal. This is an important change from the previous specification (RFC 3489) that is now deprecated. In the next sections, the STUN protocol is explained.

3.3.1 Protocol analysis

Many NAT traversal techniques require assistance from a computer server located in the public Internet. *Session Traversal Utilities for NAT* (STUN) is a light-weight client-server protocol that appeared as a tool to be used in the context of one or more NAT traversal solutions. *Interactive Connectivity Establishment* (ICE) [13], *Traversal Using Relay NAT* (TURN) [14], Client-Initiated Connections in the *Session Initiation Protocol* (SIP) [15] and NAT Behavior Discovery Using Session *Traversal Utilities for NAT* (STUN) [16] are four defined usages of STUN at the time. Other STUN usages may be defined in the future.

A STUN server can be located in the public Internet in order to give useful information to the clients behind a NAT-router and allow the private IP networks behind a NAT to interwork with public IP networks. Therefore, the STUN server allows a client to find out its public address/port pair that they are associated in the NAT-router. A possible configuration is shown in the figure 3.2.

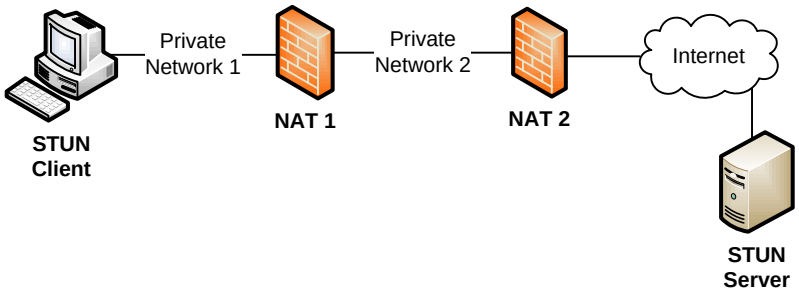


Figure 3.2: Possible STUN configuration

Typically hosts are connected to the internet through 1 or more NAT routers. In this configuration two main entities (called STUN agents) exist, a STUN client and a STUN server. The lower agent STUN client is connected to a private network 1. This network is then connected to a private network 2 through NAT 1, and consequently private network 2 connects to the Internet through NAT 2. The upper agent resides in the public Internet and it is the STUN server.

The STUN Client is the entity that starts the protocol. It sends STUN requests to the STUN Server, and receives STUN responses. The STUN Server is the entity responsible to give to the client the information that it requests. In this case, the Server receives the STUN requests and sends the STUN responses.

Figure 3.3 is one example of a STUN protocol communication and shows the operation of the STUN protocol using the configuration of the figure 3.2. The client A is in a private network 1. This network is then connected to a private network 2 through NAT 1, and consequently connected to the Internet through NAT 2. In the STUN protocol communication, the client A is the lower agent STUN client. The client A with private IP address 10.0.0.1 needs to find its external IP/Port in order to send this information to the client B. Then, it sends a STUN message (1) to the STUN server asking “What is my

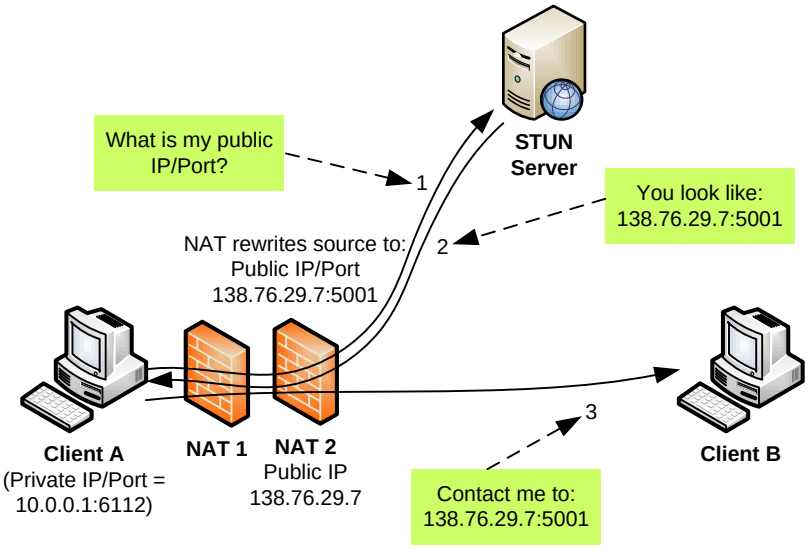


Figure 3.3: Use STUN to find external IP/Port

public IP/Port?". The message passes through NAT 1 and NAT 2 and reaches the STUN server with the source IP/Port from the NAT 2 (138.76.29.7:5001). The STUN server reads the message, writes the reflexive IP/Port in the message (2) and sends it back to the source IP/Port that corresponds to the NAT 2. At this point NAT 2 receives the message 2 and sends it to the NAT 1, and NAT 1 sends it to the client A. Now client A knows its own public IP/Port used to external communications and sends a message (3) to the client B with its own information. Now the client B knows how to contact the client A, and it can combine this information with many NAT traversal techniques to contact the client A.

3.3.2 Protocol operation

In detail, STUN is a client-server protocol and supports two types of transactions: request/response transactions and indication transactions. Request/response transactions are the ones in which a client sends a request to a server, and the server returns a response. Indication transactions are the ones in which either agent client or server sends an indication that generates no responses.

All STUN messages must start with a fixed header that includes a method, class and a transaction ID. The transaction ID is a 96-bit randomly selected number used to identify

each transaction, helping the client to associate the server responses to the requests generated. The class indicates the type of the message, which may be request, success response, error response or indication. The method indicates which of the various requests or indications it refers to. Currently binding is the only method defined, but other methods are expected to be defined in future documents. The RFC 5389 only defines a single method binding, which is used either in request/response transactions or in indication transactions. The binding method when used in request/response transactions can be used to determine the particular "binding"a NAT has allocated to the STUN client. When used in either request/responses or indication transactions, the binding method can also be used to keep these "bindings"alive [3].

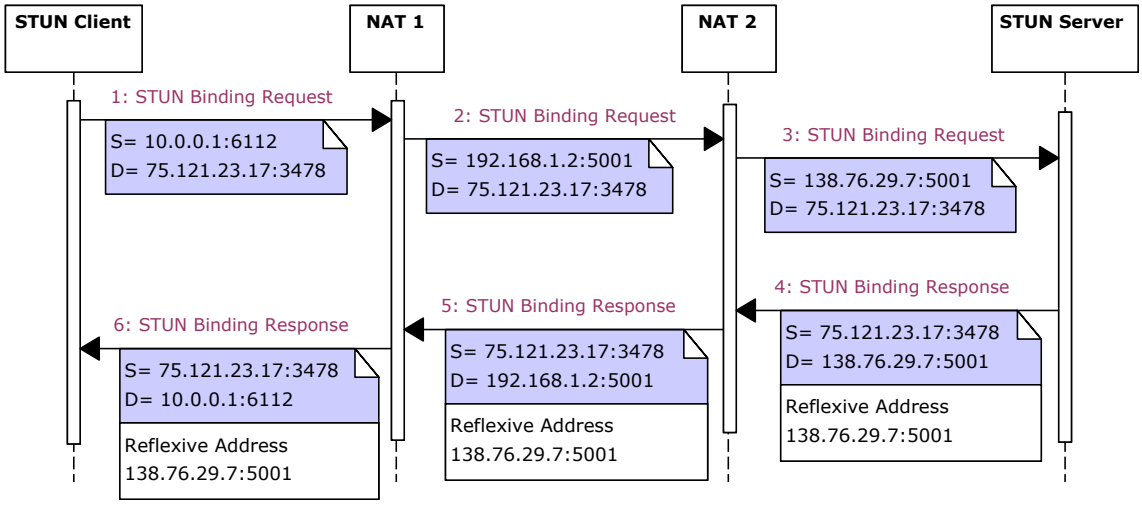


Figure 3.4: STUN Client-Server diagram

Typically, a STUN binding request/response is based on a scheme illustrated in the figure 3.2. The agent STUN client sends a STUN binding request message to the STUN server and it might pass through one or more NATs between the two agents. In this example two NATs are shown. When the STUN binding request passes through each NAT, the NAT modifies the source transport address of the packet. As a result, the source transport address of the packet received by the STUN server corresponds to the public IP address and port created by the closest NAT to the server. This is called transport reflexive address. Then, the STUN server copies the source transport address and makes it an attribute of the body for the STUN binding response packet. This response packet passes back through the two NATs that modify the destination address of the packet, but the

reflexive address in the body remains untouched. In this way, the client can learn its reflexive transport address.

STUN usually operates on a *User Datagram Protocol* (UDP). When running a STUN over UDP it is possible that some STUN messages are dropped by the network. Reliability of STUN request/response can be accomplished with retransmissions by the client that should retransmit request messages. However, UDP does not provide reliable transport guarantees. Other way is to run STUN over *Transmission Control Protocol* (TCP). When reliability is mandatory TCP may be used in order to provide a connection between the client and the server. Besides, in order to provide security, *Transport Layer Security* (TLS) may be used to accompany the TCP. For this, the STUN messages may be transported and encrypted via TCP/TLS.

3.3.3 STUN Message Structure

As explained in chapter 1, this thesis focus is a UDP implementation of the STUN protocol. Therefore, this section focuses in the specific information to reach the required specifications.

Typically in the *Internet Protocol* (IP), the messages are encoded in binary using the standard network bite order (also known as big-endian, in the sense that the most significant byte or octet is sent first). The transmission order is described in detail in Appendix B of RFC 791 [17].

STUN messages consist of one header field followed by data (attributes) fields. Therefore, the STUN messages must start with the same header format. A 20-byte header followed by zero or more attributes. Then, the STUN header contains a STUN message type, message length, magic cookie and transaction id.

Figure 3.5 illustrates the structure of the STUN header. The most significant 2 bits must be zeroes. This is used to differentiate the STUN messages from the messages of other protocols. The header contains the following fields:

- **STUN Message Type:** A 14-bit STUN Message Type defines the message class

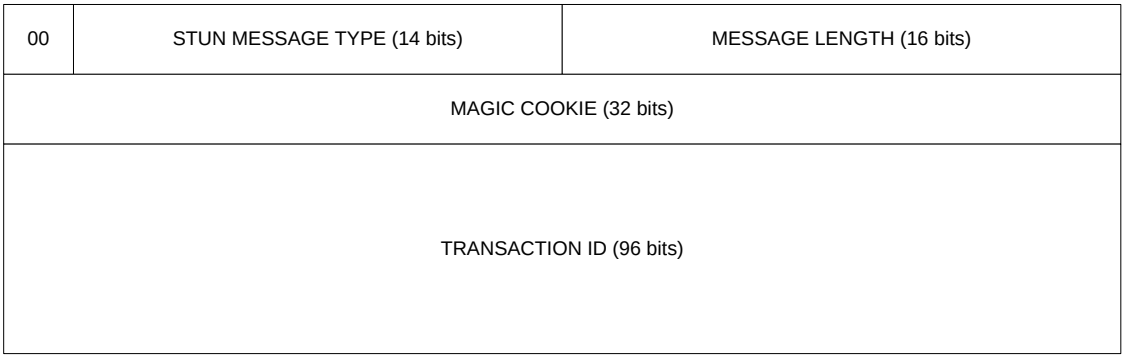


Figure 3.5: Format of STUN message header

M 11	M 10	M 9	M 8	M 7	C 1	M 6	M 5	M 4	C 0	M 3	M 2	M 1	M 0

Figure 3.6: Format of STUN Message Type field

(request, success response, error response or indication) and the message method (this specification defines a single method, binding). Although there are four message classes, there are only two types of transactions: request/response transactions (which consist of a request message and a response message) and an indication (which consists of a single indication message). Requests and indications have just one form, although the responses are split into two forms, success messages and error messages. The structure of the STUN Message Type is illustrated in figure 3.6. In the message type field, the bits are order from the most significant (M11) until the least significant (M0). The 12-bit M11 to M0 represent the encoding of the method. The C1 and C0 represent the 2-bit encoding of the class. The class 0b00 is defined as a request, the class 0b01 is an indication, the 0b10 is the success response, and the 0b11 is a error response. However, this specification defines a single method, binding, so it takes the value 0b000000000001. The bits of the method and the class may be disposed as illustrated in figure 3.6.

- **Message Length:** These 16 bits must contain the size, in bytes, of the message body (not including the 20-byte header).
- **Magic Cookie:** These 16 bits must contain the fixed numeric value 0x2112A442. This field was part of the transaction ID in the older specification RFC 3489. Now is placed in this location and it allows the server to detect if the client will understand

certain new attributes that were added in the new specification RFC 5389.

- **Transaction ID:** The transaction ID is a 92-bit randomly selected number by the initiator of the communication. It is used uniquely to identify STUN transactions. For request/response transactions it is randomly selected by the client, and echoed by the Server. For indications it is chosen by the agent that is sending the indication. This field is very important. It primarily serves to correlate requests with responses, but it also improves the security and helps to prevent certain types of attacks [3].

The details of the attributes are given in the next section.

3.3.4 STUN attributes

As explained before, the STUN messages contain a header and a data field which must contain attributes. Each attribute must be TLV (Type-Length-Value) encoded with 16-bit Type, 16-bit Length and a Value with variable length.

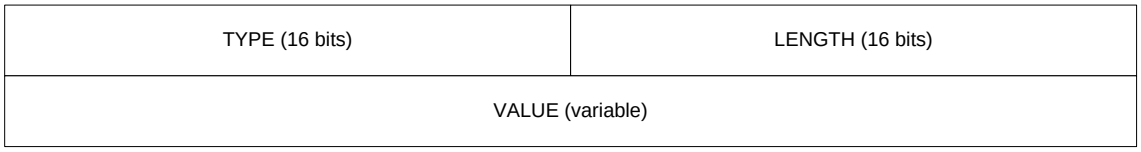


Figure 3.7: Format of STUN Attributes

The format of the STUN attributes is illustrated in figure 3.7. Each attribute contains the following three fields:

- **Type:** The Type field specifies the type of the attribute. It must be a hex number in the range 0x0000 - 0xFFFF. Values between 0x0000 and 0x7FFF are compression-required attributes, which means that these attributes have to be understandable by the agent and cannot successfully process the message unless it understands the attribute. Values between 0x8000 and 0xFFFF are compression-optional attributes, which means that these attributes can be ignored by the agent if it does not understand them. The most common attributes, those that will be used in this project are: 0x0009f for Error-Code, 0x000A for Unkown-Attributes and 0x0020 for XOR-Mapped-Address.

- **Length:** The Length field is a 16-bit numeric value and must contain the length, in bytes, of the attribute which will be posted in the Value field (not including the 32-bit Type plus Length).
- **Value:** The Value field must contain the attribute specified in the Type field and it has variable length.

The next sections describe in detail the format of the most common attributes used in this project.

XOR-Mapped-Address

The XOR-Mapped-Address attribute contains the reflexive transport address, and it is obfuscated through the XOR function. Consequently the former (STUN server) encodes the transport address by making a XOR operation with the Magic Cookie contained in the header.

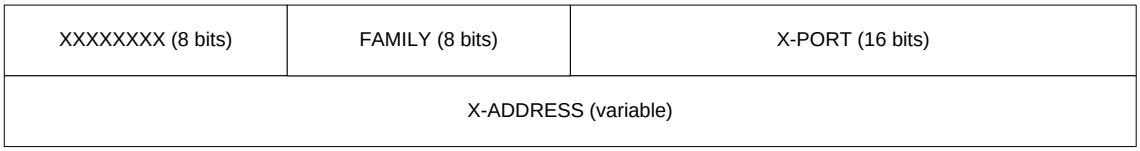


Figure 3.8: Format of XOR-Mapped-Address Attribute

The format of the XOR-Mapped-Address attribute is illustrated in figure 3.8. The most significant 8 bits specified in the older RFC 3489 [12] were zeroes, and the address was defined simply as Mapped-Address and was not encoded. However, deployment experience found that some NATs rewrite the 32-bit binary payloads (IPv4 addresses), containing the NATs public address (Mapped-Address attribute). Such behaviour was interfering the operation of STUN [3]. Now in the new RFC [3] the address is obfuscated through the XOR function, which means that the most significant 8 bits must be unknown value X (do not care) and must be ignored by the receivers. This attribute also contains the following fields:

- **Family:** The Family field is an 8-bit numeric value which represents the IP address family of the X-Address field: 0x01 for IPv4 and 0x02 for IPv6.

- **X-Port:** The X-Port field is an 16-bit numeric value which represents the transport mapped port received in the request. This field must be computed making the XOR operation with the most significant 16 bits of the Magic Cookie.
- **X-Address:** The X-Address is the transport mapped address received in the request. If the IP family is IPv4, the result should be computed by taking the mapped IP address in host byte order and making the XOR operation with the Magic Cookie of the header. If the IP family is IPv6, the result should be computed by taking the mapped IP address in host byte order and making the XOR operation with the concatenation of the MAGIC COOKIE and the TRANSACTION ID of the header. The length depends of the family, so it has a variable length.

Error-Code

The Error-Code attribute is used to identify an error response message. It usually contains a numeric error code between 300 and 699 plus a textual reason phrase encoded in UTF-8 defined in the RFC 3629 [18] whose semantics are consistent with SIP [19] and HTTP [20].

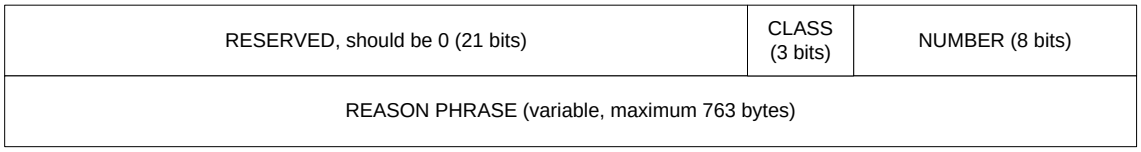


Figure 3.9: Format of Error-Code Attribute

The format of the Error-Code attribute is illustrated in figure 3.9. The most significant 21 bits should be zeroes and are for alignment on 32-bit boundaries and receivers must ignore these bits. The Class and the Number are encoded separately, just to make the processing easier. Therefore, this attribute contains the following fields:

- **Class:** The Class field is a 3-bit numeric value which contains the hundredth digit of the error code and must be between 3 and 6.
- **Number:** The Number field is a 8-bit numeric value which contains the last 2 digits, tens and units of the error code.

- **Reason Phrase:** The Reason Phrase field is encoded in UTF-8 and is a textual phrase which has a brief description of the error. It has variable length, depending of the extension of the phrase.

As mentioned above, it is recommended to use phrases defined in other protocols, such as HTTP and SIP. The most common errors are defined below and the reason phrase is defined by the sender:

- 300: "Try Alternate ". The client should contact an alternate server for this request.
- 400: "Bad Request". The request was malformed.
- 401: "Unauthorized". The request did not contain the correct credentials to proceed. The client should retry later with proper credentials.
- 420: "Unknown Attribute". The attribute contained in the message is not understandable by the server.
- 500: "Server Error". The server has suffered a temporary error, and the client should try again later.

Unknown-Attribute

The Unknown-Attributes attribute is only used in the presence of an Error-Code 420. This attribute must contain a list of 16-bit attribute types that were not understood by the server. The format of this list is illustrated in figure 3.10.

ATTRIBUTE 1 TYPE (16 bits)	ATTRIBUTE 2 TYPE (16 bits)
ATTRIBUTE 3 TYPE (16 bits)	. . . . .

Figure 3.10: Format of Unknown-Attributes Attribute

Software

The Software attribute is an attribute with variable length. It can be sent by clients and servers, and must contain a textual description of the software that is used by the

agent that sends the STUN message. Typically the attribute has no impact in the main operation of the protocol and is used as a tool for diagnosis and debugging purposes. It must contain a UTF-8 encoded textual message smaller than 128 characters (which can be a maximum of 763 bytes).

Chapter 4

Network Security

4.1 Introduction

In the early nineteenth century it took weeks and weeks for government agencies to send a message to different cities. In the late nineteenth century the telegraph cut this time and only took a few hours. At the dawn of the twenty and twenty-first century, the time has been cut to a fraction of second with the telephone and Internet, and not just governmental institutions but most of the citizens and companies can easily communicate with each other. The result is that we now conduct more our communications, whether personal, business, or civic, via electronic channels. The availability of such easy communications has transformed not just the internationalization of the commerce, but also governmental and personal relations in remote parts of the world.

These developments in technology have also had a profound impact in terms of privacy. Today, most people use telephone (including cellphones) daily, and a constant use of electronic mail, electronic commerce and much more services on the World Wide Web. However, the reality is that the Internet and the web are extremely vulnerable and can compromise communications, that by their essence can be naturally intercepted with different methods [21].

This chapter introduces the web security and its considerations, cryptography, and the *Transport Layer Security* (TLS) protocol, providing the essential information to under-

stand the implementation of a secure channel between two entities. It begins with a very brief look at Network security and electronic commerce, followed by cryptography focusing on the issues that led to the creation of the *Secure Socket Layer* (SSL) protocol. The next section contains a brief description of the SSL protocol and its transformation into TLS.

4.2 Approaches to Network Security

The rapid development of the networks and the communications naturally led to business companies, government agencies, and many individuals to easily access the World Wide Web, and the usage of graphical Web browsers became popular. They become responsible for a big part of the traffic on the Internet, and privacy issues become more serious. Unfortunately the reality is that the Internet and the Web are extremely vulnerable, and communications can be easily intercepted. Business, governmental agencies and individuals woke up to this reality, and the demand for secure Web services grows [21].

Accordingly to Oppliger [22] in the first half of the 1990s, people started to purchase items electronically. As an example, among the electronic payment systems, credit card transactions are the most widely used. Thus, providing Web transactions security, is nowadays a demanding challenge and extremely important. The greatest common dominator of all these possibilities was the use of cryptographic techniques. However, for the TCP/IP stack, there was hardly any consensus about which cryptographic techniques to use and what layer to apply them. The remaining paragraphs of this section are based on the chapter 3 of *SSL and TLS: Theory and Practice* by Rolf Oppliger [22].

After analysis and discussions, many security protocols were proposed. There are many possibilities to invoke cryptographic techniques at various layers of the TCP/IP protocol stack, in order to secure web transactions. Several Internet security protocols and their placement in the TCP/IP stack are illustrated in figure 4.1 and explained below:

- **Network Access Layer:** On the Network Access Layer (often called Link Layer on RFC 1122 [23]), IEEE 802.1AE [24] (also known as MACsec), defines data origin

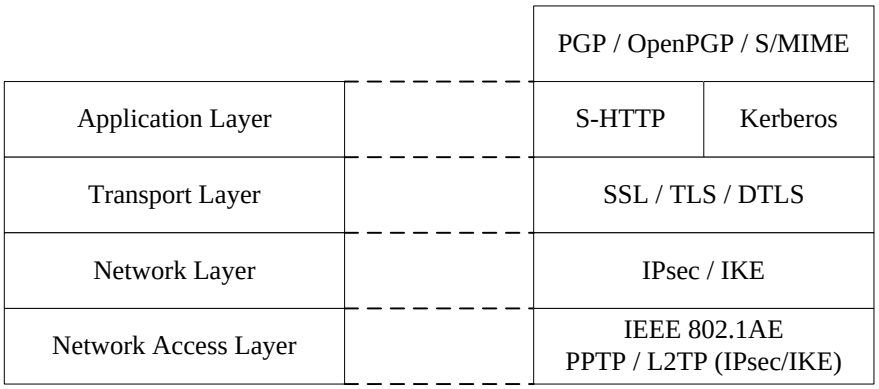


Figure 4.1: The Internet security protocols in the TCP/IP stack[1]

authentication, connectionless confidentiality and integrity services to MAC frames. Also, there are several virtual private networking (VPN) technologies and protocols, such as the *Point-to-Point Tunneling Protocol* (PPTP) or the *Layer 2 Tunneling Protocol* (L2TP) combined with IPsec/IKE, to provide users secure access to their organization's network (see next bullet). These protocols can also be used to establish secure connections in web transactions.

- **Network Layer:** On the Network Layer (often called Internet Layer on RFC 1122 [23]), to establish secure connections between two IP entities, there are *Internet Protocol Security* (IPsec) and *Internet Key Exchange* (IKE) [25]. These protocols can be used to establish secure connections in web transactions. The nice thing about applying security at this level is that it co-resides with IP, and hence all Internet applications are layered on top of them. Thus, it can be used to secure all the Internet connections. However, the minus side about IPsec/IKE is that protocols are overly complex and this makes the deployment and operation of IPsec/IKE quite tricky.

- **Transport Layer:** On the Transport Layer, the goal is to enhance the current transport protocols and invoke cryptographic techniques to provide secure communications and basic security services for web transactions. SSL/TLS is the choice to enhance *Transmission Control Protocol* (TCP) on which protocols such as HTTP, SMTP and FTP are layered. *Datagram Transport Layer Security* (DTLS) is the

[1] Adapted from Oppliger [22]

choice to provide security over *User Datagram Protocol* (UDP) connections.

- **Application Layer:** On the Application Layer, the goal is to provide security enhancing the actual application or protocol in use. In this case, it is possible either to invoke cryptographic techniques to enhance HTTP (called S-HTTP, that is the alternative to HTTPS), or to invoke an authentication and key distribution system such as Kerberos to actually archieve the same level of security.

Beyond these protocols, there is also the possibility to protect web transactions in a way that is independent from the transmission techniques. For this purpose we can have a layer above the application layer, for example to provide secure messaging like *Pretty Good Privacy* (PGP), OpenPGP or *Secure MIME* (S/MIME) [26].

The placement of the security protocols in the TCP/IP stack have different reasons and consequences, and all the possibilities have advantages and disadvantages. Providing security at a low layer has the advantage that higher layers (for example applications) become immune and need not to be modified or to care about security in web transactions. Providing security at a high layer has the advantage that it has no impact on the networking infrastructure, and hence the infrastructure need not to be modified.

For this approach, there is a famous principle that strongly recommends to provide security services at a high layer. Accordingly to the *end-to-end principle* in system designs [27], whenever possible, communications protocol operations should be defined to occur at the end-points of a communications system, or as close as possible to the resource being controlled. As an exception, the principle also states that protocol features are only justified at lower layers of the system just if there is a performance optimization.

Following the end-to-end principle, the *Internet Engineering Task Force* (IETF) chartered a *Web Transaction Security* (WTS) Working Group (WG) in the early 1990s [2]. The goal of the Web Transaction Security Working Group was to develop requirements and a specification for the provision of security services to Web transaction, e.g., transactions using HTTP. In late 1993 Allan Schiffman and Eric Rescorla, then with Enterprise Integration Technologies (EIT), came up with a proposal to enhance the HTTP protocol

[2] http://www.ietf.org/html.charters/OLD/wts-charter.html

with the possibility to encrypt data over the communication. This proposal was named *Secure HyperText Transfer Protocol* (S-HTTP or SHTTP), and was a message-oriented application-layer protocol. The IETF Web Transaction Security (WTS) working group was chartered in 1995 to consider this protocol and it concluded and published S-HTTP as an experimental RFC 2660 [28] authored by Rescorla and Schiffman (then with Terisa Systems) in 1999.

However, things evolved differently and independently from the end-to-end principle and the S-HTTP protocol proposal, and the developers at Netscape Communications (commonly known as Netscape) sustained the claim that security at the Transport Layer provides interesting security between the low and high layers. They wanted to create a protocol that could establish secure connections to applications as simply as possible. To achieve this goal, they decided to create an intermediate layer between the Application Layer and the Transport Layer, which was called *Secure Sockets Layer* (SSL). Its job was to provide to application developers an easy way to handle security, meaning that it had to establish secure connections and transmit data over the secure connections. The first SSL version was the SSL 1.0, and later upgraded to SSL 2.0 and SSL 3.0. Later, the *Transport Layer Security* (TLS) was defined as an upgrade to the SSL 3.0. The TLS 1.0 (also called as SSL 3.1) was the first version, and then upgraded to TLS 1.1 (also called as SSL 3.2). Nowadays, the TLS protocol is already in the version TLS 1.2 (also called as SSL 3.3), and it provides security with more cryptographic algorithms and extensions.

As explained in Chapter 1, this thesis focus in a TLS protocol implementation. Hence, the next sections focus on the essential contents to better understand the SSL/TLS protocol in the section 4.6.

4.3 Security Attacks

As explained before, communications can be easily intercepted by skilled programmers or hackers, that gain access to a network to intercept the normal flow of communications. The easiest way to protect a network from an outside attack is to close it off completely from the outside world. A closed network provides connectivity only to trusted known

parties and sites. A closed network does not allow a connection to public networks. However, if we are talking about a private building, or campus, not everyone is trustable, or a attacker can somehow gain access to the private network. This section is based on the chapter 1 of *Network Security Essentials: Applications and Standards*, by William Stallings, 2010.

In general, there is a flow of information between two peers. The first peer A can be considered the information source that sends information to a destination peer B. Different threats are depicted in Figure 4.2. The first figure 4.2(a) shows the normal flow in a communication, and the following four are the general categories of threats:

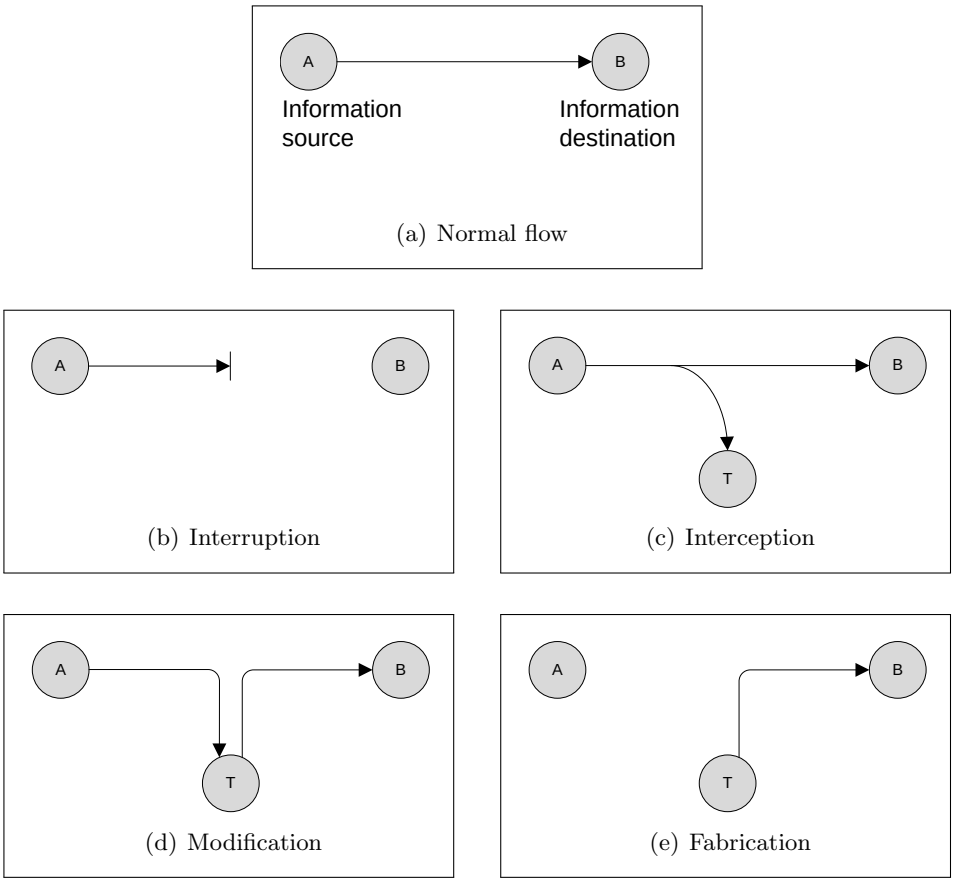


Figure 4.2: Security Threats[3]

- **Interruption:** Part of the communication system between two entities is destroyed or becomes unavailable or unusable. This can be considered an attack of **availability**. Examples include destruction of pieces of hardware such as hard disk, cutting

[3]Adapted from Stallings [29]

off the communication line, disabling management system, etc.

- **Interception:** An unauthorized Peer T gains access to the network and intercepts the information that is going on the communication line. This is an attack of **confidentiality**. The unauthorized peer can be a person, a computer or a program. Examples include capturing data in a network and unauthorized copy of programs and files.

- **Modification:** An unauthorized Peer T not only gains access to the network to intercept the information, but tampers it to send to the destination B. This is an attack of **integrity**. The unauthorized peer can be a person, a computer or a program. Examples include modifying the content of messages transmitted in a network, changing values in a data file or altering a program to perform differently.

- **Fabrication:** An unauthorized Peer T inserts data into the communication system. This is an attack of **authenticity**. Examples include the insertion of messages in a network or the addition of records to a file.

The following figure 4.3 represents these attacks in terms of passive and active attacks.

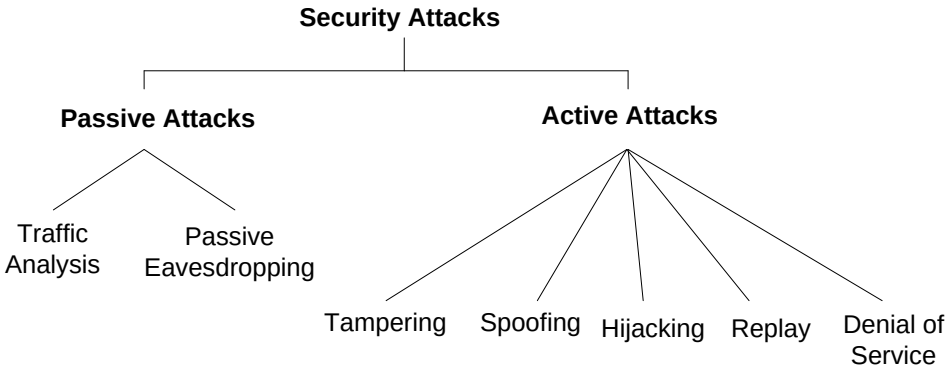


Figure 4.3: Passive and Active Security attacks

4.3.1 Passive Attacks

A passive attack is an attack where an unauthorized attacker monitors or listens the communication between two parties. Passive attacks are in the nature of eavesdropping on, or monitoring of, transmissions, and the goal is to obtain information being transmitted.

There are two types of passive attacks:

- **Traffic Analysis:** Usually, before mounting an active attack, attackers have to collect sufficient information about the network or the user. The operation of traffic analysis gives to the attacker a basic information going on the network, such as, network information, activity on the network, protocols being used, etc.
- **Passive Eavesdropping:** This attack is very similar to the traffic analysis attack, because the attacker collects information from the network, but at the same time also accesses and reads the sensitive contents of the message. If the message is encrypted it requires the attacker to break the encryption to read the message.

Passive attacks are very difficult to detect because they do not involve modification of the data. However, dealing with passive attacks is on prevention rather than detection. The common technique for masking messages is encryption.

4.3.2 Active Attacks

An active attack is an attack where an unauthorized attacker modifies a data stream or creates a false stream. The attacker has the ability to transmit data to one or both of the parties, or block the data stream in one or both directions. There are 5 types of active attacks:

- **Tampering:** Tampering or modification of messages, is when an attacker monitors the network traffic and maliciously captures the message and modifies the content. The messages can be simply altered, delayed or reordered, to produce an unauthorized effect.
- **Spoofing:** Spoofing or masquerading, is when an attacker forges network data and successfully masquerades as another by falsifying data and thereby gaining an illegitimate advantage. This sort of attack can be used to thwart systems that authenticate based on host information. Man-in-the-middle attack is a common attack where the attacker tricks both communicating parties into communicating with him. Both parties think they are talking to each other when in fact the entire conversation is

controlled by the attacker. Other examples of known attacks are IP spoofing, URL spoofing and phishing, referrer spoofing, poisoning of file-sharing networks, caller ID spoofing and e-mail address spoofing.

- **Hijacking:** Hijacking is the next step after the authentication of a legitimate user. A spoofing attack can be used to "hijack" the connection. Session hijacking is a common attack that involves taking control of the session. The attacker takes control of the session and the victim thinks that the session is no longer in operation whatever the cause. Usually this attack can make use of the session cookies to gain unauthorized access to information or services in a computer system. A common method to hijack a web session is using the technique Session Sidejacking. Another common attack is the technique SSL Hijacking that is used to hijack SSL connections.

- **Replay:** Replay attack involves the passive capture of data in a network, and its subsequent retransmission to produce unauthorized effect. An attacker can record and replay network transactions. For example in web commerce, if the protocol is not properly designed and secured, an attacker can record a single transaction and then replay it later when the stock price has dropped, doing it repeatedly until all the stock is gone.

- **Denial of Service:** *Denial of service* (DoS) attack is often used and famously known to bring down systems. The main aim is to bring down systems to prevent them from responding to the users requests. Usually this is done by sending huge amount of traffic to the system, in order to prevent a computer or service from functioning efficiently or at all, temporarily or indefinitely. Other level of a DoS attack, a Distributed Denial of Service (DDoS), is the version of a DoS in a distributed way, in which an attacker makes use of multiple computers previously compromised by taking advantage of security vulnerabilities or weaknesses, to launch the DoS attack.

Active attacks have the opposite characteristics of passive attacks. As explained before in the section 4.3.2, passive attacks are very difficult to detect but solutions to prevent the success of these attacks are available. In the case of active attacks, they are very difficult to prevent absolutely. For this reason, the goal should be to detect the security

attacks and somehow recover from them. However, past detections also contribute to future preventions, so it has a direct effect in prevention as well.

4.4 Security Services

As explained before, the primary goal in network security is to prevent attacks and to secure important data that passes through the network. There are many different ways to secure data, for example with cryptography, that can provide security services to applications. There is no universal agreement about security services. However, accordingly to Stallings [29], one useful classification of security services agrees with both ITU-T recommendation X.800 [30] and RFC 2828 [31], and is the following:

- **Authentication:** The authentication service is used to prove the identity of an entity, assuring that a communication is authentic. In single messages or connections, authentication is usually used to assure to the recipients that the message is from the source that it claims to be. In bidirectional communications it can be required for both parties of the communication to perform authentication. In both situations the service must ensure that the connection is not interfered by a third party that can masquerade as one of the two legitimate parties, to transmit and receive messages.

- **Data Confidentiality:** The idea of confidentiality is to keep secret information or data from others without proper credentials. Confidentiality can be seen as the protection of transmitted data from passive attacks. In practice, in data communications between two entities, the goal is to prevent attackers to read the data. Nowadays, cryptography is the key to provide data confidentiality. In addition, a brief clarification of cryptography will be given in the next sections.

- **Data Integrity:** As with confidentiality, data integrity service is used to keep secret messages, not just from being read, but avoiding modifications in the original messages. Integrity service relates to active attacks. The goal is to ensure that messages are received as sent, with no duplication, insertion, modification, reordering or replays. The service must cover the destruction of the data as well. Plenty of

well-known checksums exist and are discussed in the next sections.

- **Non-repudiation:** Non-repudiation service is used to prevent either sender and receiver from denying having transmitted messages. Thus, with cryptography (Digital Signatures) it is possible to enable the receiver of a message to prove that the message he received was sent by the alleged sender. Similarly the sender is enabled to prove that the message was in fact sent to the alleged receiver.

- **Access Control:** The Access Control service is used to limit and control the access to host systems and applications. To make such control and deny entities to gain unauthorized access, each entity should be first identified or authenticated, and then access rights can be tailored to the individual.

- **Availability Service:** Loss or reduction of availability can be caused by a variety of attacks. The goal is that a system should be able to prevent or recover from loss of availability of elements of a distributed system.

4.5 Cryptography

As explained in the previous section, there are many different ways to secure data, one of them being cryptography. Cryptography is the science that studies ways to keep secrecy of information. People might want to send secure information over insecure channels. A computer network or telephone line are examples of insecure channels. The common example assumes a sender *Alice* that wants to send a message m to a receiver *Bob*. The message can be intercepted and read by the eavesdropper *Trudy*. Or even worse, the message can be intercepted and modified. For this purpose, the communication might require authentication, confidentiality, integrity and non-repudiation services. Cryptography is then used to provide methods or services to prevent the type of attacks described in section 4.3. In this section we provide a brief introduction to cryptography, giving a basic overview and basic principles of cryptography as far as they are relevant for a proper understanding of the SSL/TLS protocols. Five types of cryptographic algorithms are discussed: symmetric key encryption, public key encryption, cryptographic hash functions, message authentication

codes, and digital signatures.

4.5.1 Symmetric Encryption

Symmetric Encryption, also referred to as conventional encryption, secret-key, or single-key encryption, allows encrypted communication between two endpoints using a single shared key. It is called symmetric encryption because it uses the same key for encryption and decryption. This encryption is actually used to provide Confidentiality.

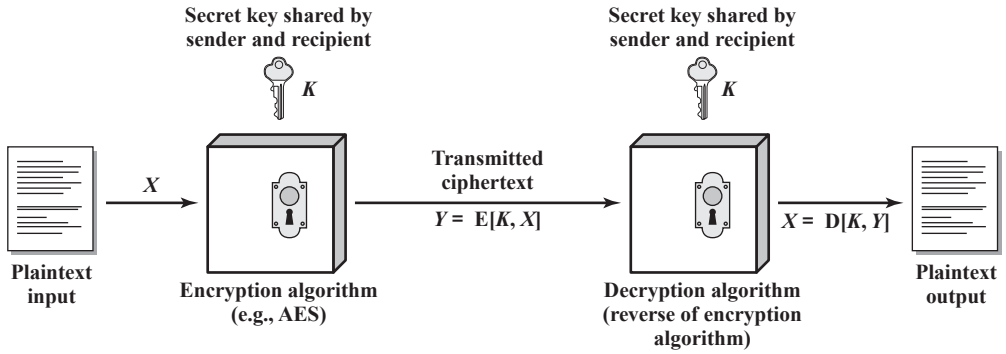


Figure 4.4: Working principle of a symmetric encryption system[4]

The working principle of a symmetric encryption system is illustrated in figure 4.4. On the left side, the sender Alice wants to send a message to the receiver Bob. Alice and Bob share the same key K to encrypt and decrypt messages. Then, Alice encrypts the plaintext message X with the encryption function E (parametrized with the key K). The resulting ciphertext $Y = E[K,X]$ is sent to the recipient Bob over a potentially insecure channel. On the right side, the recipient Bob receives the ciphertext Y and decrypts with the decryption function D (again parametrized with the key K). If the decryption process is successful, the recipient is able to recover the plaintext message $X = D[K,Y]$ sent by Alice.

The primary disadvantage of this approach, is the distribution of the key, that must remain secret. In particular, the difficulty remains in exchanging the secret key, since one has to exchange the keys in the same insecure channel. Sending the key in cleartext leaves open the possibility for attackers to capture the key. The solution for this problem is to use

[4]Reprinted from Stallings [29]

a cryptographic key exchange protocol, also known as public key infrastructure, and is explained in the next section.

Secret key cryptography schemes are generally categorized as being either *stream ciphers* or *block ciphers*:

- *Block cipher:* it operates on a single bit (byte or computer word) at a time and implements some form of feedback mechanism so that the key is constantly changing.
- *Stream cipher:* it operates on individual bits or bytes, and the actual transformation varies during the encryption process.

Block ciphers, are often used in many secure Internet protocols including PGP, SSL/TLS, and IPsec. A *block cipher* can operate for example in the *Electronic Code Book* (ECB), or more preferably in the *CipherBlock Chaining* (CBC) mode. Alternatively, a block cipher can be turned into a stream cipher by operating in the *Cipher FeedBack* (CFB) or in the *Output FeedBack* (OFB) [22]. For more information read the literature [32].

In SSL/TLS protocol, symmetric encryption is commonly used for data bulk encryption, just after the two parties agreed on a shared key. However, encrypting a message does not guarantee that the message is not changed while encrypted. Hence often a *Message Authentication Code* (MAC) is added to a ciphertext to ensure that changes to the ciphertext will be noted by the receiver.

Today, there are a number of popular block cipher and stream cipher algorithms. Thus, we are going to explain the most relevant symmetric encryption algorithms for the SSL/TLS protocol.

Data Encryption Standard (DES)

The DES was designed by IBM in the 1970s and it was adopted as a standard in FIPS PUB 46 [33] by the *National Institute of Standards and Technology* (NIST). DES is a block-cipher that has an effective key length of only 56 bits, and operates on 64-bit blocks. DES is the most common scheme used today. It has a complex set of rules and transformations that were designed specifically to yield fast hardware implementations and slow software

implementations, although this last point is becoming insignificant since the computers are much faster today than twenty years ago. The overall security of DES seems to be good and the encryption algorithm is surprisingly resistant against the most powerful cryptanalytical attacks (differential and linear attacks). The major weakness and vulnerability of DES is the restricted key length of 56 bits. This means that an exhaustive key search (brute force attack) can be done in 2^{56} operations in the worst case, with an average 2^{55} operations. Thus, with this method, specialists have built algorithms to break DES keys very easily and it became perfectly feasible today. One feasible attack is the EFF DES cracker [34], and in 1999 the duration of brute-force attack against DES could be less than 24 hours [35].

Because of the short times of the brute force attacks, the use of DES cannot be recommended anymore. In many applications, DES is replaced with a multi-iteration version of it. *Triple DES* (3DES) is one DES variant and consists of three DES keys, employing 168-bit keys (three times 56), making three encryption/decryption passes over the block. 3DES is also described in FIPS PUB 46-3 [33]. The major disadvantage of 3DES is performance, since a 3DES implementation is roughly three times slower than a normal DES implementation [22].

Advanced Encryption Standard (AES)

In 1997, the U.S. NIST initiated a public competition to develop a new cryptosystem to be the successor of DES. The result of this competition was the *Advanced Encryption Standard* (AES), that became the official successor of DES, and was published in FIPS PUB 197 [36]. AES uses secret key cryptography scheme called *Rijndael*, a block cipher designed by Belgian cryptographers Joan Daemen and Vincent Rijmen. The algorithm uses a fixed block size of 128 bits, and allows any combination of keys lengths of 128, 192 or 256 bits. The three official versions of AES are shown on table 4.1.

Unlike DES, the AES has a clean mathematical structure. Until today, there is no register of techniques to break the AES more efficiently than a brute force attack. A related-key attack can break 256-bit AES with a complexity of $2^{99.5}$, which is faster than brute force

	Block size	Key length	Number of rounds
AES-128	128	128	10
AES-192	128	192	12
AES-256	128	256	14

Table 4.1: The three official versions of AES.

but is still infeasible. 192-bit AES can also be defeated in a similar manner, but at a complexity of 2^{176} which is also infeasible. 128-bit AES is not affected by this attack [22].

RC4

Designed by Ron Rivest in 1987, *Rivest Cipher 4* (RC4) stands as a trademark, so it is often referred to as ARCFOUR or ARC4. RC4 is a stream cipher that uses variable-sized keys, meaning that it generates a stream of pseudo-random bits (a keystream). This ability to handle variable-length keys is one of the advantages of RC4. RC4 is widely used in commercial applications including Oracle SQL, Microsoft Windows and the SSL.

In spite of the fact that RC4 is more than 20 years old, no serious vulnerability had been found so far. However, while remarkable for its simplicity and speed in software, weaknesses have been found in RC4 that argue against its use in new systems. The keystream generated is biased in varying degrees towards certain bit sequences. This weakness has been exploited and can lead to very insecure cryptosystems in *Wired Equivalent Privacy* (WEP) encryption used in *Wireless Local Area Networks* (WLANs). In 2001 Scott Fluhrer, Itsik Mantin, and Adi Shamir published an analysis of the RC4 stream cipher [37]. Some time later, it was shown that this attack could be applied to WEP and the secret key could be recovered from about 4,000,000 to 6,000,000 captured data packets. In 2004 a hacker named KoReK improved the attack and the complexity of recovering a 104 bit secret key was reduced to 500,000 to 2,000,000 captured packets. In 2005, Andreas Klein presented another analysis of the RC4 stream cipher [38]. Klein showed that there are more correlations between the RC4 keystream and the key than the ones found by Fluhrer, Mantin, and Shamir which can additionally be used to break WEP in WEP like usage modes. Finally in 2007, Erik Tews, Ralf-Philipp Weinmann and Andrei Pyshkin extended

Klein's attack and optimized it for usage against WEP, breaking a 104 bit WEP key in 40,000 captured packets with 50% success probability. For 60,000 available data packets, the success probability is about 80% and for 85,000 data packets about 95% [39].

RC2

Also designed by Ron Rivest in 1987, *Rivest Cipher 2* (RC2) is a 64-bit block cipher that uses variable-sized keys between 8 and 128 bits, to replace DES. The algorithm operates in 18 rounds. In 1997 John Kelsey, Bruce Schneier and David Wagner published an analysis to various block ciphers, including RC2 that was cryptanalysed using 2^{34} chosen plaintexts [40]. Consequently, nowadays RC2 is considered to be weak, and its usage should be avoided (at least for any security-critical application).

International Data Encryption Algorithm (IDEA)

The *International Data Encryption Algorithm* (IDEA) is a block cipher that was originally designed by James Massey and Xuejia Lai, intended to substitute DES and was first described in 1991. IDEA also stands as a trademark, and it is best known for its usage in former versions of the *Pretty Good Privacy* (PGP). IDEA is a 64-bit block cipher and uses 128 bits of key length. The algorithm operates in 8.5 rounds. IDEA was designed to be resistant against differential cryptanalysis and related attacks. Consequently it has been very successful, as no successful linear or algebraic weaknesses have been reported so far [22]. In 2007 Eli Biham, Orr Dunkelman and Nathan Keller published the best attack, a high-order differential-linear attack requiring 2^{64}-2^{52} chosen plaintexts, reducing its operation to 6 rounds (instead of the original 8) with a complexity of $2^{126.8}$ encryptions [41].

Skipjack

The Skipjack is a block cipher that was developed by the U.S. *National Security Agency* (NSA) and it was intended for use in the controversial Clipper chip. The Clipper chip was not successful, so they decided to implement also in FORTEZZA cards, in addition

to a digital signature system, SHA-1 and a key exchange algorithm (KEA) (known as FORTEZZA KEA). Skipjack is a 64-bit block cipher and uses 80 bits of key length. The algorithm operates in 32 rounds [42]. In terms of weaknesses, there are no registers of a full version attack to Skipjack. In 1999 Eli Biham, Alex Biryukov and Adi Shamir published one article explaining an attack against 31 of the 32 rounds of Skipjack [43], but they were not been able to find attacks against the full version. Up to today, this is the best cryptanalysis of Skipjack known to the public.

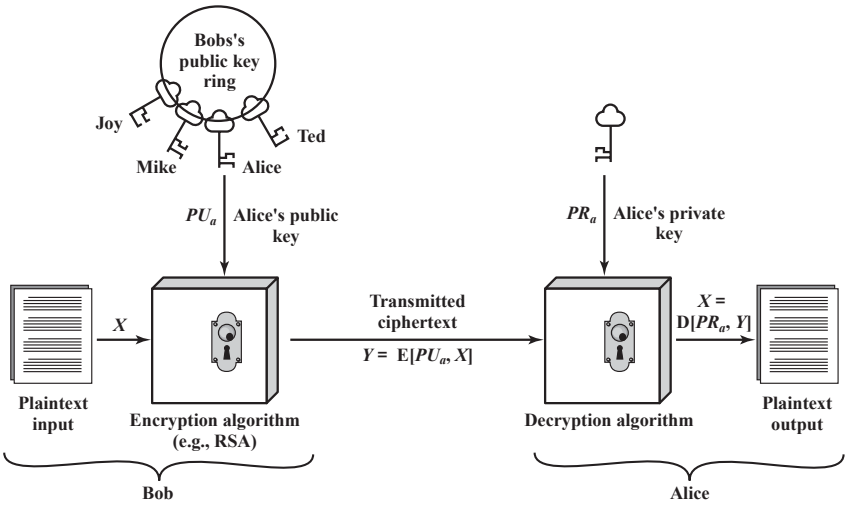
Camelia

Camelia is a block cipher that was developed by Mitsubishi and *Nippon Telegraph and Telephone* (NTT) in 2000. It is a 128-bit block cipher and similarly to AES can use 128, 192 or 256 bits of key length. The number of rounds is 18 (for 128-bit keys) or 24 (for 192-bot or 256-bit keys) [44][45]. Also Camelia was designed to be suitable for both software and hardware implementations and to be resistant against known block cipher attacks. Due to the fact that its definition is similar to AES, theoretically, in the future, it might be possible to break Camelia (and AES) using an algebraic attack. With today's technology, such an attack would take years to compute, and thus it is not realistic.

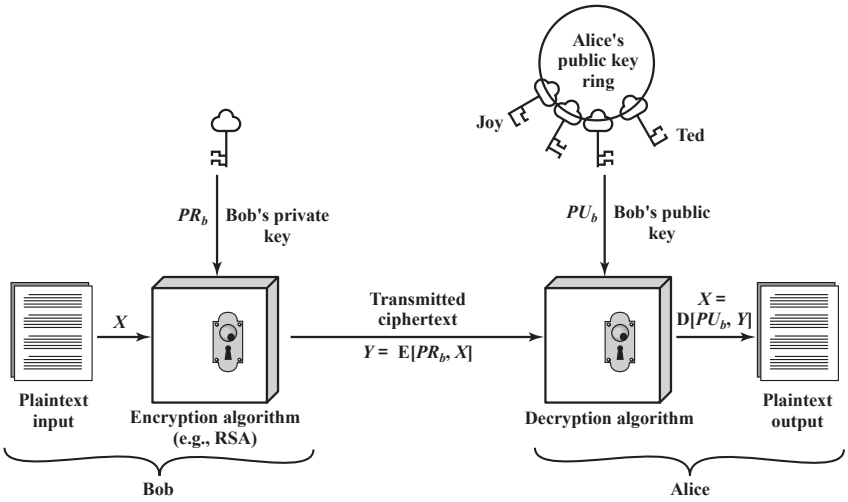
4.5.2 Public Key Encryption

Asymmetric Encryption, also commonly referred to as Public Key Encryption, allows encrypted communication between two parties without the need for prior negotiation of secret keys. It is called asymmetric encryption because it uses different keys for encryption and decryption. In the most popular form of public key cryptography, each party has two keys, the *private key* that must remain secret, and the *public key* that can be freely distributed. The two keys have a special mathematical relationship. A necessary (but usually not sufficient) condition for a public key cryptosystem to be secure is that it is computationally infeasible to compute the private key from the public key.

The working principle of a asymmetric encryption system is illustrated in figure 4.5(a). On the left side, the sender Bob wants to send a message to the receiver Alice. Alice had



(a) Encryption with public key. Working principle of an asymmetric encryption system.



(b) Encryption with private key. Working principle of a digital signature system.

Figure 4.5: Public Key Cryptography[5]

previously generated a pair (public key PU_A, private key PR_A). Bob knows Alice public key, and encrypts the plaintext message X with the encryption function E (parametrized with Alice's public key PU_A). The resulting ciphertext $Y = E[PU_A, X]$ is sent to the recipient Alice over a potentially insecure channel. On the right side, the recipient Alice receives the ciphertext Y and decrypts it with the decryption function D (parametrized with her private key PR_A). If the decryption process is successful, the recipient Alice is able to recover the plaintext message $X = D[PR_A, Y]$ sent by Bob, and Bob is assured

[5]Reprinted from Stallings [29]

that the message can just be read by Alice.

The primary disadvantage of this approach, is that a public key cryptosystem is computationally slower and less efficient than secret key cryptography, thus it is not efficient for data bulk transfer. Thus public key cryptosystems are mainly used for *authentication* and *key management* and symmetric encryption for data bulk transfers. The result of combining secret and public key cryptosystems are often called hybrid cryptosystems and are frequently used in practice, including for example the SSL/TLS protocols.

As explained above, public key systems are commonly characterized by the use of a cryptographic type of algorithm with two keys, one private and one public. Depending on the application, the sender uses either the sender's private key, the receiver's public key, or both to perform some type of cryptographic function. In broad terms, we can classify the use of public-key cryptosystems into three categories [29]:

- **Encryption/Decryption:** The sender encrypts the message with the recipient's public key.
- **Digital Signature:** The sender signs the message with its private key. Signing is achieved by a cryptographic algorithm applied to the message or to a small block of data that is a function of the message.
- **Key Exchange:** The two sides cooperate to exchange a session key. Several different approaches are possible, involving the private key(s) of one or both parties.

Algorithm	Encryption/Decryption	Digital Signature	Key Exchange
RSA	Yes	Yes	Yes
Diffie-Helman	No	No	Yes
DSS	No	Yes	No
Elliptic Curve	Yes	Yes	Yes

Table 4.2: Applications for Public-Key Cryptosystems.

Some algorithms are suitable for all three applications, whereas others can be used only for one or two of these applications. Table 4.2 indicates the applications supported by four main algorithms briefly described below: RSA, Diffie Hellman, Digital Signature Standard (DSS) and elliptic-curve cryptography [29].

RSA

The RSA public key cryptosystem was originally designed by Ronal Rivest, Adi Shamir and Lenoard Adleman in 1978 [46], and the letters RSA are the initials of their surnames, listed in the same order as on the paper. RSA is used in hundreds of software products and can be used for key exchange, digital signatures or encryption of small bulks of data. RSA uses a variable size encryption block and variable size key. The key-pair is derived from a very large number, n, that is the product of two prime numbers chosen according to special rules. These primes may be 100 or more digits in length each, yielding an n with roughly twice as many digits as the prime factors. The public key information includes n and a derivative of one of the factors of n. An attacker cannot determine the prime factors of n (and, therefore, the private key) from this information alone. This is what makes the RSA algorithm so secure [47]. Just if someone finds an efficient integer factorization algorithm, then the RSA public key cryptosystem would be broken. A more worrisome picture would be the possibility to break RSA without having to factorize integers. However, it has not been shown so far [22].

Diffie-Hellman Key Exchange

After publishing the article [48], Whitfield Diffie and Martin Hellman came up their own implementation for key exchange. Diffie-Hellman (D-H) is used for secret-key exchange only, and not for authentication or digital signatures. The purpose of the algorithm is to enable two users to exchange a secret key securely that then can be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of the keys.

Like any other protocol that employs public key cryptography, the Diffie-Hellman key exchange protocol is vulnerable to the *man-in-the-middle attack*. A person in the middle may establish two distinct Diffie-Hellman key exchanges, one with Alice and the other with Bob, effectively masquerading as Alice to Bob, and vice versa, allowing the attacker to decrypt (and read or store) then re-encrypt the messages passed between them. A method to authenticate the communicating parties to each other is generally needed to prevent this type of attack. So, in practice, D-H protocol is usually combined with mutual

authentication protocols to come up with an authenticated key exchange protocol. To accomplish that, usually D-H is authenticated using RSA signatures. Consequently, digital signatures and PKIs must be used to securely deploy authenticated key exchange protocols. More information about Diffie-Helman can be found in the book [47]).

Digital Signature Standard (DSS)

The algorithm *Digital Signature Standard* (DSS) was published in FIPS PUB 186 by the National Institute of Standards and Technology (NIST) and provides digital signature capability for the authentication of messages. DSS makes use of SHA-1 and presents a new digital signature technique, the Digital Signature Algorithm (DSA). The DSS was originally proposed in 1991 and revised in 1993 in response to public feedback concerning the security of the scheme. There was a further minor revision in 1996. The last release is the FIPS PUB 186-3. The DSS uses an algorithm that is designed to provide only the digital signature function. Unlike RSA, it cannot be used for encryption or key exchange [47].

Elliptic Curve Cryptography (ECC)

The vast majority of algorithms and standards on Internet use public key cryptograpghy for encryption and RSA for digital signature to be secure. RSA has been increasing the bit length of its algorithm, giving higher levels of security but increasing the processing time making a heavier processing load to the application. A heavier processing has performance consequences in systems such as electronic commerce that support large number of transactions, or light equipment such as Personal Digital Assistant (PDAs) that have limited compute power and/or memory.

Elliptic Curve Cryptography (ECC) came up to compete with RSA. It is a public key algorithm based on elliptic curves that appears to offer equal security than RSA for a far smaller bit size, thereby reducing processing overhead. Its theory has been around for some time but just recently a few implementations have begun to appear, as well as the interest in probing for weaknesses. Thus, the confidence level in ECC is not yet as high

as that in RSA [29]. More information can be found in the RFC 6090 [49].

Other Public-Key Cryptography Algorithms

One famous algorithm is **FORTEZZA KEA**, already mentioned above. It was designed by NSA in 1994, and it basically refers to a modified Diffie-Hellman key exchange protocol. In short, a long-term certificate-based Diffie-Hellman key exchange is combined with a ephemeral Diffie-Hellman key exchange. Furthermore it used the block cipher Skipjack, that is used to reduce the key to 80 bits long. The FORTEZZA KEA algorithm is not going to be explained, since the full mathematical description is beyond the scope of this thesis.

Another often used algorithm is the ***Secure Remote Password*** **(SRP)** protocol, and it is a password-authenticated key agreement protocol. SRP allows a user to authenticate himself to a server, it is resistant to dictionary attacks mounted by an eavesdropper, and it does not require a trusted third party. SRP creates a large private key shared between the two parties in a manner similar to Diffie-Hellman, then verifies to both parties that the two keys are identical and that both sides have the user's password. SRP is an agorithm often used in SSL/TLS implementations. More information can be found in the RFC 2945 [50] and RFC 5054 [51].

Finally, another often used algorithm is the *Pre-Shared Key* (PSK), that is more often called ***Transport Security Layer Pre-Shared Key*** **(TLS-PSK)** in SSL/TLS implemenations. TLS-PSK is a set of cryptographic protocols that provide secure communications using shared keys (symmetric keys). The main goal is to combine PSK with other algorithms and provide a secured way to create secure communications. Nowadays there are several ciphersuites: the first set of ciphersuites uses only symmetric key operations for authentication; the second set uses a Diffie-Hellman key exchange authenticated with a pre-shared key; the third set combines public key authentication of the server with pre-shared key authentication of the client. More information can be found in the RFC 4279 [52] and RFC 4785 [53].

4.5.3 Hash Functions

Hash functions, also called message digests or one-way encryption, are algorithms that, in some sense, use no key. Instead, a fixed-length hash value is computed based upon the plaintext that makes it impossible for either the contents or length of the plaintext to be recovered. Hash algorithms are typically used to provide a digital fingerprint of a file's contents, often used to ensure that the file has not been altered by an intruder or virus. Hash functions are also commonly employed by many operating systems to encrypt passwords. Hash functions, then, provide a measure of the integrity of a file [47].

Examples of cryptographic hash functions are MD5 and SHA-1.

MD5

Message-Digest algorithm 5 (MD5) was originally designed by Ron Rivest and it generates hash values of 128 bits (independent from the input message length). In terms of security, many cryptographers have been successfully exploiting the algorithm, and have been able to generate rogue CA certificates [22].

SHA

Secure Hash Algorithm (SHA), an algorithm for NIST's *Secure Hash Standard* (SHS) was designed by Ron Rivest as well. Its algorithm is conceptually similar to MD5, but a little stronger and slower. SHA-1 produces a 160-bit hash value and was originally published as FIPS 180-1 and RFC 3174. FIPS 180-2 (aka SHA-2) describes five algorithms in the SHS: SHA-1 plus SHA-224, SHA-256, SHA-384, and SHA-512 which can produce hash values that are 224, 256, 384, or 512 bits in length, respectively. SHA-224, -256, -384, and -512 are also described in RFC 4634 [47].

4.5.4 Digital Signatures

As illustrated in figure 4.5(b), public key encryption can be used in other ways. Instead of encrypting with the public key (ensuring secrecy), a sender Bob wants to encrypt the

message with his private key (ciphertext) and then send it to the receiver Alice, ensuring that the message is indeed from him. Alice receives the message and can decrypt it using Bob's public key, thus proving that the message must have been encrypted by Bob. No one else knows Bob private key, so he is the only one that can create such ciphertext to be decrypted with his public key. A secure hash code such as SHA-1 can be used to encrypt the message with the sender's private key. Therefore the message cannot be altered. Thus, if Bob sends the ciphertext and the message in cleartext to the receiver Alice, she can decrypt and compare the messages, checking its authenticity. Thus the message is authenticated both in terms of source and in terms of data integrity. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim he did not sign a message, while also claiming their private key remains secret. However, it is good to emphasize that this method does not provide confidentiality. When the message is sent in clear text, it can be safe from alteration but not from eavesdropping.

4.5.5 Message Authentication Code

It is not always necessary to encrypt messages to protect their confidentiality. Sometimes it can be sufficient to provide message authentication and integrity (meaning that a recipient can receive a message and verify its authenticity and integrity). For this purpose, *Message Authentication Code* (MAC) are used. A MAC algorithm accepts as input a secret key and an arbitrary length message to be authenticated, and outputs a MAC (known as a tag). The MAC value provides message data integrity and authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

The most widely used MAC is *Hash-based Message Authentication Code* (HMAC). HMAC is a specific construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret key. Any cryptographic hash function, such as MD5 or SHA-1, may be used in the calculation of an HMAC, and the resulting MAC algorithm is termed HMAC-MD5 or HMAC-SHA1 accordingly.

4.5.6 Certificates

In a Public Key Infrastructure, although Bob could have sent a private message to Alice, signed it, and ensured the integrity of the message, he still needs to be sure that he is really communicating with Alice. In this case Alice can be a bank. This means that Bob needs to be sure that the public key he is using corresponds to the bank's private key. Similarly, the bank may also want to verify that the message signature really corresponds to Bob's signature. If each party has a certificate which validates the other's identity, confirms the public key, and is signed by a trusted agency, then they both will be assured that they are communicating with whom they think they are. Such a trusted agency is called a *Certificate Authority* (CA), and *Certificates* are used for authentication and prevent someone from impersonating a party with false key.

Usually SSL/TLS uses digital certificates to authenticate servers. However, it can be used to authenticate clients as well. SSL and TLS uses X.509 certificates to validate entities. X.509 certificates contain information about the entity, and the structure of an X.509 v3 digital certificate is shown in the table 4.3:

Certificate
Version
Serial Number
Signature Algorithm ID
Issuer Name
Validity
• Not Before Date
• Not After Date
Subject Name
Subject Public Key Info
• Public Key Algorithm
• Subject Public Key
Issuer Unique Identifier (optional)
Subject Unique Identifier (optional)
Extensions (optional)
Subject Public Key Info

Table 4.3: Content of an X.509 v3 Certificate.

As explained above, a trusted third party outside the server and client pair is needed to validate the certificate, which is the certificate authority. Reputable certificate authorities, such as VeriSign, are responsible for ensuring the trust of all World Wide Web entities. In some cases it may be necessary to create a chain of certificates, each one certifying the previous one. This hierarchy of trust is vital to the authentication of an entity. This

process is called certificate chaining and is illustrated in figure 4.6.

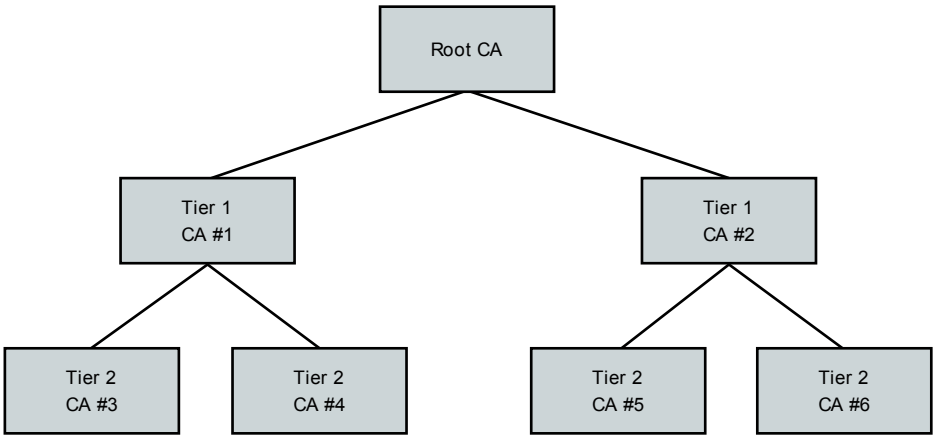


Figure 4.6: Hierarchy of Thrust[6]

4.6 Transport Layer Security (TLS)

After discussing many techniques to provide security, it is time to give one entire solution to provide secure communications, for example in web commerce or bank transactions. *Transport Layer Security* (TLS) protocol is the successor of the *Secure Sockets Layer* (SSL) protocol. In this section we will provide a historical and global overview of the TLS 1.2 protocol, already including all the enhancements from the previous SSL and TLS versions.

4.6.1 Introduction

Originally designed by Netscape, SSL 1.0 was the first version of SSL. The protocol was updated to SSL 2.0 and SSL 3.0, which introduced authentication methods, digital signatures and more security algorithms in order to provide more security. Afterwards, TLS 1.0 was first defined in RFC 2246 [55] in 1999 as an upgrade to SSL Version 3.0. As stated in the RFC, "the differences between this protocol and SSL 3.0 are not dramatic, but they are significant enough that TLS 1.0 and SSL 3.0 do not interoperate."However, TLS 1.0 does include a means by which a TLS implementation can downgrade the connection to

[6]Reprinted from Cisco [54]

SSL 3.0. Therefore TLS 1.0 can be viewed as essentially SSL 3.1. The same happens with TLS 1.1 (SSL 3.2) defined in the RFC 4346 [56] and TLS 1.2 (SSL 3.3) defined in the RFC 5246 [4]. The main upgrades between the new versions of the protocol were basically the addition of protection against CBC attacks, authentication and hash algorithms, and finally TLS extensions [22]. In the next sections we will provide a global overview of the TLS 1.2 protocol, already including all the enhancements and previous protocols.

4.6.2 SSL/TLS Architecture

In the previous section we explained how the SSL protocol evolved in three versions (SSL 1.0, SSL 2.0 and SSL 3.0) to finally become the protocol we know today, TLS. The SSL/TLS protocol is a client/server protocol that provides the following basic security services to the communication peers:

- Authentication services (peer entity and data origin authentication);
- Connection confidentiality services;
- Connection integrity services (without recovery);

In spite of the fact that the SSL/TLS protocol uses public key cryptography, it does not provide non-repudiation services [22]. In terms of cryptography, all the algorithms explained in section 4.5 can be used in SSL and TLS. As explained before, a public key cryptosystem is computationally slower and less efficient than secret key cryptography, thus is not efficient for data bulk transfer. For this reason, the common scheme is the usage of *Public Key Infrastructure* (PKI) to exchange a secret key between the two peers. Then the data bulk can be done using symmetric algorithms.

In terms of architecture, Netscape developers designed the SSL protocol as a separate layer just for security. Looking at the TCP/IP stack, an application layered protocol sends messages to the below transport layer protocol, that sends the message to the next network layer below. SSL was created right between the application layer and the transport layer, as shown in figure 4.7.

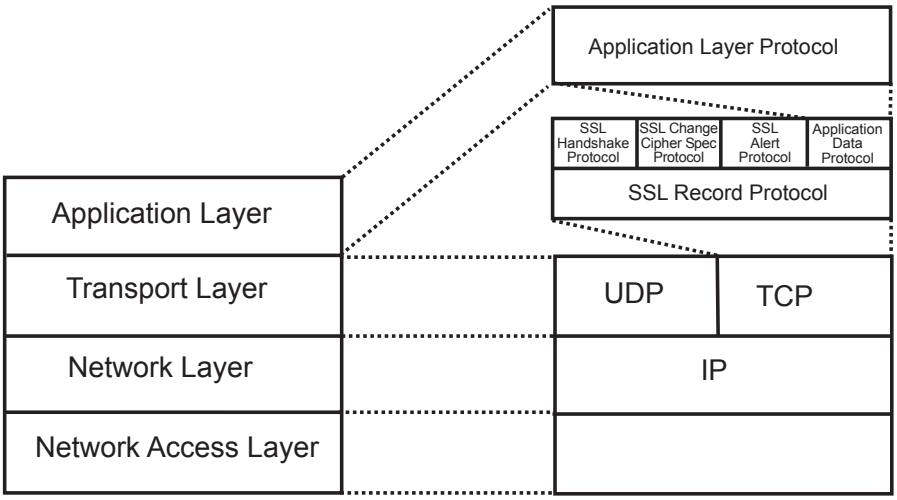


Figure 4.7: The SSL with its sub-layers and sub-protocols[7]

SSL was initially designed to make use of TCP to provide reliable end-to-end secure service (however today is possible to provide over UDP with DTLS). SSL is not a single protocol, but rather two layers of protocols. TLS inherited the SSL architecture and its sub-layers and sub-protocols are illustrated in figure 4.8.

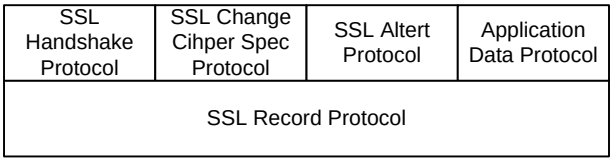


Figure 4.8: SSL Protocol Stack

The SSL/TLS protocol can be divided into two layers. The first layer consists of the application protocol and the three sub-protocols: the Handshake Protocol, the Change Cipher Spec Protocol, and the Alert Protocol. The Application data protocol just refers to the application protocol that needs security (e.g. HTTP).

The second layer is the *Record Protocol* and it is used for encapsulation of higher-layer protocol data. Its purpose is to encrypt, authenticate, and optionally compress packets.

[7]Reprinted from Stallings [29]

SSL/TLS Record Protocol

The Record Protocol is responsible for encapsulating messages for transmission over underlying communication protocols, usually TCP/IP. It takes messages from the upper layers to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients. Figure 4.9 indicates the overall operation of the SSL/TLS Record Protocol [29].

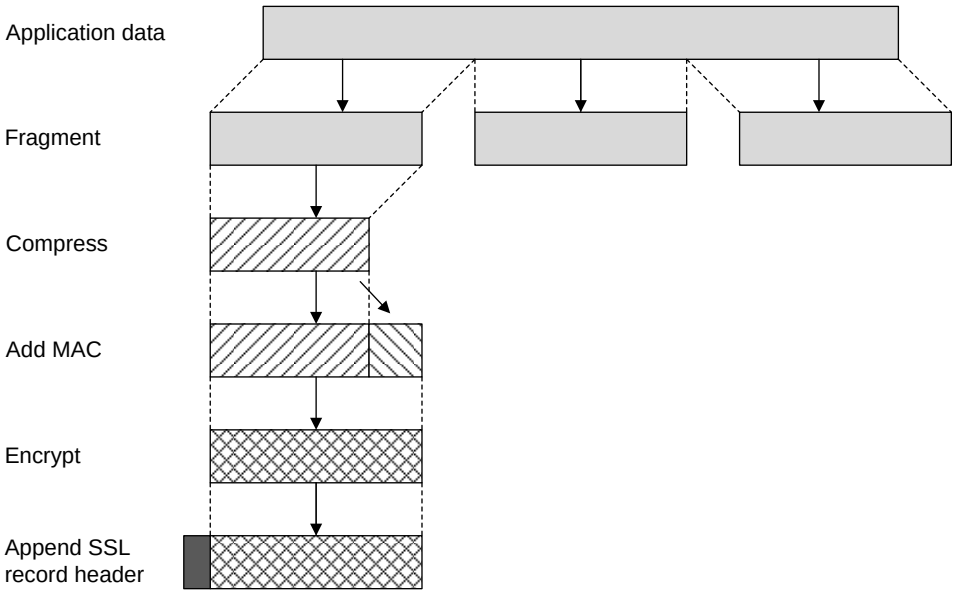


Figure 4.9: SSL/TLS Record Protocol operation[8]

As explained before, just encrypting a message does not guarantee that this message is not changed while encrypted. Hence often in SSL/TLS Record protocol a MAC is added to the ciphertext to ensure that changes to the ciphertext will be noted by the receiver. Message authentication codes can be constructed from symmetric ciphers (e.g. CBC-MAC).

The algorithms used in the record protocol operation depends on the parameters defined in the Handshake protocol. A set of parameters are defined during the Handshake protocol, and the record message is going to be encrypted accordingly to the parameters defined during the handshake (in the cipher suite). In terms of encryption algorithms used to encrypt the SSL/TLS record, there are several TLS cipher suites defined in the RFC 5246

[8] Adapted from Stallings [29]

[4]. For more information read the section 4.6.2 about the Handshake Protocol.

The SSL/TLS Record format is illustrated in figure 4.10. It begins with a header. The header includes the *Content Type* that refers to the higher layer protocol being used (refers to one of the four sub-protocols, e.g. Alert); a protocol *Version* that referes to version of the SSL protocol in use (e.g. SSL 3.3 aka TLS 1.2); and the *Length* in bytes of the plaintext fragment. The body of the SSL/TLS record contains the message plus the MAC in encrypted form.

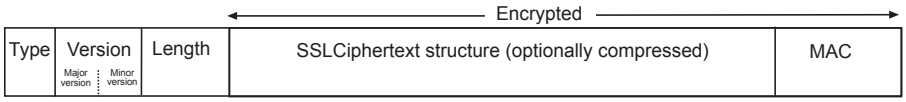


Figure 4.10: SSL/TLS Record format

SSL/TLS Change Cipher Spec Protocol

The *Change Cipher Spec Protocol* is one of the three sub-protocols that use the Record Protocol, and it is the simplest. This protocol consists of a single message, which consists of a single byte with the value 1. It is sent immediately after the handshake concludes and before any application data messages are sent. Its purpose is to cause the pending state to be copied into the current state, which updates the cipher suite to be used on the connection [29].

SSL/TLS Alert Protocol

The *Alert Protocol* is used to send messages to indicate a change of status or an error condition to the peer. A full list of messages can be found in RFC 5246 [4]. Alerts are commonly sent when the connection is closed, an invalid message is received, a message cannot be decrypted, or the user cancels the operation.

SSL/TLS Handshake Protocol

The *Handshake Protocol* is responsible for selecting a cipher suite and generating a master secret, which together comprise the primary cryptographic parameters associated with a

secure session. The handshake protocol can also optionally authenticate parties who have certificates signed by a trusted certificate authority. The session information consists of a session ID, peer certificates, the cipher suites to be used, the compression algorithm to be used, and a shared secret that is used to generate keys. The protocol and its message flows are illustrated in figure 4.11.

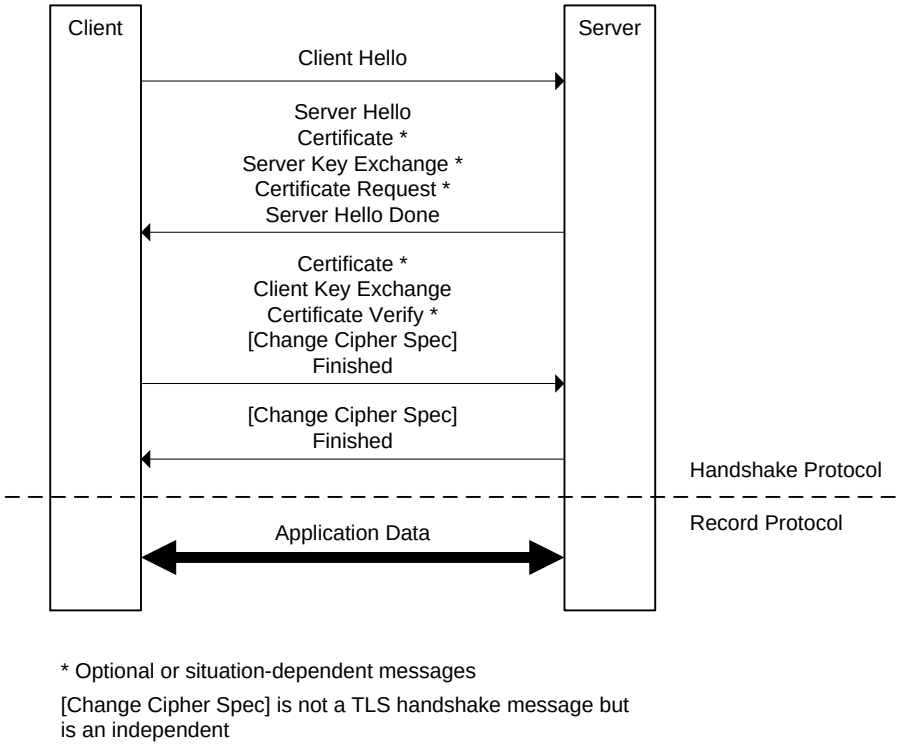


Figure 4.11: SSL/TLS Handshake Protocol

Note that there are mandatory and optional messages, depending on the need in specific situations. Note also that *Change Cipher Spec* is not actually an SSL/TLS Handshake protocol message but represents an SSL/TLS protocol of its own, and hence a content type. Table 4.4 lists the defined messages types in the handshake protocol.

The message flow with optional messages can adapt to different situations. Figure 4.12 illustrates the message flow of a Resuming Session, where the ID of the session will be resumed, and there is no need of renegotiation of ciphers.

Establish Security Capabilities

This phase is used to initiate logical connections and to establish the security capabilities

Message Type	Parameters
Hello Request	Null
Client Hello	Version, random, session id, cipher suite, compression method
Server Hello	Version, random, session id, cipher suite, compression method
Certificate	Chain of X.509v3 certificates
Server Key Exchange	Parameters, signature
Certificate Request	Type, authorities
Server Hello Done	Null
Certificate Verify	Signature
Client Key Exchange	Parameters, signature
Finished	Hash Value

Table 4.4: Handshake Protocol Message Types.

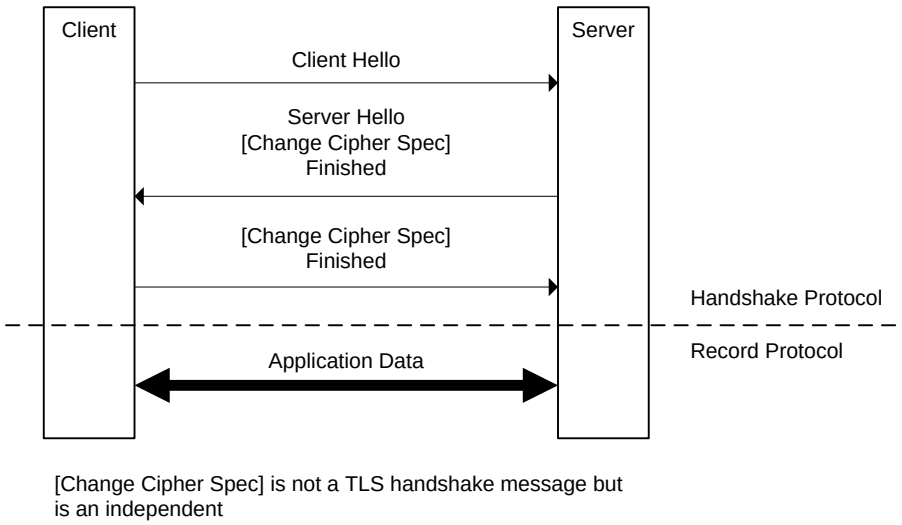


Figure 4.12: Simplied SSL/TLS Handshake Protocol (Resuming Session)

that will be associated with it. The Client sends a *Client Hello* which includes the cipher suite. After sending the Client Hello message, the client waits for the *Server Hello* from the server, which will contain a single cipher suite selected by the server from those proposed by the client.

A *Cipher Suite* is a combination of authentication, encryption, and Message Authentication Code (MAC) algorithms that are used to negotiate the security settings for security protocols. In SSL/TLS it defines a key exchange algorithm and CipherSpec (encryption algorithm for data bulk transfer, a message authentication code (MAC) algorithm and others).

An example of a cipher suite is TLS_RSA_WITH_DES_CBC_SHA, where TLS is the protocol version, RSA is the algorithm that will be used for the key exchange, DES_CBC

is the encryption algorithm (using a 56-bit key in CBC mode), and SHA-1 is the hash function. For detailed information about supported cipher suites in the TLS 1.2 protocol read the Appendix C and Appendix A.5 of the RFC 5246 [4].

The first element of the Cipher Suite is the Key Exchange method. The following key exchange methods are supported:

- **RSA**: Key exchange and server authentication are combined with RSA. The secret key is encrypted with the server's public key previously sent in the server's certificate;
- **Fixed Diffie-Hellman** (abbreviated DH): Key exchange with Diffie Hellman and authentication using certificates certified by a certificate authority (CA). The server's certificate contains fixed Diffie-Hellman public parameters signed by the certificate authority (CA). The client provides its Diffie-Hellman public key parameters either in a certificate, if client authentication is required, or in a key exchange message;
- **Ephemeral Diffie-Hellman** (abbreviated DHE): Key exchange with Diffie Hellman and authentication using DSA or RSA. DHE is used to create ephemeral (temporary, one-time) secret keys. In this case, the Diffie-Hellman public keys are exchanged, signed using the sender's private RSA or DSS key. The receiver can use the corresponding public key to verify the signature. Certificates are used to authenticate the public keys. This would appear to be the most secure of the three Diffie-Hellman options because it results in a temporary, authenticated key;
- **Anonymous Diffie-Hellman** (abbreviated DH_anon): The base Diffie-Hellman algorithm is used, with no authentication. That is, each side sends its public Diffie-Hellman parameters to the other, with no authentication. This approach is vulnerable to man-in-the-middle attacks, in which the attacker conducts anonymous Diffie-Hellman with both parties;
- **Fortezza**: The technique defined for the Fortezza scheme.

Many possible solutions (in TLS 1.2) are also using Diffie-Hellman or RSA key exchange with PSK or SRP authentication, or ECC. For more information read the literature [22]. Following the key exchange algorithm is the CipherSpec, which includes the following most

common fields:

- **Ciphers**: RC2, RC4, DES, 3DES, AES, IDEA, Fortezza (some of them can be combined with CBC);
- **MACs**: SHA and MD5.

4.6.3 Candidate Technologies

There are several implementations of the SSL and TLS protocol, which are free and open-source. Sometimes choosing between the available implementations can be tough. In this section we will provide a limited comparison of several of the most prominent libraries.

Due to the specifications of all the implementations, some of them do not fit in our list of possible choices for different reasons. For example, *Java Secure Socket Extension* (JSSE) implements SSL and TLS in Java environment, and such language was not compatible with the framework. Therefore, our list of choices is limited to three options:

- **GnuTLS**: The GNU Transport Layer Security Library [57];
- **NSS**: Network Security Services [58];
- **OpenSSL**: A Open Source toolkit implementing the SSL and TLS protocols [59].

The following sections are based on the material of *GnuTLS - Comparison of different free TLS implementations*[60] and gives a brief comparison between these three libraries, comparing the features that directly relate to the SSL and TLS protocol.

Protocol Support

One of the initial comparisons is the protocol support. GnuTLS is presented as a technology that implements the most advanced protocol versions, from the SSL 3.0 to the TLS 1.2. SSL 2.0 has been deprecated since 1996 and it has serious security flaws, but NSS and OpenSSL opted to implement it. NSS and OpenSSL only support versions from the SSL 2.0 to the TLS 1.0.

Implementation	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2
GnuTLS	No	Yes	Yes	Yes	Yes
NSS	Yes	Yes	Yes	No	No
OpenSSL	Yes	Yes	Yes	No	No

Table 4.5: Candidates' Protocol Support.

Key Exchange, Encryption and MAC Algorithms

In order to provide different variations of security in the Handshake protocol, the Key Exchange Algorithms implemented by the three options are significantly important. GnuTLS implements all the algorithms except the *Elliptic curve Diffie-Hellman* (ECDHE). NSS only provides RSA, DHE-RSA, DHE-DSS and SRP-DSS. OpenSSL provides the most used algorithms such as RSA, DHE-RSA, DHE-DSS, DH-ANON, PSK and ECDHE.

Library	RSA	DHE-RSA	DHE-DSS	DH-ANON	SRP-DSS	SRP-RSA	SRP	PSK	DHE-PSK	ECDHE
GnuTLS	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
NSS	Yes	Yes (client only)	Yes (client only)	No	Yes	No	No	No	No	No
OpenSSL	Yes	Yes	Yes	Yes	No	No	No	Yes	No	Yes

Table 4.6: Candidates' Key Exchange Algorithms.

The Encryption Algorithms implemented by the three options are significantly important as well. GnuTLS implements all the algorithms showed on table 4.7 except DES-CBC because DES is considered insecure. NSS and OpenSSL implements all of them except the AES-GCM, the *Galois/Counter Mode* (GCM) for the AES algorithm operation.

Library	AES-CBC	AES-GCM	3DES-CBC	DES-CBC	RC4-128	RC4-40	CAMELLIA-CBC
GnuTLS	Yes	Yes	Yes	No	Yes	Yes	Yes
NSS	Yes	No	Yes	Yes	Yes	Yes	Yes
OpenSSL	Yes	No	Yes	Yes	Yes	Yes	Yes

Table 4.7: Candidates' Encryption Algorithms.

In terms of MAC functions, all the technologies presented implement HMAC-MD5, HMAC-SHA-1 and HMAC-SHA-256. In addition, all technologies implement the DEFLATE compression method.

Development Environment

Other important comparison is the development environment. GnuTLS uses autoconf, automake and libtool as building tools, it is very well documented with manuals and *Application Programming Interface* (API) reference (both online and PDF format), and it has limited compatibility with OpenSSL. NSS uses makefile as building tool, it is well documented with a online manual, and it has compatibility with OpenSSL with a separate package. OpenSSL uses makefile as building tool, and it is not so well documented as the others, since the only known manuals are the *man* pages.

Portability Concerns

In terms of portability, GnuTLS requires the support of C89 (sometimes called *ANSI C*), and the libraries libgcrypt and libtasn1. It operates in most of the POSIX platforms or Windows, including GNU/Linux, MacOS X, Solaris and major BSD variants. It is also thread-safe when using *mutex* hooks.

Library	**Platform Requirements**	**Thread Safety**	**Supported Operating Systems**
GnuTLS	C89, libgcrypt, libtasn1	Thread-safe, needs custom mutex hooks if neither POSIX or Windows threads are available.	Generally any POSIX platforms or Windows, commonly tested platforms include GNU/Linux, Win32/64, Mac OS X, Solaris, OpenWRT, FreeBSD, NetBSD, OpenBSD.
NSS	C89, NSPR	Thread-safe	AIX, Android, FreeBSD, NetBSD, OpenBSD, BeOS, HP-UX, IRIX, Linux, Mac OS X, OS/2, Solaris, OpenVMS, Amiga DE, Windows, WinCE, Sony Playstation
OpenSSL	C89	Needs mutex callbacks	Unix, DOS (with djgpp), Windows, OpenVMS, MacOS, NetWare

Table 4.8: Candidates' Portability Concerns.

NSS is thread-safe, requires support for C89 and *Netscape Portable Runtime* (NSPR), and it operates in most of Unix based platforms (including Android), Windows, Sony Playstation and others. Lastly, OpenSSL requires support for C89, needs mutex callbacks for thread safety and it operates in most of Linux platforms, DOS, Windows, MacOS and others. These differences are showed on table 4.8.

Chapter 5

Protocols' Implementation

Currently there is no standardized way in DOORS to implement a communication protocol. Deploying or implementing a protocol in the DOORS framework depends on a lot of factors, and its design depends not just on its specifications but can also derive from an integration of external libraries.

The most typical case, the deployment of simple protocols applies software engineering principles in combination with formal methods towards the design of communication protocols. In this case, developers can use DOORS tools, such as Code Generators, to generate code compatible with DOORS messages and Tasks, as explained in chapter 2. Thus, it is possible to facilitate the deployment or implementation of a protocol using this set of tools. However, a protocol may not have simple specifications and may be advantageous to use external libraries. In this case other approaches may be investigated in order to make possible a perfect integration in the DOORS framework.

The main objective of this chapter is to introduce the design of the protocols, to provide a NAT traversal solution and secure communications in the DOORS framework. This chapter is important to later on present the implementations. Since this dissertation focuses on two different solutions (two protocols), this chapter is divided in two parts. Firstly, as a NAT traversal solution, the design of the STUN protocol is presented in the DOORS framework, that was previously discussed in chapter 3. Secondly, to provide secure communications the choice is the TLS protocol previously discussed in chapter

4. Since the two protocols have different specifications, their design is different in the DOORS framework. In this chapter, a brief overview and discussion of the designs are provided.

5.1 STUN Protocol

As explained before, *Session Traversal Utilities for NAT* (STUN) is a lightweight client-server protocol to be used in the context of NAT traversal solutions. Not a solution itself, STUN is rather a tool to be used with other protocols. In this section, the steps for the design and implementation of this client-server protocol in the DOORS framework are explained.

5.1.1 Considerations and Choices

After considering the STUN protocol's specifications and several design ideas for the DOORS framework architecture, the following design choices were made:

1. Client support for the STUN protocol;
2. STUN protocol operating over UDP;
3. Simple design and implementation of a STUN Task without external libraries;

STUN used in the context of NAT traversal solutions, needs a client and a server to exchange information. As shown in chapter 3, the client party is the one interested to know its public IP and port to somehow contact other parties and give them useful information to be contacted back. For this reason, for the first design choice it was decided to provide client support for the STUN protocol in the DOORS framework, assuming that there is already a public STUN server in the network to provide server responses.

In terms of operation, STUN protocol can transmit messages in different ways. The goal of this implementation is to provide NAT traversal solution for the most popular cases, and gather a better understanding of the DOORS architecture. For this reason, for the second design choice it was decided to simplify the implementation and send STUN messages just

over UDP. It is also important to refer that this implementation must support IPv4 only. The deployment of IPv6 eliminates the problem of NATs and certainly it is reasonable to observe that achieving high address utilization densities is no longer the objective. Probably IPv6 NATs will be implemented and used in the future, however it is out of the scope of this thesis.

Looking at the integration perspective in the DOORS architecture, it is important to understand the framework structure and how things work there. After considering several design ideas in the DOORS architecture, it was decided to design and implement the STUN protocol without help from other external libraries. Since STUN is a simple client-server protocol, it would be easier just to create a STUN task inside DOORS and use a auxiliary UDP task to send and receive messages over UDP. The resultant design and implementation of this application should result in asynchronous and message-based interactions between modules. Therefore, all these considerations are taken into account in designing the STUN module.

5.1.2 Design Prototyping

This section describes the design prototyping for using STUN protocol in the DOORS framework. It starts off with the background on how this design came about, and in the next section it focuses on the details of the design itself.

The presented design was approached from creating a STUN task in the DOORS environment that could give the support for the STUN protocol. The STUN task, called StunTask, is the main creation of this architecture and its architecture is illustrated in figure 5.1.

StunTask is located between a LEMon User task and the UdpTask. It should send and receive messages from other tasks to use its functionalities, not calling functions or pointers, but rather just exchanging DOORS messages. In the DOORS environment, a *Service Data Unit* (SDU) is used to identify exchanged DOORS messages between tasks. The StunTask has a upper port and a lower port. User messages come from the upper port and all the encoded and decoded PDUs should be connected to the lower port and sent to

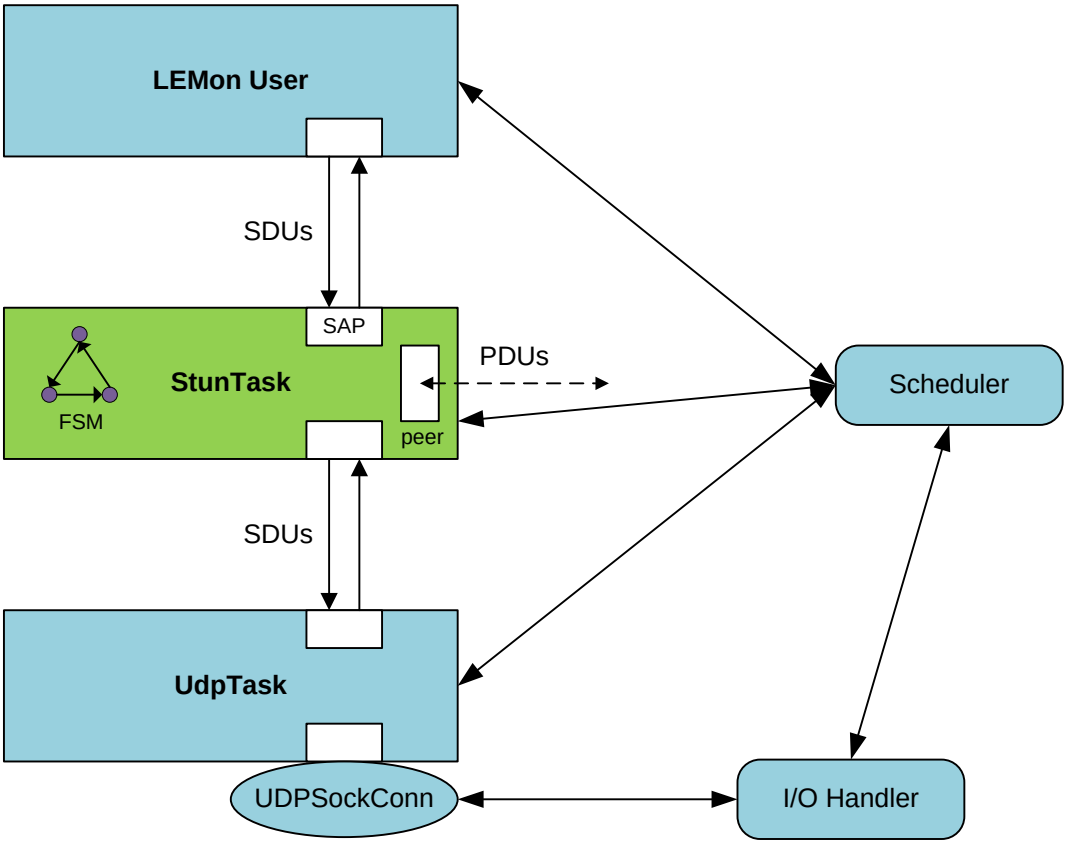


Figure 5.1: Architecture overview of STUN support in DORRS

the lower task. The structural modulation of the StunTask is presented in the sub-section 5.1.4.

The LEMon User Task is the environment responsible to monitor User events. It inherits the EnvTask and gets inputs from the User, and sends user-defined messages to the Service Access Point (SAP) of the StunTask.

The UdpTask is the intermediate between the *Operating System* (OS) and the StunTask, and it is responsible to give UDP support. It is the one responsible to receive messages from the StunTask and use a UDPSockConn device to send them via UDP to the network. Messages coming from the device are sent to the StunTask.

In terms of overall architecture, the tasks are controlled by the Scheduler, and the I/O Handler should be responsible to manage the devices required for the communication.

5.1.3 Design Architecture

The previous section introduced the main blocks of the architecture. This section contains a more detailed description, focusing on the StunTask. The design architecture is illustrated in figure 5.2.

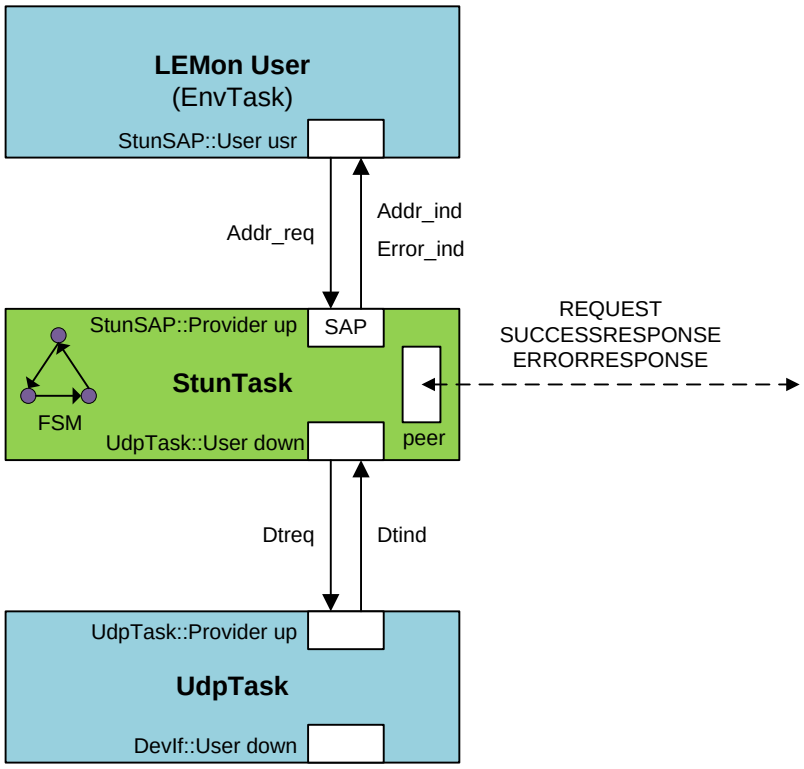


Figure 5.2: STUN Task support in DOORS

As explained before, the StunTask is the main creation of this architecture, and its implementation depends on three important formal protocol specifications:

- A *Finite State Machine* (FSM) to define the Task behaviour;
- A *Service Access Point* (SAP) to receive and transmit messages to the upper Task;
- A *peer* responsible to encode and decode *Protocol Data Units* (PDU);

The *Finite State Machine* (FSM) is a state machine that defines the behaviour of the task. The *Service Access Point* (SAP) is a DOORS port through where the StunTask sends/receives SDUs for/from the LEMon User. The port is called *up*, and the SDUs exchanged are:

- **Addr_req**: Address request. The user wants to send a STUN REQUEST and know its reflexive address;
- **Addr_ind**: Address indication. The StunTask received a SUCCESSRESPONSE and sends it to the user;
- **Error_ind**: Error indication. The StunTask received a ERRORRESPONSE and sends it to the user;

Lastly, the *peer* is the one responsible to encode and decode PDUs. The defined PDUs are:

- **Request**: It defines a STUN Request;
- **SuccessResponse**: It defines a STUN Successful Response;
- **ErrorResponse**: It defines a STUN Error Response;

The PDUs are encoded and automatically passed to the lower port *down*, to be sent using the format required by the lower layer (UdpTask).

5.1.4 Modular Interaction and Behaviour

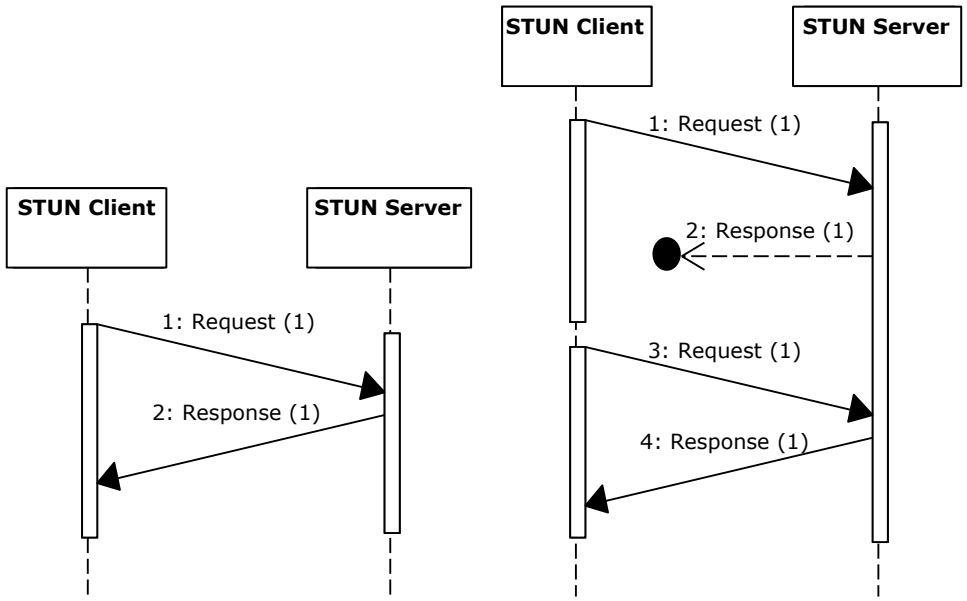
Traditionally, when designing a module in a system, it is required to evaluate the protocol to implement and use its specifications for the modulation itself. In this case, from the architecture shown before, the module to implement is the StunTask and it inherits from PTask. The basis of designing the StunTask specifications depends on the following UML and structure diagrams:

- Message Sequence Diagrams: Message sequence diagrams to define the network interactions between peers;
- *Protocol Data Unit* (PDU): Structure of messages to be exchanged with other peers;
- *Finite State Machine* (FSM): A state machine defining the client behaviour;

These formal protocol specifications are based on the specifications of the STUN protocol of chapter 3 and the RFC 5389 [3], and are explained below.

Message Sequence Diagrams

The communication between two peers in the STUN protocol is a Request/Response interaction. The STUN Client is the one interested to know its own reflexive address, and starts the communication sending a STUN REQUEST message to the STUN Server. The Server reads the REQUEST message and responds with a STUN RESPONSE message. The REQUEST message can be a SUCCESSFULRESPONSE or an ERRORRESPONSE. Regardless of the type of message, if the STUN Client receives the RESPONSE correctly, the STUN communication ends right there. The figure 5.3(a) illustrates this scheme.



(a) Message sequence diagram for normal flow (b) Message sequence diagram with retransmissions

Figure 5.3: The message sequence diagrams for the Stun Task

However, as stated before in section 5.1.1, the STUN protocol is operating over UDP, a non-reliable transport protocol. Therefore a message might be dropped by the network. For this reason, reliability of STUN transactions is accomplished through retransmissions of the REQUEST message. A STUN Client waits on a timeout for RESPONSES and retransmits the STUN REQUEST message if no message was received. This scheme is illustrated in figure 5.3(b).

Protocol Data Unit (PDU)

When sending messages through the network, it is important to define the content of each STUN message. In this section three different PDUs are illustrated: REQUEST, SUCCESSFULRESPONSE and ERRORRESPONSE.

As mentioned above in the message sequence diagrams, the client sends a REQUEST message to the server in order to ask its own reflexive address. As illustrated in figure 5.4, a STUN REQUEST message starts with the usual 20-byte header and it can be followed by a Software attribute. The Software attribute is optional and is just used for debug purposes. Therefore, this field is quite often empty.

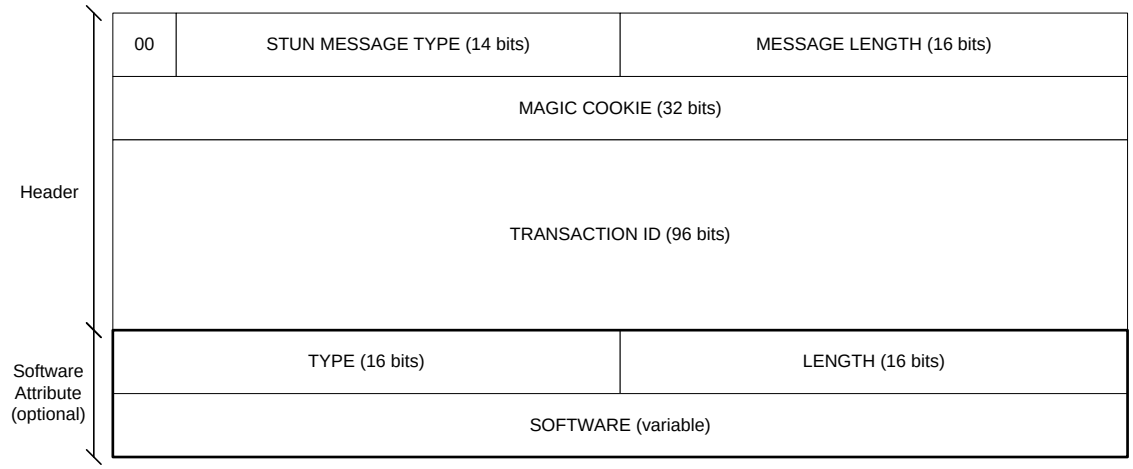


Figure 5.4: A STUN REQUEST message

After having received the REQUEST message, the server responds with a Response message: SUCCESSFULRESPONSE or ERRORRESPONSE.

If nothing fails and the communication occurs normally, the Client might receive a SUCCESSFULRESPONSE. As illustrated in figure 5.5 a SUCCESSFULRESPONSE starts with the usual 20-byte header and it can be followed by an XOR-Mapped-Address attribute that contains the client's reflexive address.

If something goes wrong the communication might fail, and the Client might receive an ERRORRESPONSE. As illustrated in figure 5.6 an ERRORRESPONSE starts with the usual 20-byte header and it can be followed by an Error-code attribute containing the error code and the reason of the error. For certain errors, additional attributes might be added to

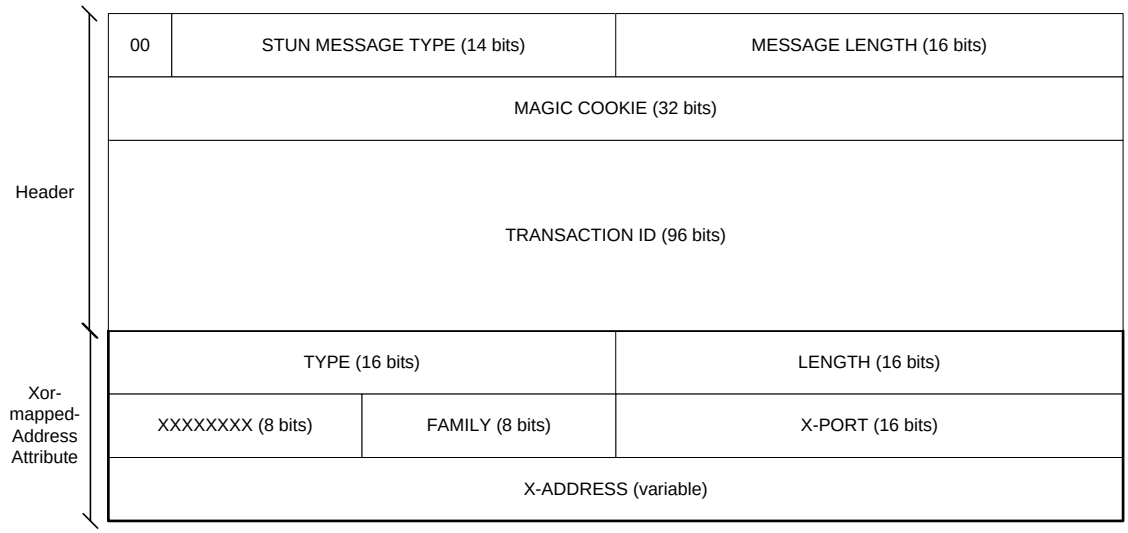


Figure 5.5: A STUN SuccessfulResponse message

the message, for example with the error 420 (Unknown Attribute).

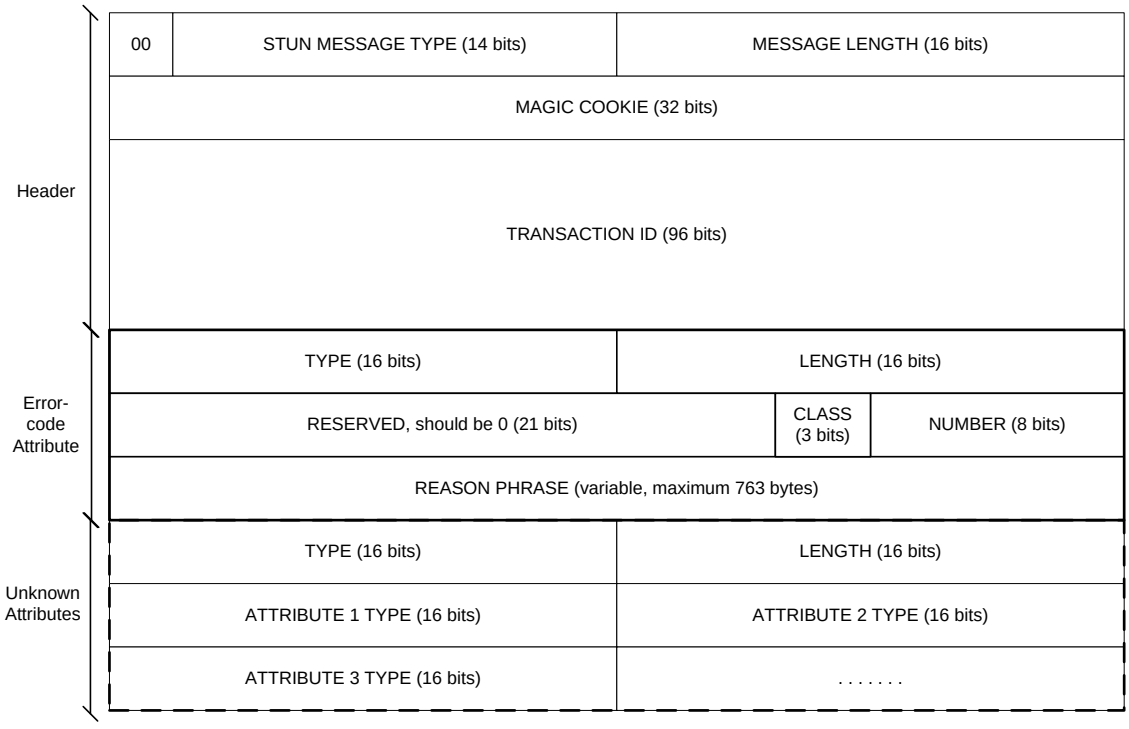


Figure 5.6: A STUN ErrorResponse message

Finite State Machine (FSM)

The state machine of the StunTask consists of two states. The initial state is "Idle"and the StunTask waits for a Addr_req to trigger the jump to the next state "Wait". When StunTask processes the trigger it will take two actions. It sends a REQUEST message to the peer and starts a timer.

After successful registration in the state "Wait", the StunTask waits for RESPONSES. If the Response is not correct it waits for more responses. If there is a timeout and the retry variable is less than five, then the StunTask should retransmit the REQUEST and restart the timer with the double of its initial time. StunTask will return to the initial state "Idle"under two conditions: if there was an error during the communication and no messages are received, then the fifth timeout triggers and StunTask should stop the timer and gets ready to try again; if the Response received was correct, then everything went well, and the StunTask should send a Addr_ind or a Error_ind to the LEMon user, depending on the type of message received from the peer. Therefore, the communication is completed and timer is set to off. The state machine of the StunTask is illustrated in figure 5.7.

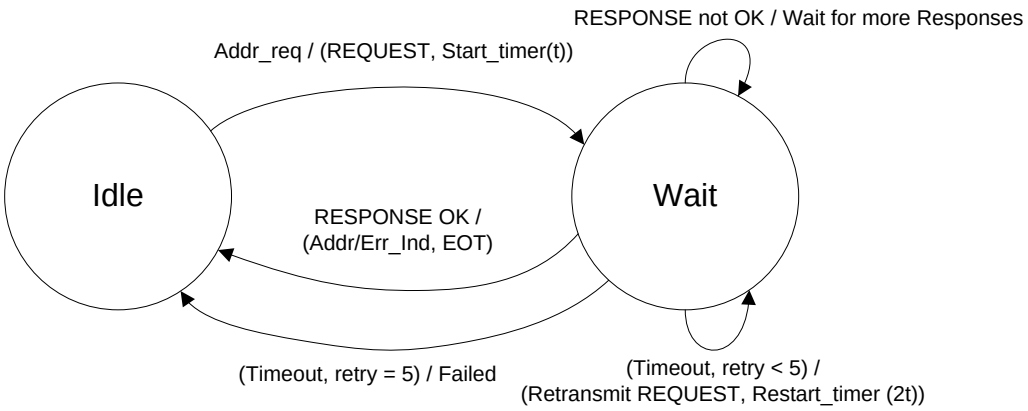


Figure 5.7: The state machine for the Stun Task

5.1.5 Implementation

The STUN protocol design presented in figure 5.2, focuses on the implementation of a StunTask module that is used to handle STUN requests and responses. The implemen-

tation can be divided in two parts: Firstly the XML specifications for the C++ code generators, and secondly the coding of the StunTask itself.

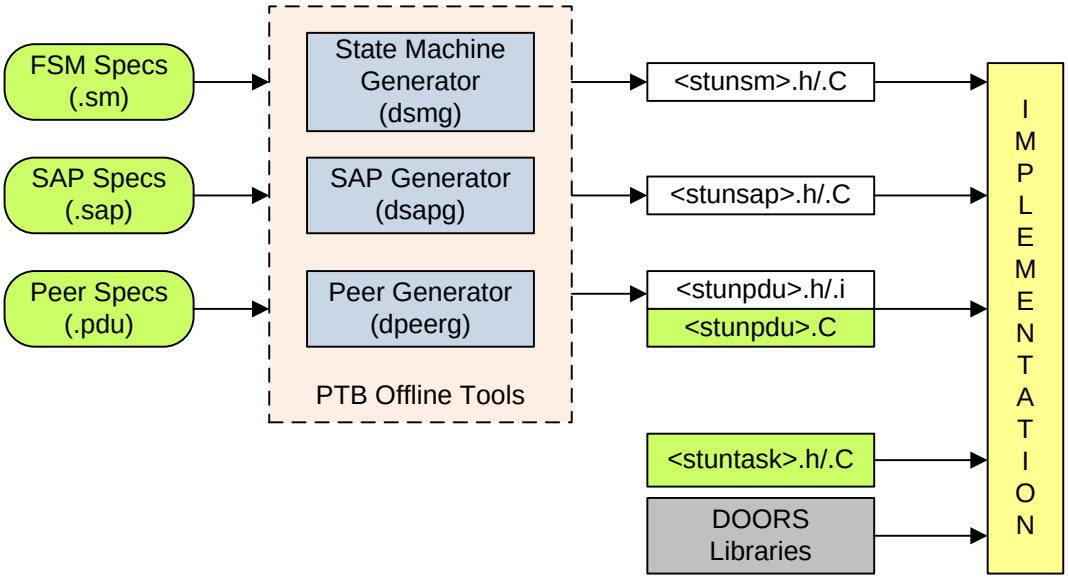


Figure 5.8: Stun Implementation

The implementation scheme is illustrated in figure 5.8. The files generated by the generators are C++ files and are used in the implementation.

XML-based Specifications

As explained in the section 5.1.3, the implementation of StunTask depends on three important formal protocol specifications: A *Finite State Machine* (FSM), a *Service Access Point* (SAP) and a *peer*. Based on the diagrams illustrated and explained in the subsection 5.1.4, it was possible to create the XML specifications to be supplied to the Code generators and transform them into C++ classes.

The PDUs were implemented using the *dpeerg* tool and its XML specification is included in Appendix A.1.1. The state machine was implemented using the *dsmg* tool and its XML specification is included in Appendix A.1.2. Finally, the SAP was implemented using the *dsapg* tool and its XML specification is included in Appendix A.1.3.

The generated files are part of the implementation. The stunpdu.C file is not generated, and it has to be manually coded, in order to give encode and decode functionality to the

PDUs. Therefore, all these files are used in implementation of the task, that is explained in the next sub-section.

StunTask Implementation

The StunTask inherits from PTask and it is the main task of this implementation. The job of this task is to give the main behaviour to the STUN protocol. With the help of the generated files and other DOORS functionalities, this task is manually coded taking care of incoming events and giving action to the behaviour defined in the figures of the section 5.1.4.

As shown in figure 5.2, StunTask inherits from PTask and it communicates with the LEMon user through the SAP port, and uses the *peer* to send PDUs. The StunTask class is included in Appendix A.2.

5.2 TLS Protocol

TLS is a security protocol to be used in the context of secure communications. It makes usage of cryptographic algorithms in order to offer security to Application layer protocols. In this section, the steps for the design and implementation of this security protocol in the DOORS framework are explained.

5.2.1 Considerations and Choices

Chapter 4 presented TLS as a security protocol. TLS uses cryptographic algorithms to offer a security layer between Application Layer and Transport Layer protocols. However, cryptographic algorithms and other features presented in the TLS protocol are quite difficult to implement and requires special attention. Thus, integrating a security protocol might require help from external libraries. Section 4.6.3 presented candidate technologies of TLS implementations and evaluated each implementation according to the following considerations:

1. Open Source library;
2. C/C++ *Application Programming Interface* (API) with proper documentation;
3. Implements the last version TLS 1.2;
4. Build automation tool with *make*;
5. Possibility of using callback functions;

As a result, GnuTLS is chosen over the others, mainly due to the fact that it has a good API and good documentation, but also due to the fact that it gives the possibility of using callbacks that might be used with event handlers in the DOORS framework. The implementation of the newest version of TLS and the usage of the build automation tool *make* are also relevant factors in this choice.

TLS is a client-server protocol to provide secure communications. Hence it needs a client and a server to exchange information. For this implementation it was decided to provide client support for the TLS protocol in the DOORS framework, expecting that during

a communication the other peer is the server that supports TLS and provide server responses.

In terms of operation, TLS protocol operates over TCP. It is possible to provide a security layer over UDP with *Datagram Transport Layer Security* (DTLS), but this protocol is out of the scope of this thesis. The goal of this implementation is to provide a TLS protocol operating over TCP for the most popular cases. The general idea is to give to the client the ability of creating a secure communication using the most typical cases. The most typical cases relies on RSA or Diffie-Hellman handshake sequences with server authentication using certificates. Thus, the implementation should support X.509 certificates, and it should be able to resume sessions as well.

Looking at the integration perspective, after considering several design ideas in the DOORS framework architecture, it was decided to support the TLS functionality with the help of GnuTLS. Since TLS is a complex client-server protocol, it was decided to create a DOORS virtual device for TLS communications over TCP, and a TLS task to test the protocol. The resultant design and implementation of this application should result in asynchronous and message-based interactions between modules. However asynchronous communications depends on the library functionalities, and some function calls might be blocking. For this reason, the handshake, send and receive data calls might block and not give a full asynchronous architecture. These matters will be discussed in the next sections.

For this implementation it was decided to give support to as many algorithms as possible, giving more flexibility to the programmer. Considering the GnuTLS library, it was decided to support the following algorithms and modes:

- **Certificate types**: X.509;
- **Protocols**: SSL3.0, TLS1.0, TLS1.1, TLS1.2;
- **Ciphers**: AES-256-CBC, AES-128-CBC, 3DES-CBC, DES-CBC, ARCFOUR-128, ARCFOUR-40, RC2-40, CAMELLIA-256-CBC, CAMELLIA-128-CBC, NULL;
- **MACs**: SHA1, MD5, SHA256, SHA384, SHA512, MD2, RIPEMD160, MAC-NULL;
- **Key exchange algorithms**: ANON-DH, RSA, RSA-EXPORT, DHE-RSA, DHE-

DSS, SRP-DSS, SRP-RSA, SRP, PSK, DHE-PSK;

- **Compression**: DEFLATE, NULL;
- **Public Key Systems**: RSA, DSA;
- **PK-signatures**: RSA-SHA1, RSA-SHA256, RSA-SHA384, RSA-SHA512, RSA-RMD160, DSA-SHA1, RSA-MD5, RSA-MD2.

5.2.2 Design Overview

This section explains the design prototyping for the TLS protocol in the DOORS framework based on the considerations of section 5.2.1. It starts off with the background on how this design came about, and in the next section it focuses in the details of the design itself.

TLS is a protocol that resides between the transport layer and application layer. For this reason it is important to create a module that can control the transport layer (i.e sockets). Thus, the presented design is approached from creating a TLS device and a TLS task to control the device. The TLS device and TLS task are the main creation of this architecture and are illustrated in figure 5.9.

The TLS device called *TlsConn* is the object responsible to initialize the socket for TCP communications, and is the one that makes use of the external library GnuTLS to take care of the security matters. The TLS task called *TlsTask* is the object responsible to use the functionalities of the device TlsConn and to give the behaviour of the TLS protocol to other DOORS tasks.

The TlsTask is the lowest layer task in the architecture. It is located right below a conceptual *Application Protocol* task in order to provide security to any application protocol that might be developed (e.g. HTTP).

The hypothetical *Application Protocol* task might need to communicate with a LEMon User task in order to provide testing of the protocol. The hypothetical exchanged messages can be called *App_req* and *App_ind*.

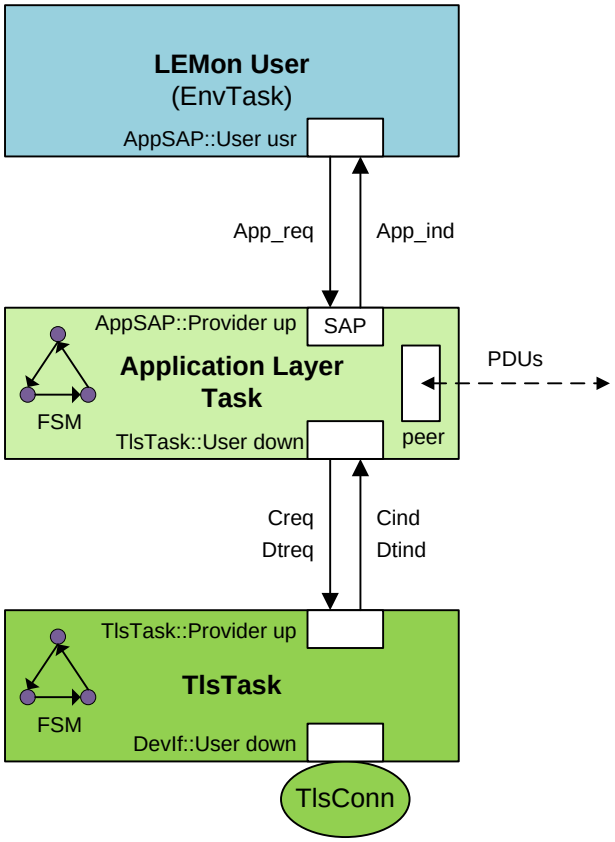


Figure 5.9: Architecture overview of TLS support in DORRS

Any application protocol that needs to communicate with the TlsTask might exchange the following messages:

- **Creq**: Connection request. An application protocol might send this message to the TlsTask in order to request for a TLS connection;
- **Dtreq**: Data transfer request. An application protocol might send this message to the TlsTask in order to send data, and depending on the protocol might wait for a response;
- **Cind**: Connection indication. Is the response to a Creq. Gives positive answer when a TCP connection and TLS handhshake are performed, and negative if some problem occurred;
- **Dtind**: Data transfer indication. It returns data received from the device. Usually is a response to a Dtreq.

When the TlsTask receives a message from the upper layer (Application Protocol), the TlsTask must use the TlsConn device to maintain the TLS connection.

To give behaviour to the TLS protocol, the TlsTask depends on a formal protocol specification: A *Finite State Machine* (FSM) that defines the task behaviour.

In terms of overall architecture in the DOORS environment, the tasks are controlled by the Scheduler, and the I/O Handler is responsible to manage the devices required for the communication.

5.2.3 Modular Interaction and Behaviour

Traditionally, when designing a module in a system, it is required to evaluate the protocol to implement and use its specifications for the modulation itself. As explained in the section 5.2.2 this implementation focus on the TlsConn device and the TlsTask. This section explains the modulation of the two modules.

TlsConn

TlsConn is designed to be a device in the DOORS environment, and to be responsible to control the transport layer in a TLS communication. For this matter, the table 5.1 describes different states of a TLS communication in two different perspectives: functionalities from the device perspective, and functionalities from the external GnuTLS library.

There are four main states. The first state is an *Initialization* state, where TlsConn should initialize all the variables and parameters necessary to stablish a TLS connection. The second state is called the *Connect* or *Open* state, where TlsConn first opens a socket with a TCP connection and after establishes a TLS session. From here on, the handshake is accomplished, and it is time to exchange data. The third state is called *Data Transfer* state, where it is possible to write/send data and read/receive data. Finally, the fourth state is called *Close* state, and is where the socket and TLS session are closed and other variables are deinitialized.

	State	Device	GnuTLS
1)	Init	Initialization of socket and local variables.	Initialization of credentials, certificates and other TLS session parameters.
2)	Open	Opens the socket and establishes TCP connection.	Performs a TLS handshake / Opens a TLS session.
3)	Write	Reads data received from the Task and writes it to a buffer.	Sends the buffer thru the present session.
	Read	Reads buffer and sends it to the Task.	Reads data from the present session and writes it to a buffer.
4)	Close	Close socket and deinitialization of local variables.	Close TLS session and deinitialization of credentials, certificates and other session parameters.

Table 5.1: TlsConn states

TlsTask

Other module to implement is the TlsTask and it makes use of the TlsConn device. TlsTask inherits from the DOORS PTask. The base of designing the TlsTask specifications depends on a *Finite State Machine* (FSM), a state machine that defines the TLS protocol behaviour. There are other protocol specifications that are hidden by the GnuTLS. Thus, it is not required to design PDUs and message sequence diagrams, since these are already used in the function calls of the library.

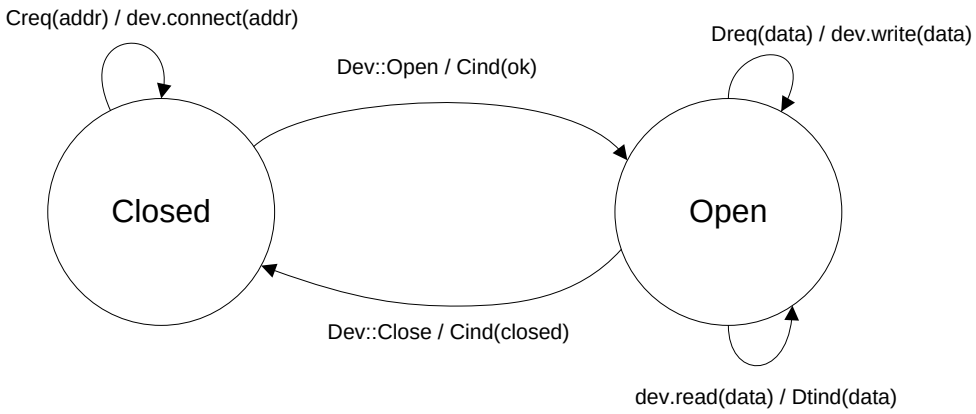


Figure 5.10: The state machine for the TlsTask

The state machine of the TlsTask consists of two states and is illustrated in figure 5.7.

The initial state, "Closed", is active when there is no TLS connection and the device is ready to use. TlsTask waits for a *Creq* from the Application protocol, calls the device to connect to the destination and waits for the device response. If the device connects successfully to the peer, it must send a "open"message to the TlsTask, and TlsTask sends a positive *Cind* message to the Application protocol and triggers the jump to the second state "Open".

While in this state "Open", the TlsTask waits for data from the Application protocol or from the device. If a *Dreq* is received from the Application protocol, then TlsTask must call the device to write and send the data to the destination. If data is received from the device, then TlsTask must send it in a *Dtind* message to the Application protocol.

Lastly, TlsTask returns to the first state when the device sends a "close"message to the TlsTask. This operation might happen if, for some reason, the peer closed the connection or if the TlsTask or device classes are destroyed.

5.2.4 Implementation

The TLS protocol design presented in figure 5.9 focuses on the implementation of a TlsConn device and StunTask task modules to handle TLS connections. For this approach, the implementation can be divided in two parts. Firstly, the TlsConn device class and secondly the TlsTask task which includes XML code and the TlsTask class itself.

TlsConn

The class TlsConn is the core of the DOORS virtual device implementation for TLS. It makes use of the GnuTLS library and it is used to provide useful TLS functionalities to the TlsTask. This means that TlsTask connects to another application by encapsulating the functionalities provided by GnuTLS and TlsConn, as described in section 5.2.3. This class is derived from class SocketConnectionAC and SockHelper which are provided in DOORS.

As described in the section 5.2.3, TlsConn is designed to provide functionalities to dif-

ferent states of TLS-based communications. The following paragraphs will give a brief explanation of the most important public functions to be used.

The TlsConn class is included in Appendix B.3. Not all the functionalities provided by TlsConn and GnuTLS are used since they might not all be needed. Firstly, the Initialization provides the instances of the DOORS I/O handler and the driver task to the device instance.

```
class TlsConn : public SocketConnectionAC, public SockHelper {
  public:
      TlsConn (IoHandler *io, EventTask *t);
      ~TlsConn (void);
}
```

There are extra initializations that might be needed to a TLS connection. Depending on the protocol and user's demands, the following functions might be used. The extra initialization functions are listed below.

```
bool tls_init_params(char *priority_list, int debug, int insecure, int
    disable_ext);
bool tls_init_x509 (char *cafile, char *crlfile, char *keyfile, char *
    certfile);
bool tls_init_srp (char *srpuser, char *srppass);
bool tls_init_psk (char *pskuser, char *pskkey);
```

tls_init_params() is used to initialize priority lists, debugging mode, insecure mode (to skip some verifications) and extensions. The priority list refers to the cipher suite preferences, and its initialization may contain some high level keyword that can be found at GnuTLS documentation [61]. *tls_init_x509()* might be used to initialize parameters required for X.509 certification, and might indicate the path of the following files: Certification Authority List, Certificate Revocation List, Private Key and Client Certificate. If the pointer is set to *NULL* to any of the arguments, the device assumes that the file does not exist. The function *tls_init_srp()* is used to initialize username and password for the SRP protocol, and *tls_init_psk()* for the PSK protocol.

The connect functions listed below are used in the *Open* state, where the device actually establishes a TLS connection with another application:

```
bool connect (Address *a);
bool connect (Address *a, Frame sessiondata);
```

The first *connect()* is used to connect to a specified address for the first time, meaning that the device establishes a session for the first time. The second *connect()* call might be used to connect to a specific address using Session Resuming capability, meaning that it resumes a session established before.

The third state is the Data Transfer state, where actually the device is already connected to a peer. The functions are listed bellow.

```
virtual int getFd (void) const;
Frame getSession (void);
void callbackWrite (void);
void callbackRead (void);
```

When a TLS connection is established with a peer, the connection file descriptor is stored and set to read mode. This means that when the I/O handler runs, it checks all the devices that are read enabled, and retrieves the associated file descriptor for them. *getFd()* is a simple function call that retrieves the file descriptor. When the *read* mode is enabled, some data is available to be read for this device, the I/O handler calls the *callbackRead()* and it saves the data in a buffer and sends it to the TlsTask. When the *write* mode is enabled, there is some data to be sent and the I/O handler calls the *callbackWrite()* to send the data to the destination. Lastly, *getSession()* saves the current session data in a Frame. This function call might be used to store the session data, and can be used later for Session Resuming.

The last state is the Close state. The function is listed below.

```
virtual bool close (void);
```

This virtual call *close()* is actually called when the device disconnects from the peer. For some reason some error might have occurred and the peer might have disconnected or the TlsTask might want to close the connection. This function call takes care of the file descriptors and deinitializes all the variables and settings set previously.

There are more functions in the TlsConn class, but they are less significant for the TlsTask to work properly. Many of them are public and other private, but all are used to internally manage the device and the GnuTLS library.

TlsTask

The TlsTask is the main task of this implementation. The job of this task is to use TlsConn device functions and give the main behaviour to the TLS protocol.

The implementation of TlsTask depends on two important XML specifications: A *Finite State Machine* (FSM) and a *Service Access Point* (SAP). Based on the diagram illustrated and explained in subsection 5.2.3, it was possible to create the XML specifications to be supplied to the Code generator and transform it into C++ classes. The state machine was implemented using the *dsmg* tool and its XML specification is included in Appendix B.1.1. The SAP was implemented using the *dsapg* tool and its XML specification is included in Appendix B.1.2.

TlsTask inherits from PTask and it communicates with the Application Protocol Task from the SAP port, as shown in figure 5.9 and its class is included in Appendix B.2.

5.3 Development Environment

This section describes related components, tools and systems which were used to implement the designed prototypes explained in the sections above. The work was done mostly under C/C++ programming language in the Linux environment. Also several tools were used for project management, testing and debugging.

Both STUN protocol and TLS protocol were developed and tested in the GNU/Linux operating system. In this case Fedora Core 12 distribution was used over the kernel version 2.6.32.26-175. The tools used for STUN and TLS implementation are listed in table 5.2. The external libraries are listed in table 5.3.

Protocol	Name	Version	Description
STUN and TLS	Gedit	2.28.3	Gnome editor
	GCC	4.4.4	GNU Compiler Collection
	GNU Make	3.81	Generation of executables
	GNU DDD	3.3.12	Data Display Debugger
	Wireshark	1.2.11	Network protocol analyser
STUN	reTurnServer	0.4	STUN/TURN Server component of the reSIProcate projects
TLS	Certtool	2.10.2	GnuTLS tool to generate X.509 certificates
	ssldump	0.9b3	SSLv3/TLS network protocol analyzer

Table 5.2: Tools used for Implementation

Protocol	Name	Version	Description
STUN and TLS	DOORS	0.6	DOORS framework
TLS	GnuTLS	2.10.2	GNU Transport Layer Security Library
	Libgcrypt	1.4.6	GNU's library of cryptographic building blocks
	Libtasn1	2.8	ASN.1 library
	Zlib	1.2.5	Compression/Decompression library
	libc	2.11.2	GNU C library
	libstdc++	6.0.13	GNU Standard C++ Library v3

Table 5.3: Libraries used for Implementation

Chapter 6

Implementation Testing and Analysis

Testing is the process of trying to find errors in the system implementations by means of experimentation. This experimentation is carried out in a special environment in order to simulate normal and exceptional procedures of the protocols. The aim of testing is to gain confidence and make sure that the system works satisfactorily during the normal and exceptional procedures. In other hand, testing only shows the presence of errors, not their absence, so it is not possible to ensure complete correctness. In this case, a successful test of the implementations means that the protocol is working correctly for the most typical cases, and is almost ready to fully integrate in the DOORS framework. *Almost ready to integrate* means that the implementation works satisfactorily in the framework, and can be fully integrated as soon as a specialist validates the feasibility of its usage. In the next sections some test cases for each implementation are presented. Corresponding test results will be presented based on the performance of the various tested modules.

6.1 Test Network

This section presents the topology of the network used for the testing of the protocols. The testing of the protocols were performed in the TLT network at the Department of Communications Engineering at *Tampere University of Technology* (TUT). Figure 6.1 illustrates the principal layout of the network used.

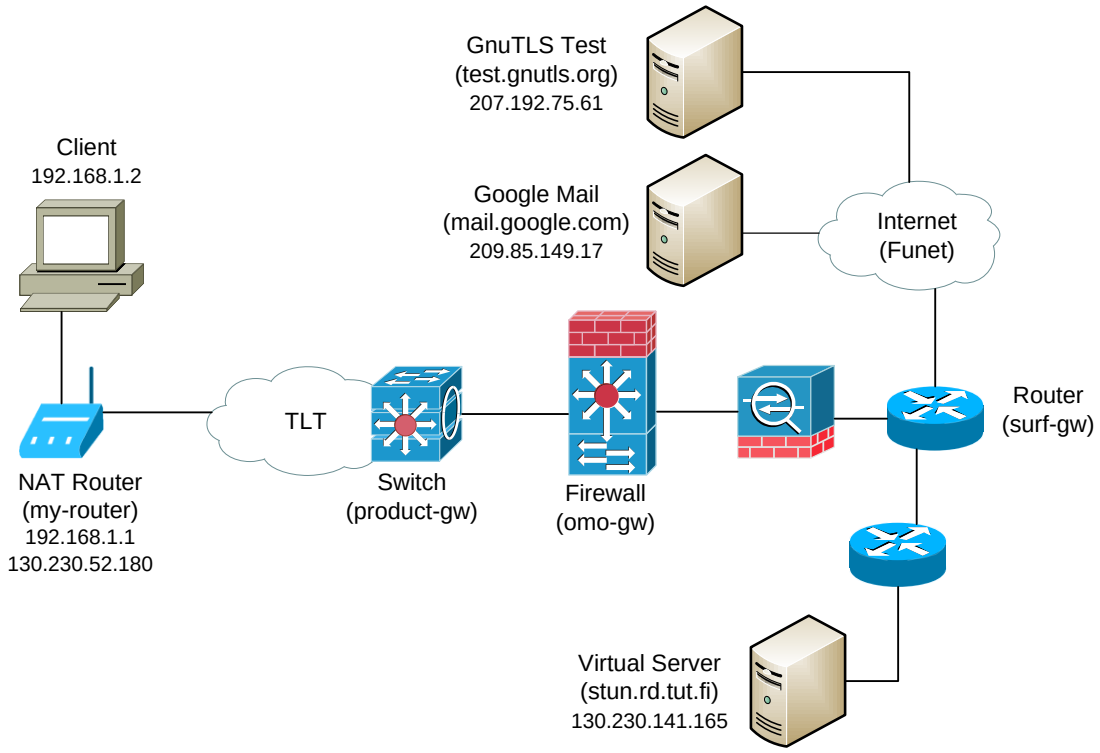


Figure 6.1: Test Network

The computer used as a client in the whole testing process is the same used for developing the protocol: a HP COMPAQ Pentium 4 running the linux distribution Fedora Core 12. The computer is connected to an Asus WL-500W router with NAT enabled, and has the IP address 192.168.1.2. The router is connected to the TLT network and has the static public IP 130.230.52.180. For the client to contact the Server, it has to pass thru the product-gw switch, the omo-gw firewall and the router surf-gw. The Virtual Server stun.rd.tut.fi with the public IP address 130.230.141.165 is a Ubuntu virtual machine connected to a router still inside TUT network. Other possible Servers such as Google Mail Server and GnuTLS Test Server are available on Internet and are used for testing as well.

6.2 STUN Analysis

This section presents the testing of the STUN protocol. Firstly, a general test case is described and then there are some important details of the results that validate the proposed test case.

6.2.1 Test Case

Considering the network of the section 6.1, the STUN client was tested in the TLT network and was connected with a router with NAT enabled. The STUN client gets the private IP address 192.168.1.2 via DHCP. The STUN server is a Virtual Server (stun.rd.tut.fi), a reTurnServer listening in the UDP port 3478 [62]. The test case is illustrated in figure 6.2.

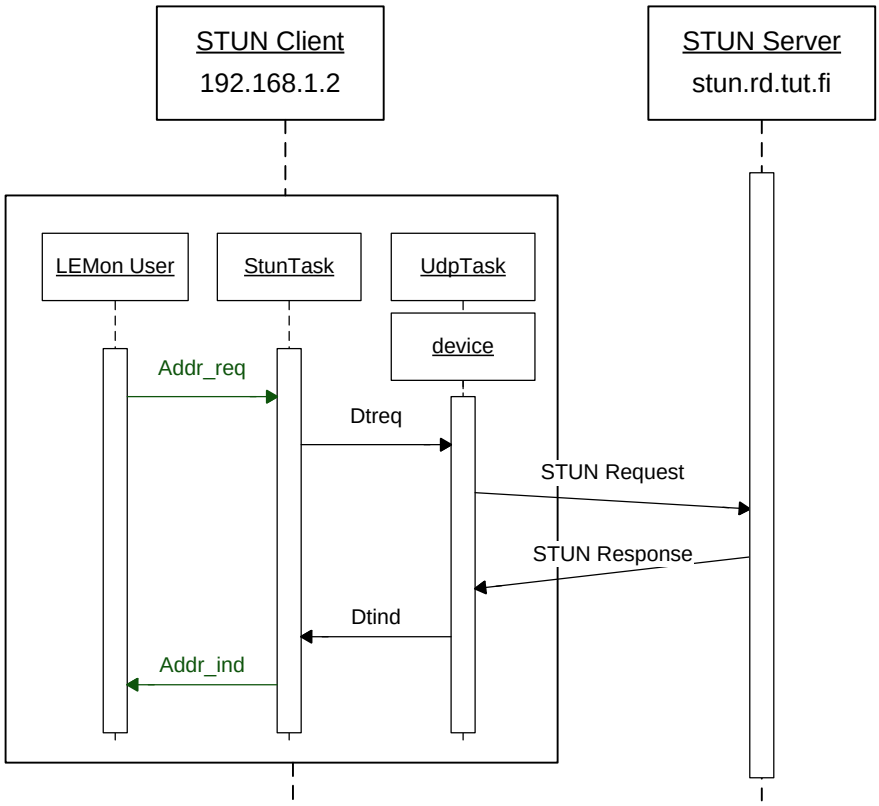


Figure 6.2: Test STUN

Considering the implementation of the protocol in section 5.1, the Client must use a LEMon user task to communicate with the STUN task. The client must send a Addr_req

to the STUN task and should wait for a Addr_ind response.

When testing a protocol some problems can occur. The encoding of the PDUs and other network errors are some examples of problems that might occur. This test case is validated if the Addr_ind response is received properly without any errors.

6.2.2 Results

This section presents the results and their validation for the test cases of the previous section. Firstly the output of the LEMon User and the content of the received messages will be presented. The results from the network were captured using the well known Wireshark, a network protocol analyzer.

```
***** stun BEFORE run 19:33:20.015148
*** Message from env:usr to stun:up ***
    Message 'addr_req' = {
        Frame data = request
    }

***** env BEFORE run 19:33:20.017114
*** Message from stun:up to env:usr ***
    Message 'addr4_ind' = {
        InetAddr req_addr = 130.230.52.180:39950
    }
```

Figure 6.3: A partial capture of the LEMon User output during the STUN test

No.	Time	Source	Port	Destination	Port	Protocol	Info
1	0.0000	192.168.1.2	39950	130.230.141.165	3478	STUN2	Binding Request
2	0.0014	130.230.141.165	3478	192.168.1.2	39950	STUN2	Binding Success Response

Figure 6.4: A partial screen capture of Wireshark, during the STUN test

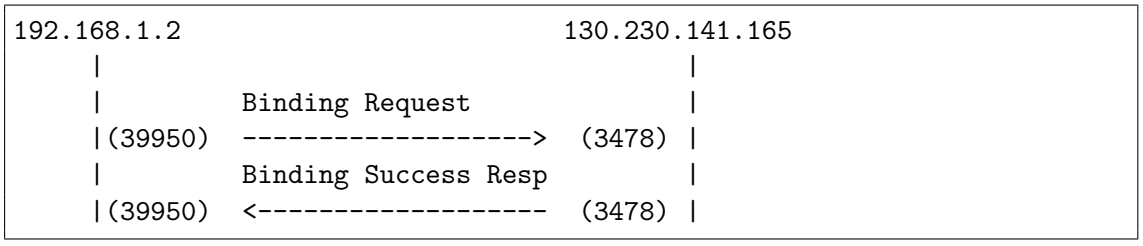


Figure 6.5: Graph Analysis of Wireshark, during the STUN test

During the STUN test it was possible to get the following results: Figure 6.3 illustrates a

partial capture of the LEMon user output; Figure 6.4 illustrates a partial screen capture of the network analyser Wireshark, detailing the messages sent by the agents and captured by the device; Figure 6.5 illustrates a graph analysis generated by Wireshark.

The Client sends an Addr_req and receives a successful Addr_ind. From figure 6.3 it is possible to verify the behaviour of the implementation, checking that the message Addr4_ind contains the reflexive address for the STUN client:

- **Public IP Address**: 130.230.52.180
- **Public Port**: 39950

From figure 6.4 and figure 6.5 it is possible to analyse the packets and validate the protocol, since Wireshark automatically recognizes the protocol as *STUN2* and in the Information column it gives the type of the STUN message.

Comparing figure 6.3 and figure 6.4 it is possible to conclude that when the client sends a UDP message, the NAT router uses the same source port 39950 (if available), and changes the source IP to its public address 130.230.52.180 to transmit the packet.

6.3 TLS Testing

This section presents the testing of the TLS protocol. Firstly a general test case is described and then some important details of the results that validates the test case are proposed.

6.3.1 Test Case

First of all, testing the TLS protocol involves the choice of an Application protocol to combine with security. In this test case, a HTTPS test case is provided, which is a combination of the HTTP protocol with the SSL/TLS protocol.

Looking at the design perspective, figure 6.6 illustrates the simple test case chosen to test HTTPS.

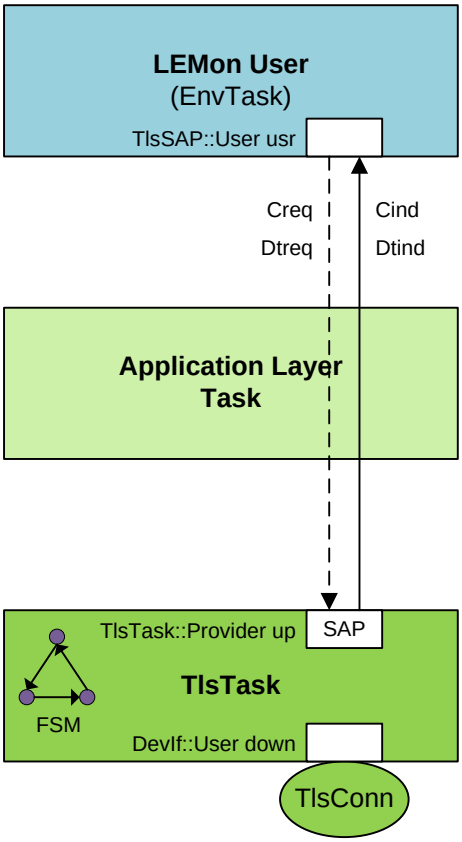


Figure 6.6: Test TLS model

Since HTTP is a simple request/response protocol with simple PDUs that only sends string commands to the server, it was decided to skip the implementation of a HTTP task, and simply leave the LEMon User to communicate straight with the TlsTask. Using this structure, the LEMon user only needs to use the TlsTask SAP to send SDUs. It only needs to connect to the Server and send string commands. Figure 6.7 illustrates a message sequence diagram of the TLS test case.

Considering the network architecture of the figure 6.1, the TLS client is tested in the TLT network and is connected to a router with NAT enabled. The TLS client gets the private IP address 192.168.1.2 via DHCP. It is quite important to note that the usage of the HTTP protocol in the TLT network is quite vulnerable, either because the client is in a private LAN, and also because of the TLT network where there are many other computers connected. If there is the necessity of sending sensitive information, the packets can be captured by a eavesdropper, and secrets can be discovered. Man-in-the-middle attacks are other often common attacks, where an attacker fools both parties of a communication.

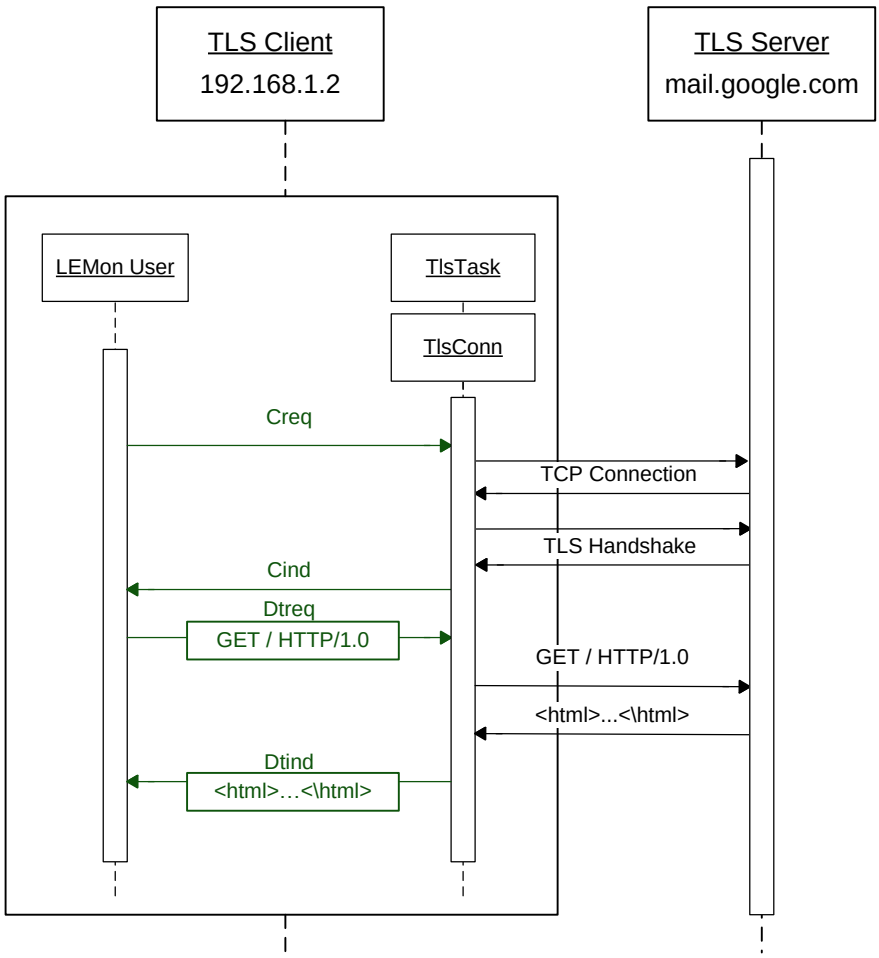


Figure 6.7: Test HTTP over TLS

Thus it is decided to test the HTTPS protocol and the chosen HTTPS test server is the Google Mail Server (mail.google.com) listening in the TCP port 443.

Considering the message sequence diagram of figure 6.7, the Client must use the LEMon user task to communicate with the TlsTask. First it sends a Creq to connect to the server and waits for the confirmation Cind. When the TLS connection is established it sends a Dtreq message to send a GET command to the Server. In the HTTP protocol, a GET command requests a representation of the specified resource. If everything goes well the Server might send a response and the LEMon user receives it back in the message Dtind.

When testing a protocol some problems can occur. Connection problems, not accepted ciphers, error in the validation of certificates, and network errors are some examples of

problems that might occur. This test case is validated if the Client successfully connects to the Server, and if it is able to send and receive responses without any errors.

6.3.2 Results

This section presents the results and its validation for the test cases of the previous section. At first it presents the output of the LEMon User and the content of the received messages. Then, it presents the results captured from the Network using ssldump, a SSLv3/TLS network protocol analyzer.

During the TLS test it was possible to get the following results: Figure 6.8 illustrates a partial capture of a Connection Request in the LEMon user; Figure 6.9 illustrates a partial capture for a Dtreq and Dtind in the LEMon user; Figure 6.10 illustrates the output of the network analyser ssldump, detailing the messages sent by the agents and captured by the device.

```
***** tls BEFORE run 21:28:24.477376
*** Message from env:usr to tls:up ***
    Message 'creq' = {
        Frame priorities = SECURE256
        Frame cafile = certificates/x509-trust.pem
    }

***** env BEFORE run 21:28:25.115928
*** Message from tls:up to env:usr ***
    Message 'cind' = {
        InetAddr addr = 209.85.149.83:443
        Frame message = TLS Connected!
    }
```

Figure 6.8: A partial capture of a Connection Request in the LEMon user during the TLS test

From the DOORS perspective, the LEMon User sends a Creq and receives a successful Cind. From figure 6.8 it is possible to validate the connection request and connection indication, meaning that the TLS Session/Connection was successfully established. From that on, the application data transfer starts with a Dtreq, when the client runs the command "GET / HTTP/1.0"and receives two Dtind messages. From figure 6.9 it is possible to validate the application data transferred during the TLS session, meaning that the Server

```
***** tls BEFORE run 21:28:31.295547
*** Message from env:usr to tls:up ***
    Message 'dtreq' = {
        Frame data = GET / HTTP/1.0
    }

***** env BEFORE run 21:28:31.329310
*** Message from tls:up to env:usr ***
    Message 'dtind' = {
        InetAddr source = 209.85.149.83:443
        Frame data = HTTP/1.0 302 Found
        Location: https://encrypted.google.com/
        Cache-Control: private
        Content-Type: text/html; charset=UTF-8
        Set-Cookie: PREF=ID=4d6097fcb0aa379a:FF=0:TM=1311008591:LM
            =1311008591:S=bdGrjSR1Rh2r1TZM; expires=Wed, 17-Jul-2013
            17:03:11 GMT; path=/; domain=.google.com
        Date: Mon, 18 Jul 2011 17:03:11 GMT
        Server: gws
        Content-Length: 226
        X-XSS-Protection: 1; mode=block
    }

***** env BEFORE run 21:28:31.336593
*** Message from tls:up to env:usr ***
    Message 'dtind' = {
        InetAddr source = 209.85.149.83:443
        Frame data =
        <HTML><HEAD><meta http-equiv="content-type" content="text/html;
            charset=utf-8">
        <TITLE>302 Moved</TITLE></HEAD><BODY>
        <H1>302 Moved</H1>
        The document has moved
        <A HREF="https://encrypted.google.com/">here</A>.
        </BODY></HTML>
    }
```

Figure 6.9: A partial capture of a Dtreq and Dtind in the LEMon user during the TLS test

receives the requests and answers accordingly to the protocol, and the device successfully decodes the messages to plaintext.

From figure 6.10 it is possible to analyse the packets and validate the protocol, since ssldump automatically recognizes the SSLv3/TLS protocol and recognizes the type of each message. From this output is also possible to recognize the new tcp connection, the

```
New TCP connection #1: pedro.tut(65000) <-> mail.google.com(443)
1 1  0.1374 (0.1374)  C>S  Handshake      ClientHello
1 2  0.1676 (0.0302)  S>C  Handshake      ServerHello
1 3  0.1677 (0.0000)  S>C  Handshake      Certificate
1 4  0.1677 (0.0000)  S>C  Handshake      ServerHelloDone
1 5  0.3504 (0.1826)  C>S  Handshake      ClientKeyExchange
1 6  0.4195 (0.0691)  C>S  ChangeCipherSpec
1 7  0.4195 (0.0000)  C>S  Handshake
1 8  0.4489 (0.0293)  S>C  ChangeCipherSpec
1 9  0.4489 (0.0000)  S>C  Handshake
1 10 6.8182 (6.3693)  C>S  application_data
1 11 6.8495 (0.0313)  S>C  application_data
1 12 6.8495 (0.0000)  S>C  application_data
1    6.8496 (0.0001)  S>C  TCP FIN
1 13 6.8530 (0.0034)  C>S  Alert
1    6.8531 (0.0000)  C>S  TCP FIN
```

Figure 6.10: Messages sent by the agents and captured by ssldump during the TLS test

successful handshake, the application data transfer, and the connection closing.

Even though it is not shown in the outputs, the TLS handshake is performed with the following parameters:

- **Key Exchange**: RSA
- **Protocol**: TLS1.0
- **Certificate Type**: X.509
- **Compression**: NULL
- **Cipher**: ARCFOUR-128
- **MAC**: SHA1

The application data transfer is encrypted with ARCFOUR-128 and cannot be understood by an eavesdropper that might attack the network. Using TLS with PKI and certificates, and the data transfer fully encrypted with a symmetric algorithm and MAC, it is provided a true sense of security with authentication, confidentiality and integrity.

Chapter 7

Conclusions

The current chapter discusses the thesis' conclusions based on previous chapters. Section 7.1 is a small synthesis where the main contents of each chapter are highlighted and section 7.2 summarizes the main conclusions. Section 7.3 enumerates a few topics to address in future investigation.

7.1 Synthesis

This section briefly describes each chapter's content.

Chapter 1 explains that the scope of this thesis is the investigation and integration of the STUN and TLS protocols in the DOORS framework environment. It also highlights their architecture and functionalities for their usage and further developments.

Chapter 2 describes the DOORS framework architecture and its usage. Moreover, it presents the general system functionalities that are used in the following chapters.

Chapter 3 overviews the NAT traversal problem and the STUN protocol as a tool to improve protocols in penetrating the NAT. This is an important chapter because it explains the importance of NAT traversal and gives some important information of the STUN specifications used for the further chapters.

Network security is discussed in Chapter 4. Here, it is discussed relevant aspects of the

usage of security in different layers on the TCP/IP stack, and important services to provide when securing a network. Further, it provides the importance of cryptography and some algorithms and methods to provide different services in network security. Finally the TLS protocol is described as a final solution to secure the transport layer, merging all the algorithms and methods discussed previously.

Chapter 5 describes the protocols' implementation. STUN and TLS protocols integration in the DOORS framework is the aim, illustrating important diagrams for its design and many other specifications for its final usage.

Chapter 6 validates the protocols' functionality. It discusses the validity of the STUN and TLS protocol in the DOORS framework presenting their usage with some outputs and network flows and compares the obtained results with the theoretical specifications.

7.2 Conclusions

The main objective of this thesis is focused on the study and integration of two different protocols for the DOORS framework.

The importance of NATs is a crucial theme to some protocols that might not work properly. DOORS as a framework for implementing protocols and network applications was not prepared for this problematic. New protocols or applications developed in the DOORS framework might be working behind a NAT, and the necessity of a NAT traversal solution might be useful. For this purpose, a design for the STUN protocol was proposed in the DOORS framework, to be used by developers as a tool to help in NAT traversal. Using this implementation, it is possible to enable applications to communicate with the *StunTask* presented in the section 5.1 to discover their reflexive address. It is proved with section 6.2 that the presented design is feasible in the DOORS framework and we can ensure that any application protocol can use *StunTask* to help NAT traversal solutions. With this design it is possible to prove the idea of loosely de-coupled objects that handle incoming events and communicate with other objects using messages. This section also tested an application sending messages to the *StunTask*, and receiving responses accordingly to the

protocol specifications and the DOORS requirements. Since NAT traversal only makes sense for IPv4, the STUN solution only works for IPv4 addresses, but in the future IPv6 can be easily integrated.

The importance of security is a crucial theme for most of the protocols exchanging messages in an insecure channel. New protocols or applications developed in the DOORS framework might need to provide an extra layer of security. The goal is to provide security to application protocols at the transport layer in the DOORS framework, encapsulating the protocol through an encrypted SSL/TLS connection. For this purpose, it was proposed the integration of the TLS protocol in DOORS framework with the help of the GnuTLS library, and its design is presented on section 5.2. The solution for this protocol goes to a lower level architecture and focuses on an implementation of two objects, a TLS device and a TLS task. It is proved with section 6.3 that the presented design is feasible in the DOORS framework and we can ensure that any application protocol can use *TlsTask* to begin a secure communication in the transport layer. With this design we proved that it is possible to use external libraries to integrate new protocols in the DOORS framework. Using a virtual device for the external communication and the usage of the TlsTask to control the device was definitely an effective solution. To prove the funcionality of this architecture it was decided to simulate the HTTPS protocol. The application layer protocol HTTP communicates successfully with the *TlsTask*, and a TLS connection is established providing a security layer to the HTTP protocol. After the successfull handshake, the application protocol sent a GET command, and received responses accordingly to the protocol specifications. This implementation supports IPv4 and IPv6 addresses. In terms of DOORS requirements, the GnuTLS library uses some blocking calls causing some synchronization problems to DOORS. However, if a full asynchronous environment is not necessary, it works perfectly.

7.3 Future Work

During the development of the proposed solutions, some assumptions were made to support our architecture models.

Firstly for the case of the STUN protocol, it was decided to support UDP for the most typical cases using IPv4 addresses. In the future this model can be extended for TCP connections and IPv6 addresses. More testing and performance evaluation is needed to ascertain that it is feasible and beneficial in practice.

Further work in the TLS integration can be conducted to improve the asynchronism of the architecture. GnuTLS function calls such as *gnutls_handshake()*, *gnutls_record_send()* and *gnutls_record_recv()* are blocking calls that do not allow a full asynchronous architecture in the DOORS framework. The goal must be to use callback functions instead and give a full asynchronous architecture to fully integrate the DOORS requirements.

Appendix A

Appendixes to the STUN Implementation

A.1 XML Specifications

A.1.1 Peer/PDU Specifications

```
<?xml version="1.0"?>

<Peer Name="StunPeer" HIncludeFiles="stunheader.h, doors/inetaddr.h"
    IIncludeFiles="stunpdu.h">

   <Message Name="REQUEST">
      <Parent>STUNHeader</Parent>
      <Field> Uint16 type </Field>
      <Field> Uint16 length </Field>
      <Field> Frame software </Field>
      <Field> InetAddr addr </Field>
   </Message>

   <Message Name="SUCCESSRESPONSE">
      <Parent>STUNHeader</Parent>
      <Field> Uint16 type </Field>
      <Field> Uint16 length </Field>
      <Field> Uint8 reserved </Field>
      <Field> Uint8 family </Field>
      <Field> Uint16 xport </Field>
      <Field> Uint32 xaddress4 </Field>
      <Field> Frame xaddress</Field>
```

```
      <Field> InetAddr srcIP </Field>
      <Field> InetAddr destIP </Field>
   </Message>

   <Message Name="ERRORRESPONSE">
      <Parent>STUNHeader</Parent>
      <Field> Uint16 type </Field>
      <Field> Uint16 length </Field>
      <Field> Uint16 reserved </Field>
      <Field> Uint8 reserved_class </Field>
      <Field> Uint8 number </Field>
      <Field> Frame reason_phrase</Field>
      <!-- new attribute -->
      <Field> Frame unknown_attr </Field>

      <Field> InetAddr srcIP </Field>
      <Field> InetAddr destIP </Field>
   </Message>

</Peer>
```

A.1.2 State Machine Specifications

```
<?xml version="1.0"?>

<SM Name="StunTaskSM" PTask="StunTask" CIncludeFiles="stunsm.h, stuntask.h
    , stunsap.h, stunpdu.h">
<SAP-File Name="stunsap.sap"/>
<Peer-File Name="stunpdu.pdu"/>

<From StunSAP="up"/>
<From StunPeer="peer"/>

 <State Name="Idle" Default="idle_Default">
  <Interface Name="up">
   <Input Name="Addr_req">idle_Addr_req</Input>
  </Interface>
 </State>

 <State Name="Wait" Default="wait_Default">
  <Interface Name="peer">
   <Input Name="SUCCESSRESPONSE">wait_SUCCESSRESPONSE</Input>
  </Interface>
  <Interface Name="peer">
   <Input Name="ERRORRESPONSE">wait_ERRORRESPONSE</Input>
  </Interface>
  <Timer>wait_Timeout</Timer>
 </State>

</SM>
```

A.1.3 Service Access Point Specifications

```
<?xml version="1.0"?>

<SAP Name="StunSAP" CIncludeFiles="stunsap.h">

   <User>
      <Message Name="Addr_req">
         <Field>Frame data</Field>
      </Message>
   </User>

   <Provider>
      <Message Name="Addr4_ind">
         <Field>InetAddr req_addr</Field>
      </Message>

      <Message Name="Error_ind">
         <Field>Frame data</Field>
      </Message>
   </Provider>

</SAP>
```

A.2 StunTask Class Code

```
#ifndef STUNTASK_H
#define STUNTASK_H

#include <string>
#include <doors/ptb.h>
#include <doors/udp.h>
#include <doors/ptask.h>
#include <doors/inetaddr.h>
#include <doors/timer.h>
#include "stunsap.h"
#include "stunsm.h"
#include "stunpdu.h"
#include "stunheader.h"
#include "stundef.h"

#define MAX_RETRIES 5

class StunTaskSM;

class StunTask : public PTask {

public:
   StunTask (std::string name, Scheduler* s, StunTaskSM* sm, InetAddr addr
       );

   virtual ~StunTask();

   UdpSAP :: User down; // SAP to UDP task
   StunSAP :: Provider up; // SAP to a user task
   StunPeer peer; // SAP to Stun peer

   // Idle state
   bool idle_Default (Message* msg);
   bool idle_Addr_req (Message* msg);

   // Wait state
   bool wait_Default (Message* msg);
   bool wait_SUCCESSRESPONSE (Message* msg);
   bool wait_ERRORRESPONSE (Message* msg);
   bool wait_Timeout (Message* msg);

   // Helper functions
private:
   void sendRequest ();
   void retransmitRequest ();
   void receiveSuccessResponse (Message* msg);
   void receiveErrorResponse (Message* msg);
```

```
    bool is_valid_header (STUNHeader header);

    // Class variables

    InetAddr destaddr_; //destination addr structure (contain just the
        value, but we don't use yet)
    Timer timer_; //timer
    Uint16 retries_; //retries

    TidStruct tid_; //transaction_id sent in the REQUEST
    StunPeer :: REQUEST *buffer_msg; //message sent in the REQUEST

};

#endif
```

Appendix B

Appendixes to the TLS Implementation

B.1 XML Specifications

B.1.1 State Machine Specifications

```
<?xml version="1.0"?>

<SM Name="TlsTaskSM" PTask="TlsTask" CIncludeFiles="tlssm.h, tlstask.h,
    tlssap.h">
  <SAP-File Name="tlssap.sap"/>
  <From TlsSAP="up"/>
  <From DevIf="down"/>

  <State Name="closed" Default="closed_Default">
      <Interface Name="up">
      <Input Name="creq">closed_Creq</Input>
      </Interface>
      <Interface Name="down">
         <Input Name="open">closed_Conn</Input>
      </Interface>
   </State>
  <State Name="open" Default="open_Default">
      <Interface Name="up">
      <Input Name="dtreq">open_Dtreq</Input>
      </Interface>
      <Interface Name="down">
         <Input Name="read">open_Data</Input>
         <Input Name="close">open_Disconn</Input>
      </Interface>
   </State>
</SM>
```

B.1.2 Service Access Point Specifications

```
<?xml version="1.0"?>

<SAP Name="TlsSAP" CIncludeFiles="tlssap.h">

   <User>
      <Message Name="Creq">
         <Field>Frame priorities</Field>
      <Field>Frame cafile</Field>
      </Message>
      <Message Name="Dtreq">
         <Field>Frame data</Field>
      </Message>
   </User>

   <Provider>
    <Message Name="Cind">
      <Field>InetAddr addr</Field>
      <Field>Frame message</Field>
    </Message>
      <Message Name="Dtind">
      <Field>InetAddr source</Field>
         <Field>Frame data</Field>
      </Message>
   </Provider>

</SAP>
```

B.2 TlsTask Class Code

```
#ifndef TLSTASK_H
#define TLSTASK_H

#include <string>
#include <doors/ptb.h>
#include <doors/devif.h>
#include <doors/inetaddr.h>
#include <map>

#include "tlsconn.h"
#include "tlssap.h"
#include "tlssm.h"

class TlsTask : public PTask {
public:
  TlsTask (std::string n, Scheduler *s, IoHandler *io, Address *a1,
      TlsTaskSM* sm);
  ~TlsTask (void);

  TlsSAP :: Provider up;
  DevIf :: User down;

  bool closed_Default (Message* msg);
  bool closed_Creq (Message* msg);
  bool closed_Conn (Message* msg);
  bool open_Default (Message* msg);
  bool open_Dtreq (Message* msg);
  bool open_Data (Message* msg);
  bool open_Disconn (Message* msg);

  Frame session_data;
  int resume;

 protected:

  TlsConn tlsdevice;
  InetAddr rem_addr; //remote address
  Address *addr_; //same as rem_addr
};

#endif // TLSTASK_H
```

B.3 TlsConn Class Code

```
//tlsconn.h

#ifndef TLSCONN_H
#define TLSCONN_H

#ifdef HAVE_CONFIG_H
# include <config.h>
#endif

#include <doors/hsi.h>
#include <doors/soconnac.h>
#include <doors/buffer.h>
#include <doors/sockhelper.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <gnutls/gnutls.h>
#include <gnutls/extra.h>
#include <gnutls/x509.h>

#define MAX_BUF 4096

/** TlsConn can be seen as a client that tries to
  connect into SockEnt. When connection is established
  another TlsConn instance is also created into the
  peer entity and the actual data transfer is done between
  these two instances. The device informs different event to the
  controlling task by sending message with proper parameters.
  TlsConn is read-write device. */

class TlsConn : public SocketConnectionAC, public SockHelper {
  public:
  /** Initializes the base class and own variables. This constructor is
    used when instance is created without actually connecting to
    the peer entity. It can be done later with connect function call.
    @param io I/O handler of the system
    @param t the task that holds the device*/
  TlsConn (IoHandler *io, EventTask *t);

  /** Initializes the base class and own variables. Constructor is
    used when instance is created with given file decriptor. This
    usually happens in server side where descriptor is created by
```

```
  SocketEntity.
  @param ioh the i/o handler of the system.
  @param t the task that holds the device
  @param sd is descriptor for connection to be
  communicate with peer entity. */
//TlsConn (IoHandler *io, EventTask *t, Uint32 sd);

/** Initializes the base class and own variables. Constructor is
  used when instance is created with given peer entity address.
  This usually happens in client side where connection is established
  by the instance itself.
  @param ioh the i/o handler of the system,
  @param t the task that holds the device
  @param a an address of the peer entity. */
//TlsConn (IoHandler *io, EventTask *t, Address *a);

/**Delete connector and stream instances. */
~TlsConn (void);

/** Function is used for connecting to peer entity. The type of
  connection is checked from the type of address and proper
  connector and stream is created.
  The controlling task is informed both the success of failure
  of the connection establishement through the DevIf.
  @param a The address of peer entity.*/
bool connect (Address *a);

/** Function is used for connecting to peer entity with a
  previous TLS Session. The type of connection is checked
  from the type of address and proper connector and stream
  is created. The controlling task is informed both the success
  of failure of the connection establishement through the DevIf.
  @param a The address of peer entity.
  @param a The Session data of a previous TLS Session.*/
bool connect (Address *a, Frame sessiondata);

/** Function is used for connecting to peer entity. The type of
  connection is checked from the type of file descriptor and
  proper stream is created.
  @param sd The filedescriptor of peer entity */
bool connect (Uint32 sd);

/**Close device and send proper message to the
  controlling task. */
virtual bool close (void);

/** @return File descriptor of this connection */
virtual int getFd (void) const;
```

```
/** Function is runned when something is to be writen into the
  socket. <p>
  If it is possible to write all data into the socket it is done without
  any breaks. If it is not possible then writing is done only partially
  and it is continued when data can be sended.<p>
  If for some reason error happens then controlling task is informed
  with DevIf :: Close message. */
void callbackWrite (void);

/** Function is runned when something is to be read from the
  socket. <p>
  If there is all data availeble then it is read without any breaks
  and SocketData message is send to the controlling task. If there
  is not enough data availeble then partial reading is used and reading
  is continued when there is new data availeble.
  The data is sent in DevIF :: Read message to the task when either
  packetlenght is zero or wsize is set for -1 (StreamDevice :: immediate
     ).
  wsize is set to zero after data is sent and it must be set with
  devices readBytes function before every reading for the
  sake of synchronising. <p>
  If for some reason error happens then controlling task is informed
  with DevIf :: Close message. */
void callbackRead (void);

/** Function wraps the system call getpeername.
  @return the address of the peer entity. */
Address *getRemoteAddress (void);

/** Function wraps the gnutls_session_t.
  @return the actual session */
Frame getSession (void);
bool setSession (Frame sessiondata);

bool tls_init_params(char *priority_list, int debug, int insecure, int
    disable_ext);
bool tls_init_x509 (char *cafile, char *crlfile, char *keyfile, char *
    certfile);
bool tls_init_srp (char *srpuser, char *srppass);
bool tls_init_psk (char *pskuser, char *pskkey);

protected:
 int bindPort(int sock, int addrType);
int get_portnumber(int s);

void reset_global_var (void);
bool tls_global_init (void);
void tls_global_deinit (void);
bool tls_init_session (void);
```

```
bool tls_handshake (void);

static int cert_verify_callback (gnutls_session_t session);
static int srp_username_callback (gnutls_session_t session, char **
    username, char **password);
static int psk_callback (gnutls_session_t session, char **username,
    gnutls_datum_t * key);

int print_info (gnutls_session_t session);
void print_x509_certificate_info (gnutls_session_t session);
void print_cert_vrfy (gnutls_session_t session);
void check_alert (gnutls_session_t session, int ret);
static void tls_log_func (int level, const char *str);
static const char *bin2hex (const void *bin, size_t bin_size);


/* GNUTLS stuff here */
const char *hostname; //hostname of the peer (server)

//int verbose; //verbose mode != 0
int debug; //debug > 0; if debug == 0 then no_debug_verbose
int insecure; //if insecure != 0 -> protocol insecure -> skip
    verifications
int disable_extensions; //only extension until now -> GNUTLS_NAME_DNS

int resume; //if resumed session

bool flag_x509; //true if X.509 auth is used
bool flag_anon; //true if auth is anonimous (DH)
bool flag_psk; //true if PSK auth is used
bool flag_srp; //true if SRP auth is used

char *x509_cafile; //path of the CAs trusted file
char *x509_crlfile; //PKI -> path of the CRL file

//not in use yet
char *x509_keyfile; //path of the Private Key File
char *x509_certfile; //path of the Certificate File

gnutls_x509_crt_fmt_t x509ctype;
char *psk_username;
gnutls_datum_t psk_key;
char *srp_passwd;
char *srp_username;

char *priorities; //priorities list

gnutls_session_t session; //session data
```

```
  gnutls_certificate_credentials_t xcred; //X.509 credentials
  gnutls_anon_client_credentials_t anon_cred; //Anon DH credentials
  gnutls_srp_client_credentials_t srp_cred; //SRP credentials
  gnutls_psk_client_credentials_t psk_cred; //PSK credentials

  static TlsConn *myclass;

/*
  //->To Use Client Authentication with certificate
  #define MAX_CRT 6
  unsigned int x509_crt_size;
  gnutls_x509_crt_t x509_crt[MAX_CRT];
  gnutls_x509_privkey_t x509_key;
*/
/*
  //->To Use Client Authentication with certificate
  static int cert_callback (gnutls_session_t session, const gnutls_datum_t
      * req_ca_rdn, int nreqs,
    const gnutls_pk_algorithm_t * sign_algos, int sign_algos_length,
       gnutls_retr_st * st);
  static void load_keys (void);
  static gnutls_datum_t load_file (const char *file);
  static void unload_file (gnutls_datum_t data);
*/
/*
  static int protocol_priority[PRI_MAX];
  static int kx_priority[PRI_MAX];
  static int cipher_priority[PRI_MAX];
  static int comp_priority[PRI_MAX];
  static int mac_priority[PRI_MAX];
  static int cert_type_priority[PRI_MAX];
*/

};


#endif
```

Bibliography

[1] B. Silverajan and J. Harju, "Developing network software and communications protocols towards the internet of things," in *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, COMSWARE '09, (New York, NY, USA), pp. 9:1–9:8, ACM, 2009.

[2] Tampere University of Technology, "DOORS - Distributed Object OpeRationS." `http://www.cs.tut.fi/~doors/`, February 2002. [Online; accessed 11-April-2011].

[3] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," RFC 5389, Internet Engineering Task Force, Oct. 2008.

[4] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246, Internet Engineering Task Force, Aug. 2008.

[5] B. Silverajan, I. Karvinen, J. Mäkihonka, and J. Harju, "The design of a flexibly interworking distributed message-based framework," in *Proceedings of the EUNICE 2000 Summerschool*, pp. 39–46, September 2000.

[6] I. Karvinen, "Distributed Event Monitoring within DOORS Middleware," Master's thesis, Tampere University of Technology, Finland, 2002.

[7] A. S. Tanenbaum, *Computer Networks*. New Jersey, USA: Prentice Hall, 4th ed., 2003.

[8] P. Srisuresh and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)," RFC 3022, Internet Engineering Task Force, Jan. 2001.

[9] J. F. Kurose and K. W. Ross, *Computer Networking: A Top-Down Approach (5th Edition)*. Addison Wesley, 5 ed., March 2009.

[10] A. Wacker, G. Schiele, S. Holzapfel, and T. Weis, "A NAT traversal mechanism for Peer-to-Peer networks," in *Peer-to-Peer Computing , 2008. P2P '08. Eighth International Conference on*, pp. 81 –83, 2008.

[11] A. Muller, G. Carle, and A. Klenk, "Behavior and classification of NAT devices and implications for NAT traversal," *Network, IEEE*, vol. 22, no. 5, pp. 14 –19, 2008.

[12] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)," RFC 3489, Internet Engineering Task Force, Mar. 2003.

[13] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245, Internet Engineering Task Force, Apr. 2010.

[14] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," RFC 5766, Internet Engineering Task Force, Apr. 2010.

[15] C. Jennings, R. Mahy, and F. Audet, "Managing Client-Initiated Connections in the Session Initiation Protocol (SIP)," RFC 5626, Internet Engineering Task Force, Oct. 2009.

[16] D. MacDonald and B. Lowekamp, "NAT Behavior Discovery Using Session Traversal Utilities for NAT (STUN)," RFC 5780, Internet Engineering Task Force, May 2010.

[17] J. Postel, "Internet Protocol," RFC 0791, Internet Engineering Task Force, Sept. 1981.

[18] F. Yergeau, "UTF-8, a transformation format of ISO 10646," RFC 3629, Internet Engineering Task Force, Nov. 2003.

[19] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261, Internet

Engineering Task Force, June 2002.

[20] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," RFC 2616, Internet Engineering Task Force, June 1999.

[21] W. Diffie and S. Landau, *Privacy on the Line: The Politics of Wiretapping and Encryption, Updated and Expanded Edition.* The MIT Press, 2007.

[22] R. Oppliger, *SSL and TLS: Theory and Practice.* Norwood, MA, USA: Artech House, Inc., 2009.

[23] R. Braden, "Requirements for Internet Hosts - Communication Layers," RFC 1122, Internet Engineering Task Force, Oct. 1989.

[24] "IEEE Standard for Local and Metropolitan Area Networks - Media Access Control (MAC) Security," *IEEE Std 802.1AE-2006*, pp. 1–142, 2006.

[25] S. Frankel, *Demystifying the Ipsec Puzzle.* Norwood, MA, USA: Artech House, Inc., 2001.

[26] R. Oppliger, *Secure messaging: with PGP and S/MIME.* Norwood, MA, USA: Artech House, Inc., 2001.

[27] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Trans. Comput. Syst.*, vol. 2, pp. 277–288, November 1984.

[28] E. Rescorla and A. Schiffman, "The Secure HyperText Transfer Protocol," RFC 2660, Internet Engineering Task Force, Aug. 1999.

[29] W. Stallings, *Network Security Essentials: Applications and Standards.* Upper Saddle River, NJ, USA: Prentice Hall Press, 4th ed., 2010.

[30] ITU, *Security architecture for Open Systems Interconnection for CCITT applications (ITU-T Recommendation X.800).* International Telecommunications Union, March 1991.

[31] R. Shirey, "Internet Security Glossary," RFC 2828, Internet Engineering Task Force, May 2000.

[32] U.S. Department of Commerce, National Institute of Standards and Technology, Computer Systems Laboratory, "DES Modes of Operation," FIPS PUB 81, December 1980.

[33] National Institute of Standards and Technology, *FIPS PUB 46-3: Data Encryption Standard (DES)*. National Institute for Standards and Technology, October 1999. supersedes FIPS 46-2.

[34] Electronic Frontier Foundation (EFF), "DES Cracker Project." `http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html`, 1998. [Online; accessed 11-April-2011].

[35] Electronic Frontier Foundation (EFF), "DES Challenge III Broken in Record 22 Hours." `http://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19990119_deschallenge3.html`, January 1999. [Online; accessed 11-April-2011].

[36] National Institute of Standards and Technology, *FIPS PUB 197: Specification for the Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, November 2001. supersedes FIPS 46-2.

[37] S. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the Key Scheduling Algorithm of RC4," *Lecture Notes in Computer Science*, vol. 2259, 2001.

[38] A. Klein, "Attacks on the RC4 stream cipher," February 2006.

[39] E. Tews, R.-P. Weinmann, and A. Pyshkin, "Breaking 104 bit wep in less than 60 seconds." Cryptology ePrint Archive, Report 2007/120, 2007. `http://eprint.iacr.org/`.

[40] J. Kelsey, B. Schneier, and D. Wagner, "Related-key cryptanalysis of 3-way, biham-des, cast, des-x, newdes, rc2, and tea," in *Proceedings of the First International Conference on Information and Communication Security*, (London, UK), pp. 233–246, Springer-Verlag, 1997.

[41] O. D. Eli Biham and N. Keller, "A new attack on 6-round IDEA," 2007.

[42] National Institute of Standards and Technology, *Skipjack and KEA Algorithm Specifications*. National Institute for Standards and Technology, May 1998.

[43] E. Biham, A. Biryukov, and A. Shamir, "Cryptanalysis of skipjack reduced to 31 rounds using impossible differentials," in *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'99, (Berlin, Heidelberg), pp. 12–23, Springer-Verlag, 1999.

[44] M. Matsui, J. Nakajima, and S. Moriai, "A Description of the Camellia Encryption Algorithm," RFC 3713, Internet Engineering Task Force, Apr. 2004.

[45] A. Kato, M. Kanda, and S. Kanno, "Camellia Cipher Suites for TLS," RFC 5932, Internet Engineering Task Force, June 2010.

[46] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[47] G. C. Kessler, "An Overview of Cryptography." `http://www.garykessler.net/library/crypto.html`, March 2011. [Online; accessed 11-April-2011].

[48] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. IT-22, pp. 644–654, Nov. 1976.

[49] D. McGrew, K. Igoe, and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms," RFC 6090, Internet Engineering Task Force, Feb. 2011.

[50] T. Wu, "The SRP Authentication and Key Exchange System," RFC 2945, Internet Engineering Task Force, Sept. 2000.

[51] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication," RFC 5054, Internet Engineering Task Force, Nov. 2007.

[52] P. Eronen and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," RFC 4279, Internet Engineering Task Force, Dec. 2005.

[53] U. Blumenthal and P. Goel, "Pre-Shared Key (PSK) Ciphersuites with NULL Encryption for Transport Layer Security (TLS)," RFC 4785, Internet Engineering Task Force, Jan. 2007.

[54] Cisco Systems, "Introduction to Secure Socket Layer." `http://www.cisco.com/warp/public/cc/so/neso/cxne/cxdimng/wpsot_wp.pdf`, 2002. [Online; accessed 11-April-2011].

[55] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," RFC 2246, Internet Engineering Task Force, Jan. 1999.

[56] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1," RFC 4346, Internet Engineering Task Force, Apr. 2006.

[57] Free Software Foundation, Inc, "The GNU Transport Layer Security Library (GnuTLS)." `http://www.gnu.org/software/gnutls/`, 2011. [Online; accessed 11-April-2011].

[58] Individual Mozilla contributors, "Network Security Services (NSS)." `http://www.mozilla.org/projects/security/pki/nss`, 2011. [Online; accessed 11-April-2011].

[59] OpenSSL Core and Development Team, "OpenSSL Project." `http://openssl.org`, 2009. [Online; accessed 11-April-2011].

[60] Free Software Foundation, Inc, "GnuTLS - Comparison of different free TLS implementations." `http://www.gnu.org/software/gnutls/comparison.html`, 2011. [Online; accessed 18-November-2010].

[61] Nikos Mavrogiannopoulos and Simon Josefsson, "GnuTLS - Priority Strings." `http://www.gnu.org/software/gnutls/manual/html_node/Priority-Strings.html#Priority-Strings`, April 2011. [Online; accessed 11-April-2011].

[62] reSIProcate Development Team, "reSIProcate projects." `http://www.resiprocate.org`, 2007. [Online; accessed 23-May-2010].