Analysis and Processing of Cryptographic Protocols

Submitted in partial fulfilment
of the requirements of the degree of

BSc. Honours

of Rhodes University

Bradley Cowie

Grahamstown, South Africa
November 2009

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2009)

C.2.2 [Network Protocols]: Protocol architecture

D.4.6 [Security and Protection]: Cryptographic controls

E.3 [Data Encryption]: Public key cryptosystems, Standards

General Terms : Cryptography, Protocols

Acknowledgements

Without the expert assistance and guidance provided by Barry Irwin and Richard Barnett, the task of completing this thesis would have been insurmountable. I would like to thank Barry Irwin for supervising my research project and keeping me focuse towards the completion of this thesis. His vast knowledge in the field of Information Security has been a valuable asset in the production of this thesis. Thanks must be given to Richard Barnett for providing morale support and dedicated proof reading of research papers.

I would like to thank my family for their continued support both emotionally and financially during the course of my university career. I would like to thank my mother and my father for their continued faith in my abilities.

Finally, thanks is given for the financial support provided by Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole, Mars Technologies, Bright Ideas, Projects 39 and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

Abstract

The field of Information Security and the sub-field of Cryptographic Protocols are both vast and continually evolving fields. The use of cryptographic protocols as a means to provide security to web servers and services at the transport layer, by providing both encryption and authentication to data transfer, has become increasingly popular. However, it is noted that it is rather difficult to perform legitimate analysis, intrusion detection and debugging on data that has passed through a cryptographic protocol as it is encrypted. The aim of this thesis is to design a framework, named Project Bellerophon, that is capable of decrypting traffic that has been encrypted by an arbitrary cryptographic protocol. Once the plain-text has been retrieved further analysis may take place.

To aid in this an in depth investigation of the TLS protocol was undertaken. This produced a detailed document considering the message structures and the related fields contained within these messages which are involved in the TLS handshake process. Detailed examples explaining the processes that are involved in obtaining and generating the various cryptographic components were explored.

A systems design was proposed, considering the role of each of the components required in order to produce an accurate decryption of traffic encrypted by a cryptographic protocol.

Investigations into the accuracy and the efficiency of Project Bellerophon to decrypt specific test data were conducted. An investigation into the accuracy of the decryption produced by the application framework showed that Project Bellerophon produced an accurate decryption for a small test file.

An investigation into the efficiency of Project Bellerophon was conducted, comparing execution times for decryption against Tshark, an industry standard protocol analyser. This investigation showed a linear relationship between execution time and file size for both Tshark and Project Bellerophon. However it was noted that in general Project Bellerophon had longer execution times. It was argued that this difference in execution time is not sufficiently significant to conclude that the framework is infeasible.

Contents

1 Introduction **1**

1.1 Problem Statement 2

1.2 Research Goals 3

1.3 Research Scope 3

1.4 Motivation for research 4

1.4.1 Prevalence of Web Based Transactions 4

1.4.2 Software Development Habits 4

1.5 Document Structure and Outline 5

2 Literature Survey **7**

2.1 Symmetric Cryptography 8

2.1.1 The Data Encryption Standard (DES) 8

2.1.2 The Advanced Encryption Standard (AES) 10

2.2 Public Key Cryptography 14

2.2.1 Mathematics primer 14

2.2.2 Diffie-Hellman key exchange 16

2.2.3 RSA 17

2.3 Cryptographic hash functions 18
2.3.1 MD5 18
2.4 TLS 20
2.4.1 Hybrid Cryptosystems 21
2.4.2 The TLS Handshake 21
2.4.3 Practices in TLS and SSL 22
2.5 SSH 23
2.5.1 Components of SSH 23
2.5.2 Architecture of SSH 25
2.6 Related tools 26
2.6.1 SSLDump 27
2.6.2 Wireshark and Tshark 27
2.7 Related Research 27
2.7.1 Detection of encrypted applications 27
2.7.2 Running Mode Analysis 28
2.8 Conclusion 28

3 TLS Technical Specification 30

3.1 TLS Specification 30
3.1.1 An overview of the packet structure 31
3.1.2 Ethernet Header 32
3.1.3 IP Header 34
3.1.4 TCP Header 35

3.1.5 SSL Record 35
3.1.6 SSL Handshake and SSL Message Types 38
3.1.7 Client Hello 39
3.1.8 Server Hello 42
3.1.9 Client Key Exchange 44
3.1.10 Change Cipher Specification 46
3.2 TLS PRF 48
3.2.1 The PRF algorithm 48
3.2.2 Master Secret generation 50
3.2.3 Key expansion 51
3.3 SSL Specification 52
3.4 Summary 53

4 Design and Implementation 54

4.1 Application Design 54
4.1.1 Logical system design 55
4.1.2 Decryption module 56
4.1.3 Packet Capture module 57
4.1.4 Traffic Analyser module 58
4.1.5 Main Application 59
4.2 Testbed Design and Construction 60
4.2.1 Configuring the TLS server 61
4.2.2 Configuring the TLS Client 64

4.3 Summary 64

5 Results and Discussion 66

5.1 Test platform 66

5.2 Verifying the accuracy of the decryption 67

5.2.1 Experimental Outline 67

5.2.2 Cryptographic parameters aquired from Server Hello and Client Hello 70

5.2.3 Premaster Secret and Client Key exchange 71

5.2.4 Master Key generation 72

5.2.5 Key expansion 74

5.2.6 Recovered Application Data 76

5.3 Testing of efficiency 78

5.3.1 Experimental Outline 78

5.3.2 Establishing a baseline 78

5.3.3 Tshark Timing Results 80

5.3.4 Project Bellerophon Timing Results 82

5.4 Comparing Project Bellerophon and Tshark in terms of efficiency 84

5.5 Summary 85

6 Conclusion 86

6.1 Summary 86

6.2 Revisiting the research goals 87

6.3 Possible Extensions 87

Bibliography **89**

A Test scripts listing **93**

B Protocol Structs **97**

C PCAP Capture Sample Code **99**

D CD Contents **101**

List of Figures

2.1 Block diagram of initialization values for registers during a MD5 hash 19

3.1 Logical packet structure considered 31

3.2 Block diagram of packet structure considered with sample hexadecimal values 32

3.3 Logical structure of an Ethernet packet 33

3.4 Example Ethernet Header with sample hexadecimal values . . 33

3.5 Logical structure of an IP Header 34

3.6 Example IP Header with sample hexadecimal values 34

3.7 Logical structure of a SSL Record 36

3.8 Example SSL Record with sample hexadecimal values 36

3.9 Example SSL Record with sample hexadecimal values 38

3.10 Logical structure of a Client Hello message 40

3.11 Client Hello Message example with sample hexadecimal values 40

3.12 Logical structure of a Client Hello Message 42

3.13 Server Hello Message example with sample hexadecimal values 43

3.14 Logical Structure of a Client Key Exchange Message 44

3.15 Client Key Exchange Message example with sample hexadecimal values . . . 45

3.16 Logical structure of a Change Cipher Specification Message . . 46

3.17 Client Key Exchange Message example with sample hexadecimal values . . . 46

3.18 Logical structure of an Application Data Message 47

3.19 Application Data Message example with sample hexadecimal values . . . 47

4.1 Flow diagram showing the proposed systems design detailing the system structure and logical flow . . . 55

4.2 Class diagram of Project Bellerophon . . . 60

4.3 Cipher Suite Configuration for Opera 10 . . . 64

5.1 Client Hello message obtained during Project Bellerophon execution . . . 70

5.2 Server Hello message obtained during Project Bellerophon execution . . . 71

5.3 Client Key Exchange message obtained during Project Bellerophon execution . . . 72

5.4 Decrypted Premaster Secret obtained after decrypting the encrypted Premaster Secret . . . 72

5.5 Block diagram of HMAC Keys used in the TLS PRF 73

5.6 Block diagram of HMAC hashes created during the TLS PRF execution . . . 74

5.7 Block diagram of the generated Master Secret from the TLS PRF . . . 74

5.8 Block Diagram of the output of the PRF Key Expansion . . . 75

5.9 Block diagram of recovered client cryptographic parameters . . 76

5.10 Block Diagram of recovered server cryptographic parameters . 76

5.11 Block diagram of Encrypted Application Data 77

5.12 Block diagram of Decrypted Application Data 77

5.13 Graph of File Size(bytes) vs.Time to decrypt encrypted files(ms) using OpenSSL 79

5.14 Graph of encrypted traffic size (bytes) vs. time taken for decrypting encrypted traffic using Tshark 81

5.15 Graph of encrypted traffic size (bytes) vs. time taken for decrypting encrypted traffic using Project Bellerophon 83

List of Tables

3.1 Table providing a reduced list of Ether Types 33

3.2 Table of SSL Content Types 37

3.3 Table providing a reduced list of SSL and TLS Versions 37

3.4 Table listing TLS Handshake types 39

3.5 5.6Table of TLS Cipher Suites 42

5.1 Table containing the hardware and software configuration used for testing 66

5.2 Table specifying the test file parameters 68

5.3 Table of File Size vs. Execution time to decrypt a file encrypted using AES-256-CBC 79

5.4 Table showing time taken to decrypt encypted traffic(ms) vs.traffic size for Tshark(bytes) 81

5.5 Table showing time taken to decrypt encrypted traffic(ms) vs. traffic time for Project Bellerophon(bytes) 82

5.6 Results comparison for Tshark, Project Bellerophon and OpenSSL 84

List of Algorithms

2.1 The AES Encryption Algorithm 13
2.2 Process Message phase of MD5 20
3.1 TLS PRF Pseudo-code 50
3.2 SSL PRF Pseudo-code 53
4.1 Decryption module pseudo-code 56
4.2 Packet Capture module pseudo-code 57
4.3 Traffic Analyser module pseudo-code 59
5.1 Output produced by Project Bellerophon when supplied with captured traffic in PCAP format 69

Glossary

PCAP

PCAP is an API that provides features for packet capture. PCAP files contain packets that have been captured using a traffic analysis tool such as Wireshark.

MITM

A man-in-the-middle attack is an attack where a third party intercepts communications of another party and then may alter the information passed onto the original recipient.

IDS

An intrusion detection system (IDS) inspects traffic passing through a network, attempting to identify potential security threats using a signature database.

HTTPS

Hypertext Transfer Protocol Secure (HTTPS) combines HTTP together with the SSL/TLS protocols in order to provide encryption and authentication to web-based communications.

RSA

A popular public-key encryption algorithm.

MAC (Message Authentication Code)

A message authentication code is used to provide authentication to messages sent by producing a cryptographic hash of the message.

OSI

Open System Interconnection is an ISO standard which defines a networking framework for implementing protocols.

HMAC

HMAC is a type of MAC that makes use of key in addition to a cryptographic hash.

Cipher Suites

A structure within TLS and SSL that defines the symmetric key used FOR encryption, the hash algorithm used for MAC and the technique used for key exchange. For example, TLS_RSA_AES_256_CBC_SHA1 defines RSA for key exchange, 256-bit AES CBC for symmetric encryption and SHA1 to provide the MAC's.

Chapter 1

Introduction

The ability to convey information between two parties in a secure fashion is a desirable property of any form of communication [19]. Cryptography, part art and part science, is the study of the ways in which information can be disguised by encoding a message in some specific way and then after the message has been transmitted, perform an inverse process to recover the original information [17]. It is important at this point to differentiate cryptography from its logical cousin, stenography. Stenography is the study of the ways in which information that is to be communicated can be hidden. Where as cryptography is concerned with keeping the message in plain sight [42], but instead performing transformations on the data so that it cannot be read without performing the reverse transformations [35].

Historically, many countries and cultures have taken an active interest in cryptography including the Romans, ancient Egyptians and the French. This interest has been necessitated by the need for military leaders to be able to communicate safely and securely [38]. Some classical ciphers such as Caesars Cipher, Simple Substitution Cipher and the Vingère Cipher were sufficiently cryptic in their respective times but are now considered trivial in complexity when compared to the complex ciphers used today [22, 35]. Many famous academics and scholars have been involved in the cryptographic arts including Edgar Alan Poe [25], Giambattista della Porta [16] and Blaise de Vigenère [22].

Modern cryptography is concerned with providing secure electronic communications. Digital cryptography has a wide number of uses within in the scope

of computing including file encryption, providing secure connections to websites for transferal of sensitive information such as credit card details, checking the validity of data through cryptographic hashing and non-repudiation for e-mail and other messaging technologies.

Cryptographic Protocols establish a sequence of steps required to perform a negotiation allowing parties to come to an agreement over certain cryptographic parameters, after which a secure communication channel can be created. Historically notable cryptographic include the Diffie-Hellman key exchange protocol [31] and the Needham-Schroeder protocol [26, 40]. In more recent times TLS, SSH and IPSec are more relevant protocols.

This introductory chapter considers the thesis problem statement, the research goals, the research scope, the motivation for research and the document structure.

1.1 Problem Statement

Cryptography and thus by extension cryptographic protocols are metaphorically double edged blades. Through cryptography it is possible to obfuscate the data being transmitted and thus preventing third parties from being able to listen in on communication this has the negative effect that in the process systems which wish to legitimately gain access to this encrypted information to perform legitimate analysis or scanning cannot make sense of the obfuscated data.Considering a firewall as an example, if one of the users behind a firewall makes a connection to an external server using a cryptographic protocol, often called an "encrypted tunnel", the firewall is incapable of performing analysis on the traffic passing through the encrypted tunnel as the firewall is missing some of the components required to build the cryptographic parameters. This is extremely problematic as it hinders firewalls and IDS (Intrusion Detection systems) from being able to perform their specified tasks effectively [2].

The core problem examined in this thesis is the development of a framework design, which was named Project Bellerophon after the protagonist of Homer's Illiad, for decrypting data that has been encrypted by an arbitrary cryptographic protocol. Project Bellerophon must be able to reconstruct the cryptographic parameters used during the handshake phase of the protocol for both live traffic and prerecorded network traffic. Once these parameters

have been constructed and the cryptographic cipher suite has been obtained the encrypted application traffic will be decrypted.

Having considered the problem statement the next section considers the research goals of this thesis.

1.2 Research Goals

The primary research goal was to provide an entry point for the generic decryption of network traffic by developing a system design that is appropriate for the decryption of traffic encrypted by an arbitrary protocol. Having developed said design, it was necessary to provide a sample implementation for a well known cryptographic protocol to show that said design is feasible.

Secondary goals include testing of the developed implementation in terms of both the accuracy of decryptions provided and comparing the efficiency of the implementation by contrasting it against another tool which provides the same basic functionality.

The tertiary research goal was to provide documentation of the TLS protocol that is more accessible and understandable to programmers by showing worked examples of the TLS Handshake and the key generation that occurs within in the TLS Handshake.

The bounds and assumptions made to contact such research is considered in the next section.

1.3 Research Scope

This thesis shall only consider an implementation for TLS. However using the concepts developed in this thesis is should be possible to extend to other cryptographic protocols such as SSH and IPSec. The actual decryption will assume legitimate and lawful access to the data that is to be decrypted and as a result it is expected that access to the complete negotiation of phase under taken by the parties will be provided. Further access to all cryptographic parameters is assumed.

Having discussed the research scope, the need for such research is considered.
.

1.4 Motivation for research

Due to the upsurge in the demand for secure transactions over the Internet there is a need for constant evaluation and research in the field of Information Security particularly in the fields of cryptography and cryptanalysis. To put further emphasis some of the potential scenarios where this type of research may be useful are now outlined.

1.4.1 Prevalence of Web Based Transactions

HTTPS has become prevalent as a means to communicate with a web server securely. However if an attacker were to use HTTPS as a means to perform an attack. It becomes difficult to detect such an attack due to the encrypted nature of the traffic. It would be useful if a system existed to decrypt this traffic and then perform analysis. This is highlighted by work done by Marklinspike [21] in developing a tool, SSLStripper [21], that removes the secure components of a SSL connection allowing for a new form of MITM (man in the middle) attack where the user believes that his connection is secured but in reality messages are passed through HTTP and are intercepted by a third-party.

1.4.2 Software Development Habits

Wang *et al.* [41] comment that in the long term software development cannot afford to consider implementing security only after the application has been developed or late in the development cycle as irreparable security compromises may already exist and that attempts to correct them would require significant resources. It can further be noted that security is one of the core metrics in McCall's Software Quality Checklist [1].

However, software development is notorious for being over budget and far exceeding its expected completion date. As a result it is often found that security is left until late in the development cycle and sometimes even after

the application has been built [41]. Often this causes poorly implemented security and this only serves to degrade the quality of the system built as it provides the user with a false sense of security. Further an insecure application that passes and receives sensitive information is equally as unusable as an application that fails to meet its specifications in terms of correctness [41].

It could be argued that the reason why security is not part of many development cycles in the earlier stages is due to the difficulty and tedium of checking the correctness of security [36, 5]. To put this in context considering the period between January 2004 and December 2008 there have been 26,139 reported security vulnerabilities [27]. It would be useful if there existed a framework that decrypted data and then provided some analysis on issues pertaining to the implemented security.

Finally the document structure is now outlined in this final section of Chapter 1.

1.5 Document Structure and Outline

The remainder of this document is organized as follows.

Chapter 2 discusses content related to the research topic. This will include concepts in cryptography such as symmetric cryptography versus asymmetric cryptography. A discussion into related research such as the detection of encrypted applications and tools relevant to cryptographic protocol decryption such as Wireshark and Tshark.

Chapter 3 provides a technical discussion of the TLS/SSL protocols.This section has been written and illustrated in a way to make it be more accessible to programmers who desire a deeper understanding of these protocols which is difficult to infer from documents such as the TLS/SSL RFC.

Chapter 4 discusses how the system was implemented, considering the data structures and algorithms required.

Chapter 5 is a listing and discussion of results obtained.

Chapter 6 concludes the research that has been done, providing a summary of the research with possible extensions for the future

Appendix A consists of coding listings for test scripts used.

Appendix B consists of coding listings for C++ protocol structs used.

Appendix C consists of coding listings for sample capture module.

Appendix D lists the contents of the accompanying CD.

Chapter 2

Literature Survey

Cryptography is the discipline, art and science of ensuring that messages are secure from possible such as eavesdropping, impersonation or data corruption. Cryptography provides security through a sequence of mathematical transformations that can be shown to be mathematically secure provided some optimum conditions [35]. It is important to be cognizant of the fact that cryptography on its own is insufficient to ensure a high level of security within an organization. That is to say that cryptography is not the silver bullet to solve all information security issues and should be used in conjunction with good security practices [37]. Cryptography, like the Information Security field itself, is an incredibly broad field involving many existing disciplines such as abstract algebra to provide mathematical proofs for the guaranteed correctness of an algorithm, statistics for the analysis of cryptographic algorithms and quantum physics for quantum based random number generation for quantum cryptography [37].

In this chapter schemes for cryptography are discussed including symmetric cryptography, public key cryptography and hybrid cryptosystems. An outline of cryptographic hash functions is provided as well. The architecture of TLS is considered and a discussion of the SSH protocol is provided. Finally some consideration is given to related work and related tools with regards to protocol analysis and decrypting encrypted protocols

The first cryptographic scheme to be considered is symmetric cryptography.

2.1 Symmetric Cryptography

Symmetric cryptography, also known as secret key cryptography, has been in use since ancient times [22] and has a wide variety of different implementations. These range from simple substitution ciphers such as Caesars Cipher to complex and supposedly mathematically unbreakable algorithms such as AES [20]. Symmetric key encryption makes use of a single key that must be kept secret. This key is used for both the encryption and decryption of messages to be sent or stored. In this section the DES and AES algorithms are discussed considering some of the history of these algorithms and the phases and processes involved in said algorithms.

The DES algorithm is considered in the next subsection.

2.1.1 The Data Encryption Standard (DES)

DES is considered within this literature survey as it is one of the classical encryption algorithms. Considering the algorithms and structures involved in DES allows for a deeper understanding of cryptographic algorithms.

The Data Encryption Standard was developed by IBM and was selected in 1976 as an official Federal Information Processing Standard for the United States [20]. The DES algorithm uses a 64-bit key of which 8-bits are used for parity and the remaining 56-bits are used to encrypt the plain-text.

The required computations for brute forcing a DES key would be 2^{56} operations, given a 64-bit plain-text and 64-bit DES key. While the DES algorithm itself is considered to be resistant to cryptanalysis the actual keys used for encryption are considered to be fairly weak [9, 35].

The DES algorithm consists of three phases, a Cryptographic Hash function and a Key Scheduler which are detailed in the remainder of this subsection.

Phase 1

The first 64-bits of plain-text, denoted by x , runs through an Initial Permutation function (IP) returning 64-bits of output denoted by x_0. Mathematically this can be represented as $x_0 = IP(x)$. The output is separated into

two sections of equal length. This separation is represented as L_0R_0, where L_0 represents the first 32-bits and R_0 represents the remaining 32-bits. An inverse function of IP named IIP is also defined [9].

Phase 2

The output obtained in Phase 1 then undergoes sixteen repetitions of a computation that is key dependent using a keyed cryptographic hash function. This function is denoted by f. This function makes use of a key scheduling function denoted as KS. A key scheduler calculates all the sub-keys for each round or iteration. The output of each iteration or round can be represented as $x_i = L_iR_i$ with $1 \leq i \leq 16$ with $L_i = R_i - 1$ and $R_i = L_i \otimes f(R_i - 1, K_i)$. The $K_i's$ are 48-bit blocks that can be derived from the original 56-bit string using KS [9].

Phase 3

In the final phase, IP is applied to x_{16} to give another 64-bit cipher block which shall be called C. It follows that $C = IIP(x_{16}) = IIP(R_{16}L_{16})$. It is noted that the inverse property applies, that is $IIP(IP(x)) = x$ [9].

The Cryptographic Hash function, f

This function expands the $R_i's$ from their 32-bit block to a 48-bit block through an expansion permutation. Essentially this function increases the bit length by reusing some of the bits in the R_i's, and also re-ordering them making use of a look-up table. This output then undergoes the XOR operation with K_i [9]. This result is then broken up in to eight blocks of six-bits each. These six-bit blocks are then passed through an S-box giving an output of four-bits. The S-box takes the first bit and the last bit of the input forming a two-bit binary value. The decimal value of this two-bit number is used to select a row [9]. The remaining inner four-bits are used to select a column number. These row and column values are used to index a value from the S-box . The four-bit output of each of these eight boxes is then concatenated to yield a 32-bit output which is finally given to the permutation function P which gives a result of 32-bits [9].

Key Scheduling

The key scheduling function, KS, is used to derive the 48-bit $K_i's$ from the original 56-bit key. It is noted that while DES keys are 64-bit only 56-bits are actually used to seed the random functions as eight-bits are used for error checking. Every eighth bit is used for parity. The key scheduling functions consist of two permutation functions, PC_1 and PC_2, where PC stands for Permutation Choice. To select the $K_i's$ the following algorithm is applied.

Given a 64-bit key K the eight-bits used for parity are discarded. The PC_1 function is then applied to the remainder of the key. This can be represented as $PC_1(K) = C_0D_0$ where C_0 represents the first 28-bits and D_0 represents the remainder. PC itself has two components, with the first half determining C_i and the second half determining D_i.

Tocalculate the individual C_iD_i the L_iS_i function is applied. The S_i function represents the number of left cylindrical shifts by which C_i or D_i are to be shifted. That is $C_i = L_iS_i(C_i - 1)$ and $D_i = L_iS_i(D_i - 1)$. The L_i function is yet another look up table function. The resultant bits of C_i and D_i are then concatenated together and PC_2 is applied to the output of the concatenation, that is $K_i = PC_2(Ci; Di)$. It is noted that $PC_2$is a similar function to PC_1.For decryption the same key is used, but the order of functions applied is reversed.

The next section considers a more modern symmetric key algorithm known as AES.

2.1.2 The Advanced Encryption Standard (AES)

AES is considered to be the state of the art in terms of modern symmetric cryptographic algorithms. It is for this reason that is AES is now discussed.

The AES accepted candidate, Rijndael, was designed by John Daemen and Vincent Rijmen from Belgium and was published in 1998 [9]. AES is an iterated block cipher that allows for a variable key length and a choice from a number of different block sizes. AES supports block sizes of 128-bits, 192-bits and 256-bits [9]. AES is byte orientated compared to the bit orientated nature of DES.

The number of rounds or iterations applied is dependent on the block size

and the key used. For example if the block size is 128-bits and letting m be the size of key and r the number of rounds is given by $r = k/32 + 6$. At the start of the algorithm a 128-bit block of plain-text is used as the initial state [9].

This initial state will be passed through a number of key-dependent transformations finally returning a 128-bit block of plain-text. A state is treated as a 4x4 matrix, where $A_{i;j}$ will represent a single byte with $0 \leq i, j \leq 3$, with i indexing the rows and j indexing the columns. For example $A_{0,0}$ is the first byte and $A_{1,0}$ is the fifth byte. AES makes use of four basics operators to allow for transformation from one state to another. The set of operators used by Rijndael include the following four operators [9] :

Byte Substitution

This is a non-linear permutation that operates on each byte in the current state independently, allowing for parallelism. In this phase eight-bytes of the sixteen-bytes to be operated on are multiplied against a 8x8 matrix, this is matrix multiplication of an 8x8 matrix by a 8x1 column vector resulting in a 8x1 column vector. This can be efficiently implemented by making use of a 256-bit lookup table or an S-box [9].

Row shift

This is a cyclic shift of the bytes in a state. This shift can be represented mathematically by $B_{i;j} = A_{i,(j+1)}\ mod\ 4$. The first row will undergo no changes, however the second row will shift one column, the third row shifts two columns and the third row will shift three columns [9].

Mix column

Each of the columns A_i undergoes a linear transformation. A transformation is applied to a column at a time and is equivalent to multiplying the columns contents by a 4x4 matrix [9].

Round Key Addition

For every round performed a round key, RK, is generated from the cipher key via the key scheduling function. The round key is the same length as the encryption block and is represented as a 4x4 matrix, similar to how the plain-text is represented. The XOR operation is then applied to the round key and the current state [9].

The AES Encryption Algorithm

As already mentioned the Rijndael encryption algorithm takes as input a state and produces a state that contains the cipher-text. Algorithm 2.1 describes the AES Encryption Algorithm.

Algorithm 2.1 The AES Encryption Algorithm

```
AESEncrypt(state, key[0, ..., 4K-1])
{
  /*The first k words of W contain 4k bytes of
  the key array*/
  InverseKeyExpansion(key[0, ..., K-1],
   W[0, ...,N(R+1) - 1])

  //Add the first round key to the state
  AddRoundKey(state, W[0,...,3N])

  //Calculate and add the other keys
  for( int i = r-2 ; i> 0 ; i)
  {
    InverseByteSubstitution(state)
    InverseShiftRow(state)
    InverseMixColumn(state)
    AddRoundKey(state,W[i,... 3+i])
  }

  //Do the Final Round
  ByteSubstitution(state)
  ShiftRow(state)
  ByteSubstitution(state)
  AddRoundKey(state,
 W[N(R+1) - 4,...,N(R+1) - 1])
}
```

The AES Decryption Algorithm

The decryption of encrypted data is achieved by applying the operations used in encryption but in reverse order.

Having considered two examples of symmetric encryption, public key cryptography is now discussed in the next section.

2.2 Public Key Cryptography

One of the difficulties involved in symmetric key encryption is key distribution. The reason for this is that for each pair of parties who wish to communicate a new key is required to encrypt and decrypt their communication. This creates a logistical nightmare when trying to manage all the keys that a party may need in order to communicate with other parties.

Public key encryption was designed to solve this problem by having a key-pair for each party. A public key that is given out to those who wish to send messages to the party and a private key used by the party to decrypt the messages that are encrypted with its public key [24].

Given the public key it should not be computationally feasible to compute the private key. Thus the private key and public key should be related in such a way that it should not be easy to derive the private key from the public key. This usually entails some sort of unsolved mathematical problem such as the factorization of large numbers or the discrete logarithm problem [37]. This chapter provides a mathematics primer required in order to understand public key cryptographic algorithms and then goes on to discuss the Diffie-Hellman key exchange and RSA.

The next section considers some of the mathematics involved in public key cryptography.

2.2.1 Mathematics primer

In order to understand some of the concepts used in public key cryptography a basic understanding of some mathematical concepts involved is required. Especially abstract algebraic concepts such as groups, co-primes,relatively prime numbers, congruency and prime roots.

Groups

A group is a set of mathematical elements together with a binary operation, that is an operation that takes two inputs and produces a single output, that together satisfy the following four properties. Let G be a group and $a, b, c \in G$ with $*$ the binary operator of G.

1. Closure : If there are two elements $a, b \in G$ then the product $a * b \in G$.
2. Associativity: The defined binary operation, $*$, must be associative. That is for $\forall a, b, c \in G$ then it follows that $a * (b * c) = (a * b) * c$
3. Identity: There is an identity element $a \in G$ such that for $\forall b \in G$, $a * b = b$ and $b * a = b$.
4. Inverse: For each element there must exist an inverse. Let $b \in G$ then there must $\exists d \in G$ such that $b * d = a$ and $d * b = a$ where a is the identity of G.

An example of group could be Z_{10}, that is the set of integers *modulo* 10 under the action of integer multiplication [34].

Greatest Common Divisor (GCD)

The greatest common divisor of two positive integers, say $a, b \in G$, is the largest positive integer that divides both integers a and b. The greatest common divisor of a and b is commonly represented as $GCD(a, b)$ [44].

Relatively prime

Two integers are said to be relatively prime to each other if the largest and thus only positive divisor of the two is the integer one. That is, if $a, b \in G$ then it follows that a and b are relatively prime to each other if and only if $GCD(a, b) = 1$ [45]. It is noted that the terminology relatively prime and co-prime are equivalent.

Congruency in Algebra

Two integers are said to be congruent if the two integer are equivalent *modulo* n. For example, 5 and 11 are congruent *modulo* 3 [43].

Prime roots

Let m and p be integers, m is said to be a prime root of p if any integer co-prime to p is congruent to the power of g *mod* n. Considering the set of integers under the operation of multiplication *modulo* 14. It follows then that 3 and 5 are the only prime roots *modulo* 14 [43]

Having considered the required mathematics the Diffie-Hellman key exchange is now considered. While the Diffie-Hellman key exchange is not truly a public key cryptographic system, it laid the foundations used by later public key systems.

2.2.2 Diffie-Hellman key exchange

The Diffie-Hellman key exchange algorithm is of importance to this thesis as it explains a technique for secure key exchange. Understanding this is helpful when trying to understand more complex key exchange mechanism.

The Diffie-Hellman key exchange algorithm was the first public-key cryptographic scheme to be published [35]. The scheme exploits the difficulty of the discrete logarithm problem for the field of the multiplicative integers *modulo* n. The Diffie-Hellman Key exchange protocol allows for the exchange of cryptographic keys through an insecure channel. This provides a solution to the key distribution problem experienced by symmetric key encryption [9]. This key exchange algorithm is now illustrated through an example.

Assuming that Alice and Bob wish to share a cryptographic key over an insecure channel, the following series of steps would allow for this using Diffie-Hellman key exchange [9].

1. Both Alice and Bob decide upon a suitable prime p and an integer m such that m is a prime root of p and that both m and p can be made public.

2. Alice then selects some integer m_a. She then computes $y_a = m^{ma} \ mod \ p$, and sends this value of y_a to Bob.

3. Bob then selects some integer m_b. He then computes $y_b = m^{mb} \ mod \ p$ and sends this value of y_b to Alice. The values y_a and y_b are commonly called Diffie-Hellman public values.

4. Alice then computes K, the secret key, by calculating the value $L = y_b^{m_a}$.

5. Bob then computes K by calculating the value $L = y_a^{m_b}$ [9, 35].

It can mathematically shown that both Alice and Bob will arrive to the same value for K. The crux of this protocol lies in the fact that it is computationally difficult to calculate m_a or m_b from y_a or y_b respectively. Being able to easily calculate these values would be equivalent to producing a solution to the discrete logarithm problem.

In the next section the successor to the Diffie-Hellman key exchange, RSA, is considered

2.2.3 RSA

While the Diffie-Hellman Key exchange protocol provides a solution to the key distribution problem, it does not provide a practical public key cryptographic system.

In 1978 Ronald Rivest, Adi Shamir and Len Adlemar created the first public key cryptographic system [35]. The processes followed in RSA can be described as follows [9].

1. Generate two large primes which shall be called p and q. The choice of p and q should be uniformly random and they should be of a similar bit length [9].

2. Calculate the product of these two primes which shall be called n. That is $n = p * q$ [9].

3. The the number of integers that are less than n and are relatively prime to n is then calculated. This can be calculated making use of the Euler-Phi functions. That is $\varphi(n) = (p-1)(q-1)$ where $\varphi(n)$ is the number of integers less than n and relatively prime to n [9].

4. A random number, which shall be called b, is selected with $1 < b < \varphi(n)$ and b is relatively prime to $\varphi(n)$. This ensures the existence of a multiplicative inverse [9].

5. Calculate $a = b - 1 \ mod \ \varphi(n)$ [9].

6. The values of a, p and q are kept secret while making n and b publicly available [9].

Encryption of the plain-text occurs in blocks with each block less than $log_2 n$ bits in length. The cipher-text can be generated by making use of b and n in the following relation, $c = x^b$. The plain-text can be regenerated by calculating $x = c^a \ mod \ n$. The crux of the this scheme is the difficulty in factoring large numbers efficiently and the task of finding the e_{th} roots of a composite number n whose factors are not known [9].

Having completed the discussion on public key cryptography, the next section will discuss cryptographic hash functions.

2.3 Cryptographic hash functions

A cryptographic hash function takes a message of arbitrary length and produces a fixed length output which is a called a fingerprint, hash or message digest. Hash functions are used to verify the integrity of messages or files that have been transferred. A good cryptographic hash function is one that is resistant to collisions. A collision occurs when two messages, x and y with$x \neq y$ but $h(x) = h(y)$ where h is a cryptographic hash. Popular hash functions include SHA-1, MD5 and RIPEMD-160 [19]. MD5 will now be considered as an example cryptographic hash.

2.3.1 MD5

The MD5 hash function was developed by Ronald Rivest at MIT as an improvement to the existing MD4 hash [35]. The MD5 hash function takes a message, which shall be named x, of an arbitrary length and produces a 128-bit hash known as$H(x)$. The MD5 algorithm consists of the following five phases [9].

Appending padding bits

The message is padded with a single one-bit and a number of zero-bits such that length of the message is a multiple of 512. Padded bits are always added

even if the original message is 64-bits. This implies that the number of bits padded on is between one and 512 [9].

Append the length

The 64-bit representation of the original message is appended to the end of the new padded message. If the length exceeds 2^{64} then only the lower order bits of the message are appended. At this point the message will be exactly divisible by 512[9].

Initialize the Message Digest Buffer

The buffer used to compute the hash is 128-bist long. This buffer is formatted as four 32-bit registers labeled A, B, C and D. These registers are initialized to the values shown in Figure 2.1 [35].

Register	Value
A	01 23 45 67
B	89 ab cd ef
C	fe dc ba 98
D	76 54 32 10

Figure 2.1: Block diagram of initialization values for registers during a MD5 hash

Process the message

The message is processed as sixteen word blocks of 32-bits each. Let X and M denote word blocks and the message $X[i]$ an element of that word block. The algorithm that describes this phase is shown in Algorithm 2.2 [9].

Algorithm 2.2 Process Message phase of MD5

```
for (int i = 0 ; i < n/16; i++)
{
  for( int j = 0 ; j < 15; j++)
    X[j] = M[i * 16 + j]

  A ^ A = A ;// bit-wise and of A with itself
  B ^ B = B; ;
  C ^ C = C D ^ D = D ;
  Round1();
  Round2();
  Round3();
  Round4();
  A = (A + AA) mod
  B = (B + BB) mod
  C = (C + CC) mod
  D = (D + DD) mod
}
```

Where Round 1 through to Round 4 are auxiliary functions that make use of a 64-bit element table T[1...64] where $T[i] = 2^{32} \times abs(sin(i))$. The exact details of these rounds has been omitted but may be found in [9].

The next section discusses the components of the TLS architecture.

2.4 TLS

It is important to understand the underlying architecture for a number of cryptographic protocols in order to provide a design for a framework that allows for the decryption of encrypted traffic by an arbitrary cryptographic protocol. This section considers hybrid cryptosystems, the TLS Handshake and trends in TLS and SSL. Note that Chapter 3 is devoted to a discussion of the specifics the TLS protocol related to this thesis while this section considers some of the literature and theory related to TLS.

2.4.1 Hybrid Cryptosystems

It has already been discussed that symmetric cryptography suffers from key distribution issues. Asymmetric cryptography solves this problem by allocating a private and public key pair to each party allowing for ease of distribution. For example, Bob generates a public and private key pair and then distributes the public key to all those who wish to communicate with Bob. Eve who received a public key from Bob then encrypts some plain-text with the key and then sends the cipher-text to Bob. This plain-text can only be decrypted using Bob's private key. Unfortunately public key cryptography relies on intensive mathematical computations which are far more costly than symmetric key cryptography in terms of computational power required.

The aim of hybrid cryptosystem is to solve these two problems of key distribution and computational expense by combining the desirable components of public key and symmetric cryptography together. This is achieved by using public key cryptography to secure the symmetric keys. Once this distribution has been completed the decrypted symmetric key is used for the actual encryption and decryption of messages being passed [10].

The TLS protocol makes use of a hybrid cryptosystem. The TLS handshake uses public key in the form of RSA to perform key exchange. Once this key exchange has been completed the decrypted symmetric keys are used to encrypt application data.

2.4.2 The TLS Handshake

The TLS Handshake is the part of the TLS protocol in which key generation and distribution occurs. Thus it follows that it is of special significance to this thesis.

During this phase decisions are made as to what cryptographic parameters are to be used for the actual TLS connection. This includes deciding on the protocol version, selecting a cipher suite and performing the secret key exchange. The remainder of this section outlines the process followed during the TLS Handshake.

Connection Establishment

The client sends a Client Hello message to the server. The server then possibly responds with a server hello message. If there is no response then a fatal error occurs and the connection is closed. These Hello messages establish: the protocol version to be used, session ID, cipher suite to be used, compression algorithm to use, clientHello.random and ServerHello.random [9].

Key Exchange

The actual key exchange may consist of up to four messages containing: the Server Certificate, the Client Certificate, the Server Key exchange and the Client Key Exchange. If the Server Certificate is to be authenticated it is sent after the connection establishment phase. If the server passes the authentication, it may request the Client Certificate. Finally the client sends the Client Key Exchange message to the server. This contains the Premaster Secret which has been encrypted using the server's public key.

Key Exchange Completed

After receiving the Client Key Exchange message the server responds with Cipher Spec Change message. This message implies that all data sent after this message will be encrypted. The client responds with Cipher Spec Change message as well. The Handshake is now completed [11].

2.4.3 Practices in TLS and SSL

It has already been mentioned that cryptographic protocols are a popular method of securing web servers. It should be considered that simply providing support for cryptographic protocols is insufficient in terms of providing adequate security. Homin *et al.* [**?**] produce a tool to perform the analysis of over 19,000 web servers employing SSL/TLS. They conclude from their results that in 2006 approximately 85.37% of the over 19,000 web servers still provided support for SSLv2.0 which is a fundamentally flawed protocol due to SSLv2.0 weakness to Man in the Middle (MITM) attacks. While 66.55%

of servers still supported DES-40 encryption even though the US export laws limiting the key length of DES to 40-bits was no longer in effect.

It is unwise to still provide support for SSLv.2.0 as its well documented that MITM attacks can force the adoption of a weak encryption protocol like DES-40 creating a large and exploitable vulnerability for brute force attacks. While adaption of new algorithms such as AES is prevalent the rate at which old standards are no longer being supported is not sufficiently rapid.

2.5 SSH

This section shall consider the components and architecture of SSH. Such a section is important as it gives insight into the structure of other cryptographic protocols which is required for the overall system design.

The SSH-1 protocol was developed in 1995 by Tatu Ylönen, a researcher at the Helsinki University of Technology in Finland . Its the logical successor of Telnet providing encryption to the communications made. Like Telnet, SSH's primary use is to provide a command line interface for a remote machine. SSH allows for transparent encryption, that is the user is unaware of the encryption and decryption occurring in the background. It is critical to realize that SSH is a protocol and not a product and as such has a number of implementations. SSH provides its users with three basic security features : authentication, encryption and integrity. SSH provides support for secure remote login, secure file transfer, secure remote command execution and port forwarding.The core of SSH is the Binary Packet Protocol (BPP) which performs the underlying symmetric encryption and authentication [33].

2.5.1 Components of SSH

In this section the various components that make up the SSH protocol are considered. These components include :

- SSH Server
- SSH Client
- SSH Session

- SSH Keys
- Key Generator
- Agent

SSH Server and SSH Client

A SSH Server is a program on the host machines that handles incoming SSH connection dealing with the authentication and authorization of users. In UNIX this is usually performed by a program named SSHD but there are Windows implementation such as Bitvise WinSSHD. A SSH Client is a program that makes requests for secure remote login and secure file copy. Typical SSH clients include Putty, SCP, SFTP and Bitvise Tunnelier [33].

SSH Session

A SSH Session is a persistent connection made between a SSH Client and a SSH Server. The SSH Session begins when the server authenticates the client and ends once the connection is closed [33].

SSH Keys

Keys are used as the random component to initialize the cryptographic functions. Keys used by SSH are the user key, host key and session key. The user key is the asymmetric key used by a server as a way to identify the client. The host key is also an asymmetric key that is used to prove the identity of the server to the client. The session key is a randomly generated symmetric key used for encrypting the communication between an SSH client and server. It is shared by the host and client in a secure manner during the SSH connection setup so that an eavesdropper will not discover this key. Both sides then have the session key which they use to encrypt their communications. When the SSH session ends the key is destroyed [33].

Key generator

A program that creates persistent keys for both users and hosts.[33].

Agent

An Agent is a program that caches user keys in memory so users do not have to keep retyping their pass phrases. The Agent responds to requests for key-related operations, such as signing an authenticator, but the Agent will not disclose the keys themselves. It is a convenience feature. OpenSSH and Tectia have the agent ssh-agent, and the program ssh-add loads and unloads the key cache[33].

2.5.2 Architecture of SSH

The SSH protocol consists of four independent protocols listed below :

SSH Transport Layer Protocol (SSH-TRANS) :

SSH-TRANS allows for the initial connection to be made and provides server authentication, basic encryption and integrity services. Once a SSH Transport Layer Protocol connection is made a SSH Client has a full-duplex byte stream connection to an SSH Server [33].

SSH Authentication Protocol (SSH-AUTH)

Following a successful SSH-TRANS connection the SSH Client may use the SSH-AUTH protocol, using the SSH-TRANS connection, to authenticate with the server. SSH-AUTH defines an abstraction in which many different implementations of authentication could potentially be used. SSH-AUTH only specifies the format and order of authentication.[33].

SSH Connection Protocol (SSH-CONN) :

After authentication has occurred the SSH client may call the SSH-CONN protocol to provides additional services using the SSH-TRANS connection. These services include support for multiple interactive and non-interactive sessions, terminal handling; data compression; and remote program execution [33].

SSH File Transfer Protocol (SSH-SFTP)

A client application may use SSH-SFTP over an SSH-CONN channel to allow for secure file transfer for file manipulation [33].

SSH is designed to be modular and extensible. All of the core protocols mentioned above provide abstract services that ensure a minimum level of functionality provided and requirements they must met. However multiple mechanisms for fulfilling these requirements and functionality are allowed. All the critical parameters of an SSH connection are negotiable [33] . including the methods and algorithms used in :

- Session key exchange
- Server authentication
- Data privacy and integrity
- User authentication
- Data compression.

In conclusion it is interesting that the basic structure of the SSH protocol is similar to the TLS protocol. That is, both are client-server orientated and they both make exchange cryptographic parameters and make use of key generator function in order to derive the keys used for decryption.

2.6 Related tools

A number of tools exist that allow for the decryption of encrypted traffic. Some of these are considered in this section. It is noted the problem with

all these tools is that they are sufficiently generic and will only allow the decryption of traffic encrypted by a specific protocol.

2.6.1 SSLDump

SSLDump [32] is an SSL and TLS network protocol analyser which identifies TCP connections on the chosen network interface and then attempts to interpret them as SSL/TLS traffic. When it identifies SSL/TLS traffic it decodes the records and displays them in a textual form to standard out. If SSLDump is given the cryptographic keys involved it can be used to decrypt the traffic passing through.

2.6.2 Wireshark and Tshark

Wireshark [8] is a GUI based network protocol analyser logs all incoming and outgoing traffic filtered by user specified rules. Additionally Wireshark can read files from other applications that produce PCAP captures. Wireshark provides the ability to decrypt SSL and TLS encrypted streams provided the negation protocol uses RSA for key exchange and the RSA private key used by the server is available. Tshark [7] provides the same functionality as Wireshark but is command-line based.

2.7 Related Research

This section deals with some topics related to analysis of cryptographic algorithms and the analysis of encrypted network traffic. In particular discussions on the detection of encrypted applications and running mode analysis are provided.

2.7.1 Detection of encrypted applications

The use of libraries such as *OpenSSL* allows for techniques to add encryption to generic traffic. This creates a problem for the analysis of network traffic as the traffic is now encrypted. For example, most common p2p clients provide

a means to encrypt traffic by means of an encrypted tunnel provided through SSH . This makes it difficult to block or limit certain types of traffic which may be the goal of a network administrator.

Bernaille and Teixeira [2] suggest a system for the early recognition of encrypted applications is outlined and developed with a high degree of success in terms of identification of applications within an SSL connection. They take the approach of using specific parts of the TCP payload to identify the SSL connection by studying said traffic in detail and then producing patterns to be used in detection methods [2]. A similar methodology of analyzing the TCP payloads could be incorporated into the research topic.

2.7.2 Running Mode Analysis

Running Mode Analysis is a technique for the formal analysis of cryptographic protocols. It makes use of conclusions derived from model checking [46]. The central component of Running Mode Analysis involves creating a system including an attacker, a protocol and two parties attempting communication and then discovering all of the possible modes the system can enter. For example, in a three-principal security system there are seven running modes[46]. If it can shown that these seven modes do not exist then the protocol is deemed to be safe within the system. When working with complex protocols, such as SSL, it is a matter of decomposing the more complex protocol into a number of smaller protocols and then performing Running Mode Analysis on each of the simpler protocols. This sort of analysis is often done by hand and provides an interesting means of the verification of the correctness of a protocol. In a by paper Zhang and Liu [46], running mode analysis is performed on the SSL Handshake protocol. While it may not be important to reperform such an analysis, as such research already exists, it's important to understand that many protocols are fundamentally flawed and identification of such flaws when providing analysis of application security would be a useful.

2.8 Conclusion

Some of the core concepts involved in cryptography and cryptographic protocols have been considered. Though, a considerable amount of work done

in this field has omitted, due to its vastness. It is clearly apparent that it is no longer possible to be an expert within Information Security but rather an expert in one of its subsidiary fields. Cryptography is a field of great interest both academically and economically and the intelligent use of cryptography will lead to improved user satisfaction and safety when using networks to perform confidential tasks.

Chapter 3

TLS Technical Specification

This chapter discusses the technical specifications and structures of the TLSv1.0 [11] protocol that are relevant to building Project Bellerophon through detailed worked examples. Some consideration is provided for the SSL protocol as well. How the negotiation phase of the protocols occurs considering the messages and message fields involved in the generation of the symmetric keys for TLS form the primary focus of this chapter. Block diagrams are used to show the results obtained in both hexadecimal format and in ASCII where applicable. This chapter is part experimentation and part theoretical as the messages are obtained experimentally and meaning is interpreted from this results using theoretical knowledge. It is noted that TLS and SSL differ in very minor ways and thus they are often used interchangeably.

3.1 TLS Specification

In this section a broad overview of the structure of a network packet is discussed, followed by the specifics of the structure of the individual protocols headers within a network packet that are relevant to this thesis. Having considered the higher level packet structure, the details of the underlying TLS/SSL messages is discussed. It is noted that the TLS Handshake was considered in Chapter 2.4.2. While the TLS Handshake is revisited shortly it would be well advised to read the before mentioned section before continuing.

The packet structure of messages involved in the TLS Handshake is consid-

ered in the next subsection.

3.1.1 An overview of the packet structure

The way in which data is represented within a packet data structure needs to carefully considered in order to be able to correctly parse said packets. Thus packet structure shall be the primary focus of this section.

For the purposes of this research the following was assumed : only Ethernet shall be considered as the Data Link Layer, IPv4 as the Transport Layer and TCP shall implement the Session, Presentation and Application Layers of the OSI model [39]. The reasoning behind this is that these are the protocols that typically implement those particular tiers of the OSI model and further this implementation is intended to be a proof of concept and not a definitive framework. Appendix B provides C++ structs that were used by Project Bellerophon to parse packets.

Figure 3.1 represents the logical packet structure that is to be considered.

<table>
<tr><td>Ethernet</td><td>IPv4</td><td>TCP</td></tr>
<tr><td colspan="3">SSL Record Layer</td></tr>
</table>

Figure 3.1: Logical packet structure considered

Data in each of these fields may be treated as blocks of hexadecimal values with each value consisting of two hexadecimal digits. At least two hexadecimal digits are needed to represent all of the characters present in the ASCII table. The reasoning behind this is that a single hexadecimal digit can represent a total of sixteen unique values. Thus it follows that two digits can represent $16^2 = 256$ values. Which is sufficient to display all possible ASCII Characters.

Figure 3.2 shows an example of a packet with the various regions marked and containing sample hexadecimal data.

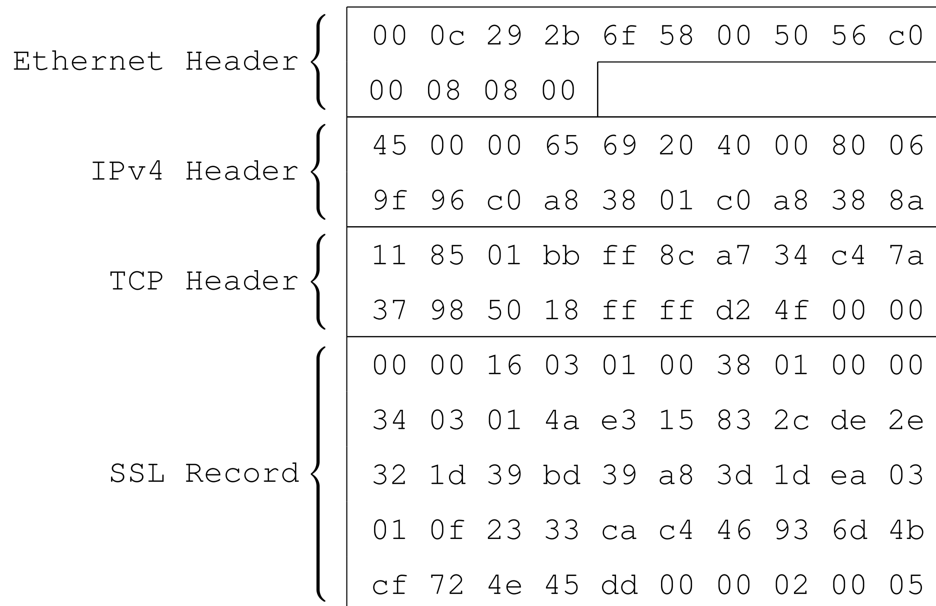

Figure 3.2: Block diagram of packet structure considered with sample hexadecimal values

The individual headers that make up the logical packet structure are now considered.

3.1.2 Ethernet Header

A typical packet will begin with an Ethernet Header [28], which is of a fixed fourteen-byte length. The Ethernet Header consists of a Source MAC, Destination MAC and a Ether Type. The Ether Type is a value that is used to determine which version of Ethernet that is being used. Figure 3.3 shows the logical structure of the Ethernet Header. Note that in Figure 3.3 “ET” is the Ether Type.

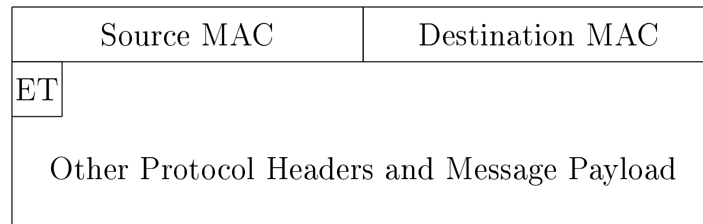

Figure 3.3: Logical structure of an Ethernet packet

Figure 3.4 shows sample values for the fields of the Ethernet Header.

```
     Source MAC { 00 0c 29 2b 6f 58
Destination MAC { 00 50 56 c0 00 08
  Ethernet Type { 08 00
```

Figure 3.4: Example Ethernet Header with sample hexadecimal values

Source and Destination MAC

The Source and Destination MAC addresses can be read directly from their hexadecimal as MAC addresses are usually written in hexadecimal.

Ether Type

For the Ether Type, a value of 0x80 was read. Using Table 3.1 it was determined that the Ether Type was in fact Ethernet. A complete list of Ethernet Types can be obtained from the IANA Ether Types [6].

Hexadecimal Value	Decimal Value	Ethernet Type
0x08 0x00	2048	IPv4
0x08 0x01	2049	X.75 Internet
0x08 0x02	2050	NBS Internet
0x08 0x06	2054	ARP

Table 3.1: Table providing a reduced list of Ether Types

3.1.3 IP Header

The IP Header [29] consists of a number of fields of which only the IP version, Total header length, Source IP Address and Destination IP Address are of interest. Figure 3.5 shows the logical structure of an IP header that was considered.

Version	IHL	
IP Fields not relevant to this framework		
Source IP Address		
Destination IP Address		

Figure 3.5: Logical structure of an IP Header

Figure 3.6 shows sample values for the fields of the IP Header.

Field	Value
Version and IHL	45
Source IP Address	C0 A8 9D 01
Destination IP Address	C0 A8 9D 8A

Figure 3.6: Example IP Header with sample hexadecimal values

Header Length

Following after the Ethernet Header, the next 20 bytes will consist of the IP header in the case of IPv4. It is noted that for different versions of IP this length differs. The IP version used can be verified by considering the value of the first byte. In the given example, the value of the first byte is 0x45 or 01000101 in binary. Further it is known that the IP version number takes up the first four bits of the first byte. In this example the first four bits of the first byte have a binary value of 0100 which converts to a decimal value of four implying the use of the IPv4. The last four bits of the first byte contains

the total header length. In the given example this is a binary value of 0101 which converts to a decimal value of five. As the total header length is the total number of 32-bit words this implies a total header length of 20 bytes, as $32 \times 5/8 = 20$.

Source and Destination IP Addresses

The Source and Destination IP Addresses are of interest as they allow the framework to differentiate between the client and server. This ensures that the correct key is used to decrypt the application data. The Source IP is contained in bytes 27 to 30 of the IP header. For this example the value of the Source IP Address is 0xC0 0xA8 0x9D 0x01. Converting each of the hexadecimal values to decimal an IP Address of 192.168.157.1 is obtained. The Destination IP address follows on directly from the source IP address. In this case the hexadecimal representation is 0xC0 0xA8 0x9D 0x8A from which an IP address of 192.168.157.138 is obtained.

3.1.4 TCP Header

Within the TCP Header [30] only the TCP Header Length field is of interest. The header length is used to determine how long the TCP Header is so that the remainder of the TCP Header can be skipped in order to reach the SSL record.

Header Length

Directly following the IP header is the TCP header. The twelfth byte from the start of the header will contain the Header Length field. The header length calculations are the same as shown in the IP Header length subsection.

3.1.5 SSL Record

The SSL Record is of vital importance to construction of Project Bellerophon. This structure contains all of the cryptographic parameters required to reconstruct the symmetric keys. The basic logical structure consists of a Content

Type, Version, Length and Message Data. It is noted that this is exactly the same structure that is used by SSL. The basic structure of an SSL record is shown in Figure 3.7.

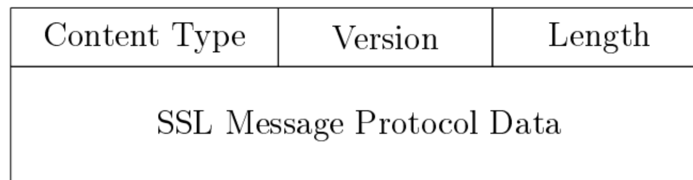

Figure 3.7: Logical structure of a SSL Record

Figure 3.8 shows a sample SSL Record. These fields are now discussed.

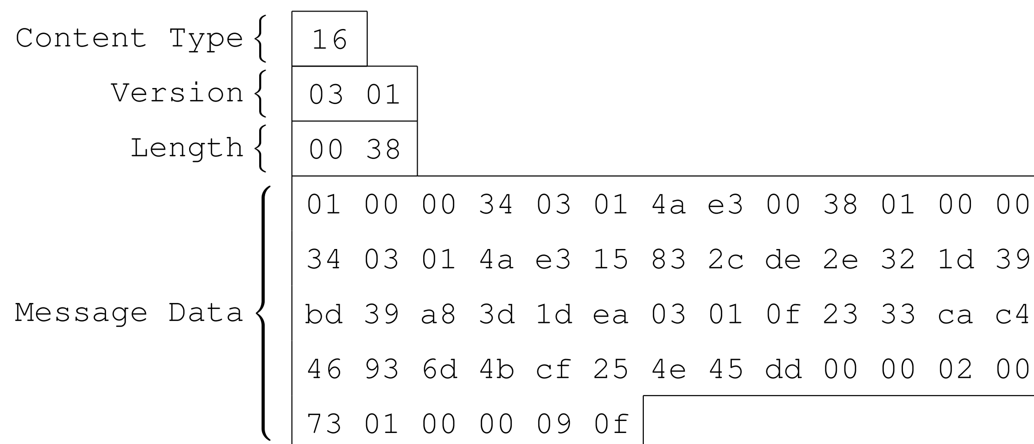

Figure 3.8: Example SSL Record with sample hexadecimal values

Content Type

The first byte of the message, as shown in Figure 3.8, is 0x16 or 20 in decimal, contains the content type of the message sent. Table 3.2 specifies the different types of SSL messages available. Using Table 3.2 it can be concluded that this example message was a SSL Handshake message.

Hexadecimal Value	Decimal Value	Content Type
0x14	20	Change Cipher Specification nessage
0x15	21	SSL Alert nessage
0x16	22	SSL Handshake message
0x17	23	Application message

Table 3.2: Table of SSL Content Types

Version

The next two bytes, 0x03 0x01 in this case, represent the TLS version supported by the sender. The first byte, 0x03, represents the major version of the protocol supported while the second byte, 0x01, represents the minor version supported. Table 3.3 specifies the different version types. Using the table it can be concluded that the TLS version supported by the sender was TLSv1.0.

Major Version	Minor Version	SSL Version
0x03	0x0	SSLv3
0x03	0x1	TLSv1.0
0x03	0x2	TLSv1.1
0x03	0x3	TLSv1.2

Table 3.3: Table providing a reduced list of SSL and TLS Versions

Length

The next two bytes, 0x00 0x38 , following on from the version specify the length of the SSL message. In this case this means a length of 56 bytes. This was determined by treating the two values as a single hexadecimal value and then converting to decimal.

The specifics of each of the message types and there relevance will be discussed in the next section.

3.1.6 SSL Handshake and SSL Message Types

When a TLS enabled client wishes to establish a connection with a TLS server, the client is required to perform a TLS Handshake with the server. The TLS handshake is of great importance to Project Bellerophon as it is during the handshake that a number of cryptographic parameters are passed between the client and server and the generation of symmetric keys takes place. As mentioned in the previous section, the first byte of the TLS record or the "content type" is set to a value of 0x16 if the message is involved in the handshake. Figure 3.9 shows an example SSL Record.

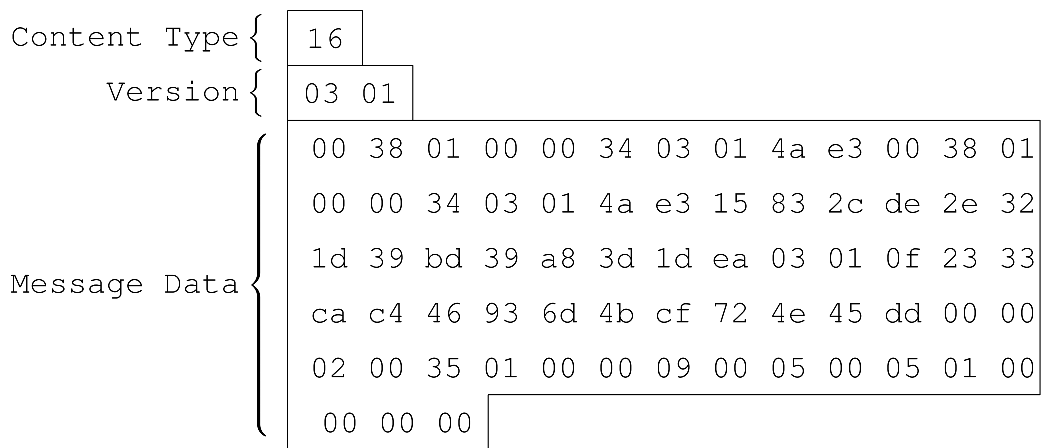

Figure 3.9: Example SSL Record with sample hexadecimal values

The first byte (also known as the Content Type) in the SSL Record as shown in Figure 3.9 is 0x16 and thus it follows that this is a SSL Handshake message.

The two bytes following on from the Content Type represent the version number, which is not of any real interest as this information was already listed previously.

The next two bytes that follow directly on, in this case 0x00 0x01 represent the SSL Handshake message type. Table 3.4 provides a listing of all SSL Handshake message types [11].

Hexadecimal	Decimal Value	Handshake Type
0x00	0	Hello Request
0x01	1	Client Hello
0x02	2	Server Hello
0x0B	11	Certificate
0x0C	12	Server Key Exchange
0x0D	13	Certificate Request
0x0E	14	Server Hello Done
0x0F	15	Certificate Response
0x10	16	Client Key Exchange
0x20	32	Finished

Table 3.4: Table listing TLS Handshake types

Following after the Version field is the actual payload, which is usually a handshake message, of the SSL Record. The rest of this section considers the structure of some of the handshake messages relevant to Project Bellerophon. In particular the Client Hello, Server Hello, Client Key Exchange and Change Cipher Specification messages are discussed.

The next subsection discusses the structure and importance of the Client Hello message.

3.1.7 Client Hello

To establish a TLS handshake the client must begin by sending a Client Hello Message to the server. The Client Hello message consists of a length field, version field, ClientHello.Random field, cipher suite length field and a list of client accepted cipher suites.. The length field is used to determine the number of bytes used by the Client Hello Message. The version field is the same as discussed in previous sections. The ClientHello.Random field is a randomly generated 32-byte value that is used in conjunction with the ServerHello.Random to seed the TLS PRF. The cipher suite length is a two byte value which specifies the number of bytes used to represent all the cipher suites supported by the client. The cipher suites listis of variable length but dependent upon the cipher suite length. This field contains a list of all of cipher suites that the client is willing to accept. Figure 3.14 shows the logical structure of a Client Hello message, note that “VE” represents the version field, ”Len” the length and “CSL” represents the cipher suite length.

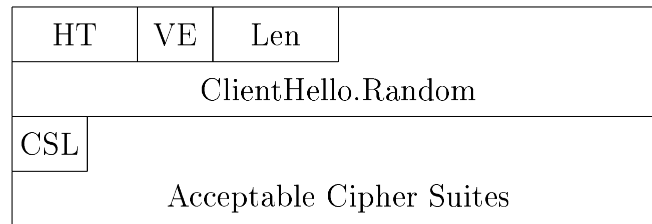

Figure 3.10: Logical structure of a Client Hello message

Figure 3.11 shows an example Client Hello message with sample values for the fields.

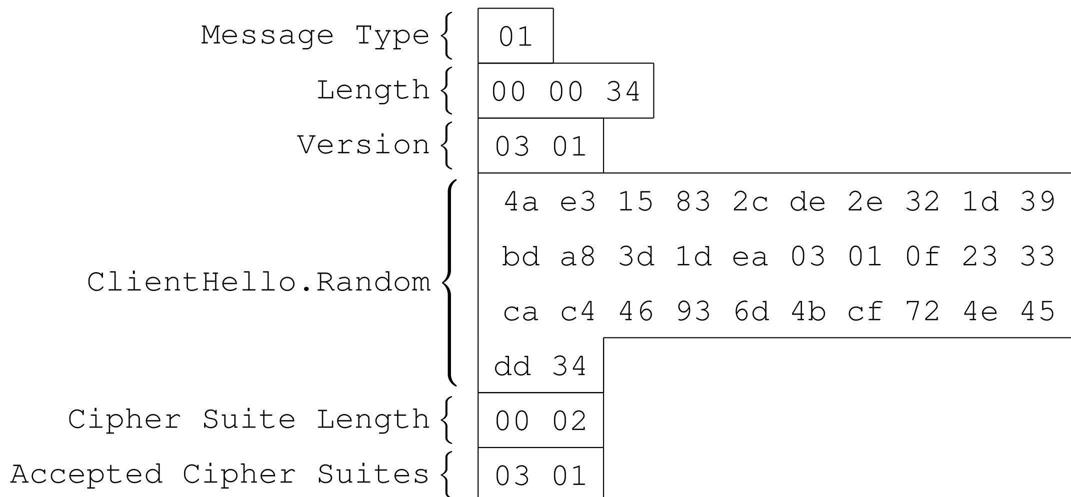

Figure 3.11: Client Hello Message example with sample hexadecimal values

Considering Figure 3.11 the following was concluded :

Handshake message type

The first byte in Figure 3.11 is 0x02. Considering Table 3.4 it can be concluded that this message is a Server Hello Message.

Length

As the first byte is 0x01 it follows from Table 3.4 that message is indeed a Client Hello Message. The next three bytes specify the length of the Client Hello Message, for the given case this is a value of 0x00 0x00 0x34, this corresponds to a decimal value of 52. Thus it follows that the Client Hello Message will consist of 52 bytes.

ClientHello.Random

The next 32-bytes make up what is known as the ClientHello.Random value. The ClientHello.Random value is made of two components, the first eight bytes are the current date and time in standard UNIX 32-bit time format at which the message was sent.. Considering the example values given, 0x4a 0xbc 0x25 0x33 0x7d 0x11 0x64 0x6d, this corresponds to Sep 25, 2009 04:04:35, the time at which the original request was sent from the client. The remaining twelve bytes are random bytes generated by a secure random function. The implementation of this secure random function is dependent upon the client application.

Cipher Suite Length

After the ClientHello.Random field the next field of interest is the cipher suite length. This field specifies the number of bytes used to represent the list of cipher suites supported. In this case two bytes have been given to this representation. Each cipher suite requires two bytes and thus only one cipher suite is supported by the client.

Client Accepted Cipher Suites

Following on from the cipher suite length is a list of the cipher suites supported by the client. It is noted that the next two bytes are 0x00 0x35, this relates to the cipher suite TLS_RSA_WITH _AES_256_CBC_SHA. Table 3.5 shows a reduced list of cipher suites, a full list may be obtained from the TLS RFC [11].

Hexadecimal Value	Cipher Suite
0x00 0x01	TLS_RSA_WITH_NULL_MD5
0x00 0x02	TLS_RSA_WITH_NULL_SHA
0x00 0x3B	TLS_RSA_WITH_NULL_SHA256
0x00 0x05	TLS_RSA_WITH_RC4_128_SHA
0x00 0x35	TLS_RSA_WITH _AES_256_CBC_SHA

Table 3.5: 5.6Table of TLS Cipher Suites

This concludes the relevant sections of the SSL record in Client Hello message. The Server Hello message is considered in the next subsection.

3.1.8 Server Hello

After the server has received a Client Hello message, the server will then respond to the Clients Hello message with a Server Hello message. Figure 3.11 shows an example Server Hello Message with sample values for the various fields. The Server Hello Message consists of a length field, version field, ServerHello.Random field and a accepted cipher suite. The length field is used to determine the number of bytes used by the Server Hello Message. The version field is the same as found previously. The ServerHello.Random field is a random generated 32 byte value that is used in conjunction with the ClientHello.Random to seed the TLS PRF. The cipher suite accepted is a two byte field which specifies which cipher suite which is to be used for the encryption of application data traffic. It is noted that this cipher suite was chosen from the list of supported cipher suites provided in the Client Hello message. Figure 3.12 shows the logical structure of a Server Hello message. Note that in 3.12 “VE” represents the version and “Len” the length.

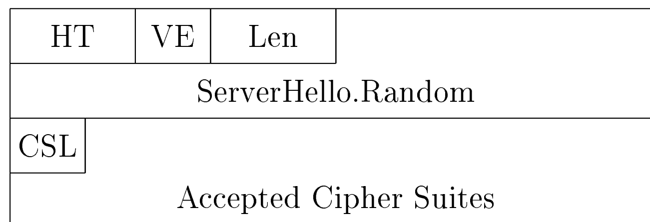

Figure 3.12: Logical structure of a Client Hello Message

Figure 3.13 shows an example Server Hello message with sample values for the fields.

Field	Value
Handshake Type	02
Length	00 00 46
Version	03 01
ServerHello.Random	4a bc 05 02 66 70 e9 d0 f4 96 54 51 14 60 23 ce be 0e 6e ea 6c 2a 1a 1d 09 2f 1b f4 b1 d0 5a 8c
Accepted Cipher Suite	03 01

Figure 3.13: Server Hello Message example with sample hexadecimal values

Considering Figure 3.11 the following was concluded :

Handshake message type

The first byte in 3.13 is 0x02. Considering Table 3.4 it can be concluded that this message is a Server Hello Message.

Length

As with the Client Hello message, the next three bytes represent the number of bytes used by the record. In this case the values of these bytes are 0x00 0x00 0x46 which was then converted to a decimal value of 80

ServerHello.Random

The next 32 bytes makes up a ServerHello.Random. In this case the first component of the random structure is 0x4a 0xbc 0x05 0x02, which corresponds to a date/time of Sept 25. 2009 01:47:14. The other 28 bytes of

random data have no other interpretable meaning other than they form part of the ServerHello.Random structure.

Accepted Cipher Suite

Finally the accepted Cipher Suite is 0x00 0x35. It was concluded using Table 3.5 that TLS_RSA_WITH_AES_256_CBC_SHA1 is the accepted cipher suite.

3.1.9 Client Key Exchange

After the client receives the Server Hello message it will respond with a Client Key Exchange Message. It is at this point that the client will send the encrypted Premaster Secret to the server. Assuming that RSA will be used in the key exchange, the premaster secret will then be encrypted using RSA. The Client Key Exchange message is a simple structure and only consists of a Handshake Type field, length and the Encrypted Premaster Secret. Figure 3.14 describes the logical structure of the Client Key Exchange message.

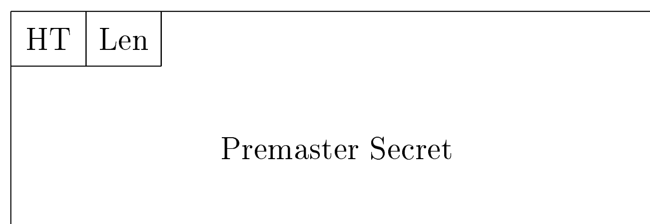

Figure 3.14: Logical Structure of a Client Key Exchange Message

Figure 3.13 shows an example Client Key Exchange Message with sample hexadecimal values for the fields.

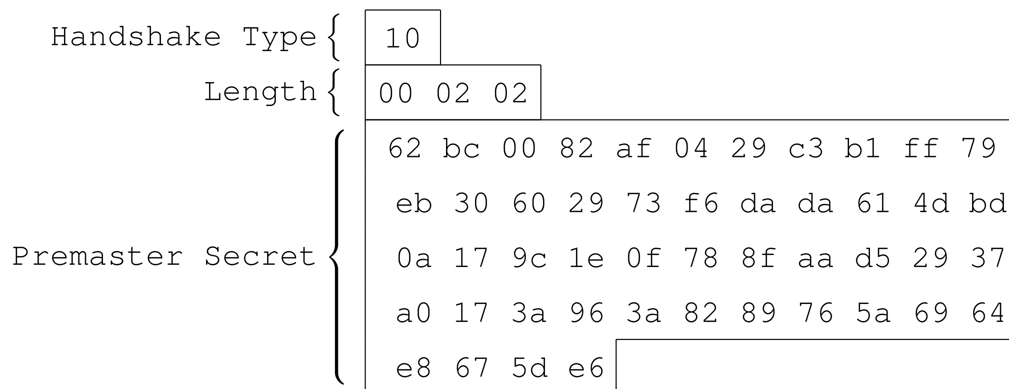

Figure 3.15: Client Key Exchange Message example with sample hexadecimal values

Content Type

The first byte in this case, 0x10, implies that the message is a Client Key Exchange Message. Table 3.5 was used to confirm this result.

Message Length

The next three bytes specify the message length. Which in this case are the bytes are 0x00 0x02 0x02. This converts to a decimal value of 514. This message length includes the bytes used to represent the length and thus it follows that the rest of the message consists of 512-bytes.

Encrypted Premaster Secret

The remaining bytes in this message contains the RSA encrypted premaster secret. It is noted that the Premaster Secret is 512 bytes long. It would be impractical to show the entire Premaster Secret, so only the last 48-bytes has been presented in Figure 3.15.

3.1.10 Change Cipher Specification

After the Client Key Exchange has taken place, the client will send a Change Cipher Specification message to the server. This indicatse that all messages that are now sent from the client will be encrypted. The server then will also send a Change Cipher Message, indicating all messages that are sent after this point will be encrypted. It is important to note these messages when using CBC encryption algorithms as the last sixteen bytes of the messages will form the IV's for decrypting the application data for the client and server. The Change Cipher Specification is a simple message consisting of a content type, length and a Change Cipher Spec message. Figure 3.16 shows the logical structure of a Change Cipher Specification message.

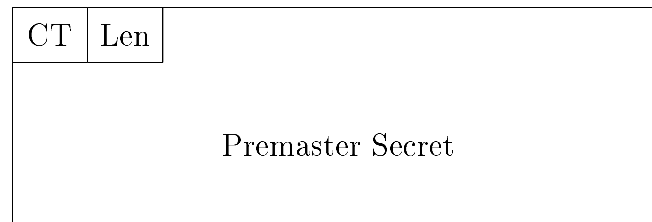

Figure 3.16: Logical structure of a Change Cipher Specification Message

Figure 3.17 shows an example Change Cipher Specification message with sample values for the fields.

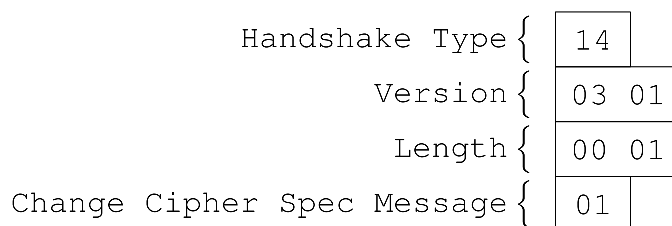

Figure 3.17: Client Key Exchange Message example with sample hexadecimal values

The contents of this mess sage are of little importance. However, the key generation process may begin once both of these messages have been received.

Application Data

After the Change Cipher Specification messages have been sent encrypted application is transferred between the client and server. This data is transferred in the SSL Application Data messages. An Application Data message consists of the following fields : Content Type, Version, Length and Encrypted Application Data. Figure 3.18 describes the logical structure of an Application Data Message.

Figure 3.18: Logical structure of an Application Data Message

Figure 3.19 shows an example Application Data Message with sample values for the fields.

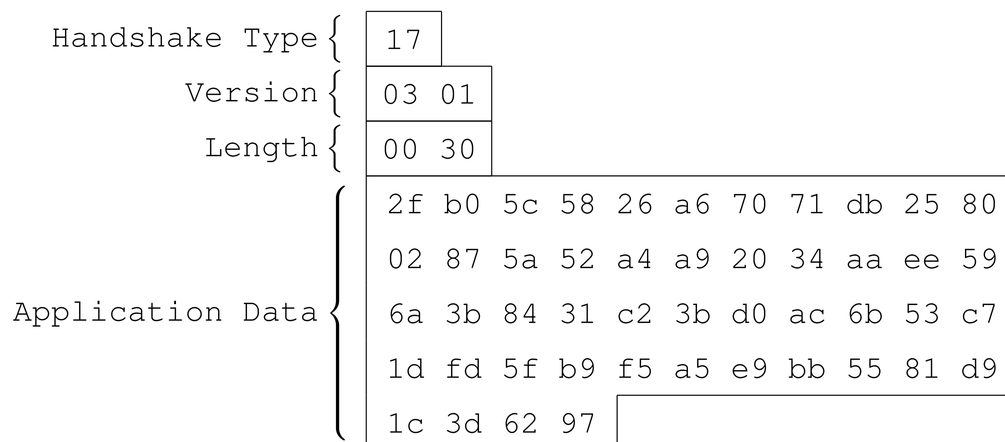

Figure 3.19: Application Data Message example with sample hexadecimal values

Content Type

The first byte in this case, 0x17, implies that the message is an Application Data message. Using Table 3.5 this result was confirmed.

Message Length

The next three bytes specify the message length. Which in this case are the bytes are 0x00 0x30. This converts to a decimal value of 48.

Application Data

This field contains the encrypted application data. Considering HTTPS this data would consists of the HTTP methods and files transfered. Using the symmetric keys generated by the TLS PRF function the encrypted application data is decrypted.

3.2 TLS PRF

The PRF (Pseudo random Function) is the core component of the TLS protocol. This function that takes a number of cryptographic parameters and produces the symmetric keys, initialization vectors and MAC keys that are used for the decryption of application data. The key generations process consists of two separate phases which each involve a separate call to the TLS PRF function. These two phases are namely Master Secret generation and key expansion. In this section the TLS PRF algorithm, Master Secret generation and key expansion is discussed..

3.2.1 The PRF algorithm

The TLS PRF algorithm is vital to the key negotiation phase. The method header to the TLS PRF has the form shown below.

```
TLS_PRF(String Secret,Integer Secret Length,String
 Label,String Random,Int OutputLen)
```

These parameters are now discussed. The secret field is either the Premaster Secret or the Master Secret. The secret length is usually set to a value of 48 bytes. The label string is used to identify whether a Master Secret or key block is being generated by the TLS PRF.

At core, the TLS PRF essentially produces a HMAC_MD5 hash of the secret value and a HMAC_SHA1 hash of the secret and then performs the XOR operation on these two hashes. It is important to note that a MD5 hash produces output of only sixteen bytes while SHA-1 produces a 20 byte output. It follows that these hashes are repeated until the output length is reached. Algorithm 3.1 shows pseudo-code describing the TLS PRF.

Algorithm 3.1 TLS PRF Pseudo-code

```
TLS_PRF(String Secret ,Integer Secret Length,String
Label,Int OutputLen)
{
  MD5Key = ClientHello.Random + ServerHello.Random
  TempHash = HMAC_MD5(Secret ,MD5Key + Label);
  hashedLength = 16;

  while hashedLength < OutputLen
        TempHash = HMAC_MD5(TempHash,MD5Key + Label);
        hashedLength += 16;
        MD5HashedValue += TempHash;

  SHA1Key = ServerHello.Random + ClientHello.Random
  TempHash = HMAC_SHA1(Secret ,SHA1Key + Label);
  hashedLength = 20;

  while hashedLength < OutputLen
        TempHash = HMAC_SHA1(TempHash,SHA1Key + Label);
        hashedLength += 20;
        SHA1HashedValue += TempHash;

        Return SHA1HashedValue XOR MD5HashedValue
}
```

3.2.2 Master Secret generation

The Master Secret is a 48-byte value that is used as one of the final inputs to the TLS PRF in order to generate the key block. The Master Secret acts as a source of entropy for this generation. In order to generate the Master Secret the following components are required.

- Premaster Secret
- ClientHello.Random
- ServerHello.Random
- A label ”

The Premaster Secret is obtained from decrypting the encrypted Premaster Secret which is contained within the Client Key Exchange message. This is 512 bytes of data, however the PRF only takes the last 48 bytes of the Premaster Secret. The ClientHello.Random and ServerHello.Random are the 32 byte values generated by the client and server contained respectively in he Client Hello and Server Hello messages. The label is an ASCII string which is used to identify which phase the TLS PRF is in. For Master Secret generation the label will be “master secret”.

The TLS PRF method header has the following format :

```
TLS_PRF(String Secret,Integer Secret Length,String
Label,String Random,Int OutputLen)
```

Assuming this header, in order to perform the Master Secret generation a the following call would be required :

```
TLS_PRF(PremasterSecret,48,master secret,
ClientHello.Random + ServerHello.Random,48)
```

Note that in the above call ClientHello.Random + ServerHello.Random implies the concatenation of ClientHello.Random and ServerHello.Random.

3.2.3 Key expansion

The TLS PRF is also used to generate the key block from which the symmetric keys, IV’s and MAC keys are cut. Essentially, the HMAC hashes are repeatedly applied until sufficient amount of data is generated. For key expansion, the following parameters are required :

- Master Secret
- ClientHello.Random

- ServerHello.Random
- A lable

In the case of key expansion, the label used is "key expansion". In order to perform key expansion, the following call would be required :

```
TLS_PRF(MasterSecret,48,key expansion,
ClientHello.Random + ServerHello.Random,keyLength)
```

The key length property will be dependent on the symmetric encryption algorithm chosen.

3.3 SSL Specification

As TLSv1.0 is the successor to the SSv3 protocol it follows that the two protocols are similar in nature. The only significant differences worth mentioning is the difference between the TLS PRF and the SSL PRF and the cipher suites supported by said protocols. . Algorithm 3.2 describes the SSL_PRF in pseudo-code and should be contrasted with Algorithm 3.1 which describes the TLS_PRF in pseudo-code.

Algorithm 3.2 SSL PRF Pseudo-code

```
SSL_PRF(String Secret,Integer Secret Length,String
Label,Int OutputLen)
{
  while hashedLength < OutputLen
        val = ' A'
        /*
          The value of val increments
          in the following pattern
          iteration 0 : val = 'A', iteration 1 :
          val ='BB', iteration 2 : val = 'CCC' ...
        */
        TempHash = MD5(secret + SHA(val + Secret +
        ClientHello.Random + ServerHello.Random);
        hashedLength += 16;
        Output += TempHash;
}
```

3.4 Summary

This chapter has discussed some of the technical details of the TLS protocol. A discussion of the protocols relevant to the transfer of network traffic was considered within this chapter. Namely Ethernet, IPv4 and TCP were discussed and logical models and sample headers were provided. A discussion of the TLS Record Layer was provided. This was achieved by considering the packet structure of messages in terms of both logical models and sample message sent during the TLS Handshake phase. All critical parameters required in order to generate the symmetric keys were discussed. These parameters included : ClientHello.Random, ServerHello.Random, Premaster Secret, Master Secret and the TLS PRF labels . The TLS PRF function was considered in terms of its purpose within the TLS protocol and the requirements for key generation and a pseudo-code implementation were provided. Finally some brief discussion on the differences between TLS and SSL was provided. It was concluded that these protocols differ in their implementation of their PRF's and the cipher suites supported.

Chapter 4

Design and Implementation

This chapter discusses the framework design for Project Bellerophon and the experimental design used for testing. The application design section discusses the design of Project Bellerophon considering the framework's structure and the modules necessary for such a framework. In the experimental design section discussion relating to the configuration of a TLS testbed consisting of a TLS server and a TLS client for the purpose of debugging and testing Project Bellerophon is provided. Note that an outline of the tests performed on Project Bellerophon can be found in Chapter 5.

4.1 Application Design

As assumed in the introductory chapters, legitimate access to the data or network connection is assumed. Further it is assumed that the private keys which are required to build the symmetric keys are also available. If these assumptions are not made the thesis scope would need to include methods related to feasible brute forcing or side channel attacks. This section discusses the system design which is then broken up into three core modules and a main (loader) module. These modules are outlined and pseudo-code implementations are provided.

The next subsection considers the logical system design .

4.1.1 Logical system design

In order to build such a system an overall systems design is required which details the individual components of the system and the general flow control. This model was created after considering the structure of the protocols discussed in Chapter 2 and Chapter 3 .Figure 4.1 shows the logical structure and flow of Project Bellerophon.

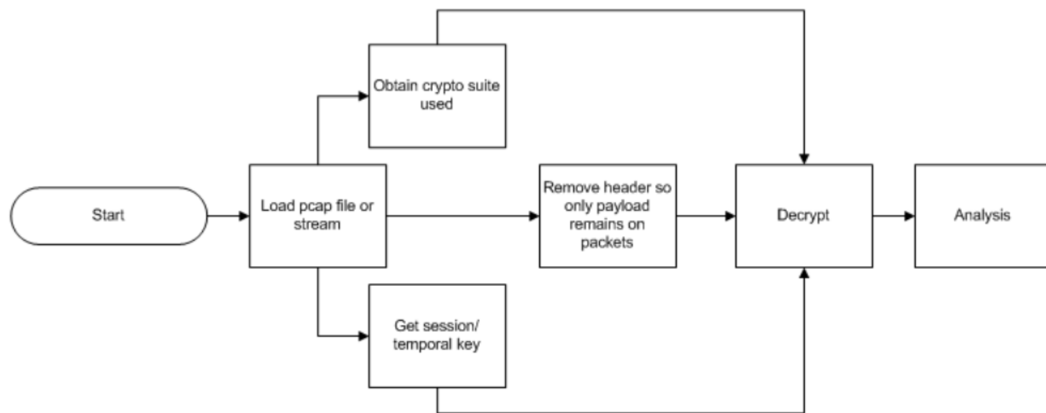

Figure 4.1: Flow diagram showing the proposed systems design detailing the system structure and logical flow

This logical design diagram is now phrased as a textual scenario.

Once the system has been started, a PCAP file or stream is loaded. Project Bellerophon then processes the stream or file attempting to find the the cryptographic suite used by the protocol. It then attempts to rebuild the session keys used for the encryption and decryption of application data. Once these parameters have been obtained any application data that passes through the cryptographic is then decrypted and the plain-text is produced. This plain-text can then undergo analysis by other applications.

From this textual scenario and the logical design it follows that the following modules are required.

- A Decryption module
- Traffic Analyser module

- Packet Capture module
- A main application

Each of these modules are now considered discussing the requirements of each model, the purpose of each module and some of the implementation choices and details. The first module to be considered is the Decryption module.

4.1.2 Decryption module

The Decryption module needs to be able to decrypt generic application traffic when given the decryption key and the name of encryption algorithm used to encrypt the traffic. Thus in order to decrypt encrypted traffic from both the client and server this module will require both the client and server symmetric keys. OpenSSL [12] was chosen as the library to provide the decryption functionality for two reasons. Firstly, *OpenSSL* is cross platform and thus it should be possible to port code relatively easily. Secondly, the EVP Interface [15] provided by *OpenSSL* is an excellent interface for dealing with decrypting traffic encrypted with an arbitrary encryption algorithm. Algorithm 4.1 provides simplified pseudo-code describing the functioning of the Decryption module.

Algorithm 4.1 Decryption module pseudo-code

```
Decypt(Application Data)
{
  Get the decryption;
  Get cipher suite;
  EVP_Decrypt(CipherSuite, Application Data,Key);
  Pass output back to main application;
}
```

The next module to be considered is the Packet Capture module.

4.1.3 Packet Capture module

The Packet Capture module filters traffic from either a live or prerecorded PCAP stream and then presents the appropriate traffic to the protocol analyser module. The Packet Capture module requires functions to stop and start the actual capture and a callback function which passes packets onto an analyser dependent upon some criteria. In addition, the Packet Capture module needs to be able to filter for specific traffic.

As development of Project Bellerophon took place in a Linux environment, *LibPCAP* [18] was chosen to provide the packet capturing abilities. The advantage of using this library is that in theory minimal effort is required in porting to the Windows platform as there is a native Windows implementation of PCAP called *WinPCAP* [4]. *LibPCAP* also allows for the construction and application of Berkeley Packet Filters (BPF) [23].

Algorithm 4.2 provides simplified pseudo-code describing the functioning of the Packet Capture module. Sample implementation of the Packet Capture Module can be found in Appendix C.

Algorithm 4.2 Packet Capture module pseudo-code

```
Start()
{
  Load PCAP Stream
  BuildFilter(TCP)

  while there Is Traffic
  {

    packet = new packet form the stream;

    if packet meets filter requirements
        Give packet to analyser;
  }

}
```

The next module to be considered is the Traffic analyser.

4.1.4 Traffic Analyser module

The Traffic Analyser must be able to parse packets received from the Packet Capture module in order to find the cryptographic parameters required for building the symmetric keys that are to be used by the Decryption module.

Considering a TLS implementation of such a module implies that an implementation of the TLS PRF function is required as this function is used to generate the final key block from which the symmetric keys are cut. The Traffic Analyser needs to be able to differentiate different messages received and be able to identify the Client Hello message for the ClientHello.Random value, the Server Hello Message for the ServerHello.Random value and the accepted cipher suite, the Client Key Exchange message for the encrypted Premaster Secret and finally the Application Data messages for the encrypted data held within. Further the Traffic Analyser module needs to load the RSA Private key used by the TLS server in order to decrypt the Premaster Secret. Once all the parameters required have been collected, the TLS PRF is invoked to perform the symmetric key generation.

OpenSSL was chosen to load the RSA key and to decrypt the encrypted Premaster Secret. The TLS PRF function implementation was built from the TLS PRF function provided in XYSSL (now know as the PolarSSL library) [3] with some modifications to the original source code due to the TLS PRF from XYSSL incorrectly performing bitwise XOR on the HMAC hashes produced.

It is noted that it would be possible to develop other Traffic Analyser modules for other protocols and in this way allow for extensibility.

Algorithm 4.3 provides simplified pseudo-code describing the functioning of the Traffic Analyser module for TLS.

Algorithm 4.3 Traffic Analyser module pseudo-code

```
Analyse
{
  If Message valid SSL Content Type
  {
        If HandShake Type is Client Hello Message
          Store ClientHello.Random;
          Store Client IP Address;

        If HandShake Type is Server hello Message
          Store ServerHello.Random;
          Store Server IP Address;

        If HandShake Type is Client Key Exchange Message
          Store encrypted Premaster Secret;
          Load RSA Private Key;
          Decrypt Premaster Secret;
          and store;

        If Seen Both Cipher Spec Changes
          Generate Master Secret using ServerHello.Random,
          ClientHello.Random and decrypted Premaster Secret;
          Generate Key Block using ServerHello.Random,
          ClientHello.Random and master secret ;
          Cut symmetric keys;

        If Application Data
          Get IP Address;
          Determine if Client/Server;
          Pass to decryption module for decryption;
  }
}
```

4.1.5 Main Application

The main application simply needs to load the other modules and start the Packet Capture module. In this implementation of Project Bellerophon the Main Application is very simplistic.

Finally a complete class diagram consisting of all the modules is shown in Figure 4.2 .

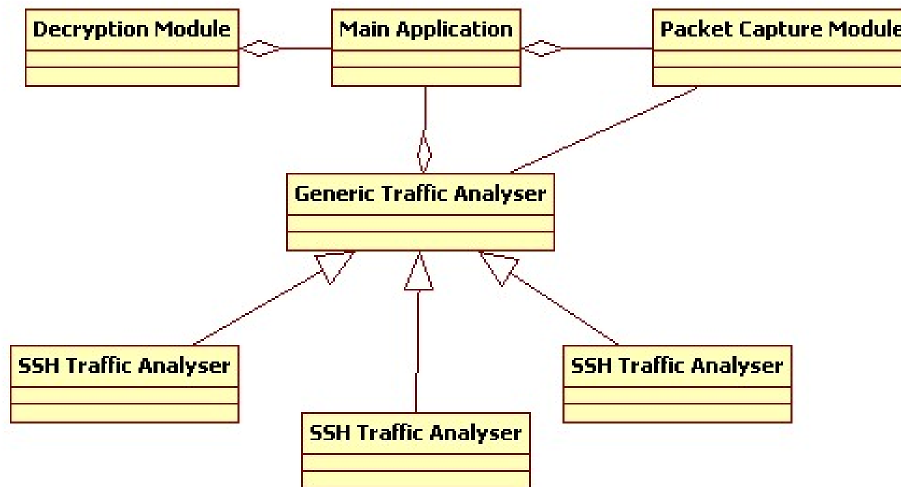

Figure 4.2: Class diagram of Project Bellerophon

The next section considers the design of a testbed used for testing the implementation.

4.2 Testbed Design and Construction

This section discusses constructing a TLS server and a TLS client for the purposes of testing Project Bellerophon. Again it is noted that that an outline of each experiment performed is provided in the appropriate section in chapter 5. An *OpenSSL* enabled *Apache web server [13]* was configured to accept TLS 1.1 connections thus acting as the TLS server and the Opera web browser was configured to act as the TLS client.

The configuration of the TLS server is now discussed.

4.2.1 Configuring the TLS server

An Ubuntu 9.04 Virtual Machine was prepared, running in VMWare 6.5.1. The configuration process started off by installing *OpenSSL* and *Apache*. The console command for the installation of these software packages follows below :

```
sudo apt-get install apache2 apache2.2-common
apache2-utils openssl
```

After *Apache* and *OpenSSL* had been successfully installed, it was necessary to generate a RSA private key and a certificate that can be used during the negotiation phases in TLS. OpenSSL was used for all of the key generation and signing requirements. The command below was issued in order to generate a RSA private key :

```
openssl genrsa -out keyName keyLength
```

A “keyName” of serverkey and a “key length” of 4096 were chosen as the parameters in the above command. After creating the key, a CSR (Certificate signing request) was needed in order to create a certificate. The command below created a CSR :

```
openssl req -new -key server.key -out server.csr
```

This request is usually then sent to a Certification Authority (CA) such as Thwate for processing. However as this was not a real world production environment the CSR was self signed. Web browsers will not accept self signed certificates without confirmation from the user but again as this was not a production environment this issue was neglected. The command below self signed the CSR for 365 days :

```
openssl x509 -req -days 365 -in server.csr
-signkey server.key -out server.key
```

The generated certificate and key are now installed on the web server. This was a simple process of creating a directory to house these files and then moving the files there. The following commands achieved this :

```
sudo mkdir /etc/apache2/ssl/
sudo cp server.crt /etc/apache2/ssl/
sudo cp server.key /etc/apache2/ssl/
```

Next, the *Apache* SSL module was enabled. This was achieved by entering the following command :

```
sudo a2enmod ssl
```

Next, a SSL enabled site was created. This was achieved by adding a configuration file to /etc/apache2/sites-available/. The configuration used is shown below :

```
NameVirtualHost *:443
<virtualhost *:443>
ServerAdmin webmaster@localhost
SSLEngine On
SSLCertificateFile /etc/apache2/ssl/server.crt
SSLCertificateKeyFile /etc/apache2/ssl/server.key
SSLCipherSuite ALL:+HIGH:+MEDIUM:+LOW
#other configuration
<directory /var/www/>
....
</virtualhost *:443>
```

The SSLCipherSuite parameter specifies what cipher suites *Apache* will accept. *Apache* was configured to accept all valid SSL Cipher Suites. In order to accept HTTPS connections *Apache* needed to know to listen on port 443. This was achieved by editing ports.conf to listen on port 443 as shown below.

```
<IfModule mod_ssl.c>
Listen 443
</IfModule>
```

The new SSL site was then enabled by entering the following command :

```
sudo a2ensite siteName
```

Finally *Apache* was restarted. This was accomplished by :

```
sudo /etc/init.d/apache2 restart
```

Files in the /var/www directory were now available through a web-browser pointed at https://hostname/.

Having configured the TLS server, the TLS client configuration was considered.

4.2.2 Configuring the TLS Client

For testing purposes it would useful to configure which cipher suite will be used by TLS for the encryption functionality. As the client specifies what protocols it is willing to accept it is possible to force a specific cipher suite to be chosen assuming the server is willing to accept said cipher suite . This was achieving by disabling all other cipher suites except for the desired suite on the TLS client. Opera, acting as the TLS client, was configured by changing what cipher suites are accepted under the Security Protocols tab in Security Menu in the Preferences tab. Figure 4.3 shows the Security Protocols configuration window for Opera 10.

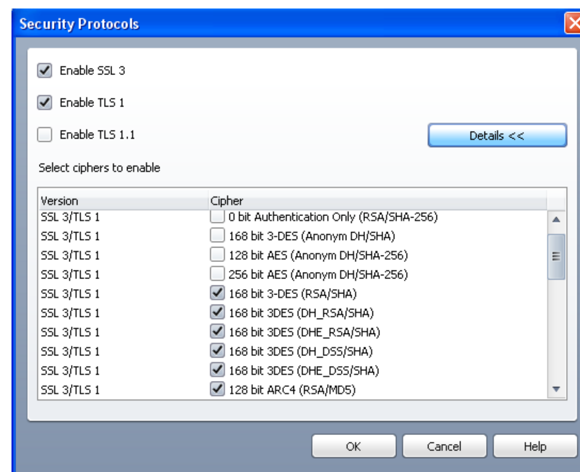

Figure 4.3: Cipher Suite Configuration for Opera 10

4.3 Summary

This chapter has considered the system design and experimental design for Project Bellerophon. In the system design section the various components

for building a generic decryption framework were identified. These components included a Decryption module, a Packet Capture module and a Traffic Analyser module. Each of these modules were discussed with regards to their purpose, requirements and pseudo-code implementation. In the experimental design section it was considered how to construct a TLS server and a TLS client in order to test Project Bellerophon. The chosen TLS server was a TLS enabled *Apache* web server and within this section detailed instructions as to how said server was configured was provided. The chosen TLS client was the Opera Web browser and details concerning its configuration were provided.

Chapter 5

Results and Discussion

This chapter considers results obtained during experimentation. Section 5.1 considers the platform on which testing was perfomed on. Section 5.2 considers the accuracy of the decryption provided by Project Bellerophon by providing the application with a sample capture and observing the resultant output. Section 5.3 evaluates the performance of Project Bellerophon in terms of the effiency of decyptions provided by Project Bellerophon by comparing the time taken for the decryption of encrypted traffic against Tshark. Section 5.4 considers the meaning of the results obtained in Section 5.3.

5.1 Test platform

Testing was performed on the hardware and software platform specified in Table 5.1.

Category	Type
CPU	Intel Core 2 Quad Q6600 clocked at 2.6 GHz
Memory	1x 2Gb Kingmax DDR2 800
Motherboard	MSI P35 Neo 3 (socket 775)
Base OS	Windows XP SP3

Table 5.1: Table containing the hardware and software configuration used for testing

It is noted that while the base operating system running on the test platform was Windows XP, the actual application was run in an Ubuntu 9.04 virtual machine running within VMWare.

5.2 Verifying the accuracy of the decryption

This section considers experiments conducted in order to determin whether Project Bellerophon produces accurate decryption. The experimental methodology is considered and results obtained are provided.

5.2.1 Experimental Outline

A series of simple experiments were conducted to determine whether Project Bellerophon could indeed provide accurate decrypted output, as correctness is a critical goal of such a project. To this end a number of standardized tests were performed which involved copying relatively small files through HTTPS and capturing the resultant traffic as a PCAP file. This PCAP file is then loaded by the decryption tool which then produces the decrypted file as output. The MD5 and SHA-1 hash of the original file can then be compared against the MD5 and SHA-1 of the decrypted output. If these hashes match it can assumed that the decryption was successful. Copying files through HTTPS has the additional effect of producing a HTTP Request from the client and a HTTP Response from the server in addition to the file being copied. For the purposes of this implementation these HTTP messages are ignored. .

For brevity only a single file that was copied through HTTPS will be considered within this section. Additional results can be found on the accompanying CD. A file prepared to the specifics shown in Table 5.2 was created. This file was then transferred through HTTPS. The TLS client used was forced use the TLS_RSA_WITH _AES_256_CBC_SHA cipher suite as described in Section 4.2. Using Wireshark the resultant traffic was captured into a PCAP file.

File Size	27 bytes
Contents (ASCII)	Knowning is half the battle
Contents (Hexadecimal)	4b 6e 6f 77 6e 69 6e 67 20 69 73 20
	68 61 6c 66 20 74 68 65 20 62 61 74 74 6c 65
MD5 of Contents	54 03 d2 9b f8 7f 9d f6 60 8c d3 4d 08 a3 f8 27
SHA-1 of Contents	31 84 bb 58 24 eb 74 19 db 9d 12 79 0d 6b 1b de 42 d0 8a ef

Table 5.2: Table specifying the test file parameters

The PCAP file was given as input to the Project Bellerophon producing the results found in Algorithm 5.1.

The server key was required for decryption as the file was sent by the server and thus it follows that it was encrypted with the server's symmetric key. The server's initialization vector (IV) was required in addition to the key as a chain block cipher type encryption algorithm. The Hex Decrypted-Trimmed field shows the data after it has been decrypted and additional padding bytes have been removed. Performing MD5 and SHA-1 hashes on the resultant plain-text it was observed that these hashes produce the same values as shown in Table 5.2. Clearly, for this small test file Project Bellerophon has produced an accurate decryption.

Algorithm 5.1 Output produced by Project Bellerophon when supplied with captured traffic in PCAP format

```
Project Bellerophron Decryption Output File

Encryption Used : AES-256-CBC

Start of encrypted applicaiton data

The message was sent from the server (192.168.56.1)

Server key :

42 7a 1d 5a 47 7c da a7 3c 8e 48 6d b3 e2 59 92
6e 15 0a 96 69 8d 5b b4 ab 80 83 e4 cd 4f 76 db

Server IV :

c7 1d fd 5f b9 f5 a5 e9 bb 55 81 d9 1c 3d 62 97

Encrypted Form [48] :

2f b0 5c 58 26 a6 70 71 db 25 80 02 87 5a 52 a4
a9 20 34 aa ee 59 6a 3b 84 31 c2 3b d0 ac 6b 53
c7 1d fd 5f b9 f5 a5 e9 bb 55 81 d9 1c 3d 62 97

Hex Decrypt (without padding) [27] :

4b 6e 6f 77 6e 69 6e 67 20 69 73 20 68 61 6c 66
20 74 68 65 20 62 61 74 74 6c 65

PlainText Decrypt :

Knowning is half the battle

End of encrypted applicaiton data
```

Arguably this simple result should show that Project Bellerophon can produce accurate decryptions. However to be pedantic the researcher traced through the entire key negation, encryption and decryption processes to jus-

tify that for this small data set that Project Bellerophon was functioning correctly. The remainder of this section traces through how the symmetric keys were built and how the application data was decrypted. It is noted that additional sample output from Project Bellerophon in terms of the plain-text decrypt and cryptographic parameters generated are provided on the accompanying project CD.

The next sections consider messages parsed and parameters collected by Project Bellerophon in order to obtain the plain-text.

5.2.2 Cryptographic parameters aquired from Server Hello and Client Hello

The ServerHello.Random and ClientHello.Random values were retrieved from the Client Hello, as shown in Figure 5.1 and Figure 5.2.

Field	Bytes
Message Type	01
Length	00 00 34
Version	03 01
ClientHello.Random	4a e3 15 83 2c de 2e 32 1d 39 bd 39 a8 3d 1d ea 03 01 0f 23 33 ca c4 46 93 6d 4b cf 72 4e 45 dd
Cipher Suite Length	00 02
Accepted Cipher Suites	03 01

Figure 5.1: Client Hello message obtained during Project Bellerophon execution

Field	Bytes
Handshake Type	02
Length	00 00 46
Version	03 01
ServerHello.Random	4a e2 f3 72 66 70 e9 d0 f4 96 54 51 14 60 23 ce be 0e 6e ea 6c 2a 09 ff 1b f4 b1 d0 5a 8c 1a 1d
Accepted Cipher Suite	03 01

Figure 5.2: Server Hello message obtained during Project Bellerophon execution

Figure 5.2 also shows that the chosen cipher suite, in this case, is the cipher suite associated with the value 0x00 0x35. Using the values from Table 3.5 it is concluded that this suite is TLS_RSA_WITH _AES_256_CBC_SHA. As RSA was chosen for negotiation, the encrypted Premaster Secret must be obtained and then decrypted using the RSA private key.

5.2.3 Premaster Secret and Client Key exchange

As mentioned in Chapter 3, the encrypted Premaster Secret is stored in the Client Key Exchange message. However, this Premaster Secret is 512 bytes long which is difficult to display within this context and thus only the last 48-bits have been displayed. Figure 5.3 shows the obtained Client Key Exchange message.

Field	Bytes
Handshake Type	10
Length	00 02 02
Premaster Secret	62 bc 00 82 af 04 29 c3 b1 ff 79
	eb 30 60 29 73 f6 da da 61 4d bd
	0a 17 9c 1e 0f 78 8f aa d5 29 37
	a0 17 3a 96 3a 82 89 76 5a 69 64
	e8 67 5d e6

Figure 5.3: Client Key Exchange message obtained during Project Bellerophon execution

Using the RSA decryption algorithm and the RSA private key the encrypted Premaster Secret was decrypted yielding the Premaster Secret as shown in Figure 5.4. Note that "PS" stands for Premaster Secret.

Field	Bytes
Decrypted PS	03 01 9d 62 22 5c ab 2e 80 fa 7b
	22 1b 59 eb 82 2a 5d 7d 9e 55 69
	ff e1 95 e0 3a 18 32 f6 b5 4c 0f
	06 44 82 ef 1a 62 27 5b fc 31 59
	f6 2b bf 49

Figure 5.4: Decrypted Premaster Secret obtained after decrypting the encrypted Premaster Secret

5.2.4 Master Key generation

Using the parameters derived in earlier sections, the Master Secret was generated through the use of the TLS PRF. The Master Secret is generated by calculating the HMAC_MD5 of the Premaster Secret and then calculating the HMAC_SHA1 of the Premaster Secret and then performing the XOR operation on these two results. The ClientHello.Random and ServerHello.Random form the key used by the HMAC functions. The HMAC_MD5 key is formed by appending ClientHello.Random to ServerHello.Random.

The HMAC_SHA1 key is formed by appending ServerHello.Random to ClientHello.Random. Figure 5.5 shows these keys in block diagram format.

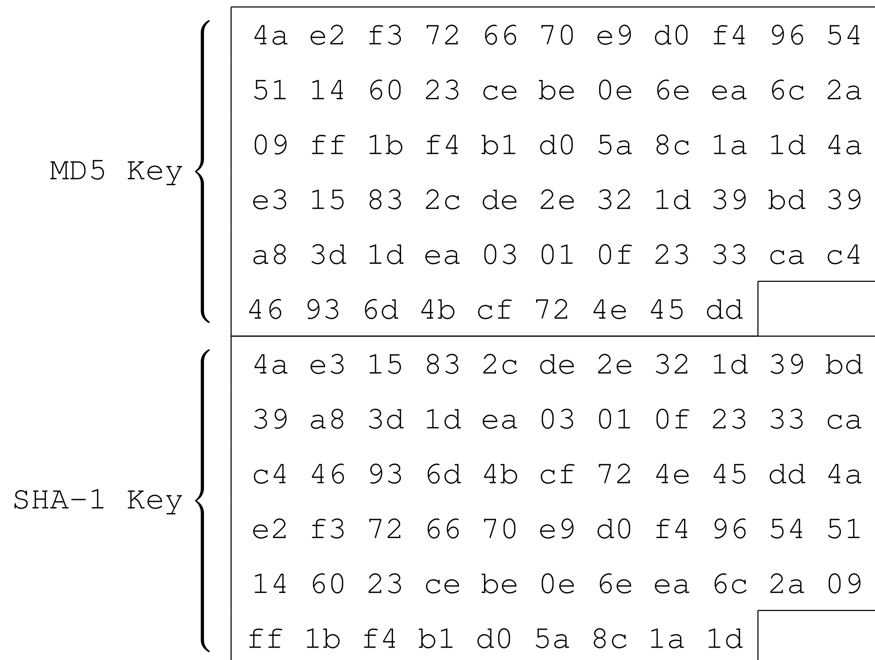

Figure 5.5: Block diagram of HMAC Keys used in the TLS PRF

Figure 5.6 shows the resultant HMAC hashes of the Premaster Secret.

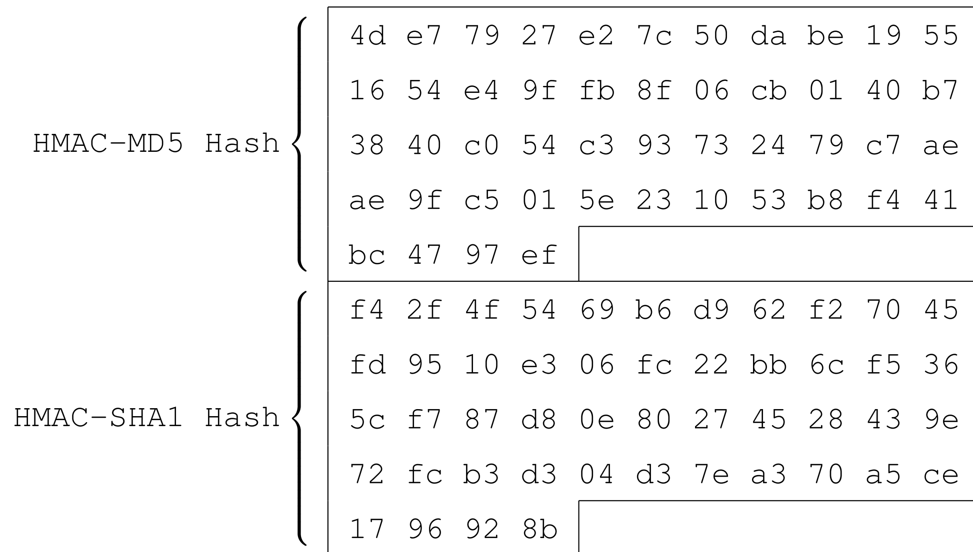

Figure 5.6: Block diagram of HMAC hashes created during the TLS PRF execution

Performing the XOR operation on each of the bytes in the two hashes results in the Master Secret. Figure 5.7 shows the Master Secret generated.

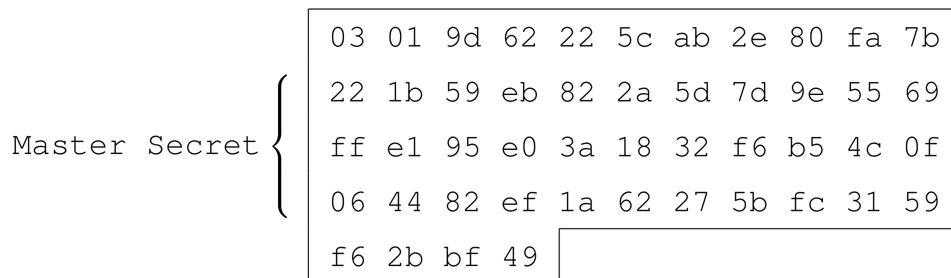

Figure 5.7: Block diagram of the generated Master Secret from the TLS PRF

Now that the Master Secret has been obtained, key expansion may take place using the TLS PRF.

5.2.5 Key expansion

Using the Master Secret as input to the TLS PRF in key expansion mode the following key block was obtained as shown in Figure 5.9.

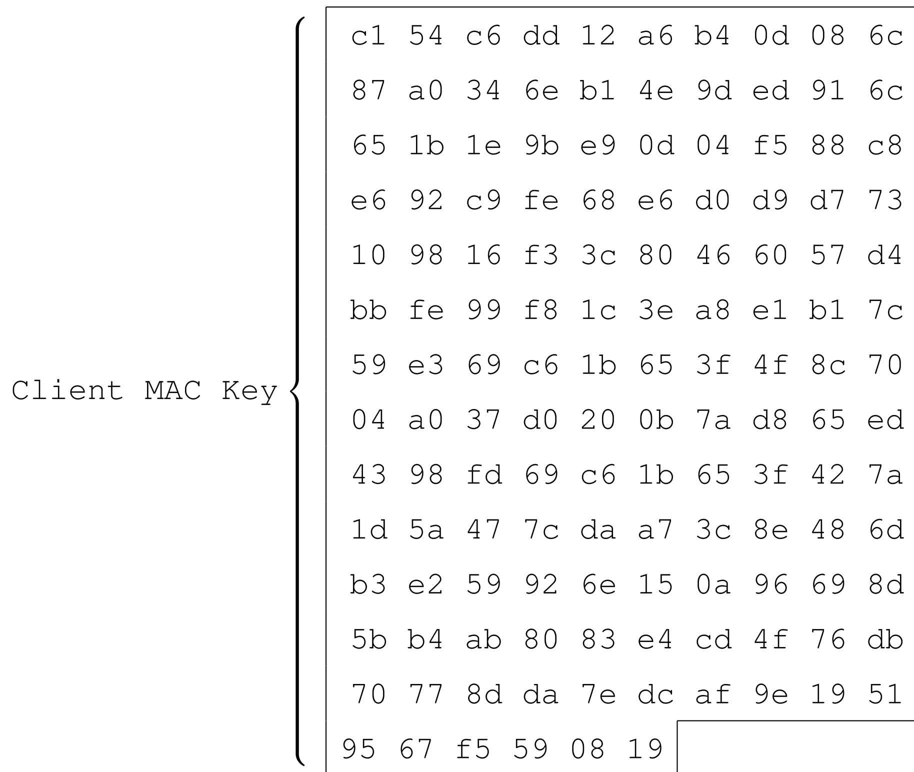

Figure 5.8: Block Diagram of the output of the PRF Key Expansion

This key block was then dissected into the required cryptographic components. It is noted that AES-256- CBC is used as the encryption algorithm, this implies a key length of 32-bytes. As SHA1 is used for authentication this implies that tbe MAC keys of twenty-bytes must be generated. Finally as a chain block cipher is used it follows sixteen-byte IV's are required. Two of each of these cryptographic parameters were required as both the client and server needs one of each. This amounts to a total block size of 136-bytes.. Using the key sizes previously discussed and the fact that two sets of these keys are generated, the key block was cut correctly into a Server Write key, a Client Write key, a Server MAC key, a Client MAC key, a Server IV and a Client IV. Figure 5.9 shows the recovered cryptographic parameters for the encryption and decryption of application data sent by the client. Figure 5.10 shows the recovered cryptographic parameters for the encryption and decryption of application data sent from the server.

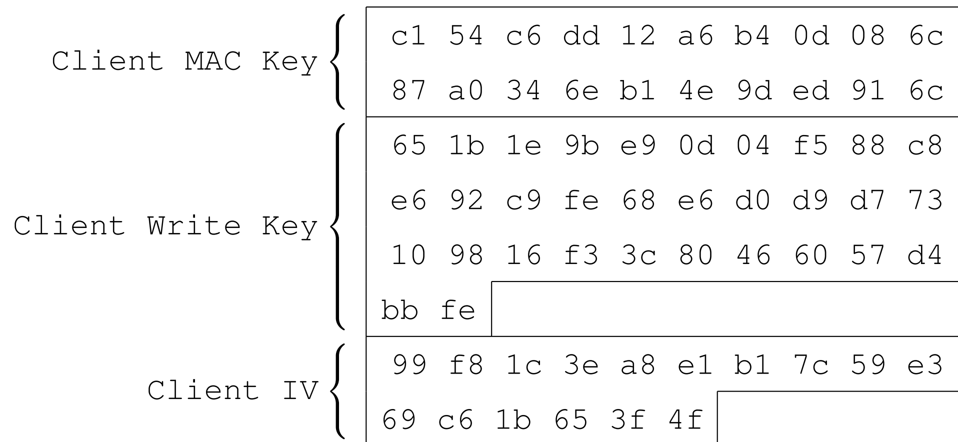

Figure 5.9: Block diagram of recovered client cryptographic parameters

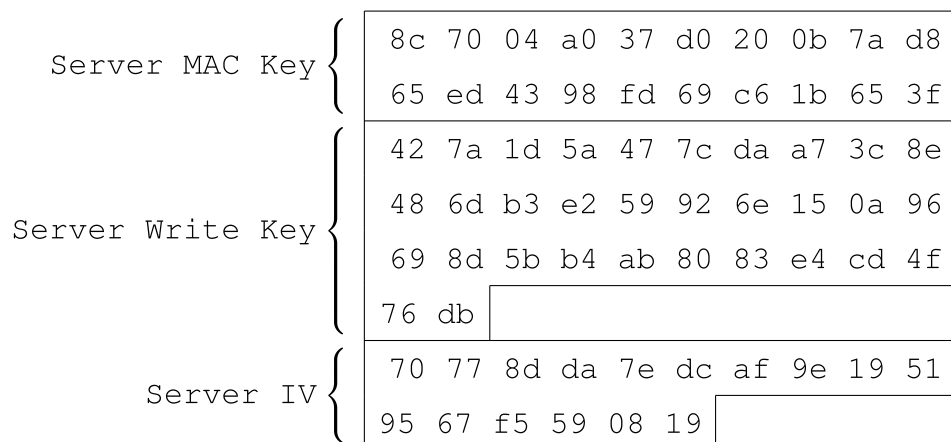

Figure 5.10: Block Diagram of recovered server cryptographic parameters

Having obtained the cryptographic keys it is now possible to decrypt encypted application data.

5.2.6 Recovered Application Data

The encrypted application data was recovered from the application messages. Figure 5.11 shows the message that contains the desired plain-text in encypted form.

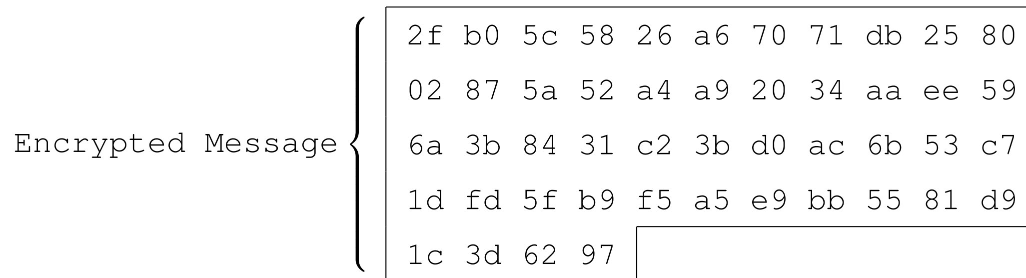

Figure 5.11: Block diagram of Encrypted Application Data

Using the cryptographic keys obtained and identifying which keys are to use based upon the message's IP address, omitted here for brevity, the message was decrypted. Figure 5.12 shows the decrypted message in both hexadecimal and ASCII form. Note that Decrypted Message(H) is the comple decrypted block, Decrypted Message(T) is the decrypted block without the padding bits and Decrypted Message(A) is the ASCII form of the decrypted application (without padding bits).

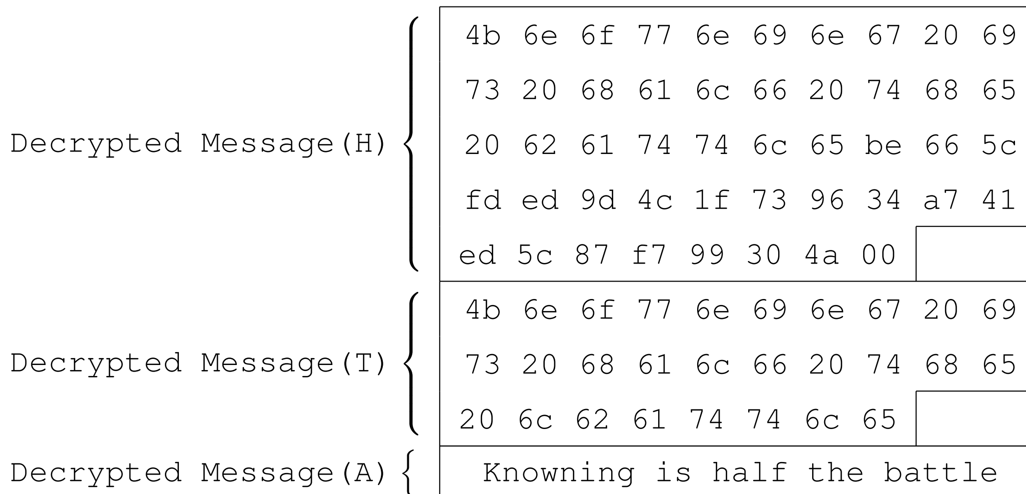

Figure 5.12: Block diagram of Decrypted Application Data

Having shown that Project Bellerophron can produce accurate decryption, the next section now considers the efficiency of Project Bellerophon

5.3 Testing of efficiency

This section discusses the efficiency of the developed implementation. An outline to the experimental process is provided together with results obtained for Tshark and Project Bellerophan

5.3.1 Experimental Outline

To show that the implementation was feasible, the time taken for Project Bellerophon to decrypt encrypted traffic was tested and contrasted with Tshark. A PHP script was developed to create random data of a specific length. This data was then copied to the web directory of the machine running the TLS server. This data was copied through HTTPS and the resulting traffic was recorded to a PCAP file. To measure the time taken for Project Bellerophon the perform the decryption of the encrypted traffic a simple test script was developed using shell scripts to provide timing to millisecond precision. A similar script was used to determine the time taken to decrypt the encypted traffic using Tshark . Details of said scripts can be found in Appendix A.

It would useful to understand the general relationship that exisits bewteen the time taken to decrypt an encrypted file versus the the size of the file as this results can be constrasted with the relationships obtained between time taken to decrypt encrypted traffic and the size of the encrypted traffic for Tshark and Project Bellerophon. To achieve this a baseline was established by considering the time that it takes to decrypt files that have been encrypted using *OpenSSL*.

5.3.2 Establishing a baseline

Knowing the relationship between files size and time to decrypt is useful information as a similar relationship bewteen time to decrypt encrypted traffic and the amount of encrypted traffic is expected. To this end a file was encrypted using AES-256-CBC and then decrypted a 1000 times, measuring the execution time and recording the average of all of the executions. *OpenSSL* was used as the library to provide the encryption and decryption facilities. The use of *OpenSSL* was appropriate as it is the library that provides the

decryption functionality for Project Bellerophon. The script that was used to generate these results can be found in Appendix A. Table 5.3 shows the acquired results.

File Size	100	500	1000	10000	15000	20000
Execution Time	9.32	9.33	9.43	9.35	9.53	9.56
File Size	25000	30000	35000	40000	450000	50000
Execution Time	9.74	9.83	9.89	9.96	10.02	10.13

Table 5.3: Table of File Size vs. Execution time to decrypt a file encrypted using AES-256-CBC

This data was then graphed producing Figure 5.14. It was apparent from 5.14 that there appears to be a linear relationship between time taken to decrypt and the size of the file decrypted. This relationship was then investigated through the use of simple linear regression.

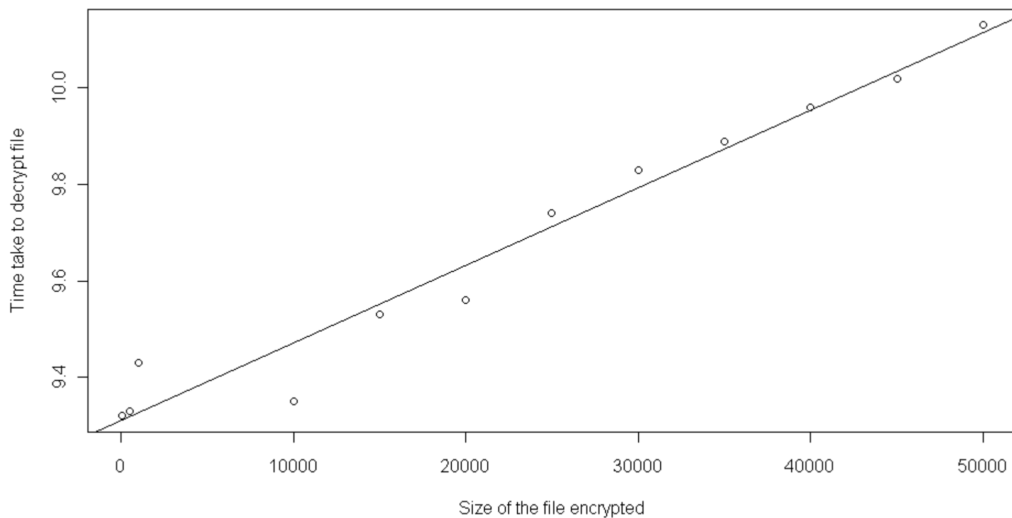

Figure 5.13: Graph of File Size(bytes) vs.Time to decrypt encrypted files(ms) using OpenSSL

The data shown in Table 5.3 was entered into into R (the statistical analysis software package) [24, 14]. A simple linear regression was performed and the

correlation co-efficient was calculated to determine whether the result obtained was significant. From the results of the linear regression and letting x represent the file size and y represent the time taken to decrypt, the following equation was derived :

:$y = (1.607 \times 10^{-5})x + 9.31$

The correlation co-efficient, R^2, was calculated as well :

$R^2 = 0.963136$

The correlation co-efficient reveals how much of the deviation in the dependent variable(y) can be explained by variation in in the independent variable(x) and is used as a measure of the significance of a linear relationship. Generally a correlation co-efficient of greater that 0.95 is considered to be very significant. From the R^2 value it can concluded that the regression is significant and that a significant linear relationship exists between the time taken to decrypt the file and the file size.

In the next section the efficiency of Tshark is considered.

5.3.3 Tshark Timing Results

A similar experiment was then conducted using Tshark. In this case the random file generation script was used to create files of the appropriate length. These files were then copied through HTTPS with the resultant traffic captured using Wireshark. The resultant PCAP was file was given as input to a script which then decrypted the application data using TShark. The script was written in a way such that it repeated this process 1000 times and produced the average time to perform the entire key generation and decryption. Table 5.4 shows the obtained results.

Figure 5.14 depicts the relation between traffic size and the time taken to decrypt encrypted traffic in milliseconds. It appears from the graph that there is a linear relationship between execution time and the amount of traffic decrypted.

Traffic Size	100	200	300	400	500	1000
Decryption Time	391.1	391.61	391.49	392.52	392.65	393
Traffic Size	2000	4000	6000	8000	10000	20000
Decryption Time	392.84	394.89	398.03	394.97	399.47	404.92
Traffic Size	40000	60000	80000	100000	200000	400000
Decryption Time	417.69	432.76	441.15	452.14	510.55	628.8
Traffic Size	600000	800000	1000000			
Decryption Time	712.39	854.96	983.75			

Table 5.4: Table showing time taken to decrypt encypted traffic(ms) vs.traffic size for Tshark(bytes)

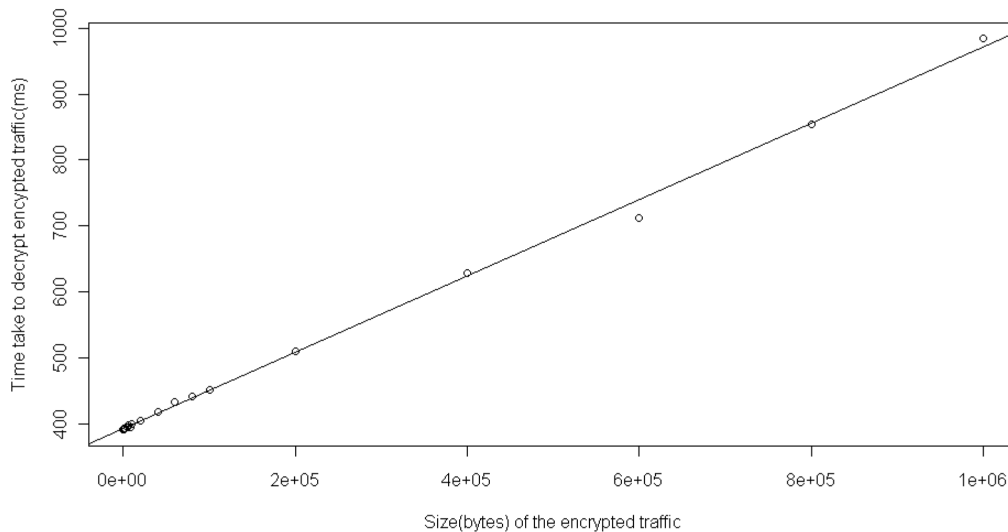

Figure 5.14: Graph of encrypted traffic size (bytes) vs. time taken for decrypting encrypted traffic using Tshark

Linear regression was then applied to the results in Table 5.14. From the results of the linear regression and letting x represent the traffic size and y represent the time taken to decrypt, the following equation was acquired :

$y = (5.780 \times 10^{-4})x + 3.9626 \times 10^{2}$

The correlation co-efficient, R^2, was calculated as well :

$R^2 = 0.999825$

From these results it can be concluded that the regression is significant and that there is a slope of approximately 5.78×10^{-4}, that is for every byte increase in traffic size there is an increase of about 0.58 of a millisecond change in time to decrypt. When compared to the baseline a similar trend in increase in execution time is noted, however Tshark has a much larger y intercept and x co-efficient. The larger y intercept is attributed to the overhead in starting the traffic dissectors and analysers and building the relevant symmetric keys from the cryptographic parameters.

Both Figure 5.13 and Figure 5.14 show clear linear relationships between execution time to decrypt data and the size of the data. This is easily justified as cryptographic algorithms tend to deal with blocks of encrypted data which are then decrypted. The linear relationship follows from the fact that it should theoretically be no more computationally expensive to perform the decryption operations for a single block of encrypted plain text than any other block of encrypted plain-text.

5.3.4 Project Bellerophon Timing Results

Finally, the experiment was conducted using Project Bellerophon to provide the decryption functionality. A script was modified for the testing of the Project Bellerophon and can be found in Appendix A. The results captured are presented in Table 5.4

Traffic Size	100	200	300	400	500	1000
Time to Decrypt	415.63	416.23	415.89	416.76	416.88	417.11
Traffic Size	2000	4000	6000	8000	10000	20000
Time to Decrypt	420.25	426.59	430.243	436.3	442.63	460.69
Traffic Size	40000	60000	80000	100000	200000	400000
Time to Decrypt	475.48	485.17	497.79	530.16	563.35	715.76
Traffic Size	600000	800000	1000000			
Time to Decrypt	850.78	1004.47	1163.56			

Table 5.5: Table showing time taken to decrypt encrypted traffic(ms) vs. traffic time for Project Bellerophon(bytes)

Figure 5.15 depicts the relation between encrypted traffic size and the time taken to decrypt the encrypted traffic in milliseconds. There appears to be

a linear relationship between time taken to decrypt encrypted traffic and the amount of data decrypted, as was discovered in the results for Tshark's execution times. Some "inaccuracies" have been noted, that is for small values of file size it sometimes occurs that it takes longer to decrypt a shorter encrypted traffic size. This is highly illogical and it can be concluded that the accuracy of the timing mechanism may be called into question as the correct decryption is produced.

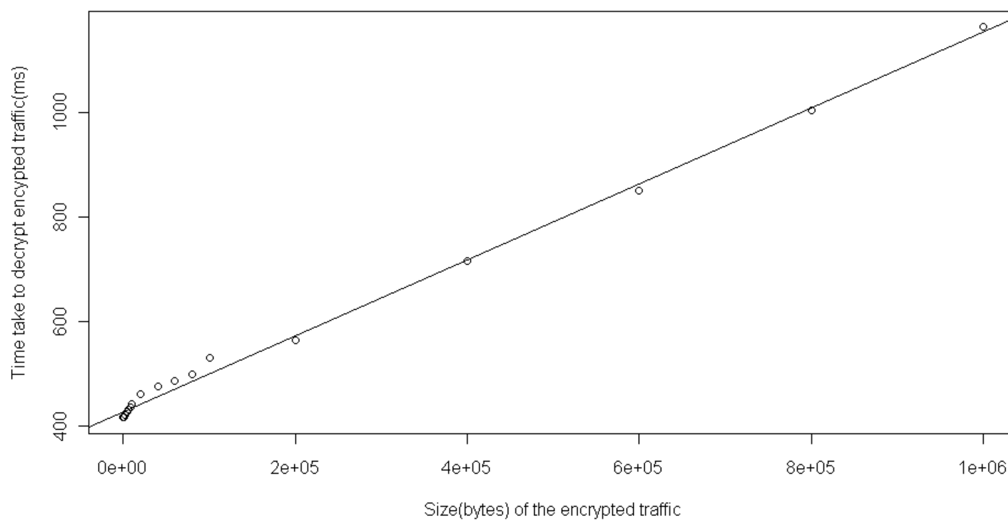

Figure 5.15: Graph of encrypted traffic size (bytes) vs. time taken for decrypting encrypted traffic using Project Bellerophon

The data shown in Table 5.5 was entered in to R and simple linear regression was performed, letting x represent the file size and y represent the time taken to decrypt the encrypted traffic, the following equation was acquired :

$y = (7.272 \times 10^{-4})x + 4.275 \times 10^2$

The correlation co-efficient, R^2, was calculated as well :

$R^2 = 0.99637$

From these acquired results it can be concluded that there is a linear relationship between time taken to decrypt encrypted traffic and the size of the

encrypted traffiic for Project Bellerophon. When compared to the baseline it is noted that both the y-intercept and x co-efficients are much greater than that for the baseline. This is expected due to the additional overhead of parsing packets and symmetric key generation.

The next section considers the results obtained and provides some comparison.

5.4 Comparing Project Bellerophon and Tshark in terms of efficiency

Reconsidering the results of the linear regression in the previous section yields Table 5.6.

	TShark	Project Bellerophon	OpenSSL Decrypt
x co-efficient	5.78×10^{-4}	7.272×10^{-4}	1.607×10^{-5}
intercept	392.6	427.5	9.31

Table 5.6: Results comparison for Tshark, Project Bellerophon and OpenSSL

It is clear from Table 5.6 that Tshark is more efficient at obtaining the symmetric keys required and producing the decrypted output. This is evident by the smaller gradient between execution time and file size that Tshark has when compared to the larger gradient for Project Bellerophon. However, it should be noted that the execution times that Project Bellerophon produced could still be considered reasonable. When both of these results are compared against using *OpenSSL* to decrypt an encrypted file a large difference in both the gradient and intercept was noted. It can be concluded that a considerable amount of additional processing is required to analyze and decrypt encrypted traffic when compared to file decryption.

5.5 Summary

This section has considered the efficiency and the accuracy of Project Bellerophon by performing a series of tests. The accuracy of Project Bellerophon was tested by recording the transfer of a file through HTTPS and storing this recording as a PCAP file. The PCAP file was then run through Project Bellerophon producing a plain-text decrypt and the cryptographic parameters produced. A complete reconstruction of the key generation that had occurred was provided. It was shown that Project Bellerophon produces accurate decryptions for small files encrypted by Project Bellerophon.

The efficiency of Project Bellerophon was considered by comparing the time that it took to decrypt encrypted traffic of varying sizes. These results were contrasted against results for the same experiment but using Tshark to perform the key generation and decryption. It was found that that Project Bellerophon was slower in its key generation and decryption than Tshark. It could however be argued that this difference is not sufficiently significant to suggest that Project Bellerohpron provides inefficient decryptions.

Chapter 6

Conclusion

6.1 Summary

The intention of this thesis was to produce a framework design that could be used to decrypt data that had been encrypted through the use of an arbitrary cryptographic protocol. A literature survey was performed considering symmetric cryptography, public key cryptography, cryptographic protocols and work related to cryptographic protocols. The structure and specification of TLS protocol was considered experimentally by considering the traffic generated when a TLS connection is created. The results of this experimentation were recorded and was complemented with theoretical knowledge of the TLS protocol. After having considered the related literature and the functioning of the TLS protocol a design for a framework that could be used to decrypt traffic generated by an arbitrary protocol was developed including a logical design and the modules required for the correct functioning of the system. A test bed was constructed consisting of a TLS enabled *Apache web server* acting as the TLS server and a configured web browser acting as the TLS client. A number of experiments testing the efficiency and accuracy of the provided implementation were conducted. Tshark was used to contrast the performance of the implementation by comparing the execution times for sample encrypted traffic decryption. It was concluded that the implementation provided accurate decryptions but was less efficient than Tshark.

6.2 Revisiting the research goals

The primary goal of this thesis was to develop a framework design for decrypting traffic that had been encrypted by an arbitrary cryptographic protocol and to provide sample implementation for the TLS protocol. This was achieved and a suitable design was discussed in Chapter 4 with the overall framework consisting of a Decryption module, Traffic Analyser module and a Packet Capture module. This design was shown to be feasible by using *OpenSSL, LibPCAP* and the *XYSSL* to provide sample implementation for TLS.

The secondary goal was to provide some analysis of the implementation in terms of the accuracy and efficiency of the decryptions provided. To this ends a test bed was constructed consisting of a TLS server and a TLS client. The details of which were discussed in Chapter 4.2. The accuracy of the implementation was tested by copying files through HTTPS and comparing the decrypted output from the implementation against the original file. It was found that the implementation provides accurate decryptions for small files. The efficiency of the implementation was also tested and the implementation was shown to produce decrypted traffic slower than Tshark. However it was concluded that the implementation still completed the task in a sensible amount of time.

The tertiary goal was to provide some documentation of the TLS protocol through practical examples. Chapter 3 achieves this goal by providing a step by step walk through of the processes involved in obtaining the parameters used for key generation in the TLS protocol. Chapter 5 provides a further example of the key generation process.

6.3 Possible Extensions

The application framework shown in this thesis was developed as a proof of concept and was not intended to be an industry ready product. At the present time a number of potential features have not as of yet been implemented. Below some possible extensions are outlined :

- The implementation may benefit from techniques to optimize performance. These techniques could possible include parallelization tech-

niques using libraries such as Linda or making use of NVIDIA's CUDA technology to allow for massive parallelization on modern GPU's. These techniques would allow for an increase in performance and would be useful when dealing with large set input sets, seeing as after the symmetric keys have been recovered the application data could be decrypted in parallel.

- Additional modules could be developed to deal with other cryptographic protocols such as IPSec and SSH. As the system has been designed in a way to make it easy to add additional modules. The real difficulty in providing support for additional protocols is understanding the underlying processes involved in the generation of the symmetric keys for the protocol to be implemented.

Bibliography

[1] R. Abrams. A checklist for developing software quality metrics. *Proceedings of the ACM '82 conference*, pages 5–6, 1982.

[2] L. Bernaille and R. Teixeira. *Early Recognition of Encrypted Applications.* pages 165–175. Pierre and Marie Curie University, 2007.

[3] Brainspark. Polar SSL. Online : `http://www.polarssl.org/`, [Last accessed 19/10/2009].

[4] J. Bruno. WinPCAP, The Packet Capture and Network Monitoring Library for Windows. Online : `http://www.winpcap.org/`, [Last accessed 07/11/2009].

[5] B.Schneier. Why Cryptography Is Harder Than It Looks. Online : `http://www.schneier.com/essay-037.html`, [Last accessed 15/04/2009].

[6] A. Cherenson. IANA Ether Types. Online : `http://www.iana.org/assignments/ethernet-numbers`, [Last accessed : 01/09/2009.

[7] G. Combs. Tshark. Online : `http://www.wireshark.org/docs/man-pages/tshark.html`, [Last accessed 05/11/2009].

[8] G. Combs. Wireshark. Online : `http://www.wireshark.org/`, [Last accessed 05/11/2009].

[9] C. Davis. *IPSec, Securing VPNs.* McGraw Hill Publishers, 2001.

[10] A. Dent. Hybrid cryptography. *Information Security: 8th International Conference – ISC 2005, volume 3650 of Lecture Notes in Computer Science*, pages 203–217, 2007.

[11] T. Dierks and C. Allen. RFC 2246 : The TLS Protocol Version 1.1. Online : http://www.ietf.org/rfc/rfc2246.txt, [Last accessed 14/10/2009].

[12] R. Engelschall. OpenSSL. Online : http://www.openssl.org/, [Last accessed : 01/11/2009].

[13] Apache Foundation. Apache - HTTP Server Project.

[14] The R Foundation. The r project for statistical computing. Online : http://www.r-project.org/, [Last accessed : 07/11/09].

[15] J. Jansen. OpenSSL and EVP. Online : http://www.nlnetlabs.nl/downloads/publications/hsm/hsm_node17.html, [Last accessed : 12/08/2009].

[16] D. Kahn. *The Codebreakers*. Scribner, 1967.

[17] G. Kessler. An Overview of Cryptography. Online : http://www.garykessler.net/library/crypto.html, [Last accesed : 01/10/09].

[18] C. Leres. TCPDump/LibPCAP. Online : http://www.tcpdump.org/, [Last accessed 26/06/2009].

[19] T. Longstaff. Security of the internet. Online : http://www.cert.org/encyc_article/tocencyc.html. [Online : 06/09/2009].

[20] M. Mactaggart. Introduction to cryptography. Online : http://www.ibm.com/developerworks/library/s-crypt02.html, [Last accessed 02/06/2009].

[21] M. Marlinkspike. New Tricks For Defeating SSL In Practice. Online : http://www.thoughtcrime.org/software/sslstrip/, [Last accessed : 21/05/2009].

[22] J. Mathai. History of Cryptography and Secrecy Systems. Online : http://www.dsm.fordham.edu/~mathai/crypto.html, [Last accessed : 07/10/2009].

[23] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *Proceedings of the USENIX Winter 1993*, pages 1–2, 1992.

[24] P. Mohapatra. Public Key Cryptography. Online : http://www.acm.org/crossroads/xrds7-1/crypto.html, [Last accessed 16/10/2009].

[25] R. Morelli. Edgar Poe and Cryptography. Online : http://starbase.trincoll.edu/~crypto/historical/poe.html, [Last accessed : 06/08/2009].

[26] R. Needham and M. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978.

[27] NIST. National vunerability database. Online : http://nvd.nist.gov/, [Last accessed : 24/05/2009].

[28] D. C. Plummer. RFC 826 : An Ethernet Address Resolution Protocol. Online : http://www.faqs.org/rfcs/rfc826.html, [Last accessed : 14/10/2009].

[29] J. Postel. RFC 791: Internet Protocol. Online : http://www.faqs.org/rfcs/rfc791.html, [Last accessed : 20/10/2009].

[30] J. Postel. RFC 793 : Transmission Control Protocol. Online : http://www.faqs.org/rfcs/rfc793.html, [Last accessed : 14/10/2009].

[31] E. Rescorla. Diffie-Hellman Key Agreement Method. Online : http://www.ietf.org/rfc/rfc2631.txt, [Last accessed : 21/05/2009].

[32] E. Rescorla. SSLdump. Online : http://www.rtfm.com/ssldump/, [Last accessed : 05/05/2009].

[33] Robert G. Byrnes. Richard E. Silverman Daniel J. Barrett. *SSH, the secure shell, 2nd Edition.* O'Reilly, 2005.

[34] T. Rowland and E. Weisstein. Groups, mathsworld. Online : http://mathworld.wolfram.com/Group.html, [Last accessed 11/06/2009].

[35] B. Schneier. *Applied Cryptography.* Wiley and Sons, 1996.

[36] B. Schneier. Security in the real world: How to evaluate security. *Computer Security Journal, v 15*, pages 1–14, 1999.

[37] B. Schneier and N. Ferguson. *Practical Cryptography.* Wiley Publishing, 2003.

[38] K. Taylor. Cryptography of Ancient Times. Online : http://math.usask.ca/encryption/lessons/lesson00/page1.html, [Online : 11/10/2009].

[39] D. Teare. OSI Reference Model. Online : http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/introint.htm, [Last accessed : 11/09/2009].

[40] R. Treinen. Needham-Schroeder Public Key. Online : http://www.lsv.ens-cachan.fr/Software/spore/nspk.html, [Last accessed : 27/10/2009].

[41] H. Wang. Taxonomy of security considerations and software quality. *Commun. ACM*, 46(6):75–78, 2003.

[42] Webopedia. Steganography. Online : http://www.webopedia.com/TERM/S/steganography.html, [Last accessed : 21/09/2009].

[43] E. Weisstein. Congruence, Mathsworld. Online : http://mathworld.wolfram.com/Congruence.html, [Last accessed : 06/07/2009].

[44] E. Weisstein. Greatest Common Divsor, Mathsworld. Online : http://mathworld.wolfram.com/RelativelyPrime.html, [Last accessed : 09/07/2009].

[45] E. Weisstein. Relatively prime, Mathsworld. Online : http://mathworld.wolfram.com/RelativelyPrime.html, [Last accessed : 08/07/2009].

[46] Y. Zhang and L. Xiuying. Running-mode analysis of the security socket layer protocol. *SIGOPS Oper. Syst. Rev.*, 38(2):34–40, 2004.

Appendix A

Test scripts listing

This appendix provides source code listings for all the scripts used for testing Project Bellerophon.

PHP Script to generate a random file of a given byte size

Appendix A details the scripts used in Chapter 5 to produce timing results.

```
1  <?php
2    $numBytes = (int)$argv[1];
3    $fileName = (string)$argv[2];
4
5    if($numBytes == "")
6      die("You must specify a byte length\n");
7
8    if($numBytes <= 0 )
9      die("Invalid byte length\n");
10
11   if($fileName == "")
12     $fileName = "genOut";
13
14   $fh = fopen($fileName,"w");
15
```

```
16      while($numBytes != 0)
17      {
18        $numBytes--;
19        $x = chr(rand(33,122));
20        fwrite($fh,"$x"); }
21        exec("sudo cp $fileName /var/www/$filename");
22        exec("sudo rm $fileName"
23      );
24     ?>
```

Bash shell script used to determine the average time to decrypt a file using OpenSSL

```
1    #!/bin/bash total=0
2    iter=0
3    max=100
4    while [ "$iter" != "$max" ]
5    do
6      start=$(date +%s%N)
7      openssl aes-256-cbc -d -k $passphrase
8     -in /var/www/encryptedTest -out /var/www/plaintext
9      end=$(date +%s%N)
10     diff=$(($end - $start))
11          total=$((total + diff))
12          iter=$((iter +1))
13   done
14
15   echo "scale=2" > aTest
16   echo "$total / (1000000*$max)  " >> aTest
17   echo "quit" >> aTest
18   echo "A timing to test to compare execution
19         time for openSSL."
20   echo "This test was run $max times"
21   echo -n "On average openSSL completed the decrypt
22             in $total (measured in ms) " > out
23   bc -q aTest >> out
24   cat out rm aTest rm out
```

Bash shell script used to determine the average time to decrypt a file using Tshark

```
#!/bin/bash total=0
iter=0
max=100
while [ "$iter" != "$max" ]
do
  start=$(date +%s%N)
  tshark -q -o "ssl.desegment_ssl_records: TRUE"
   -o "ssl.desegment_ssl_application_data: TRUE"
   -o "ssl.keys_list:
  192.168.56.138,443,tcp,/home/samba/server.key"
   -o "ssl.debug_file: /home/samba/eDecrypt"
   -r  /home/samba/FinalCaptureSet/cap8000.pcap
   -R "tcp.port == 443 and ssl"
  end=$(date +%s%N)
  diff=$(($end - $start))
      total=$((total + diff))
      iter=$((iter +1))
done

echo "scale=2" > aTest
echo "$total / (1000000*$max)  " >> aTest
echo "quit" >> aTest
echo "A timing to test to compare execution time
     for TShark."
echo "This test was run $max times"
echo -n "On average Tshark completed the decrypt
         in $total (measured in ms) " > out
bc -q aTest >> out
cat out rm aTest rm out
```

Bash shell script used to determine the average time to decrypt a file using Project Bellerophon

```
1   #!/bin/bash total=0
2   iter=0
3   max=100
4   while [ "$iter" != "$max" ]
5   do
6     start=$(date +%s%N)
7     tshark -q -o "ssl.desegment_ssl_records: TRUE"
8     ./NewCap
9     end=$(date +%s%N)
10    diff=$(($end - $start))
11        total=$((total + diff))
12        iter=$((iter +1))
13  done
14
15  echo "scale=2" > aTest
16  echo "$total / (1000000*$max)  " >> aTest
17  echo "quit" >> aTest
18  echo "A timing to test to compare execution time
19        for Project Bellerophon."
20  echo "This test was run $max times"
21  echo -n "On average Project Bellerophon completed the
22           decrypt in $total (measured in ms) " > out
23  bc -q aTest >> out
24  cat out rm aTest rm out
```

Appendix B

Protocol Structs

This appendix discusses the structure of C++ structs used to parse packet headers produced by PCAP.

Ethernet Header Struct

```
1  struct ethernet
2  {
3    unsigned char ucDestination;
4    unsigned char ucSource;
5    int type_length;
6  };
```

IP Header Struct

```
1  struct ip_header
2  {
3    u_char  verLen;// Version and header length
4    u_char  service;// Type of service
5    u_short tLen;// Total length
6    u_short ident;// Identification
7    u_short flags;// Flags
8    u_char  ttl;// Time to live
9    u_char  protocol;// Protocol
10   u_short crc;// Header checksum
```

```
11    ip_address  saddr;// Source address
12    ip_address  daddr;// Destination address
13    u_int   padding;//Padding
14  };
```

TCP Header Struct

```
1   struct tcp_header
2   {
3     u_short sport; // source port
4     u_short dport; // destination port
5     tcp_seq seqNo;// sequence number
6     tcp_seq ackNo; // acknowledgement number
7     u_char  dOffSet;// data offset
8     u_short winSize;// window
9     u_short csum// checksum
10    u_short urgPointer;// urgent pointer
11  };
```

Appendix C

PCAP Capture Sample Code

This appendix details an example capture module using LibPCAP.

```
void packet_handler(u_char *buff, const
struct pcap_pkthdr* header, const u_char* packet)
{
      //Packet Analysis Occurs Here
}
void packetCapture::start()
{
  pcap_t *adhandle;
  char errbuf[PCAP_ERRBUF_SIZE];
  if(isLive)
  {
    struct bpf_program fp;
    bpf_u_int32 mask;
    bpf_u_int32 ipaddy;
    pcap_lookupnet(devName,&ipaddy,&mask,errbuf);

    adhandle =
    pcap_open_live(devName,5000,1,1000,errbuf);

    if(pcap_compile(adhandle,&fp,"tcp port 443",
               1,ipaddy) == -1)
    {
          cout<< "Filter compilation failure\n";
      exit(1);
```

```
25          }
26
27
28          if(pcap_setfilter(adhandle,&fp) == -1)
29          {
30                cout << "Failed to apply filter\n";
31             exit(1);
32          }
33       }
34       else
35       {
36          adhandle =
37          pcap_open_offline(devName,errbuf);
38          pcap_loop(adhandle, 5000000,
39          packet_handler, NULL);
40       }
41    }
```

Appendix D

CD Contents

The associated project CD contains all files used in order to produce and test the decryption application framework. The following can be found on said CD.

- PCAP files which be open using any traffic analyser which supports the PCAP format, such as Wireshark, Tshark and SSLDump
- Additonal worked examples of the negation phase of the SSL/TLS protocols
- Copies of LibPCAP and OpenSSL
- The Source Code used to produce the Application Framework
- R workspace containg the aquired results
- All images used within this document
- Copies of both associated published papers
- Scripts used to generate the results
- A copy of the associated development website
- A copy of this thesis in PDF and Lyx format. Including the Bibtex reference database.
- Project Poster