Secure Channel Establishment in Disadvantaged Networks: TLS Optimization Using Intercepting Proxies[1]

Sam McVeety, Roger Khazan, Joseph Cooley

Information Systems Technology Group

MIT Lincoln Laboratory

244 Wood Street, Lexington, MA 02420

email: {rkh, cooley}@ll.mit.edu

August 18, 2009

Abstract

Transport Layer Security (TLS) is a secure communication protocol that is used in many secure electronic applications. In order to establish a TLS connection, a client and server engage in a handshake, which usually involves the transmission of digital certificates. In this thesis we develop a practical speedup of TLS handshakes over bandwidth-constrained, high-latency (i.e. disadvantaged) links by reducing the communication overhead associated with the transmission of digital certificates. This speedup is achieved by deploying two specialized TLS proxies across such links. Working in tandem, one proxy will replace certificate data in packets being sent across the disadvantaged link with a short reference, while the proxy on the other side of the link will restore the certificate data in the packet. The certificate data will be supplied by local or remote caches. Our solution preserves the end-to-end security of TLS and is designed to be transparent to third-party applications, and will thus facilitate rapid deployment by removing the need to modify existing installations of TLS clients and TLS servers. Testing shows that this technique can reduce the overall bandwidth used during a handshake by over 50%, and can reduce the time required to establish a secure channel by over 40% across Iridium links.

The project report presented here is the MIT Master's of Engineering thesis document by Sam McVeety. Sam McVeety's Master's project and the thesis document were done in collaboration with and under the supervision of Dr. Roger Khazan and Mr. Joseph Cooley.

[1]This work is sponsored by the Department of Defense under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
18 AUG 2009		**00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Secure Channel Establishment in Disadvantaged Networks: TLS Optimization Using Intercepting Proxies	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Massachusetts Institute of Technology,Lincoln Laboratory,244 Wood Street,Lexington,MA,02420**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**72**	

Secure Channel Establishment in Disadvantaged Networks: TLS Optimization Using Intercepting Proxies

by

Sam McVeety

B.S., Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2009

Author .....
Department of Electrical Engineering and Computer Science
August 19, 2009

Certified by .....
Dr. Roger Khazan
Research Scientist
MIT Lincoln Laboratory
Thesis Supervisor

Certified by .....
Joe Cooley
Research Scientist
MIT Lincoln Laboratory
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Secure Channel Establishment in Disadvantaged Networks: TLS Optimization Using Intercepting Proxies

by

Sam McVeety

Submitted to the Department of Electrical Engineering and Computer Science
on August 19, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

Transport Layer Security (TLS) is a secure communication protocol that is used in many secure electronic applications. In order to establish a TLS connection, a client and server engage in a handshake, which usually involves the transmission of digital certificates. In this thesis we develop a practical speedup of TLS handshakes over bandwidth-constrained, high-latency (i.e. disadvantaged) links by reducing the communication overhead associated with the transmission of digital certificates. This speedup is achieved by deploying two specialized TLS proxies across such links. Working in tandem, one proxy will replace certificate data in packets being sent across the disadvantaged link with a short reference, while the proxy on the other side of the link will restore the certificate data in the packet. The certificate data will be supplied by local or remote caches. Our solution preserves the end-to-end security of TLS and is designed to be transparent to third-party applications, and will thus facilitate rapid deployment by removing the need to modify existing installations of TLS clients and TLS servers. Testing shows that this technique can reduce the overall bandwidth used during a handshake by over 50%, and can reduce the time required to establish a secure channel by over 40% across Iridium links.

Thesis Supervisor: Dr. Roger Khazan
Title: Research Scientist
MIT Lincoln Laboratory

Thesis Supervisor: Joe Cooley
Title: Research Scientist
MIT Lincoln Laboratory

Acknowledgments

I would like to thank MIT Lincoln Laboratory for its generous support of this project, and Roger Khazan for his advice and guidance. I would also like to thank Joseph Cooley for his helpful suggestions and sage wisdom toward building the prototype, as well as his general knowledge of the TLS network stack.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We begin with a problem statement that describes the scope of the project. Next, we outline the various concepts in security and information technology that our project builds upon and list the open-source tools that aided us in our research.

1.1 Problem Statement

Transport Layer Security (TLS) encryption [9] is a means of establishing and using a secure communication link between two endpoints via a mutually understood cipher. For the purposes of this project, a fundamental building block of TLS is the digital certificate, which consists of a public key that is signed by a trusted third party (*certificate authority*) allowing the validity of the certificate to be verified. As is standard in public key encryption, the owner of a certificate also knows the private key that corresponds to the public key in the certificate; this allows the owner to digitally sign messages in a way that can be verified by any party possessing the public key.

Creating a TLS connection between a server and a client begins with a *handshake* protocol, toward the end of authenticating each other and computing an authentic, shared master secret. During the handshake protocol, the server and client exchange digital certificates in the clear, along with auxiliary information. Certificates are large blocks of data, which (depending on the length of the *session*) often comprise a large part of the total data communicated over the lifetime of the secure connection. For example, a certificate can represent over 50% of the bandwidth used in a given handshake. The TLS protocol dictates that certificates be transmitted whenever a new session is begun. However, these certificates often have very long lifetimes (1-2 years), making their exchange at the start of *every* TLS handshake a redundant exercise (as this could happen every five minutes).

Even if the certificate size is small compared to the aggregate data exchange over the session, it still impacts the speed with which the connection can be established. If we could eliminate the overhead of certificate exchange for a large number of TLS handshakes, we conjecture that the

overall speedup would be non-trivial. Matthew Low's summer project at MIT Lincoln Laboratory, which regards certificate caching, supports this hypothesis [1]. It is highly desirable to reduce this response time as much as possible, even if the cost of transmitting the certificate can be amortized to a negligible cost over the lifetime of the connection, because this corresponds to the delay before any content reaches the client. Whatever the type of content, the user cannot continue work (viewing a web page, filling out a web form) until the handshake is complete.

We note that the problem of certificate size will continue to be relevant even as technology improves. Wireless bandwidth continues to be an issue for mobile devices, and as computational power grows (and correspondingly, our ability to break encryption schemes of a given strength) this necessitates the use of larger certificates, such that this issue will continue to be relevant in the coming years. Additionally, we acknowledge the fact that TLS supports the reuse of session IDs, allowing connection resumption within a given amount of time. However, there are a number of problems with this approach, enumerated by Schacham and Boneh [15]. These problems include dependence on server-maintained state, as well as the prevalence of obsolete TLS implementations which do not support session resumption.

Our solution uses a set of proxies to cache certificates on the downstream side of a disadvantaged link. This allows the proxy on the upstream side to compress the certificate(s) in future handshakes by replacing them with small references, because the downstream proxy will intercept the message and insert the full certificate(s) on its side of the link. It is possible to deploy these proxies in a way that is transparent to the two endpoints, such that TLS security is necessarily maintained between the endpoints.

Proxy transparency obviates the need of either client or server to support an extension to the TLS protocol, because all of the custom operations take place within the scope of the proxies. Thus, we conjecture that this solution will be robust in practice, as any IP link can be modified to support the proxied setup without altering third party code on either the client or server. That being said, there is a limit to what we can accomplish using only the intercepting proxies, without modifying the client or server. Accordingly, we will also look at the possible benefits that can be derived from altering the client TLS implementation (but not the server), as this approach is still practical on a variety of platforms, including mobile phones.

1.2 Technical Underpinnings

We briefly review the major specifications that our project depends on. Citations are provided for further information.

TLS relies on other protocols to transport its messages between endpoints. In this section we

[1] Matthew Low (with Joseph Cooley and Roger Khazan), A Communication Efficient TLS Extension, MIT Lincoln Laboratory Summer Project, 2008

Table 1.1: IP Packet (Header + Data)

Bits	0-3	4-7	8-15	16-18	19-31
0	Version	Header Length	Service Type	Total Length	
32	Identification			Flags	Fragment Offset
64	Time to Live		Protocol	Header Checksum	
96	Source Address				
128	Destination Address				
160	Options (Optional)				
160/192+	Data (e.g. TCP)				

review them, starting with the IP protocol. Once we have outlined this *network stack*, we provide further background on the TLS protocol itself. Additionally, we include specific definitions of other entities which are essential to this project, in particular, the proxies that we use.

1.2.1 IP Layer

The Internet Protocol (IP) [12] is a connectionless protocol designed to allow two systems to communicate over a computer network. For our purposes, all relevant Internet traffic is routed over IP. This is a general purpose protocol which, given an address (e.g. 192.168.10.1) and a payload, attempts to deliver a given *packet* of data from one computer to another. IP provides no guarantees about delivery, save for a checksum which allows for rudimentary corruption detection. Within IP, there are fields denoting the entire length of the packet, the source and destination addresses, as well as a checksum over the contents of the packet.

The total length field in an IP packet is the only length field in the packet with a constant offset from the beginning. We will use this fact when parsing packets.

Table 1.1 shows the byte arrangement in the IP header, attached to a payload containing data (e.g. a TCP packet.)

1.2.2 TCP Layer

The Transport Control Protocol (TCP) [13] is a communication protocol over which data delivery is accomplished in a FIFO, gap-free manner. TCP is a connection-based protocol: it begins with a three-part handshake that establishes a connection between two computers. During this handshake, each computer randomly generates a SYN number, which corresponds to the starting point of each computer's respective data stream. Thereafter, all packets that the computers send are tagged with the appropriate offset from the SYN number (called sequence numbers) such that the other computer can determine the intended data ordering. The receiving computer then acknowledges the receipt of packets up to a given sequence number with ACK messages. Table 1.2 shows the byte order of a TCP packet header and data payload.

These sequence numbers are particularly important for building a proxy system in the presence

Table 1.2: TCP Packet (Header + Data)

Bits	0-3	4-7	8-15	16-23	24-31
0	Source Port			Destination Port	
32	Sequence Number				
64	Acknowledgement Number				
96	Data Offset (Header Length)	Reserved	Flags	Window Size	
128	Checksum			Urgent Pointer	
160	Options (Optional)				
160/192+	Data (e.g. TLS)				

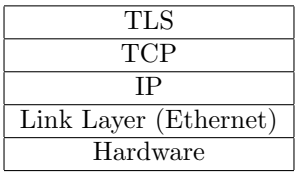

Figure 1-1: Network Stack.

of TCP, as they are the only way to correlate packets with each other and thus with the appropriate TCP connection. Accordingly, the proxy system specially indexes the SEQ and ACK numbers of packets so that it can quickly identify consecutive packets in a given stream.

1.2.3 Transport Layer Security

Transport Layer Security (TLS) [9] is the newest iteration of a ubiquitous protocol which allows two computers to establish a secure channel between them. Once the channel is established, all data transmitted across it is guaranteed to be authenticated and confidential. Connections begin with a handshake, during which certificates and other information are exchanged in order to create a mutual secret by which the computers can securely communicate. For our purposes, the TLS protocol runs on top of TCP, as shown in Figure 1-1, such that a TCP connection must first be established, before a TLS handshake can occur.

We define a *round* as a one-way transmission of information, after which the transmitting party must wait for a response from the receiver before continuing. The TLS handshake consists of two main tiers of granularity, namely, records and messages [9]. Over the course of a handshake, a record acts as a container for one or more messages. Messages correspond to a specific piece of information, such as a Certificate or a ClientHello message. Differing implementations of TLS alternately enforce a one-to-one or one-to-many correspondence between records and messages.

1.2.3.1 Records

TLS attempts to send all the information of a given round as a single packet, which may be fragmented by a lower level protocol. Each TLS packet contains one or more records, which correspond to the state that a given TLS connection is in (initializing, established, terminating, etc.). A com-

Table 1.3: TLS Record Types

Hex	Code	Record
0x14	20	ChangeCipherSpec
0x15	21	Alert
0x16	22	Handshake
0x17	23	Application

Table 1.4: TLS Record

Bytes	0	1	2	3	4
0-4	Record Type	Major Version	Minor Version	Record Length	
4-7	Record Data (TLS Message(s))				
...	Record Data (TLS Message(s))				

plete list of TLS record types is shown in Table 1.3. TLS Records have a small header containing the version information as well as the record type and length. This is shown in Table 1.4.

In this project, we will mainly be dealing with handshake records, as these are the records which contain messages related to digital certificates.

1.2.3.2 Messages

Within each record, there is at least one message, which constitutes a given action related to the connection. A list of TLS message types and their corresponding codes is shown in Table 1.5. Messages contain a very small header, which simply contains a message type and a 3-byte length. This is illustrated in Table 1.6. Of primary interest to us is the Certificate message, which communicates a digital certificate between parties. We will also look at the ClientHello message, as we attempt to optimize our system. The ClientHello message supports a number of extensions, which we will examine in more detail in Chapter 2.

Table 1.5: TLS Message Types

Code	Message
0	HelloRequest
1	ClientHello
2	ServerHello
11	Certificate
12	ServerKeyExchange
13	CertificateRequest
14	ServerHelloDone
15	CertificateVerify
16	ClientKeyExchange
20	Finished

Table 1.6: TLS Message

Bytes	0	1	2	3
0-3	Message Type	Data Length		
4-7	Message Data			
...	Message Data			

1.2.3.3 Digital Certificates

Public Keys and Secure Communications Most modern asymmetric public key encryption implementations use the RSA algorithm [14]. RSA encryption relies on two corresponding pieces of information - a public key and a private key, which can alternately be used to compute and invert a trapdoor function. These keys are used as an asymmetric means of encryption, where Alice can securely send information to Bob by encrypting data using Bob's public key. Bob can then decrypt the data using his private key.

Digital Signatures Digital signatures allow the owner of a private key to *digitally sign* information in a way that cannot easily be forged. Anyone possessing the corresponding public key can then verify that the information was signed by the private key in question [10].

Note that the ability to create a digital signature is not a proof of identity, per se, but rather a proof of possessing a given private key. In order to achieve some measure of authentication through digital signatures, a trusted signing *authority* (i.e. Verisign), which has a widely distributed public signing key, can verify Alice's credentials and sign Alice's public key, effectively indicating that Verisign believes the owner of the corresponding private key to be Alice.

Certificates A *certificate* packages a public key, identifying information, and a digital signature into a single entity, meant for distribution; the signature is that of a trusted authority, indicating that the authority believes that the identifying information corresponds to the owner of the private key.

Take the following example. Say that all computers on Earth ship with the root certificate for Certificate Authority C. Accordingly, if C signs Alice's certificate A, then Alice can transmit A to any computer that has C, which can then verify that C's signature on A is valid. If the user trusts the authority, they should also believe that Alice is who she says she is.

Additionally, if the user also trusts Alice to verify the credentials of others, then Alice can now sign Bob's certificate B, which can then be verified by possessing both C and A. In this way, authentication can be *chained* from a single root trusted authority through trusted intermediaries.

x.509 Certificates The x.509 format [11] is a standard way of formatting certificate information, such that all entities involved can easily verify the owner of a given certificate and perform actions

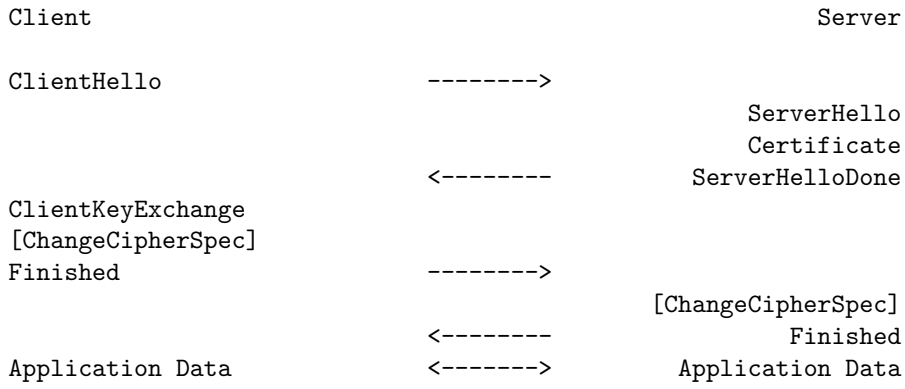

Figure 1-2: A Server-Only TLS Handshake.

with its public key.

In practice, some of the details are a bit archaic, which is why we offload most of this processing to an openssl library [4], which is designed to work with differing certificate formats.

1.2.3.4 TLS Handshakes

A TLS connection begins with a handshake protocol, during which information is exchanged between client and server in an attempt to authenticate each other and create a secure channel.

There are two types of TLS handshakes, each of which requires that the server authenticate itself to the client. One additionally requires client authentication; the other does not. The server-only handshake can be seen in Figure 1-2, where each line corresponds to a different TLS message. This project supports both types of handshake, though we begin by focusing on the server-only authentication handshake. Once the server-only authentication case is covered, we continue to the server-client authentication handshake.

TLS uses a combination of random numbers to create a shared master secret between client and server. These random numbers are session-specific, while certificates have a much longer lifetime. Accordingly, it is possible to reuse the parts of the handshake that can be directly computed from a certificate, over the course of several handshakes. The rest of the information cannot be reused.

Below, we briefly outline the different TLS message types used in the server-only handshake, and their effect on the handshake.

ClientHello In order to initiate a handshake, the client sends a ClientHello message to the server. The message lists various compatibility information about the client, including supported ciphers (such as RSA) and its TLS protocol version. The client also generates a random number and includes this in the message.

ServerHello The server responds to the client with its selection of protocol version and cipher, as well as its own random number.

Certificate The server sends its certificate in the same round as the ServerHello message.

ServerHelloDone This message simply marks the end of the round for the server, prompting a response from the client.

ClientKeyExchange In an RSA-based handshake, the client creates and transmits a PreMasterSecret in a ClientKeyExchange message, encrypted using the server's RSA key. Using the PreMasterSecret and the random numbers from the ServerHello and ClientHello messages, both client and server calculate a shared master secret. The rest of the exchange will be encrypted according to this secret.

ChangeCipherSpec Though technically not a handshake record, the ChangeCipherSpec record appears in the same round as the final handshake negotiations. It simply states that all transmissions from now on will be encrypted.

Finished The Finished message is sent by both client and server, and is the first message that is encrypted using their shared master secret. The message contains a message authentication code (MAC) over the entire handshake, to ensure that both client and server have been receiving messages with total fidelity. If any of the handshake messages have been altered in transit, this step will fail, and the connection will be torn down.

The MAC in the Finished message means that one cannot selectively alter pieces of the handshake in transit without being detected, and causing the endpoints to abort the connection. Therefore from the points of view of the client and the server, our proxies will have to be transparent in their processing of the TLS handshake.

1.2.3.5 Server-Client Authentication Specific Messages

In server-client authentication handshakes, additional message types are used to authenticate the client to the server. A diagram of the overall handshake is shown in Figure 1-3.

CertificateRequest The CertificateRequest message simply indicates that a server requires client authentication.

CertificateVerify The CertificateVerify message is the crucial step in authenticating the client to the server. The client signs all handshake messages that it has received, and transmits these to

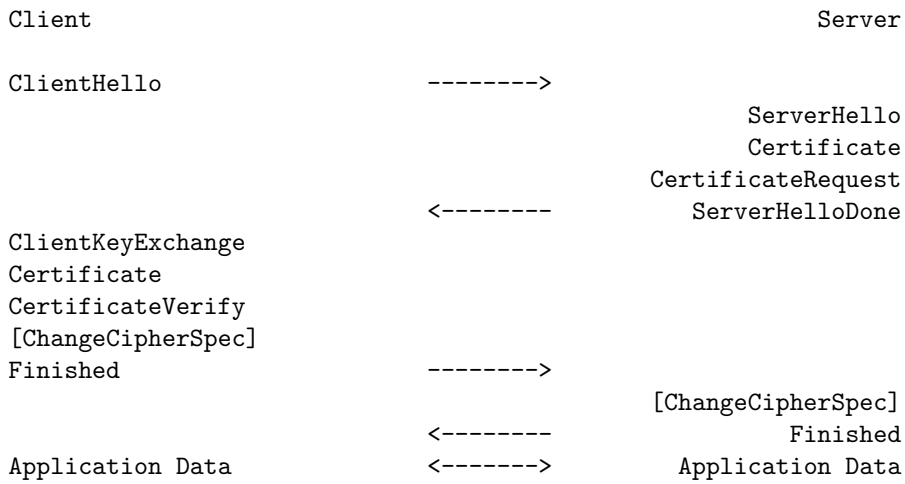

Figure 1-3: A Server-Client-Authenticating TLS Handshake.

the server. Per our earlier discussion of digital signatures, the client will only be able to create a valid signature if it possesses the private key corresponding to the certificate that it sends.

Since the CerticateVerify message requires the client's private key, a proxy cannot sign a message on behalf of the client.

1.2.4 Intercepting Proxies

This project will make heavy use of intercepting proxies, which we define as an intercepting process operating at the IP level, through which all traffic passes.

Traffic is directed through the proxy within the network stack, so that it is independent of the presence of other third-party applications. The proxies do not necessarily have to reside on a separate box from the client or server – they can run locally, or be simulated at the OS level if need be.

Our proxies make small modifications to the packets that they intercept. These modifications include certificate compression as well as tagging a packet with additional information that it did not originally contain.

Additionally, we require that our proxies be *transparent*, that is, that the endpoints cannot explicitly detect the modifications that the proxies make. In order to accomplish this, we use proxies in pairs, such that whenever the first proxy makes a modification to a packet, the second proxy undoes this modification before forwarding the packet. Using this technique, the proxies can communicate with each other without compromising the end-to-end protocol between the connection endpoints.

1.2.5 Disadvantaged Links

We examine the efficacy of modifying the TLS handshake protocol over disadvantaged links that feature constraints regarding latency and/or bandwidth. Given the correspondence between encryption strength and certificate size, the bandwidth utilization of a given connection quickly becomes a concern as the size of the certificate scales, as compression can make the difference between sending one packet or many. We accomplish this compression by caching certificates on each end of the disadvantaged link. For high-latency, low-bandwidth connections, we will examine the effect that smaller messages have on the speed of the handshake.

1.2.5.1 Mobile Phones

As a case study of disadvantaged links, we briefly discuss mobile phones. As Internet-capable phones become increasingly popular, the problem of establishing fast and secure connections for mobile phones is a very important one. We note that our solution is particularly relevant in this case, given that the available bandwidth, even over 3G networks, is small, and the set of secure websites that are used is likely to be small, such that many users will be requesting the same server certificates repeatedly.

Furthermore, even in future high-bandwidth networks, carriers will still seek to reduce their cost-per-bit, as well as to increase the number of concurrent users on their network. Reducing bandwidth usage in TLS handshakes offers an improvement in both of these categories.

In this case, the disadvantaged link exists between the phone and the cell tower, necessitating that certificate data be cached locally, on the phone itself. Since our solution is entirely transparent to third-party applications, the proxy can be readily adapted to any phone that supports TLS.

This is illustrated in Figure 1-4, with Proxy C and Proxy S acting as client-side and server-side proxies, respectively.

1.2.6 Supporting Software

We relied on the following tools and libraries in order to prototype and develop this project. For each tool, we will cover its basic functionality and then explain the ways in which it was particularly useful for this project.

1.2.6.1 Wireshark

The successor to the Ethereal project, Wireshark is a packet capture tool which allows us to capture and inspect live network traffic, including traffic over TLS [6].

We use Wireshark in all phases of development. First, we use it to examine the structure of TLS handshake packets, and ensure that we are parsing them correctly. After we start to alter packets,

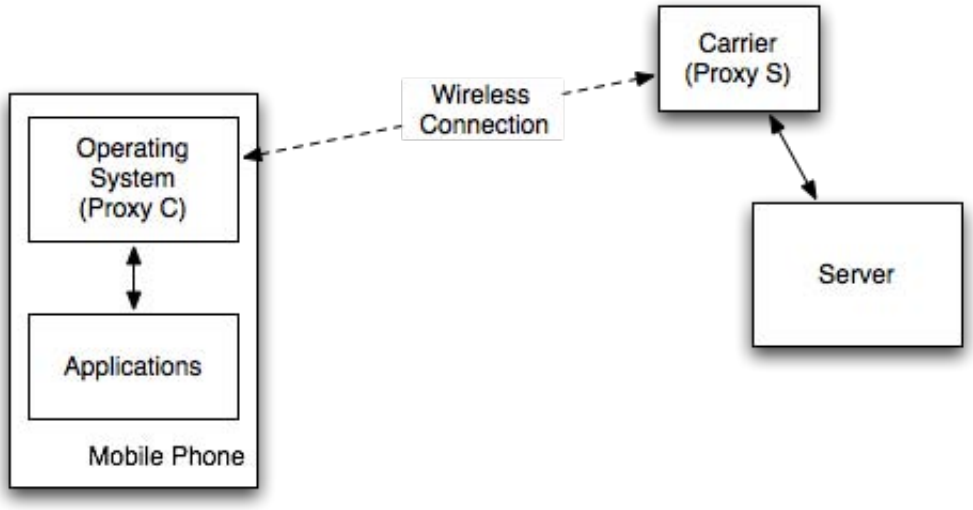

Figure 1-4: Proxies Deployed for a Mobile Phone.

we use it to look at the changes and ensure that they appear on actual network devices as they are supposed to.

Though somewhat excessive for this task, Wireshark is also a useful tool for ensuring that our test machines are configured with the correct routes, such that packets are passing through the appropriate gateways.

The Wireshark source code also provided inspiration for our methods of parsing TLS traffic.

1.2.6.2 Openssl

Openssl is an all-purpose cryptography project that provides a multitude of libraries and utilities for creating and maintaining applications which use cryptography [4].

The software provided by openssl proved itself useful in many different ways throughout the project.

Certificate Generation Openssl allowed us to create test certificates to use in TLS transactions. Rather than turning to a signing authority, we created our own trusted authority using openssl, and signed our own certificates with this authority.

s_client and s_server Two of the command line programs that are included with openssl allow for the simulation of a rudimentary TLS client and server. Running `s_server` deploys an extremely lightweight TLS server with a slew of user-configurable parameters, including the supported encryp-

tion algorithms and optional client authentication. `s_client` functions as a simple TLS client which can connect to an arbitrary TLS server. Both applications include debugging and logging tools.

These programs allowed us to test different parameters of the handshake in a standard and repeatable way, and check that our proxies' meddling is indeed transparent to the endpoints.

x.509 Handling Openssl provides an extensive library for handling x.509 certificates, which allows one to manipulate all aspects of the certificate.

We use this to parse certificates in a unified way, since a variety of subtly different certificate implementations exist on the Internet today. Rather than handle each certificate on a case-by-case basis, we let openssl perform this logic for us.

1.2.6.3 SQLite

SQLite is a lightweight transactional database which is both fast and reliable [2].

SQLite allows us to rapidly prototype any classes which require data storage, as a lightweight framework for reading and writing local data. This precludes the need to develop our own data format for storing certificates and state information.

Additionally, SQLite supports *triggers*, which allow us to conditionally run operations when certain events occur. We can use this functionality to periodically update or purge the records in our database, to ensure longevity of the system.

1.2.6.4 Netfilter

Netfilter is a set of Linux kernel modules which allow one to connect directly into the network stack at the IP layer [3]. Netfilter code is written using a series of *hooks*, which specify a point at which to capture upstream or downstream traffic as it interacts with the rest of the system.

Netfilter is ideal for our purposes, as it allows us to inspect and alter IP packets before they reach any third party applications on the system. We use a queuing mechanism provided by netfilter to capture packets in kernelspace, hand them off to userspace for processing, and then return them to kernelspace for transmission. The processing that takes place in userspace is the main logic developed in this project.

The data flow for netfilter can be seen in Figure 1-5.

1.2.6.5 getopt

`getopt` is a basic library for parsing command-line options that are passed to a given program [1].

In the project, we used this library to allow us to easily pass command-line parameters to our program in a clean and efficient way. This meant that we didn't have to write any specialized code for command-line parsing, and could quickly add additional options when necessary.

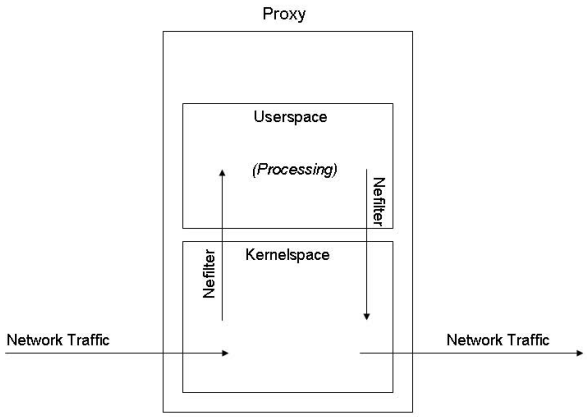

Figure 1-5: Data Flow in a Netfilter System.

1.3 Roadmap

Having outlined the problem that we wish to solve and surveyed the relevant tools, we now seek to more specifically design how we will be using those tools to create a functional system. Since we are dealing with a secure communication protocol, it is important that the design be robust. After outlining the design, we delve into some of the more nuanced aspects of the actual implementation. Having built a working system, we run a series of experiments to test its efficacy. Finally, we analyze these results and discuss the possible direction of future work.

Chapter 2

System Design

In order to accomplish data compression and expansion, we will look in detail at each step in the processing of packets. Beyond that, we will look at the different cases for TLS handshakes that we will be modifying, and the different pieces of information that we can bootstrap upon.

2.1 Packet Processing Overview

We will be using netfilter [3] to intercept packets as they pass through the intercepting proxy. Given a packet, we must first identify whether it is relevant, and then process it accordingly. The processing should take place in discrete steps, so as to increase the clarity of the code and speed of debugging.

1. Identify packet as a TLS handshake record;
2. Detect certificates and other relevant fields;
3. Compress/expand certificates; alter the packet;
4. Forward the resulting packet.

2.1.1 Capturing and Queuing Packets

As described in Section 1.2.6.4, netfilter allows us to process packets in the following way. The network hooks operate in kernel space, where we can intercept entire packets. However, in order to use the auxiliary libraries that we want, such as sqlite, openssl, etc., we need to process the packets in userspace. To allow for this functionality, we use the netfilter *queuing* mechanism, which intercepts packets and then sends them to queue processing in userspace, which then hands the packets back to the kernel when it is done with them.

We queue packets simply, by differentiating only between packets headed in one direction versus the other. These are placed in separate queues according to their arrival order and handed off to userspace, keeping the netfilter code lightweight.

2.1.2 Altering Packets

When netfilter intercepts a packet, it is given to us in the form of a continuous buffer. Accordingly, it must be returned in a similar form. Since many of our processing techniques alter the length of the packet, we must copy the data into an appropriately sized buffer.

Beyond using a new buffer, we also have to mind the length fields defined in various protocol headers. All of these must be changed to agree with the altered data.

2.1.3 Generating Packets

When we are reconstructing compressed certificates, it may become necessary to dynamically generate entire packets and then inject them into the network. This is somewhat trickier than altering queued packets, as the netfilter functionality cannot be used to generate new packets. Instead, we start by copying an existing packet, and alter the copied packet's header to properly place it in the TCP sequence. Then we can use a network library to transmit the packet.

2.2 Performance, Correctness, and Security

We examine the correctness of our system with regard to its required functionality, casting an eye toward performance bounds. Additionally, we show that the system retains the security of TLS.

2.2.1 Consistent Proxy State

Taking the requirements from Section 1.1 into account, it is clear that the server-side proxy must have at least some knowledge of the state of the client-side proxy in order to decide whether or not to compress the certificate(s). Ideally, the server-side proxy will always know if the client-side proxy has a cached copy of a given certificate, and will keep this information up-to-date. With a few further assumptions, it turns out that we can indeed achieve this degree of awareness in the server-side proxy, and design our communication protocol accordingly. Abstractly, these assumptions are:

1. Given a collection of server certificates, it is possible to identify a server certificate uniquely, given only the server domain name and the expectation that the certificate is valid for the current date;

2. The data transmitted by the TLS client in its ClientHello message uniquely identifies the domain name of the server.

2.2.2 Communicating Proxy State

In practice, the assumptions from Section 2.2.1 hold true, such that we can incorporate them into our design.

We have the ability to communicate some amount of information from client-side proxy to server-side proxy via the ClientHello message, before the Certificate message is sent. If we use this information to its fullest extent (detailed below), the server-side proxy can gather all relevant knowledge about the state of the client-side proxy at the cost of very few bits. To use this information, the server-side proxy keeps a persistent database of its record of the client-side proxy's state (i.e., what certificates the client-side proxy possesses).

What we propose is that the client-side proxy modify the ClientHello message to include a short list of certificates that it *anticipates* that the server will return. Clearly, the viability of such a system rests on making such predictions accurately. In our implementation, we find a solution to this problem that meets these requirements.

2.2.3 TCP Fragmentation

One problem that we encountered during implementation is the prevalence of TCP fragmentation over the course of the TLS handshake. This results from the fact that some TLS payloads exceed the pre-defined maximum segment size (MSS) for the TCP protocol, forcing them to be fragmented over several packets.

At the IP proxy level, there is no good way to tell whether the TCP payload of a packet is a continuation of the previous packet's payload or not. This is exemplified in Figure 2-1, where TLS record boundaries do not line up with TCP packet boundaries, and the length of each record is only reported at the beginning of the record itself. For example, if we received TCP Packet 2 out of order, the payload would begin in the middle of TLS Record 1, without any additional information to guide us.

The best solution to this is to ensure that the packets have all been received in order, first, and then to scan across their payloads to determine whether one TCP packet is the continuation of another. We would do this by starting with TCP Packet 1, and reading the length of TLS Record 1. Reading the TLS Record would cause us to wait for the next packet, and then to continue reading to the end of the record.

Hence, we can only begin our processing at the start of a packet that coincides with the start of a TLS record.

As an example of this mechanism at work, consider the following situation. Packets 17, 18, and 19 are consecutive segments of a TCP stream. A TLS record spans packets 17 and 18. Packet 19 contains a single TLS record. If the proxy received these packets out-of-order, such they arrived as

<table>
<tr><td>IP Packet 1</td><td colspan="3">IP Packet 2</td></tr>
<tr><td>TCP Packet 1</td><td colspan="3">TCP Packet 2</td></tr>
<tr><td colspan="2">TLS Record 1</td><td>TLS Record 2</td><td>TLS Record 3</td></tr>
<tr><td>TLS Message 1</td><td>TLS Message 2</td><td>TLS Message 3</td><td>TLS Message 4</td></tr>
</table>

Figure 2-1: TLS Data Alignment.

17, 19, and 18, they would be processed in the following way.

1. Proxy receives 17 and discovers its payload is incomplete; it is requeued.
2. Proxy receives 19 and discovers it is complete; it is forwarded.
3. Proxy receives 18 and cannot process it (as it begins in the middle of a TLS record); the contents are stored and the packet is dropped.
4. Proxy receives 17 (again) and combines it with 18 for processing; the resulting packet is forwarded.

2.2.4 Transmission Reliability Mechanisms

In order to create a robust system, we must incorporate resilience to failed packets, whether they are dropped or altered in transit.

A salient problem arises in the case of dropped packets, as we may be intentionally altering the TCP state markers. We develop the technique of *packet predication*, where all of our altered packets are sent, predicated on the receipt of one or more normal packets that are required for the transaction. In other words, all of the actions that the proxy takes can be expressed as, "When I receive packet A, do action B." This allows us to retain the delivery guarantees of TCP, as the failure of action B will necessarily cause a normal packet (in this case, A) to fail. This will cause the normal packet to be re-sent, triggering B again. The underlying TCP connection is a best-effort attempt to deliver these packets, so that our system's functionality cascades from that.

2.2.4.1 Failsafe

No system is perfect, and we acknowledge the possible existence of bugs in our proxy. In accordance with this, we have an additional failsafe mechanism, where both proxies keep track of the IP packets that they have already forwarded. If the proxy has never seen a given packet before, the normal control flow dictates the way in which it is processed. However, if the proxy has seen the packet before, it assumes that something has gone wrong, and short-circuits any processing that would normally take place, in favor of forwarding the unaltered packet. This allows the system as a whole to fail gracefully, and fall back to simply forwarding packets unaltered. This concept is illustrated in Figure 2-2.

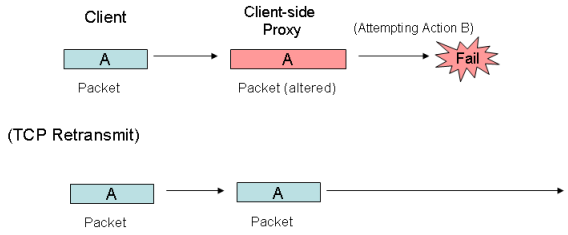

Figure 2-2: Transmission Failsafe.

2.2.5 Security

Since we are building our proxy system to coexist with the TLS protocol, we get to keep the guarantees of security from TLS "for free". Our interference in the transmission can be no more invasive than any man-in-the-middle attack, which the TLS protocol is designed to be resilient against. Put another way, our software is no more than a man-in-the-middle, and so it cannot compromise the security of a TLS connection any more than any other man-in-the-middle.

2.3 Server-Only Authentication

We first examine the case where a server authenticates itself to a client, and the two parties create a shared secret in order to establish a secure channel. Recall that this is shown in Figure 1-2.

2.3.1 Overview

The server is the origin of its certificate with regard to the handshake, and this certificate must somehow make its way to the client. We would like for this transmission not to be redundant, in that, if a client (or, rather, the client's proxy) has ever received the server's certificate, it will never need to receive that same certificate again. Essentially, what we want is the ability for a client-side proxy to be able to cache a certificate at its end of the disadvantaged link, such that all subsequent handshakes that contain the certificate are modified as follows; also illustrated in Figure 2-3:

1. Servers send unmodified TLS packets containing their certificates;
2. Server-side proxy compresses the certificates by replacing them with short references;
3. (Thus, Certificates do not traverse the (disadvantaged) link between proxies;)
4. Client-side proxy reconstructs the original TLS packet by replacing certificate references with cached certificates;

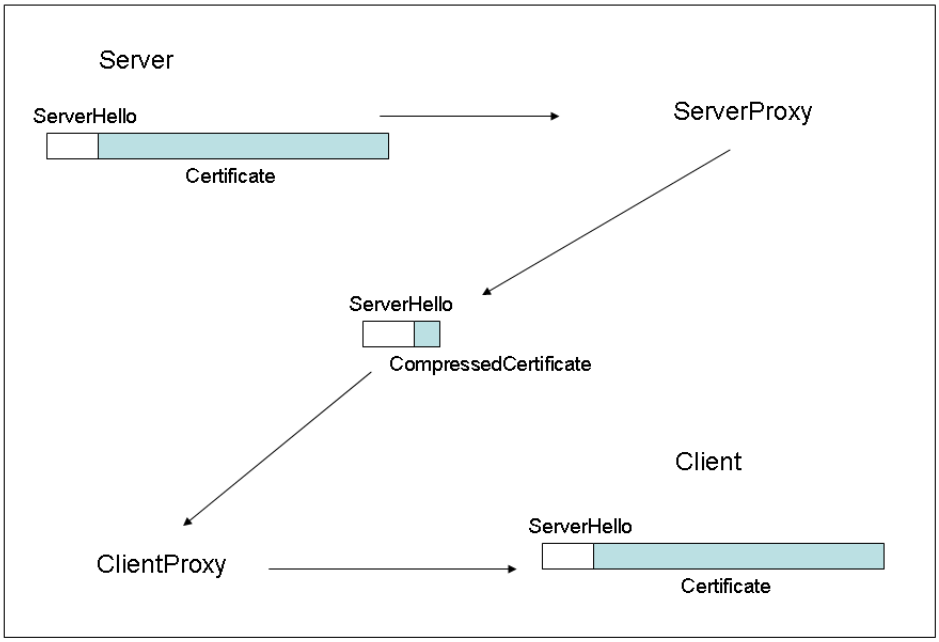

Figure 2-3: Certificate Compression Via Proxies.

5. Clients receive identical packets to the ones that the servers sent.

Given this as the basic idea of what we want to do, it is clear that we will have to include some amount of auxiliary information in the exchange. We can do much better than repeatedly sending the entire certificate, as the number of bits required to uniquely identify a given certificate is much smaller than the size of the certificate itself. We need a protocol for interproxy communication.

There are two principal approaches to the problem of keeping the proxies synchronized and communicating auxiliary information between them:

1. Piggyback on the existing TLS protocol between proxies, so as to preserve all of the guarantees of TLS while conveying the necessary auxiliary information;

2. Create a new protocol over TCP that is designed expressly for interproxy communication.

It should be clear that the first approach is probably the simpler one, while the second approach is more extensible. We will implement the first approach in this project, because it relies on the transmission guarantees of the TLS protocol and is simpler.

We briefly digress here to discuss the necessity that the proxy operations be transparent to the endpoints. At base, this is required by the TLS handshake, which performs a Message Authentication Code (MAC) step at the end of the handshake (in the Finished message, see 1.2.3.4) to ensure the

integrity of the handshake data. Here, an encrypted trapdoor function of all of the handshake data is calculated, and the client and server compare their values. If there is any mismatch, the connection is aborted. Accordingly, the data that reaches the client and server endpoints must be identical to the data that each of them transmits; the proxies cannot leave any fingerprints on the data.

2.3.2 Client Hello Extensions

The ClientHello message standard includes a mechanism for including additional information, called *extensions* [7]. Effectively, this format allows for a list of arbitrary values (prepended by *extension types*) to be tacked on to the end of a ClientHello message in a backwards compatible way. The receiving server can ignore extensions that it does not know how to handle.

In modern TLS implementations, ClientHello extensions are used for Server Name Indication as well as session resumption, and we in turn can use a ClientHello extension to transmit our own interproxy message within the confines of the TLS protocol.

For a given message, the extension list has the following format:

```
EXTENSIONS LENGTH (2 bytes)
   EXTENSION 1 TYPE (2 bytes)
   EXTENSION 1 LENGTH (2 bytes)
      EXTENSION 1 DATA
   EXTENSION 2 TYPE (2 bytes)
   EXTENSION 2 LENGTH (2 bytes)
      EXTENSION 2 DATA
   ...
```

2.3.2.1 Extension 1: Server Name Indication

The Server Name Indication (SNI) extension was created to allow the client to specify a domain name when initiating a TLS handshake with a given server [7]. This is a useful feature, because the IP and TCP headers can only specify an IP address, so that there was no way to disambiguate between multiple virtual domains before this extension.

This extension is additionally used by web browsers, since certificates are tied specifically to domain names rather than IP addresses; the latter can be transient. We assume that there is only one certificate for a given domain, and will use SNI in a similar fashion, to correlate certificates with a handshake at the ClientHello round. Note that without SNI, we would only have access to an IP address at the ClientHello round, which cannot reliably resolve to a certificate.

The SNI extension has the following format:

```
TYPE (2 bytes) // server_name = 0x0000
LENGTH (2 bytes)
        DATA
        LENGTH (2 bytes) // TYPE + SERVER
                TYPE (1 byte) // So far there is only one, 0x00 = host_name
                LENGTH (2 bytes)
                SERVER (n bytes)
```

The above is a practical summary of the original RFC definition, reprinted in Figure 2-4.

```
struct {
  NameType name_type;
  select (name_type) {
      case host_name: HostName;
  } name;
} ServerName;

enum {
  host_name(0), (255)
} NameType;

opaque HostName<1..2^16-1>;

struct {
  ServerName server_name_list<1..2^16-1>
} ServerNameList;
```

Figure 2-4: Server Name Extension Specification.

With SNI in mind, let us take an inventory of the tools available to us. At the beginning of a handshake, the client sends a ClientHello message to the server. This passes through both proxies, so that the client-side proxy now knows which server the client is connecting to, and thus (with reasonable confidence) which certificate the server is going to send. At this point, it can check its cache to see whether or not it has the certificate. If it does, the client-side proxy can indicate to the server-side proxy that it already has the certificate, and have the server-side proxy compress the certificate in the next round. This leads to our first interproxy message.

2.3.2.2 Extension 2: Certificate Possession Indication

Building on the existing ClientHello extension framework, we add a Certificate Possession Indication (CPI) extension to the ClientHello message. Put simply, the CPI extension contains a unique identifier(s) for one (or more) certificate(s), and effectively means, "I have this (these) certificate(s) in my cache."

Let's discuss the utility of the CPI extension. It appears to be just another ClientHello exten-

sion, so any TLS implementation that can read ClientHello extensions should be able to read this extension. Additionally, this means that TLS code should also fail gracefully if it doesn't know how to handle this extension – just like it would for any other ClientHello extension.

Together, the compatibility feature and graceful failure of this extension give us a few implementation options. We can use the extension in a transparent way, such that the client-side proxy adds the extension to a given ClientHello message, and then the server-side proxy strips the extension back out before it reaches the server (but after updating its client-side state) such that neither client nor server see the extension.

We could also implement this in the client TLS application itself (assuming the certificate cache was accessible from the client) without altering the server application, since the server will simply ignore the code. Along the way, the server-side proxy would read the extension and update its value of the client-side state.

The extension layout can be seen in Figure 2-5

```
TYPE (2 bytes) // certificate_possession = 0x0099
LENGTH (2 bytes)
        CERTIFICATE REFERENCE 1
        CERTIFICATE REFERENCE 2
        ...
```

Figure 2-5: ClientHello Extension: CPI.

2.3.3 TLS Message: Compressed Certificate

We now reach the stage where we must decide how to transmit compressed certificates between proxies. Again, we opt for a solution that is highly compatible with existing TLS code, by creating a new TLS message type: CompressedCertificate.

2.3.3.1 Certificate Compression

To compress a certificate, we need to correlate a unique identifier with the certificate, such that there is no ambiguity as to which certificate is compressed. Taking a cryptographic hash over the certificate data is a natural choice, as the probability of a collision is acceptably negligible.

The CompressedCertificate message is simply an array of compressed certificates. The message header indicates the size of the array, and no other metadata is given. Assuming certificates are compressed to 16 bytes (say, the 16 bytes of a MD5 hash) the message structure is illustrated in Table 2.1.

As in the case of the CPI extension, notice the flexibility of implementation that this design gives us. It is a valid TLS message format; any proxy code that we write which is designed to iterate through all TLS messages in a record can easily handle this, as it simply adds an additional message

Table 2.1: TLS Message: CompressedCertificate

Bytes	0	1	2	3
0-3	Message Type (0x99)	Data Length		
4-7	Compressed Certificate 1			
...	...			
...	...			
20-23	Compressed Certificate 2			
...	...			

type to the mix. As an aside, we also note that building certificate compression into a new TLS implementation would be much easier than if we had created a new format on our own.

In our implementation, we again opt for transparency, such that the server-side proxy will replace a Certificate message with a CompressedCertificate message, which the client-side proxy then replaces with a Certificate message. From the point of view of the client and server, the transaction proceeds normally. From the point of view of the proxies, the full certificate does not traverse the disadvantaged link.

2.3.4 Reliability Analysis

As far as the proxies are concerned, their sole purpose in a server-only authentication handshake is to ensure that the client has the desired server certificate, ideally without having to transmit the certificate over the disadvantaged link in its entirety. This comprises at most two rounds of the handshake, so there can be at most four proxies involved, though there will usually only be two. This can be seen in Figure 2-6, which shows the ClientHello and Certificate rounds of the handshake. The rounds pass through the proxies at points A, B, B', and A'.

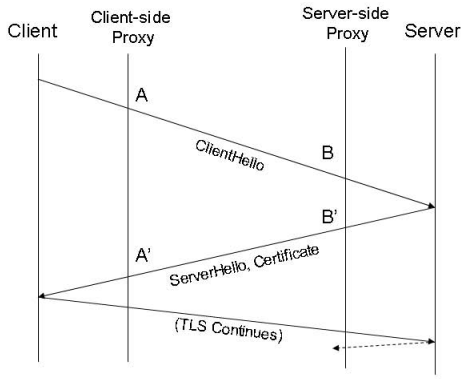

Figure 2-6: Server-Only Handshake: First Two Rounds.

For the purposes of our analysis, we assume that B and B' represent the same proxy, since we

are assuming a fast local connection between the server-side proxy and the server. We will, however, consider the case when the proxies at A and A' are not the same, since the location of the client could have changed significantly if the disadvantaged link has high latency, or there could be multiple links between client and server.

We will now analyze the different cases that can arise. We begin with the expected case, and then add possible extenuating circumstances, showing how the system handles these contingencies. The interesting cases are summarized in Table 2.2. Remember the failsafe mechanism of our system which disallows repeated attempts to send an altered packet: on the second try or later, the packet is forwarded without changes. Thus, the rest of the cases fall back to the original TLS handshake.

Table 2.2: Server-only Authentication Use Cases

Case	B = B'	A = A'	A ≠ A'	B resets	A resets	A has Z	A' has Z
0	X	X				X	X
1	X		X			X	X
2	X	X					
3	X	X			X		
4	X		X			X	
5	X	X		X			

2.3.4.1 Case 0: Two Proxies, Proxy A has Certificate Z

- By matching the SNI field in ClientHello with a certificate that it has in its cache, A indicates that it has Z in a modified ClientHello message;
- B notes that A has Z in its database, and forwards a compressed certificate, when it receives the certificate from the server;
- A receives the compressed certificate and expands it for the client.

This is by far the most likely case, where things go according to plan and the certificate is properly compressed and expanded.

2.3.4.2 Case 1: Three Proxies, Proxies A and A' have Certificate Z

- By matching the SNI field in ClientHello with a certificate, A indicates that it has Z in a modified ClientHello message;
- B notes that A has Z in its database, and forwards a compressed certificate, when it receives the certificate from the server;
- A' receives the compressed certificate and expands it for the client.

Notice that the compressed certificate is not client-side-proxy specific – any client-side proxy with the certificate in its cache can decompress the certificate. Practically, this means that even if a moving client (or the routing setup) causes A $\neq$ A' often, the overall system will quickly become saturated with common certificates, and function independently of whether A and A' are the same.

2.3.4.3 Case 2: Two Proxies, A doesn't have Z

- A fails to match the SNI field to a certificate, and sends a normal ClientHello;
- B forwards the full certificate message when it receives it from the server.

We must encounter this case at least once for the first-ever connection from a client-side proxy to a given server, unless we pre-populate our certificate database. Fortunately, it is no more costly than a normal handshake.

2.3.4.4 Case 3: Two Proxies, A claims to have Z, A loses cache

- By matching the SNI field in ClientHello with a certificate, A indicates that it has Z in a modified ClientHello message;
- B notes that A has Z in its database, and forwards a compressed certificate when it receives it from the server;
- A, having lost its cache, cannot decompress the packet, and drops it;
- The server retransmits the Certificate packet (via TCP guarantees) and B forwards the packet (unchanged) when it receives it from the server.

This case is also unlikely, but is an important consideration. Even if there is a mismatch between the recorded states of A and B, the handshake succeeds in at most one additional round, after the TCP retransmission timeout.

2.3.4.5 Case 4: Three Proxies, A has Z, A' does not; B forwards through A'

In this case, the client has moved behind a new proxy during the time that B is communicating with the Server. The result is similar to the previous case.

- By matching the SNI field in ClientHello with a certificate, A indicates that it has Z in a modified ClientHello message;
- B notes that A has Z in its database, and forwards a compressed certificate when it receives it from the server;
- A' cannot expand the certificate and drops the packet;

- The server retransmits the Certificate packet (via TCP guarantees) and B forwards the packet (unchanged) when it receives it from the server.

Again, this case is unlikely, and the handshake succeeds in at most one additional round, after the TCP retransmission timeout.

2.3.4.6 Case 5: Two Proxies, A has Z, B loses state

- By matching the SNI field in ClientHello with a certificate, A indicates that it has Z in a modified ClientHello message;
- B resets and loses all record of client-side proxy state;
- Without knowing A's state, B forwards the full certificate message when it receives it from the server.

This case is rather unlikely, as our proxy software will be robust against losing state. Also, the interval in question is very small, so it is very unlikely that B will reset in the roundtrip time from the server-side proxy to the server. Nevertheless, we would still fall back to the original handshake in this case.

2.4 Server-Client Authentication

The addition of client authentication makes the task of compressing certificates in an efficient way considerably more difficult. It is much more difficult to correlate the initial steps of a TLS handshake with a specific client certificate, because multiple clients can correspond to the same IP address, and the client certificate is not sent until the third round of the handshake. Thus, the server has no way of knowing a priori what certificate the client might try and send, so the server-side proxy does not know what part of its state to communicate to the client-side proxy. This setup can be seen in Figure 2-7, where several clients share a single proxy.

2.4.1 Overview

If the client-side proxy does not know the state of the server-side proxy, it cannot know whether or not to compress a given certificate. Compressing all certificates would likely lead to a high failure rate, so we need to think of a solution that takes other factors into account.

A naïve approach would attempt to identify some aspect of the ClientHello message in a one-to-one correspondence with a certificate; this approach quickly shows itself to prove fruitless. As seen in Figure 2-8, which shows the different rounds of a server-client authenticating handshake, the ClientCertificate does not appear until the third round of the handshake. This fact is built in to the

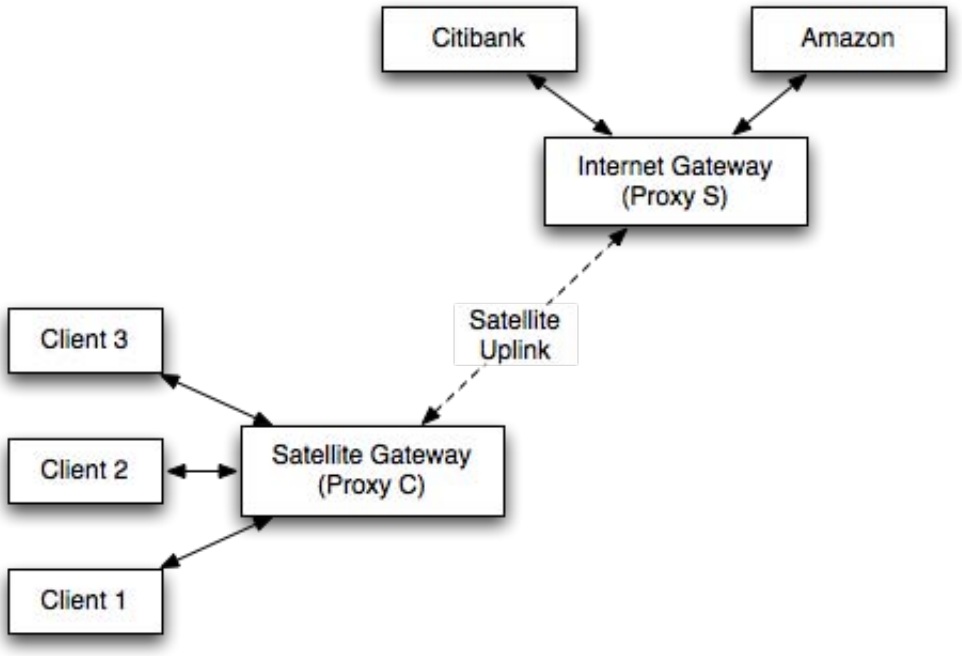

Figure 2-7: Proxies Deployed Across a Satellite Link.

TLS state machine itself, which does not request a certificate until it receives a CertificateRequest message.

Accordingly, the ClientHello message contains at most identifying information about the client's computer, and does not initiate any certificate selection on the part of the user until the third round. Thus, in the most basic case, several people sharing the same computer would break any possibility of establishing such a correspondence. Even if we could be assured of an injective correspondence between computers and users, the existence of NATs and dynamic address spaces indicate that it is not possible for the server to determine which certificate is about to be sent from the ClientHello message, or any transparent modification thereof.

This leaves us three possibilities for attempting to extend certificate compression to the server-client-authenticating version of the handshake.

2.4.1.1 Proxy Broadcast

It is possible for the server-side proxy to advertise its state by broadcasting some format of messages containing state information to the client-side proxy. However, this requires additional bandwidth, and could be very wasteful, depending on the number of certificates associated with a given address. Again, a priori, there is no way to know which state information will be useful, and if there are 1000 users behind a NAT with a single address, any bandwidth savings from certificate compression would quickly disappear. Granted, there are more clever ways to go about compressing such a list of certificates, perhaps by using a bloom filter [8] or other such construct. This approach will not

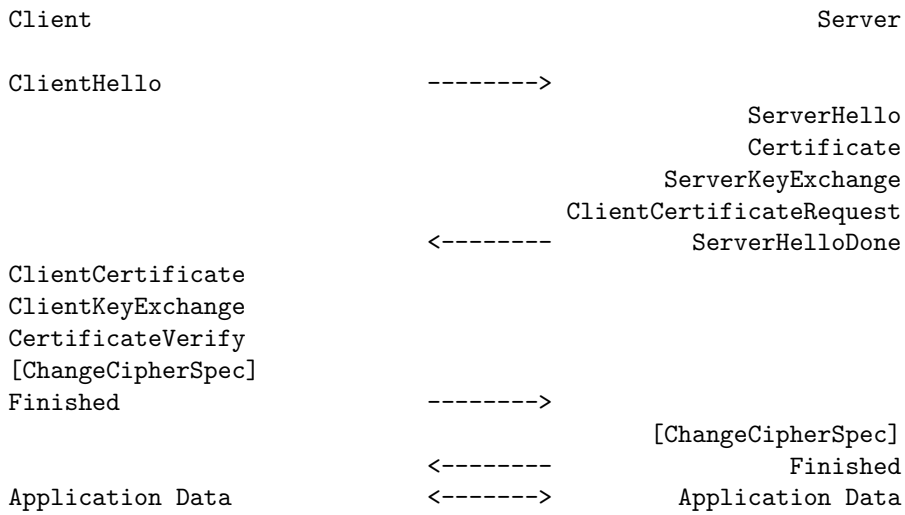

Figure 2-8: A Server-Client-Authenticating TLS Handshake.

be explored further in this project.

2.4.1.2 Persistent Client-side Proxy Statefulness

Given a reasonable guarantee of persistence for the client-side and server-side proxies, it is possible for the client-side proxy to keep track of the state of the server-side proxy by itself, without broadcasts from said proxy. In this scenario, whenever the client-side proxy forwards a certificate in full to the server-side proxy, it updates its internal record of the state of the server proxy. If it sees the same certificate again, it assumes that the server-side proxy has it cached, and forwards a unique identifier in place of the actual certificate in the second round.

If this assumption proves incorrect, we add an extra round to the transaction where the client-side proxy explicitly sends the full certificate. This functions similarly to the cases in Section 2.3.4 where the server-side proxy sends a compressed certificate to a client-side proxy that lacks the certificate. The packet is simply dropped, and the proxy transmits it in full when TCP forces a retransmission.

The chief drawback of this strategy is its dependence on consistency between the client-side and server-side proxies. Whenever there is an inconsistency, an extra round is necessary to remedy the state mismatch, effectively corresponding to a cache miss. If the expected number of cache misses is low (whenever the client-side proxy incorrectly assumes that the server-side proxy has a certificate) then the extra round adds a negligible expected cost to the overall exchange.

2.4.1.3 Protocol Modification

Finally, we can modify the problem of transmitting the client certificate to the server-side proxy in an essential way by removing the chief difficulty, namely, that the client does not select their certificate

until the third round of the handshake. Building upon the Fast Cert system [1], we can modify the TLS protocol itself to require certificate selection in the first round, such that the client-side proxy can explicitly state (also in the first round) which certificate it will attempt to send during the handshake.

This effectively combines the ideal cases of the previous sections. The client-side proxy would have perfect knowledge of the (relevant) state of the server-side proxy, as far as the certificates that it is going to transmit. Also, since the certificate is explicitly indicated by the user, there is no guessing, and no redundant state is transmitted. Since not all servers require server-client authentication, the client system would need to remember which servers did, and only request a client certificate for those sites.

This is the option which we chose to explore in our implementation.

2.4.2 Client Hello Extensions, Take 2

Our previous extension to the ClientHello message finds use here in our desire to have the client communicate to the server-side proxy which certificate it is about to send. For clarity, we will differentiate this from the previous extension, calling it Certificate Intention Indication (CII). The fact that this is a valid message extension is useful for several reasons:

- Both client and server can see the extension, and include it in their MAC calculations, even though the server will ignore the extension. This obviates the need to alter the MAC code in any way;
- Both proxies can now treat the extension as read-only, and thus have to perform very little processing on the packet.

Accordingly, we will alter our client TLS implementation (in this case, openssl) such that it supports an option that automatically includes a CII extension when given a client certificate. Given such a message, the implementation and case analysis of the server-client authentication handshake is no different than that of the server-only handshake.

[1] Matthew Low (with Joseph Cooley and Roger Khazan), A Communication Efficient TLS Extension, MIT Lincoln Laboratory Summer Project, 2008

Chapter 3

Software Implementation

Having outlined the system design, we will now delve into the lower level implementation, discussing specific algorithms and code snippets. Though our code is untested outside of the reference environment, it does not make use of any environment-specific features, and thus should be portable.

3.1 Architecture

3.1.1 Class Design

We model several different entities in our system as C++ classes.

3.1.1.1 Packet

Whenever our system receives or creates a packet, a Packet class is instantiated. The Packet class contains methods for reading and writing all relevant fields in the packet, including checksums and header values. Additionally, the Packet class contains specialized methods for processing TLS packets in order to compress or decompress certificates.

3.1.1.2 TcpStream

As we have mentioned before, we may have to combine several TCP packets in order to reconstruct a full TLS record. The TcpStream class is designed to let us do that, such that every Packet belongs to a TcpStream, and we can search for subsequent packets by querying the TcpStream object.

3.1.1.3 Certificate

The Certificate class contains all certificate-specific methods for data reading and compression.

3.1.1.4 Proxy

The Proxy class is a static class that provides access to the persistent data store for a given proxy. Proxy contains methods for looking up cached certificates, as well as storing new certificates.

3.1.2 Netfilter Modules

In order to use netfilter to alter packets, we need to set up a kernel module that sends the packets to userspace for alteration. This is done using the queuing mechanism [3] which then triggers a callback in userspace whenever a packet is queued. This process is explained in more detail in Section 3.2.2.

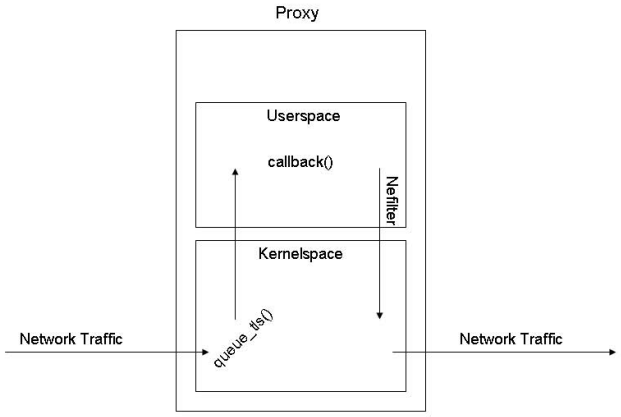

Figure 3-1: Data Flow in a Netfilter System.

3.2 Code Implementation

3.2.1 Environment Setup

Our development environment of choice is the Ubuntu Linux distribution, running as a virtual machine via VMWare's virtualization software [5]. This allows for rapid recovery when missteps in the kernel module cause a kernel panic or other such nuisance.

3.2.2 Control Flow

When using netfilter, all actions are triggered by the interception of a packet. At this point, we can choose how to handle the packet: whether to drop it, alter it, or simply to forward it. Such an

entry point in packet control is termed a *hook*. To implement a hook, we first instantiate it, and then specify where it should intercept packets, and how to handle them. This can be seen in the following code snippet:

```
static struct nf_hook_ops netfilter_ops_in;
netfilter_ops_in.hook = queue_tls; // Specify a handler
netfilter_ops_in.pf = AF_INET;
netfilter_ops_in.hooknum = NF_INET_PRE_ROUTING; // Intercept incoming packets
nf_register_hook(&netfilter_ops_in);
```

In this case, the specified handler is `queue_tls`, which is called whenever a packet is intercepted. We check the port number (4433 for openssl testing) of the packet, to determine if it is likely to be of interest to us. If so, we forward it to a queue in userspace, as shown in Figure 3-1 and the code below.

```
unsigned int queue_tls(unsigned int hooknum,
                       struct sk_buff *skb,
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff*))
{
  struct sk_buff *sock_buff = skb;
  struct iphdr *ip;
  struct tcphdr *tcp;
  ip = (struct iphdr*) sock_buff->data;
  tcp = (struct tcphdr*) (sock_buff->data + ip->ihl * 4);
  if (ntohs(tcp->source) == 4433 || ntohs(tcp->dest) == 4433) {
    return NF_QUEUE; // Queue to userspace
  } else {
    return NF_ACCEPT;
  }
}
```

If `tls_queue` returns `NF_QUEUE`, we pick up processing the packet in a separate application, which lives in userspace. Userspace processes are allowed to *subscribe* to a netfilter queue, using the following code:

```
  h = nfq_open(); // Open handle
  if (nfq_bind_pf(h, AF_INET) < 0) { // bind to AF_INET
```

```
    fprintf(stderr, "error during nfq_bind_pf()\n");
    exit(1);
  }
  qh = nfq_create_queue(h,  0, &callback, NULL); // link callback to queue
```

Again, we have a callback (in this case, named "callback") whenever a new packet enters the queue. This function is free to alter the packet as it sees fit, before returning a final *verdict* as to whether the packet should be forwarded, dropped, or re-queued. The processing that takes place while the packet is in userspace is the focus of our implementation. After this processing finishes, the packet is passed back to the kernel module, and onto the network (or not).

3.2.3 Key Processing Concepts

3.2.3.1 Stream Reading

One initial handicap in this project is our desire to read TCP packets as random-access data structures, when in fact they are meant to be used as streams. Accordingly, there is very little overall information about the structure of a TCP payload contained in the header; one is expected to discern it as the packet is read. For example, there is no indicator as to whether a given handshake record contains multiple messages or not. One simply reads the overall record length, and then reads messages from the record until one reaches the end. This is an extension of the problem shown in Figure 2-1.

With this problem in mind, much of our data processing is done with a scanning model, which inspects the data from a given read point and informs future decisions based on what it finds. Useful methods here include reading 2- and 3-byte length fields, while advancing the pointer. While this precludes jumping directly to the data we want, we are able to store information in auxiliary variables to prevent repeated readings of the data stream.

To visualize this process, we present the following code sample from the ClientHello processing code.

```
void Packet::handleClientHello(u8* data, u32* scan_offset) {
  u32 offset = 0;
  offset += 1;
  u32 message_length = parse3BLength(data, &offset);
  offset += 34;
  u32 sid_length = parse1BLength(data, &offset);
  offset += sid_length;
  u32 cipher_length = parse2BLength(data, &offset);
  offset += cipher_length;
```

```
  u32 comp_length = parse1BLength(data, &offset);
  offset += comp_length;
  ...
}
```

As you can see, we start at the beginning of the data buffer, and step through the data, bypassing fields based on their reported length.

3.2.3.2 Scatter-Gather

In order to manipulate the data in streams, we need a robust way of dealing with bits and pieces of a packet, while retaining the ability to stitch everything back together in the end. While we could maintain a continuous buffer corresponding to the "current" state of the packet that is being processed, this would end up incurring a significant cost through repeated memory allocations and block copying. For example, whenever we attempt to compress data that is in the middle of a given payload, we would have to retain pointers to the endpoints of the data and shift the buffer backward. To expand, we would have to copy the data to a larger buffer. Particularly in concert with the streaming nature of the data, this quickly becomes a nuisance.

To alleviate these problems, we use a technique called *scatter-gather*, which is used extensively in the Linux kernel to handle non-continuous blocks of data. The underlying idea is simple: for each data section, store a pointer and a length value; perform a final consolidation at the end. This is also a useful solution because it gives us a natural place to store the length of different parts of the data, which can only be learned by reading the stream itself.

The scatter-gather technique is also a useful way to keep track of relevant pointers into the stream structure, such as key length values.

In C++, we can implement scatter-gather using the `iovec` structure, which contains fields for the length and starting point of a given fragment. An illustration of how scatter-gather can be used to alter a buffer is shown in Figure 3-2.

As an example, when we are compressing certificates, we store iovecs for

- The start of the packet (including header) through the start of the TLS record;
- The start of the record through the start of the certificate;
- The entirety of the certificate;
- The end of the certificate through the end of the packet.

Notice that, by storing a pointer to the beginning of the TLS record, we can quickly correct the length field. Otherwise, we would have to re-scan most of the packet, which would now have some invalid length values. As one might imagine, this would be a rather perilous task.

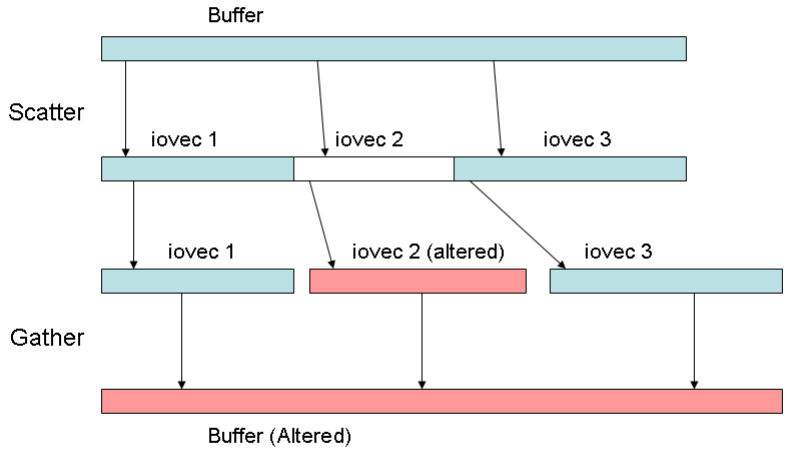

Figure 3-2: Scatter-Gather in Action.

```
IsComplete(A)
1  while HasNextPacket(A)
2        if PacketContainsNextRecord(A)
3              ScanToNextRecord(current-pointer(A))
4        else return FALSE
5        if end-of-packet(A)
6              return TRUE
7  return FALSE
```

Figure 3-3: Packet Completeness Code.

3.2.3.3 Packet Completeness

As discussed in Section 2.2.3, we cannot determine whether a given TCP payload contains a complete TLS record or not without reading most of the packet. To perform this necessary reading, we employ a scanning pass which simply runs through the stream, counting down the number of bytes left according to the IP header, and determines whether the payload has been split among multiple packets. We separate this from the actual handling of the packet in order to clarify code and keep operations modular.

If the successor to a given packet has already been seen, we append it to the data stream and determine whether the combined payloads comprise an entire TLS record. If this is the case, the multiple packets are consolidated into a single packet that is "correct" by the standards of TCP, but too large to be sent on the wire. We then process it however we may like, with the guarantee that it is a complete packet.

Conversely, if we have not seen the successor to a given packet, or if the necessary packets to complete its payload have not been seen, we return the packet to the netfilter queue. This allows us to process all other packets in the queue before returning to the packet in question, by which time we should have seen a successor. If a packet has already been requeued once, we allow it to pass through on all subsequent appearances.

Pseudocode for this process is shown in Figure 3-3.

3.2.3.4 Packet Storage and Packet Requeueing

One limitation of the netfilter queuing mechanism is that it can only examine a single packet at a given time. Since a TLS packet may span several TCP packets, we need some way to work around this, which we accomplish by storing potentially useful TCP packets in a packet store, and requeuing incomplete packets. When we are able to aggregate a complete TLS packet, we alter the accumulated TCP packets and forward them to the relevant destination. The basic structure for the netfilter callback is shown in Figure 3-4.

```
CALLBACK(A)
 1  ..
 2  if ISTLS(A)
 3       ..
 4       if ISCOMPLETE(A)
 5            B = MAKECOMPLETEPACKET(A)
 6            PROCESS(B)
 7       else REQUEUE(A)
 8       ..
 9  else STORE(A)
10  ..
```

Figure 3-4: Netfilter Callback.

3.2.3.5 Packet Fragmentation

Because network devices can only accommodate a finite packet size, we sometimes end up with a complete packet after alteration that exceeds the maximum TCP payload size. We work around this by expanding an oversized TLS packet into several properly sized TCP packets in a canonical way, such that we can be reasonably sure that the packets are now indistinguishable from the packets that were recently compressed. To do this transparently, we only need to get the SEQ, ACK, and checksum values right; the rest can be copied from the parent header.

3.2.3.6 Packet Modification

Once we have a series of iovec pointers into the packet, we can perform operations on one of the blocks without affecting the others. This corresponds to the *scatter* in scatter-gather. By the same token, we finishing our processing by *gathering* these iovecs and creating a single continuous packet from them.

For handling Certificate messages, we store 4 iovec pointers into a given packet. In a given operation, we are able to alter the contents of the Certificate message without affecting our ability to read into other sections of the packet. We can then quickly alter length fields and reconstitute a proper and complete packet by sewing together the iovecs.

3.2.4 Server-Only Authentication Implementation

We use the SNI to uniquely pair a certificate with a transaction, and identify certificates according to their serial numbers.

1. The client-side proxy A observes that a client C is initiating a TLS handshake with server S.
2. Given the ClientHello message from C, A parses the `server_name` field from the record, and attempts to match it against the `commonName` of a certificate in its cache.

3. If A's cache contains such a certificate X, it can be quite sure that this is the certificate that S will provide, unless it has expired.

4. Accordingly, A appends the hash of X to the ClientHello message and forwards it to the server-side proxy B.

5. If certificate X is part of a certificate chain, A appends the hashes corresponding to the rest of the chain, as well.

6. If A's cache does not contain such a certificate, A forwards the message to B, unchanged.

7. When B receives the packet, it takes note of the hashes L (if any) that are appended to the ClientHello message as a CPI extension.

8. Using these hashes, B updates its knowledge of the state of A, where A must have a copy of any certificate with a hash in L.

9. B removes L from the packet and forwards the packet to S (note that the packet is again identical to the one that C sent.)

10. Upon receiving a response from S, B alters the Certificate message accordingly.

11. For each certificate in the message, if L contained the hash of the certificate, then B replaces the entire certificate with its hash, truncating the packet accordingly.

12. B forwards the packet to A, which then replaces any hashes with the full certificates (Again, note that this packet is now identical to the one sent by S.)

13. A also adds any new certificates to its cache, and replaces any expired certificates with their new versions.

There are a few assumptions here. One is that the client supports the `server_name` extension to TLS. Empirically, this is true in the vast majority of cases, so this is a safe thing to bank on. Additionally, we are assuming that there is a bijection between hashes and certificates. It is quite likely that this will hold for the lifetime of the universe. Formally, we have assumed:

- There is an injective correspondence between hashes and certificates;
- There is a bijective correspondence between certificates and secure server names at any given time interval;
- The server name extension is ubiquitous enough to be depended on.

The approach of piggybacking on the rest of the TLS protocol provides us with a variety of advantages. For one, it makes for a very clean addition to the existing data, making the idea easy to conceptualize and inspect. In the same vein, we are not altering the protocol significantly, such that we get all of the other benefits of TLS for free, in particular, the ease of adapting TLS-centric code.

3.2.5 Server-Client Authentication Implemenation

Implementing certificate compression for a server-client authenticated handshake is done in a similar way to the server-only case. A subtle point here is that, because we have chosen to communicate between proxies within the confines of the TLS message structure, compression of the server certificate is effectively transparent to code that is trying to compress client certificates, and vice versa. Basically, when the processing code is looking for a client certificate, it will only see a CPI extension as just another ClientHello extension, and a CompressedCertificate message as just another handshake message.

Here, we have to alter the openssl implementation of the TLS client to force certificate selection and delivery in the first round, together with ClientHello.

3.2.5.1 Forced Certificate Send

Toward compressing client certificates, recall that we have discussed altering the client TLS implementation in order to indicate what certificate is going to be sent in the ClientCertificate message. We will do exactly that in our implementation, using openssl as our TLS software of choice.

To preserve flexibility in the system, we add a `-force_client_cert` option to the `s_client` utility. When this option is selected, the hash of the client certificate is sent as an extension of the ClientHello message. This allows the server-side proxy to send a certificate-specific message to the client-side proxy, indicating whether or not it has the certificate in question.

Note that the server TLS implementation does not need to be altered, because any unknown ClientHello extensions will be discarded by the server.

This transparency between client and server certificate compression allows code that accomplishes one operation to interoperate freely with code which accomplishes the other. Accordingly, the following is an overview of how client certificates are compressed.

1. Client C, with modified TLS software, attaches a hash for certificate X to ClientHello.
2. Given the ClientHello message from C, server-side proxy B attempts to match the hash against a certificate in its cache.
3. B receives a ServerHello message from S.

4. If B's cache contains X, it appends the hash to the ServerHello message and forwards it toward the client.

5. If B's cache does not contain such a certificate, it forwards the message to the client unchanged.

6. When client-side proxy A receives the packet, it takes note of the hashes L (if any) that are appended to the ServerHello message.

7. Using these hashes, A updates its knowledge of the state of B, where B must have a copy of any certificate with a hash in L.

8. A removes L from the packet and forwards the packet to C (Note that the packet is again identical to the one that S sent.)

9. Upon receiving a client certificate X from C, A alters the Certificate message accordingly.

10. If B indicated that it had certificate X, A replaces the entire certificate with its hash, truncating the packet accordingly, and forwarding it toward S.

11. B intercepts the packet, and then replaces any hashes with the full certificates (Again, note that this packet is now identical to the one sent by C.)

12. If B did not indicate that it had the certificate, the full Certificate message is sent toward S, and intercepted by B.

13. B also adds any new certificates to its cache, and replaces any expired certificates with their new versions.

As you can see, the compression operations are very similar to the server-only authentication operations, and we are able to reuse a large amount of code.

Chapter 4

Experimentation

Having first outlined and then built a working system of intercepting proxies, our proof-of-concept is complete. We will now show its performance advantages through a series of experiments. In order to test the different aspects of our system, we first establish a reliable testing environment. Within this environment, we profile the code we have written, to suggest future directions for optimized proxies. Next, we test the latency reduction of our system during TLS handshakes over disadvantaged links. Finally, we test the robustness of the system.

4.1 Test-bed Setup

We can profile the performance of the system using benchmarking software developed at MIT Lincoln Laboratory. In order to test our networking code in a controlled and repeatable way, we have reserved two computers on the Laboratory's networking test-bed. With one computer acting as the client and the other computer acting as the server, we deploy the proxy software locally on each computer, treating the link between them as the disadvantaged link.

4.1.1 Servers

In order to use this test-bed, we need to predicate certain sections of the netfilter code on the Linux kernel version, as our development machine and the test machines are running different versions of the kernel. An example of this can be seen in the code snippet in Figure 4-1, where the signature of the `queue_tls` prototype is different between different kernel versions. For compatibility, we predicate the signature on preprocessing directives.

```
unsigned int queue_tls(unsigned int hooknum,
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,24)
                       struct sk_buff *skb,
#endif
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
                       struct sk_buff **skb,
#endif
                       const struct net_device *in,
                       const struct net_device *out,
                       int (*okfn)(struct sk_buff*)) {
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,24)
  struct sk_buff *sock_buff = skb;
#endif
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,6,24)
  struct sk_buff *sock_buff = *skb;
#endif
...
}
```

Figure 4-1: Preprocessing Predicated on Kernel Version.

4.1.2 Link Emulation

The test-bed includes a kernel module designed to simulate a network device of the specified capability. This emulation software allows us to set specific parameters of a network device dynamically, by setting *Command* directives. We accomplish this via a shell script.

Once the emulation software has been configured, we can use the `tls` device as a valid network device.

4.1.3 Openssl

We use the `s_server` and `s_client` utilities to reliably generate TLS handshakes in a controlled environment.

For the server, we can test various certificate chain lengths. Note that we disable the session ticket extension [7] for the purposes of testing. Normally, this extension would attempt to re-use a session ID when reconnecting to a server. However, we have been designing for servers which do not support this feature, and so we disable it. If present, this extension interferes with our ability to test the proxies' performance.

The specific commands can be found in Appendix A.1.

4.2 Code Profiling

In order to better understand the processing overhead incurred by the proxies, we profiled the proxy code. From our testing, it is clear that we need to account for approximately 200 ms of processing

time, between the two proxies, per handshake.

4.2.1 Userspace Processing

By timing our code execution, we determined that the majority of packet processing time comes from userspace processing. This accounts for more than 90% of the total processing time. Over the entirety of the handshake, these processing delays at both client and server combine to create a delay of up to 0.20 seconds in the server-client authenticating handshake.

Having isolated userspace processing, we made our profiling more granular and examined the different operations which occur in userspace. From this analysis, we determined that over 90% of the userspace processing time is taken up with SQLite operations. This is a reasonable result, since the proxy is performing a disk access whenever it uses the database, and since it does not store a persistent database handle or attempt to store the database in memory.

4.2.2 Context Switching

We also profiled the context switching between the kernel and userspace. In order to do this, we had to output the current state of the CPU TSC (time stamp counter) immediately before leaving the kernel, and then output the clock value immediately upon entering userspace. We assumed that threads were not migrating between CPUs. In each of 5 trials, the difference was approximately 500,000, which, on a 1.8 Ghz machine, is negligible.

4.3 Baseline Latency Comparison

We continue our experiments by establishing a baseline with which to compare previous results. Having done this, we explore the rest of the parameter space, including fallback scenarios and certificate size. Since his work was performed under similar conditions, we use Matthew Low's summer work at Lincoln Laboratory as a baseline for comparison. We use the link emulation software to simulate five different real-world link types, listed in Table 4.1.

4.3.1 Userspace Processing

By timing our code execution, we determined that the majority of packet processing time comes from userspace processing. This accounts for more than 90% of the total processing time. Over the entirety of the handshake, these processing delays at both client and server combine to create a delay of up to 0.20 seconds in the server-client authenticating handshake.

Having isolated userspace processing, we made our profiling more granular and examined the different operations which occur in userspace. From this analysis, we determined that over 90%

of the userspace processing time is taken up with SQLite operations. This is a reasonable result, since the proxy is performing a disk access whenever it uses the database, since it does not store a persistent database handle or attempt to store the database in memory.

4.3.2 Context Switching

We also profiled the context switching between the kernel and userspace. In order to do this, we had to output the current state of the CPU clock immediately before leaving the kernel, and then output the clock value immediately upon entering userspace. The difference was approximately 500,000, which, on a 1.8 Ghz, machine, is negligible.

4.4 Latency Comparison

Since his work was performed under similar conditions, we use Matthew Low's summer work at Lincoln Laboratory as a baseline for comparison. We use the EML software to simulate five different real-world link types, listed in Table 4.1.

Table 4.1: Simulated Wireless Link Parameters

Connection type	Client data rate (bps)	Server data rate (bps)	Latency (ms)
iridium	2400	2400	400
inmarsat-64	64000	64000	442
milstar	2400	256000	500
tcdl-10	10000000	10000000	0
cell modem	97877	981525	90

Using these different links, we ran the server-only handshake over a normal and proxied connection. The data for these trials is shown in Figures 4-2 and 4-3. Each bar represents the average of 20 trials. For each set of trials, the standard deviation was negligible.

From the graphs, we can see that for high-latency links, our proxies acheive a considerable speedup if the link is also low-bandwidth. For example, over an Iridium link, the average handshake delay is reduced from 9.5 seconds to 6.7 seconds by the presence of the proxies. If the link has high bandwidth, the proxies have little effect on latency. On low-latency links, the proxies also have little effect on the latency of the TLS handshake.

This data illustrates several key aspects of the proxies' performance:

4.4.1 Processing (Needn't Be) Expensive

When comparing the proxied handshakes to their non-proxied counterparts, it is clear that there is a constant size overhead that is added to the handshake delay in the proxied case. We have explored the exact cause of this in Section 4.2, and it is clear that this overhead is incurred by the data

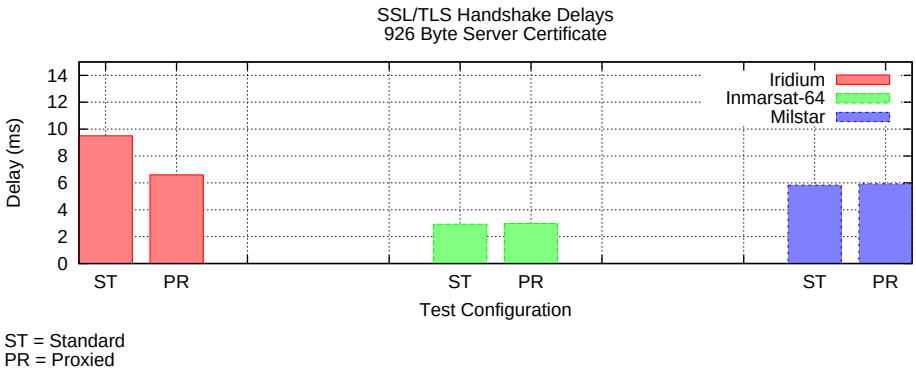

Figure 4-2: Server-Only Handshake Performance, High Latency.

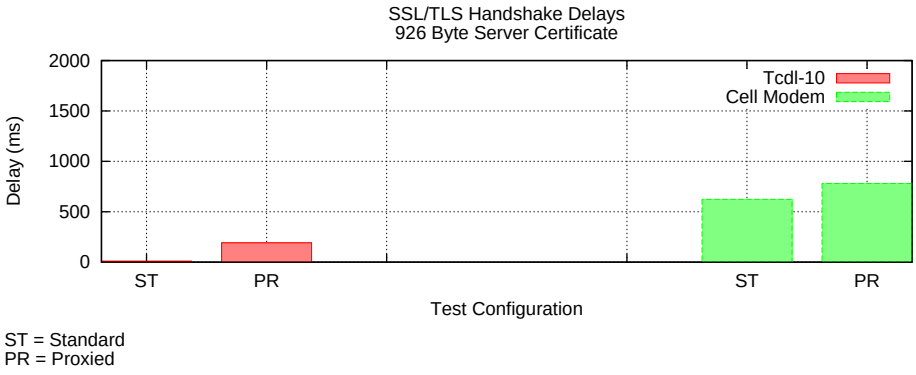

Figure 4-3: Server-Only Handshake Performance, Low Latency.

processing that the proxies perform. Since our setup was a proof-of-concept, lacking anything in the way of optimizations, this overhead can effectively be ignored in considering the suitability of a proxied solution for a real-world deployment. Such a deployment would feature optimized code, negating any overhead.

4.4.2 Bandwidth Savings Beget Latency Savings

As we anticipated, the ability to reduce packet size through certificate compression can actually allow us to reduce the total number of packets involved in the handshake. On high-latency, low-bandwidth connections, this effect is readily apparent.

Looking at the Iridium data, we take note of the bandwidth cap of 2400 bits. Notice that the server certificate, weighing in at 926 bytes, exceeds this cap, and thus cannot be transmitted over the Iridium link in a single trip. However, a compressed certificate packet can slip in under this cap, and make the trip in a single packet. Since Iridium features relatively high latency, the proxies effect a considerable reduction in the overall handshake delay, on the order of 3 seconds.

4.4.3 Better Performance for Server-Client Authentication

Since the server-client-authenticating handshake involves both client and server certificates, the amount of data that we can compress increases along with the potential savings that our proxies can effect. In our experiments, this hypothesis is confirmed, as the proxies are able to reduce the latency of a server-client-authenticating handshake on the Iridium link by almost 50%, or by more than 6 seconds. Again, this is due to the high-latency, low-bandwidth characteristic of the link, where the proxies reduce the total number of packets that need to traverse the link, and thus the total latency.

Our data is illustrated in Figures 4-4 and 4-5. These graphs show the performance of proxied handshakes over high-latency and low-latency links, respectively.

4.5 Bandwidth Comparison

The proxies have a large effect on the bandwidth cost of a handshake, reducing the aggregate bandwidth usage of a given handshake by up to 50%. In Figure 4-6, we compare the bandwidth usage of a standard server-only handshake with a proxied handshake. Since the certificate is compressed to a constant size, the unproxied handshake's cost scales with certificate size, while the proxied handshake's size does not.

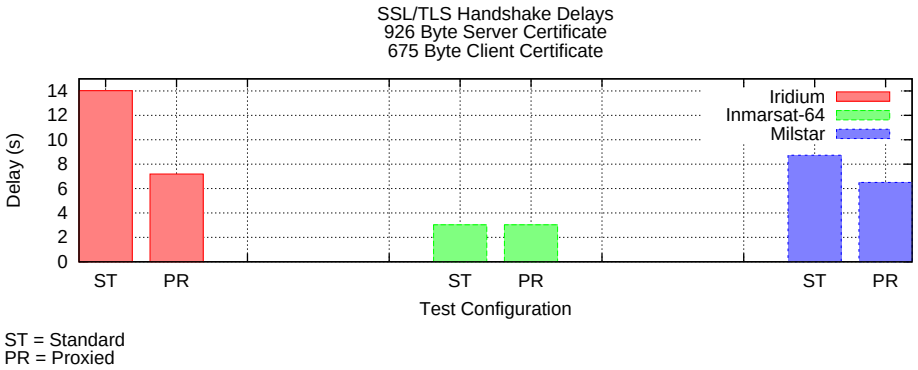

Figure 4-4: Server-Client-Authenticating Handshake Performance, High Latency.

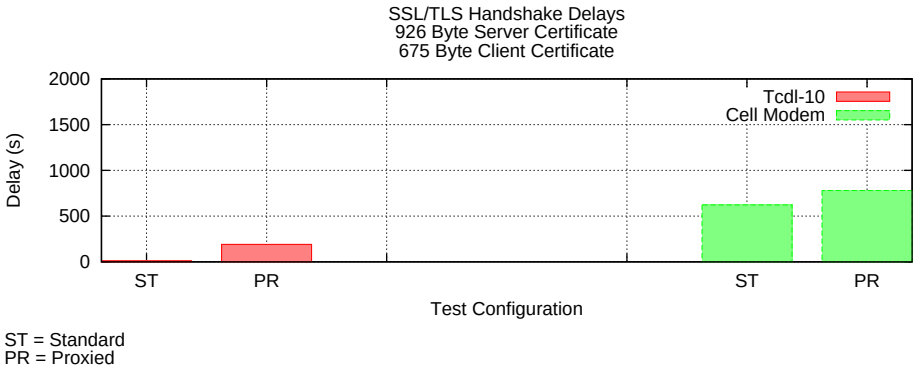

Figure 4-5: Server-Client-Authenticating Handshake Performance, Low Latency.

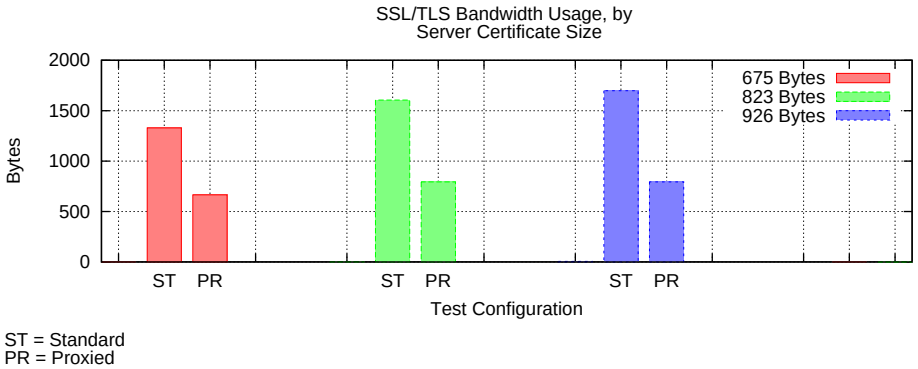

Figure 4-6: Server-Only Bandwidth Performance.

4.5.1 Certificate Chains

The proxies are able to compress each certificate in a chain individually, as shown in Figure 4-7. Wireshark interprets the unknown message type (0x99) as Encrypted Handshake message. Recall that we defined this format in Section 2.3.3. In the capture, one can see the length field (0x30) and the three 16-byte certificate references that follow.

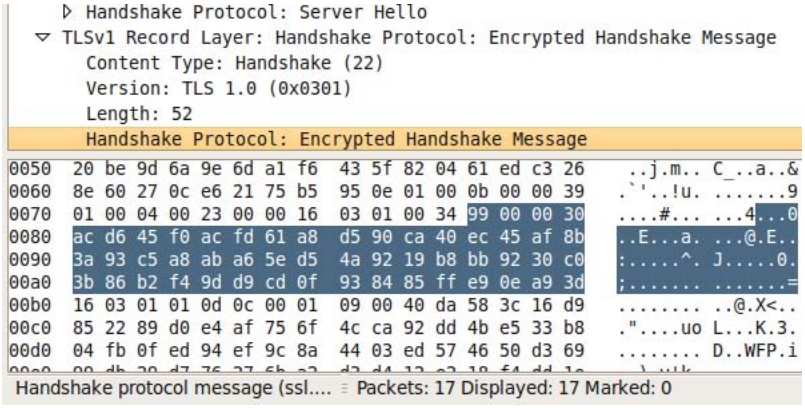

Figure 4-7: A Compressed Certificate Chain.

The unproxied handshake would be parsed as in Figure 4-8.

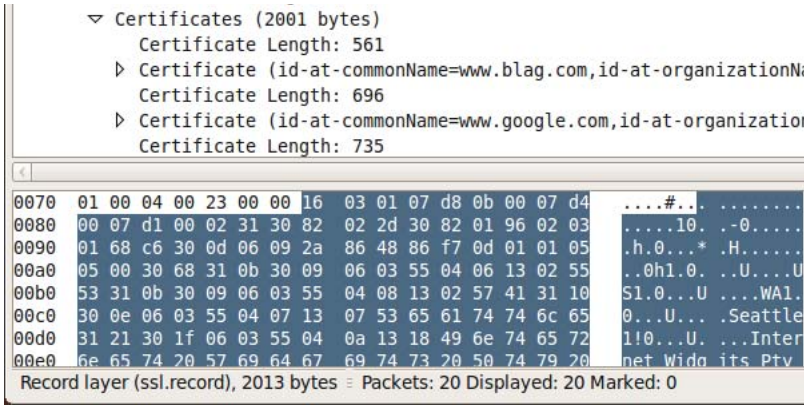

Figure 4-8: A Normal Certificate Chain.

4.6 Robustness

As we discussed in Section 2.3.4, it is possible for there to be a cache miss between the two proxies, caused by some error in their mutual state records. When a cache miss occurs, a proxy is given a certificate reference that it cannot expand. We handle this case by dropping the packet, which causes the underlying TCP connection to attempt to re-transmit the packet. The second time around, the

proxies recognize that they have already seen this packet, and do not attempt to alter it. The packet then reaches its destination, and the handshake continues correctly.

In Figure 4-9, we show the packet flow of such a cache miss, where the server-side proxy sent the client-side proxy a compressed certificate that the latter did not have in its database. Shortly thereafter, the server retransmitted the packet, causing the server-side proxy to forward the entire certificate to the client-side proxy, and from there to the client. Notice the retransmission delay between packets 6 and 7, which is approximately 750 ms.

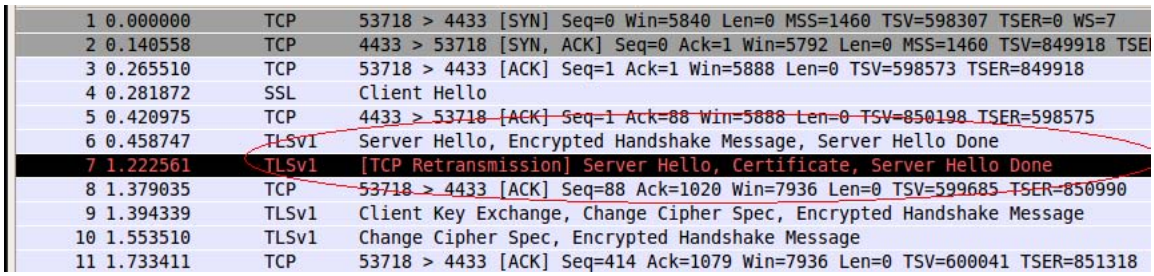

Figure 4-9: Fallback Packet Flow.

The performance of the system in this fallback scenario is almost entirely dependent on the underlying TCP connection and its retransmission time. This retransmission time increases the handshake delay by a constant amount, whenever there is a cache miss. Future work could reduce this time by crafting a TCP packet at one of the proxies to explicitly request retransmission.

Chapter 5

Conclusions

We have designed and built a working proof-of-concept system of intercepting proxies. Our experiments clearly show the advantages of such a system. Further, this work is contributing to a larger effort at Lincoln Laboratory to establish viable approaches to secure communication in tactical networks.

5.1 Benefits of Proxied Links

Our analysis has shown that a major benefit can be derived by deploying intercepting proxies across a low-bandwidth, high-latency link. Additionally, our data shows that performance across other links does not noticeably suffer from the presence of the proxies. Further, we note that the links that our proxies do not benefit would not be considered disadvantaged in the first place, and would not be candidates for proxy deployments. In all cases, there is a large bandwidth savings incurred by deploying the proxies that is proportional to the size of the certificates involved.

5.2 Applications

Our solution considerably outperforms a standard TLS handshake over any link with low bandwidth and moderate to high latency. The presence of the proxies incurs bandwidth savings of over 50% and latency savings of nearly 30% for server-only handshakes, in the case of the Iridium link. This speedup is gained without requiring any modification to the client or the server, operating in an entirely transparent fashion. If we alter the client TLS implementation, we can reduce the total handshake delay by over 40% in a server-client-authenticating handshake over an Iridium link. Further, the bandwidth savings that could be effected by applying this solution to existing links is considerable, such that any entity with a high cost-per-bit could benefit from our solution.

While it is clear that our proxy solution does not exceed the performance of certificate-removal

solutions which alter the client and server TLS implementations, it comes close to replicating their performance over disadvantaged links. Further, our solution is significantly more viable for deployments where the owner of the link is not the owner of the client or server.

5.3 Acquired Skills

5.3.1 Openssl

This project afforded me an excellent opportunity to familiarize myself with many parts of the openssl project. I became familiar with the command-line utilities, `s_client` and `s_server`, and their usefulness as diagnostic tools. While I did not spend as much time delving into the TLS implementation code as I would have liked, I gained a cursory knowledge of the way in which openssl implements the TLS state machine, and the programming conventions that are used throughout the codebase.

5.3.2 TCP/IP Details

Though I had studied the protocols in previous courses, this project gave me a very good opportunity to become more comfortable with programming in the Linux network stack. This allowed me to think more about how network traffic is handled by actual programs rather than abstract state machines, and I think this will prove very valuable in the future.

5.3.3 Wireshark

Having never used wireshark (or ethereal) before this project, the discovery of this packet sniffing tool was certainly a pleasant one. Wireshark was incredibly useful for testing network code that alters packets, allowing for much more readability than a hex dump to stdout.

5.3.4 SQLite

This project allowed me to gain more hands-on experience with the SQLite library. This was very useful for dealing with large chunks of binary data that might otherwise have caused memory management headaches.

References

[1] Getopt - The GNU C Library. `http://www.gnu.org/s/libc/manual/html_node/Getopt.html`.

[2] SQLite. `http://www.sqlite.org/`.

[3] The netfilter.org Project. `http://www.netfilter.org/`.

[4] The OpenSSL Project. `http://www.openssl.org/`.

[5] VMWare. `http://www.vmware.com/`.

[6] Wireshark. `http://www.wireshark.org/`.

[7] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 4366 (Proposed Standard), April 2006. Obsoleted by RFC 5246.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), January 1999. Obsoleted by RFC 4346, updated by RFC 3546.

[10] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17:281–308, 1988.

[11] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile. RFC 2459 (Proposed Standard), January 1999. Obsoleted by RFC 3280.

[12] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFC 1349.

[13] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168.

[14] RL Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. 1978.

[15] Hovav Shacham and Dan Boneh. Fast-track session establishment for TLS. In Mahesh Tripunitara, editor, *Proceedings of NDSS 2002*, pages 195–202. Internet Society (ISOC), February 2002. Extended abstract of SBR04 journal paper.

Appendix A

Auxiliary Architecture

This appendix details our architecture for testing and analysis.

A.1 Handshake Execution

In order to execute a series of handshakes, we use the following commands.

```
date;time for((i=0;i<1;i++));do echo === test $i ===;echo|openssl s_client -host
    192.168.10.2 -tls1 -mtu 3000 -cert client700.pem -key client700np.pem ;done
    |tee data-no-proxy.txt
openssl s_server -cipher AES256-SHA -cert server926.pem -key server-np.pem -tls1
    -mtu 3000 # For server-only auth
openssl s_server -cipher AES256-SHA -cert server926.pem -key server-np.pem -tls1
    -mtu 3000 -Verify 1 # For client auth
```

A.2 Packet Capture

In order to capture packets, we set up the `s_client` and `s_server` instances on separate NLET machines. We can then run a packet capture tool on one of the computers (for consistency, the client computer) in order to read data from the disadvantaged link.

We capture packets locally by using the `dumpcap` utility, which produces `.pcap` files. We can then extract information from these using a variety of tools, including ssldump and wireshark.

```
/usr/sbin/dumpcap -i eth1.1015 -w ../analysis/client-wo-proxy-iridium.pcap &
```

A.3 Data Post-Processing

We then run these through various processing scripts to extract the bandwidth usage and handshake delays. Given a series of captures in `.pcap` format, we use the following code to parse the handshake delays and output them to a `.dat` file.

```
ROWS=5
FILE="avg-with.dat"

echo -n -e "" > $FILE
echo -n "#␣" >> $FILE
for i in iridium inmarsat tcdl milstar cell; do echo -e -n "$i\t" >> $FILE; done
echo -n -e "\n" >> $FILE
echo "#␣Matt's␣baseline" >> $FILE
#echo -e "1       9.54398\t 6       2.94101\t 11      5.86757\t 16      0.0350083
     \t     21      0.683184        " >> $FILE
echo -e "#␣averages␣without␣proxy" >> $FILE
COL=2
for i in iridium inmarsat milstar tcdl  cell; do
    echo -e -n "$COL␣\t␣" >> $FILE
    tshark -r server-wo-proxy-$i.pcap -n -d ethertype==1546,ip -d
        tcp.port==4433,ssl | awk -f s1.awk | awk -v ORS="\t"  '{sum += $1} END
        {print sum/20}'  >> $FILE
    echo -e -n "\t" >> $FILE
    let COL+=ROWS;
done
COL=3
echo -n -e "\n" >> $FILE
echo -e "#␣averages␣with␣proxy" >> $FILE
for i in iridium inmarsat milstar tcdl  cell; do
    echo -e -n "$COL␣\t␣" >> $FILE
    tshark -r server-with-proxy-${i}2.pcap -n -d ethertype==1546,ip -d
        tcp.port==4433,ssl | awk -f s1.awk | awk -v ORS="\t"  '{sum += $1} END
        {print sum/20}'  >> $FILE
    echo -e -n "\t" >> $FILE
    let COL+=ROWS;
done
```

(Note that the above listing is specific to the server-only handshakes, but all of the other cases are trivially similar.)

The `s1.awk` script parses the actual delays from the modified tshark output.

```
# interleaving will break this script -- must track flows...
/\[SYN\]/ {s=$2}
/TLS(v[0-9]+)?.*Encrypted Alert/ {print $2-s}
```

A.4 Graph Creation

`gnuplot` is our graphing software of choice. We simply create a bar graph from the various data points in the `.dat` file, like the one below. The format is simply a column number followed by its value, separated by whitespace.

```
# iridium inmarsat tcdl milstar cell
# Matt's baseline
1    9.54398  6 2.94101  11   5.86757  16  0.0350083   21   0.683184
# averages without proxy
2   9.50572 7   2.92503 12   5.82993 17   0.01062 22   0.623161
# averages with proxy
3   6.68903 8   3.184 13   6.11165 18   0.455612 23   1.08356
```

A.5 File List

queue.c Source code for the netfilter kernel module.

queuetest.cc Source for the proxy executable. Must be run after the module has been added to the kernel.

proxyssl.h Genereal purpose header file for the project.

packet.h,cc Source for the Packet, Proxy, and TcpStream classes.

x509cert.h,cc Source for the Certificate class.

analysis/*.gnuplot Graph generation scripts.

analysis/*.sh Data post-processing scripts.

certs/* Test certificates and keys.

A.6 Command-line Flags

The executable can be run with the following command-line flags:

- `--server` Runs the program as the server-side proxy;
- `--client` Runs the program as the client-side proxy.

A.7 Database Schema

Our tables track certificates and individual TCP packets. We store key indexing information about these objects, as well as their insertion time, so that we can periodically purge dead TCP streams.

```
CREATE TABLE certs (
  csum blob,
  data blob,
  issuer blob,
  subject blob,
  commonname string(32),
  insert_time date);

CREATE TABLE tcp (
   syn int(32),
   ack int(32),
   saddr int(32),
   daddr int(32),
   data blob,
         insert_time date);


CREATE TABLE client_state_message (
   syn int(32),
   ack int(32),
   source int(32),
   dest int(32),
   checksum blob,
```

```
        insert_time date);


CREATE TRIGGER insert_cstate AFTER INSERT ON client_state_message
BEGIN
UPDATE client_state_message SET insert_time=strftime ('%s','now','utc') WHERE
    ROWID=new.ROWID;
END;

CREATE TRIGGER insert_cert AFTER INSERT ON certs
BEGIN
UPDATE certs SET insert_time=strftime ('%s','now','utc') WHERE ROWID=new.ROWID;
END;

CREATE TRIGGER insert_tcp AFTER INSERT ON tcp
BEGIN
UPDATE tcp SET insert_time=strftime ('%s','now','utc') WHERE ROWID=new.ROWID;
END;

CREATE TRIGGER clean_tcp AFTER INSERT ON tcp
BEGIN
DELETE FROM tcp WHERE strftime ('%s','now','utc') - insert_time > 60;
END;
```

A.8 Test Certificates

Using openssl, we can generate test certificates and certificate chains in order to use our code in the real world. First, we have to create a new certificate authority. Next, we sign certificate chains, starting with this CA, in order to use test certificates in our TLS handshakes.

To create a new CA, use the following code. Note that the location of `CA.pl` may vary from system to system.

```
/.../CA.pl -newca
/.../CA.pl -newreq
/.../CA.pl -sign
```

To remove the passcode from a key, use the following method:

```
openssl rsa -in keyfile.pem -out keyfile-no-password.pem
```

Appendix B

Aside: TLS Round Alteration

As a curiosity, we present the following exploration of our ability to alter TLS rounds when modifying only the client. None of the following was implemented.

We observe that it is possible to modify the TLS protocol on the client to alter the structure of rounds in the handshake, without modifying the implementation on the server, using the proxies to make the process transparent to the server. This makes deployment significantly more feasible than other fast certificate strategies [15], as it removes any burden of installation from the server operators. The data flow of this modified handshake is shown in Figure B-1. This also shows that round reduction is impossible without altering the TLS implementation on the server, as the server must receive CertificateVerify before it can send ChangeCipherSpec.

The implementation of this server-transparent idea is highly non-trivial. We will now explore the necessary steps toward implementing such a *server-transparent patch*.

B.0.0.1 Patch Specifics

In order to send the client certificate in advance, we will need to alter the TLS state machine inside of openssl so that the certificate can be sent in the first round. Beyond that, much of the server authentication code is reusable as far as compressing certificate messages. The TLS messages for client and server certificates are identical, so the problem of compression itself is effectively solved. What remains is to successfully modify packets in a manner that is transparent to the server, such that a client with the server-transparent patch can communicate with an unpatched server. To do this, we need to consider the following factors:

- MAC - The TLS handshake authenticates the entire exchange by performing a message check over the entire handshake as it expects for it to take place. If we are going to transparently modify the payload, we need to be sure that this authentication step is not affected. Accordingly, we need to alter the way the MAC is computed in the client application so that it

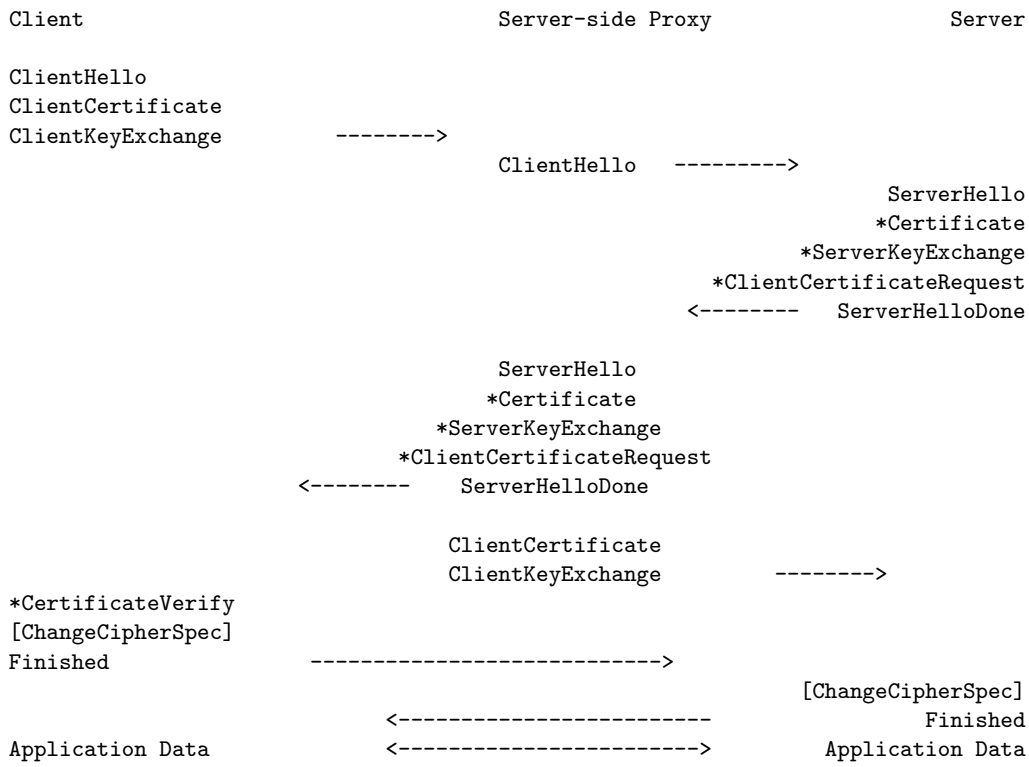

Figure B-1: A Server-Transparent Round Alteration.

matches the order that the server sees.

- Server-side Storage - Once the Certificate (and other messages) has been received by the server-side proxy, it must be held until an appropriate time in the handshake, when the server will be expecting it. Accordingly, we need to track the state of the handshake and create a means of persistent storage in the server-side proxy.

- TCP Transparency - Since the server-side proxy is re-ordering packets that are sent from the client, this will alter the SEQ/ACK numbers involved in the transaction. Further, once a single packet is offset, then the whole stream is then offset. Thus, the server-side proxy will have to dynamically adjust all TCP packets so that they correspond to the "correct" numbers that are expected by client and server.

Respecting these three concerns at the same time represent a significant undertaking that is beyond the scope of this project.

Appendix C

Development Environment

C.1 Installation

In order to create a working copy of the code, one can reconstruct our setup of Ubuntu 9.04 with the following packages installed:

```
sqlite3
libsqlite3-dev
subversion
tshark
wireshark
g++
emacs
libssl-dev
libnetfilter-queue-dev
```

This should allow the netfilter module, userspace code, and openssl code to be compiled.