Optimizing Secure Communication Standards for Disadvantaged Networks

by

Stephen Hiroshi Okano

B.S., Massachusetts Institute of Technology (2005)

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

September 2009

Author ....
Department of Electrical Engineering
and Computer Science
September 1, 2009

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

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

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

Optimizing Secure Communication Standards for Disadvantaged Networks

by

Stephen Hiroshi Okano

Submitted to the Department of Electrical Engineering
and Computer Science
September 1, 2009,
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We present methods for optimizing standardized cryptographic message protocols for use on disadvantaged network links. We first provide an assessment of current secure communication message packing standards and their relevance to disadvantaged networks. Then we offer methods to reduce message overhead in packing Cryptographic Message Syntax (CMS) structures by using ZLIB compression and using a Lite version of CMS. Finally, we offer a few extensions to the Extensible Messaging and Presence Protocol (XMPP) to wrap secure group messages for chat on disadvantaged networks and to reduce XMPP message overhead in secure group transmissions. We present the design and implementation of these optimizations and the results that these optimizations have on message overhead, extensibility, and usability of both CMS and XMPP. We have developed these methods to extend CMS and XMPP with the ultimate goal of establishing standards for securing communications in disadvantaged networks.

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

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

Acknowledgments

I would like to thank Dr. Roger Khazan for leading me through this thesis. Without his help and guidance I surely would have been lost. I would also like to thank Joe Cooley for his help in all coding aspects and in setting up everything I needed to do work. I learned a great deal from him about Linux, coding, and how fast a Kimball's special can be eaten. I also would like to thank Ben Fuller for helping me with the ASE library and with many other code and conceptual problems. I would also like to thank Roger, Joe, and Ben for taking the time to help revise this thesis.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Motivation

The Internet Engineering Task Force (IETF) and other standards organizations have recently defined standards for cryptographic message formats to be used for securing communications in computer networks [8]. These standards focus on addressing interoperability and extensibility issues; they have predominantly been driven by the needs of terrestrial networks. Little or no consideration has been given to mobile, wireless networks, whose links maybe constrained in their communication capacities and connectivity.

At the same time, the use and importance of mobile, wireless networks has been growing due to smartphones, mobile devices capable of ad-hoc networking, and the general trend of networking on-the-go.

In the national defense sector, the Department of Defense is pursuing a transformational vision, called Network-Centric Operations (NCO) [8]. The tenets of the NCO vision express the idea that a robustly inter-networked force improves information sharing and collaboration, which ultimately lead to a dramatic increase in mission effectiveness. Much of forward deployed force will be connected via mobile, wireless networks, which in the defense sector are known as *tactical networks.* Tactical networks are also known as *disadvantaged networks* because they are often made up of communication links that are low-bandwidth, high-latency, and intermittent.

This thesis is motivated by these two trends and considers the problem of using and adapting standard cryptographic message formats for the use in disadvantaged networks. In addition, this thesis is contributing to a specific effort at MIT Lincoln Laboratory. This effort, called Dynamic Group Keying (DGK), is developing solutions for securing communication and applications in tactical networks, and in particular, group-oriented applications, such as those that involve information sharing and collaboration. The ultimate goal of DGK is to standardize its solutions for the use in tactical networks and, outside the defense sector, in general mobile, wireless networks. A step towards this goal is to define solutions proposed by DGK atop existing and accepted standards, and in particular, atop accepted standards for cryptographic messages. But in order to maintain relevance to tactical, disadvantaged networks, the underlying standards need to be optimized for the use on such networks.

1.2 Thesis Overview

Therefore, in this thesis, we investigate methods for optimizing and implementing standardized cryptographic message formats for use in disadvantaged networks. We first provide an assessment of the current standards and their relevance to disadvantaged networks. Then, we offer methods to reduce message overhead in packing Cryptographic Message Syntax (CMS) [35] structures by using ZLIB compression and investigate a possibility of creating a Lite version of CMS. Finally, we offer a few extensions to the Extensible Messaging and Presence Protocol (XMPP) [52]: one – to secure group chat messages on disadvantaged networks, and the other – to reduce XMPP message overhead in secure group transmissions.

We present the design, implementation, and results that these optimizations have on message overhead, extensibility, and usability of both CMS and XMPP. We have developed these optimizations to extend CMS and XMPP to support the ultimate goal of eventually establishing a standard for the use of these optimizations in disadvantaged networks.

1.2.1 Roadmap

This thesis is separated into two distinct parts. The first part deals with finding a standard for cryptographic message packing and with the optimizations to that standard for disadvantaged networks. The second part is a self-contained section that deals with applications of secure group communications. It describes an extension and optimization to XMPP in order to support secure group messages.

In Chapter 2, we begin with the standards organizations and evaluate the standards currently used for cryptographic message packaging. As the result, we choose to focus on Cryptographic Message Syntax (CMS) [35] as the target of our optimizations for its use in disadvantaged networks. Consequently, in Chapter 3, we overview CMS and its advantages and disadvantages. Chapter 4 describes the design of two optimizations to reduce CMS messages size: ZLIB compression and a CMS Lite approach. Chapter 5 covers the implementation of these optimizations in OpenSSL [13], and Chapter 6 offers an evaluation of these two optimizations against each other.

As mentioned above, Chapter 7 is a self-contained description of the design and implementation of a group end-to-end encryption protocol extension to enable secure group communications in XMPP. It also includes the design and implementation of *subset addressing*, an optimization to XMPP group chat protocol. These extensions constitute another example of optimizing a message packaging standard for secure group communications.

Chapter 8 concludes the thesis and details further work.

Chapter 2

Standards Options

In the Introduction we discussed our motivation for this project and gave an overview of the thesis. We discussed our desire to use standardized communications for secure group communications in disadvantaged networks. However, we must understand what standards are relevant to secure group communications before we choose one and optimize it for disadvantaged networks.

In order to understand which standardized messaging protocol is right for us, we must learn which organizations currently standardize internet protocols and security mechanisms. There are several major players in the development of computer security standards.

- The National Security Agency creates guidance for the government on information assurance and has several security standards profiles which it recommends for interoperability among government assets.

- RSA Security defined and maintains the packaging and API for wrapping public key certificates and associated data necessary for asymmetric encryption, called Public Key Cryptography Standards (PKCS) [39]. For security mechanisms that use Public Key Cryptography, this standard defines how PKCS implementations interoperate with one another.

- The Institute of Electrical and Electronics Engineers (IEEE) specializes in developing industry-wide standards. Wireless networking and networking standards were all developed by IEEE. In relation to key management, IEEE Group 1619.3 focuses on key management issues.

While these three organizations all publish standards, for this project we focus on the following organizations:

- the Internet Engineering Task Force (IETF),
- the Organization for Advancement of Structured Information Standards (OASIS),
- the National Institute of Standards and Technology (NIST).

These three organizations have been involved in the creation of Internet standards related to communications and security and have open collaboration in creating new standards. Both of these qualities make them attractive to our project.

2.1 IETF

2.1.1 Area of Expertise

The IETF accepts Requests for Comments (RFCs) which specify how to perform tasks on the Internet and which promote the operation of the Internet. Usually these standards are a best common practice. These standards are subject to update whenever a better way of performing the same task has been identified. Some important RFCs which have been developed by the IETF are the Simple Mail Transport Protocol (SMTP) to send e-mail [41], the Hypertext Transfer Protocol (HTTP) [31] providing how to load and view web-pages, the Security Architecture for IP (IPSec) tunneling protocol, which enables security mechanisms for Internet traffic at the IP level [40], and Transport Layer Security (TLS), which provides privacy and data integrity over a transport protocol, traditionally HTTP [30]. An RFC that relates to our project is the specification for Secure Multipurpose Internet Mail Extensions (S/MIME) [47], which describes how to use public key encryption and signing of mail-formatted messages.

Another is the Cryptographic Message Syntax described in RFC 3852 [35]. It defines how to package cryptographically protected messages and is specified to be used in S/MIME encryption. The NSA has been known to adopt technology standardized by the IETF.

2.1.2 Process to Create Standards

A proposed standard must go through several states in order to become a full standard in the IETF [26]. First, the specification must be stable and contain no omissions. An implementation of the specification is generally desired in order to test the specification. This state is called a proposed standard. When separate interoperable implementations with different code bases have been created for a proposed standard, and there has been some operational use of the standards, then the standard may advance to draft status. A working group in the IETF reviews the draft standard and it may become a full standard when it is accepted as beneficial to the Internet by the general community. An experimental standard may be submitted to the IETF in relation to research efforts [27]. These may be changed into proposed standards if they are seen as beneficial and stable by the RFC Editor.

2.2 OASIS

2.2.1 Area of Expertise

The OASIS organization is a collection of companies and organizations that develops open standards for global information needs. The organization focuses on web security and e-business standards. Most of their standards are based on the eXtensible Markup Language (XML). A couple of the relevant standards created by OASIS are the Web Services Security Specification (WS-Security), which is a means to apply public key security to Simple Object Access Protocol (SOAP) messages [23] and the standard for Security Assertion Markup Language (SAML) [50], which specifies how to send encryption and security related information in an XML stream. Many of the technologies standardized by OASIS, including WS-Security, underlie and are used in the Department of Defenses Network Centric Enterprise Services.

2.2.2 Process to Create Standards

A new standard must be submitted to a technical committee in order to be considered for standardization. Technical Committees consist of members from industry. Technical Committees edit the submission to produce a standard draft and submit it to a public review. If over half of the present Technical Committee members vote for the standard, the standard becomes a Committee Specification. Then all OASIS members comment and then vote on the standard and require greater than a 2/3 vote for the specification and less than 1/4 vote against it in order for the specification to become an OASIS Standard.

2.3 NIST

2.3.1 Area of Expertise

The National Institute of Standards and Technology develops many standards in many different disciplines of engineering. In computer security, NIST standardized the Data Encryption Standard algorithm in the 1970s and the Advanced Encryption Standard algorithm in 2002 used to encrypt and decrypt data from shared symmetric keys. NIST has not specified any new cryptographic message packing standards lately, instead relying on organizations like the IETF, OASIS, and IEEE to develop them.

2.3.2 Process to Create Standards

Standards are submitted to NIST and are approved in publications called Federal Information Processing Standards (FIPS) publications. NIST requests papers when a standard is needed and reviews all submissions before selecting one to make a standard.

2.4 Related Work

There has been a lot of work done in creating standards used for the storage and transport of cryptographic information. They build upon previous specifications for WS-Security, encryption information packaging schemes, like S/MIME encryption, and cryptographic information messaging standards like the Cryptographic Message Syntax (CMS) [47][35].

2.4.1 Key Management

Key management lately has become an important issue given the number of disparate applications and devices. In order to protect the data in these applications and devices from malicious users, encryption and data signing are necessary. To accomplish this goal, standard ways of managing keys across applications and platform are needed [33]. Several efforts are being made by the standards organizations mentioned earlier to accomplish this goal.

2.4.2 Enterprise Key Management Infrastructure

OASIS has created the Enterprise Key Management Infrastructure Technical Committee in response to the need for standardized key management. The EKMI Technical Committee is involved in developing a Symmetric Key Services Markup Language for key transport. These specifications use XML to provide a standardized mechanism for symmetric key transport.

2.4.3 Guidelines for Cryptographic Key Management

In response to the need for key management, the IETF published RFC 4107 [25]. The specification guidelines on when automated key management should be used and when it is sufficient to manually manage the keys needed in encryption. According to the specification, automated key management should be used when the number of keys needed is n^2 where n is the number of users, or if using a symmetric stream cipher, similar initialization vectors, sending large amounts of data in a short time, or using a key needed by more than two parties. Thus, according to RFC 4107, automated key management is important to support in any cryptographic message packing standard.

2.5 Recent Operational Standards

We now look at the current security standards defined by the organizations mentioned above. The following standards have been created to transport and package cryptographic messages: The S/MIME Encryption Standard, Cryptographic Message Syntax, and Web Services Security standard.

2.5.1 S/MIME Encryption

S/MIME encryption was developed as a way of protecting Multipurpose Internet Mail Extensions (MIME) data with the PKCS 7 secure message format [39]. MIME data format was specified by the IETF and used generally for e-mail messages. The PKCS 7 algorithm and packaging format was generated by RSA Data Security. The specification was changed to use the CMS format, which is very similar to PKCS 7. The S/MIME specification details how to package MIME-type data and specifies how to encrypt and decrypt these packaged messages. S/MIME messages can be encrypted to multiple recipients using their public keys, providing a way to securely share data with a group of participants [39].

2.5.2 CMS Message Formatting

The Cryptographic Message Syntax (CMS) is similar to PKCS 7 as described above and is used to digitally sign, digest, authenticate and/or encrypt arbitrary message data [35]. Data in an S/MIME message is formatted according to CMS. The CMS RFC 3852 defines data structures which hold information on keys and the encapsulated data along with digests and other properties needed to implement public key encryption and digital signing and message digests. It may be possible to extend CMS to encapsulate the types of secure group communications being developed at MIT Lincoln Laboratories. Cryptlib [1] and OpenSSL [13] are examples of the software libraries that implement a subset of CMS.

2.5.3 Web Services Security (WS-Security)

WS-Security standards also define a method to send authentication data needed for public key cryptography [16]. The difference between the CMS format and Web service security format is that CMS defines a format which can be laid out bitwise on a transfer medium. CMS messages also can be encoded in several formats for transfer. In contrast, WS-Security specifies how to secure Simple Object Access Protocol (SOAP) [23] messages in XML. An XML security token format inside a SOAP message defines the authentication information stored within a secured SOAP message. The XML encoding therefore must be used by WS-Security. WSS4J [17] coded by the Apache project [11] is an open-source implementation of WS-Security. SOAP messages using WS-Security could also become the message packaging standard for secure group communications in disadvantaged networks.

2.6 The Choice... and Why

We chose the Cryptographic Message Syntax as the standard on which to focus to develop optimizations for disadvantaged networks. We chose this message standard for several reasons:

- CMS is widely used in the security community. There are many different extensions to CMS, and it supports a wide variety of algorithms and configurations.
- CMS is used in the most commonly used group security protocol, S/MIME [47].
- CMS is also more compact than WS-Security in its native form, which uses base-64 encoded structures and human-readable XML elements, while CMS

uses ASN.1 structures which can be encoded in many ways, to reduce message size.

- CMS has almost become the de-facto basis for sending cryptographic messages in IETF. Applications such as certificate management already use CMS [53].
- OpenSSL, which is used in related work at Lincoln Laboratories, has backported support for CMS.
- CMS has a diverse RFC author distribution, with around 30 contributors and over 30 related RFCs.

We next describe CMS in much greater detail and then we show our design for optimizing CMS for disadvantaged networks.

Chapter 3

Cryptographic Message Syntax

In the previous chapter we selected Cryptographic Message Syntax as a message syntax standard on which to focus our optimizations for disadvantaged networks and offered our reasoning behind that choice. In this chapter we describe CMS in greater detail. First we discuss the original purpose of CMS and its history and current usage. We also examine some of the message structures and types in CMS and their use in securing and authenticating data. Finally, we describe the advantages and disadvantages that CMS presents in the context of disadvantaged networks.

3.1 Purpose and History

CMS is defined in the IETF RFC 3852 [35]. CMS grew out of another standard developed by RSA Laboratories, called PKCS 7 version 1.5 [39], for packaging cryptographic data being sent over electronic mail. The PKCS 7 syntax was made to be easily convertible into Privacy-Enhanced Mail (PEM) [42]. The PKCS 7 syntax was adopted, developed, and maintained by the IETF after its initial development. CMS was developed to encapsulate and protect data transferred over the Internet. The syntax can support digital signatures, digests, key transport, and encrypted message content. The structures and values in CMS are generated using ASN.1 with basic encoding rules and are typically represented as octet-strings [21]. Since its creation, in RFC 3369, CMS was modified to add mechanisms to support more key management schemes and separate the cryptographic algorithms used by the message structure from the makeup of the structure itself [34]. CMS was then later extended by RFC 3852 to support different certificate formats and revocation list formats [35].

3.2 Usage

We now consider the applications of CMS. CMS has become the IETF de-facto standard for cryptographic material transmission, so many related cryptographic protocols generated by the IETF use CMS. The typical usage for CMS has been in S/MIME secure e-mail messaging [47]. This technology provides authentication, message integrity checking, non-repudiation, privacy, and data security to any MIME data [32], and is not limited to just e-mail messages. It can protect MIME encoded data sent via HTTP [31] or other protocols. In S/MIME, enveloped CMS messages are created to provide data security and privacy functions, while signed and authenticated CMS structures are used to provide integrity, non-repudiation, and authentication. Due to the flexibility of CMS, new algorithms for key management, key wrapping, signing, and encrypting data in CMS are easily supported without requiring changes to the base CMS structure.

We have already seen several follow-on RFCs which relate to CMS. RFC 2797 [43] established a method of passing Public Key Infrastructure (PKI) [46] information through the use of CMS messages. The protocol describes an interface to exchange certificates. Requests for certificates are first made using PKCS 10 objects. Then, responses to requests for certificates are created using signed CMS structures. This protocol is then later updated in RFC 5272 through 5275. In RFC 5272, PKI requests and responses are both wrapped in CMS structures [54]. RFC 5273 provides file extensions which correspond to PKI requests and responses and methods to use PKI requests and responses encapsulated in MIME data, HTTP/HTTPS [31] protocol data, and TCP-based data [55]. RFC 5274 deals with terminology [53] and RFC 5275 describes a symmetric key management protocol and architecture. The symmetric key management protocol is created using PKI Requests/Responses encapsulated by

CMS enveloped or signed structures [56]. RFC 4108 describes the use of CMS to protect firmware packages in transmission [36].

As the reader may see, CMS is quickly evolving and growing to enable cryptography on the Internet. The open source cryptography library, OpenSSL has even included CMS into its message packaging [13]. Also, it is fairly easy to incorporate new algorithms into CMS, potentially including those used by Lincoln Laboratory's secure group communications project. This makes CMS an appropriate standard to focus on in developing optimizations for disadvantaged networks.

3.3 Message Structures

Having described the different applications of CMS, we now consider the content and syntax of CMS messages. The structure of CMS is fairly simple, but looks drastically different depending on the type of message.

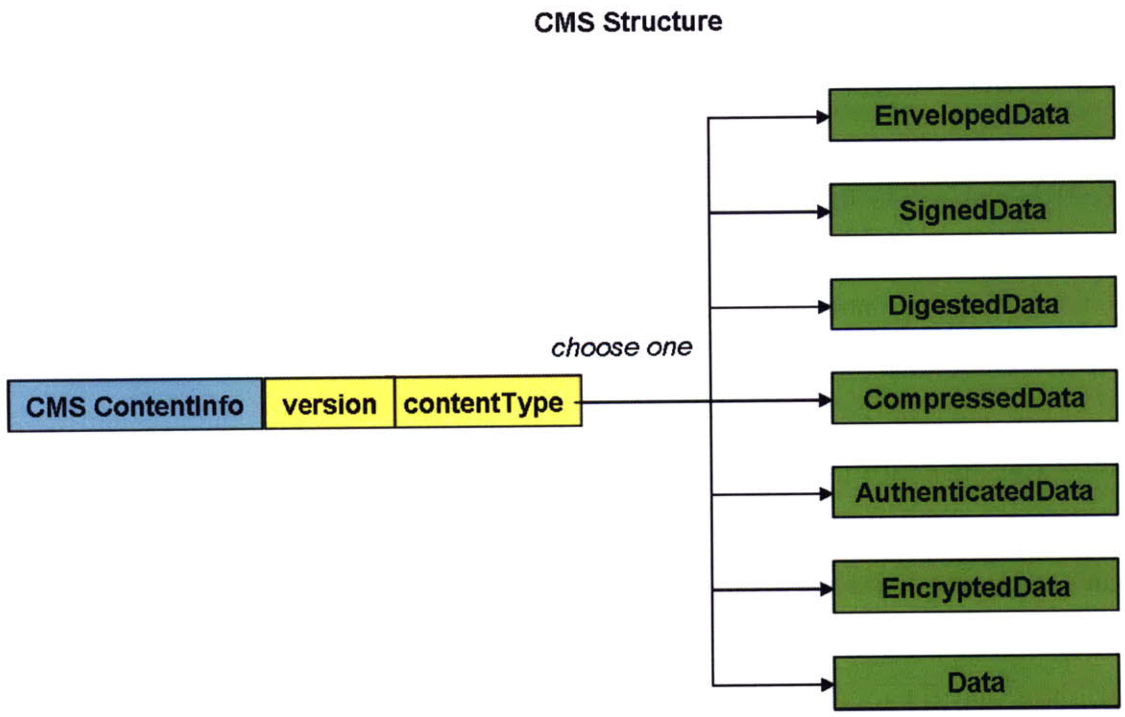

Figure 3-1: A CMS message: the content type defines the structure of the rest of the message

The top level element in every CMS message is the `ContentInfo` element, which simply contains the version number of the CMS structure followed by an object which defines the content type of the structure. This content type defines the data as a CMS structure and informs anyone processing the data what to expect next. All elements in the CMS structure are defined as ASN.1 objects and thus can be encoded/decoded just like any other ASN.1 objects [35]. Figure 3-1 shows the top level structure.

There are several different content types available in CMS: `SignedData`, `EnvelopedData`, `DigestedData`, `EncryptedData`, `AuthenticatedData`, `Data`, and `CompressedData`. More content types can be added to CMS by writing a new RFC to update CMS. In general, the content types all permit an entity to process the content in a single pass using Basic Encoding Rules (BER) [21].

`SignedData` takes arbitrary encapsulated content and applies an arbitrary number of signers to the content. Usually `SignedData` transmits digital signatures or PKI information that needs to be transmitted with integrity and authentication [46]. `EnvelopedData` content encrypts the content of the package with a content encryption key (CEK). This key is then wrapped via a specified algorithm to any number of recipients. This data type is typically used to digitally envelope data in schemes such as S/MIME [47]. The `DigestedData` type simply consists of any type of content which includes a message integrity digest calculated with a specified algorithm. The digest is computed only on the content of the structure and a recipient can verify the content by independently calculating a digest with the same algorithm on the same content that the message contains. `EncryptedData` is similar to enveloped data except there is no CEK included in the package. Key management with `EncryptedData` packages must be done through some other means. `AuthenticatedData` encrypts a message authentication key to any number of recipients, which is used to verify a message authentication code made from the content of the package as well as any other attributes that the user chooses to authenticate. Finally, `Data` content in CMS usually encapsulates arbitrary data being sent over the wire and itself is usually encapsulated by other CMS content types.

3.3.1 Encapsulation

One of the main concepts in CMS is encapsulation. An unlimited number of types of structures can be made by creating one type of CMS structure and then encapsulating it in another. For example, let's say that you wanted to take a text file and sign, then encrypt it in CMS for transmission to another entity. First, the text file would be described by `Data` content, which is essentially an octet-stream. This data content would then be encapsulated in a `SignedData` structure where the signature would verify the ASCII text encoded in the octet-stream. Finally, the whole `SignedData` structure would be encapsulated in an `EnvelopedData` structure, which would have CEKs for all recipients of the data. Encapsulation allows data to have multiple attributes and many cryptographic operations completed on the same piece of data in large nested CMS structure. Applications can process CMS structures until they reach a point where they do not have the necessary permissions to view further, and chains of signatures or encrypted packages can be created to ensure confidentiality of a message or authentication by multiple entities. The structure therefore, integrates very well into PKI.

3.3.2 Used Content Types

The main content types in CMS are the `EnvelopedData` content type and `SignedData` content type. `EnvelopedData` can be used for encrypting information while including in the same package the content encryption key needed to decrypt the message. This content encryption key can be encrypted to multiple recipients via several key management schemes. This makes `EnvelopedData` a flexible structure for encryption. `SignedData` is used for date authentication. Also, as discussed in 3.2, PKI requests and responses use `SignedData` packages in order to authenticate certificates and other PKI information. Let's look in more detail at these two structures.

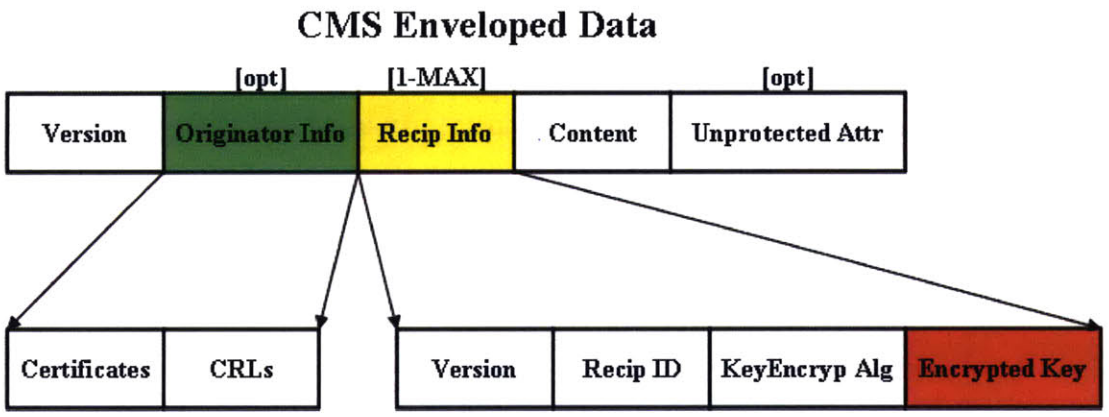

Figure 3-2: `EnvelopedData` structure laid out in memory.

3.3.3 Enveloped Messages

`EnvelopedData` contains encrypted content, optional certificates or certificate revocation lists, and unprotected attributes attached to the structure on the top level. All key management is done in the `RecipientInfo` set of data elements. There is one `RecipientInfo` for every entity the message is encrypted to. Each contains the key which encrypts the message content, which itself is encrypted according to a specified key management scheme.

3.3.4 Signed Messages

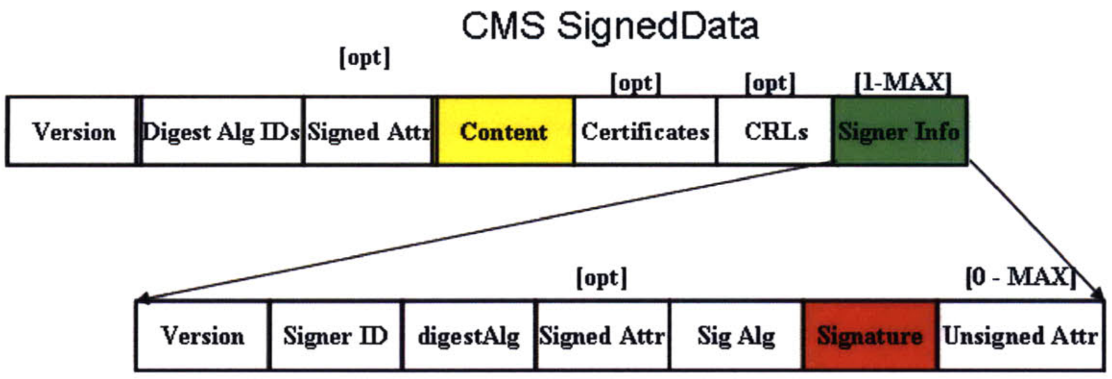

Figure 3-3: `SignedData` structure laid out in memory.

SignedData at the top level contains the encapsulated content, attributes, a list of digest algorithms used by the structure, and optional certificates or revocation lists. All the signatures are contained in SignerInfo elements. One SignerInfo is created for each signature added to the structure. These elements also contain optional signed attributes as well as specifiers for the digest and signature algorithms used in the signature.

3.4 CMS Usage in Disadvantaged Networks

CMS can be used in disadvantaged networks to package secure communications. The EnvelopedData structure could be used to encrypt messages to groups of entities while the SignedData structure could be used to send authenticated information over the networks. CMS has an advantage in in that it is generic and descriptive. These characteristics allow it to support many different types of algorithms and content types. New algorithms are supported easily using the CMS syntax, making it extensible. This is an advantage since new encryption algorithms may need to be developed for use on disadvantaged networks. Also, since CMS requires no specific transfer method or lower level details, it is interoperable. However, a disadvantage with CMS is that these structures can add a significant amount of message size overhead. In this thesis we investigate and propose methods for how to reduce this overhead.

Chapter 4

Optimizing CMS for Disadvantaged Networks

In the previous chapter we examined Cryptographic Message Syntax in detail, focusing on a couple of message structures within the syntax, the `EnvelopedData` type and the `SignedData` type. These message types can be considered the main ones in CMS and they are useful in packaging group secure information in the Lincoln Laboratory Dynamic Group Keying project. However, there are significant problems toward adopting CMS as the standard for packaging group data in disadvantaged networks. First, CMS must have a method for supporting the algorithms which are used in disadvantaged networks. Second, CMS must not introduce too much overhead data into each message. This is because a disadvantaged network can not reliably send large quantities of data quickly. Bandwidth, latency, and connectivity may all be limited in a disadvantaged network.

In this chapter, we present strategies for eliminating overhead when sending CMS-packaged data. We first determine our methods of evaluating CMS and any optimizations we develop. Then we look into the sources of overhead in CMS. Finally we delve into the potential methods of reducing this overhead and explain the advantages and disadvantages of each method.

4.1 Methods of Evaluation

There are several ways of evaluating the efficiency of a cryptographic message packing method. Two of the different methods are listed below:

- **Computations:** Estimates the number of computations that the processor would have to make in order to create the message.
- **Message Size:** Evaluates the overall efficiency of a structure based on the amount of data that is sent from entity to entity.

We chose to evaluate the efficiency of Cryptographic Message Syntax based on the message size of the syntax. We assume that the computation power needed to encrypt, decrypt, sign, and create mesages would cause delays far less than the delays caused by the latency and bandwidth of the disadvantaged networks that send that data. This is because our environment may include links with less than 4096 bits/sec, while the links may connect computers that have 2 or more cores operating at greater than 2GHz. Also, the power required for communications is generally greater than that required for computations. Thus, processor cycles can be relatively free compared to bandwidth on the network.

4.2 Sources of Overhead in CMS

This section details the sources of overhead in the CMS structure. The documentation for CMS shows how the structure is laid out, but does not tell us much about how large each structure is. These questions can be answered through some testing. We ran a few initial tests on simple CMS structures with varying byte payloads which showed where the sources of overhead were. Figure 4-1 shows the sizes of sample `EnvelopedData` and `SignedData` structures as they would appear in transmission between two entities. Figure 4-2 then shows the CMS overhead data in each of these messages.

There are several options available to create an `EnvelopedData` package. In OpenSSL, CMS structures can pack data using either binary data or text format.

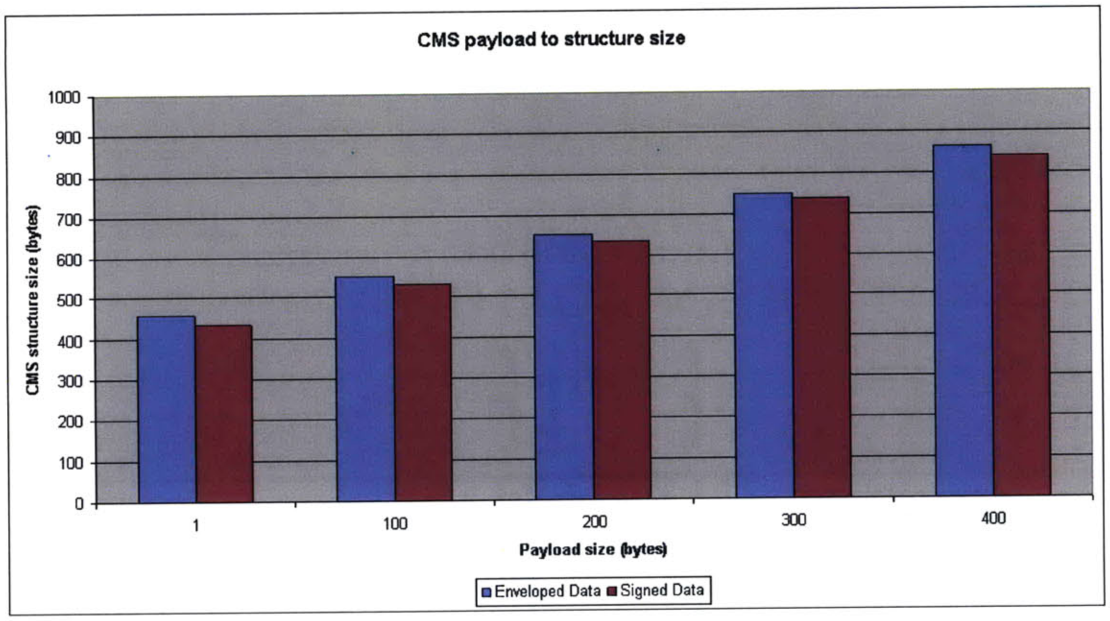

Figure 4-1: `EnvelopedData` with AES-256-cbc CEKs and 2048 bit RSA encryption keys with differing payloads

They also can have many different types of encryption and include attributes that are signed, encrypted, or unmodified. Certificates and CRLs can be added to the structure as well. For our purposes, we use binary data since CMS by default does not package data in text format. As a default, in our test `EnvelopedData` we encrypt a message with 1 byte of payload to one user. We use a CEK which is encrypted to each user with RSA Encryption [49] using X.509 certificates [37] and 2048 bit keys. The CEK encrypts the packaged data with an AES 256-bit cipher [44].

Our `SignedData` packages have 1 signer using an RSA X.509 certificate to digitally sign the encapsulated data. The package has a 1 byte payload of binary data again with no signed attributes or S/MIME Capabilities and no certificates attached so we can look at the bare minimum `SignedData` package.

As you can see in 4-1, for only 1 byte of payload, there is a considerable amount of overhead, 450 bytes of CMS-related data. However, as the size of the data increases, the overhead related to the CMS structure stays the same, which results in a linear increase in CMS package size with a linear increase in payload size. The CMS overhead remains fixed no matter the payload. This makes sense since the package does not

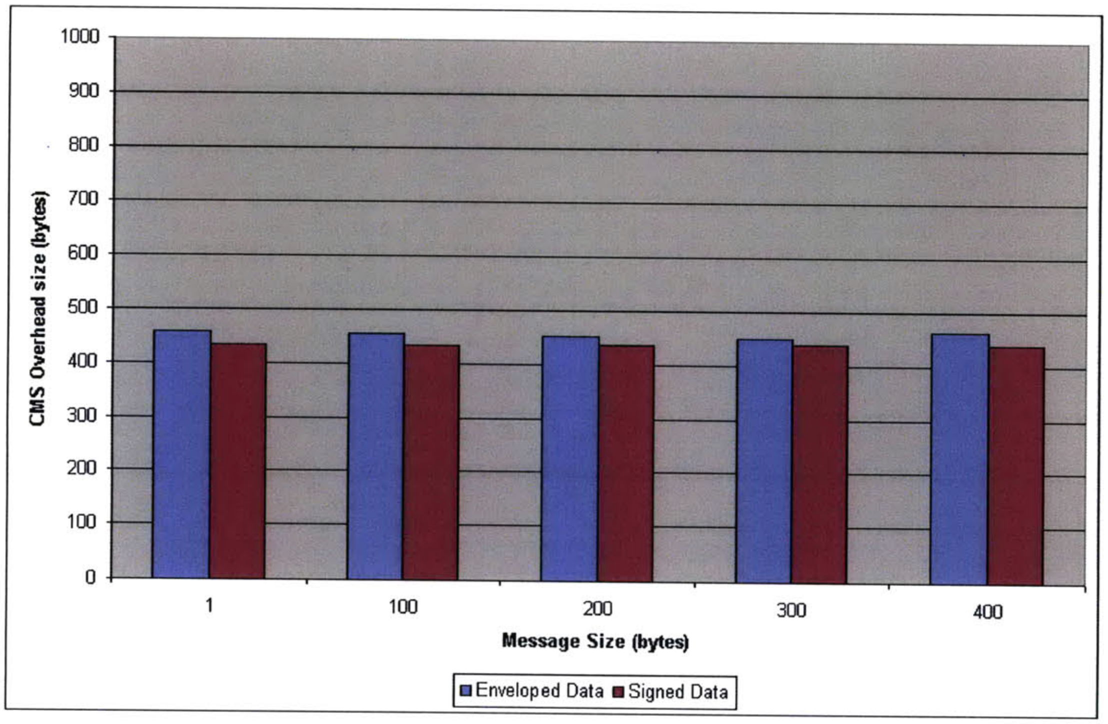

Figure 4-2: `EnvelopedData` overhead without payloads included

depend on the data included in the package.

In Table 4.1 we show the size breakdown for each CMS component.

EnvelopedData		SignedData	
`version`	2	`version`	2
`contentType`	6	`contentType`	6
`recipientInfo`	377	`signerInfo`	375
`encryptedContentInfo`	73	`encapContentInfo`	18
		`algorithms`	9

Table 4.1: `EnvelopedData` and `SignedData` message breakdown

We can see that most of the overhead in the CMS package comes from the `RecipientInfo` structure and the `EncryptedContentInfo` structure. In the `SignedData` structure, most of the CMS-related overhead is in the `SignerInfo` package. The `RecipientInfo` and `SignerInfo` structures are largest because they contain the variable CEK and signature. The signature algorithms and content specifiers are fixed data structures and are relatively small. We also show the size breakdown for each structure when we eliminate the data accounted for by the CEK or signature. The overhead in the `RecipientInfo` and `SignerInfo` structures still outweighs the overhead outside these structures.

4.2.1 Specification of Algorithms

One of the sources of overhead is in the specification of algorithms in both of these data structures. `SignedData` has a set of digest algorithms specified in the top level so that a processing program can determine if it supports all the digest algorithms used in the set of `SignerInfo` structures. Each `SignerInfo` corresponds to one signature. Thus each `SignerInfo` has a signature and digest algorithm specified. Each digest algorithm takes around 10 bytes of space in the structure. Each signature algorithm takes about 10 bytes of space as well. In the `EnvelopedData` structure, the `EncryptedContentInfo` specifies an encryption algorithm which uses 10 bytes of space, and more if initialization vectors (IVs) are needed, like the IV needed by AES in cipher block chaining mode [44]. For each recipient the CEK encryption algorithm is specified through one of the key management methods.

4.2.2 Content Encryption Keys

Much of the extra data in CMS packages is used by the CEK sent to each recipient of an encrypted package. This extra data is not overhead because it is needed to decrypt the encapsulated content. The more recipients that an `EnvelopedData` structure has, the more CEKs that will be generated when using S/MIME. This is because S/MIME dictates that a CEK be sent to each recipient separately so that each recipient can use his own private key to decrypt the data.

The most expensive (in size) method of key management in CMS is the *Key Transport* mode where X.509 public keys are used to encrypt the CEK separately to each recipient. These keys are generally large since they are asymmetric keys (2048 bits or greater). In the other key management schemes, *Key Agreement*, *Password*, and *Key Encryption Key*, the content encryption keys are smaller as they use symmetric keys.

4.2.3 Formatting and Encoding

CMS uses ASN.1 structures [21] in order to describe all of its elements. ASN.1 structures offer flexibility in how a piece of data is represented and in how the structure can be encoded to pass to other entities. The normal encoding for CMS is Basic Encoding Rules (BER). This encoding has two subsets, Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER). BER encoding defines each data element as a type identifier, a length description, and then the actual data content which may be ended by an end-of-content marker. This format allows a receiver to obtain the necessary information about a data element without any pre-existing knowledge about the structure and without the full stream of data. The CMS data may also be DER encoded, which differs from BER only in that the length specifications for each value must be made in one way; i.e., no end-of-content marker exists.

4.2.4 Optional Data

Finally the multitude of optional items in CMS structures can add much extra data to the package. These options are included in CMS to support situations when no form of key management infrastructure are in place or when the sender wants the receiver to be able to use the CMS messages without any other outside information, or to transport authentication material such as certificates and certificate revocation lists. For example, in the `EnvelopedData` structure, the sender may include information about itself including certificates which may be necessary to verify signatures.

The structure also allows for an undetermined number of unprotected attributes. The SignedData structure supports optional signed and unsigned attributes for each signature as well as certificates or certificate revocation lists. All these options add extra data to the CMS structure, but are not mandatory for the base functionality of CMS.

4.2.5 Potential Solutions

As mentioned in 4.2 the major sources of overhead in the CMS package are generally from specifying algorithms, formatting and encoding the data, adding cryptographic material necessary to sign and/or encrypt the encapsulated data, and allowing for flexibility by providing means to add optional data such as timestamps, certificates, and certificate revocation lists. Solutions to the overhead problem must be able to recreate CMS structures without disturbing the integrity of the data or fundamentally altering the algorithms and methods necessary to perform cryptographic operations on the data.

4.3 Requirements

There are several requirements to any solution which attempts to reduce overhead in CMS.

1. *Recoverable*

   When a CMS package is transformed in order to reduce its overhead, the package should then be able to be completely restored to its original state. This is the most important rule as our encryption, signing, and authentication will not work without this guarantee.

2. *Flexible*

   Any solution should work with the main structures available in CMS. Also, algorithms which are supported by CMS should continue to be supported by the solution. Variable numbers of recipients and signers should also be supported. While not all algorithms need to be supported, a solution should enable the passing of cryptographic data associated with different algorithms. For example, AES with cipher block chaining requires initialization vectors and Diffie-Hellman algorithms require passing of public information.

3. *Extensible*

   When the CMS structure is extended via RFCs, minimal effort should have to be made to make the solution support the new changes. From the past 10 years there are 55 RFCs with CMS in the title. This means that the syntax is being changed as it finds new uses and one of its strengths is its extensibility. Our changes should take this strength into account.

4. *Simple*

   The more complexity involved in creating a solution to reduce overhead, the more opportunity for error the solution can provide.

With these thoughts in mind, we attempted to find and evaluate a few solutions to reduce the overhead introduced by CMS.

4.4 General Purpose Compression to Reduce Overhead: ZLIB

In the previous sections, we described the sources of overhead in a CMS structure. This section details one potential solution which will reduce the size of a CMS structure. Here we propose using ZLIB [29] compression on the CMS structure in order to reduce its overhead. We then detail our thoughts on how to best compress CMS structures.

4.4.1 ZLIB Overview

ZLIB is an algorithm that provides lossless compression [29]. It is an open standard with free source code, and was already used in libraries we depend on, like OpenSSL. Let us look at the inner workings of ZLIB so we can understand how it applies to compressing CMS.

Background

ZLIB was developed as a lossless compression format that would be

- Independent of the other parts of the system it was used on, including CPU, operating system, file system, etc.;
- Efficient in compressing data compared to the best compression algorithms;
- Free from patents so the algorithm could be used freely;
- Usable with the previously developed gzip file format [29].

The Structure

A ZLIB encoded structure is described by a series of blocks which correspond to substrings of the data. Each of the block sizes have variable length under 65535 bytes. Each block compressed into literal strings with distance pairs using LZ77 [57] and Huffman coded [38] to create a Huffman tree. For each block of compressed data, the block may reference data which occurs in a previous block up to 32KB before the current position. Each block has two parts, Huffman code trees that describe the compressed data, and the actual compressed data. Each of the Huffman code trees also is compressed using Huffman coding. The compressed data is represented by a series of elements based on two types. The series is listed in the order that it appears in the structure. The first type is a *literal* which is simply a string of bytes which has not been duplicated in the previous 32KB of data. The next type is a *pointer* which is a represented as a pair `<length, distance>` which describes the length of the literal string and the distance (maximum 32KB) backwards in the structure to that literal. The length of a literal is limited to 258 bytes and the length of an output block is limited to 8KB under the C implementation [28].

Deflate/Inflate Processing

So how does ZLIB compress? ZLIB calls compression a *deflate* operation and decompression an *inflate* operation. As the compressor parses through the data, it determines when to start new blocks of data, which is when the buffer becomes full or when it determines that having new Huffman trees would be useful. For every input block, the compressor looks at 3 byte sequences and writes it out to the output block if it has not been seen. That sequence is written to a hash table for lookup later on. If the sequence has been seen, then it will be in the hash table and the compressor writes a pointer to the nearest previous 3 bytes and the total number of bytes (length) that are the same. This compression process is shown in Figure 4-3. To improve compression, as the compressor finds sequences in the hash table, it will make extra passes through the previous data in an attempt to find a longer matching sequence. This is called *lazy matching* by the ZLIB authors. The algorithm can be configured to spend less time trying to find longer matches to improve speed of compression at the cost of a worse compression ratio [28].

Interface

Deflate		Inflate	
Basic	Advanced	Basic	Advanced

`deflateInit`
`deflate`
`deflateEnd`
	`deflateInit2`
	`deflateCopy`
	`deflateReset`
	`deflateSetDictionary`
		`inflateInit`
		`inflate`
		`inflateEnd`
			`inflateInit2`
			`inflateSync`
			`inflateReset`
			`inflateSetDictionary`

Table 4.2: Core ZLIB API pertinent functions

The interface in Table 4.4.1 defines a `z_stream` structure which stores the compressor/decompresser data needed for compression. A simple example of using the ZLIB interface would be to initiate the stream with `deflateInit`. Then the data would be fed into the `z_stream` structure and processed with `deflate`. The stream is closed with `deflateEnd`. In order to inflate the data, a user would use `inflateInit`, then `inflate` and `inflateEnd`. The API offers options for the compression level from `Z_NO_COMPRESSION` to `Z_BEST_COMPRESSION` and also when to flush the data. Using the functions `deflateSetDictionary` and `inflateSetDictionary` a pre-placed dictionary of literals may also be input to prime the compressor [24]. Table 4.4.1 shows the low-level functions available to the API users.

Our Takeaways

We concluded that ZLIB offers a simple way of compressing data while maintaining *recoverability* and also offers tunable compression on different data sets using the dictionary option. We next look at how ZLIB could be applied in the context of a disadvantaged network. The following sections describe our studies into the best way of using ZLIB to compress CMS data.

4.4.2 Optimal Conditions: Reliable TCP with Repeated CMS Structures

The first exploration into using ZLIB we use an optimal situation, assuming a user was sending the same CMS structure over the network with no network slowdowns or outages. The performance of this test is shown in 6.3.3. The only difference in the messages was the actual data sent in the package, which was set to random bytes of data each time. We use ZLIB with its best compression and partial flushing of the data for each CMS structure. When partial flushing is used, ZLIB maintains the 32KB of previous data from which to compress the data. When CMS structures are repeated with the same senders and recipients, most of the CMS overhead will be repeats of the same information. Therefore, ZLIB can compress this data out. Only the random encapsulated data would be uncompressable. This test served as a lower limit baseline for our use of ZLIB as real-world usage of CMS may not replicate this scenario.

4.4.3 Worst Case Conditions: Disadvantaged Networks

The second case we investigated was the worst case condition. Results for this test are discussed in section 6.3.3. In this case, we assume that there is no reliable uninterrupted stream of data coming from the network. Assuming CMS structures smaller than one TCP or UDP packet, the ZLIB compression structures may need to be reset many times because the connection between entities is unreliable. To simulate this condition, we compressed every message separately. This assumes the compressor can not utilize the previous data in the stream and represents the worst case when every message must reset the ZLIB stream.

4.4.4 Priming the Pump: Generating a CMS ZLIB Dictionary

The first two studies with optimal conditions and worst case conditions set the upper and lower bounds for what kind of compression we could expect to generate from compressing CMS messages. However, we would like to achieve near the compression level similar to the one on the optimal network while being on an unreliable network.

Strategy

Our first solution to the problem was to preplace a static dictionary of commonly used literals in CMS in the ZLIB stream in order to simulate ZLIB compressing a number of CMS messages before sending each message. This way, the ZLIB stream can be 'primed' with the CMS literals before compressing the new message to be sent. This method improves compression toward what we get when there is optimal conditions while assuming that compression can not rely on previous messages sent on the network.

This strategy relies on how ZLIB creates compressed data. As mentioned in 4.4.1 ZLIB has a memory buffer which stores the last 32KB of data compressed by the structure. In optimal conditions, the last 32KB contain previously sent CMS structures which can be referenced for compression gains. When we preplace a dictionary in ZLIB, the dicitonary's strings fill some of the memory buffer and provide a reference that ZLIB can use to better compress the new message. However, since the memory buffer is only 32KB, the dictionary can only be up to 32KB in size. Any larger and it will not be able to be referenced when compressing the new message. The side effect of the limit on dictionary size is that we must choose the literals to put into the dictionary carefully.

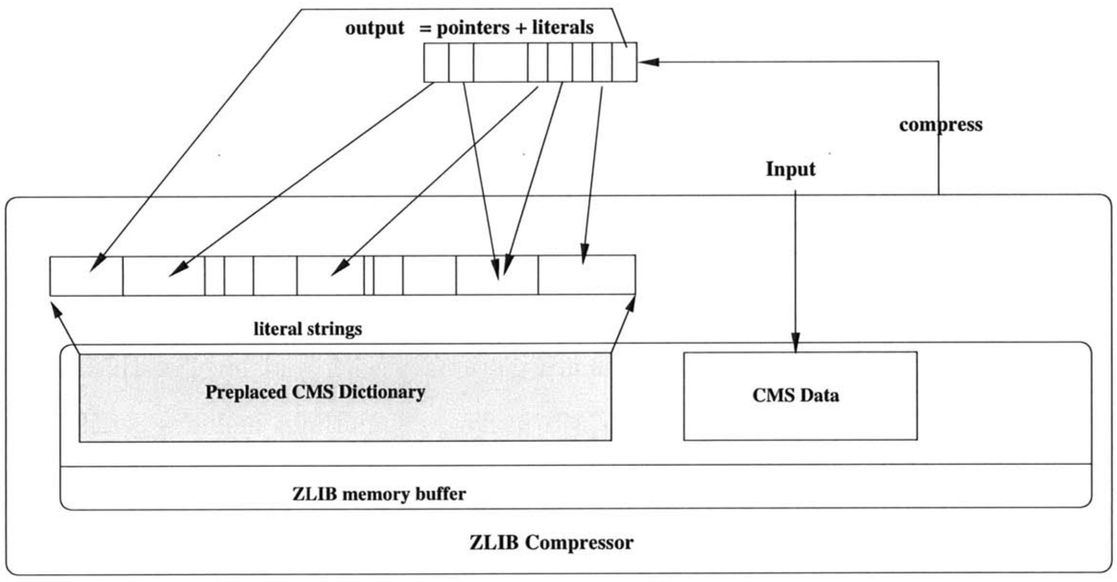

Figure 4-3: *ZLIB is preplaced with a dictionary full of literal strings which contain common CMS data. The compressor can then reference those strings when creating the output and use pointers rather than copying the original data.*

Assembling a Dictionary

We attempted to find an optimal dictionary for compressing CMS structures, focusing on the larger structures (Enveloped/Signed). The first approach we took was to simply create every type of CMS message supported by OpenSSL and DER encode them. Then we concatenated all those messages together as the dictionary. Next we looked at how we could reduce the size of this dictionary by writing only the relative constant data in all the CMS structures to the dictionary. Our assumption was that any random-appearing or user-specific data in the CMS structure would not compress out of the structure on average as these strings would not appear in another CMS message.

The other method by which we attempt to reduce CMS overhead is by including certificate information. As part of `SignedData` and `EnvelopedData` messages, as mentioned in section 3.3 there are options to include both certificates and certificate revocation lists. These pieces of data are sent so that recipients know which certificate is being used to sign or encrypt the data. We created a dictionary made from a series of commonly used root-level certificates using the pre-installed root certificates from the Mozilla Firefox web browser [7] and DoD root certificates which may commonly be used on DoD networks. In CMS, most messages will include user certificates as root-level certificates are many times installed into hardware through other means. However, user certificates are signed by higher-level certificates to establish trust, and thus, most will have issuer names which may be included in root-level certificates. This common data would compress when a dictionary with root level certificates is used. Also, optional data included in CMS packages sometimes includes certificates and certificate identifiers in CRLs. By including commonly used certificates in the dictionary, these certificates will cost negligible space in transmission.

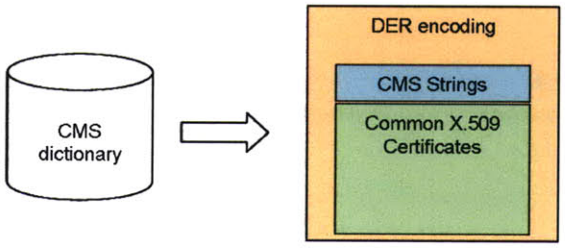

Figure 4-4: Composition of one CMS dictionary. We also assembled dictionaries with full CMS structures and dictionaries with certificate identifiers for commonly-used certificates added to the dictionary.

Finally we found that identifiers for certificates are commonly sent in CMS messages so that recipients know which certificate is being used to sign or encrypt a message. We added this identifier information for all the certificates we were testing CMS with in order to compress this information.

We then concatenated the CMS dictionaries with the certificate dictionaries to test their effectiveness compared to the worst case and ideal compression. The results are shown in section 6.3. We show the different dictionaries tested in Table 4.3.

Compressing and Decompressing

The compression and decompression procedures are shown in Figures 4-5 and 4-6.

The CMS structure created is encapsulated by a ZLIB structure which contains references to data in the CMS dictionary. Thus, during decompression, the same

Dictionary Id	Description
CMSStringDict	CMS Shortened Strings
CMSFullDict	CMS Full Structures
CMSFullCertDict	CMS Full Structures + Root Certs
CMSStringCertDict	CMS Strings + Root Certs
CMSFullIssuer	CMS Full Structures + Dummy User Certificate Issuer and Serials
CMSStringIssuer	CMS Strings + Dummy User Certificate Issuer and Serials

Table 4.3: Tested dictionary compositions

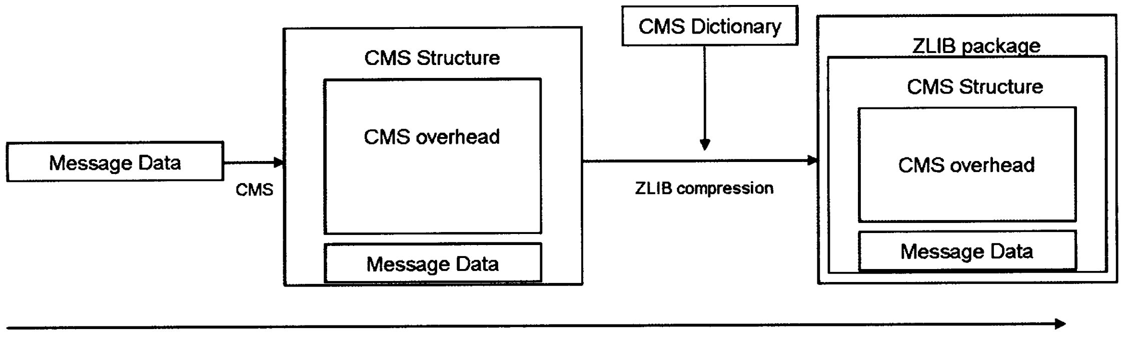

Figure 4-5: Compressing a CMS structure with a ZLIB dictionary

dictionary is needed on the receiver side. This creates a small problem with decompression. The recipient must have the preplaced CMS dictionary in order to decompress the data, but if the dictionary is used, currently in CMS there is no way of determining which version of the dictionary to use.

This problem with the dictionary can be solved by wrapping the ZLIB structure in a CMS `CompressedData` structure and supplying it with a hash of the dictionary data to use. The CMS parser with the recipient can then have a set of dictionaries and compute the hash of each dictionary against the hash in the message in order to determine which dictionary to use. The ASE authenticated data transmission system may be used to exchange dictionaries in order optimize this procedure for disadvantaged networks [1].

Alternatively, for a solution using less overhead, the `CompressedData` structure could add an integer or name which acts as a universal resource identifier (URI) that

[1]Benjamin W. Fuller, Roger I. Khazan, Joseph A. Cooley, Galen E. Pickard, and Dan Utin. ASE: Authenticated Statement Exchange, Submitted for publication, 2009.

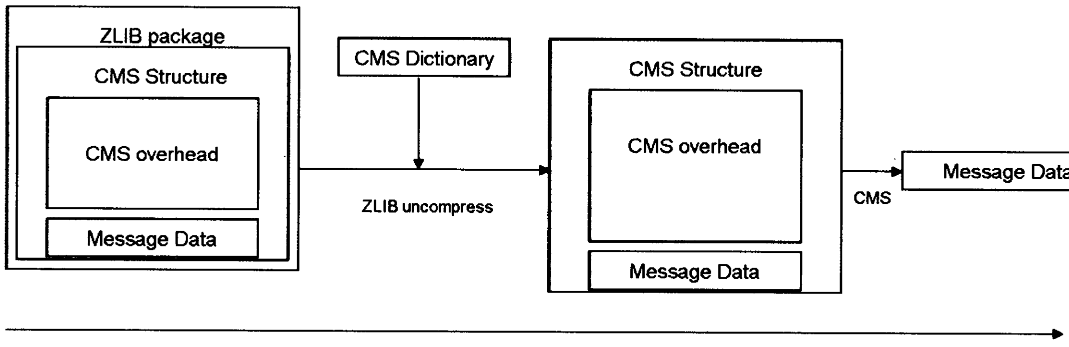

Figure 4-6: Decompressing a CMS structure with ZLIB

uniquely identifies the dictionary to use in decompressing the message. Both of these data elements could be optional parts of the `CompressedData` structure to be included if a dictionary is used in compression.

We chose to use the MD5 hash solution and augmented the CMS `CompressedData` structure with a string which stores the path to the dictionary file, an octet string which stores the hash of the dictionary, and an algorithm identifier for the digest used in the hash.

4.5 Content-Aware Compression to Reduce Overhead: *CMS Lite*

We previously described how ZLIB compression reduces the overhead of CMS. This method utilized a preplaced dictionary with CMS messages. We also explored another way to reduce CMS-related overhead. We hoped that a custom translation of the CMS structures to optimized structures could take advantage of our knowledge of the structures of CMS. We reasoned that we should be able to create our own CMS-specific compressed data type using lookups to replace data to compress messages. Thus, we created CMS Lite. This version of CMS is a shortened and compressed version of CMS which can be used to transfer CMS data over a disadvantaged network. CMS Lite can then be inflated back into a normal CMS structure when the data is unpacked by a recipient.

4.5.1 CMS Lite Design

The Lite version of CMS requires more intimate knowledge of the inner structures in CMS than the ZLIB-powered generic data compression. In order to create a CMS Lite structure, Lite versions of each subtype of CMS structure has to be created. In these Lite versions, content type descriptors and algorithm descriptors are referenced by a table lookup from a single integer value.

To limit the scope of what CMS Lite could compress, we put further limitations on how the structures could be created for the Lite type. The number of algorithms supported by this structure becomes more structured compared to the normal CMS type. If there is an algorithm used that is not included in the lookup, then the Lite structure can not be created and the original is preserved.

Next, we flattened structures in CMS, bringing the important data out of nested structures so that more overhead would not be used to describe the inner structures. Lite structures also can require specific configurations and key management structures rather than have multiple nested ASN.1 structures to support any configuration so that more data can be saved.

Finally, we found that identifiers for certificates could be hashed instead of sent whole. The hash of the identifier can then be compared to hashes of certificate identifier information rather than the full identifiers in order to recreate the original messages. Similarly, a prefix of the identifier could be used in order to reference that certificate.

4.5.2 Encode and Decode

CMS Lite is designed as a transfer data type only, which is similar to compressed data types. Thus, no CMS API functions can be called using the CMS Lite data structures even though Lite structures may resemble their normal CMS counterparts. Figure 4-7 shows a possible use of CMS Lite. The structure is meant to be the transfer structure for the CMS data, which is then reverted to the original CMS structure.

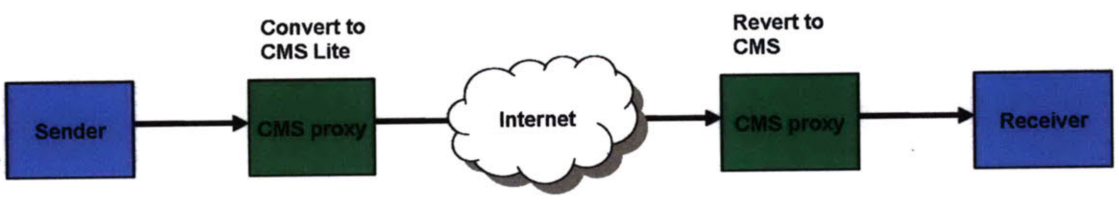

Figure 4-7: CMS Lite encode/decode

4.5.3 CMS Lite Transformations

For CMS Lite we define a `LiteData` content type at the same level as all the other CMS content types. It contains a choice of parameterized lite versions of each of the other CMS content types. Its ASN.1 structure is shown below in Listing 4.1.

```
// CMS LiteData includes a choice of parameters representing
// All of the different CMS contentTypes
CMS LiteData ::= CHOICE {
  signedParams CMS_Signed_params,
  envelopedParams CMS_Enveloped_params,
  digestedParams CMS_Digested_params,
  encryptedParams CMS_Encrypted_params,
  authenticatedParams CMS_Authenticated_params,
  compressedParams CMS_Compressed_params
}
```

Listing 4.1: CMS Lite

The transformations and changes for each CMS `contentType` are detailed below:

CMS `EnvelopedData` Transform

In order to reduce overhead in EnvelopedData structures, the `encryptedContentInfo` is flattened by taking out the encrypted data and `contentEncryptionAlgorithm` and the `contentType` and key management encryption algorithm (used to encrypt the content encryption key) are defined by the parameter. The `encryptAlg` information stores the initialization vector and algorithm information for the content encryption key which is used to encrypt the data. `OriginatorInfo` (certificates and CRLs) can be sent with the data as an option, but will not be compressed in the CMS Lite structure. The updated structures are shown in Listings 4.2 and 4.3.

The key management schemes all have parameter versions as well. In general the parameterized versions insert a lookup for any static algorithm identifiers and also instead of using the full certificate issuer name and serial number or subject key identifier to identify any certificates, uses a hash of that data instead.

One limitation is that since the parameter can only give one key management key encryption algorithm, only one type of key encryption algorithm can be used for all recipients, making the Lite type a little more restrictive than the normal CMS `EnvelopedData` type. As seen in Listing 4.3 the `recipientInfos` do not have encryption algorithms defined. They instead all use the parameter in the `Enveloped Params` structure to define the algorithm instead. The algorithms used are necessarily limited to ones defined in the lookup.

```
EnvelopedData ::= SEQUENCE {
        version CMSVersion,
        originatorInfo [0] IMPLICIT OriginatorInfo OPT,
        recipientInfos SET SIZE (1...MAX) OF RecipientInfos,
        encryptedContentInfo EncryptedContentInfo,
        unprotectedAttrs [1] IMPLICIT UnprotectedAttributes OPT
}
RecipientInfo ::= CHOICE {
        ktri KeyTransRecipientInfo,
        kari KeyAgreeRecipientInfo,
        ...
}
KeyTransRecipientInfo ::= SEQUENCE {
        version CMSVersion,
        rid CMSSignerIdentifier // issuer/serial or subjectkey
        keyEncryptionAlgorithm X509_ALGORITHM
        encryptedKey OCTET STRING
}
```

Listing 4.2: Normal `EnvelopedData`

```
Enveloped_params ::= SEQUENCE {
        version CMSVersion,
        recipientInfos SET SIZE (1...MAX) OF Recipient_params,
        parameter LONG,
        encryptAlg X509_ALGORITHM,
        eContent OCTET STRING
}
Recipient_params ::= CHOICE {
        ktp KeyTransParams,
        kap KeyAgreeParams,
        kekp KEKParams,
        pwdp PasswordParams
}
KeyTransParams ::= SEQUENCE {
        version CMSVersion,
        sidHash LONG, //issuer/serial or subjectkey ID hash
        type LONG, // tells whether sidHash is issuer/serial or
           subjectkey
        encryptedKey OCTET STRING
}
```

Listing 4.3: Reduced EnvelopedData

CMS SignedData Transform

In the SignedData structure, we eliminate the OPTIONAL certificates and CRLs to be sent with the data. The encapsulatedContentInfo is flattened with the contentType of the encapsulated data then identified by the parameter. The content is placed in eContent. The set of digest algorithms in the normal type can be eliminated as it is normally included for one-pass processing efficiencies, which we gladly give up for less space usage. These updates and the original structures are shown in Listings 4.4 and 4.5.

In the set of SignerInfo we replace digest algorithm and signature algorithm identifiers with a lookup table parameter. Then we hash the signerIdentifier once more to reduce its overhead.

There are a few limitations introduced by the CMS Lite transformation. The algorithms used for signature and digest must be in the lookup table. Also certificates and CRLs should be sent via another mechanism. Signed attributes can be included in the `SignerInfo` `Params` structure but we choose not to implement including unsigned attributes since OpenSSL generally does not add any unsigned attributes in its implementation.

```
SignedData ::= SEQUENCE {
        version CMSVersion,
        digestAlgorithms SET OF DigestAlgorithmIdentifiers,
        encapContentInfo EncapsulatedContentInfo
        certificates SET OF CertificateSet OPTIONAL,
        crls SET OF RevokationInfoChoice OPTIONAL,
        signerInfos SET OF (1...MAX) SignerInfo,
}
SignerInfo ::= SEQUENCE {
        version CMSVersion,
        sid SignerIdentifier // issuer + serial or subject key ID
        digestAlgorithm DigestAlgorithmIdentifier
        signedAttrs SET OF (0...MAX) SignedAttributes OPTIONAL,
        signatureAlgorithm SignatureAlgorithmIdentifier,
        signature OCTET STRING,
        unsignedAttrs SET OF (0...MAX) UnsignedAttributes
           OPTIONAL,
}
```

Listing 4.4: Normal `SignedData`

CMS `DigestedData` and CMS `EncryptedData` Transforms

In the `DigestedData` structure shown in Listings 4.6 4.7 we are able to create a lookup for the content type and for the digest algorithm to save a 10-20 bytes. In the `EncryptedData` structure shown in Listings 4.8 and 4.9 we can only flatten the `encryptedContentInfo` structure and add a lookup for the content type.

```
Signed params ::= SEQUENCE {
        version CMSVersion,
        contentType LONG,
        eContent OCTET STRING,
        signerInfos SET OF (1...MAX) SignerInfo params,
}
SignerInfo params ::= SEQUENCE {
        version CMSVersion,
        type LONG, // tells us what type sidHash is
        sidHash LONG, // signerIdentifier hash
        parameter LONG, // LOOKUP for signature and digest
           algorithms
        signedAttrs SignedAttributes OPTIONAL,
        signature OCTET STRING
}
```

Listing 4.5: Reduced SignedData

```
DigestedData ::= SEQUENCE {

        version CMSVersion,

        digestAlgorithm X509_ALGORITHM,

        encapContentInfo EncapsulatedContentInfo, //holds data

        digest OCTET STRING,

}
```

Listing 4.6: Normal DigestedData

```
Digested params ::= SEQUENCE {

        version CMSVersion,

        contentType LONG,

        eContent OCTET STRING,

        digest OCTET STRING,

}
```

Listing 4.7: Reduced DigestedData

```
EncryptedData ::= SEQUENCE {
        version CMSVersion,
        \\ Data + Encryption Algorithm
        encryptedContentInfo EncryptedContentInfo,
        unprotectedAttrs SET OF unprotectedAttrs OPTIONAL,
}
```

Listing 4.8: Normal EncryptedData

```
Encrypted params ::= SEQUENCE {
        version CMSVersion,
        contentType LONG,
        encryptAlg X509_ALGORITHM,
        eContent OCTET STRING,
}
```

Listing 4.9: Reduced EncryptedData

CMS CompressedData Transform

We had already modified the original CompressedData according to section 4.4.4 by adding an identifier for a dictionary. Our method of adding an identifier was to add a digest to the CompressedData. Our other transforms to this structure were to add lookups for algorithms and the content type. These transforms are shown in Listing 4.11.

```
CompressedData ::= SEQUENCE {
        version CMSVersion,
        compressionAlgorithm X509_ALGORITHM,
        encapContentInfo EncapsulatedContentInfo,
        dictionaryAlgorithm X509_ALGORITHM,
        dictionaryDigest OCTET STRING,
}
```

Listing 4.10: Normal CompressedData

```
Compressed params ::= SEQUENCE {
        version CMSVersion,
        parameter LONG, // lookup for compressionAlg + digestAlg
        eContent OCTET STRING,
        dictionaryDigest OCTET STRING,
}
```

Listing 4.11: Reduced CompressedData

CMS AuthenticatedData Transform

AuthenticatedData contains a message authentication code algorithm and optional digest algorithm. The Lite version of AuthenticatedData implements parameters for these algorithms and repeats the same modifications done to the EnvelopedData for its RecipientInfo data. The results are shown in Listing 4.13.

```
AuthenticatedData ::= SEQUENCE {
        version CMSVersion,
        originatorInfo originatorInfo OPTIONAL,
        recipientInfos RecipientInfos,
        macAlgorithm MessageAuthenticationCodeAlgorithm,
        digestAlgorithm DigestAlgorithmIdentifier OPTIONAL,
        encapContentInfo EncapsulatedContentInfo,
        authAttrs AuthAttributes OPTIONAL,
        mac MessageAuthenticationCode,
        unauthAttrs UnauthAttributes OPTIONAL
}
```

Listing 4.12: Normal AuthenticatedData

```
Authenticated params ::= SEQUENCE {
        version CMSVersion,
        originatorInfo originatorInfo OPTIONAL,
        recipientInfos RecipientInfo params,
        parameter LONG, // for algorithms and contenttype
        eContent OCTET STRING
        authAttrs AuthAttributes OPTIONAL,
        mac MessageAuthenticationCode,
        unauthAttrs UnauthAttributes OPTIONAL
}
```

Listing 4.13: Reduced `AuthenticatedData`

4.5.4 Using CMS Lite

The CMS Lite structure performs tradeoffs to achieve smaller message size. Some of the data elements in CMS are used to increase efficiency in parsing the message. These are cut out in CMS Lite. Other data elements increase the flexibility of the structure. We use parameters to perform lookups into tables instead of allowing any type of algorithms and contentType. This saves space but limits the number of algorithms that can be expressed to those which are already known and accounted for. Finally, eliminating some OPTIONAL certificate, CRL, and attribute data limits the range of data that can be passed through CMS. All these changes help reduce the size of the message in a disadvantaged network situation.

We note CMS's encapsulation properties in section 3.3.1. CMS Lite can also be used in these circumstances, but should be used in a specific order. When creating encapsulated structures, each level of structure should be encoded into a CMS Lite structure before moving to the next level. This is because none of the data inside each structure is known. On receipt, the recipient would then have to run CMS Lite decoding on each level of the structure to recreate the original message. The reverse procedure would have to be done for the recipient to unfold and decode the structure. A nested CMS structure encoding procedure is shown in Figure 4-8.

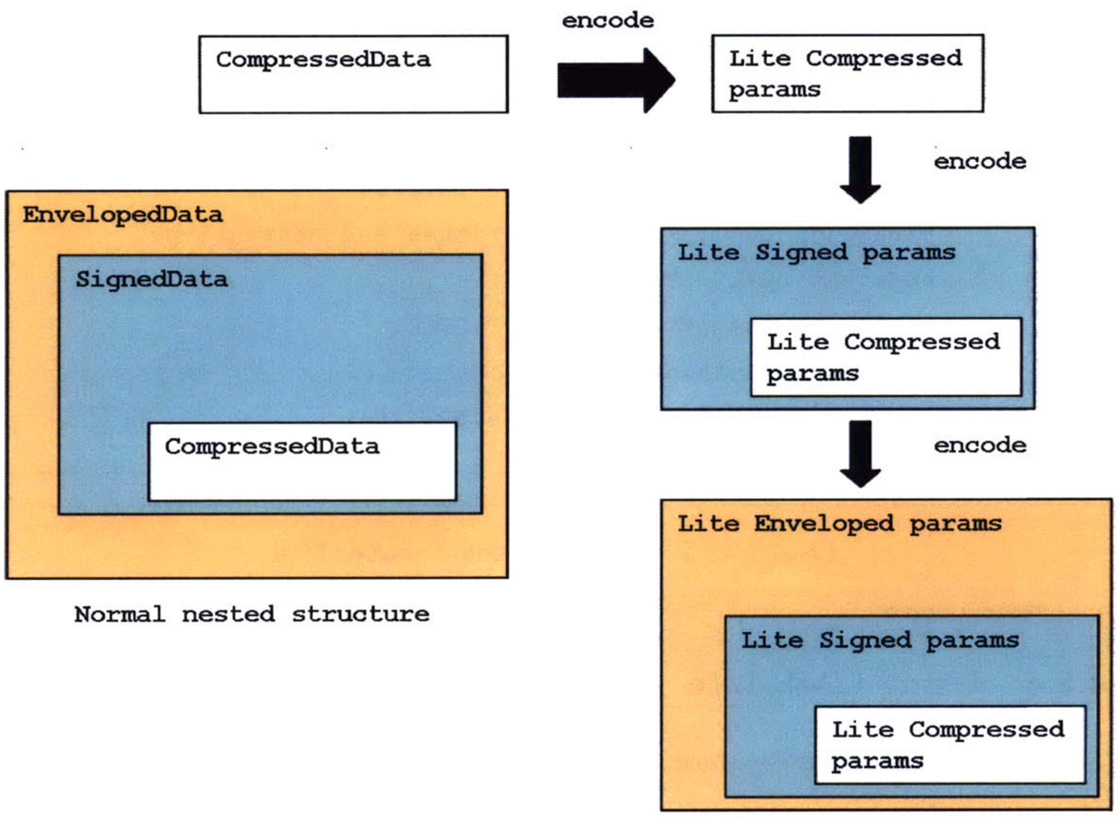

Figure 4-8: Encoding encapsulated CMS Lite structures

4.5.5 Table Lookups

One of the ways we reduce overhead is by using lookups for certain types of information in CMS. We reduce several table lookups into one parameter which is stored as a `LONG` type. The information in the long is stored bitwise. Each type of information has a bitmask with 4 bits of information unmasked for each content type in the table. On encode, CMS Lite processes all the constant data to be put in the parameter and creates integer identifiers for each piece of data according to the lookup table. The identifiers are bit shifted according to their type and added to the parameter. The lookup table with the bit masks and shifts is shown in Table 4.4.

4.6 CMS Lite and ZLIB

A limitation of the CMS Lite transform is that optional data like attributes, certificates, and CRLs can not be easily reduced in size. These optional structures may have a large number of configurations, and thus specifying a transform for each one would be both time consuming and only beneficial in certain circumstances.

However, a possible solution to this problem is to create a CMS Lite transform, then compress it using ZLIB with preplaced dictionaries as described in section 4.4.4. The dictionary could be modified to add in common optional data in certificates, CRLs, and attributes. CMS Lite would eliminate overhead due to ASN.1 encoding and fixed structures while ZLIB would eliminate overhead from the optional data. In order to regain the original CMS structure, the ZLIB-compressed CMS Lite structure would have to be decompressed by ZLIB, then decoded from CMS Lite to normal CMS structures. This potential solution has not been tested, but is an idea for further research in this field.

4.7 Summary

We performed a study on Cryptographic Message Syntax in an attempt to optimize the use of CMS in a disadvantaged network. We presented a few possible solutions to optimize CMS by reducing its message overhead. The first method to reduce overhead was through compression of the constant literals in CMS. In this method we used ZLIB while priming the compressor with a preplaced static dictionary containing CMS messages. The second method was to create a Lite version of CMS which reduced the size of each CMS message. These solutions are evaluated in Chapter 6 according to their *recoverability*, *flexibility*, *extensibility*, and *simplicity*. For both ZLIB compression and CMS Lite there is an issue of applying them to a CMS message that contains encrypted CMS messages. However, we can get around this limitation if we apply the compression methods first before encryption.

Name	**Parameter mask**	**Value**	Data Type
AES-128bit-cbc	0x00000F	1	Data Encryption Alg
AES-256bit-cbc	0x00000F	2	Data Encryption Alg
RC4	0x00000F	3	Data Encryption Alg
RC4-32-12-16-cbc	0x00000F	4	Data Encryption Alg
CAMELLIA256cbc	0x00000F	5	Data Encryption Alg
bfcbc	0x00000F	6	Data Encryption Alg
ideacbc	0x00000F	7	Data Encryption Alg
cast5cbc	0x00000F	8	Data Encryption Alg
des-ede3-cbc	0x00000F	9	Data Encryption Alg
des-ede-cbc	0x00000F	0	Data Encryption Alg
data	0x0000F0	0	Content Type
signed	0x0000F0	1	Content Type
enveloped	0x0000F0	2	Content Type
signedandenveloped	0x0000F0	0	Content Type
digested	0x0000F0	4	Content Type
encrypted	0x0000F0	5	Content Type
compressed	0x0000F0	6	Content Type
lite	0x0000F0	7	Content Type
RSA Encryption	0x000F00	0	Key Management Alg
RSA Encryption	0x00F000	0	Signature Alg
DSA-with-SHA1	0x00F000	1	Signature Alg
ECDSA-with-SHA1	0x00F000	2	Signature Alg
md2	0x0F0000	0	Digest Alg
md4	0x0F0000	1	Digest Alg
md5	0x0F0000	2	Digest Alg
sha1	0x0F0000	3	Digest Alg
dss	0x0F0000	4	Digest Alg
ecdsa	0x0F0000	5	Digest Alg
sha224	0x0F0000	6	Digest Alg
sha256	0x0F0000	7	Digest Alg
sha384	0x0F0000	8	Digest Alg
sha512	0x0F0000	9	Digest Alg
mdc2	0x0F0000	10	Digest Alg
ripemd160	0x0F0000	11	Digest Alg

Table 4.4: CMS Lite parameter lookup table

CMS Structure	Parameterized Variables	Data Type
EnvelopedData	encryptedContentType contentEncryptionAlgorithm keyEncryptionAlgorithm	Content Type Data Encryption Alg Key Management Encryption Alg
SignedData	encapContentType digestAlgorithm signatureAlgorithm	Content Type signerInfo Digest Alg signerInfo Signature Alg
EncryptedData	encryptedContentType	ContentType
DigestedData	encapContentType digestAlgorithm	Content Type Digest Alg
CompressedData	encapContentType dictionaryAlgorithm	Content Type Digest Alg

Table 4.5: Variables in each CMS Structure which are replaced by parameters

Chapter 5

Implementation of Techniques

In the previous chapter we examined a few methods to eliminate overhead in CMS messages. We described ZLIB in detail and how to improve per-message compression using a static, preplaced dictionary composed of common message elements. We also described methods used to create a Lite version of CMS which used lookups, hashes, and reduced optional data in order to decrease the size of the CMS message. These methods can encode a message for transport on a network and decode it in order to restore the original messages.

This chapter is geared toward an implementor possibly working on or extending this work. Here we discuss the details of our implementation environment. Then we discuss how we extended OpenSSL to use dictionaries. Finally, we describe how to manipulate ASN.1 structures in the OpenSSL library to create the new CMS Lite structure and how we encode and decode from normal CMS messages to CMS Lite messages.

5.1 Technologies Used

Before we delve into problem details, we describe the development environment, tools, and libraries used to implement and test our software. We mention tools and libraries that we used to build our solutions as well as systems that we depended upon and used to test our implementation.

5.1.1 Tools and Environment

The development environment consisted of Ubuntu 8.04 Hardy Heron running as a `VMware Workstation` [15] image on a Windows XP laptop. The majority of code was written with the `vim` [14] text editor, and `ctags` [2] allowed `vim` to quickly navigate among symbols and files. In order to incorporate code into larger projects we used `GNU Autotools` [12]. The tool `gdb` [3] proved invaluable in debugging. Finally, this document was created using LaTeX with the editor `Kile` [5].

5.1.2 Libraries and APIs

Several libraries were important to the successful completion of this study. The CMS optimizations rely heavily on previously developed code, as it would have been impossible to complete without it, and because there's no sense in reinventing the wheel.

ZLIB

We chose ZLIB not only because it offered lossless compression, but also because the API for ZLIB was readily available. The ZLIB API has an associated Perl API and also can be added as an option to OpenSSL. Our code used the C API of ZLIB version 1.2.3 described in Table 4.4.1.

OpenSSL

Our implementation is highly dependent on OpenSSL [13]. OpenSSL is an open-source effort that implements a general-purpose cryptographic library, various cryptographic message packing formats, SSLv2/v3, and TLSv1. In version 0.9.8h, OpenSSL included support for handling CMS. We modified that support in version 0.9.8j to implement our ZLIB compression scheme and CMS Lite. The CMS library is disabled by default and must be enabled while configuring OpenSSL.

We enable OpenSSL's use of ZLIB compression. We also build OpenSSL as a shared library for use by our testing infrastructure. We used the OpenSSL version 0.9.8j as the base version for our tests. During the build process, we configure OpenSSL for our needs by using the following command line:

```
./configure enable-cms zlib shared
```

The first option builds CMS support, including our modifications. The second links in ZLIB support, which OpenSSL can use internally, and which we rely on for our ZLIB performance enhancements. The final option causes the build system to create a shared library for use by our testing infrastructure.

As of version 1.0.0-beta2, the OpenSSL library supports only a subset of CMS. For instance, the OpenSSL CMS library only supports one form of key management, Key Transport mode. Within Key Transport, only RSA certificates can be used. This limits our interoperability with GROK [1] and ASE, sister projects for developing efficient cryptographic systems on disadvantaged networks, since they require different key management modes and elliptic curve certificates.

Here are some helpful hints for users modifying the OpenSSL source: OpenSSL offers several facilities to help find and reduce bugs in code that relies on the library. They include error messages, a safe stack implementation, and a set of regressions tests, which can be found in the UTIL/ directory of the source code. Errors can be generated with `mkerr.pl` by specifying a function code for the name of the function in which the error occurs and a reason code explaining why the error occurred. For instance, if the user wanted to generate an error in the function `CMS_Encrypt_Data`, then the associated function code might be `CMS_F_ENCRYPT_DATA` and reason code might be `CMS_R_NO_CERTIFICATE`. The `mkstack.pl` script will create type-specific functions for stack access for every ASN.1 type for which `DECLARE_STACK_OF(TYPE)` has been called.

5.2 Compression Implementation

This next section details some of the structures and functions used to create a compressed CMS type with a preplaced dictionary. We will also look at some of the

[1] Joseph Cooley, Roger Khazan, Benjamin Fuller, and Galen Pickard. GROK: A Practical System for Securing Group Communications. Submitted for publication, 2009.

functions we used to create CMS dictionaries to support zlib compression. For each part we will look at the high-level functionality of the code, then give some code snippets which should help an implementer understand the low-level details.

5.2.1 Strategy

We used the OpenSSL BIO facility to implement CMS ZLIB compression. These structures are used to perform input and output over a variety of interfaces. BIO structures can read and write from network interfaces, files on disk, and in-memory structures. Other BIO structures act as filters for data. Message digest computation, encryption algorithm input/output, and compression are also implemented using BIO structures. In order to create the compressed data from the input data, we use the following function calls.

```
BIO *memory;
BIO *zlib;
BIO *data;
BIO *chain;

memory = cms_content_bio()      // create a new memory storage BIO
zlib = BIO_f_zlib()             // Initialize the ZLIB BIO filter
BIO_setDict(zlib,dictionary)    // set the dictionary file
chain = BIO_push(zlib, memory)  // Add the memory BIO to the end
                                // of the ZLIB BIO

// Write the input data BIO through the
// zlib filter to the memory bio
SMIME_crlf_copy(data,chain);
CMS_dataFinal(cms,chain);
// Store the resulting data into the cms structure

cms_copy_content(chain,out);  // Decompress
```

Listing 5.1: Creating compressed data

A similar procedure to the one in listing 5.1 is used to compress and decompress the data in the compressed data type. We add the `BIO_setDict` function in this listing. This function takes a character string representing the path to the dictionary file and loads the dictionary file into memory with a maximum size of 32KB (since ZLIB doesn't support more than a 32KB dictionary). The BIO chains are constructed as shown in Figure 5-1 Two other pieces of data are used to reference the dictionary in the CMS `CompressedData` structure. We took the approach using a dictionary digest to reference the dictionary. To create the digest, we re-used OpenSSL code to create a digest, and copied the content and the digest algorithm into the `CompressedData` structure. To uncompress a structure, a user must select a dictionary that matches the dictionary used to compress the message. The user's dictionary digest is matched to the digest sent with the payload before uncompressing the structure with the input dictionary. The function returns `NULL` and error messages if the dictionaries do not match or if the data can not be uncompressed.

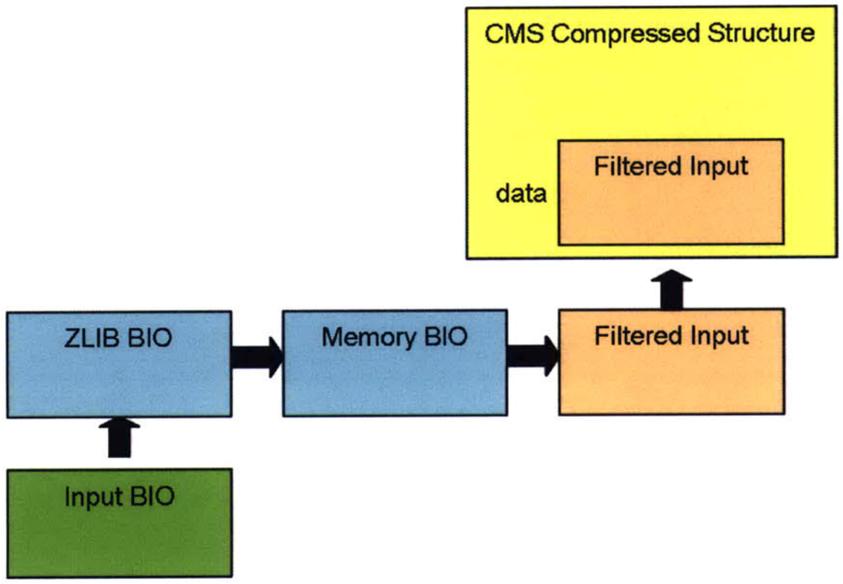

Figure 5-1: The ZLIB filter compresses data and writes it into the adjacent BIO. The adjacent BIO, a memory BIO, stores the data into the data section of the `CompressedData` CMS structure.

5.2.2 Extending ZLIB BIO

In order to support a dictionary, the ZLIB BIO structure needed to have pointers to the data and the length of the data. `BIO_setDict` calls a macro `BIO_setDict(b, f)=> BIO_ctrl(b, BIO_CTRL_COMP_ZLIB_DICT, 0, f)` This calls functions available only to the zlib `BIO`s and will return errors if called on other `BIO` types. In ZLIB, this will call, `BIO_setDictionary(BIO *b, char *filename)`, which is listed below in Listing 5.2.

```
int BIO_setDictionary(BIO *b, char *filename)
        {
        BIO_ZLIB_CTX *ctx;
        FILE *f;
        if(!b) return 0;
        ctx = (BIO_ZLIB_CTX *)b->ptr;
        // throw away previous dictionary
        if(ctx->dbuf != NULL){
                OPENSSL_free(ctx->dbuf);
                ctx->dbuf = NULL;
                ctx->dbufsize = 0;
                }
        f = fopen(filename, \"rb\");

        // Error if the file can not be opened
        if (!f)
                {
                COMPerr(COMP_F_BIO_ZLIB_SETDICT,
                        COMP_R_ZLIB_FILEOPEN_FAILURE);
                return 0;
                }
        fseek(f, 0, SEEK_END);
        ctx->dbufsize = ftell(f);
        fseek(f, 0, SEEK_SET);

        // Allocate the dictionary
        ctx->dbuf = (unsigned char *)
                OPENSSL_malloc(ctx->dbufsize+1);
        // Read in to the buffer from the file
        fread(ctx->dbuf, ctx->dbufsize, 1, f);
        fclose(f);
        return 1;
        }
```

Listing 5.2: BIO setDictionary: allocates memory for a dictionary in the ZLIB BIO and stores the dictionary size

When reading and writing through this BIO, a couple of modifications were made to incorporate the dictionary using the ZLIB API. When reading in a BIO, inflate returns a code if it needs a dictionary to inflate the data.

```
ret = inflate(zin, 0);
```

If the dictionary is needed, the dictionary buffer is checked and the ZLIB function `inflateSetDictionary` called if the dictionary is found, as shown in Listing 5.3.

```
while (data available to inflate)
{
        if(ret == Z_NEED_DICT)
        {
                // If no dictionary set
                if (!ctx->dbuf)
                {
                COMPerr(COMP_F_BIO_ZLIB_READ,
                   COMP_R_ZLIB_NO_DICT_ERROR);
                return 0;
                }
                // Give ZLIB the dictionary from the
                // BIO buffer dbuf
                ret = inflateSetDictionary
                        (zin,ctx->dbuf,ctx->dbufsize);
                continue;
        }
}
```

Listing 5.3: ZLIB inflateSetDictionary

If writing out a ZLIB BIO with a dictionary, `deflateSetDictionary(zout, ctx->dbuf, ctx->dbufsize)` is called before writing any data to set up the preplaced dictionary. The major modifications for these changes are in the OpenSSL source directory in CRYPTO/COMP/C_ZLIB.C.

5.2.3 Extending the CMS `CompressedData` Type

Few changes are required to support dictionaries in the `CompressedData` structure. Two new library functions are defined in `cms.h`:

```
CMS_ContentInfo *CMS_compress_dict(BIO *in, char *dictionary,
const EVP_MD *md, unsigned int flags)

int CMS_uncompress_dict(CMS_ContentInfo *cms, char *dictionary,
BIO *dcont, BIO *out, unsigned int flags)
```

`CMS_compress_dict` takes the input, dictionary path, message digest structure, and output flags and creates a new `CMS_ContentInfo`. The function `CMS_uncompress_dict` uses the `CompressedData` structure, a path to a dictionary, an output BIO, and flags to reverse the process. It returns a code indicating success and stores uncompressed data in the output BIO. These two functions are listed below in Listing 5.4 and Listing 5.5.

```
        CMS_ContentInfo *cms = NULL;
        CMS_ContentInfo *digestedDict = NULL;
        CMS_CompressedData *cd = NULL;
        BIO *dict;
        dict = BIO_new_file(dictionary, "r");

        digestedDict = CMS_digest_create(dict, md, flags);
        cms = cms_CompressedData_create(NID_zlib_compression);
        cd = cms->d.compressedData;

        cd->dictionaryAlgorithm = X509_ALGOR_new();
        if (!md)
```

```
            md = EVP_sha1();

    cms_DigestAlgorithm_set(cd->dictionaryAlgorithm,md);
    cd->dictionaryDigest = ASN1_OCTET_STRING_new();

    ASN1_OCTET_STRING_set(cd->dictionaryDigest,
            digestedDict->d.digestedData->digest->data,
            digestedDict->d.digestedData->digest->length);
    cd->dict = (char *)OPENSSL_malloc(strlen(dictionary));
    strcpy(cd->dict,dictionary);

    CMS_ContentInfo_free(digestedDict);
    BIO_free(dict);
    if (CMS_final(cms, in, NULL, flags))
            return cms;
```

Listing 5.4: CMS compress dict

```
{
        BIO *cont;
        BIO *new;
        BIO *dict;
        BIO *digest;
        BIO *pushed;
        int r;
        CMS_CompressedData *cd;

        dict = BIO_new_file(dictionary, "r");
        cd = cms->d.compressedData;

        new = BIO_new(BIO_s_mem());
        digest = cms_DigestAlgorithm_init_bio
                (cd->dictionaryAlgorithm);
        pushed = BIO_push(digest,new);
        SMIME_crlf_copy(dict, pushed, flags);

        (void)BIO_flush(pushed);
        r = cms_CompressedData_verify(cms, pushed);
        if (r == 0)
        {
        CMSerr(CMS_F_CMS_UNCOMPRESS_DICT,
                        CMS_R_COMPRESSED_NOT_VERIFIED);
                BIO_free(new);
                BIO_free(dict);
                BIO_free(digest);
                return 0;
        }
        if (cd->dict)
        {
                OPENSSL_free(cd->dict);
                cd->dict = NULL;
        }

        cd->dict = (char *)OPENSSL_malloc
                (strlen(dictionary));
        strcpy(cd->dict,dictionary);

        cont = CMS_dataInit(cms, dcont);
        if (!cont)
                return 0;
        r = cms_copy_content(out, cont, flags);
        do_free_upto(cont, dcont);
        BIO_free(new);
        BIO_free(dict);
        BIO_free(digest);
        return r;
}
```

Listing 5.5: CMS uncompress dict

CMS `contentType`	**Data Included in Dictionary**	**Data Excluded from Dictionary**
All Content Types	Version numbers, contentType identifiers	encapsulated content
`EnvelopedData`	content encryption algorithm, key encryption algorithm, certificate public key identifiers, originator identifying information, unprotected attributes	content encryption keys
`SignedData`	certificate public key identifiers, digest algorithms, signature algorithms, signed attributes	signatures, certificates, certificate revocation lists
`EncryptedData`	content encryption algorithm, unprotected attributes	encrypted data
`DigestedData`	digest algorithm	digest
`CompressedData`	compression algorithm, signer identifier, key encryption algorithms	dictionary digest, content encryption keys, key encryption keys

Table 5.1: Strings included in condensed CMS dictionary

5.2.4 Creating a Dictionary

Writing CMS Strings

To test ZLIB compression, as discussed in section 4.4.4, we need to generate a dictionary of partial CMS structures. Normally, OpenSSL outputs a full DER encoded structure using the function `i2d_CMS_bio(BIO *out, CMS_ContentInfo *cms)`. We augmented this functionality in the following function, using the OpenSSL ASN.1 API and its CMS structures: `int CMS_getStrings(BIO *out, CMS_ContentInfo *cms)`
In the function, we determine which `contentType` the CMS structure refers to, then write the fixed parts of the different CMS messages to the `out BIO` structure. Generally we define the fixed parts as version numbers, content types, algorithm definitions, and other non user-specific data. We define the variable parts as user identifiers, signatures, digests, keys, and content. All this fixed data is written in DER encoding to the `BIO` structure. Table 5.2.4 shows which data is included for each content type.

We created a sample CMS message for each contentType using one sender and one receiver for each `contentType` and wrote all these samples to condensed string versions excluding variable data.

We also used full CMS structures in the dictionary instead of condensing them first to non-user specific data. For both these versions, we then wrote all the structures out to disk in a row via a file BIO. This concatenation of CMS structures or strings is the CMS part of the dictionary.

Certificates

In order to generate a set of certificates which may be commonly referenced or transmitted, we relied the methods that Internet browsers use to establish trust. Mozilla's Firefox [7] browser as well as Microsoft's Internet Explorer [4] and any other major browser all have root level certificates pre-installed in the browser to establish chains of trust. If websites provide certificates which are signed by some chain of certificates which eventually goes back to a root-level installed certificate, the browser assumes we can trust that chain of certificates. Thus, information in these root level certificates should be referenced quite frequently in many cryptographic messages.

We exported a subset of the pre-installed root level certificates as well as some DoD root certificates from DISA all in DER encoding. These certificates were then concatenated all together to form a certificate package.

Cutting Down on Certificate Size in the CMS Dictionary

We found that the CMS part of the dictionary was 250 bytes for CMS strings and 450 bytes for full CMS structures. However, the certificate package part of the dictionary encompassed much much more of the space. Each certificate we included in the original dictionary took around 1024 bytes. When the size of the dictionary is limited to 32KB of data, that means 32 certificates can be stored in the dictionary. Considering the large number of certificates available even in the Internet browser, this is a serious limitation. Thus, we considered as an optimization, parsing out just the issuer names of the certificates since the issuer name is sent to identify which certificate was used in signing or encrypting in `SignedData` and `EnvelopedData` structures. We could've also included other fields common to many certificates, such as algorithm types. We then made a certificate package of just these issuer names. Each issuer name only accounted for $\tilde{5}0$ bytes of space compared to the 1024 bytes used before. This means that we could fit about 640 certificates in the certificate part of the dictionary instead of 32, a 20x improvement. A dictionary was then constructed with just issuer names. Later, in section 6.3 we explore the performance associated with each dictionary type.

Putting it Together

Partial CMS strings and the certificate package were concatenated together to form a full ZLIB dictionary for CMS. The dictionary is preplaced and used in `BIO_setDictionary` to compress and uncompress CMS messages. The total sizes of the dictionaries were kept under 32KB in order to stay under the limit imposed by ZLIB, as discussed in section 4.4.

5.3 CMS Lite

We just described extensions to OpenSSL for creating a CMS `CompressedData` type, and the methodology used to generate a CMS-specific dictionary. This section details some of the structures and functions used to create a CMS Lite data type, which uses content-aware compression to reduce CMS overhead. We also detail how an implementor can extend this work and use the library code in OpenSSL to create other types. For each part we will look at the high-level functionality of the code, then give some code snippets which should help an implementor understand the low-level details. We will first describe some requirements for the code, then discuss how we created an ASN.1 type for CMS Lite in OpenSSL. We then describe the encoding and decoding processes and summarize.

5.3.1 Requirements

CMS Lite is meant to trade off some of the flexibility afforded by ASN.1 notation and CMS for a reduction in message size. In addition, the implementation meets the requirements specified in section 4.3 (recoverable, flexible, extendable, and simple to implement) by adhering to the following:

- *Copy user-specific data:* Data which is used once per message, such as a content encryption key or initialization vector, must be copied straight from the CMS message to the CMS Lite message. Signatures and encrypted data must also be copied byte for byte.
- *Use ASN.1 API:* The current CMS implementation uses the OpenSSL ASN.1 API to perform input/output and allocation operations on data structures. Reuse of these functions allows our code to rely on well-tested code.
- *Encode and decode:* Encoding and decoding between CMS Lite and CMS content types requires different encoding and decoding functions for each type.
- *Adding algorithms:* Combinations of algorithms should be easy to add to the implementation.

5.3.2 Adding ASN.1 Types for the CMS API

This section details how we used the ASN.1 syntax and API to create a CMS Lite type. The overview will help elucidate our CMS Lite extension.

ASN.1 Notation

ASN.1 describes a method of representing data in two ways. First, it separates the way the data is encoded from how it is sent. The ITU standard X.209 defines the methods of encoding data, of which two are Distinguished Encoding Rules (DER) [22] and XML Encoding Rules (XER) [20]. DER specifies data as a series of triplets of data type, length, then value while XER specifies each data type as an XML tag with the content inside the tag.

The second part of the ASN.1 standard involves the notation of data. In ITU standard X.208 [21]the basic syntax for ASN.1 structures is defined. Some of the common terms used in ASN.1 are `SEQUENCE` which means that a series of data is next. The `OCTET STRING` type identifies an eight-bit byte string of data. Some other common types are `INTEGER` which defines a byte representation of an integer, and a `BOOLEAN` which defines a 0 or 1 value. CMS definitions contain several terms with ASN.1 data qualifiers. The `OPTIONAL` qualifier means that the data element may not be included while `SET OF` means that there is a grouping of multiple pieces of a data type. The `CHOICE` qualifier means that one of the following pieces of data is chosen, `UNION` means that a set of the options is chosen. These types are all used in the CMS definitions set in RFC 3852 [35].

OpenSSL ASN.1 API

The OpenSSL ASN.1 API attempts to abstract away the details of how the different data structures for each ASN.1 object are created and allow the developer to use a syntax similar to the one used in defining ASN.1 objects. The Table 5.2 shows some of the useful functions for defining new ASN.1 structures. All these functions are accessible through the OpenSSL header files path/base.

Function	**Description**
ASN1_SEQUENCE	Define a named sequence of data types. This connects a previously defined C structure to this ASN.1 type name. Both must have matching data types in order to correctly store the data
ASN1_CHOICE	Define a CHOICE data element
ASN1_SIMPLE	Create a simple ASN1 Object
ASN1_IMP	Creates an IMPLICIT object
ASN1_IMP_SET_OF	Creates a SET OF a data type
AN1_OPT	OPTIONAL data added to a sequence
DECLARE_ASN1_ITEM	Allows a previously defined sequence or item to be used. Also defines allocation and encoding/decoding functions for the item using the ASN.1 library
DECLARE_STACK_OF	Defines allocation, push and pop, and free functions for a safe stack implementation of an ASN.1 type

Table 5.2: OpenSSL ASN.1 library functions

CMS Lite Definitions

We liberally used the OpenSSL ASN.1 API in order to create the CMS Lite definitions and modify the CMS implementation to include a CMS Lite structure. We called this new data type `LiteData` and it is defined as a top-level `contentType` under CMS. Its CMS implementation is shown in Listing 5.6. It is defined as a choice of parameters representing the other `contentTypes`. This ASN.1 definition then matches with the designed data type shown in Listing 4.1. Similar ASN.1 structures were created for every other type of transforms shown in section 4.5.3. The full ASN.1 definitions for these structures are in CRYPTO/CMS/CMS_ASN1.C in our modified OpenSSL implementation. They are also listed in Appendix A.

```
 ASN1_CHOICE(CMS_LiteData) = {
        ASN1_IMP(CMS_LiteData, params.signedParams,
        CMS_Signed_params, 0),
        ASN1_IMP(CMS_LiteData, params.envelopedParams,
        CMS_Enveloped_params, 1),
        ASN1_IMP(CMS_LiteData, params.digestedParams,
        CMS_Digested_params, 2),
        ASN1_IMP(CMS_LiteData, params.encryptedParams,
        CMS_Encrypted_params, 3),
        ASN1_IMP(CMS_LiteData, params.authenticatedParams,
        CMS_Authenticated_params, 4),
        ASN1_IMP(CMS_LiteData, params.compressedParams,
        CMS_Compressed_params, 5)
} ASN1_CHOICE_END(CMS_LiteData)
```

Listing 5.6: CMS LiteData

5.3.3 Encoding and Decoding

Encoding and decoding through the CMS Lite implementation is done through two functions, `CMS_LiteEncode(CMS_ContentInfo *message)` and `CMS_LiteDecode(CMS_ContentInfo *message, STACK_OF(CMS_RecipientInfo)*recipients)` `LiteEncode` takes normal CMS messages stored in `message` and returns a `LiteData` structure which is the encoded message. LiteDecode takes the encoded message and a stack of X.509 [37] certificates which represent the possible recipients of the message and decodes the message into a normal CMS message. Encoding and decoding functions are all written into CRYPTO/CMS/CMS_LITE.C.

Because each CMS contentType is very different in its makeup compared to other CMS contentTypes, each structure must have a separate encoding and decoding function. We implement the changes described in section 4.5.3 using the ASN.1 functions that are defined for each ASN.1 data type. We use these functions to allocate and set the values for the CMS Lite ASN.1 structures.

SignedData

For SignedData we copy any content over from the CMS structure and allocate and define a Signed_params structure. Each SignerInfo in the normal structure is handled separately by the function CMS_SignerInfo_params_init. It converts all the normal structure's SignerInfo structures into SignerInfo_params structures, eliminating overhead for each structure. Below we give the code snippets for encoding and decoding SignedData.

```
  CMS_LiteData *CMS_LiteEncode_Signed
   (CMS_ContentInfo *message, CMS_ContentInfo *cms)
 {
        CMS_SignedData *sd;
        CMS_LiteData *ld;
        CMS_Signed_params *sp;
        ASN1_OCTET_STRING **data;
        int ret;

        sd = message->d.signedData;

        // Create the LiteData structure and set pointers to it
        ld = cms_LiteData_create(cms);
        sp = cms_LiteSign_create(ld);
        // Replace the contentType object with an integer.
        sp->contentType =
           (encodeAlgorithm(sd->encapContentInfo->eContentType)
        << CMS_CONTENTTYPE_SHIFT);
        sp->version = sd->version;
        data = CMS_get0_content(message);

        // Allocate and copy the data from the SignedData
        if (sp->eContent)
        {
                M_ASN1_OCTET_STRING_free(sp->eContent);
                sp->eContent = M_ASN1_OCTET_STRING_new();
        }
        else
        {
                sp->eContent = M_ASN1_OCTET_STRING_new();
        }
        ASN1_STRING_set(sp->eContent, (*data)->data,
           (*data)->length);

        // Transform each SignerInfo into SignerInfo_params
        ret = cms_SignerInfo_params_init(sd,sp);
        if (ret==0)
                goto mallerr;

        // return a pointer to the LiteData part of the
        // created CMS message
        return cms->d.liteData;
}
```

Listing 5.7: Encoding SignedData

```
int cms_SignerInfo_params_init(CMS_SignedData *sd,
  CMS_Signed_params *sp)
{
        STACK_OF(CMS_SignerInfo) *sis;
        STACK_OF(CMS_SignerInfo_params) *sps;
        CMS_SignerInfo *si;
```

```
CMS_SignerInfo_params *sip;
X509_ATTRIBUTE *attr;
X509_ATTRIBUTE *dup;
int i;
int digestParam = 0;
int signParam = 0;

sis = sd->signerInfos;
sps = sp->signerInfos;

// Iterate through the stack of SignerInfo
for (i=0;i<sk_CMS_SignerInfo_num(sis);i++)
{
        // Stack accessing functions
        si = sk_CMS_SignerInfo_value(sis,i);
        // ASN.1 allocation for the new type made possible
        // by DECLARE_ASN1_ITEM(CMS_SignerInfo_params)

        sip = M_ASN1_new_of(CMS_SignerInfo_params);

        // store the hash of the issuer and serial number
        sip->sidHash =
           cms_getSignerIdentifier_hash(si->sid,EVP_md5());
        sip->type = si->sid->type;

        ... Copy over the signature ...

        // Encode the signature and digest algorithms
        // into integers, which are combined later
        signParam =
        encodeAlgorithm(si->signatureAlgorithm->algorithm);
        digestParam =
        encodeAlgorithm(si->digestAlgorithm->algorithm);

        // Bit shifts the data to store both parameters
        // in one integer
```

```
                sip->parameter = (signParam <<
                   CMS_SIGNATUREALG_SHIFT) +
                        (digestParam << CMS_DIGESTALG_SHIFT);

                ... copy over signed attributes here ...

                // Push the new structure onto the
                   SignedData_params
                if (!sk_CMS_SignerInfo_params_push(sps,sip)
                 ... deallocate if failure ...
        }

        return 1;
}
```

Listing 5.8: Encoding SignedData continued

```
int CMS_LiteDecode_Signed(CMS_ContentInfo *message,
  CMS_ContentInfo *cms, STACK_OF(X509) *recipients)
{
        CMS_LiteData *ld;
        CMS_Signed_params *sp;
        CMS_SignedData *sd;
        STACK_OF(CMS_SignerInfo_params) *sips;
        CMS_SignerInfo_params *sip;
        CMS_SignerInfo *si;
        int i,ret = 0;
        int contenttype =0;

        ... allocate all the data structures ...

        // find the contentType from the parameter
        contenttype = (sp->contentType & CMS_CONTENTTYPE_MASK) >>
        CMS_CONTENTTYPE_SHIFT;
        sd->encapContentInfo->eContentType =
        getContentTypeFromParam(contenttype);
        ... copy over the encapsulated data ...

        sips = sp->signerInfos;

        // Iterate through the SignerInfo_params
        // to recreate SignerInfos
        for (i=0;i< sk_CMS_SignerInfo_params_num(sips);i++)
        {
                sip = sk_CMS_SignerInfo_params_value(sips,i);
                si = M_ASN1_new_of(CMS_SignerInfo);
                ret = cms_liteDecode_signer(sip, si, recipients,
                   sd);
        }
        ret = 1;

        return ret;
}
```

Listing 5.9: Decoding Signed_params

```
int cms_liteDecode_signer(CMS_SignerInfo_params *sip,
  CMS_SignerInfo *si, STACK_OF(X509) *signers, CMS_SignedData *sd)
{
        X509 *recip = NULL;
        X509_ATTRIBUTE *attr;
        X509_ATTRIBUTE *dup;
        X509_ALGOR *alg;
        int digestAlg = -1;
        int signatureAlg = -1;
        int i;

        // Compare our certificates signerIdentifier hashes
        // against the one in the message. Returns the certificate
        // which matches

        if ((recip = cmp_issuer_and_serial(signers,
           sip->sidHash)) == NULL)
```

```
                return 0;

        ... copy over the signerIdentifer data from the
           certificate...

        ... copy over the signature here ...

        // get the digest and signature algorithms
        // from the parameter
        digestAlg  = (sip->parameter & CMS_DIGESTALG_MASK)>>
        CMS_DIGESTALG_SHIFT;
        signatureAlg = (sip->parameter & CMS_SIGNATUREALG_MASK) >>
        CMS_SIGNATUREALG_SHIFT;

        si->digestAlgorithm = getDigestFromParam(digestAlg);
        si->signatureAlgorithm =
           getSignatureAlgFromParam(signatureAlg);

        ... copy over the signed attributes ...

        ... push the digest algorithms on the stack ...

        return 1;
}
```

Listing 5.10: Decode `Signed_params` continued

`EnvelopedData`

In order to decode and encode the `EnvelopedData` structure, we use the ASN.1 functions just like we did in encoding and decoding `SignedData`. Again, we flatten the normal data structure, taking out the encrypted data. However, we found that we needed to simply copy over the `contentEncryptionAlgorithm` from the normal structure to the `encryptAlg` data field instead of creating a parameter for it. This is because, the `contentEncryptionAlgorithm` stores the initialization vector for symmetric encryption algorithms if one is needed. In encoding and decoding EnvelopedData, the RecipientInfo structures are handled separately through the function CMS_LiteEnvelopedRecipient_init. We again include some functions as we did for SignedData in Listings 5.11 and 5.12.

```
CMS_LiteData *CMS_LiteEncode_Enveloped
  (CMS_ContentInfo *message, CMS_ContentInfo *cms)
{
        CMS_LiteData *ld;
        CMS_EnvelopedData *env;
        CMS_Enveloped_params *envparams;
        ASN1_OCTET_STRING **data;
        int keytransencrypt = 0;
        int symmetrickey = 0;
        int encryptedcontent = 0;

        if (cms->d.other == NULL)
        {

              ... allocate the structures and set the pointers ...

              // Copy the encryption algorithm to encryptAlg
              envparams->encryptAlg = X509_ALGOR_dup
              (env->encryptedContentInfo->
              contentEncryptionAlgorithm);

              // table lookup for the algorithms used
              symmetrickey  = encodeAlgorithm(env->
              encryptedContentInfo->contentEncryptionAlgorithm->
              algorithm);
              encryptedcontent
              = encodeAlgorithm(env->encryptedContentInfo->
              contentType);

                ... copy encrypted data over ..
        // Copy appropriate data from the message recipient
        // infos to the litedata infos
        // Only handles KeyTransport recipient infos as OPENSSL
        // doesnt handle anything but this type as well.


                cms_LiteEnvelopedRecipient_init(message,
                envparams,&keytransencrypt);

                // set the version and the parameter for
                // enveloped lite data
                envparams->version = env->version;
                envparams->parameter = (encryptedcontent<<
                 CMS_CONTENTTYPE_SHIFT) +
                 (keytransencrypt << CMS_KEYTRANSPORT_SHIFT)

                return cms->d.liteData;
        }

}
```

Listing 5.11: Encoding EnvelopedData

```
 int cms_LiteEnvelopedRecipient_init(CMS_ContentInfo *cms,
  CMS_Enveloped_params *envparams, int *enc)
{
        STACK_OF(CMS_RecipientInfo) *ris;
        STACK_OF(CMS_Recipient_params) *rps;
        CMS_RecipientInfo *ri;
        CMS_Recipient_params *rp;
        CMS_KeyTransParams *ktp;
        CMS_KeyTransRecipientInfo *ktri;
        int i;

        ... set up the pointers ...
```

```
for (i = 0;i < sk_CMS_RecipientInfo_num(ris); i++)
{
        ri = sk_CMS_RecipientInfo_value(ris, i);
        ktri = ri->d.ktri;

        rp = M_ASN1_new_of(CMS_Recipient_params);
        rp->d.ktp = M_ASN1_new_of(CMS_KeyTransParams);
        ktp = rp->d.ktp;
        ktp->version = ktri->version;
        rp->type = CMS_RECIPINFO_TRANS;

        // get the hash for the signeridentifier (md5)
        ktp->sidHash =
        cms_getSignerIdentifier_hash(ktri->rid,EVP_md5());

        ...set the type of the signerIdentifier

        ... copy the encrypted key over ...
        // replace the encryption algorithm with an
           integer
        if (enc)
                *enc = encodeAlgorithm(ktri->
                keyEncryptionAlgorithm->algorithm);

         ... push the lite recipient parameter on the
            stack

}
return 1;
```

Listing 5.12: Encoding EnvelopedData continued

Other CMS types

The same concepts were applied to the other CMS contentTypes in order to encode and decode to and from those types into a LiteData structure. We will not include code snippets from those because their structures are similar, and in some cases, more simple. We did not attempt to encode and decode the `AuthenticatedData` structure into CMS Lite. This was because the OpenSSL implementation of the CMS library did not have a function to create `AuthenticatedData`. The encode and decode functions for the `CompressedData`, `DigestedData`, and `EncryptedData` types is in CRYPTO/CMS/CMS_LITE.C.

5.3.4 Summary of CMS Lite

We updated the CMS API to enable it to transform normal CMS structures into Lite structures and back again. We added restrictions with this Lite type by limiting algorithms and hashing the issuer name and serial numbers identifying X.509 certificates. In order to decode the CMS Lite structure we check all our known certificates to find the one that corresponds to each issuer name and serial number. We also were limited by the implementation of the CMS in OpenSSL. Not all the functions and types in the latest CMS RFC [35] are implemented in OpenSSL; Only one key management technique is implemented and there is no function to created the `AuthenticatedData` data type. However, our extended CMS Lite implementation decodes and encodes all the structures supported by OpenSSL.

Chapter 6

Evaluation

In the previous chapter, we described extensions that incorporate ZLIB compression with preplaced dictionaries and a new CMS Lite type into the OpenSSL library. We included code snippets and discussed which OpenSSL APIs supported the implementation.

This chapter describes a performance evaluation of how CMS Lite and ZLIB dictionary compression affect CMS message size. To show our results, we first detail our test infrastructure and our test data. We then compare the optimization methods against each other, discuss results, and assess their message size overheads as mentioned in section 4.1. Finally, we offer conclusions and recommendations on using these methods to optimize CMS.

6.1 Testing Infrastructure

In order to encrypt and sign messages using CMS, a sender and a recipient must be defined. In many cases, entities sending and receiving messages use the PKI in order to attach cryptographic information to a user's identity. Public key certificates are generated for every entity that wishes to participate in signed and/or encrypted communications. The OpenSSL CMS API uses X.509 certificates to sign or encrypt `EnvelopedData`, `SignedData`, `EncryptedData`, and `AuthenticatedData`. Thus, in order to test the performance of our API extensions, we used X.509 certificates for dummy entities created using a plugin developed for the program Pidgin [10].

We also needed a method for storing and retrieving certificates during test runs. Authenticated Statement Exchange (ASE) uses a database for each user to store previously transmitted certificates along with other information, so we included it in our software implementation.

Finally, we needed a platform on which to create and use the CMS API. Since ASE also uses OpenSSL in order to perform cryptographic operations, we leveraged ASE to access user information and use our CMS API by writing a custom CMS API wrapper for ASE.

6.2 Test Methodology

To test our CMS Lite and preplaced dictionary methods, we first created different types of CMS messages using OpenSSL's standard CMS API. Table 6.1 shows which functions were used to create the different types. All the functions and algorithms we used can be found in CRYPTO/CMS/CMS.H in OpenSSL.

For `EnvelopedData` tests, we sent an enveloped message to each of our dummy users: *Alice*, *Bob*, *Carol*, *Eve*, and *Mallory*. Each message contained one `recipientInfo` structure with a key encrypted to the user. The average size of these messages was used as the measurement for `EnvelopedData` messages. Similarly, for `SignedData` messages we made structures signed by each dummy user and again averaged the optimization results. After the normal CMS structures were created, we used the `i2d_CMS_bio` function discussed in section 4.4.4 to DER encode the structures. Then, we used the various preplaced dictionaries described in section 4.3 to compress these packages.

To test CMS Lite, we encoded the CMS structures in CMS Lite and then used `i2d_CMS_bio` to DER encode the CMS Lite structure. The size of the DER encoded message was used as the message size. We additionally decoded the CMS structure and ensured that the encapsulated data sent inside the package was intact and that the CMS structure that was recreated could be used in the same manner as the

CMS structure	Function used	Algorithms Used
EnvelopedData	`CMS_encrypt(STACK_OF(X509) *certs, BIO *in, const EVP_CIPHER *cipher, unsigned int flags)`	Key Transport mode-RSA Encryption, AES-256-CBC mode
SignedData	`CMS_sign(X509 *signcert, EVP_PKEY *pkey, STACK_OF(X509) *certs, BIO *data, unsigned int flags)`	RSA-Signatures, SHA-1 digest
DigestedData	`CMS_digest_create(BIO *in, const EVP_MD *md, unsigned int flags)`	SHA-1 digest
EncryptedData	`CMS_EncryptedData_create(BIO *in, const EVP_CIPHER *cipher, unsigned int flags)`	AES-256-CBC mode
CompressedData	`*CMS_compress(BIO *in, int comp_nid, unsigned int flags)`	ZLIB compression

Table 6.1: Functions used to create CMS messages

original.

We also tested our ideal and worst-case conditions. The ideal conditions we simulated by creating the same type of CMS message with differing payloads and compressing those messages one at a time without flushing the dictionary in between. We simulated the worst case conditions by simply using ZLIB compression without any previous messages or a preplaced dictionary. For both these tests, the structures were DER encoded.

6.3 Results

6.3.1 Dictionary Comparisons

We now compare ZLIB with preplaced dictionaries to CMS Lite. First, Figure 6-1 shows the ZLIB compression savings when using different dictionaries.

The compression savings are greatest for `SignedData` and `EnvelopedData` messages. The best dictionary in compression was one created by concatenating the full CMS structures with root certificates. The worst dictionary at compression was the one containing CMS strings taken from full structures, though its performance is very close to the others.

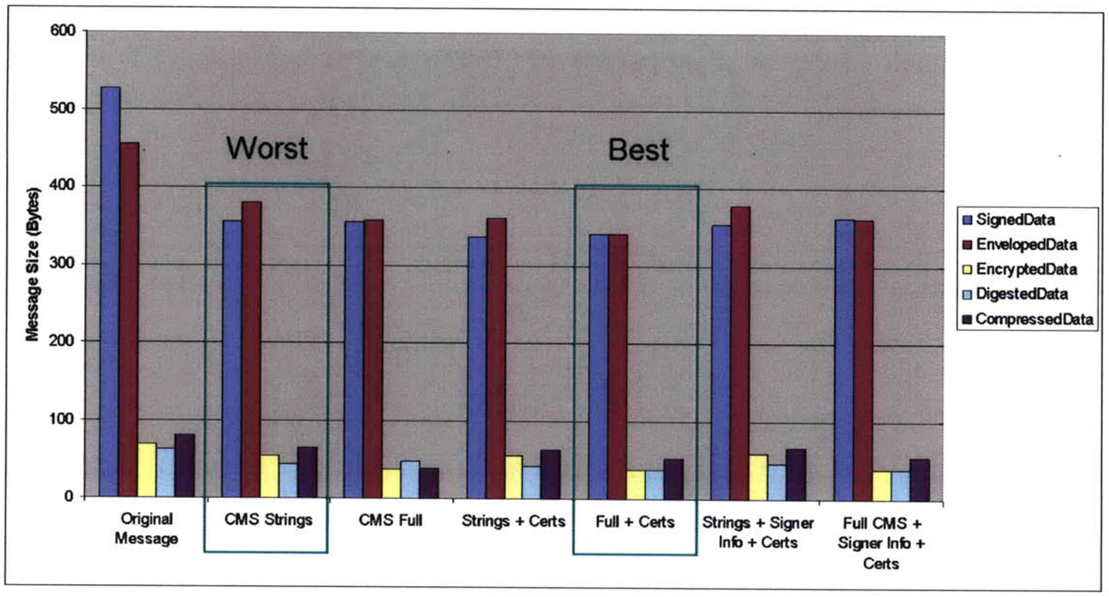

Figure 6-1: ZLIB compression comparing dictionaries

We then accounted for the 256 byte CEK, 256 byte signature, message payload, and 16 byte digest which all must be recreated perfectly to preserve CMS functionality. Since the rest of the CMS structure acts as metadata, we called it message overhead. We found that the best dictionary was compressing 270 bytes of overhead to 85 bytes in `SignedData` and 200 bytes to 85 bytes in `EnvelopedData`. These represent a savings of nearly 190 bytes per message. The worst dictionary still resulted in a savings of 170 bytes for `SignedData` and 70 bytes for `EnvelopedData`. Figure 6-2 shows these overhead sizes.

We also show the percentage of the non-CEK/signature/digest related overhead which is compressed by each dictionary. Figure 6-3 shows the best dictionary, Full + Certs, was able to compress nearly 70% of `SignedData` overhead and 60% of `EnvelopedData` overhead. Table B.2 in Appendix B shows the full data results for dictionary compression.

6.3.2 CMS Lite Comparsion

We also show the effect that the CMS Lite optimizations had on the message size of the structures. CMS Lite achieves some message overhead compression on all

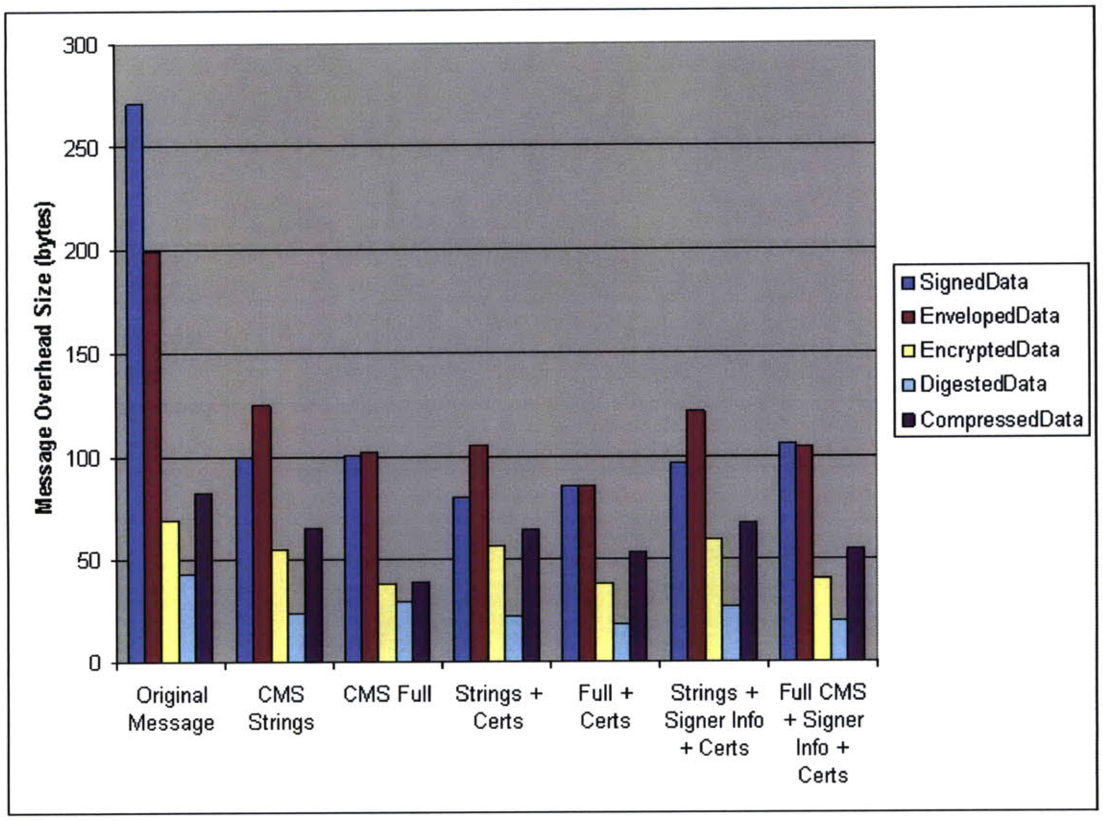

Figure 6-2: Overhead comparison after compression w/dictionaries

CMS structures. The largest reduction in message size occurs with **SignedData** and **EnvelopedData** structures. CMS Lite reduces 120 bytes of **SignedData** overhead and 100 bytes of **EnvelopedData** overhead. These results are shown in Figures 6-4 and 6-5 and in Table B.4 in Appendix B.

We also show in Table 6.2 the percentage compression of the non-key/signature/digest overhead for each CMS structure.

SignedData	EnvelopedData	DigestedData	CompressedData	EncryptedData
43.1	48.9	67.4	12.2	27.5

Table 6.2: CMS Lite overhead compression percentage

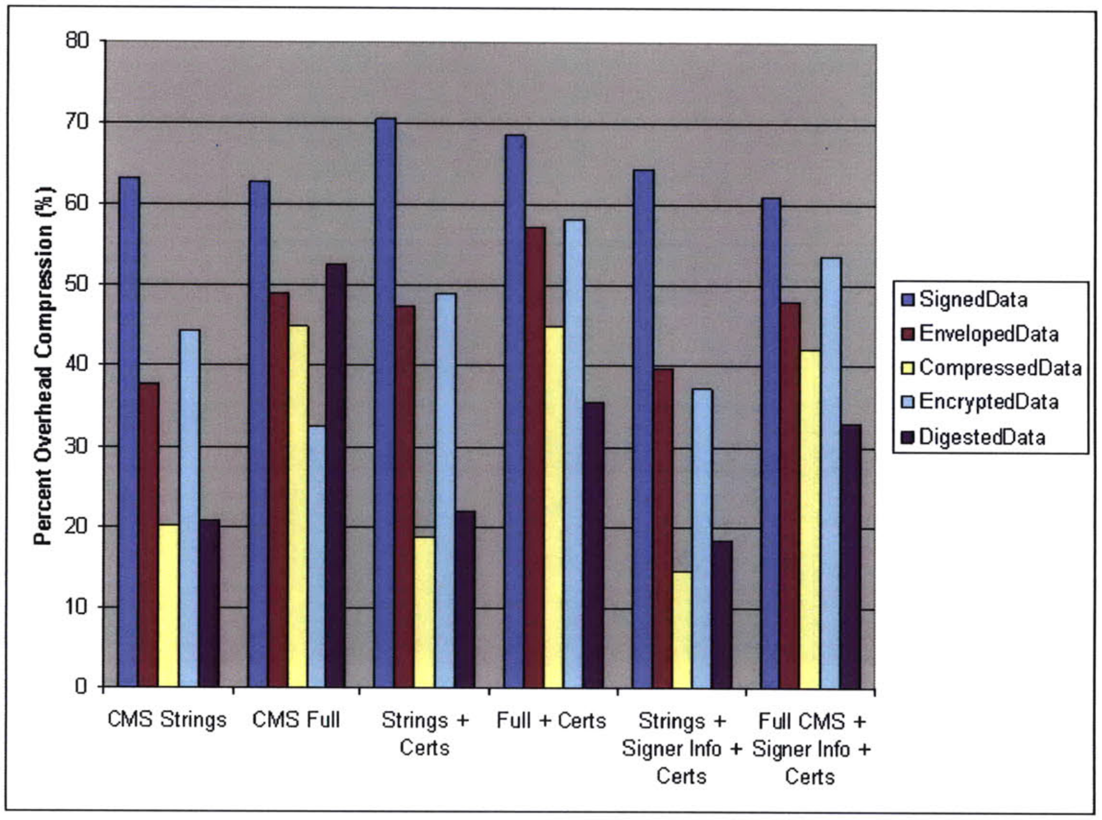

Figure 6-3: Overhead percentage compression w/dictionaries

6.3.3 Total Results

We finally compared the best dictionary results to the tests simulating the ideal and worst-case conditions. We expect to fall somewhere in between these two results, but hope to be closer to the ideal. We also compare CMS Lite to these results as well. We chose to only show the `SignedData` and `EnvelopedData` structures as these contained the most significant savings throughout the study. As you can see in Figure 6-6 the original messages do benefit from ZLIB compression without any optimizations. However, the added dictionary produces results closer to the ideal conditions test. CMS Lite performed better than ZLIB without a dictionary, but ZLIB with a preplaced CMS dictionary performed better than CMS Lite. In Figure 6-7 we see that the overhead reduction due to ZLIB compression and CMS Lite in fact is nearly the compression achieved with best case conditions. Overhead reduction using ZLIB without a dictionary is less than 30 bytes for `EnvelopedData` and less

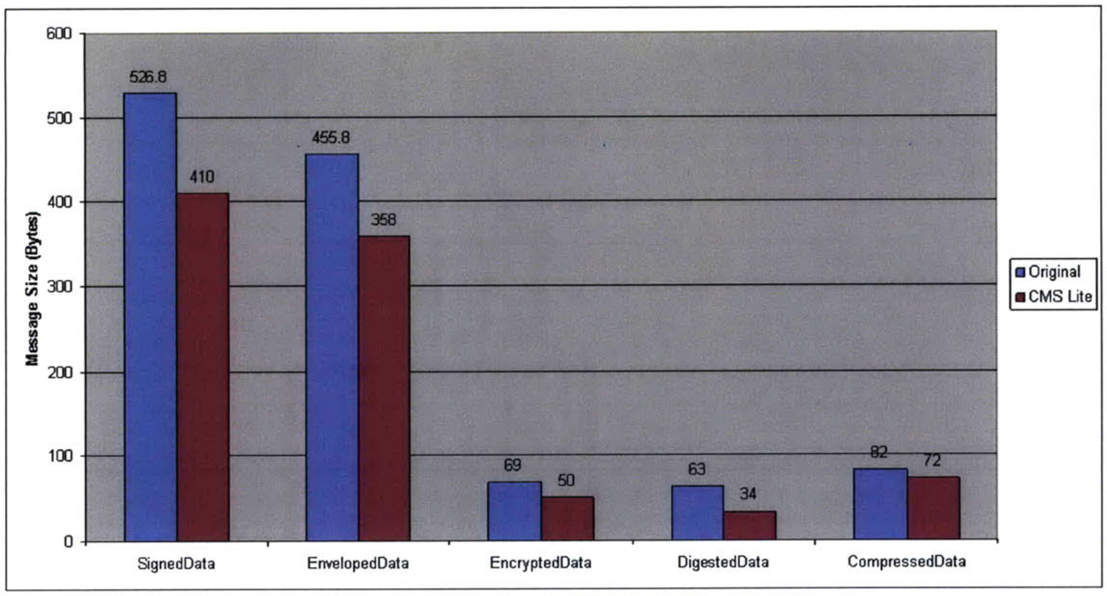

Figure 6-4: CMS Lite optimizations

than 70 bytes for SignedData, however it increases to 110 and 190 bytes respectively with a good preplaced dictionary. The full data set of results is available in Appendix B.

6.4 Discussion

In this section we discuss why we saw the results above and evaluate the different methods of optimizing CMS according to the requirements of each solution to be *recoverable, flexible, extensible, and simple* as described in section 4.3.

6.4.1 ZLIB with Dictionary Compression

Adding a preplaced CMS dictionary allows ZLIB to use references to previously loaded data, allowing for better compression. We found that the best dictionary used was one with full CMS structures and certificates concatenated as described in section 4.4.4. The full CMS structures have all the structures which are sent to the user,

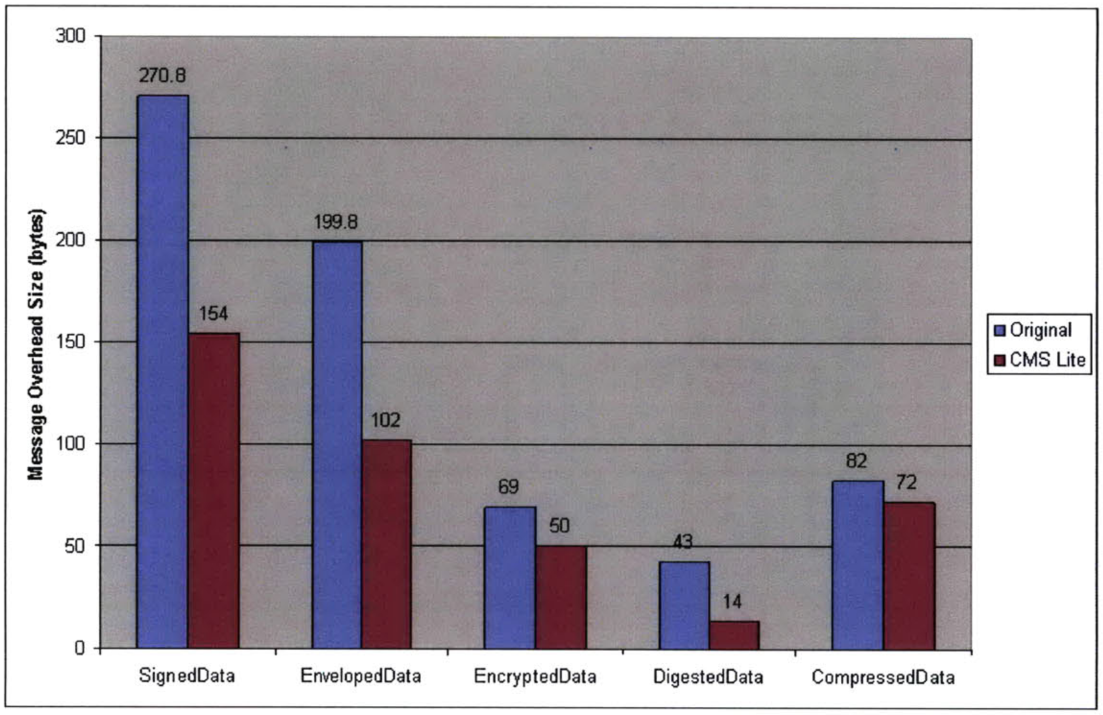

Figure 6-5: CMS Lite overhead size

allowing ZLIB to reference rather than output strings [1]. To further reduce messsage size, the preplaced dictionary could be optimized for CMS structures by aligning the dictionary strings in more efficient ways for ZLIB to find and reference them.

Using a dictionary with ZLIB compression does add a problem in message recoverability. To use a dictionary with ZLIB compression, all message recipients must possess the exact dictionary used during compression. They must also determine which of the many possible dictionaries to use in uncompressing the CMS structure. However, since a dictionary is a long-lived, public data item, it could be preplaced or downloaded from a network accessible repository and referenced by a unique identifier sent with the CMS message.

Since ZLIB compression does not affect the contents of the CMS data, it is flexible with any changes that occur in CMS. In terms of extensibility, if CMS is extended,

[1]We expect the actual performance of the dictionary to be a few bytes worse than the best dictionary's performance as we used messages sent to dummy user *Alice* in order to create the dictionary and also used *Alice* in testing the dictionary. Thus, we were able to re-use some of her certificate identifying information, which would not be possible if a canonical dictionary was used by all users.

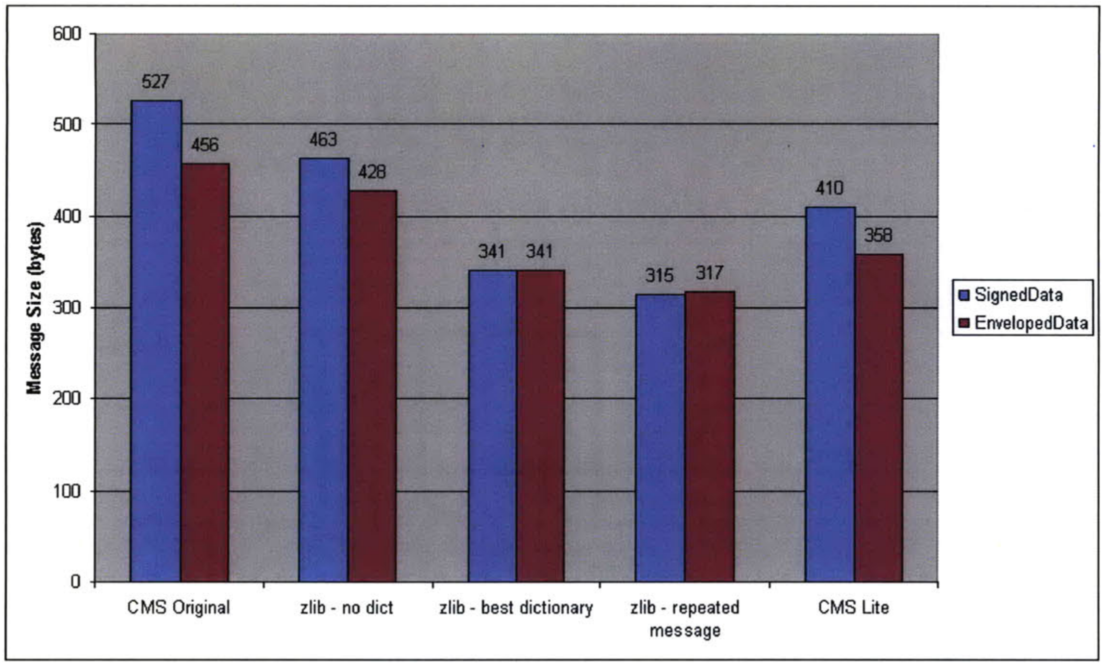

Figure 6-6: All optimizations compared to best and worst case conditions

then a new dictionary can be optimized for the changed structures. The only problems would arise in distributing the new dictionary to all participants in optimized CMS communications.

Implementing the ZLIB optimization was simple. Our extensions to the OpenSSL implementation of `CompressedData` required only about 350 lines of code change. Building the dictionary was not too difficult either. We implemented custom functions to extract literals from each CMS structure, which required around 300 lines of code change. We then were able to use the CMS library functions to create the CMS structures for the dictionary. To get certificates, we easily downloaded certificates from our dummy store and from the Mozilla web browser and the DoD root certificate webpage. We assembled the combinations of dictionaries quickly with `cat`.

6.4.2 CMS Lite

CMS Lite was able to transform normal CMS structures and save data by eliminating specifiers for certificates, objects, and algorithms while replacing them with shortened

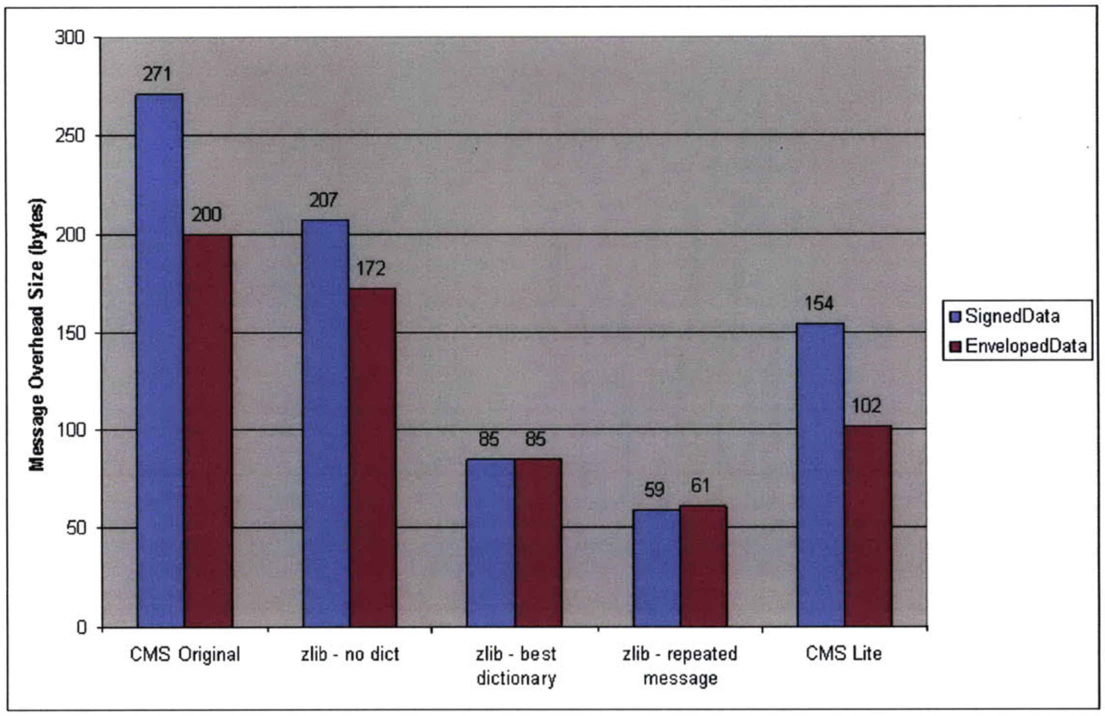

Figure 6-7: All optimizations overhead sizes

parameter values. One optimization that reduced a significant amount of overhead was replacing certificate's issuer name and serial numbers with a SHA-1 hash of their combination. This eliminated around 60 bytes of overhead, but also presented further challenges. Because of this change, CMS Lite messages do not enable the recipient to identify the certificates used in the message. However, if the recipients have all the certificates used in the message, then they can iterate through their certificates to find the ones in the message. We also believe that CMS Lite could be further optimized, reducing message size. This could be done by reducing more of the data in CMS Lite into a parameter by using stricter guidelines what configurations can be used.

CMS Lite can be fully recoverable just like ZLIB compression, however adding extra attributes to CMS structures unknown to CMS Lite would not result in these attributes being reduced in size. Also, because CMS Lite supports a reduced set of configurations, some optional data in CMS may not be recoverable after being transformed to CMS Lite. In terms of flexibility, CMS Lite can support any algorithms with some modifications and can be extended to support any changes to CMS. These changes would take more implementation work than creating a new dictionary, however. CMS Lite is simple in concept, but can be more difficult to code in OpenSSL as we had to deal with ASN.1 parsing and more OpenSSL code.

6.4.3 Wrap Up

We showed the results for both CMS and CMS Lite and then compared the two methodologies according to our requirements. Table 6.3 shows these results.

Requirement	**ZLIB w/dictionary**	**CMS Lite**
Recoverability	**high** - must distribute dictionary	**medium** - CMS Lite must be used to decode the message and some optional data not recovered
Flexibility	**high** - does not depend highly on CMS options used to CMS	**low** - can only support subset of algorithms and options
Extensibility	**medium** - new updates to CMS can be incorporated into a new dictionary	**medium** - Can support any updates to CMS, but must change CMS Lite to implement these changes
Simplicity	**high** - dictionary must be present with both sender and recipient	**medium** - implementation of all changes in OpenSSL

Table 6.3: Evaluating ZLIB compression w/dictionary and CMS Lite

6.5 Conclusion

Both CMS Lite and ZLIB compression offer benefits over simple ZLIB compression in a disadvantaged network. Our use of general purpose compression with preplaced dictionaries reduced overhead by 110 bytes per `EnvelopedData` message, nearly 60% of the overhead. For `SignedData` the overhead was reduced by 190 bytes for a 70% compression ratio. This was only about 30 bytes less reduction than the optimal compression test assuming a non-disadvantaged network. CMS Lite also performed respectably, achieving a 48% compression ratio for overhead on `EnvelopedData` structures and 43% compression ration for `SignedData` structures. In comparison, general compression without our optimizations only compressed 30 bytes of overhead for `EnvelopedData` messages and 60 bytes for `SignedData` messages, resulting in compression ratios of 14 and 23 percent respectively.

However, both methods have a couple of drawbacks. ZLIB compression with a dictionary requires the dictionary to be preplaced to compress and uncompress the message. CMS Lite requires a complex implementation at the message sender and recipient. Both could be implemented with a proxy in between users as shown in Figure 4-7. However, because of its greater simplicity, flexibility, and recoverability we believe ZLIB compression with preplaced dictionaries to be a better solution. ZLIB compression may be optimized further by improving the placement of CMS strings in the dictionary while CMS Lite could be improved by creating strict configurations which flatten CMS more and combine more CMS algorithms and content types into a parameter. Although ZLIB compression with a dictionary may be a better solution in our study, CMS Lite is worth exploring, as it does offer a possibility of the best message overhead compression.

Chapter 7

Extensible Messaging and Presence Protocol (XMPP) Optimizations for Disadvantaged Networks

In the previous chapters we discussed the implementation of ZLIB general purpose compression with a prepended dictionary in OpenSSL as well as the implementation of a content-aware compression CMS structure, CMS Lite. We discussed the testing infrastructure we set up to examine these optimizations and compared the results that these optimizations had on reducing the size of CMS messages.

We now describe a self-contained chapter which again follows our goals outlined in Chapter 1. We again attempt to optimize standardized message protocols for secure group communications. However, we now transition to another standardized form of communication, XMPP [52]. XMPP is an XML-based protocol derived from the open source Jabber protocol. XMPP has been adopted as a message packing protocol in the open source community using XML stanzas to deliver chat, video, and other content. We explore XMPP because although CMS can be used to create cryptographic structures, we wished transfer our standard optimizations and extensions to a higher layer, to enable standard cryptographically packaged information to be used on disadvantaged networks in applications such as chat.

To optimize XMPP for secure group communications in disadvantaged networks, this chapter offers methods to mitigate these problems by extending XMPP to enable and optimize the passing of group end-to-end secure messages and adding an optimization to reduce the overhead of setting up secure group communications on XMPP. We wrote and implemented specifications to package group-encryption data and authenticating information via XMPP. These optimizations and protocol additions can be used on a XMPP chat client and a XMPP chat server, but can also be extended to any client and server which implements the XML and XMPP standards. These changes present a standardized method for passing group secure messages and metadata in XMPP and reduce the amount of XMPP overhead. We present these changes as another example of an optimization to a standardized message structure for secure group communications on disadvantaged networks.

7.1 Motivation

Normally, secure communications in chat programs have been accomplished through TLS connections created between server and user and from server to server [30]. However, the cost of establishing TLS connections can be be to high in networks with low bandwidth, high latency, and intermittant connectivity. This is because a TLS handshake must be performed from every server to server and user to server link.

One way of mitigating the costs of encryption on disadvantaged networks is through the use of end-to-end encryption. When encryption is accomplished at the endpoints, the encrypted messages can be sent from server to server without establishing costly TLS connections. Also, with end-to-end security, no servers can access the content of the messages, only members in the encryption groups, adding another layer of security to the communications.

Currently XMPP has a specification for sending end-to-end encrypted messages from one user to another singular user [51]. There is no current standard for the use of group end-to-end messages in XMPP or in other protocols, which leads to inefficient implementation of end-to-end encryption with XMPP. Also, the mechanisms for sending XMPP messages to different users are not as efficient as they could be, especially when sending information needed to make encryption possible.

7.2 XMPP Background

XMPP's core was invented in 1998 and refined by the Jabber open-source community. It is based on XML structures and used to send real-time data. Although the core was defined in 2004 by IETF standards, the protocol is continually updated through the XMPP Standards Foundation (XSF) by means of XMPP Extensions. The core protocol has been updated to include instant messaging, presence protocols, multimedia streams, encrypted streams, and the sending of XML data [52].

The closest extensions available to send secure group messages are the end-to-end encryption extension [51], the S/MIME Encryption [47] protocol for XMPP, and the use of XML streams and TLS within XMPP [30]. However, none of these schemes are optimized for disadvantaged networks as they require multiple rounds of communication for TLS and XML streams and the sending of multiple certificates in S/MIME.

7.3 XMPP Extensions

XMPP Extension Protocols (XEPs) are the way that XMPP is extended with new capabilities. As of August 17, 2009 there were 271 extensions proposed to XMPP. To enable efficient group security, we propose two XMPP extensions, one for group end-to-end encryption and the other for subset addressing in multi-user chat. Draft extensions can be submitted according to the policy defined by the XMPP Standards Foundation to the XMPP Extensions Editor. They gain approval through the XMPP Council after a community driven approval and implementation process [18].

To extend XMPP to support group end-to-end encryption, we recognized the requirements for group encryption protocols and then develop a protocol on XMPP that satisfies these requirements. For subset addressing we identify the shortcomings with the current multi-user chat protocol and define ways to increase its efficiency.

7.3.1 Secure Group Messages Requirements

A group encryption protocol must achieve security through confidentiality, integrity, and authentication while also being interoperable and efficient. These qualities are defined below.

Confidentiality

Any member in the current chatroom and in the current encryption group must be able to decrypt and view a message sent to the chatroom. Each user may create encryption groups depending on who they wish to encrypt messages. Entities without membership to the encryption group of a message must not be able to decrypt the messages even if they are a member of the chatroom. Encrypting to existing chatroom membership might be a reasonable approach to maintaining confidentiality. In some cases, it might be desirable to include a server in the group. In those cases, the server could include itself in the chatroom membership as a user and users could choose to include the server in encryption groups just as they would include other users.

Integrity

The integrity of each message sent in an end-to-end encryption protocol is important because the transmission medium between the endpoints of each message is expected to be insecure. If no integrity checks are made, an attacker can modify the contents of a sent message in order to compromise the security of the encrypted communications. A message digest such as (SHA-1/2, MD5, etc.) [45] [48] should be included in the message in order to verify the integrity of the message.

Authentication

Each party to a conversation must have some method to cryptographically verify the authenticity of a message (note that this does not preclude the source from being anonymous, nor the source from being any member of a particular group, i.e., group-authentication). Such authentication requires that users share cryptographic material. In groups, users could accomplish this by authenticating to every other user separately. We use the sample users *Alice*, *Bob*, *Carol*, and *Dave* all participating in a secure group chatroom to illustrate: (*Alice* must authenticate to *Bob*, *Carol*, and *Dave*) and (*Bob* must authenticate to *Alice*, *Carol*, and *Dave*). Users alternatively could choose to delegate authentication to other entities (other users/servers).

A database may be used in order to store authentication information and to store bindings between XMPP IDs and identifying information. There is a problem of verifying that a digital identity corresponds to a real, valid, trusted user. Group encryption protocols may rely on public key infrastructures, webs-of-trust, key-continuity, and/or any other mechanism to provide such assurances.

Flexibility

The protocol must be upgradeable given the changing nature of cryptographic algorithms. It must also support the real time exchange of keying material, bindings, and IDs.

Efficiency

In group encryption, efficiency can be difficult to achieve. The communications volume is larger as group cryptographic data structures must be distributed to a larger number of participants, resulting in an increase in messages needed to be sent. Group distribution of these cryptographic data structures also adds more communication overhead per message than user-to-user encryption. These problems are compounded by the fact that whenever a new member joins or leaves an encryption group, new keys may have to be distributed. While these problems may be not as important on a wired high-bandwidth, low latency network, our group encryption mechanisms must take into account low-bandwidth, high latency networks and implement efficient group encryption schemes.

7.3.2 Secure Group Protocol

To satisfy the requirements for secure group messages, a secure group protocol has three parts. The first part is a group encryption scheme for binding cryptographic IDs to application IDs and performing end-to-end encryption to create secure messages. The second part is a scheme for exchanging cryptographic information necessary to accomplish encryption, and the third is a secure message format to send the data.

Group Encryption Schemes

A group encryption scheme encompasses methods which are used to pass encrypted data in between entities and dynamically rekey users as encryption-group memberships change. A scheme is needed to provide group end-to-end security. This scheme defines methods encryption groups use to rekey and the exact mechanisms and algorithms by which encrypted data is passed. One sample group end-to-end security scheme is S/MIME [47]. Another developed specifically for disadvantaged networks by Lincoln Laboratory is GROK.

Bootstrapping Secure Group Communications

When a client enables group encryption in a chatroom, a separate protocol is needed for exchanging information with other clients supporting group encryption in the room. The client may need to exchange public keys, name-key bindings or other information (methods used to distribute new keys to a group, supported encryption/signature algorithms, etc). Clients may automatically send this information when entering a group, or may wait until it is requested by other members in the encryption group. For XMPP, there is no specified method of sending this needed authentication information, though the information may be packaged in any manner, including CMS.

Secure Messaging Process and Format

In order for plugin software to encrypt a message to a group of users we propose a protocol which uses a `ge2e` element to specify secure group communications. The entire process is shown in Figure 7-1 and given as an example of a group message created by OpenSSL and sent via an XMPP client. We detail the process and format next.

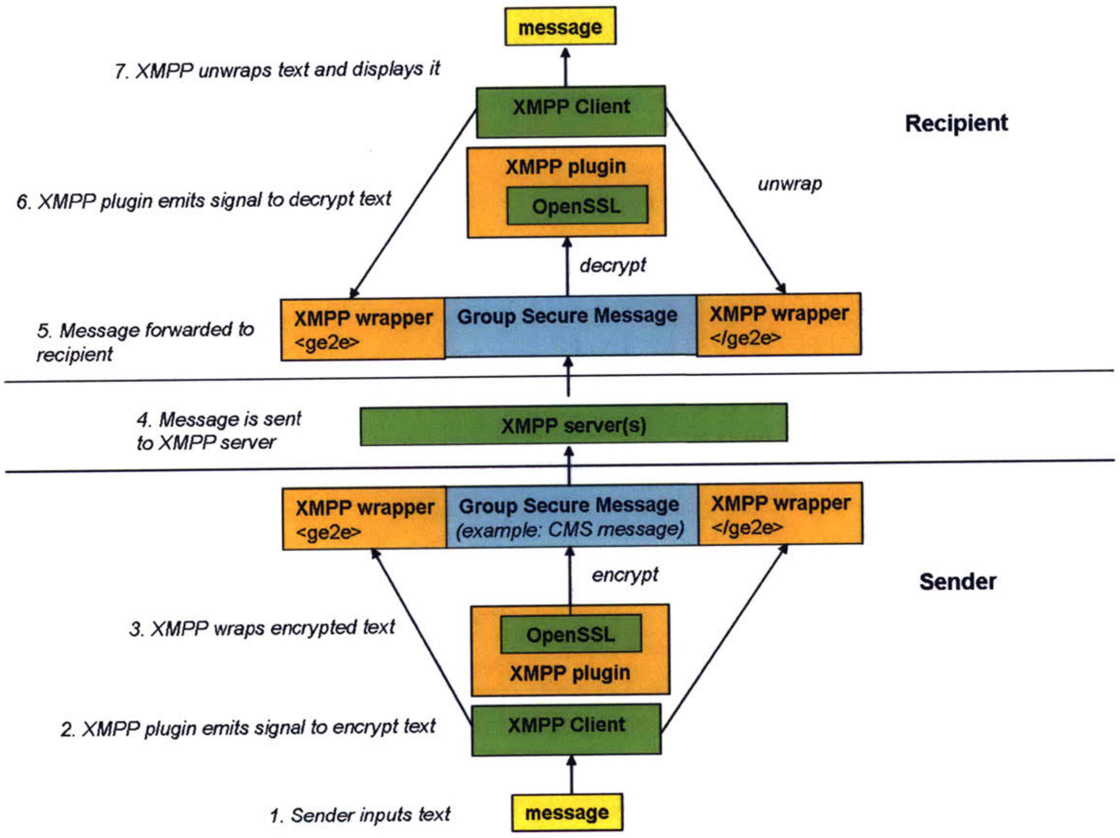
message
7. XMPP unwraps text and displays it
XMPP Client
Recipient
XMPP plugin
OpenSSL
6. XMPP plugin emits signal to decrypt text
unwrap
decrypt
5. Message forwarded to recipient
XMPP wrapper <ge2e>
Group Secure Message
XMPP wrapper </ge2e>
4. Message is sent to XMPP server
XMPP server(s)
XMPP wrapper <ge2e>
Group Secure Message (example: CMS message)
XMPP wrapper </ge2e>
encrypt
Sender
3. XMPP wraps encrypted text
OpenSSL
XMPP plugin
2. XMPP plugin emits signal to encrypt text
XMPP Client
1. Sender inputs text
message

Figure 7-1: XMPP group secure message sending process

First, the XMPP plugin encrypts the message using the encryption scheme of the user's choosing. Next the encrypted message is encoded in base-64 for text output to be sent.

```
Text: Testing the encrypted message system.
=> < - - encrypted text plus any necessary headers-- >
=> < -- base-64 encoded encrypted text -- >
```

The encrypted and base-64 encoded payload is enclosed in text element in the **ge2e** element whose attribute of *encrypt* tells what encryption scheme to use. This attribute specifies the group encryption scheme used to handle the inside text. This encompasses steps 1-3 in Figure 7-1. The message is passed from server to server in steps 4 and 5 until it reaches the destination client. Clients decrypt a message by first recognizing a **ge2e** element and then passing it to the group encryption process specified by the encrypt attribute in the text element as shown in steps 6 and 7 in Figure 7-1. The following listings show a sample XMPP chat protocol message with our extension. In this example, we use the group encryption scheme GROK.

```
<message
  From=alice@example.com/laptop
  To=chat@conference.example.com
  Type=groupchat>
  <ge2e xmlns=urn:xmpp:tmp:ge2e>
    <text encrypt=grok>
      <![CDATA[---encrypted message---]]>
    </text>
  </ge2e>
</message>
```

The server forwards the message to all group members using the multi-user chat protocol [19] and ignores the **ge2e** element unless it's a group member.

```
// Alice sends a ge2e message to the chatroom containing
// herself and Dave
<message
        from=chat@conference.example.com/alice
        to=bob@example.com/pda
        type=groupchat>
        <ge2e xmlns=urn:xmpp:tmp:ge2e>
                <text encrypt=ge2e>
                        <![CDATA[---encrypted message---]]>
                </text>
        </ge2e>
</message>
```

```
// The chatroom forwards the message to both users
// without dealing with the encrypted data
<message
        from=chat@conference.example.com/alice
        to=alice@example.com/laptop
        type=groupchat
        <ge2e xmlns=urn:xmpp:tmp:ge2e>
                <text encrypt=grok >
                        <![CDATA[---encrypted message---]]>
                </text>
        </ge2e>
</message>
// The chatroom forwards the message to Dave
<message
        from=chat@conference.example.com/alice
        to=dave@jabber.org/desktop
        type=groupchat>
        <ge2e xmlns=urn:xmpp:tmp:ge2e>
                <text encrypt=groupEncrypt>
                        <![CDATA[---encrypted message---]]>
                </text>
        </ge2e>
</message>
```

Listing 7.1: Sending a secure group message to a XMPP chatroom

In some cases, clients may not be able to decrypt a message received in the chatroom. This may occur if the message is not encrypted to them. The response of the receiving client depends on the group encryption scheme used in sending the message. However, the recipient may choose to ignore the message or show a identifier saying an indecipherable message was sent to the recipient. The recipient may also choose to send a response to the sender.

7.3.3 Subset Addressing

We created another XMPP optimization to help efficiency and functionality of XMPP in disadvantaged networks by reducing the amount of data needed to be sent. The extension allows servers to forward messages sent to a subset of users in a chatroom and requires changes to how XMPP chat clients and servers handle group chat messages.

Problem Description

Subset addressing is the ability to send a message to a subset of the users in a chatroom without having to create a new conversation with those users. In XMPP multi-user chat the chat server normally forwards all messages to all room members, but in some cases, messages are only intended for a subset of the room membership. This means unneeded messages can be sent and bandwidth unnecessarily used. One example of this problem can occur during secure group chat. When a secure group is a subset of the room membership, secure group membership changes, and the group encryption scheme needs to distribute a new key to the new secure group, messages may need to be sent to a subset of users in the chatroom. Using the subset extension, the server saves network bandwidth transmissions by eliminating rekey messages to those not in the secure group in the chatroom and by not creating new conversations for every user in the secure group membership.

Subset Addressing Usage

Subset addressing also supports other capabilities. For example, subset addresssing can also function as a limited form of security as an ad-hoc secure group can be formed by messaging to a specific subset. Finally, subset addressing could be an interesting social tool which could be used to exclude friends from conversations without their knowledge, pass secrets, and to avoid confusing others when messages arent meant for them. With slow or disadvantaged networks, and encryption, a user can use subset addressing to send cryptographic information to only the subset of users in a chatroom that need that data, thus eliminating the waste of copying a message to users who already have the information.

Sending a Subset-Addressed Message

In order to send messages to a subset of chatroom membership, a client includes a subset attribute in the XMPP message element. The subset attribute lists user nicknames delimited by the forward-slash symbol (/). If the list has no tilde symbol (˜) in front, then it specifies the subset of the members to whom the message should be delivered; otherwise, it describes the members that should be excluded from receiving the message.

```
<message from='alice@example.com' to='test@chat.example.com'
        type='groupchat subset=alice/bob/carol>
        <html><body> Hello subset</body></html>
</message>
```

Listing 7.2: Test chatroom subset message from *Alice*

Listing 7.2 shows a message that *Alice* wants to send to *Alice*, *Bob*, and *Carol*, but not *Dave*. The server test@chat.example.com parses the chat subset-addressed message and copies it to *Alice*, *Bob*, and *Carol*. The subset attribute is dropped and the original message is also dropped. The resulting messages shown in Listing 7.3 should look like normal messages from the *Alice* in the chatroom.

```
Example 1 cont:

<message from='test@chat.example.com/alice'
        to='alice@example.com/desktop' type='groupchat'>
        <body>Hello subset</body>
</message>

<message from='test@chat.example.com/alice'
        to='bob@example.com/desktop' type='groupchat'>
        <body>Hello subset</body>
</message>
```

```
<message from='test@chat.example.com/alice'
        to='carol@example.com/desktop' type='groupchat'>
        <body>Hello subset</body>
</message>
```

Listing 7.3: Resulting message sent to chatroom

A user can also send to a subset of people where the subset excludes certain nicknames. Listing 7.4 shows an example where *Alice* sends a message to test@chat.example.com now specifying to exclude *Dave*.

```
<message from='alice@example.com ' to='test@chat.example.com'
        type='groupchat subset=~dave>
        <html><body> Hello subset example 2</body></html>
</message>
```

Listing 7.4: *Alice* excluding *Dave*

The server would send the same copies of the message to the same users as in Listing 7.2. The exclusion list is more efficient to represent when the room membership is greater than half the total members. The inclusion subset representation is more efficient when the subset is less than half the room membership. The server ignores any invalid subset names and any users who become members after a sender transmits an exclusion list, but before a server parses a message, will receive the message.

Figure 7-2 shows how enabling subset addressing can help reduce the amount of data sent on XMPP. Normal XMPP protocol in a chatroom creates a message for all participants in the chatroom even when the message is intended for a subset of the people in the chatroom. Thus, a message costs $(n + 1) * m$ units, where n is the number of people in the chatroom and m is the message size. However, with subset addressing, a message costs $(k + 1) * m$ where k is the size of the subset. Thus, we save $(k + 1)/(n + 1)$ units, which obviously depends on the subset size.

The other method of sending to this subset would be to create separate conversations with each member. However, creating a new conversation requires has more message overhead than the user's name in the subset listing.

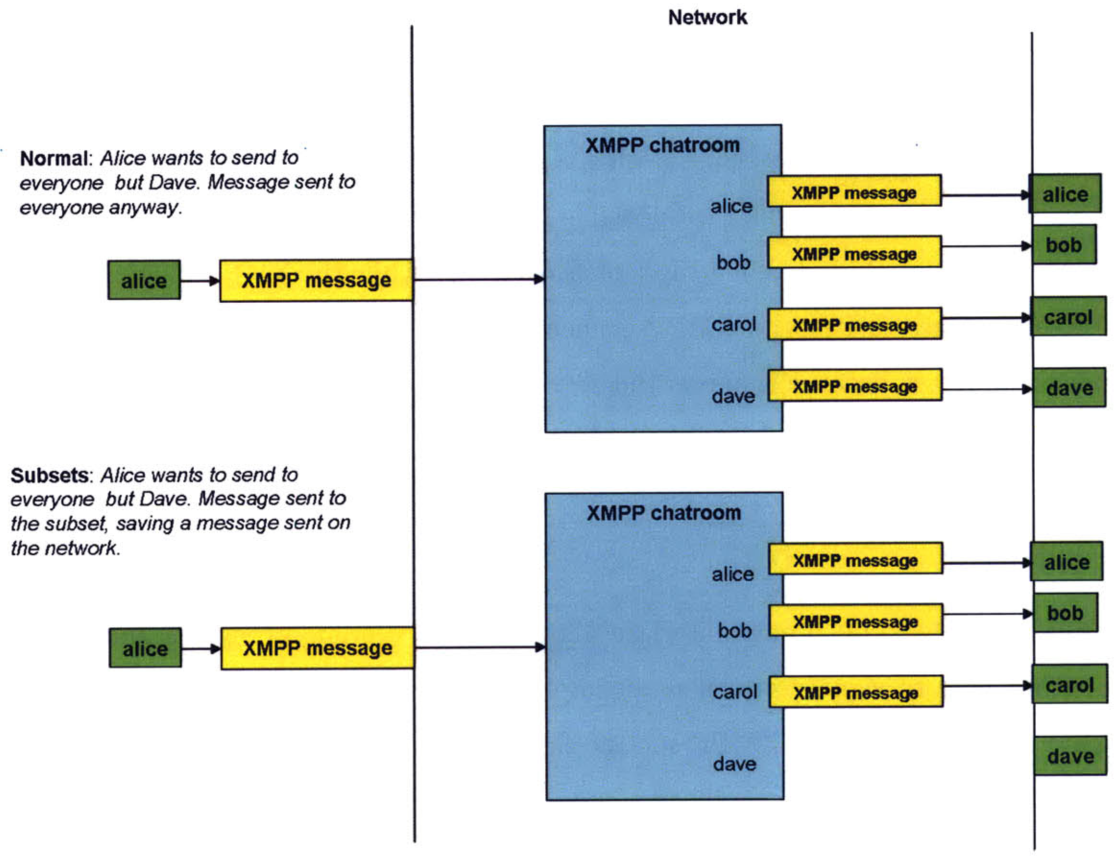

Figure 7-2: Message savings due to subset addressing

7.4 Implementation

We described two ways of optimizing XMPP to use group secure communications, the use of `ge2e` elements to specify when group secure messages are sent and the use of subset addressing.

7.4.1 Platform

We implemented our XMPP protocol extensions in the Pidgin chat client [10] and the Openfire [9] chat server. We chose Pidgin and Openfire because previous work in our research group built security extensions into the same software, the code for both of these programs is available through open source projects, and these clients both use the XMPP protocol to pass messages.

7.4.2 Group Secure Message Plugins

Pidgin implements a robust plugin interface that enables developers to extend functionality. The XMPP protocol is itself a type of plugin to the the LIBPURPLE message passing library. LIBPURPLE is responsible for the low level implementation of gathering user input and sending messages on the network while the XMPP plugin modifies these messages to wrap its XMPP elements around user input. We use the XMPP plugin interface because it allows Pidgin to benefit from any group security libraries that can encrypt chat data sent in XMPP.

Signals

Pidgin's LIBPURPLE library interface allows for the sending of signals and attaching handlers to signals, which is an implementation of events and event handlers paradigm. We define an XMPP-specific signal that group secure plugins can connect a handler to in order to encrypt communications.

```
purple_signal_register(plugin, "jabber-ge2e-encrypt",
        purple_value_new(PURPLE_TYPE_BOOLEAN), 5,
        purple_value_new(PURPLE_TYPE_SUBTYPE,
                PURPLE_SUBTYPE_CONVERSATION),
        purple_value_new_outgoing(PURPLE_TYPE_STRING),
        purple_value_new_outgoing(PURPLE_TYPE_STRING),
        purple_value_new_outgoing(PURPLE_TYPE_STRING),
        purple_value_new_outgoing(PURPLE_TYPE_ENUM));

purple_signal_register(plugin, "jabber-ge2e-decrypt",
        purple_value_new(PURPLE_TYPE_BOOLEAN), 4,
        purple_value_new(PURPLE_TYPE_SUBTYPE,
                PURPLE_SUBTYPE_ACCOUNT),
        purple_value_new(PURPLE_TYPE_STRING),
        purple_value_new(PURPLE_TYPE_STRING),
        purple_value_new_outgoing(PURPLE_TYPE_STRING));
```

Listing 7.5: Definition of the XMPP ge2e signal

The Listing 7.5 shows code to register a signal called *jabber-grok-encrypt* and *jabber-grok-decrypt* with pidgin and associated with the XMPP protocol plugin. The signal takes arguments which contain pointers to the text data being sent in Pidgin as well as the attributes that the **ge2e** message should have. This is defined in LIBPURPLE/PROTOCOL/JABBER/LIBXMPP.C in the pidgin source. The encryption plugin can then use `purple_signal_connect` to connect a handler function to these signals which takes the input text data and encrypts or decrypts it before returning.

```
static gboolean
grok_encrypt_cb (PurpleConversation *conv,char **encrypt,
        char **msg, char **u_msg, unsigned int *type, void *data)
{
        assert (u_msg);          // Unencrypted message
        assert (conv);           // The current conversation
        assert (msg);            // Text being sent
        assert (data)            // Memory structure for group
           encryption
        char *e;
        PurpleAccount *account;

        account = purple_conversation_get_account(conv);
        assert (account);

        // Use group encryption plugin to encrypt msg
        // Store unencrypted messages in u_msg
        if (send_msg (conv, msg, u_msg, (grok_i_t **) data,
           "chat") < 0)
        {
                if (*msg) {
                        free (*msg);
                        *msg = 0;
                }
                if (*u_msg){
                        free (*u_msg);
                        *u_msg = 0;
                }
                return FALSE;
        }

        // Sets the encrypt attribute for the ge2e element to
            'grok'
        // and the type of data to GE2E_TEXT
        g_free(*encrypt);
        e = "grok";
        *encrypt = e;
        *type = 0x1;    // GE2E_TEXT = 1 DATA = 2

        return TRUE;
}
```

Listing 7.6: Encryption signal handler

The encryption plugin callback shown in Listing 7.6 is connected in the implementation at PIDGIN/PLUGINS/GROK/SIGNAL_HANDLERS.C. It changes the message sent to the signal and returns. Finally, we show in Listing 7.7 where the XMPP protocol calls the signal to potentially encrypt the sent chat data before wrapping XMPP elements around the message and sending it on the network.

```
        ...In function: jabber_message_send_chat
        // Emit the signal: emsg changed to encrypted text
        // if signal is connected
        jm->encryptOn = GPOINTER_TO_INT(
        purple_signal_emit_return_1(plugin,"jabber-ge2e-encrypt",
                chat->conv,&encrypt,&emsg,&u_msg,&jm->g_type)
        );

        jm->g_encrypt = g_strdup(encrypt);

        // If the message was encrypted and stored in emsg
        // then wrap the emsg and send it.
        if( jm->encryptOn == FALSE || !strcmp(emsg,msg))
        {
                if(emsg == NULL)
```

```
                    return 0;
            buf = g_strdup_printf("<html
            xmlns='http://jabber.org/protocol/xhtml-im'>
            <body xmlns='http://www.w3.org/1999/xhtml'>
            s</body></html>", emsg);
        }

        purple_markup_html_to_xhtml(buf, &jm->xhtml, &jm->body);

        // Send the message adding XMPP elements then free it
        jabber_message_send(jm);
        jabber_message_free(jm);
```

Listing 7.7: XMPP Protocol emitting the signal to use group secure communications

Upon receiving the data, the process is reversed as the XMPP elements are parsed and a signal is emitted to decrypt the data. The group secure communications plugin catches the signal and decrypts to create plaintext. The text is then passed to the functions that display data on the client's screen. The message encryption process is detailed in Figure 7-3.

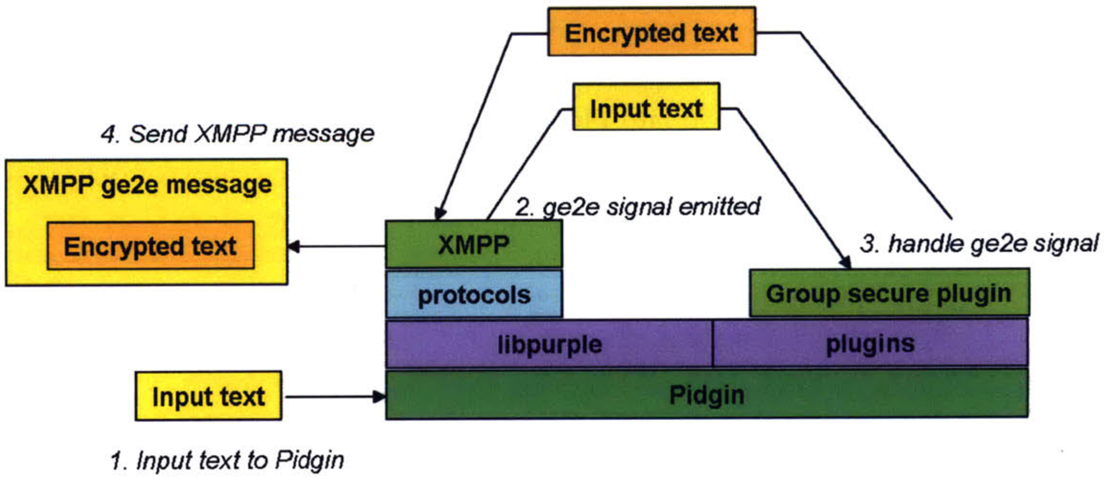

Figure 7-3: Detailing the implementation of the ge2e element in Pidgin

7.4.3 Subset Implementation

Enabling subsets requires changes on both Pidgin, the XMPP client, and the XMPP Server, which we chose to be an Openfire server [9]. On the client, we use a command interface. We defined an XMPP command for XMPP subsets which fires if the user chooses to send to a subset within the chatroom. As a reference implementation, we implemented this like a chatroom command. If the user types `/subset` in their chatroom window followed by the nicknames of the people to send the message to, `alice/bob/carol`, then the message, `Hello to only the subset`, is sent to only the subset of users. The Listing 7.8 shows the subset command being registered for the XMPP protocol.

```
purple_cmd_register("subset", "ws", PURPLE_CMD_P_PRPL,
        PURPLE_CMD_FLAG_CHAT | PURPLE_CMD_FLAG_PRPL_ONLY,
        "prpl-jabber", jabber_cmd_chat_subset_msg,
        _("subset &lt;subset list&gt; &lt;message&gt;
        : Send a message to a subset of the room.
        Subset list is a list of nicknames to
        send to separated by '/' and preceded by '~'
        if it is a list of users to exclude."), NULL);
```

Listing 7.8: Registering a command for subsets in XMPP

This command signals the signal handler, `jabber_cmd_chat_subset_msg` whenever the string `/subset` is typed into Pidgin while using XMPP. This signal handler is detailed in Listing 7.9. Both these listings are in LIBPURPLE/PROTOCOLS/JABBER/JABBER.C

```
static PurpleCmdRet jabber_cmd_chat_subset_msg
        (PurpleConversation *conv, const char *cmd,
        char **args, char **error, void *data)
{
        JabberChat *chat = jabber_chat_find_by_conv(conv);
        if (!chat)
                return PURPLE_CMD_RET_FAILED;

        PurpleConnection *gc = purple_conversation_get_gc(conv);
        JabberStream *js = gc->proto_data;

        // Must have two arguments
        if(args[0] == NULL || args[1] == NULL)
                return PURPLE_CMD_RET_FAILED;

        // The subset of user is the first argument
        // and stored in js->subset
        js->subset = args[0];

        // The message is stored in args[1] and sent to
        // the chatroom.
        jabber_message_send_chat(gc, chat->id, args[1], 0);

        js->subset = NULL;
        return PURPLE_CMD_RET_OK;
}
```

Listing 7.9: Subset command handler in XMPP

We modified XMPP to add the subset attribute to a message when the `subset` command was given. The modified message is then sent to the Openfire XMPP server, which parses the message in the call to `processMessage`. If there is a subset of users to send to, `processMessage` in Listing 7.11 sends the message to the subset of users specified by the *subset* attribute. Then in Listing 7.10 the server throws a `PacketRejectedException` in `interceptPacket` to stop the message from being routed to any other users connected to the chatroom. This protocol is located in the MUCSubsetPlugin.java file.

```
public void interceptPacket(Packet packet, Session session,
    boolean read,
        boolean processed) throws PacketRejectedException
{
        // Create a copy of the packet
        Packet original = packet.createCopy();
        boolean isMUCSubset = false;
        if(m_pluginEnabled){
                if (packet instanceof Message){
                        // process the message return
                        // true if it is a subset message
                        isMUCSubset =
                           processMessage((Message)packet, read);
                }
                // If not a subset forward as usual
                if(!isMUCSubset){
                        packet = original;
                }
                // Drop the packet if it is a subset message
                else{
                        throw new PacketRejectedException("message
                                dropped");
                }
        }
}
```

Listing 7.10: Subset message packet handling on the Openfire server

```
private boolean processMessage(Message message,boolean read)
{
        ...declare variables

        from = message.getFrom();
        to = message.getTo();

        ...initialize variables ...

        // Get the subset elements
        subset = message.getElement().attributeValue("subset");

        // determine if its an exclude subset
        if(subset.charAt(0) == '~')
        {
                exclude = true;
                subset = subset.substring(1);
        }
        room = m_muc.getChatRoom(to.getNode());
        st = subset.split("/");

        if ((st!=null)  && room!=null )
        {
        processed = true;

        // Iterate through all room occupants
        for(int i = 0;i<st.length;i++) {
                m_temp = message.createCopy();
                occupants = room.getOccupants();
                for(MUCRole occupant : occupants){
                        if(occupant.getNickname().equals(st[i]))
                        {
                                inroomto =
                                occupant.getUserAddress();
                                break;
                        }
                        // send to everyone in the subset
                        if(inroomto != null){
                                m_temp.setTo(inroomto);
                                m_temp.setFrom(new
                                    JID(node,domain,resource));
                                messageRouter.route(m_temp);
                                inroomto = null;
                        }
                }
        }
        }
        return processed;
}
```

Listing 7.11: Processing subset messages on Openfire

Openfire uses a plugin mechanism which allowed us to implement these changes on top of the existing XMPP code for Openfire. If multiple plugins catch the signal, the plugins are executed on the messages in order of priority, which is defined when registering the plugin. If the message is thrown away in this subset plugin, then it will not be accessible for later plugins. Plugins always run in a priority lower than the native Openfire code, however. Figure 7-4 shows how a subset command message from Pidgin uses the handler function to attach the subset attribute. The Openfire server then uses that attribute to route the message to the appropriate user membership.

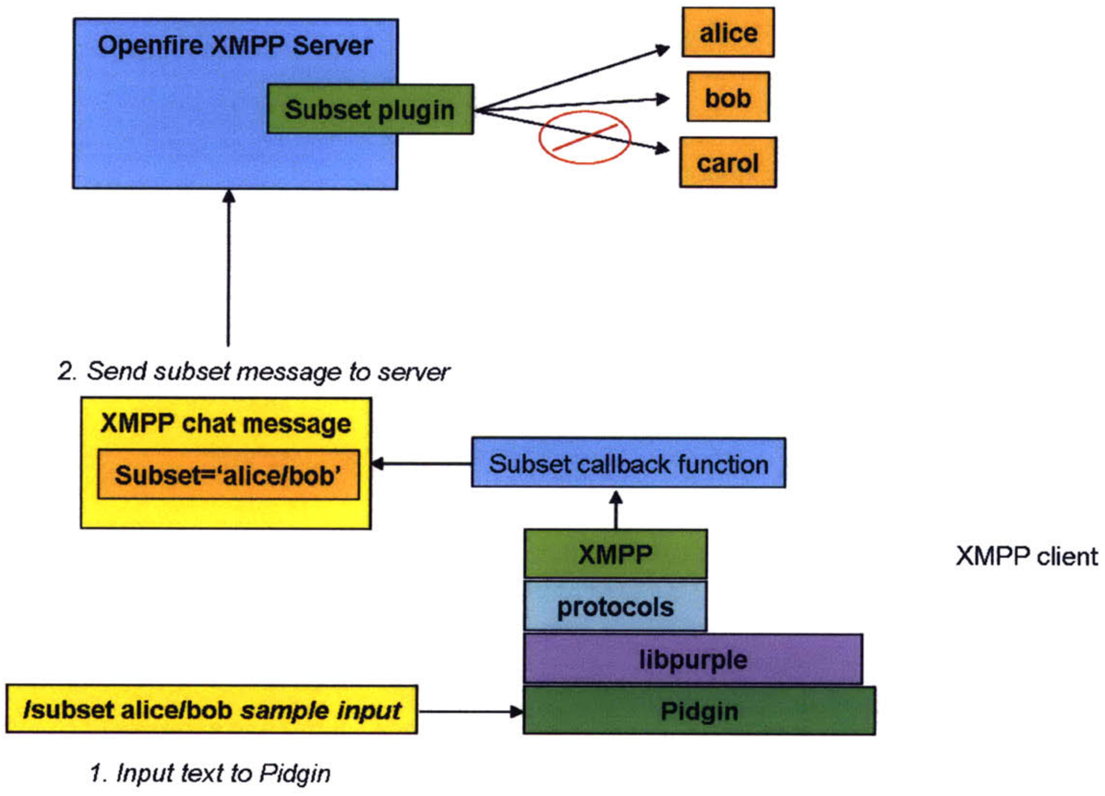

Figure 7-4: Subset message creation and routing on XMPP client and server

7.5 Evaluation and Conclusions

These XMPP chat messages were deployed on a military exercise called CAPSTONE II [6]. The exercise contained a disadvantaged network with low-bandwidth, high-latency links. Each site had users connected to a geographically local server and servers connected to each other over the disadvantaged network. Approximately 30 users exercised the chat system and sent approximately 2500 messages. The `ge2e` elements were used to encapsulate group secure communications. Unfortunately, subsets were not used during the exercise because the implementation was not complete by the exercise start date, but functionally worked in isolated testing where users sent subset messages to a local server and a local chatroom.

We implemented a group security `ge2e` element and associated `jabber-ge2e-encrypt` and `jabber-ge2e-decrypt` signals to allow for group encryption schemes to interface with XMPP. This creates an extensible mechanism for using secure group communications on any XMPP supported application.

We also created an optimization for XMPP and multi-user chat to eliminate redundant message transmission. The implementation on the XMPP client and server may reduce XMPP overhead for secure group communications in situations where large messages need to be sent to a subset of the membership of a chatroom.

Chapter 8

Conclusions

In this thesis, we investigated methods for optimizing and implementing standardized cryptographic message formats for the use in disadvantaged networks. Both methods, ZLIB-compression with a static, preplaced dictionary and CMS Lite, reduce message-size overhead of CMS messages. The former uses generic data compression and is easy to implement. CMS Lite implements domain-specific, hand-crafted compression, and as such can potentially outperform the generic approach; on the other hand, implementing CMS Lite is incomparably harder. A promising middle-ground for future research is to combine the two approaches: process CMS messages to consolidate all the formatting and configuration parts into one section and all the random, ciphertext parts into the other, and run zlib with a preplaced dictionary over the former part; then on the deflation side, reverse the process.

8.1 What I Learned

While working on this thesis project, I learned more about C programming and the use of GNU tools such as `gdb` and `Autotools`. I examined the Pidgin and OpenSSL source code and developed extensions using the CMS and ASN.1 APIs in OpenSSL while creating plugins for Pidgin. I developed better coding practices and became proficient in the `vim` and `ctags` text editor tools for programming in Linux. Finally, I freshened up my knowledge of LaTeX by writing this thesis.

Appendix A

CMS ASN.1 Library Listing

```
 ASN1_CHOICE(CMS_LiteData) = {
        ASN1_IMP(CMS_LiteData, params.signedParams,
           CMS_Signed_params, 0),
        ASN1_IMP(CMS_LiteData, params.envelopedParams,
           CMS_Enveloped_params, 1),
        ASN1_IMP(CMS_LiteData, params.digestedParams,
           CMS_Digested_params, 2),
        ASN1_IMP(CMS_LiteData, params.encryptedParams,
           CMS_Encrypted_params, 3),
  ASN1_IMP(CMS_LiteData, params.authenticatedParams,
  CMS_Authenticated_params, 4),
        ASN1_IMP(CMS_LiteData, params.compressedParams,
           CMS_Compressed_params,5)
} ASN1_CHOICE_END(CMS_LiteData)
```

Listing A.1: CMS LiteData

```
 ASN1_NDEF_SEQUENCE(CMS_Signed_params) = {
        ASN1_SIMPLE(CMS_Signed_params, version, LONG),
        ASN1_SIMPLE(CMS_Signed_params, contentType, LONG),
        ASN1_SIMPLE(CMS_Signed_params, eContent,
           ASN1_OCTET_STRING),
        ASN1_SET_OF(CMS_Signed_params, signerInfos,
           CMS_SignerInfo_params)
} ASN1_NDEF_SEQUENCE_END(CMS_Signed_params)

ASN1_SEQUENCE(CMS_SignerInfo_params) = {
        ASN1_SIMPLE(CMS_SignerInfo_params, version, LONG),
        ASN1_SIMPLE(CMS_SignerInfo_params, sidHash, LONG),
        ASN1_SIMPLE(CMS_SignerInfo_params, type, LONG),
        ASN1_SIMPLE(CMS_SignerInfo_params, parameter, LONG),
        ASN1_IMP_SET_OF_OPT(CMS_SignerInfo_params, signedAttrs,
           X509_ATTRIBUTE,
0),
        ASN1_SIMPLE(CMS_SignerInfo_params, signature,
           ASN1_OCTET_STRING)
} ASN1_SEQUENCE_END(CMS_SignerInfo_params)
```

Listing A.2: CMS Signed_params : SignedData

```
 ASN1_NDEF_SEQUENCE(CMS_Enveloped_params) = {
        ASN1_SIMPLE(CMS_Enveloped_params, version, LONG),
        ASN1_SET_OF(CMS_Enveloped_params, recipientInfos,
  CMS_Recipient_params),
        ASN1_SIMPLE(CMS_Enveloped_params, parameter, LONG),
        ASN1_SIMPLE(CMS_Enveloped_params, encryptAlg, X509_ALGOR),
        ASN1_IMP_OPT(CMS_Enveloped_params, eContent,
    ASN1_OCTET_STRING_NDEF, 0),
} ASN1_NDEF_SEQUENCE_END(CMS_Enveloped_params)

ASN1_CHOICE(CMS_Recipient_params) = {
        ASN1_SIMPLE(CMS_Recipient_params, d.ktp,
           CMS_KeyTransParams),
        ASN1_IMP(CMS_Recipient_params, d.kap, CMS_KeyAgreeParams,
           1),
        ASN1_IMP(CMS_Recipient_params, d.kekp, CMS_KEKParams, 2),
        ASN1_IMP(CMS_Recipient_params, d.pwdp, CMS_PasswordParams,
            3)
} ASN1_CHOICE_END(CMS_Recipient_params)

ASN1_SEQUENCE(CMS_KeyTransParams) = {
        ASN1_SIMPLE(CMS_KeyTransParams, version, LONG),
        ASN1_SIMPLE(CMS_KeyTransParams, type, LONG),
        ASN1_SIMPLE(CMS_KeyTransParams, sidHash, LONG),
        ASN1_SIMPLE(CMS_KeyTransParams, encryptedKey,
    ASN1_OCTET_STRING)
} ASN1_SEQUENCE_END(CMS_KeyTransParams)

ASN1_SEQUENCE(CMS_KeyAgreeParams) = {
        ASN1_SIMPLE(CMS_KeyAgreeParams, version, LONG),
        ASN1_EXP(CMS_KeyAgreeParams, originator,
    CMS_OriginatorIdentifierOrKey, 0),
        ASN1_EXP_OPT(CMS_KeyAgreeParams, ukm, ASN1_OCTET_STRING,
           1),
        ASN1_SIMPLE(CMS_KeyAgreeParams, keyEncryptionAlgorithm,
  ASN1_INTEGER),
        ASN1_SEQUENCE_OF(CMS_KeyAgreeParams,
           recipientEncryptedKeys,
CMS_RecipientEncryptedKey)
} ASN1_SEQUENCE_END(CMS_KeyAgreeParams)
```

Listing A.3: CMS Enveloped_params : EnvelopedData

```
ASN1_SEQUENCE(CMS_KEKParams) = {
        ASN1_SIMPLE(CMS_KEKParams, version, LONG),
        ASN1_SIMPLE(CMS_KEKParams, kekid, CMS_KEKIdentifier),
        ASN1_SIMPLE(CMS_KEKParams, keyEncryptionAlgorithm,
    ASN1_INTEGER),
        ASN1_SIMPLE(CMS_KEKParams, encryptedKey,
           ASN1_OCTET_STRING)
} ASN1_SEQUENCE_END(CMS_KEKParams)

ASN1_SEQUENCE(CMS_PasswordParams) = {
        ASN1_SIMPLE(CMS_PasswordParams, version, LONG),
        ASN1_SIMPLE(CMS_PasswordParams, keyParam, LONG),
        ASN1_SIMPLE(CMS_PasswordParams, encryptedKey,
           ASN1_OCTET_STRING)
} ASN1_SEQUENCE_END(CMS_PasswordParams)
```

Listing A.4: EnvelopedData continued...

```
 ASN1_NDEF_SEQUENCE(CMS_Digested_params) = {
        ASN1_SIMPLE(CMS_Digested_params, version, LONG),
        ASN1_SIMPLE(CMS_Digested_params, parameter, LONG),
        ASN1_SIMPLE(CMS_Digested_params, eContent,
           ASN1_OCTET_STRING),
        ASN1_SIMPLE(CMS_Digested_params, digest,
           ASN1_OCTET_STRING)
} ASN1_NDEF_SEQUENCE_END(CMS_Digested_params)
```

Listing A.5: CMS Digested_params : DigestedData

```
 ASN1_NDEF_SEQUENCE(CMS_Compressed_params) = {
        ASN1_SIMPLE(CMS_Compressed_params, version, LONG),
        ASN1_SIMPLE(CMS_Compressed_params, parameter, LONG),
        ASN1_SIMPLE(CMS_Compressed_params, eContent,
           ASN1_OCTET_STRING),
        ASN1_OPT(CMS_Compressed_params, dictionaryDigest,
           ASN1_OCTET_STRING)
} ASN1_NDEF_SEQUENCE_END(CMS_Compressed_params)
```

Listing A.6: CMS Compressed_params : CompressedData

```
 ASN1_NDEF_SEQUENCE(CMS_Encrypted_params) = {
        ASN1_SIMPLE(CMS_Encrypted_params, version, LONG),
        ASN1_SIMPLE(CMS_Encrypted_params, contentType, LONG),
        ASN1_SIMPLE(CMS_Encrypted_params, encryptAlg, X509_ALGOR),
        ASN1_SIMPLE(CMS_Encrypted_params, eContent,
           ASN1_OCTET_STRING)
} ASN1_NDEF_SEQUENCE_END(CMS_Encrypted_params)
```

Listing A.7: CMS Encrypted_params : EncryptedData

Appendix B

CMS Optimizations Data

Test	SignedData Size	EnvelopedData Size
Original CMS Size	455.8	463
Worst Case Conditions (ZLIB no dictionary)	427.6	463
ZLIB Best Dictionary	341.2	341.4
Best Case Conditions (ZLIB repeated)	316.8	314.5
CMS Lite	410	358

Table B.1: Summary results of all optimizations on CMS

Dictionary	Signed	Enveloped	Digested	Compressed	Encrypted
Original Message	526.8	455.8	63	82	69
CMS Shortened Strings	355.8	380.6	44	65	55
CMS Full Structures	356.6	358	49	39	38
Strings + Certs	336	361.2	42	64	56
Full + Certs	341.2	341.4	38	53	38
Strings + SignerIDs + Certs	352.4	376.8	47	67	59
Full + SignerIDs + Certs	361.8	360.2	40	55	40

Table B.2: Dictionary ZLIB compression results

Dictionary	Signed	Enveloped	Digested	Compressed	Encrypted
Original Message	270.8	199.8	43	82	69
CMS Shortened Strings	99.8	124.6	24	65	55
CMS Full Structures	100.6	102	29	38	39
Strings + Certs	80	105.2	22	64	56
Full + Certs	85.2	85.4	18	53	38
Strings + SignerIDs + Certs	96.4	120.8	27	67	59
Full + SignerIDs + Certs	105.8	104.2	20	55	40

Table B.3: Dictionary ZLIB overhead compression results

CMS Structure	Original CMS	CMS Lite
SignedData	526.8	410
EnvelopedData	455.8	358
DigestedData	63	34
CompressedData	82	72
EncryptedData	69	50

Table B.4: CMS Lite optimization results

CMS Structure	Original CMS	CMS Lite
SignedData	270.8	154
EnvelopedData	199.8	102
DigestedData	43	14
CompressedData	82	72
EncryptedData	69	50

Table B.5: CMS Lite overhead optimization results

Bibliography

[1] Cryptlib Security Software Development Toolkit. http://www.cryptlib.com/.

[2] Exuberant Ctags. http://ctags.sourceforge.net/.

[3] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[4] Internet Explorer. http://www.microsoft.com/windows/internet-explorer/default.aspx.

[5] Kile - An Integrated LaTeX Environment. http://kile.sourceforge.net/.

[6] MILSTAR. http://www.lockheedmartin.com/products/Milstar/index.html.

[7] Mozilla Firefox. http://www.mozilla.com/en-US/firefox/personal.html/.

[8] Network Centric Operations. http://en.wikipedia.org/wiki/Network-centric_warfare.

[9] Openfire. http://www.igniterealtime.org/projects/openfire/index.jsp.

[10] Pidgin, the universal chat client. http://www.pidgin.im/.

[11] The Apache Software Foundation. http://www.apache.org/.

[12] The GNU Operating System. http://www.gnu.org/.

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

[14] Vim the Editor. http://www.vim.org/.

[15] VMware Workstation. http://www.vmware.com/products/ws/.

[16] Web Services Architecture. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[17] WSS4J. http://ws.apache.org/wss4j/.

[18] XEP-001: XMPP Extensions. http://xmpp.org/extensions/xep-0001.html.

[19] XEP-0045: Multi-User Chat. http://www.xmpp.org/extensions/xep-0045.html.

[20] Information Technology - Abstract Syntax Notation One (ASN.1) ASN.1 encoding rules: XML Encoding Rules. Recommendation X.693, December 2001.

[21] Information Technology - Abstract Syntax Notation One (ASN.1) ASN.1 encoding rules. Recommendation X.680-X.695, July 2002.

[22] Information Technology - Abstract Syntax Notation One (ASN.1) ASN.1 encoding rules: Distinguished Encoding Rules. Recommendation X.690, July 2002.

[23] A. Nadalin, C. Kaler, R. Monzillo, P. Hallam-Baker. Web Services Security: SOAP Message Security 1.1. OASIS, February 2006.

[24] Adler, M. zlib 1.1.4 Manual. `http://zlib.net/manual.html`.

[25] S. Bellovin and R. Housley. Guidelines for Cryptographic Key Management. RFC 4107 (Best Current Practice), June 2005.

[26] S. Bradner. The Internet Standards Process – Revision 3. RFC 2026 (Best Current Practice), October 1996. Updated by RFCs 3667, 3668, 3932, 3979, 3978, 5378.

[27] S. Bradner. The Internet Standards Process – Revision 3. RFC 2026 (Best Current Practice), October 1996. Updated by RFCs 3667, 3668, 3932, 3979, 3978, 5378.

[28] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[29] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996.

[30] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008.

[31] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[32] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), November 1996. Updated by RFCs 2184, 2231, 5335.

[33] Greg Goth. Key Management Standards Hit the Fast Track. *IEEE Distributed Systems Online*, 8(9), 2007.

[34] R. Housley. Cryptographic Message Syntax (CMS). RFC 3369 (Proposed Standard), August 2002. Obsoleted by RFC 3852.

[35] R. Housley. Cryptographic Message Syntax (CMS). RFC 3852 (Proposed Standard), July 2004. Updated by RFCs 4853, 5083.

[36] R. Housley. Using Cryptographic Message Syntax (CMS) to Protect Firmware Packages. RFC 4108 (Proposed Standard), August 2005.

[37] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and CRL Profile, 1999.

[38] Huffman, D. A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[39] B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315 (Informational), March 1998.

[40] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301 (Proposed Standard), December 2005.

[41] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001. Obsoleted by RFC 5321, updated by RFC 5336.

[42] J. Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC 1421 (Historic), February 1993.

[43] M. Myers, X. Liu, J. Schaad, and J. Weinstein. Certificate Management Messages over CMS. RFC 2797 (Proposed Standard), April 2000. Obsoleted by RFC 5272.

[44] National Institute of Standards and Technology. *FIPS PUB 197: Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, November 2001.

[45] National Institute of Standards and Technology. *FIPS PUB 180-2: Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, August 2002.

[46] R. Perlman. An Overview of PKI Trust Models. *IEEE Networks*, 13(6):38–43, 1999.

[47] B. Ramsdell. S/MIME Version 3 Message Specification. RFC 2633 (Proposed Standard), June 1999. Obsoleted by RFC 3851.

[48] R. Rivest. The MD5 Message-Digest Algorithm. RFC 1321 (Informational), April 1992.

[49] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[50] S. Cantor, J. Kemp, R. Philpott, E. Maler. Security Assertion Markup Language v2.0. OASIS, March 2005.

[51] P. Saint-Andre. End-to-End Signing and Object Encryption for the Extensible Messaging and Presence Protocol (XMPP). RFC 3923 (Proposed Standard), October 2004.

[52] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core, October 2004. `http://xmpp.org/rfcs/rfc3920.html`.

[53] J. Schaad and M. Myers. Certificate Management over CMS (CMC). RFC 5272 (Proposed Standard), June 2008.

[54] J. Schaad and M. Myers. Certificate Management over CMS (CMC). RFC 5272 (Proposed Standard), June 2008.

[55] J. Schaad and M. Myers. Certificate Management over CMS (CMC): Transport Protocols. RFC 5273 (Proposed Standard), June 2008.

[56] S. Turner. CMS Symmetric Key Management and Distribution. RFC 5275 (Proposed Standard), June 2008.

[57] Ziv, J. and Lempel, A. A Universal Algorithm for Sequential Data Compression. *Information Theory, IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.