

Securing Group Communication in Dynamic, Disadvantaged  
Networks: Implementation of an Elliptic-Curve  
Pairing-Based Cryptography Library

by

Rob Figueiredo

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

Master of Engineering

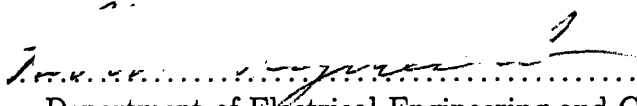
at the

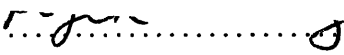
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

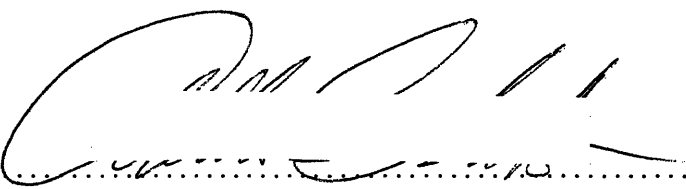
September 2006

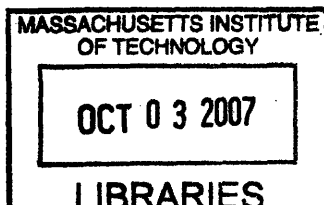
© Rob Figueiredo, MMVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly  
paper and electronic copies of this thesis document in whole or in part.

Author .....  .....  
Department of Electrical Engineering and Computer Science  
September 1, 2006

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

Accepted by .....  .....  
Arthur C. Smith  
Chairman, Department Committee on Graduate Students



ARCHIVES



# Securing Group Communication in Dynamic, Disadvantaged Networks: Implementation of an Elliptic-Curve Pairing-Based Cryptography Library

by  
Rob Figueiredo

Submitted to the Department of Electrical Engineering and Computer Science  
on September 1, 2006, in partial fulfillment of the  
requirements for the degree of  
Master of Engineering

## Abstract

This thesis considers the problem of securing communication among dynamic groups of participants without relying on an online group keying service. As a solution, we offer the design and implementation of the Public Key Group Encryption (PKGE) service. It is a cryptography library, written in C, and designed to be shared among all communications applications on any particular system. PKGE imposes low communication overhead and embraces disconnected operation, making it especially appropriate for deployment in low-bandwidth tactical environments.

PKGE provides forward-secure confidentiality and authentication among any subset of users using small communication overhead by bringing together a number of modern cryptographic developments, with the *pièce de résistance* being the elliptic curve-based Collusion-Resistant Broadcast Encryption<sup>1</sup>.

The focus of this thesis is primarily the engineering and synthesis of known theoretical schemes; we also present novel extensions to the Boneh-Gentry-Waters encryption scheme.

1. *Forward secrecy*: Add forward secrecy to the scheme at a cost of  $T$  private keys for  $T$  security epochs.
2. *Optimized session protocols*: Sidestep the majority of costs in computation and bandwidth.
3. *Cheap over-provisioning of system capacity*: Support up to  $2^{32}$  users for resource costs proportional only to the number actually registered.
4. *Chosen Ciphertext Attack (CCA) Security*: Elevate security from CPA to CCA strength.

Using PKGE, we have developed a plugin for Gaim<sup>2</sup> as a motivating launch application. The plugin both demonstrates the use of PKGE and enables secure conferencing over the range of Gaim-supported protocols, including Jabber, IRC, AIM, and ICQ. PKGE and its Gaim plugin may be run and further developed under MS Windows, Mac OS X, and Linux operating systems.

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

---

<sup>1</sup>Boneh, Gentry, and Waters, "Collusion Resistant Broadcast Encryption With Short Ciphertexts and Private Keys", Crypto 2005

<sup>2</sup>A popular open-source multi-platform IM chat client.



## Acknowledgments

I would like to extend my most sincere gratitude to Roger Khazan for extreme generosity with his seemingly unbounded creativity, intelligence, and guidance. Co-conspirator Joe Cooley comes in a close second for generously shouldering much of the implementation burden and lending insight to innumerable nests of gnarled code. Additionally, this project would not have been possible without the cryptographic expertise and support of Ran Canetti at MIT CSAIL or the BCE research code by Matt Steiner and PBC library by Ben Lynn at Stanford. And finally, thank you to Dan Boneh, Craig Gentry, and Brent Waters for the fantastic crypto that makes this possible.



# Contents

<b>1</b>	<b>Underpinnings</b>	<b>23</b>
1.1	Introduction . . . . .	23
1.1.1	Motivation . . . . .	23
1.1.2	Vision . . . . .	24
1.2	System Overview . . . . .	24
1.2.1	Design Goals . . . . .	24
1.2.2	Technical Foundation . . . . .	25
1.2.3	Communication Applications . . . . .	26
1.3	Broadcast Encryption . . . . .	26
1.3.1	Keeping State . . . . .	26
1.3.2	PBC Library . . . . .	27
1.3.3	Forward Secrecy . . . . .	29
1.4	Summary . . . . .	30
1.4.1	Contributions . . . . .	30
1.4.2	Getting around . . . . .	31
<b>2</b>	<b>A Design Blossoms</b>	<b>33</b>
2.1	Stateless Group Encryption and Authentication . . . . .	33
2.1.1	Stage 1: Broadcast Encryption . . . . .	34
2.1.2	Stage 2: Symmetric Encryption . . . . .	34
2.1.3	Stage 3: Signing . . . . .	35
2.2	Assorted Topics in Cryptography . . . . .	35
2.2.1	Order of Sign and Encrypt . . . . .	35
2.2.2	Overhead of Broadcast Encryption . . . . .	35
2.2.3	Forward Secrecy . . . . .	36
2.2.4	CPA $\rightarrow$ CCA . . . . .	37
2.2.5	Key Derivation . . . . .	37
2.2.6	Authentication . . . . .	38
2.2.7	Rolling Admission . . . . .	39
2.3	Stateful Protocols . . . . .	40
2.3.1	Overview . . . . .	40
2.3.2	Details . . . . .	41
2.3.3	Protocols: Properties . . . . .	44
2.3.4	Protocol Pseudocode . . . . .	45
2.4	Big Public Key Optimization . . . . .	49
2.4.1	Objective . . . . .	49
2.4.2	Candidates . . . . .	49

2.4.3	BigPK: A Scalable Block-based Public Key . . . . .	50
2.4.4	Incremental Deployment . . . . .	50
2.4.5	A Key in the Dark . . . . .	51
2.4.6	Rolling Admission, take 2 . . . . .	51
<b>3</b>	<b>The Game is Afoot</b> . . . . .	<b>53</b>
3.1	Development . . . . .	53
3.1.1	Tools . . . . .	53
3.1.2	Testing . . . . .	54
3.1.3	Libraries . . . . .	54
3.2	Directory structure and File formats . . . . .	55
3.2.1	Generating a system . . . . .	55
3.2.2	Adding users . . . . .	55
3.2.3	Generating a distribution . . . . .	56
3.3	Code Trace . . . . .	58
3.3.1	Objects and Conventions . . . . .	58
3.3.2	Initialization . . . . .	58
3.3.3	Load a user . . . . .	58
3.3.4	Add a recipient . . . . .	60
3.3.5	Seal a message . . . . .	60
3.3.6	Unseal a message . . . . .	60
3.3.7	Shutdown . . . . .	61
3.4	Implementation of Stateful Protocols . . . . .	62
3.4.1	Structure of a Protocol . . . . .	62
3.4.2	Getting a Session . . . . .	63
3.4.3	Add private Information . . . . .	64
3.4.4	Get the symmetric key . . . . .	64
3.4.5	Add public information . . . . .	64
3.4.6	Send the message . . . . .	64
3.4.7	Read the public header . . . . .	65
3.4.8	Get the symmetric key . . . . .	67
3.4.9	Get the private protocol information . . . . .	67
3.4.10	Session Hash . . . . .	67
3.4.11	Error Handling . . . . .	68
3.4.12	Message Format . . . . .	69
3.4.13	Optimizations . . . . .	70
3.5	Forward Secrecy: Implementation . . . . .	78
3.5.1	Mapping . . . . .	78
3.5.2	Keys . . . . .	78
3.5.3	Expiration . . . . .	79
3.5.4	Updates . . . . .	79
3.6	BigPK: Implementation . . . . .	83
3.6.1	Setup . . . . .	83
3.6.2	Calculating with <i>Squarings</i> . . . . .	84
3.6.3	Adding Users . . . . .	84
3.6.4	Usage . . . . .	84
3.7	Gaim-PKGE . . . . .	85
3.7.1	Overview . . . . .	86



3.7.2	Setup . . . . .	87
3.7.3	PINs . . . . .	87
3.7.4	Seal and Unseal . . . . .	88
3.7.5	Preferences . . . . .	88
3.7.6	Certificate Management . . . . .	88
<b>4</b>	<b>What Have We Done?</b>	<b>91</b>
4.1	Testing . . . . .	91
4.1.1	It's Alive! . . . . .	91
4.1.2	Nonsense . . . . .	92
4.1.3	Session Inspection . . . . .	92
4.2	Evaluation . . . . .	94
4.2.1	Methods . . . . .	95
4.2.2	Responsibility . . . . .	95
4.2.3	Unit Profiles . . . . .	96
4.2.4	Framework . . . . .	97
4.2.5	Results . . . . .	99
4.2.6	Live . . . . .	99
4.2.7	Comparison . . . . .	99
4.2.8	Evaluation Finale . . . . .	100
4.3	Conclusion . . . . .	104
<b>A</b>	<b>Terminology</b>	<b>105</b>
<b>B</b>	<b>Administrator Manual</b>	<b>107</b>
B.1	Configure . . . . .	107
B.1.1	An Elliptic Curve to Call Your Own . . . . .	107
B.2	Generate . . . . .	108
B.3	Add Users . . . . .	108
B.4	Distribute . . . . .	108
B.5	Conclusion . . . . .	109
<b>C</b>	<b>Client Manual</b>	<b>111</b>
C.1	Client API . . . . .	111
C.2	Initialization . . . . .	112
C.3	Groups . . . . .	113
C.4	Certificates . . . . .	113
C.5	Cryptographic Operations . . . . .	114
C.6	Utility . . . . .	115
C.7	Error Handling . . . . .	115
C.8	<code>pkge_shutdown()</code> will set you free . . . . .	115
C.9	Administrative API . . . . .	116
C.9.1	Initialization . . . . .	116
C.9.2	Users and Updates . . . . .	116
C.9.3	Serialization . . . . .	116
C.9.4	The Store . . . . .	117

<b>D</b>	<b>PBC Library 0.2 Manual</b>	<b>119</b>
D.1	Overview . . . . .	119
D.2	Testing the Library . . . . .	119
D.2.1	Quick Start . . . . .	119
D.2.2	Test Programs . . . . .	120
D.3	The PBC API . . . . .	120
D.4	Pairing Types . . . . .	121
D.4.1	Type A . . . . .	121
D.4.2	Type B . . . . .	121
D.4.3	Type C . . . . .	122
D.4.4	Type D . . . . .	122
D.4.5	Type E . . . . .	122
D.4.6	Type F . . . . .	123
D.4.7	BGN Curves . . . . .	123
<b>E</b>	<b>Cryptographic Fundamentals</b>	<b>125</b>
E.1	Symmetric . . . . .	125
E.1.1	AES . . . . .	125
E.2	Key Agreement . . . . .	126
E.3	Contributory Keying . . . . .	126
E.4	Public Key . . . . .	127
E.5	Elliptic Curves . . . . .	127
E.6	Authentication Fundamentals . . . . .	127
E.7	DSA . . . . .	128
E.8	X.509 Certificates . . . . .	128
E.9	ECDSA . . . . .	128
E.10	Identity Based Signature . . . . .	129
<b>F</b>	<b>Cryptography Options</b>	<b>131</b>
F.1	DH Key Agreement . . . . .	132
F.2	IPSec . . . . .	132
F.3	Secure Spread . . . . .	133
F.4	Identity Based Encryption . . . . .	133
F.5	Broadcast Encryption . . . . .	134
F.6	Authentication . . . . .	134
<b>G</b>	<b>Group Chat</b>	<b>135</b>
G.1	AIM . . . . .	135
G.2	Jabber . . . . .	135
G.3	Skype . . . . .	136
G.4	JXTA . . . . .	136
G.5	Distributed Center . . . . .	137
<b>H</b>	<b>PKGE File Formats</b>	<b>139</b>

<b>I</b>	<b>Technical Notes</b>	<b>143</b>
I.1	Development Environment Setup . . . . .	143
I.2	Conventions . . . . .	144
I.3	New Features . . . . .	144
I.4	Performance Optimization . . . . .	146
I.5	Cleanup . . . . .	147
<b>J</b>	<b>Listings</b>	<b>149</b>
J.1	default.cfg . . . . .	149
J.2	OpenSSL Cipher Listing . . . . .	150
J.3	OpenSSL Curves . . . . .	152



# List of Figures

1-1	Graph of $y^2 = x^3 + x$ , an example Type A elliptic curve. . . . .	28
2-1	A block diagram showing some of the stages involved in sending a PKGE message. . . . .	34
2-2	The total ordering of sessions <i>A</i> and <i>B</i> . . . . .	42
2-3	Diagram of a partially filled public key vector, for <i>i</i> registered users. . . . .	50
3-1	Graphical message component breakdown, for Stateless, Unoptimized, and Optimized messages respectively. . . . .	71
4-1	Initial message exchange before testing error channels. . . . .	94
4-2	A snippet of an evaluation script. . . . .	99
4-3	Message size overhead. . . . .	100
4-4	Message size overhead. . . . .	101
4-5	Computation required to seal a sequence of messages. . . . .	102
4-6	Computation required to unseal a sequence of messages. . . . .	103
E-1	Example STR Node Tree[49]. . . . .	127



# List of Tables

1.1	Tasks . . . . .	25
1.2	Profiles of elliptic curves supported by PBC v0.2.16. . . . .	27
2.1	Fields available for application use in X.509v1. . . . .	38
2.2	Fields responsible for functionality in a X.509v1 certificate. . . . .	38
2.3	PKGE accepts tab-delimited user information in a text file. . . . .	38
2.4	Number of random 32-bit numbers generated before reaching a 1% collision probability. . . . .	44
2.5	Structure of the <i>session</i> object. . . . .	45
3.1	File structure for newly generated “my_system”. . . . .	55
3.2	Files created when adding “r0b” and “rkh”. . . . .	56
3.3	Encrypted user keys, ready for distribution. . . . .	56
3.4	File set distribution for a new user, “r0b”. . . . .	57
3.5	Major PKGE objects and their primary purpose. . . . .	59
4.1	Scenario to ensure observed resource usage meets expectations. . . . .	93
B.1	Listings of files that Alice needs on setup and subsequent updates. . . . .	109
C.1	Error codes for <code>pkge_unseal</code> . . . . .	114
E.1	Vital information about AES. . . . .	126
E.2	Diffie-Hellman key agreement. . . . .	126
E.3	Key sizes (bits) for equivalent security. . . . .	129
H.1	OpenSSL ECDSA key serialization properties. . . . .	139
H.2	File format for administrative system file. . . . .	140
H.3	File format for a <code>.usr</code> file. . . . .	141
J.1	Comparison of elliptic curves available for the OpenSSL v0.9.8 <code>ECDSA_with_sha1()</code> signature algorithm. Times in milliseconds, sizes in bytes. . . . .	152





# Algorithms

1	Seal a message. . . . .	46
2	Unseal a message. . . . .	46
3	Encryption subroutine. . . . .	46
4	Decryption subroutine . . . . .	47
5	Safe-Get & Safe-Put: Wrapping a hash table. . . . .	48
6	Gaim-PKGE sends a message. . . . .	88
7	Gaim-PKGE receives a message. . . . .	89



# Code Snippets

1	Generating a PBC Type A elliptic curve, from <code>pbctest/genaparam.c</code> . . . . .	29
2	Example of memory mapping a file using Win32 API. . . . .	49
3	A span table node. . . . .	50
4	Structure of a protocol object. . . . .	62
5	Structure of the session's C representation. . . . .	63
6	Getting a session (using the group and timestep) under the <i>sessions</i> protocol. . . . .	65
7	Affixing private protocol information. . . . .	66
8	Retrieve the symmetric key from a session object. . . . .	66
9	Affixing public header information. . . . .	67
10	Interpreting public header information. . . . .	73
11	Interpreting header information, continued. . . . .	74
12	Interpreting header information, continued 2 . . . . .	75
13	Save the private protocol information. . . . .	75
14	Hash of a session. . . . .	76
15	Putting a session into the session table. . . . .	76
16	Functions for working with data blocks. . . . .	77
17	Forward secrecy mapping between real-world time and timestep. . . . .	78
18	Generating random taus on system setup. (in <code>curve_setup</code> ) . . . . .	79
19	Procedure for calculating keys in the forward secrecy scheme. . . . .	80
20	Procedure for deleting expired keys. . . . .	81
21	Code to generate updates to temporal keys. . . . .	82
22	The support structure for a big public key and a structure to hold successive squarings of $\alpha$ . . . . .	83
23	Implementation of ( <i>block number</i> ) $\leftrightarrow$ ( <i>public key index</i> ) mapping . . . . .	83
24	Setup procedure for the public key. . . . .	84
25	Calculating the first element of a new public key block. . . . .	85
26	Adding a user to the big public key. . . . .	86
27	Element retrieval procedure for the big public key. . . . .	86
28	Registering a callback with Gaim. . . . .	87
29	Programmatically sending messages in Gaim. . . . .	88
30	Functions to measure and test cpu/space overhead. . . . .	93
31	Helper functions for testing session handling. . . . .	95



# Console Listings

1	Top 3 lines from a sample user-info text file. . . . .	56
2	[ . . . . .	77
3	Output of <code>profpkge</code> on a Powerbook G4, 1 Ghz PowerPC, 1 GB SDRAM, OS X 10.4.6 . . . . .	97



# Chapter 1

## Underpinnings

### 1.1 Introduction

We present the Public Key Group Encryption (PKGE) system as a universal solution for efficiently securing group communications, optimized for minimal bandwidth usage. It is a shared cryptography library written in the C programming language, designed to be easily integrable with communication applications. We present its design and implementation in the pages that lie within, discussing the decisions made along with their justifications and implications.

#### 1.1.1 Motivation

First, let's elaborate on why exactly this service is so useful. The research is motivated by the Department of Defense's vision of Network-Centric Operations[29, 16] (NCO). A key part of this vision is improved information sharing and collaboration among dynamic groups of participants, or "communities of interests" [44, 60].

For example, one scenario may be a unit of soldiers on foot in a hostile urban environment. They may use handheld PDAs to relay information or act as intelligent walkie-talkies. Our solution would provide a way to secure their voice communication over the PDA ad-hoc network, where past cryptographic solutions have been prohibitive due to the communication and processing overhead inherent to the group problem.

Another military application is that of planes carrying dozens of operators that use an intra-plane wired network to communicate with each other and the plane's antenna to communicate with operators in nearby planes or on the ground. Such a hybrid network, effectively a number of small broadcast networks connected by single low-bandwidth links, should also be efficiently supported.

A third application is a command and control tactical tool to bridge forward deployed troops with rear commanders. Such a tool could allow common map annotations as well as be locationally and situationally aware.

These real-life applications require a solution for secure and authenticated communication among a dynamic group of participants, without relying on an online, external group keying service (due to the disconnected nature of some target scenarios). This can be achieved by using pre-placed data (e.g. public keys, secret keys, certificates) and/or using data exchanged by participants as part of a group keying protocol.

The ideal solution should be sufficiently versatile to work well across a wide spectrum of computing power and bandwidth. It must be viable for mobile embedded platforms with

scarce resources and unreliable connections, but robust and complete so that environments with abundant resources will not eschew it in favor of less versatile but more targeted software.

### 1.1.2 Vision

We envision our contribution to be in the form of a ubiquitous secure infrastructure that provides the services described above. All members of the Dept. of Defense would be enrolled upon employment, receiving a smart card to store their cryptographic keys. Existing users may be migrated easily since they currently carry smart cards called Common Access Cards<sup>1</sup> (CAC's) that already are used for authenticating access and digitally signing email and documents.

The ideal smart card would actually implement the cryptography functions<sup>2</sup>, providing a hardware-based API of our design and serving as a convenient black box for applications to use. The host computer simply passes bytes to be encrypted in conjunction with a set of recipient ids to the smart card, and takes the ciphertext that gets produced. Likewise for decryption. The hardware device makes use of the cryptographic scheme we present in the paper, as well as a group management protocol for optimizing bandwidth / computation usage.

We envision the DOD setting up a system with enough capacity for all of its employees, running a web server that authenticates users using their existing certificates, and allows them to download the PKGE library along with their keys. Henceforth, they have the ability to use PKGE-enabled applications. For specific applications the library may be loaded onto a smart card to take advantage of a hardware-specific implementation. In either case it serves as a shared library for all processes to use as a common service for securing communication.

## 1.2 System Overview

### 1.2.1 Design Goals

Here we cover together the operations that must be performed on a regular basis by the different participants (users, administrators, developers) in order to form design goals that will be able to most efficiently serve these tasks and requirements. These issues deal with the engineering aspects of the project. They are the practical items of importance for which our secure communication scheme must be tuned for. We have outlined the tasks that need to be performed in Table 1.1.

To these ends, we transformed the user tasks into design goals. We feel the resulting goals reflect realistic logistical requirements that are achievable by our system. Below we explicitly state the goals we had going into the design and research phases, as both our API design and underlying cryptography must support these goals in order to succeed. Note that in the context of PKGE, we use *seal* and *unseal* to refer to encryption with authentication.

---

<sup>1</sup>CAC's were designed primarily to be used as universal authentication and to enable the digital signing of email and documents. Over 5.4 million have been issued as of July 2004. Quote:

The CAC ICC will have a cryptographic co-processor to enable it to carry the PKI identity, email, and encryption certificates. ... Target Population: Approximately 4 million Active Duty military, Selected Reserve, DoD Civilian employees, and eligible Contractors. [50, 28].

<sup>2</sup>This has already been demonstrated [66, 42].



Central Authority	Users	Developers
<ul style="list-style-type: none"> <li>•Generate a group encryption system based on any set of parameters</li> <li>•Serve and Authenticate requests for private keys</li> <li>•Change keys periodically</li> <li>•Low marginal deployment cost</li> </ul>	<ul style="list-style-type: none"> <li>•Load the system on any communication device</li> <li>•Obtain keys in a scalable, convenient way from the central authority</li> <li>•Not have their use of a device be hampered by the presence of the security system</li> <li>•Able to use system correctly without special training</li> </ul>	<ul style="list-style-type: none"> <li>•Able to develop secure applications quickly and correctly</li> <li>•Be unconcerned with specifics of the cryptographic mechanisms</li> <li>•Encrypt and decrypt data for any given recipient set</li> </ul>

Table 1.1: Tasks

### Design Goals

- Reasonable system generation time and space overhead for tens of millions of users.
- Simple API structure:
  1. Single API call to seal or unseal messages e.g. `seal(recipients, message)`.
  2. Intuitive group management e.g. `add_to_group(recipients, "Rob")`.
  3. Provide access to advanced optimizations, but do not require them.
- Ability to run on current PDA-level hardware.
- Electronic key distribution.
- Versatility: desirable for all environments, from resource-scarce real-time PDA wireless communications up to luxurious broadband workstation email.
- Secure: confidentiality, authentication, forward secrecy.

### 1.2.2 Technical Foundation

The prototype is based on Boneh-Gentry-Waters (BGW) Broadcast Encryption (BCE) scheme[35] due mostly to the promise of  $O(1)$ -overhead ciphertext, as well as the relatively small space that elliptic curve keys require due to their  $2\times$  security multiplier<sup>3</sup>.

We perform necessary modifications and work out engineering issues to make the scheme practical. For example, we modify it to make it forward secure, integrate authentication in the form of X.509 certificates, discover efficient ways to handle a potentially many-gigabyte public key vector, and implement efficient session establishment protocols that obviate the processing power and bandwidth required by encryption. It is worth noting that our decision between different cryptographic schemes makes the classic tradeoff: allowing a lot of space to be used in order to optimize for bandwidth and computation. This decision is a smart one even ignoring our predisposition for bandwidth minimization: space is the cheapest commodity of any and can most easily be provisioned for.

<sup>3</sup>“Security multiplier” is the estimated factor that must be applied to symmetric key sizes in order to get equivalent security. Examples are AES: 1, Elliptic Curves: 2, RSA: 10–20[64]. See Table E.9.

### 1.2.3 Communication Applications

Although the principal contribution of this thesis is the secure communication infrastructure, we also present an archetypal application that fulfills two purposes. First, it demonstrates best programming practice for interfacing with our system in a variety of situations. Second, it is a robust, targeted communication solution for our sponsors particular usage scenarios. It is versatile enough to be the best option for both resource-scarce ad-hoc arenas such as soldiers in the field, as well as for the desktop PC with wired connection. Technology has little intrinsic value without motivating applications, so this is meant to provide one that demonstrates the power of our system.

We decided on a plugin for Gaim[10], a popular multiplatform, multiprotocol, open-source IM chat client, to fill the role. The choice to integrate with an established software package saves a lot of effort of reinventing the wheel, and it allows us to leverage their existing support for all chat protocols under the sun, as well as a large number of other plugins. In particular, by writing a single plugin, PKGE is currently usable on 7 different chat networks<sup>4</sup>, runs on Windows, Mac OS X, and Linux, and works with plugins that provide functionality such as a distributed whiteboard, voice chat, and language translation.

## 1.3 Broadcast Encryption

In this section we describe our thoughts behind using BGW (Boneh-Gentry-Waters) broadcast encryption. It is *strongly* recommended that the reader familiarize him/herself with this scheme before continuing. Despite being first published in January 2005, when we arrived on the scene in September 2005, a research implementation had already been created by Matt Steiner using Ben Lynn's Pairing-Based Cryptography (PBC) library, both graduate students at Stanford. They generously provided their code, and as a result we managed to get off the ground very quickly.

Much of our thought surrounding this scheme had to do with its overhead. Specifically, it would require a public key vector installed on every system, containing the public key of all users. In the paper, BGW suggest that 160-bit group elements would be sufficient for security in the present day. If that is the case, then for a symmetric mapping, each user's public key component would require 40 bytes. In order to support 10 million users, the public key vector would need to be  $40 \cdot 10^7 = 400$  MB. This is the theoretically best possible case — it is possible that symmetric curves having these optimal properties are not available, or that their corresponding bilinear map operations intrinsically are too slow. The different curves currently available are discussed in section 1.3.2.

This is not too much of a barrier; we can begin deploying on a smaller scale since 40 MB for 1 million users is very reasonable, and key deployment on flash drives or smart cards remains an intriguing possibility that opens up after proof of concept. Additionally, we describe in section 2.4 a technique we discovered for provisioning for many but only requiring space for as many as are actually registered.

### 1.3.1 Keeping State

Any scheme that accomplishes some sort of key agreement must be made up of some combination of *stateless* and *stateful* protocols. We define *stateless* encryption to be that which

---

<sup>4</sup>Gain-supported protocols/networks: AIM, ICQ, MSN Messenger, Yahoo!, IRC, Jabber, Gadu-Gadu, SILC, Novell GroupWise Messenger, Lotus Sametime, and Zephyr.

Type	$ \mathbb{G}_1  \times  \mathbb{G}_2  \mapsto  \mathbb{G}_T $	Bilinear Map	Symmetric?
A	$128 \times 128 \mapsto 128$	45 ms	✓
BGN	$260 \times 260 \mapsto 260$	83 ms	✓
C159	$40 \times 120 \mapsto 120$	154 ms	✗
C201	$52 \times 156 \mapsto 156$	200 ms	✗
C224	$56 \times 168 \mapsto 168$	296 ms	✗
E	$256 \times 256 \mapsto 128$	316 ms	✓
F	$40 \times 80 \mapsto 240$	884 ms	✗

Table 1.2: Profiles of elliptic curves supported by PBC v0.2.16. Sizes in bytes. Pairing times averaged over 10 `bilinear_map` operations on a 1 GHz G4 PowerPC, 1 GB SDRAM, OS X 10.4.6.

is possible without context of prior session communication. For example, I have all of my friends' RSA public keys, and without any prior communication I can encrypt a message readable only by them. Essentially, it is like a piece of addressed mail that can only be read by whomever was intended, at any point in time.

In contrast, *stateful* encryption requires session history in order to send a message to the group. One example is a Group Diffie-Hellman[61] key agreement. The initial agreement can be seen as setting up state (a symmetric key) that is subsequently used to communicate. However, if a group participant is absent for the setup phase, he is not able to participate in the communication without setting up a session again.

Typically, communication under *stateless* schemes has greater time and space overhead. This is balanced by its more robust properties as well as by the even greater cost that is incurred when events happen in *stateful* schemes that require the reestablishment of the group state: adding a recipient, for example.

The BGW broadcast encryption methods as described do not involve the formation of any state, and as a result it would be prudent to incorporate a protocol for agreeing on a session key to allow an ongoing conversation among a fixed group to take advantage of much more efficient symmetric key encryption. This should be separate from the cryptographic implementations and managed by a set of runtime-selectable protocols.

### 1.3.2 PBC Library

The Broadcast Encryption scheme makes use of the Pairing-Based Cryptography library[54]. It is very much a work in progress, on the bleeding edge of innovation (we currently use version 0.2.16). It provides elementary data types and operations for describing elliptic curves and representing their group elements. It currently supports four different types of elliptic curves, on which bilinear pairings are based, each with different properties. Work is ongoing for generating superior curves from a standpoint of faster pairings or smaller group elements. The current selection of curves is given in Table 1.3.2. Their pairing performance was measured using a script included with the library (`./report_times` in particular). For mathematical details about the underlying equations or how these curves were generated, see Appendix D.

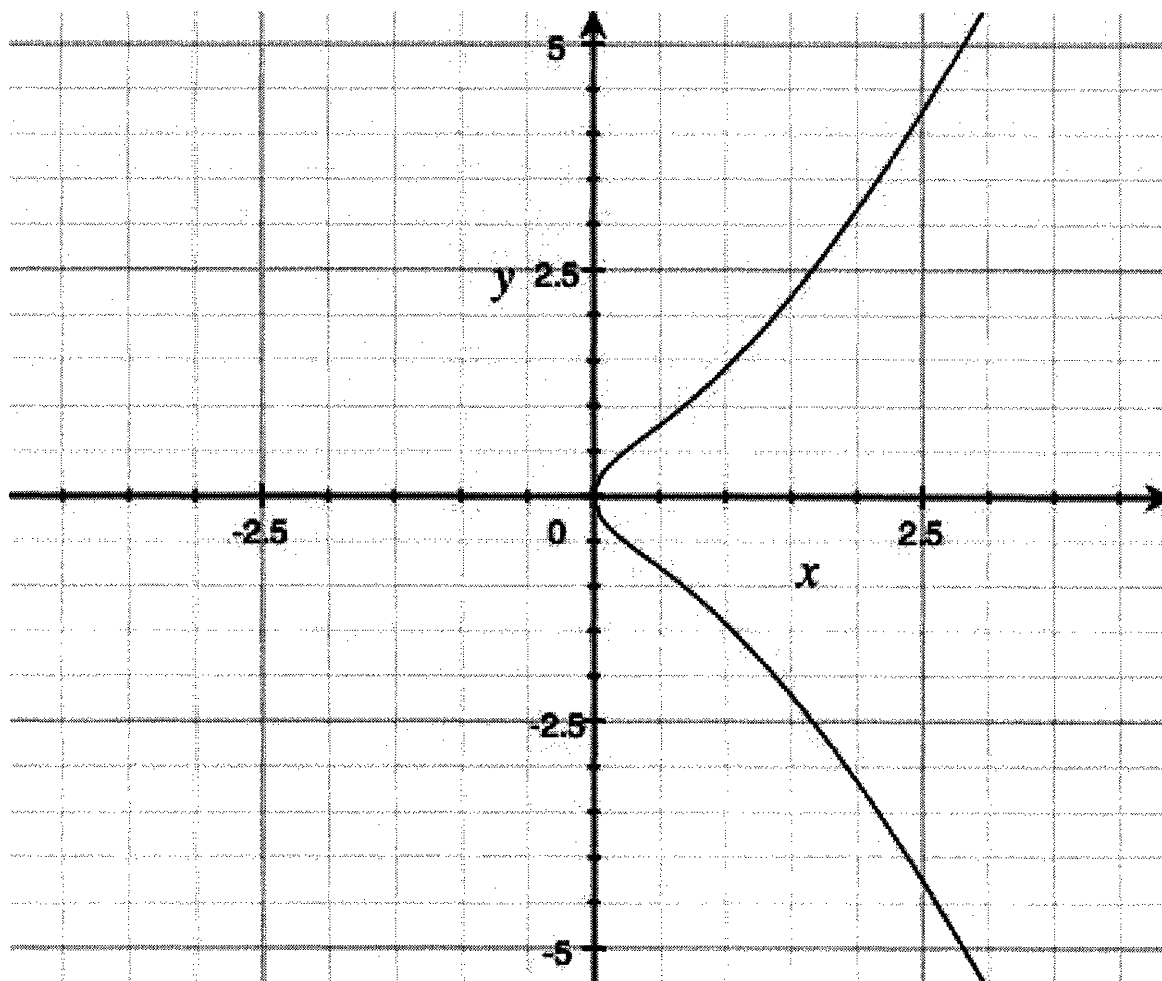


Figure 1-1: Graph of  $y^2 = x^3 + x$ , an example Type A elliptic curve.

### Point Compression

One feature of elements on an elliptic curve is that for a given  $x$ -coordinate, there are only 2 possible  $y$  values. See Figure 1.3.2 for an example. Thus, it is possible to reduce the size by half by storing only the  $x$  value of the point, along with a bit to disambiguate  $y$ . There is some small computational cost associated with having to calculate  $y$ , but the savings in bandwidth greatly outweighs the minor cost. One note is that PBC does not currently support compression for elements in  $\mathbb{G}_T$ , that is, elements that are the result of a bilinear map.

### Curve Generation

PBC additionally provides the ability to generate new elliptic curves that may then be used with PKGE. In particular, `pbcc/ecc/<curve_type>_param.c` files provide functions for generating the curve-specific parameters required by each type. Programs are even provided that use this functionality: `pbcc/test/gen<curve_type>param.c` in particular. See the example in Snippet 1.

Simple, isn't it? For more in-depth information on the mathematics behind the curve

```
a_param_t ap;

a_param_init(ap);
a_param_gen(ap, 160, 512);

a_param_out_str(stdout, ap);
a_param_clear(ap);
```

**Code Snippet 1:** Generating a PBC Type A elliptic curve, from `pbctest/genaparam.c`.

types and their method of construction see the PBC manual, attached in section D for convenience.

### 1.3.3 Forward Secrecy

Despite being one of our security requirements, there is no forward secrecy provided by the BGW scheme. Below I present two approaches for extending the scheme to allow for it. One scheme is *interactive*, meaning that it requires action on the user's part to ensure forward secrecy. The other scheme is *non-interactive*, meaning that the cryptography/system naturally evolves in such a way to guarantee this property.

#### Interactive

We can add *interactive* forward secrecy by allowing asynchronous updates to the public key vector. To begin with, we will require a minimum freshness for the system to function — the user periodically must connect to the key distribution server and update his public and private keys. Outside of these regularly-scheduled rekeyings, the user will have the option of obtaining a new key as often as he desires. This allows an adaptive degree of security across the user base, which is reasonable since the communications of ensigns may only be pertinent at the time the transmissions are made, whereas the more illustrious officials may desire more security and would have the ability to re-key to their hearts' content.

Despite implicitly assuming the re-keying to be an onus on the user, it would be possible to automate such a process to be either transparent or extremely painless, and it is a more powerful, personalized approach. For example, we could provide a utility that obtains new keys on a daily basis, prompting the user for his PIN or whatever information is necessary. In this way, it is no more painstaking than logging in to a workstation.

This entire scheme obviously results in inconsistent keys, but may be solved by the update process for an identity file storing the  $n$  previous keys that the user has had. Since they may all be timestamped, and there is a well-known forced-rekey interval, there are a bounded number of keys that a user would store in his history. At  $\approx 50$  bytes a key, it is not a large burden.

However, a more serious problem is the scalability: it requires a completely new public key for every interval. For a realistic system size of 10 million potential users, the public key vector would be at least 400 MB, which is an unacceptable cost.

It is not necessary to connect to the central authority every security interval to get new keys if a series of keys can be loaded at one time (and discarded as they expire), but this is also prohibitive. Supposing a reasonable security epoch of one day, loading enough key

data for a year would require 146 GB. Luckily, there turned out to be a superior solution, mentioned in the following section.

## Non-Interactive

A non-interactive re-key in the context of our cryptography scheme means that the system automatically cycles through keys and is able to store enough so that the user rarely has to retrieve new ones. If a way could be discovered that it could generate new keys in secure way independently of any central authority, that would be superior.

However, we have discovered a way to provide forward secrecy at a cost of storing  $T$  private keys for  $T$  security epochs, with the requirement that the keys are provided by the central authority. The users' public keys remain the same, which makes this extension a huge improvement over the naïve method of achieving forward secrecy by duplicating both the private **and** public keys. The naïve scheme selects the private/public **set** of keys for a particular security epoch, requiring  $T$  public key vectors, whereas our scheme requires only an extra set of private keys. The details are included in section 2.2.3.

## 1.4 Summary

### 1.4.1 Contributions

In this thesis, we have designed and implemented a shared C library that employs the groundbreaking  $O(1)$  ciphertext-to-any-recipient-set broadcast encryption scheme of Boneh, Gentry, and Waters in conjunction with novel extensions to provide forward-secure confidentiality and authentication for group communications. Using this library, PKGE-enabled applications can **seal** (encrypt and sign) messages in such a way that the ciphertext may be **unsealed** (decrypted and verified) only by those in the specified recipient groups.

Developing such a system required us to solve a number of engineering challenges. Some of the solutions can be considered as “novel extensions”:

#### Forward secrecy

*Add forward secrecy to the scheme for a cost of  $T$  private keys for  $T$  security epochs.*

The naïve way of adding forward security is to duplicate the private and public keys, essentially creating  $T$  full systems, associating each one with a particular security epoch, and iterating through them as convention dictates. For millions of users, our public key grows to gigabytes in size, so this solution is impractical. We have discovered a small trick that allows us to duplicate only the private keys and not the user's public keys.

#### Optimized session protocols

*Sidestep the majority of costs in computation and bandwidth.*

PKGE implements three modes of operation (*protocols*). The most basic is a **stateless** mode, in which each sealed message carries a full cryptographic header that allows it to be unsealed by any authorized recipient, before the expiration of its security epoch, independent of any other message.

Second is the **sessions** protocol which, transparently to the client application, adds session-establishment information to the standard **stateless** message. In the session protocol, the encryption/decryption keys are hashed and reused; hence, only the first session message incurs non-trivial processing time. Moreover, once the session is acknowledged by all intended recipients, the BGW headers are removed from messages. This substantially reduces the bandwidth overhead.

Third is the **optimistic** protocol, which is identical to the **sessions** protocol except the BGW header is sent once and is henceforth omitted. The **optimistic** protocol assumes reliable delivery. The optimized messages contain session ids to identify their sessions and the corresponding keys.

### Cheap over-provisioning of system capacity

*Support up to  $2^{32}$  users for resource costs proportional only to the number actually registered.*

The BGW cryptography scheme does not allow for changing the capacity of the system after the **setup** phase, which puts the administrator into a quandary: by how much should he over-provision the system? Raising the ceiling on the number of users who may register offers more flexibility and fault-tolerance, but every extra user requires more up-front resources in terms of public key elements that all users are required to store.

We have developed an efficient solution to this problem by using a block-based public key (with, for example, 64 elements per block), and calculating which elements are needed for the current membership. It turns out you can get away with only generating 3 public key elements per new user<sup>5</sup>, allowing the administrator to provision the maximum supported size ( $2^{32}$  users) without worrying about unnecessary resource drain.

### Chosen Ciphertext Attack (CCA) Security

*We outline a design and proof to upgrade the security from CPA to CCA strength (over the standard broadcast encryption).*

BGW is proven to be secure against Chosen Plaintext Attacks. We outline an approach for achieving security against Chosen Ciphertext Attacks by combining BGW and authentication. This approach is simpler than the extension proposed by the authors of BGW. However, our approach is still a work-in-progress and is not part of the current PKGE implementation.

#### 1.4.2 Getting around

In Chapter 2, we begin describing the system by considering the process that a message goes through to be secured. We get more and more in depth, even providing pseudo-code for our protocol procedures. The contributions mentioned above are described in more detail in their respective sections: forward secrecy in 2.2.3, the protocols in 2.3, the over-provisioning trick in 2.4, and the CCA upgrade in 2.2.4.

Once the design is fully specified, Chapter 3 details our implementation, giving the C code for functions that implement core functionality, with associated narratives to explain any tricky parts. This is useful for developers, and it helps to make the connection between

---

<sup>5</sup>The full public key is  $2N$  elements long, for  $N$  users.

our specification and the real world. In the final chapter, we discuss tests to evaluate the performance of PKGE in terms of computational load and overhead in message size. The Appendix contains a wealth of useful information, including a glossary, a manual for system administrators, a manual for developers looking to integrate with PKGE or choose a new elliptic curve or cipher to use, and even a checklist for great new features on the horizon



## Chapter 2

# A Design Blossoms

Here we detail the design of the Public Key Group Encryption service. It is a versatile, robust toolkit designed to be the panacea for securing communication within any size of organization. More specifically, it is a shared library that abstracts away incredible cryptographic complexities from application developers to provide a simple API that can encrypt and authenticate messages to arbitrary groups of users, addressed using only a textual name present in the users' X.509 certificates.

First we discuss the cryptographic techniques used to encrypt and authenticate messages. We also justify our method of accomplishing a secure “encrypt-then-sign” and offer overviews of our forward secrecy and CPA→CCA extensions. After, we specify stateful protocols that may be layered on top to accomplish wondrous savings in computation and bandwidth. Lastly, we look at our solution to the challenge of managing a public key vector that should have the capability to scale to enormous sizes.

It is *strongly* suggested that the reader first review the Boneh-Gentry-Waters broadcast encryption scheme. Familiarity is assumed in a number of places, the forward secrecy and CPA→CCA sections in particular.

### 2.1 Stateless Group Encryption and Authentication

We begin with the particular cryptographic technologies that will ensure a messages confidentiality and authenticity. Fundamentally, three primary phases of cryptographic processing are required to secure a message (shown in Figure 2-1). After covering this sequence, we examine the various decisions that resulted in this data flow and discuss our forward secrecy and CPA→CCA extensions in the next section (2.2).

First, the user selects a set of recipients and the security epoch (or *timestep*) to address the message to. These are the necessary inputs to select the correct BCE keys and calculate the header + broadcast key.

The second stage is to symmetrically encrypt the message with the broadcast key. We use AES, and to obtain a compatible key we use the PKCS#5 key derivation algorithm on the broadcast key. The header calculated in stage 1 is appended to the encrypted message, along with the group and appropriate timestep.

For the third, and last, stage, the encrypted message and its headers are signed, and the signature is affixed. After this, the message is ready for delivery! Let's cover each of these stages in a bit more detail. A basic overview is given in Figure 2-1, although it is important to note that it offers only a simplistic view compared to the complexities seen later on.

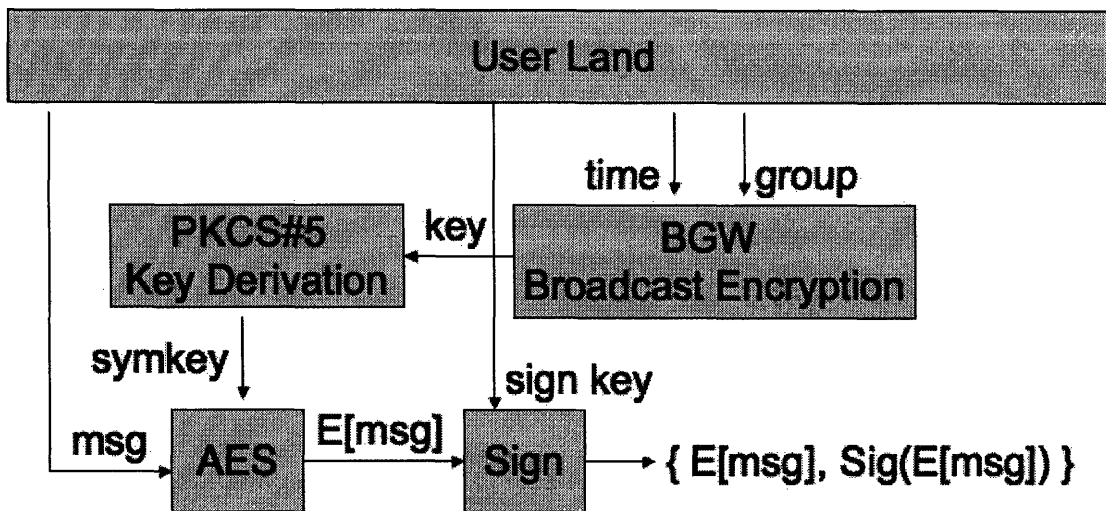


Figure 2-1: A block diagram showing some of the stages involved in sending a PKGE message.

### 2.1.1 Stage 1: Broadcast Encryption

Messages are encrypted by choosing a random value  $t$  and calculating the pairing  $e(g_n, g_1)^t$  over a user-selectable elliptic curve. The bytes from this element, combined with 8 bytes of random salt, are fed into the PKCS#5 salt-and-stretch key generation process which outputs a valid OpenSSL user-configurable symmetric key and initialization vector. These are used to encrypt the message. The salt is sent publicly with the message, along with a header calculated using an *encryption product* that includes individual recipients associated public key elements. This header allows the user's recipient list to calculate the random  $t$  selected, using their private key, and then they follow an analogous process is followed to accomplish decryption.

The elliptic curves used to compute encryption and decryption products under the BCE scheme are elements of the PBC library. Recall the characteristics of the various choices, given in Table 1.3.2. Despite the ostensible buffet of choices, we really don't have much of one. However, it is possible to generate any pairing configuration using PBC tools (see 1.3.2) to use in the PKGE configuration.

### 2.1.2 Stage 2: Symmetric Encryption

Once the broadcast encryption scheme outputs a key, along with a header that allows any other recipient to calculate that same key, we use the key with a symmetric encryption algorithm to actually accomplish confidentiality. To use the broadcast key with our symmetric cipher of choice, AES, (section E.1.1), we need to pass it through a key derivation algorithm.

First, we use OpenSSL's cryptographically-secure pseudo-random generator to harvest 8 bytes of salt. Then we send the salt + broadcast key through OpenSSL's PKCS#5 key derivation algorithm (section 2.2.5). The output is an OpenSSL-compatible symmetric key plus an initialization vector.

The time has come: we encrypt the message, along with the sender's PKGE id (see sec-

tion 2.2.1 for the reason), using AES-256 in CBC (cipher block chaining) mode, affixing the group, timestep, broadcast header, and salt. Any valid recipient can use that information to derive the same AES key and thus read the message.

### 2.1.3 Stage 3: Signing

To digitally sign a message, the SHA-1 digest is taken and processed using a 224-bit ECDSA key, authenticated to others by their X.509v1 certificate. Each certificate contains the user's "Common Name", the textual representation for a user that is the currency of group membership operations. It binds this name to their public ECDSA key that others may use to verify their signature, as well as their PKGE unique identifier which is exactly the id referenced in the BGW scheme.

## 2.2 Assorted Topics in Cryptography

The previous section gave a data-flow overview of how we use cryptographic techniques to encrypt a message. There are a number of issues relating to our cryptography use that do not fit well into such an organization, so this section is for in-depth discussion, justification for decisions, description of extensions, etc.

### 2.2.1 Order of Sign and Encrypt

The order of sign and encrypt (and the details therein) are a prickly subject that many a professional cryptographer has gotten wrong.

Charged with the task of putting together a confidential, authenticated message using the "sign" and "encrypt" primitive operations, how would you begin? This is an age-old problem in cryptography where nothing is as simple as it seems[31, 33] and there are many papers debunking previous proposals[38]; even illustrious internet standards can't seem to get it right[39]! The current (ie. correct!) solutions boil down to identifying the sender/signer on both sides of the encryption curtain. Mr Davis identifies the "secret sauce" of schemes that work in his comprehensive report on the subject:

... a common feature: when signing and encryption are combined, the inner crypto layer must somehow depend on the outer layer, so as to reveal any tampering with the outer layer.[39]

We take the encrypt-then-sign approach, identifying the sender on the outside of the encryption, so recipients know whose public key they must use to verify the signature, and on the inside so that the message may not be *surreptitiously forwarded*.

For a while we considered the convention that the sender **must** be included in the recipient list for a message to be valid (instead of including the sender id separately on the inside of the encryption). It turns out this is a valid solution to the problem of outsiders surreptitiously forwarding a message as their own, but malicious group members would still have that power. In the end, we decided to simply include the sender id on the inside of the encryption, due to the increase in security for little overhead.

### 2.2.2 Overhead of Broadcast Encryption

BGW correctly note in their paper that "for any large number of receivers, decryption time will be dominated by the  $|S|-2$  group operations needed to compute  $p = \prod_{j \in S, j \neq i} g_{n+1-j+i}$ ."

However, they go on to observe that had the receiver previously computed  $w = (\text{decryption product for } S')$ , he could calculate  $p$  with just  $\delta$  group operations (add/remove a user), if  $S$  and  $S'$  differ by  $\delta$  group operations.

In other words, beginning with user  $i$ 's decryption product  $w$  for user set  $S'$ , multiplying by  $g_{n+1-j+i}$  effectively adds user  $j$  to  $S'$  and multiplying by the inverse removes him. Again we find that caching can save immense amounts of computation.

### 2.2.3 Forward Secrecy

Now we describe our novel extension of BCE to support non-interactive forward secrecy at a marginal cost of only 1 private key per *security epoch*<sup>12</sup>. The birds-eye view is that each user will get many private keys, each corresponding to a distinct security epoch. Keys are deleted as their security epoch expires, and it becomes impossible to unseal messages sealed with that epoch's set.

Recall from the BGW BCE scheme that the system wide public key has the following form:

$$PK = (g, g_1, \dots, g_n, g_{n+2}, \dots, g_{2n}, g^\gamma) \in \mathbb{G}^{2n+1}$$

where  $g_i = g^{(\alpha^i)}$  and  $\gamma$  are chosen randomly. The calculation to encrypt a message to a user  $i$  requires  $g_i$  and  $g^\gamma$ . To decrypt, a user needs his private key  $d_i = g_i^\gamma = g^{\gamma\alpha^i}$ .

We introduce a new series of variables, appropriately named  $\tau_T$  and consequently modify the encryption calculation, for the security epoch  $T$ , to use  $g^{\gamma\tau_T}$ . The math works out so that users may decrypt these messages by using their corresponding private key  $g^{\alpha^i\gamma\tau_T}$ . A consequence of this scheme is that users are given keys in batches — the maximum amount of time a user may encrypt a message to the future is referred to as the *lookahead*. This defines the number of  $\tau$ 's a user is allowed to possess ahead of time.

Thus, we generate many random  $\tau$ 's, associating each of them with a security epoch, or *timestep*. This is the maximum period of time that a message remains vulnerable to key compromise—if an adversary compromises a key right before the end of a timestep, he can read messages sent at the beginning, but as soon as the timestep elapses, they become sealed forever (unless the user stored them in plaintext!).

To restate, with an endless sequence of random  $\tau$ 's, we calculate users' private keys for each timestep and require that their old keys are irrevocably deleted once their timestep expires. That leads to a new version of the  $v$  public key element and private keys: they both become arrays with one element per timestep, swapped out automatically by the system. Specifically, for a sequence of random  $\tau \in \mathbb{Z}_p = (\tau_1, \tau_2, \dots)$

$$v = g^{\gamma\tau_t}, \quad d_{i,t} = g_i^{\gamma\tau_t}$$

**Key Roll-over** Upon expiration of a security epoch, the service begins using the new set of keys and invalidates all existing sessions. However, in order to avoid the problem of sending a message at the end of one epoch and having it rendered unreadable due to expiration a moment later, we introduce a *window*. For a certain period of time at the start

<sup>1</sup>A *security epoch* is the discrete unit of forward secrecy — messages are only rendered forward secure after expiration of the security epoch they are sealed to.

<sup>2</sup>We do not prove our result formally here, as the focus of this thesis is the implementation. It is destined for a future publication.

of an epoch, the previous epochs keys are kept for the purpose of unsealing but may not be used for sealing new messages. After this (short) time period, the previous epoch's keys are deleted. Thus we define the *key roll-over* process to be that of safely switching keys to the subsequent security epoch while using a window to fend off any problems inherent in attempting a synchronous global swap between incompatible keying material.

**Updates** Forward secrecy comes with a small price tag: from time to time users must reconnect with the central authority to retrieve a new set of keys. This may be done at any time, preferably prior to a user running out of keys. We introduce another timing parameter in conjunction: *lookahead*. This refers to how far ahead of time a user is allowed to possess keys.

An example set of forward secrecy timing parameters might be a *timestep* of one day, a *window* of 10 minutes, and a *lookahead* of 1 year.

### 2.2.4 CPA $\rightarrow$ CCA

The authors of BCE prove in their paper[35] that their scheme is CPA-secure (ie. secure against the Chosen Plaintext Attack) and fully collusion-resistant. This means that if you are not an intended recipient of a message, there is nothing you can do to read it. However, we desire the stronger promise of CCA-security (Chosen Ciphertext Attack). They do propose a CCA extension in their paper, but it requires integration with an IBE (identity-based encryption) system, which was one voyage of complexity we were unwilling to embark on.

In the context of our grander scheme, it is possible to achieve this by using a combination of the authentication mechanism, the requirement that the message sender is always included in the recipient set for a message to be valid, and including a HMAC (Hash Message Authentication Code) of the plaintext contents of the encryption, recipient set, and header, as an integrity check.

The proof works by showing that a CPA adversary can successfully simulate the system that the CCA adversary tries to break. Since the system was shown to be CPA secure, and our proof shows that the CPA adversary can interact with the system in such a way to be indistinguishable to the real system as far as the CCA adversary can tell. Thus, if the CCA adversary can be fooled into thinking the CPA adversary is actually the system, we are guaranteed CCA-security. Remember: on the internet, no one knows you're a CPA-adversary.  $\square$

### 2.2.5 Key Derivation

Key derivation is important for several purposes. Most prominently, it is used to derive a properly formatted AES key usable by OpenSSL from the BCE-generated bytes. In other words, the header is used to calculate the decryption key, which is then run through the derivation algorithm to get something to use with AES. Quite involved.

It is also used to derive a key and initialization vector (IV) from the user-provided PINs, necessary to decrypt their keys from disk. This is done in `crypto.c:password_to_key` function, using `EVP_BytesToKey`, an abstraction that implements the PKCS#5 (Password-Based Encryption) standard[47]. In particular, using a configurable hash function and with a specifiable salt and number of iterations, it takes the byte-string password and follows this procedure (from the OpenSSL documentation):

name	title	localityName	uniqueIdentifier
givenName	description	stateOrProvinceName	serialNumber
surName	organizationalUnitName	countryName	dnQualifier
commonName	organizationName		
initials			

Table 2.1: Fields available for application use in X.509v1.

notBefore	pubkey	version
notAfter	signature	

Table 2.2: Fields responsible for functionality in a X.509v1 certificate.

The key and IV is derived by concatenating  $D_1$ ,  $D_2$ , etc until enough data is available for the key and IV.  $D_i$  is defined as:

$$D_i = HASH^{count(D_{i-1})} || data || salt$$

where  $||$  denotes concatenation,  $D_0$  is empty,  $HASH$  is the digest algorithm in use,  $HASH^1(data)$  is simply  $HASH(data)$ ,  $HASH^2(data)$  is  $HASH(HASH(data))$  and so on.

The initial bytes are used for the key and the subsequent bytes for the IV.

Of note is that since the security of the scheme comes from the ability of all intended recipients (and no one else) to calculate the same bytes using BCE, the password-derivation is simply a known protocol for massaging the BCE bytes into something suitable for OpenSSL's AES to use. As such, the salt, which could itself act as a password, is sent in the clear along with the message so that the recipient, upon calculating the BCE key, can use it to derive the same AES key.

## 2.2.6 Authentication

X.509v1 certificates are issued to all users as they are added to the system. The available fields for use are shown in Table 2.1. Additionally, the certificates' correct functioning relies on the fields in Table 2.2.

PKGE uses the `commonName` field as the textual human-readable/human-rememberable identifier, and uses the certificate to bind it to the PKGE *id*, stored in `uniqueIdentifier`. Users are added to the system using a tab-delimited user information file. The format of the file is as follows: on the first line is the name of the CA. On the subsequent lines lie the information for users 1..*n* according to the format given in Table 2.2.6.

Users that do not have information entered for them in the file are left uninitialized. Users may be added at any time, and will be assigned sequential PKGE ids.

SurName	GivenName	CommonName	Title	GroupName
---------	-----------	------------	-------	-----------

Table 2.3: PKGE accepts tab-delimited user information in a text file.

## Signature Algorithm

OpenSSL 0.9.8 is the first version of OpenSSL that supports ECDSA[30] signatures. We chose this algorithm due to its smaller key sizes – signatures are  $\approx 60$  bytes instead of the equivalent-security  $\approx 256$  bytes provided by RSA signatures. Additionally, despite elliptic curve operations in general taking a long time, the fact that keys can be much shorter causes the processing times to be roughly equivalent[65].

OpenSSL provides a bevy of curves that signatures that can be used, controlling the signature size, processing time, and security properties. Since the descriptions that OpenSSL provided were Greek to me, I profiled and compared them. They are listed along with their resource demands in Table J.1.

## Hash Function

A X.509 certificate is signed by taking a digest of all of the bytes that make it up (well known byte structure is part of the standard). This much-smaller set of bytes is signed. The hash algorithm that takes the digest is selectable, but OpenSSL supports only SHA1 in conjunction with ECDSA. This is potentially problematic due to the ability to to generate different certificates with equivalent SHA1 digests[63]. OpenSSL supports SHA-256, so it is likely that the ability is already present to upgrade to a more secure hash function. (It isn't switched already because signing and digests come as a pair, e.g. we are using `ECDSA_with_sha1()`, and I imagine it would be simple for someone to add the extra permutation.) At any rate the risk is reduced to essentially 0 by simply doing a sanity check on the certificate — a discovered collision would almost certainly have gibberish in the certificate fields. Additionally, if we require that the interval of validity (e.g. *timeAfter*–*timeBefore*) is exactly the “lookahead” of the system, the probability really is completely negligible.

### 2.2.7 Rolling Admission

One item that should be explicitly addressed is the incremental nature of the system growth: users are admitted on a rolling basis. We do not expect the administrator to know all possible users at creation time, and requiring a static user system membership would be excruciatingly stifling.

However, this complicates the process of communication — you cannot possibly (non-interactively) verify the signature of someone that had joined the system after you. This would require preplaced knowledge binding his identity to his key (*non-interactive*), which is not possible to know in advance of him signing up.

It turns out that it was not our intention for everyone to store X.509 certificates (although that decision is up to the administrator) due to prohibitive space costs. Instead, PKGE will provide functionality for the client application to use for either implementing its own certificate-exchange mechanism or integrating with a general certificate management framework<sup>3</sup>. Such a mechanism is extremely common for any PKI-enabled application, and adds little complexity. The certificate may be kept for the duration of the communication and then deleted (or kept permanently as a type of long-term cache).

The take-away for this section is that PKGE explicitly allows and encourages dynamic exchange of certificates as a mechanism to reduce dependence on connected operation. We

---

<sup>3</sup>We recommend Mozilla NSS for this purpose: <http://www.mozilla.org/projects/security/nss>.

feel it is both a requirement to achieve reasonable scalability as well as a strong promoter of versatility.

## 2.3 Stateful Protocols

Until now, the Broadcast Encryption has been described as a stateless scheme requiring nothing but pre-placed data to decrypt any message you happen to come across (as long as it's addressed to you, of course!). It turns out you can do a *lot* better by building state and relying on the context of a *session* to avoid retransmitting and/or recalculating things you already knew.

We developed two very efficient protocols, one that is guaranteed to work over any sort of lossy connection, e.g. UDP, and one that is guaranteed to work on orderly connections, e.g. TCP. Henceforth we refer to the basic Broadcast Encryption scheme as the *stateless* protocol.

The overarching goal of the two stateful protocols is to make an association between a particular decryption key and a unique id. Once this association is made by all recipients, simply prefixing encrypted bytes with this id is enough to indicate which decryption key recipients should use. Since elliptic curve cryptography is only used in the calculation of the symmetric encrypting/decrypting key, this completely circumvents the most time-consuming calculations. Additionally, this saves an enormous amount of bandwidth because instead of having to include a 2-element header along with an encrypted AES key, we simply need to include some identifier that allows recipients to look up the already-calculated keys.

To do this, we introduce the extra structure of sessions. A session caches information about a particular encryption or decryption calculation, so that it may be byte-copied to outgoing messages or provided to the AES decryption. This is immediately different from the stateless approach: the stateless procedure generates a new encryption key for every message using a random parameter ( $t$  in the paper). A session saves the message header and symmetric key generated and reuses them as long as possible. Note that the longest-lived it can be is one (forward secrecy) timestep. As a result, any session-using protocol only requires significant computation from each user once for any particular group he is a member of.

The two protocols we describe differ only in when they stop including all of the session information in the header, prefixing messages exclusively with a session id. One is a pessimistic protocol: it waits until it sees equivalent headers from all members of a group before switching to the session id. The other is optimistic, sending a header exactly once before switching. There are a large number of security and fault-tolerance details that are required to make these simple protocols robust. Additionally, we offer computation and space complexity measurements achieved using these protocols under various conditions.

### 2.3.1 Overview

#### Stateless

This protocol is the basic operating mode of the BCE library. All stateless messages come with the recipient set and timestep a message is encrypted for, according to the forward secrecy scheme in section 2.2.3. The timestep is abbreviated `ct_msg` in code snippets, short for “canonical time”, in the sense that it is a timestamp that uniquely identifies a security epoch. Additionally, the sender id appears at the front of the message so that the signature



may be verified before the decryption is attempted. This is invariant across all the protocols; it is not mentioned again.

## Sessions

As previously stated, the goal is to attach single numerical identifier to the message in order to indicate to all recipients which key should be used to decrypt. To accomplish this **reliably**, all recipients must agree on an association between the *session id* and the locally cached decrypting information, and the fact that this association has been made by all recipients must itself be made known.

Our solution calls for senders to make up a new nonce, called the *session id*, *s\_id* for short. This would be included on the inside of an encrypted message, sent with full headers. When recipients receive this, they cache the header and resulting decryption key in a table, associated with the session id. Each recipient keeps track of which group members he has seen send this header, checking them off a list when they send a message to the group. That's it; it's **that** simple.

## Optimistic

The *optimistic* mode follows the same game-plan as the session protocol described above. The difference is that here the library assume that the act of sealing a message ensures that the message will reach all of the intended recipients. This presupposes that the message is sent to all of them, and that it is not lost in transit. For some applications, this is a good assumption, and for others it is terrible. At any rate, after a single header has been sent out, the switch is made to session id's only.

### 2.3.2 Details

Actually, this is one case where the devil is in the details. Here is a taste of the problems that lie just below the surface of the simplistic explanation given above:

- Session lookup
- Hiding the group
- Messages sent concurrently to the same recipient set
  - Agreeing on one
  - Disambiguating
- Error conditions—recipient receives a session id of which he has no record
- Collisions between session ids

#### Session Lookup

An observation: the encryption key depends on the set of recipients and the target decryption timestep. Thus, a session should be easily findable given this information; we enter it in the session table by hashing the recipient set in conjunction with the desired timestep. Also, since a session must be found based only on the session id when receiving messages, it is entered twice in the table.

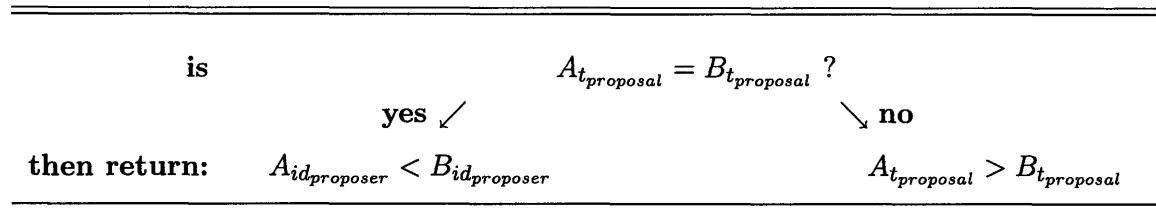


Figure 2-2: The total ordering of sessions  $A$  and  $B$ . The newer timestamp wins, with more senior id as a tiebreaker.

## Hiding the Group

Technically speaking, switching to the session id does nothing for us in terms of security, since the whole header (including the recipient set) has already travelled in the clear. However, in the usage scenarios, it does make sense to broadcast such information as little as possible. As a result, it is desirable to send the private id in the encrypted part of the message when full headers are sent, so that upon switching, no one outside of the recipient set could know the (*session id*  $\mapsto$  *header*) association. It is no accident this detail is observed first — the session id being encrypted carries implications down the road.

## Agreement

When designing peer-to-peer protocols, it is necessary to take “simultaneous” actions into account especially since the large latency between users of the system effects a large temporal window in which actions are “simultaneous”. If two users send messages to the same group, they will both generate their own headers and session ids. What is one to do when faced with two competing sessions? He has to decide.

By imposing a total ordering on sessions, everyone has a deterministic method for determining the “winning” session when given a bevy of choices. In particular, for sessions  $A$  and  $B$ ,  $A$  *precedes*  $B$  ( $A \prec B$ ) if  $time(A) > time(B)$ , where  $time(A)$  is the creation timestamp for header  $A$ : we take the most recent header as the “winner”. If the timestamps are the same, there is a tiebreaker  $user\_id(A) < user\_id(B)$ : if the user id of the proposer of  $A$  is less than (senior to) that of  $B$ , session  $A$  is declared the winner. A succinct summary of this policy is given in Figure 2.3.2. For completeness, we also specify:  $\forall S : S \prec null$ .

However, the “losing” session is not simply discarded, due to the possibility of receiving future (header-less) messages with that session id. Instead, we employ the classic technique of hash table chaining — the buckets in the session table actually contain linked lists, with the “best” session at the head. This way, when the user wants to send a message to a group, he simply calculates which bucket to look in, picking the session found at the front. When receiving a message, he looks for a corresponding session, going down the list until he finds one that matches. If none matches, he inserts it into the list.

This is a simple protocol for ensuring that users always use the “best” session, while agreeing on exactly which one that is.

## Disambiguation

Suppose we have just received two messages sent to the same recipient set. They both have headers, and propose different session ids. This poses no difficulty for the agreement algorithm, and I calmly calculate both decryption keys, store both headers under their

session ids, and begin my two-element linked list at the hash bucket serving that group, secure in the knowledge that everything will turn out alright.

Then things get tricky. *Another* message comes addressed to the group (with the full header), and I am already wheezing from the effort of calculating those keys the first time — I definitely don't want to go through *that* again! But, which session do I use? Recall that the session id is encrypted for security reasons, so I have no way to decide which session to associate this message with.

To solve this situation, we designate a second session id: the *public session id* or `pub_id` for short. This is a nonce that accompanies headers, exactly for the purpose of accomplishing this disambiguation.

## Collisions

Another class of serious faults that must be considered are collisions for the identifiers used for hash table insertion. In particular, the private session ids and the  $\{group, timestep\}$  hashes used to insert sessions into the table are bounded size and thus different sessions may randomly be identical in either property despite being unrelated. Let's look at the two cases.

*private\_id* Session ids are chosen at session creation completely at random. If a new session is created with a randomly chosen private id that already exists, it should be re-chosen. However, if a message arrives with a private id that belongs to a *different group altogether*, we have to provision for disambiguating future messages prefixed by nothing but a private id.

One possible solution to the private id collision is to establish a new session — the session agreement protocol causes all participants to agree on the newest proposal, so you could certainly suggest a session id that did not conflict. However, we decided that this situation is too unlikely (Figure 2.4) to justify the amount of extra mechanism, and we take the brute force approach of attempting to decrypt with each session that has a matching private id. This is slow, but correct and unlikely to ever be encountered.

*hash(group, ct\_msg)* To find a session in the table based on the group and timestep it is addressed to, we take a deterministic hash. In the previous section (2.3.2), we saw how sessions could collide due to being addressed to the same group and timestep — this situation is different because it is a technical failure. It results from the hash function being unable produce different output for different input, rather than the semantic challenge of how to handle the concurrency inherent in our usage model. In fact, this is more serious than the private id collision because the private ids are identifiers without meaning, whereas this hash is strictly determined by the parameters of a message.

As already observed, these collisions are exceedingly unlikely (Table 2.4), so it makes sense to use the simplest mitigation procedure. The important thing is that the result of such a collision is well-defined and correct, rather than causing some sort of catastrophic failure. Thus, our approach is simply to store collisions in the same session chain exactly as the agreement procedure (section 2.3.2) dictates. Whenever we retrieve a session, simply verify the group and timestep is the one requested, which is information we are guaranteed to have since they serve as the hash function input.

---



---


$$\begin{aligned}
& \# \text{ Numbers: } N \\
& \# \text{ Pairs: } \binom{N}{2} = (N \cdot (N - 1))/2 \\
& \text{Pr}(1 \text{ pair is equal}): 1/2^{32} \\
& \quad \downarrow \\
& \# \text{ Pairs required for a Pr( collision ) of 1\%:} \\
& \frac{1}{2^{32}} \cdot \frac{N^2 - N}{2} = 0.01 \implies \lceil N \rceil = 9269
\end{aligned}$$


---



---

Table 2.4: Number of random 32-bit numbers generated before reaching a 1% collision probability.

## Error Conditions

These protocols must overcome the same protocol fault situation: receiving a message prefixed only by a session id, of which it has no record. This situation can only be the result of some sort of fault, such as a crash or other loss of host state, or perhaps (for the optimistic protocol) a lost packet. It should be noted that an obviously correct, if mildly inefficient, solution is to indicate that the client must leave and rejoin the group, allowing the standard group join procedure to take care of things. We try to do better; there are many courses of action we considered; here are two.

1. Passive: Do a lookup for that group/timestep: if it exists, set a `force_header` flag; the next message sent from the user will cause everyone to switch to that session header via the normal session agreement protocol. If it doesn't exist, then a new session will be started on the next message anyway.
2. Active: Return an error code from `unseal`, and either have the user call a `get_error_msg` function or register a callback that accepts data to send. A "help" message is sent to the group, and they reply with the header.

We chose the active solution for two reasons. First, it is extremely extensible: any message authored by the protocol can use the channels set up by this error handling method. Second, it explicitly specifies an error recovery process the user application should go through to be included and be able to decrypt the message, assuming the un-decryptable message is saved until the user gets back in the loop. Third, and most importantly, the first solution does not allow decryption of the message, since it is based on establishing a new session, known to this user, rather than recovering the current one.

### 2.3.3 Protocols: Properties

#### Transparency

One constraint on our design for the protocols was *transparency*. The library should be able to optimize and have the protocols work without the user even realizing it, except for the leaps in speed and reduction of required bandwidth. As a result, we rely on attaching messages to the user's data, rather than taking an active approach and sending protocol-specific ourselves (or having the user send on our behalf).

Session Structure	
<b>group</b>	The recipient set
<b>ct_msg</b>	Timestep encrypted for (“canonical timestamp”)
<b>header</b>	BCE Header $\{C_0, C_1, ct\_msg\}$
<b>key</b>	Decryption key
<b>pub_id</b>	Public session id
<b>priv_id</b>	Private session id
<b>next</b>	Next session in the session chain ( $this \prec next$ )
<b>prev</b>	Previous session in the chain ( $prev \prec this$ )

Table 2.5: Structure of the *session* object.

## Incremental Deployment

The **sessions** protocol is a big win. It guarantees that a packet header only gets eliminated after all members of the group have seen it, by keeping track of the set of recipients that have also sent a message. This in itself is not impressive. However, it has the feature of *incremental deployment*: even if the session has not switched to the slim header, to generate the header, each sender has only to copy the bytes on top, from the saved session. Additionally, receivers that are already in the know have saved the BCE key. Thus, while each message remains individually unsealable, (ie. stateless) all the group members may already reap much of the benefit.

### 2.3.4 Protocol Pseudocode

This is the section that describes the completed protocol in all of its nuanced glory. We put together the cryptography techniques, error handling, and stateful optimizations to end up with the beastly procedures given below. To assist in understanding we break up the functionality into logical pieces, separating the encryption and authentication stages, as well as the *Get* and *Put* interactions with the session table, as these must handle a number of error conditions.

First, for reference, the structure of the session object is given in Table 2.5. Then we develop the algorithms, assuming the whole process is kicked off by a user wanting to send a message *plaintext* to a group *group*, available until the timestep indicated by *ct\_msg* has expired. These are complex procedures, incorporating everything covered up to this point.

The authentication is layered cleanly above the encryption. The error conditions (session agreement, private id collision) are handled by detecting the error on insertion to the hash table. An error is signalled and a message entered into an error table, to be sent. The error message is authenticated, and the error id is checked for and handled during decryption.

For interacting with the session table, we use functions that wrap the standard *Get*, *Put*, and *Remove* primitives, for the purposes of handling collisions and disambiguation as described previously. We call the wrapper functions *Safe-Get* and *Safe-Put*. We omit *Safe-Remove* because it is not important for correctness. Additionally, these functions have magical access to the session hash table — we do not bother to show mundane data passing or error handling.

---

**Algorithm 1** Seal a message.

---

SEAL-MESSAGE(*plaintext*, *group*, *ct\_msg*)

- 1 *encrypted\_msg*  $\leftarrow$  ENCRYPT-MESSAGE(*plaintext*, *group*, *ct\_msg*)
  - 2 *signature*  $\leftarrow$  SIGN(*encrypted\_msg*)
  - 3 *sender\_id*  $\leftarrow$  LOCAL-ID()
  - 4 *sealed\_msg*  $\leftarrow$  {*encrypted\_msg*, *signature*, *sender\_id*}
  - 5 **return** *sealed\_msg*
- 

---

**Algorithm 2** Unseal a message.

---

UNSEAL-MESSAGE(*ciphertext*)

- 1 *sender\_id*  $\leftarrow$  POP(*ciphertext*)
  - 2 *signature*  $\leftarrow$  POP(*ciphertext*)
  - 3 **return** DECRYPT-MESSAGE(*ciphertext*)
- 

---

**Algorithm 3** Encryption subroutine.

---

ENCRYPT-MESSAGE(*plaintext*, *group*, *ct\_msg*)

- 1 *my\_id*  $\leftarrow$  LOCAL-ID()
  - 2 *S*  $\leftarrow$  SAFE-GET(*group*, *ct\_msg*)
  - 3 **if** *S*  $\neq$  NIL **then**
  - 4     *ct*  $\leftarrow$  ENCRYPT({*plaintext*, *my\_id*}, *key*[*S*])
  - 5     **return** {*ct*, *priv\_id*[*S*]}
  - 6 **else**
  - 7     *header*, *key*  $\leftarrow$  CALCULATE-HEADER(*group*, *ct\_msg*)
  - 8     *S*  $\leftarrow$  {*group*, *ct\_msg*, *header*, *key*, RAND(), RAND() }
  - 9     SAFE-PUT(*S*  $\leftarrow$  *priv\_id*[*S*])
  - 10    SAFE-PUT(*S*  $\leftarrow$  {*group*, *ct\_msg*})
  - 11    *ciphertext*  $\leftarrow$  ENCRYPT({*plaintext*, *priv\_id*[*S*]}, *header*[*S*])
  - 12    **return** {*ciphertext*, *header*[*S*], *group*, *pub\_id*[*S*]}
-

---

**Algorithm 4** Decryption subroutine

---

DECRYPT-MESSAGE(*id\_signer*, *ciphertext*)

```
1  id ← POP(ciphertext)
2  if PUBLIC-ID?(id) then
3    header, group ← POP(ciphertext)
4    if id_signer ≠ SENDER-ID(group) then
5      return ⊥key ← CALCULATE-KEY(group, header)
6    plaintext ← DECRYPT(ciphertext, key)
7    plaintext, priv_id ← POP(plaintext)
8    S ← {group, ct_msg, header, key, pub_id, priv_id}
9    PUT(S ← priv_id)
10   PUT(S ← {group, ct_msg})
11   return plaintext
12 else if PRIVATE-ID?(id)
13   priv_id ← POP(ciphertext)
14   S ← SAFE-GET(priv_id)
15   plaintext ← DECRYPT(ciphertext, key[S])
16   return plaintext
17 else if ERROR-MESSAGE?(id)
18   asked_for ← POP(ciphertext)
19   S ← LOOKUP(asked_for)
20   REGISTER-ERROR(header[S])
21   return ⊥
22 else
23   return ⊥
```

---

---

**Algorithm 5** Safe-Get & Safe-Put: Wrapping a hash table.

---

SAFE-GET( <i>group</i> , <i>ct_msg</i> )	SAFE-GET( <i>priv_id</i> )
1 $h \leftarrow \text{HASH}(\textit{group}, \textit{ct\_msg})$	1 $S \leftarrow \text{GET}(\textit{priv\_id})$
2 $S \leftarrow \text{GET}(h)$	2 <b>if</b> $S \leftarrow \text{NIL}$ <b>then</b>
3 <b>while</b> $\textit{group}[S] \neq \textit{group}[S']$	3 <b>return</b> NIL
4 <b>do</b> $S \leftarrow \textit{next}[S]$	4 <b>else</b>
	5 <b>return</b> $S$
SAFE-PUT( $S \leftarrow \{\textit{group}, \textit{ct\_msg}\}$ )	SAFE-PUT( $S \leftarrow \textit{priv\_id}$ )
1 $h \leftarrow \text{HASH}(\textit{group}, \textit{ct\_msg})$	1 $S' \leftarrow \text{SAFE-GET}(\textit{priv\_id})$
2 $S' \leftarrow \text{LOOKUP}(h)$	2 <b>if</b> $S' = \text{NIL}$ <b>then</b>
3 <b>if</b> $S' = \text{NIL}$ <b>then</b>	3 $\text{PUT}(S \leftarrow \textit{priv\_id})$
4 $\text{PUT}(S)$	4 <b>else</b>
5 <b>else if</b> $S \prec S'$	5 $\text{REPORT-ERROR}(\textit{PRIV\_ID\_NOT\_FOUND},$
6 $\text{REMOVE}(h)$	6 $\textit{priv\_id})$
7 $\text{PUT}(S \leftarrow h)$	
8 $\textit{next}[S] \leftarrow S'$	
9 $\textit{prev}[S'] \leftarrow S$	
10 <b>else</b>	
11 <b>while</b> $S' \prec S$	
12 <b>do</b> $S' \leftarrow \textit{next}[S']$	
13 $\textit{next}[S] \leftarrow S'$	
14 $\textit{prev}[S] \leftarrow \textit{prev}[S']$	
15 $\textit{next}[\textit{prev}[S']] \leftarrow S$	
16 $\textit{prev}[S'] \leftarrow S$	



```
HANDLE hFile = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ, 0,
    OPEN_EXISTING, 0, 0);
hMemMap = CreateFileMapping(hFile, 0, PAGE_READONLY, 0, 0, 0);
length = GetFileSize(hFile, 0);
UnmapViewOfFile(filebuffer);
MapViewOfFile(hMemMap, FILE_MAP_READ, 0, baseoff, mappedlength);
```

**Code Snippet 2:** Example of memory mapping a file using Win32 API.

## 2.4 Big Public Key Optimization

### 2.4.1 Objective

In order to support a possible user base of up to  $\approx 10$ million, we need a solution to storing all of the public key information which is theoretically lower-bounded by  $\approx 20 \cdot N$ [35] for  $N$  addressable users. It must require a not-unreasonable amount of system resources to maintain and access, scaling gracefully to gigabytes of public key data. It should have the following properties:

- Fast access, read only
- Requests likely repeated
- Caching should make this point immaterial to disk representation
- Medium speed for first-time access acceptable
- Random access

Given the scenarios we are designing for, it seems most prudent to optimize for objectives in this order:

1. Repeated read access
2. Memory use
3. First-time read access
4. Size on disk

### 2.4.2 Candidates

Doing a little bit of research into the realm of representing large amounts of data on disk yielded a few observations.

Firstly, memory-mapping large files is an excellent way to go; from the standpoint of usability it is much easier to randomly access data in a file without having to rewind, seek, and read for every piece. This is wonderful because our data usage pattern is effectively random access. Additionally, it is much more efficient than any general approaches since the OS is at liberty to play any sort of performance tricks it wants. Snippet 2 shows how to accomplish this, using the Windows API.

Another candidate that emerged was *span tables*. These are effectively lists of “windows” into the file. They are named as such because each node in the list represents a span of data in the file. One archetypal span structure is shown in Snippet 3.

I mention these because they are interesting and possibly useful to others, but a more critical observation is that 32-bit Windows supports a maximum file size of 4 GB. This

```

struct span
{
    unsigned int buffer;    /* identifies the references */
    unsigned int offset;   /* start of data range this span covers */
    unsigned int length;   /* length of data */

    span *prev, *next;    /* linked-list */
};

```

Code Snippet 3: A span table node.

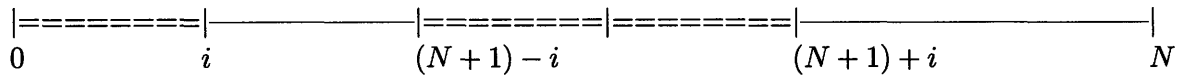


Figure 2-3: Diagram of a partially filled public key vector, for  $i$  registered users.

basically kills the idea of relying on a single file to store all public key information. So, left with the task of assembling an (incrementally growing) array, I proposed the following.

### 2.4.3 BigPK: A Scalable Block-based Public Key

We settled on a configurable block-based public key. On system creation, the number of elements per block is specified (`elements_per_block` in `default.cfg`). Thereafter, any given element can be located by generating the filename (`<block\#>.pk`), and jumping straight to an offset in that file, based on the configuration parameters.

### 2.4.4 Incremental Deployment

Additionally, one observation we made is that when adding a user to the system, only 3 more group elements are necessary for the user to encrypt or decrypt, assuming the elements are already present for all preceding users to do so. A diagram of this is given in Figure 2-3.

$$\text{Hdr} = \left( g^t, (v \cdot \prod_{j \in S} g_{n+1-j})^t \right) \in \mathbb{G}^2$$

$$K = e(g_i, C_1) / e(d_i \cdot \prod_{j \in S, j \neq i} g_{(n+1)+(i-j)}, C_0)$$

Above are equations for the encryption header and decryption key. Note that each new user widens the range of public key elements required. In particular, for all possible recipients  $i, j$ , and defining the halfway point  $A := n + 1$ , we require elements:  $i, A - i, A + (i - j)$  (reading them off from the above equations). The regions required are shown graphically in Figure 2-3.

Due to this result, none of the public key need be generated up front; instead we prefer *incremental deployment*. As users are added to the system, their 3 required elements are checked. If an element does not exist, the block it belongs to is generated and saved. This

is a big win because it allows us to provision for an unbounded number of users while only storing key data for the users that have actually signed up.

The limitation imposed by this scheme is that users' PKGE id's must count up one at a time: since the indices of the elements that go into the decryption product depend on  $i - j$ , the difference of PKGE id between the receiver and other recipients, it is essential that this difference not result in an index that references an uninitialized element (as is guaranteed by generating id's in order). Hopefully Figure 2-3 can convince you of this.

### 2.4.5 A Key in the Dark

Attentive readers will surely cast me an astutely quizzical glance at this point because they recall that the three elements required by user 1, for example, are elements 1,  $n$ , and  $n + 2$ . They will also pointedly assert that the elements are generated by raising the generator  $g$  to successive powers of  $\alpha$ . What then, they aggressively opine, do I plan to do?

Well, on system creation we create a file called `squarings.pk` — true to its name, it begins with  $\alpha$  and (have you guessed yet?) successively squares it. In particular, we only need  $\lceil \log_2 2N \rceil$  of these. For  $N = 10,000,000$ , that means 25 elements. Once we have these elements are our disposal, we can calculate any given public key element  $g_i = g^{\alpha^i}$  using a linear combination of these elements to calculate the  $\alpha^i$  (requiring at most  $\lceil \log_2 2N \rceil$  multiplications) to which we then raise  $g$ .

### 2.4.6 Rolling Admission, take 2

Again we broach the subject of rolling admission (see section 2.2.7 if you missed it the first time around!). As mentioned before, it is necessary for the application using PKGE to engage in some sort of certificate exchange, and also necessary to contact the central server for new keys (section 2.2.3) fairly rarely. Now the public key adds its requirements to the heap: the user needs to somehow acquire the new public key blocks as they are generated.

To accomplish this, we have to rely on the user returning to the central authority for new keys, at which time he can also receive any new public key blocks. It is the best we can do — blocks are potentially too large for including in a certificate exchange protocol.



## Chapter 3

# The Game is Afoot

Chapter 2 laid out the specification for exactly how PKGE would meet its goals. My *raison d'etre* for the past year has been loftily sketched, and now we get into the gory details. This chapter is devoted to details relating to our implementation of Chapter 2. All of the major code blocks will be covered in detail as they relate to the specification, and upon completion of this chapter the reader should feel at home developing new code within the PKGE framework. For simplicity, checking of error codes is omitted from displayed code, except where important for understanding, as is routine memory management.

We begin the chapter lightly (and code-free!) with the organization and format of PKGE files and directories, enabling the reader to comfortably work with the data on disk, and even to zero in on a particular piece of binary information. Continuing the easy introduction, we trace the function call sequence for common library operations to get a sense of where the important functionality lays and how the code objects interact to fulfill their roles. Afterwards, we describe the protocol implementation, look at snapshots of sealed message formats in the wild, and finally wax poetic on our forward secrecy extension and choice of cryptography primitives.

### 3.1 Development

I want to take a second to provide some context. This section describes the environment in which PKGE was developed, including the supporting tools and the libraries depended upon for large swaths of functionality.

#### 3.1.1 Tools

The development and testing environment were run on Windows XP workstations, using the Cygwin[4] environment. I use `emacs`[6] to code, `mingw gcc 3.4.4` to compile/build, and `make` to automate the process. Finally, `gdb` and `Dr Mingw` proved invaluable for debugging, `memcheckdeluxe`[15] for tracking down memory leaks, and `gprof` for identifying slow code.

Near the end of this one-year project, researcher Joe Cooley of MIT Lincoln Laboratories jumped in and led the effort to make it (PKGE and its Gaim plugin) fully compatible with the Linux and Mac OS X environments, integrating the package with GNU Autotools[21] in the process.

### 3.1.2 Testing

Due to the enormous importance of the reliability of this system, testing was a top priority. Extensive tests were written, ensuring that methods fail gracefully despite data corruption and invalid or nonsensical parameters. The test suite is described in section 4.1.

### 3.1.3 Libraries

PKGE leverages a number of libraries to accomplish its goals effectively. This section describes the dependencies and their function.

**openssl-0.9.8b** “The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library. The project is managed by a worldwide community of volunteers that use the Internet to communicate, plan, and develop the OpenSSL toolkit and its related documentation.” [26]

PKGE uses OpenSSL in particular for its X.509 handling capabilities, including such core functionality as “sign” and “verify”. Fortunately, version 0.9.8, the first version to support elliptic curve cryptography, was released on July 5, 2005, just prior to the start of this project.

**pbk-0.2.16** The Pairing-Based Cryptography library [54], written by Ben Lynn at Stanford, proved invaluable as it was the only C library we could find with support for the `bilinear_map` operation. Additionally, one of his colleagues, Matt Steiner, wrote a module implementing the BGW BCE scheme. His code was without a doubt invaluable to us, and some of the core calculations have been directly used (in `bigpk.c`).

**gmp-4.2.1** The GNU Multiprecision library [11] is the fastest available framework for developing applications using arbitrary-precision arithmetic. PBC depends on this library to implement its elliptic curves, and we use it for exponents to those group elements, the `mpz_t` type in particular.

**hashtab** We needed a hash table, and Bob Jenkins [3] obliged. He is a relatively well-known for contributions to public-domain code<sup>1</sup>. He has spent his professional career at Oracle (for the past 16 years), has a laundry list of impressive optimizations for their namesake database to his name, has quite a bit of expertise in the area of hash functions (for which he has been honored in the Wikipedia article on hash functions), and has placed it all in the public domain. We modified his basic table to integrate better with PKGE and verified that nothing fishy is going on.

**SuperFastHash** On the topic of hashing, it turns out that an engineer named Paul Hsieh had set himself the challenge to discover the fastest hash function possible (with a nice distribution of course), using Bob Jenkins’ as a starting block. He provides a very detailed account of his quest [45] including exactly what technical properties he investigated and what effect they had on performance. Overall it is the most interesting and informative

---

<sup>1</sup>Well-known enough for his own Wikipedia article. at least!

Bytes	Filename
D	./my_system
3290	./my_system/my_system.pkge
4184	./my_system/my_system.admin
226	./my_system/certs/ca.params
305	./my_system/bigpk/squarings.pk

Table 3.1: File structure for newly generated “my\_system”. The name is provided by the user, and file contents are specified in Appendix H. The “.pkge” system file lacks administrative keys required to add new users or generate a user’s keys.

narrative on hash functions I have yet read. More relevantly, he ends up with ostensibly the “best” open hash function in existence. We decided to use it, integrating with Bob Jenkins’ hash table implementation.

**MemCheckDeluxe** MemCheckDeluxe[15] (mcd.h) manages to be an effective way to track down memory leaks, and validate that a program is leak-free. Including the header provides a replacement for C’s malloc/free, showing allocations that have not been freed and alerting to double-frees on request.

## 3.2 Directory structure and File formats

As PKGE is primarily an engineering effort, care must be taken to ensure it is simple to set up. That means making it easy to refer to the system when configuring an application to use it, with the additional stipulation that it should organize and make accessible files that the user may find interesting, their key-file or certificate collection for example. The technical challenge is to create a cross-platform file-naming system that is resilient to relocation and extension. I came up with the following solution.

This section also describes the files, their format, and contents. We will walk through an example system creation, configuration, and distribution, looking at the generated files along the way.

Let’s start at the very beginning.

### 3.2.1 Generating a system

```
$ pkggen setup my_system default.cfg2
```

A reasonable default.cfg is given in Appendix J.1. It is well-commented and self-explanatory. The resulting file structure is shown in Table 3.1, for a user-generated system named, intuitively enough, “my\_system”. The binary formats of these files is included in Appendix H.

### 3.2.2 Adding users

```
$ pkggen add my_system txt users.txt
```

---

<sup>2</sup>Note: running “pkggen” displays the help, listing the options available.

```
MIT Lincoln Laboratory
r0b   Figueiredo   Robert   Student   Group 62
rkh   Khazan       Roger    PI        Group 62
```

Console Listing 1: Top 3 lines from a sample user-info text file.

Bytes	Filename
2841	./my_system/users/_r0b.usr
2841	./my_system/users/_rkh.usr
592	./my_system/certs/r0b.der
581	./my_system/certs/rkh.der
4737	./my_system/bigpk/0.pk
4737	./my_system/bigpk/26259.pk
4737	./my_system/bigpk/2625a.pk

Table 3.2: Files created when adding “r0b” and “rkh”.

An excerpt from a typical user information file is given in Listing 1. The format is very simple: the central organization of the system is meant to serve appears on the first line. On all subsequent lines, a user’s PKGE name, (often referred to as his “common name”) appears, along with his family name, given name, title, and group name. Executing the command on this info file generates the structure given in Table 3.2.2.

The `.usr` files contain the users’ private keys, up until their temporal “horizon” — the time at which they need new keys due to key consumption by forward secrecy. The `.usr` files are unencrypted. The certificates are filled in with the information provided in the text file and are written in standard DER format [46]. Now that our system has users, we have to give out the files so they can do something with them.

### 3.2.3 Generating a distribution

```
$ pkgegen dist my_system r0b 12345
```

The `dist` command encrypts the keys with a specified PIN (via a cryptographically secure salt-n-stretch[47]). The administrator would not distribute the generated files at his discretion. We envision the solution to be web-based, wherein the user authenticates himself to the central authority and is sent the package given in Table 3.2.3, but any other way would do. The user is able — nay, recommended — to treat this package/directory like a black box (well, really a black directory...). Once downloaded, he simply provides its path to any application that requests it, but should never need to look inside.

Bytes	Filename
2841	./my_system/users/r0b.usr
2841	./my_system/users/rkh.usr

Table 3.3: Encrypted user keys, ready for distribution.



Bytes	Filename
D	./my_system
3290	./my_system/my_system.pkge
2841	./my_system/users/r0b.usr
592	./my_system/certs/r0b.der
4737	./my_system/bigpk/0.pk
4737	./my_system/bigpk/26259.pk
4737	./my_system/bigpk/2625a.pk
20934	Total Size

Table 3.4: File set distribution for a new user, “r0b”. “r0b.usr” is a set of BCE keys, and one X.509 key, encrypted with r0b’s PIN. The “\*.pk” files contain public key elements. The “r0b.der” is the DER-encoded X.509 certificate for r0b.

## 3.3 Code Trace

This section is meant to ease you into the code by giving a high level view of the code path that PKGE takes when sealing a message. No implementation details are covered here other than the sequence of function calls. The functions are well-named and there is a narrative accompanying each snippet, geared to be the most straightforward possible to newcomers.

### 3.3.1 Objects and Conventions

PKGE is really a C++ design trapped in a C body<sup>3</sup>. All functionality can be related to an object that in principle carries the responsibility. The function naming convention is to prefix the function name with the object that “owns” it, and pass a handle of that object type as the first parameter. The code for said function would also reside in that objects namesake file.

One notable outcast from this object-oriented policy lies in `crypto.c`, which stores the identities of the crypto functions in static globals instead of in an object. The other resides in `pkge_store.c`, which is a static associative list of loaded systems<sup>4</sup>. To introduce you to the major objects that serve as the principle data containers and calculational tools we have drawn up Table 3.5. With that in mind, let’s start tracing some code!

### 3.3.2 Initialization

We first cover system initialization. The action during initialization is pretty well distributed with each object responsible for initializing itself. First the `pkge_store` is checked for a loaded system associated with the given path; if extant, its protocol is set to the one given and nothing more. Otherwise, the full load procedure is followed as shown below.

```
⇒ pkge_init
  ↪ _pkge_store_get
    OpenSSL_add_all_algorithms
    crypto_load
    curve_load
    certs_load
    _pkge_store_add
    set_protocol
```

### 3.3.3 Load a user

To do anything with PKGE, one must first load a user by providing the system path, the user’s textual PKGE name, and a PIN that is used to decrypt the user’s key file. First, the system handle is retrieved from the `pkge_store`, using the path (if not present, we call `_pkge_init` with that path, only failing if that also fails).

---

<sup>3</sup>Then why choose C? The biggest reason is that the libraries we wanted to use were also written in C (PBC, OpenSSL, Gaim), and although one can convince C and C++ to play nicely, we figured the tedium of passing the object as an explicit, rather than implicit, parameter was not a deal-breaker. Additionally C offers a slight increase for hardware portability.

<sup>4</sup>The `pkge_store` maps the path to the system handle, allowing a system to be identified by a simple string and freeing the user from storing a system handle.

Object	Identity and responsibility
<code>pkge_t</code>	Top level system, carries global parameters and handles to subobjects.
<code>elliptic_t</code>	Represents an elliptic curve, contains the encryption functions, manages <code>bigpk_t</code> , and implements forward secrecy.
<code>bigpk_t</code>	Manages the big public key and implements elliptic curve calculations.
<code>squarings_t</code>	Residing in <code>bigpk.c</code> , manages the <i>alpha</i> squarings.
<code>certs_t</code>	Manages all certificate operations, including verification ( <code>certs_verify()</code> ).
<code>user_t</code>	Represents a user, container for user info, manages <code>privkey_t</code> , and contains signing function.
<code>privkey_t</code>	Represents a users' set of private BCE keys (there are multiple keys as a result of forward secrecy).
<code>group_t</code>	Represents a group, including the recipient set and current sender identity.
<code>crypto.c</code>	No object, but contains all cryptographic operations (e.g. <code>user_sign</code> calls <code>crypto_sign</code> with appropriate information).
<code>config_t</code>	A simple text buffer of configuration data and associated operations to extract parameter values.
<code>data_t</code>	A block of data, containing only a <code>byte_t</code> pointer and a length. Has associated functions to compose, decompose, tag, and identify blocks of data.
<code>protocol_t</code>	Collection of functions that define a protocol.
<code>session_t</code>	Represents a session, an encryption stream addressed to a particular recipient set and timestep.

Table 3.5: Major PKGE objects and their primary purpose.

Once in possession of a system handle (`pkge_t` object), the handles “loaded user” list is checked for the requested user name. If it is found, that user object is immediately returned. If not, we resort to actually doing all the work, and `user_load` is called.

The rest is pretty straightforward, loading the users information and private keys through an encrypted stream (obtained from `crypto_fopen_r`). Additionally, the user’s certificate is verified, with load failing if it is either invalid or not present.

```

⇒⇒ pkge_user_load
   ↪ _pkge_store_get
     user_load
       ↪ user_filename(user_dir, user_name, pin)
         certs_get_x509
           X509_verify
             crypto_fopen_r(user_filename, pin)
               privkey_load
                 d2i_PrivateKey_bio

```

### 3.3.4 Add a recipient

Once the system and user are loaded, all that remains is to create useful groups, representing a chat room, for example. This is done in the simplest way possible: applications communicate identities to PKGE by a plaintext identifier, for example their chat alias or email address.

The procedure works by looking up the textual name using certificates. The certificates bind such a name to its corresponding PKGE id, which are directly equivalent to the ids of the BGW BCE scheme. Once this id is determined, it gets passed to a helper function, `group_add`, which does the real work. In particular, it first scans the recipient list, returning if the member is already present. If not, his user id gets inserted into the list, maintaining an ascending sorted order. The particular function calls are shown below.

```
⇒ pkge_group_add(g_chatroom, ‘‘Rob")
   ↪ certs_lookup(‘‘Rob")
     group_add(g_chatroom, (Rob’s numerical PKGE id))
```

### 3.3.5 Seal a message

Now let’s look at how a message is sealed. First item of note is that the `pkge_seal` is actually an alias for the slightly more complicated `pkge_seal_delayed`, which allows specification of the timestep that a message is addressed to. It passes a fixed parameter of “0” for the *days* parameter, indicating that it should be addressed to the present timestep. After that, most of the action occurs in `pkge_encrypt`, where the currently selected protocol’s functions are called, and the encryption actually happens. The parameter `g_chatroom` is a PKGE *group* object that represents the recipient list corresponding to our example chat room, and `msg` is the data to be communicated.

```
⇒ pkge_seal(g_chatroom, msg);
   ↪ pkge_seal_delayed(g_chatroom, msg, 0)
     ↪ pkge_encrypt(g_chatroom, msg, 0)
       ↪ session_get
         session_symkey
         session_push_private
         crypto_encrypt
         session_push_public
         free_session
       user_sign
```

### 3.3.6 Unseal a message

The next point of interest is the unsealing of a message. This is accomplished by passing in an initialized group and the message to be unsealed. First, the signature on the message is verified, and if successful, the message is passed to the decryption function. This uses protocol functions to unpack the message, get the group it was addressed to, and return the associated symmetric key (calculated or cached). Then the real decryption function is called (`crypto_decrypt`).

```
⇒ pkge_unseal(g_chatroom, msg)
  ↪ certs_verify
    pkge_decrypt
      ↪ session_pop_public
        session_group
        group_recipient
        session_symkey
        crypto_decrypt
        session_pop_private
        free_session
```

### 3.3.7 Shutdown

Shutdown proceeds by removing the *path*  $\mapsto$  *handle* mapping from the store, freeing the session hash table, and then notifying all of the primary objects. No surprises here.

```
⇒ pkge_shutdown(const char* path)
  _pkge_store_get
  _pkge_store_remove
  free_sessions
  certs_free
  curve_free
  _pkge_store_free
```

```

struct protocol_s {
    session_t (*session_get)      (elliptic_t, htab*, group_t, time_t);
    int       (*session_push_private)(session_t, data_t*);
    int       (*session_push_public) (session_t, data_t*);
    session_t (*session_pop_public)  (htab*, data_t*, const group_t);
    int       (*session_pop_private) (htab*, session_t, data_t*);
    data_t*   (*get_error_msg)      (htab*, const group_t, data_t*);
};

```

Code Snippet 4: Structure of a protocol object.

## 3.4 Implementation of Stateful Protocols

This section describes the implementation of the session protocols. The relevant code is located in `session.c`, `session.h`, `session_stateless.h`, `session_sessions.h`, and `session_optimistic.h`. The first two handle common functionality, and the latter three implement their namesake protocol functionality. To understand what they contain, we need to look at the structure of a protocol object.

### 3.4.1 Structure of a Protocol

We use `protocol_s` structures to embody a protocol's functionality; they are packages of function pointers (defined as a static 3-element arrays in `session.h`) that dispatch requests to the real functions (in `session*.h`), allowing `pkge.c` to do its thing in a protocol-agnostic manner. The prototype is given in Snippet 4. In addition to those functions, two more (protocol-independent) functions are required: `session_symkey` and `free_session`. Let's see what these functions do during the seal/unseal procedures.

#### Seal:

1. `session_get` returns a session object for the given { group, timestep }. The session may be pre-existing (and have a previously-calculated key), or brand new, in which case the header is generated.
2. `session_push_private` pushes any session information that is not required for the recipient to unseal the message.
3. The symmetric key is retrieved via `session_symkey`.
4. (The message gets encrypted via `crypto_encrypt`.)
5. `session_push_public` appends everything required by recipients to unseal the message. In particular, the sender id, the recipient set, the BCE header ( $C_0$  &  $C_1$ ) and the message's timestep. The sender id is required to verify the signature, the recipient set and header are required for BCE key calculation, and the timestep is required for the forward secrecy extension.
6. (The message gets signed via `user_sign`.)
7. `session_free` decreases the reference count on the session object.

#### Unseal:

```

typedef uint32_t session_id_t;

typedef struct session_s {
    data_t    hdr;
    data_t    symkey;
    user_id_t id_proposer;
    time_t    t_proposal;
    time_t    ct_msg;
    group_t   group;
    group_t   verified;
    session_id_t priv_id;
    session_id_t pub_id;
    int       force_header;
    int       outgoing;
    session_t older;
    session_t newer;
    int       refs;
    int       probe;
} *session_t;

```

Code Snippet 5: Structure of the session's C representation.

1. (The signature is verified via `certs_verify`.)
2. `session_pop_public` skims the unencrypted information off the top, returning a new or pre-existing session object containing that information. This is where an error code may be signaled for receiving an optimized message carrying an unrecognized id.
3. The symmetric key is retrieved via `session_symkey`.
4. (The message is decrypted via `crypto_decrypt`.)
5. `session_pop_private` adds secured information to the session, e.g. the private id.
6. `session_free` decreases the reference count on the session object.

The rest of this function follows the map given above. We traverse the protocol functionality paths that seal, and then unseal, a message. Regarding the location of code: in general, the functionality common to all protocols is extracted to `session.c`, with protocol-specific code in `session_<protocol name>.h`. Also, we cover the `sessions` protocol in detail because it is the most illustrative: *stateless* is far simpler and *optimistic* is nearly identical (recall that the only difference is the point at which they switch to the private-id-only header). And finally, we provide for reference the session structure, in Snippet 5.

### 3.4.2 Getting a Session

To begin, Snippet 6 shows the `session_get` for the `sessions` protocol, which employs a call to the common function `_session_get()`. To *get* a session, we require a few arguments. First, we require an elliptic curve object, in case the session does not already exist and we want to calculate the header and symmetric key anew — however it has become vestigial and is not actually used in this stage. Second, we require the session hash table, to be able

to lookup any existing sessions. The next two, the group and timestep, are necessary for creating a new session or verifying an existing session is valid.

A couple notes about Snippet 6:

- The common function `_session_get()` implements a simple lookup: return the session associated with the given key or NULL.
- The group is stored in the session and *frozen* — a session may not have its recipient set change. A caveat is that since the message sender is also stored in the group, it needs to be set to the current user for outgoing messages to avoid accidentally impersonating someone else (and generating invalid messages!).
- The session's *outgoing* flag is set to indicate an outgoing message; however, it is vestigial.
- If the lookup fails, a session is created and added to the table using `_session_put()`.

### 3.4.3 Add private Information

The next stage in the *seal* is to add protocol-specific information to the message that will end up being encrypted along with it. In particular, for an optimized header, we only put the sender id and a tag stating that it is in optimized mode. In stateless mode, we attach the session's private id, the time that the session was created, and the PKGE id of the user that created it. See Snippet 7 for the code.

### 3.4.4 Get the symmetric key

Once we collect all the information to be secured, we need the key with which to secure it. The code shown in Snippet 8 gives our approach to the symmetric key. This implements a straightforward lazy calculation of the symmetric key.

### 3.4.5 Add public information

The message has just been encrypted with the symmetric key. At this point, we have to add any information that a recipient might need to unseal the message. In particular, Snippet 9 details the procedure. The first *if* block affixes the full header onto the message: first the common function `_session_push_public()` adds the group, BCE header, and timestep, and then the *sessions* code adds the public session id and a tag indicating that the full header is present. The alternative is to affix only the private session id plus a tag, which is taken (in the *sessions* protocol) only if the session is *verified*<sup>5</sup> and there is no flag telling us that we need to send a header next time (*force\_header*).

### 3.4.6 Send the message

→ *whoosh* →

The user's communication application takes the secured message from PKGE and sends it across the transport medium.

---

<sup>5</sup>A verified means that the local user has received a message from everyone in the group with that header and private id, guaranteeing that the association is known to everyone.



```

session_t sessions_session_get(elliptic_t curve, htab* tab,
    const group_t g, time_t ct_msg) {

    // Try to retrieve the session from the table
    session_t s = _session_get(tab, g, ct_msg);

    if (s) {
        // sender may not be local user
        group_set_sender_id(s->group, user_id(group_owner(s->group)));
    } else {
        // session not found in table, generate a new one
        s = new_session(g, ct_msg);
        _session_put(tab, s);
    }
    s->outgoing = 1;
    return s;
}

// Return session from table if present, NULL otherwise
session_t _session_get(htab* tab, const group_t g, time_t ct_msg) {
    uint32_t h = _session_hash(g, ct_msg);
    if (FALSE == hfind(tab, h))
        return NULL;
    session_t s = (session_t)hstuff(tab);
    while (s && !group_equal(g, s->group)) { s = s->next; }
    if (NULL == s)
        return NULL;
    s->refs++;
    return s;
}

```

**Code Snippet 6:** Getting a session (using the group and timestep) under the *sessions* protocol.

### 3.4.7 Read the public header

Oh! We've just gotten something in our inbox! We look at the (public) header to see what to do with it. This is accomplished by the procedure shown in Snippet 10 which can interpret full headers, optimized headers, error messages, and even responses to error messages. However, due to the large number of possibilities, the function has grown quite long. Additionally, this is where disambiguation is handled, in the *while* loop in the *PKGE\_TAG\_SESSION* block.

The following list describes the functionality, so you may follow along in the code (Snippet 10).

- If a full header was received:
  1. A session has been created and returned using the information provided in the header.

```

int sessions_session_push_private(session_t s, data_t* msg) {
    // Don't need anything if we have switched to session key
    if (!s->force_header && !_session_verified(s)) {
        // Push a placeholder tag
        user_id_t sender = group_sender_id(s->group);
        data_push_block(msg, PKGE_TAG_ID_SENDER, &sender, sizeof(sender));
        data_push_tag(msg, PKGE_TAG_VERIFIED_SESSION, 0);
        return 0;
    }
    _session_id_push_private(msg, s);
    return 0;
}

int _session_id_push_private(data_t* msg, session_t s) {
    data_push_block(msg, PKGE_TAG_ID_SESSION, &s->priv_id, sizeof(s->priv_id));
    data_push_block(msg, PKGE_TAG_TIME_PROPOSAL,
        &s->t_proposal, sizeof(s->t_proposal));
    data_push_block(msg, PKGE_TAG_ID_PROPOSER,
        &s->id_proposer, sizeof(s->id_proposer));
    return 0;
}

```

**Code Snippet 7:** Affixing private protocol information.

```

const data_t* session_symkey(session_t s, elliptic_t curve) {
    if (s->symkey.data == NULL || s->symkey.len == 0)
        return _session_calc_symkey(s, curve);
    return &s->symkey;
}

```

**Code Snippet 8:** Retrieve the symmetric key from a session object.

2. A session with the same *group* and *timestep* has been found in the session table and returned.
3. The header was determined to be invalid/corrupt and *NULL* was returned.
- An optimized header was received:
  1. The session was found using the private id and returned.
  2. No session found, so an error message was inserted into the error table, and *PKGE\_ERR\_SESSION* was returned.
- A “*Unrecognized private id*” error message was received<sup>6</sup>.
  1. Perform a lookup on the requested private id<sup>7</sup>. If it succeeds, the header corresponding to that id is added to a blank message, and inserted into the error table. *PKGE\_ERR\_SESSION* is returned.
  2. The private id is not found, so ignore the message and return *PKGE\_ERR\_SESSION*.

<sup>6</sup>This is the only error message at present

<sup>7</sup>Prior to this, the requestor’s signature has been verified, so this is not a security hole.

```

int sessions_session_push_public(session_t s, data_t* msg) {
    if (s->force_header || !_session_verified(s)) {
        s->force_header = 0;
        _session_push_public(msg, s);
        data_push_block(msg, PKGE_TAG_PUBLIC_ID_SESSION,
                        &s->pub_id, sizeof(s->pub_id));
        data_push_tag(msg, PKGE_TAG_SESSION, 0);
    } else {
        _session_id_push_public(msg, s);
        data_push_tag(msg, PKGE_TAG_ID_SESSION, 0);
    }
    return 0;
}

int _session_push_public(data_t* msg, session_t s) {
    group_set_sender_id(s->group, user_id(group_owner(s->group)));
    group_push(s->group, msg);
    data_push_block(msg, PKGE_TAG_HDR, s->hdr.data, s->hdr.len);
    data_push_block(msg, PKGE_TAG_CT_MSG, &s->ct_msg, sizeof(s->ct_msg));
    return 0;
}

int _session_id_push_public(data_t* msg, session_t s) {
    data_push_block(msg, PKGE_TAG_ID_SESSION,
                    &s->priv_id, sizeof(s->priv_id));
    return 0;
}

```

Code Snippet 9: Affixing public header information.

### 3.4.8 Get the symmetric key

Identical to the procedure already covered (section 3.4.4)!

### 3.4.9 Get the private protocol information

Now that the message is decrypted, let's collect any protocol information that might aid us later on. Again we see the two-block structure, one case for the optimized message format and one for the stateless. The optimized case gets the id of the sender and returns. The stateless case gets the id of the header creator (*id\_proposer*), the time the header was created (*t\_proposal*), and the private session id (*priv\_id*). The procedure is straightforward, and shown in Snippet 13.

### 3.4.10 Session Hash

We have covered the primary functionality, but in doing so we managed to gloss over the details surrounding the hash table and hash function. Let's look at them now. To begin, we use the Hsieh hash function[45] (details in Appendix 3.1.3) to produce 32-bit keys from

various objects and store the objects using the Jenkins hash table[3] for our associative lookup.

For example, in order to look up a session by the recipient set, groups are kept with sorted recipient lists, and these recipient index arrays are simply passed directly to the hash function which returns a 32-bit unique key which can be used to find any existing sessions with that recipient set. Snippet 14 shows how the hash function is applied to a group and timestep to produce 32 bits.

The private id is already a unique 32-bit identifier, so we use that directly, instead of bothering to hash it. This can be seen in the `_session_put()` function, shown in Snippet 15, which doubly inserts a session into the table. A few notes:

- If there is an equivalent session (same session ids, group, and timestep), remove that and insert the new one (it's an updated session, so keep it).
- If an unequal session is found with the same private id, it signals a **private id collision**, as discussed in section 2.3.
- If sessions already exist that are addressed to the same group and timestep, the session object's *next* and *prev* are used to implement a linked list of sessions, sorted by the total ordering described in section 2.3.2<sup>8</sup>.

Happily, the other pieces of functionality are implemented simply. The *get* was previously shown in Snippet 6, and *remove* is too simple to bother with. Thus only a single topic remains.

### 3.4.11 Error Handling

In this section we cover the method PKGE uses to recover from non-menial errors like state or message loss. Without a communication channel to call its own, we also look at the way PKGE attempts to transport error messages and their subsequent responses.

Here is the exact procedure PKGE follows when it receives an unknown session id.

1. Push nonce onto message
2. Insert error message into hash table keyed on nonce
3. Return `PKGE_ERR_SESSION` from unseal
4. Pass error to sending function registered with `pkge_register_sender` or
5. Return error message from `pkge_get_error_msg`
6. Client sends this error message to the group.

Now from the point of view of someone receiving this error message:

1. Message received
2. Unseal detects the error condition, and inserts a full message header in the error table
3. Returns `PKGE_ERR_SESSION`, invoking the same error message channels as a real error
4. Response gets sent

---

<sup>8</sup>Briefly,  $A \prec B$  ("A precedes B") if A is newer, with the more senior *id\_proposer* preceding as a tiebreaker.

### 3.4.12 Message Format

Now that we have seen how the protocols are implemented, let's see what sort of messages they produce. We also introduce the message packing primitives and tools that we use to work effectively with a varied and complex set of data.

#### Working with Data

PKGE has effectively defined its own meta-language for working with data (native abilities in C are mildly limited). The grammar was evolutionarily-realized—there was no single design session that spawned the data-handling functions, and as a result the design is not exactly a paragon of finesse. Even so, it has proved an extremely effective mechanism. All of the relevant functionality is contained in `data.c`. In particular, here is how the “language” turned out:

The basic unit of data is a `data_t` structure, containing only a pointer and a size. From this, we can add, examine, and remove data from the end, block by block, enabling the system to work directly with first-class data rather than having to worry about serialization details. The primary functions are shown in Snippet 16 and do not require explanation, although a number of other convenience functions operating at finer levels (e.g. `data_push` for data alone, or `data_peek_tag` to peek at the topmost tag). There are also other functions to step through a message, printing out tags and sizes of blocks (`data_str`), etc.

#### Results

We previously calculated what the cost of our message-sealing strategy would be, in section 2.3. Let's see how close we can get under the different protocols.

Here is the **stateless** message format:

```
msg = [ E[PT ID_SENDER] SALT (GROUP (C0 C1 T_MSG) CT_MSG) SIG ID_SENDER ]
```

Legend		
Bytes	Tag	Description
< 16	E[ x ]	AES-256-CBC, 128-bit block size
0	PT	plain text
8	SALT	random bytes used in key derivation(2.2.5)
4n	GROUP	the set of recipients
153	(C0, C1, T_MSG)	two compressed curve elements and the msg timestamp <sup>9</sup>
4	CT_MSG	“canonical timestamp” of a msg (= <i>security epoch, timestep</i> )
63	SIG	ECDSA_with_sha1() over a 224 bit prime field <sup>10</sup>
4 · 2	ID_SENDER	the PKGE user id of the sender, used to verify the signature
236 + 4n	total bytes	

This omits the data-packing overhead – the tags and block sizes that allow for dynamic composition and decomposition of messages as well as error checking and data-driven dispatch using the functions in Snippet 16. The upshot is that each block has an extra 5 bytes of overhead, making the following contribution:

$$\begin{array}{r}
236 + 4n \quad \text{Previous Grand Total} \\
5 \cdot 9 \quad \text{Overhead for tags/sizes on the nine blocks given above} \\
\hline
281 + 4n \quad \text{bytes, Grand Total}
\end{array}$$

Besides the 9 primary blocks, the “HDR” block is actually made up of three sub-blocks (“C0 C1 T\_MSG”). So, it turns out that  $60/281 \approx 21\%$  of the total message size is made up of nothing more than tags and block sizes. On the bright side, these can be reduced by changing the implementation of the data handling functions to, for example, only use 2 bytes for a block size instead of 4, which would bring the total to 36 bytes, saving 40% of the overhead for 10 minutes of work.

At any rate, let’s have a look at a message formatted to support sessions. Recall that both the pessimistic and optimistic session protocols use the same format, merely differing in the point at which they switch to the optimized format.

### Session Message Format

```
msg = [ E[PT ID_SESSION T_PROPOSAL ID_PROPOSER]
        SALT (GROUP (C0 C1 T_MSG) CT_MSG) SIG ID_SENDER ]
```

This is what a first message looks like using either of the session protocols. This looks even worse! However, when we finally get to the optimized session state:

```
msg = [ E[PT ID_SENDER] ID_SESSION SIG ID_SENDER ]
```

That’s wonderful; the optimized message format has transformed PKGE into a lean and mean encryption & authentication machine.

Listing 2 gives the (gently massaged) output from a message analysis routine (`data.c:data_str`), as it goes through and prints out the tags and sizes of the associated block in a message. This makes a good summary of exactly what is included in each type of message and the all-inclusive overhead of each piece. Additionally, the message formats are broken down graphically in Figure `graphic-msgs`.

The messages in these examples represent a secure 100 bytes of encrypted, signed plaintext, addressed to 5 recipients. I have annotated sizes in bytes along the left column of each format, giving the true size contributed, since the size reported by the routine does not take the packing and tagging overhead into account. The graphs use exactly the numerical data given in the listing.

### 3.4.13 Optimizations

By now we’ve seen all of the issues and costs at hand. How can we drive down the space cost to end up with something that is really bandwidth efficient?

#### Group Representation

The single biggest bandwidth windfall appears to lie with efficient representations of the recipient set. As observed before, a naïve list representation requires  $O(n)$  which, to a large extent, nullifies the huge advantage we otherwise reap by using a constant-size ciphertext

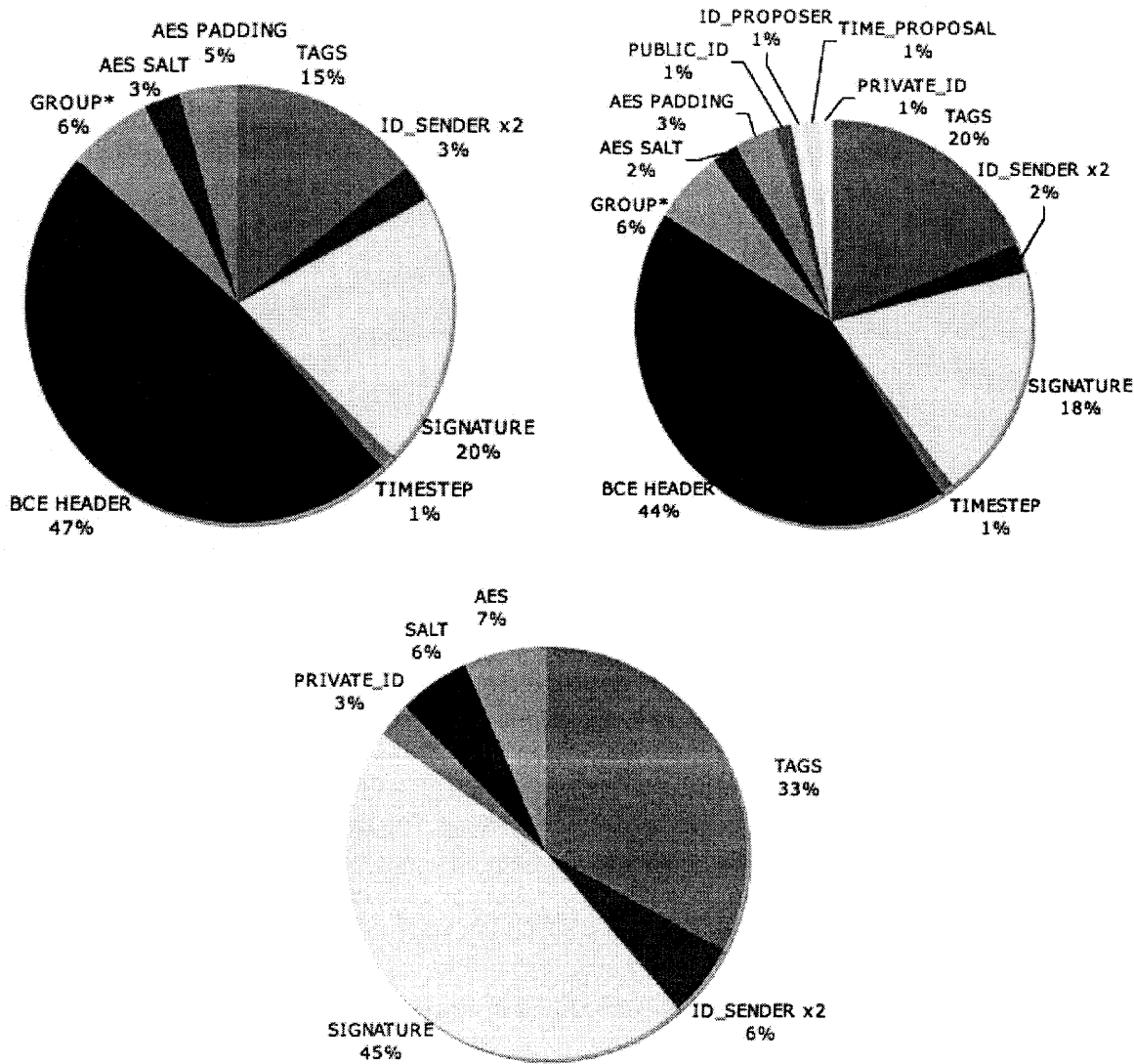


Figure 3-1: Graphical message component breakdown, for Stateless, Unoptimized, and Optimized messages respectively.

encryption scheme (although the constant in front is *much* smaller than any  $O(n)$  encryption scheme).

There are several things to do about this. One is to assign a “group id” to every organizational unit. Then, a recipient list could be represented by performing a set cover using the organizational units and individual ids. This is also good in a heuristic sense since it seems likely that a user would be predisposed to address messages to his organizational group.

Another option is to adopt some sort of convention for compressing recipient lists. For example, we could specify that “3 8 12 x 30 51” includes all ids from 12 to 30. Such a convention could be found by analyzing communication logs for the target usage environment, simulating the use of each convention and choosing the optimal.

Yet another option is to offer a no-group option in the API. That is, the client can switch PKGE to a mode where the group is not included in messages at all, and instead the

application is responsible for keeping track of it, and is required to pass in the correct group to unseal a message. This makes sense because it is very likely that the client application already knows who is in what group and which message belongs to which group. Thus, the group in the message is duplicated information that should be dispensed with.

## Data Packing

We have already seen how the overhead in data tags and sizes becomes a substantial contribution; in the optimized message format, it is the second largest contributor, behind the ECDSA signature! Theoretically speaking, once we are secure that our design has settled down, it should be possible to use pure binary format, and specify offsets for particular data within a message. Although these would act as hardcoded message formats, it could actually be read from plaintext configuration on startup.

So, for example, if a routine in the message needed the group for something, it could simply call a function that looks up the offset, based on a parameter (e.g. "GROUP") and a handle to a pointer to allocate and copy the message component to. This does not allow for treating the message as a stack and doing data-driven dispatch, but that ability has not turned out to be incredibly useful, so this way makes a good alternative that could completely eliminate the data packing overhead.

## Signature

Using a 224 bit field for signatures is undoubtedly overkill. It might make sense to use a 224 bit field to achieve stronger confidentiality, but I can't imagine a reason to require signatures to be 112 symmetric bits strong, when 80 symmetric bits is deemed acceptable for contemporary confidentiality. That being said, we could save 15 bytes on every message ever sent by switching to the 160 bit field, for example "*NIST/SECG curve over a 163 bit binary field*", which only has 50 byte signatures.



```

session_t sessions_session_pop_public(htab* tab, data_t* msg, group_t g) {
    session_t s;
    byte_t tag = data_pop_tag(msg);

    if (tag == PKGE_TAG_SESSION) {

        // Protocol specific session data; the session id hash:
        uint32_t pub_id;
        if (PKGE_TAG_PUBLIC_ID_SESSION !=
            data_pop_tagged(msg, &pub_id, sizeof(pub_id))) {
            return NULL;
        }

        // Full header included, pop it & check to see if we have already
        // seen this header
        s = _session_pop_public(msg, g);

        session_t s2 = _session_get(tab, s->group, s->ct_msg);
        session_t saved_s2 = s2;

        // Make sure the session we got really is the right one
        while (s2 && s2->pub_id != pub_id) { s2 = s2->older; }

        if (s2) {
            // Yup found it, free the popped one and return this one
            s2->outgoing = 0;
            s2->refs++;
            free_session(saved_s2);
            free_session(s);
            group_add(s2->verified, group_sender_id(g));
            return s2;
        }

        // Nope, we don't have it
        if (saved_s2) free_session(saved_s2);
        s->pub_id = pub_id;
        s->outgoing = 0;
        group_add(s->verified, group_sender_id(g));
        return s;
    }
    ... continued...
}

```

**Code Snippet 10:** Interpreting public header information.

```

...
else if (tag == PKGE_TAG_ID_SESSION) {

    // Got a session id, look it up to (hopefully) retrieve the session
    session_id_t priv_id;
    data_pop_tagged(msg, &priv_id, sizeof(priv_id));

    if ((s = _session_lookup(tab, priv_id)) {
        // Found a session matching that id, make it incoming
        s->outgoing = 0;
        return s;

    } else {
        // No session matching that id was found in table, send an error msg
        data_t* d = malloc(sizeof(data_t));
        d->data = 0; d->len = 0;
        data_push_block(d, PKGE_TAG_ERR_MSG_REQUEST, &priv_id, sizeof(priv_id));
        _set_error_msg(tab, g, msg, d);
        return NULL;
    }

} else if (tag == PKGE_TAG_ERR_MSG_REQUEST) {

    session_id_t priv_id;
    data_pop(msg, &priv_id, sizeof(priv_id));

    // Received a message indicating someone lacks the current session
    if ((s = _session_lookup(tab, priv_id)) {
        // Got the session, prepare a response to the error message
        data_t* d = malloc(sizeof(data_t));
        d->data = 0; d->len = 0;
        s->force_header = 1;
        sessions_session_push_private(s, d);
        sessions_session_push_public(s, d);
        data_push_tag(d, PKGE_TAG_ERR_MSG_RESPONSE, 0);
        _set_error_msg(tab, g, msg, d);
        free_session(s);
        return NULL;

    } else {
        // We couldn't find session either, he must be mistaken
        return NULL;
    }

    ... continued...

```

Code Snippet 11: Interpreting header information, continued.

```

...
// Got a response from our session lookup request
} else if (tag == PKGE_TAG_ERR_MSG_RESPONSE) {

    session_t s = sessions_session_pop_public(tab, msg, g);
    sessions_session_pop_private(tab, s, msg);
    free_session(s);
    return NULL;

} else { // Unrecognized tag
    return NULL;
}
}

```

Code Snippet 12: Interpreting header information, continued 2

```

int sessions_session_pop_private(htab* tab, session_t s, data_t* msg) {

    if (data_peek_tag(msg) == PKGE_TAG_VERIFIED_SESSION) {
        // Don't need anything if we have switched to session key
        data_pop_tag(msg);
        user_id_t sender;
        if (data_pop_tagged(msg, &sender, sizeof(sender)) != PKGE_TAG_ID_SENDER)
            return PKGE_ERR_CORRUPT_MSG;
        group_set_sender_id(s->group, sender);
        group_add(s->verified, sender);
        return 0;
    }

    user_id_t id_p;
    session_id_t id_s;
    time_t t_p;
    if (data_pop_tagged(msg, &id_p, sizeof(id_p)) != PKGE_TAG_ID_PROPOSER ||
        data_pop_tagged(msg, &t_p, sizeof(t_p)) != PKGE_TAG_TIME_PROPOSAL ||
        data_pop_tagged(msg, &id_s, sizeof(id_s)) != PKGE_TAG_ID_SESSION) {
        return PKGE_ERR_CORRUPT_MSG;
    }

    s->id_proposer = id_p;
    s->t_proposal = t_p;
    s->priv_id = id_s;
    _session_put(tab, s);

    return 0;
}

```

Code Snippet 13: Save the private protocol information.

```

uint32_t _session_hash(const group_t g, time_t t) {
    uint32_t g_hash = group_hash(g);
    data_t d = {0, 0};
    data_push(&d, &g_hash, sizeof(g_hash));
    data_push(&d, &t, sizeof(t));
    uint32_t s_hash = SuperFastHash((char *) d.data, d.len);
    free_data(&d);
    return s_hash;
}

```

Code Snippet 14: Hash of a session.

```

int _session_put(htab* tab, session_t s) {

    // If a session's already in the table, return that.
    session_t s2;
    if ((s2 = _session_lookup(tab, s->priv_id))) {
        if (s == s2) {
            free_session(s2);
            return 0;

        } else if (session_equal(s, s2)) {
            // Refers to the same session
            _session_remove(tab, s2);
            free_session(s2);

        } else {
            // A colliding private id:
            // a session with the same id that is not a duplicate
            free_session(s2);
        }
    }

    uint32_t h = _session_hash(s->group, s->ct_msg);

    if (TRUE == htab_find(tab, h))
        <insert into session ordering>
    else
        htab_add(tab, h, s);

    htab_add(tab, s->priv_id, s)

    group_freeze(s->group); // ensure the group can't change
    s->refs+=2;
    return 0;
}

```

Code Snippet 15: Putting a session into the session table.

```

int    data_push_block(data_t* stack, byte_t tag, void* to_write, int len);
byte_t data_pop_block (data_t* block, data_t* stack);
byte_t data_pop_tagged(data_t* stack, void* to_read, int len);

char*  data_str (const data_t* d);
data_t data_copy(data_t);

```

Code Snippet 16: Functions for working with data blocks.

Message component breakdown after sealing 100 bytes of plaintext.]{Message component br

Stateless	Sessions Unoptimized	Sessions Optimized
[Data Stack: 405	[Data Stack: 441	[Data Stack: 237
9 ID_SENDER	9 ID_SENDER	9 ID_SENDER
67 SIG	69 SIG	68 SIG
9 CT_MSG	5 SESSION tag	5 ID_SESSION tag
154 HDR	9 PUBLIC_ID_SESSION	9 ID_SESSION
25 GROUP	5 CT_MSG	13 SALT
13 SALT	154 HDR	133 CT =>
128 CT =>	25 GROUP	
	13 SALT	9 [AES overhead ]
14 [AES overhead]	149 CT =>	119 [Data Stack: 119
114 [Data Stack: 114		5 OPTIMIZED tag
9 ID_SENDER	12 [AES overhead ]	9 ID_SENDER
105 PT	132 [Data Stack: 132	105 PT
]	9 ID_PROPOSER	]
	9 TIME_PROPOSAL	
	9 ID_SESSION	
	105 PT	
	]	

Console Listing 2: [

```

// Map timestamp -> array index
int curve_fs_index(elliptic_t curve, time_t timestamp) {
    return (timestamp / curve->fs_T) % curve->fs_num_keys;
}

// Map timestamp -> a timestep's identifying timestamp
time_t curve_fs_canonical_time(elliptic_t curve, time_t time) {
    return time - time % curve->fs_T;
}

// Map index -> (canonical) timestamp
time_t curve_fs_timestamp(elliptic_t curve, time_t from, int index) {
    from = curve_fs_canonical_time(curve, from);
    int start_index = curve_fs_index(curve, from);
    int indices_forward = (index - start_index) % curve->fs_num_keys;
    if (indices_forward < 0)
        indices_forward += curve->fs_num_keys;

    time_t seconds_forward = indices_forward * curve->fs_T;
    return seconds_forward + from;
}

```

**Code Snippet 17:** Forward secrecy mapping between real-world time and timestep.

## 3.5 Forward Secrecy: Implementation

The challenge that the implementation must tackle is how to maintain a consistent *timestamp*  $\mapsto$  *array\_index* mapping in the face of asynchronously updating users and incremental generation of new public key elements.

### 3.5.1 Mapping

Most of the implementation details lie in `curve.c` and `privkey.c`. The former provides the mapping and uses the latter as a fairly dumb container for an array of keys. The code works by figuring out how many keys a user might have at one time. This derived value is stored in `elliptic_t->fs_num_keys`, and calculated as  $num\_keys = \frac{lookahead}{T}$  where *lookahead* is the length of time users may possess keys ahead of time, and *T* is the length of a timestep. All times are stored in seconds.

The fundamental implementation idea is not too complicated. The implementation of the mapping code to convert between the key's storage location and physical timestep that the key is responsible for is given in Snippet 17.

### 3.5.2 Keys

Now that the mapping business makes sense, you know how to find keys that you need at a particular time. But how do you calculate the keys, and isn't there sensitive data involved to be able to calculate any users' keys on the spot? Let's look at key generation.

```

for (int i=0; i<curve->fs_num_keys; i++) {
    mpz_init(curve->tau[i]);
    pbc_mpz_random(curve->tau[i], curve->pairing->r);
    curve->fs_timestamps[i] = curve_fs_timestamp(curve, now, i);
}

```

**Code Snippet 18:** Generating random taus on system setup. (in `curve_setup`)

Recall that keys for a particular security epoch are generated by beginning with the original BCE private key,  $d_i = g^{\gamma \alpha^i}$  to a power  $\tau$ . So  $\tau$  and  $\gamma$  form the administrative key and must be kept secret; they are only present in administrative systems. In particular, the administrative system generates an array of random  $\tau$  values, and uses them to fill in `privkey_t` structures that are given to `user_t` objects. Snippet 18 shows the simple initialization, and Snippet 19 continues to show how a `privkey_t` is created and filled in.

### 3.5.3 Expiration

Every time the library is directed to perform an operation using a set of user keys, it calls the `curve_fs_key_erase` function. Shown in Snippet 20, it quickly scans the array for any timesteps which have expired, deleting the corresponding key and setting the timestamp to `PKGE_TIMESTAMP_CLEARED`. The function is efficient and idempotent, so call early and call often.

### 3.5.4 Updates

The procedure by which a user refreshes the set of keys in his possession so that he can operate independently from the central authority is called *update*. The administrative function in particular that is meant to be called by administrative software is `_pkge_gen_update(user_t u, pin_t pin)`. This function takes a loaded user and a PIN with which to encrypt the keys, intended to be supplied by the user. The function is idempotent — it simply checks the user's keys, generating new ones as necessary, and saves the user to file, encrypted with the PIN. In order to generate the new user keys, it requires a set of new  $\tau$ 's, which are generated and saved into the system file as a side effect. The administrative software that is actually interacting with the user would then send the newly generated user file.

`_pkge_gen_update(user_t u, pin_t pin)` works by calling:

```
curve_update(user_sys(u)->curve, user_decrypt_key(u))
```

followed by save functions. `curve_update()` does the real work, and is shown in Snippet 21.

```

privkey_t curve_new_privkey(elliptic_t curve) {
    pairing_ptr pairing = curve->pairing;

    privkey_t key = malloc(sizeof(struct privkey_s));
    key->T = curve->fs_T;
    key->w = curve->fs_w;
    key->num_keys = curve->fs_num_keys;

    element_init(key->g_i, pairing->G1);
    element_init(key->g_i_gamma, pairing->G1);
    key->timestamps = malloc(key->num_keys * sizeof(time_t));
    key->g_i_gamma_tau = malloc(key->num_keys * sizeof(element_t));
    for (int i=0; i<key->num_keys; i++)
        element_init(key->g_i_gamma_tau[i], pairing->G1);
    return key;
}

int curve_gen_privkey(privkey_t* out, elliptic_t curve, user_id_t index) {
    privkey_t key = *out = curve_new_privkey(curve);
    key->id = index;
    key->T = curve->fs_T;
    key->w = curve->fs_w;
    key->num_keys = curve->fs_num_keys;

    element_clear(key->g_i);
    bigpk_add_user(curve->pk, index);
    bigpk_get(key->g_i, curve->pk, index);
    element_pow(key->g_i_gamma, key->g_i, curve->admin_key);

    time_t now = curve_fs_canonical_time(curve, time(0));
    for (int i=0; i<curve->fs_num_keys; i++) {
        key->timestamps[i] = curve_fs_timestamp(curve, now, i);
        if (key->timestamps[i] != curve->fs_timestamps[i])
            pbc_mpz_random(curve->tau[i], curve->pairing->r);
        element_pow(key->g_i_gamma_tau[i], key->g_i_gamma, curve->tau[i]);
    }
    return 0;
}

```

**Code Snippet 19:** Procedure for calculating keys in the forward secrecy scheme.



```

int curve_fs_key_erase(elliptic_t curve, privkey_t key) {
    time_t now = curve_fs_canonical_time(curve, time(0) - curve->fs_w);

    // Clear old taus for curve
    for (int i=0; i<curve->fs_num_keys; i++) {
        if (curve->fs_timestamps[i] &&
curve->fs_timestamps[i] < now) {
            mpz_set_ui(curve->tau[i], 0);
curve->fs_timestamps[i] = PKGE_TIMESTAMP_CLEARED;
        }
    }

    // If they passed in a key, cleanse
    if (key) {
        for (int i=0; i<curve->fs_num_keys; i++) {
            if (key->timestamps[i] &&
key->timestamps[i] < now) {
                element_set1(key->g_i_gamma_tau[i]);
                key->timestamps[i] = PKGE_TIMESTAMP_CLEARED;
            }
        }
    }
    return 0;
}

```

**Code Snippet 20:** Procedure for deleting expired keys.

```

int curve_update(elliptic_t curve, privkey_t key) {
    curve_fs_key_erase(curve, key);

    // Generate new taus if necessary
    time_t now = curve_fs_canonical_time(curve, time(0));
    int i_now = curve_fs_index(curve, now);

    for (int i=0; i<curve->fs_num_keys; i++) {
        int abs_i = (i + i_now) % curve->fs_num_keys;

        if (curve->fs_timestamps[abs_i] == PKGE_TIMESTAMP_CLEARED
|| curve->fs_timestamps[abs_i] < now) {
            curve->fs_timestamps[abs_i] = now + curve->fs_T * i;
            pbc_mpz_random(curve->tau[abs_i], curve->pairing->r);
        }
    }

    // Save the new bigpk elements & update private key passed in
    bigpk_update(curve->pk, curve->tau);
    privkey_update(key, curve->tau, curve->fs_timestamps);
}

int privkey_update(privkey_t key, mpz_t* tau, time_t* timestamps) {
    for (int i=0; i<key->num_keys; i++) {
        element_pow(key->g_i_gamma_tau[i], key->g_i_gamma, tau[i]);
        key->timestamps[i] = timestamps[i];
    }
    return 0;
}

```

**Code Snippet 21:** Code to generate updates to temporal keys.

```

struct bigpk_s {
    pairing_ptr pairing;
    mpz_t alpha;
    element_t g;
    element_t g_gamma;
    element_t* g_gamma_tau;
    unsigned num_taus;

    unsigned num_users;
    unsigned elements_per_block;

    char* dir;
    char* subdir;

    squarings_t* sqrs;
};

typedef struct {
    user_id_t* indices;
    mpz_t* alphas;
    unsigned n;
} squarings_t;

```

**Code Snippet 22:** The support structure for a big public key and a structure to hold successive squarings of  $\alpha$ .

```

// Figure out which block this element is in
int block_num(bigpk_t pk, user_id_t index) {
    return (index-1) / pk->elements_per_block;
}

// Reverse mapping , block # => first index it contains
int first_index_in_block(bigpk_t pk, unsigned nblock) {
    return nblock * pk->elements_per_block + 1;
}

```

**Code Snippet 23:** Implementation of (*block number*)  $\leftrightarrow$  (*public key index*) mapping

## 3.6 BigPK: Implementation

This section describes the mechanism that implements the block-based public key. For reference, the structure of the `bigpk_t` and `squarings_t` objects are given in Snippet 22. Recall that the fundamental mechanism behind managing an effective, large public key was using a squarings ladder to quickly calculate any given element in the long chain, after which a simple exponentiation could generate the next elements. Random-access to the public key is accomplished by calculating the block # and offset, generating the filename for the block, and reading it in. The implementation for this mapping is shown in Snippet 23.

### 3.6.1 Setup

On system setup, the public key runs through the procedure shown in Snippet 24

```

int squarings_setup(bigpk_t pk) {
    squarings_t* s = pk->sqrns = malloc(sizeof(squarings_t));
    s->n = (unsigned)logb(pk->num_users) + 2;
    s->indices = malloc(s->n * sizeof(user_id_t));
    s->alphas = malloc(s->n * sizeof(mpz_t));

    //Make the 1st element equal to g^alpha
    mpz_init(s->alphas[0]);
    s->indices[0] = 1;

    for(unsigned i = 1; i<s->n; i++) {
        // Square the current element
        mpz_init(s->alphas[i]);
        mpz_pow_ui(s->alphas[i], s->alphas[i-1], 2);
        s->indices[i] = 1<<i;
    }
    return 0;
}

```

Code Snippet 24: Setup procedure for the public key.

### 3.6.2 Calculating with *Squarings*

Once we have an array of successively squared  $\alpha$ 's, how do we calculate an arbitrary public key element  $g_i = g^{\alpha^i}$ ? We use the obvious approach to forming a linear combination: add constituent elements in descending order, decrementing the target for each addition. In other words, if we want to calculate  $\alpha^7$  given  $\alpha^i$   $i \in 1..4$ , check if  $2^4 \leq 7$ : nope,  $2^3 \leq 7$ : nope,  $2^2 \leq 7$ : yes! Decrement 7 by  $2^2$  to indicate it's inclusion in the product, and continue:  $2^1 \leq 3$  and  $2^0 \leq 1$ , leaving us with  $\alpha^7 = \alpha^4 \cdot \alpha^2 \cdot \alpha = \text{sqrns}[2] * \text{sqrns}[1] * \text{sqrns}[0]$ , in terms of array indices. The only difference in the code is that it first checks to see if the previous element is present, in which case it avoids this work altogether. See for yourself in Snippet 25.

### 3.6.3 Adding Users

Now that we know how to calculate with the array of alpha squarings, we can painlessly see how users are added to the system, from the public key's point of view. The exact procedure is shown in Snippet 26; it simply calculates the three marginal elements that are required for a new user in the system, as shown graphically back in Figure 2-3, checks if they are present, and calculates their block if not.

### 3.6.4 Usage

Until now we have totally ignored the primary purpose of the public key, namely to provide elements of the public key to the system! That's because it is anti-climatically simple. The procedure is shown in Snippet 27. It calculates the block # in which the element must reside  $((index - 1)/elements\_per\_block)$ , since public key elements are 1-based), and uses the remainder as offset into the file.

```

int bigpk_calc_first_block_element(element_t elem,
    bigpk_t pk, int nblock) {
    user_id_t first_elem = first_index_in_block(pk, nblock);

    // If block 'nblock - 1' exists, use that to calculate
    // Else...
    squarings_t* s = pk->sqrs;

    element_t temp_elem;
    element_init(elem, pk->pairing->G1);
    element_init(temp_elem, pk->pairing->G1);
    element_set1(elem);
    int diff = first_elem;
    for (int i = s->n-1; i >= 0 && diff > 0; i--) {
        if (diff >= s->indices[i]) {
            diff -= s->indices[i];
            element_pow(temp_elem, pk->g, s->alphas[i]);
            element_mul(elem, elem, temp_elem);
        }
    }
    element_clear(temp_elem);

    return 0;
}

```

**Code Snippet 25:** Calculating the first element of a new public key block.

This particular implementation generously leaves room for optimization: this paragraph explains the  $(len + 9)$  in the mapping calculation. The elements' size and tag (indicating *compressed* or *uncompressed*) are stored with each element, which is a huge redundancy that may easily be removed in a future version. Even worse, the elements' size is stored twice, once for the function to read it in (`elem_in_bio`) and another for the data tagging system (`data.c`). Our justification for not having fixed this already is that the public key resides in cheap storage and does not contribute to message size.

### 3.7 Gaim-PKGE

This section describes the implementation of a plugin for the open-source chat-client jurgernaut, Gaim[10]. Gaim allows us to implement a single protocol-agnostic plugin and have it work over any of the major (and even the not-so-major) chat networks. Since one of the primary goals of the plugin was ease of integration, this was a valuable experience that allowed us to evaluate and respond to interface usability, as well as producing a valuable piece of software demonstrating the technology described in this thesis.

```

int bigpk_add_user(bigpk_t pk, user_id_t id) {
    // user 'id' requires these element indices:
    user_id_t ids[] = { id, pk->num_users+1-id, pk->num_users+1+id };
    for (unsigned i=0; i<3; i++) {
        // See if this element has to be calculated
        char* fn = index_filename(pk, ids[i]);
        if (fn && !check_file(fn, R_OK))
            continue;          // Nope, file exists

        // Calculate the containing block
        bigpk_calc_block(pk, block_num(pk, ids[i]));
    }
    return 0;
}

```

**Code Snippet 26:** Adding a user to the big public key.

```

int bigpk_get(element_t e, bigpk_t pk, user_id_t index) {
    char* fn = index_filename(pk, index);
    BIO* b = crypto_fopen_r(fn, 0);
    element_init(e, pk->pairing->G1);
    int len = element_length_in_bytes_compressed(e);
    int pos = BIO_tell(b);
    BIO_seek(b, pos + ((index-1) % pk->elements_per_block) * (len + 9));
    elem_in_bio(b, e);
    crypto_fclose(b);
    free(fn);
    return len;
}

```

**Code Snippet 27:** Element retrieval procedure for the big public key.

### 3.7.1 Overview

Gaim provides a powerful hooking system that allows our plugin to register callbacks that get called for any particular event you can imagine. The approach our plugin takes to securing communication is message-oriented: we register a callback for the “sending-im-msg” and “sending-chat-msg” signals (different signals for group chat and 1-to-1 instant message). The callback gets passed the sender and recipient screen names, and the (`char*`) message buffer that is in the process of being sent. The plugin has the opportunity to mangle the message to its heart’s content.

These callbacks obtain the necessary PKGE objects from a function call that manages a static instance variable. Since a loaded PKGE system keeps track of all loaded users, there is no reason not to let PKGE take care of managing user objects for you: every time a message is sent, our callback is passed the sender and receiver screen name, which we translate into the PKGE name by chopping off the “@jabber-server.com” that might be present, and calling the `pkge_user_load` function to get the user object that has the ability to seal the message.

```
void *conv_handle = gaim_conversations_get_handle();

gaim_signal_connect(conv_handle, "sending-im-msg",
    plugin, GAIM_CALLBACK(sending_im_msg_cb), 0);
```

**Code Snippet 28:** Registering a callback with Gaim.

Once a message is sealed, it gets encoded in base-64 and delimited (by `GPKGE_TAG_CT`<sup>11</sup>). Upon receiving a message, these constants are scanned for, and the string extracted. Due to asymmetries in Gaim’s message protocol, it turns out that the message seen by the receiver’s GPKGE instance is not exactly what was sent by the sender’s instance — there is extra metadata of unknown origin. The sealed message is extracted from within the tokens, unsealed, and the buffer is reconstructed to preserve the unknown data.

The plugin also takes care of certificate transfer. When a conversation is created (and a corresponding signal sent to our plugin), we send out the local user’s certificate. Upon receiving a certificate, the plugin notifies the user and indicates to whom it belongs, as well as verifying that it is authentic.

The plugin additionally provides niceties like replacing undecryptable text with a friendly notification (`GPKGE_UI_CT_REPLACEMENT`), allowing messages to be plaintext when explicitly requested<sup>12</sup>, and tagging plain text messages as insecure (`GPKGE_TAG_INSECURE`).

Given all the capabilities of the plugin, let’s examine each in a little more detail to see exactly how it accomplishes everything.

### 3.7.2 Setup

As mentioned, GPKGE registers callbacks with Gaim. The exact process is shown in Snippet 28.

When a conversation is created — “conversation” is Gaim terminology for either 1-on-1 IM or group chat — the `conversation_created_cb` function is called. Regardless of the type, it retrieves the user’s certificate from PKGE and encodes it in base 64. If any part fails, a friendly message is displayed indicating the cause of failure to the user.

If all goes well, the certificate is tagged (delimited with tokens in `tag_certificate`) and sent. Gaim unfortunately does not make programmatically-generated messages simple to send, so the sending takes a slightly uglier form shown in Snippet 29.

### 3.7.3 PINs

Any time a user tries to send a message, his user file will be loaded. The first time requires entry of a PIN to decrypt the user file from disk; subsequent loads return the previously-loaded user object without going to disk, and thus the user only has to enter a PIN once. This mechanism is accomplished by checking the return code of `pkge_user_load` and popping up a modal dialog demanding authentication, should `PKGE_ERR_PIN_REQD` be returned. This functionality is given in the `get_pin` function. Should the PKGE library begin implementing periodic invalidation of a user object to require reauthentication, this mechanism will not require any modification.

<sup>11</sup>All delimiting tags are specified by their base name but implicitly have `_PREFIX` and `_SUFFIX` suffixes for the starting and tokens

<sup>12</sup>The user must type “`INSECURE(msg)`” for `msg` to be sent in plain text.

```

GaimConnection* conn = gaim_conversation_get_gc(conv);

if (gct == GAIM_CONV_CHAT) {
    GAIM_PLUGIN_PROTOCOL_INFO(conn->prpl)->chat_send
        (conn,
         gaim_conv_chat_get_id(GAIM_CONV_CHAT(conv)),
         (char *) cert.data);
} else if (gct == GAIM_CONV_IM) {
    char* buddy;
    asprintf(&buddy, "%.*s", strstr(conv->name, "/")-conv->name, conv->name);
    GAIM_PLUGIN_PROTOCOL_INFO(conn->prpl)->send_im
        (conn, buddy, (char *) cert.data,
         GAIM_MESSAGE_SYSTEM | GAIM_MESSAGE_NO_LOG);
}

```

**Code Snippet 29:** Programmatically sending messages in Gaim.

---

**Algorithm 6** Gaim-PKGE sends a message.

---

```

SEND-MESSAGE(conversation, msg)
1  if TAGGED-INSECURE?(msg) then
2    return msg
3    group ← GET-GROUP(conversation)
4    SEAL(group, msg)
5    return msg

```

---

### 3.7.4 Seal and Unseal

The plugin demonstrates the suggested protocol for communications: it exchanges certificates on conversation startup, checks for error codes from `pkge_unseal`. The specifications for the *send* and *receive* procedures are given as Algorithms 6 and 7. The actual code looks very similar due to pleasant variable/function names, so we will not bother to show snippets.

### 3.7.5 Preferences

The plugin uses the GTK+ toolkit for displaying a *Settings* panel in the Gaim *Preferences* menu. The panel allows a user to set the PKGE system path, as well as the protocol to be used. If the plugin is compiled with the `DEBUG` symbol defined, there is also a checkbox to enable debugging output by popup windows. All preferences are stored in the Gaim preference registry, so the settings are persistent.

### 3.7.6 Certificate Management

This effort also raised our awareness of the Mozilla Foundation's Network Security Services project[17]. This is a well-developed framework for the management and sharing of cryptographic material in general. This has applications to our use of certificates, since the way



---

**Algorithm 7** Gaim-PKGE receives a message.

---

```
RECEIVE-MESSAGE(conversation, msg)
1  if IS-CERTIFICATE?(msg) then
2    IMPORT-CERTIFICATE(msg)
3    return  $\perp$ 
4  else if UNTAGGED?(msg) or IS-TAGGED-INSECURE?(msg)
5    TAG-INSECURE-MSG(msg)
6    return msg
7  else
8    ciphertext  $\leftarrow$  GET-CIPHERTEXT(msg)
9    group  $\leftarrow$  GET-GROUP(conversation)
10   plaintext, error  $\leftarrow$  UNSEAL(group, ciphertext)
11   if error  $\neq$  NIL then
12     HANDLE-ERROR(conversation, msg, error)
13     return  $\perp$ 
14   else
15     RECONSTRUCT-MESSAGE(msg, plaintext)
16   return msg
```

---

PKGE handles certificates is remarkably possessive: it had better find a certificate named `<user name>.der` in its certificate subdirectory or else! A centralized database that allows lookup by fields and nicely formatted output would be great for extensibility, and so in the short time we had I demonstrated a minor integration with NSS that will hopefully be expanded upon in the future. Namely, when users exchange certificates, NSS is used to display information about it, since GAIM is not linked with openssl and it would be inappropriate to add such functionality to the PKGE API.



## Chapter 4

# What Have We Done?

In the previous three chapters we have introduced you to a need for securing group communication that exists in the world today, described a design that correctly and efficiently addresses that need, and then expanded on the details of the work we have done to realize that design. In the final chapter, we must take a step back, look at the Public Key Group Encryption system as a whole, and evaluate it in a number of meaningful ways.

The first (and most important!) part of evaluation is correctness. Does our design fulfill the specifications we have laid out? What steps have we taken, and provided for others, to verify that this is the case? Let us pause to observe an exceptionally pertinent aphorism.

**Untested code is broken code.**

Once correctness is assured, we must evaluate **performance**. In particular, does this library have compatible statistics in the relevant metrics to be applied to the scenarios we had in mind way back when we began this journey? The answer is not yes or no; an effective answer to this question is the characteristic resource usage of PKGE. This allows anyone to walk up and, without necessarily understanding our library at all, make a quick judgment on whether or not our library can be of value to his application.

Part of performance evaluation is identifying bottlenecks; an explicit roadmap for how a future developer should go about customizing it to fit his application. During development there are literally  $N$  points in time when you have to make a tradeoff: do I want to use more *foo*'s or more *bar*'s here? Some decisions have major consequences, and these should be identified.

With all that in mind, let's dive right in to the QA dept.

### 4.1 Testing

For a library running on possibly life-critical systems, testing becomes an extremely important facet of development. In this section I describe the paces that PKGE is put through. Additionally, this section serves as a good review of the expected operation of the system.

#### 4.1.1 It's Alive!

The first part of the test suite is very straightforward. When using PKGE as the instructions say, in every possible situation, does it have the correct input-output characteristics? This is a basic sanity test that exercises every API function under a wide range of (valid) user

input. We test system generation, generating key updates, loading users (with valid and invalid PINs), sealing long sequences of random bytes, unsealing as both a recipient and non-recipient, certificate import and export, etc.

These tests may be found in `testpkge.c` and provide a good guide for how the API should be used. It is all fairly standard, though, and although it ensures that it will perform reasonably under normal use, it says nothing about robustness or if it is meeting the performance goals. Let's tackle robustness first.

#### 4.1.2 Nonsense

One technique, known as *fuzz testing*[55], or sometimes just *fuzzing*, is to subject the scrutinized entity to a lot of completely random input. It is an interesting way of testing because it assumes nothing about the system. It does not even request anything, not even correct answers. To pass a round of fuzzing, the API merely has to avoid infinite loops and crashes. This is easier said than done, especially as a C API. In fact, as far as the author can tell, it is impossible. There is simply no way to protect against being passed random pointers that are actually invalid.

So, the class of misuse that the tests ensure we guard against are null pointers for any argument, strings with invalid characters being passed as a user name or path, trying to add random bytes as a certificate, etc. In all cases, we ensure that the call did not cause a crash, and that an appropriate error was returned.

#### 4.1.3 Session Inspection

Now we have tests for correctness and robustness. The next step is tests for performance, with more of an eye towards performance correctness — does the optimistic protocol actually drop the overhead size drastically after the first message? Does the session protocol still send big messages but spend no time computing keys? If two users send messages to the same group, proposing different private ids, are the sessions ordered correctly in the table so that subsequent messages use  $A$  such that  $A \prec B$ ?

To answer these burning questions, we reread section 2.3 very carefully and wrote down all of the expectations we had that were implicit in the protocols, data structure organization, and performance promises. Then we wrote a program to act out conversations and supporting tools that allow us to reach inside the library to make sure everything is going according to plan. Essentially, this fulfilled the function of unit tests — as a cryptography system, parts are coupled relatively tightly and the effort it would take of setting up context to write unit tests for each of the pieces was prohibitive, so these tests are the replacement.

That being said, it is quite a hodgepodge, operating at multiple levels of abstraction. It calls user level functions to initialize and manage users and groups, and then replaces a pointer to the session table inside one of the objects with one that we control, and then runs.

We tested three distinct parts: the performance, the error message channel, and the session handling mechanisms. They are described in detail below, and all of the relevant code may be found in `testprotocols.c`.

#### Proper Performance Profiles

To ensure that protocols were actually performing the appropriate optimizations as they should, we came up with some useful test primitives:

```

overhead_t measure      (group_t g1, group_t g2);
void      assert_overhead(int, int, int, overhead_t measured);

```

**Code Snippet 30:** Functions to measure and test cpu/space overhead.

r0b		rkh	Stateless	Sessions	Optimistic
⚡	→	⚡	⚡ Ⓢ	⚡ Ⓢ	⚡ Ⓢ
⚡	→	⚡	⚡ Ⓢ	⚡ ⚡	⚡ ⚡
⚡	←	⚡	⚡ Ⓢ	⚡ ⚡	⚡ ⚡
⚡	→	⚡	⚡ Ⓢ	⚡ ⚡	⚡ ⚡

Table 4.1: Scenario to ensure observed resource usage meets expectations. The arrows show a message traversal, and the protocol columns show our expectations for the resource costs. The anchor (⚡) signifies a full header (ie. heavy) message, with a leaf (⚡) signifying an optimized (ie. light) message. The stop sign (Ⓢ) signifies that full computation is required, with lightning (⚡) for none.

Perhaps unsurprisingly, `measure()` simulates a message transmission:  $g_1 \rightarrow g_2$ . It records the amount of overhead on the message and the number of milliseconds the seal and unseal (separately) require, packaged into an `overhead_t` structure. `assert_overhead()` takes 3 booleans representing our expectations for the resource usage of a particular message: 1 for full size/full computation, 0 for no header/cached key. The determination on whether the performance is “good enough” to believe that the **optimized** protocol is doing its job is done in a portable way by clearing all state and then (simulating) sending a single message, saving that time and overhead as a reference and calculating the factor by which subsequent tested transactions differs.

This scheme makes it really easy to write scenarios to your heart’s content:

```
assert_overhead(1, 0, 0, measure(g_r0b, g_rkh))
```

for example, tests that a message  $r0b \rightarrow rkh$  requires full header space but both parties should have the key cached — the second message  $r0b \rightarrow rkh$  in the `sessions` protocol would have this profile. The scenario ran through is shown in Table 4.1.3.

### Bail out, bail out!

We implemented another important class of tests: the error channels. In particular, recall that there are two ways for a protocol to emit messages (aside from piggybacking, of course). It may cause `pkge_unseal` to return an error code, such as `PKGE_ERR_SESSION`, in which case we specify that the client is to call `pkge_get_error_msg()` and transmit the resulting data, or the client may register a sender callback with `pkge_register_sender()`, which the protocol calls with its message.

These are not anticipated to experience much use, but it is imperative that they work properly, of course. Testing the error message channels involved grabbing a handle to the session table used by our actors, and flushing it, simulating a loss of state from a crash or state that never arrived due to a faulty connection.

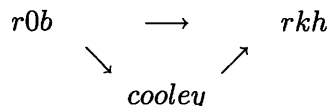


Figure 4-1: Initial message exchange before testing error channels.

To begin, our actors exchange messages as shown in Figure 4-1. At this point, *rkh* has a verified session and prefixes his messages only with private session ids. So we clear *r0b*'s state, and have him try to unseal a message from *rkh*. We ensure that `pkge_unseal()` returns `PKGE_ERR_SESSION` and that the sending callback we registered was called with an appropriate “I-don’t-recognize-this-id” message. For good measure we also grab the error message from `pkge_get_error_message()` and make sure they are the same.

The first time around we send the error message to *rkh*, repeating the process with *cooley* as the “answerer” to make sure that both senders and recipients in both optimized and unoptimized modes all provide the answer correctly. We test that the answerer’s error channel (since the answer to the error comes through their error channel) works properly in a similar manner to *r0b*'s. Finally, *r0b* unseals the answer to his cry for help, returning success, and then unseals the message he was previously unable to read, ensuring that the recommended strategy of caching messages until you can decrypt them is a valid one.

### There can be only one

The last class of tests is specifically designed to verify the complex session table protocol. In particular the function that inserts sessions into the table (`session.c:_session_put()`) is responsible for enforcing the total ordering on sessions that we established previously (section 2.3.2), as well as finding and signalling collisions that occur when inserting sessions by `hash(group, ct_msg)` or `private_id`. Managing reference counting and double-insertion hash tables and linked-list equivalence classes gets complicated; needless to say, this program was invaluable for my piece of mind.

To verify this functionality, we call the protocol functions (e.g. `push_session_public()`) directly and examine the session objects and hash table afterward. For example, in the testing function `testprotocols.c:test_session_handling()`, we have our actors send messages simultaneously to each other and assert that `(A->older == B) && (B->newer == A)`. Then they send a second message, and we ensure that the second message from both actors uses the winning session.

In order to make testing the session handling simple, we came up with two more helpers that may prove of interest. These are given in Snippet 31. These simulate A. the seal and unseal of a single message, providing the resulting sessions as output, and B. a simultaneous seal and unseal, providing the 4 sessions as output (sending and receiving sessions for both parties should be different).

## 4.2 Evaluation

Having written programs to convince us of the correctness of our solution and its adherence to the specifications, we now want to develop a performance profile. We want to describe its performance quantitatively so a casual observer is able to evaluate just how onerous

```

void do_send          (session_t* s_send_out, session_t* s_rcv_out,
                      group_t g_send, group_t g_rcv);
void do_parallel_send_rcv(session_t* a_send_out, session_t* a_rcv_out,
                          session_t* b_send_out, session_t* b_rcv_out,
                          group_t g_a, group_t g_b);

```

**Code Snippet 31:** Helper functions for testing session handling.

integration would be (in terms of resource usage). This section describes the methods we used and the results that we gleaned.

### 4.2.1 Methods

Evaluations of performance/operating characteristics are important to make an informed decision on if integration with an application or process is feasible and that the benefits will outweigh the costs in resource usage. To these ends, I pursued three wholly separate classes of evaluation. For lack of standardized terminology, let's call them *unit*, *scenario*, and *integration* evaluation. One note is that the message is taken as a whole throughout this section — for detail on which components in the message contribute to its size, see section 3.4.12.

**Unit** Similar to unit testing, this refers to having a program run each function provided by the PKGE API, measuring the computation time and, for message-providing functions, the size of the output.

**Scenario** We describe a framework for generating and playing scripts that have the ability to simulate a wide range of scenarios. They allow specification of the communication patterns you find interesting, such as message size, drop rate, frequency of recipient add/remove operations, etc. The simulation outputs per-message measurements of message size and computational load.

**Integration** The integration looked more at the relative performance of the toolkit when integrated in a real production setting: the Gaim environment. As proper part and parcel, we compare the overhead of PKGE to that of other commonly-used solutions that provide equivalent security. Naturally, we win in a big way.

### 4.2.2 Responsibility

Before we jump in, it bears asking what we should do with the results when we get them. In particular, where does the responsibility lie for the computation times and message sizes? Is the encryption procedure slow because it has to do eleventy billion exponentiations? Is the AES encryption procedure even worth considering as a time sink? Maybe keeping groups sorted by using the Bogosort[18]-&-check algorithm is causing a bigger hit than I expected. At any rate, here are some possible resource hog culprits for computation time.

- BCE encryption product
- Pairings

- SHA
- ECDSA Signature / Verification
- AES

The computation questions were evaluated using the GNU Profiler (`gprof`). It discovered, unsurprisingly, that the computation time was far and away dominated by the elliptic curve operations, by orders of magnitude.

This means that the great majority of time can be made up by adroit avoidance of elliptic curve operations. This is why stateful protocols are incredibly effective. Caching products would also be an extremely good investment, that unfortunately only makes an appearance in the “Technical Todo” in Appendix I.

Turning to message size, let’s list the usual suspects:

- BCE Header (2 elliptic curve points and a timestamp)
- Signature
- Recipient set

The answer here is a little less obvious (unless you’ve actually read the preceding chapters!). This requires no special tools to determine, since there are a relatively small number of ways a message can be packed, which we explicitly enumerated and analyzed. Looking at any of the message breakdowns in 2 leads to the conclusion that the header and signature, weighing in at a constant 150 bytes and 60 bytes overhead each, are the (relative) juggernauts, with some more small change here and there. That 210 bytes of overhead is incredibly svelte when you consider that overhead for any typical application absolutely dwarfs that. For example, a simple email containing the word “test” sent from `rkh` (`at`) `ll.mit.edu` to `r0b` (`at`) `mit.edu` required almost exactly *1 kilobyte* of headers(!).

To be fair, email is a red herring since it naturally requires more overhead to send stateless messages rather than a single chat packet, for example. To compare, sending a message containing only “test” using an unencrypted Jabber connection triggered the sending of a  $\approx 400$  byte packet. So our overhead turns out to be very reasonable, a good discovery.

However, one silent-but-deadly contribution that we have left out thus far is the *group*. In particular, our implementation simply stores the recipient set as a list of their PKGE ids, requiring exactly  $4 \cdot n$  space. There are a number of proposals for doing a lot better (see section I), and regardless, the group is one of the things that gets optimized away when using either of the stateful protocols. It should be noted that existing solutions for encrypting to groups of recipients (actual applications, not just papers), are linear in # of recipients, with a *much* bigger constant.

### 4.2.3 Unit Profiles

The goal here is to use knowledge of the internal workings of the system to dissect it and put the pieces in test harnesses to see how they perform. We calculate the (very rough) expected results of these performance tests and compare them to the actual results.

First, I created a program, `profpkge.c` to evaluate the following actions.

- First message to a group of participants
- Subsequent messages to the group
- Changes in membership: joins and leaves



```

Profiling PKGE library...

Stateless
-----
           1st      2nd      many      many fk left      previous
time, send:  88      90      88      88      84      90
time, rcv:   79      79      78      78      75      78
size of:     302     302     303     303     298     302
-----

Completed profiling protocol 0

Sessions
-----
           1st      2nd      many      many fk left      previous
time, send:  88      16      16      16      86      16
time, rcv:   67      19      19      19      61      19
size of:     348     349     136     136     345     137
-----

Completed profiling protocol 1

Optimistic
-----
           1st      2nd      many      many fk left      previous
time, send:  88      14      16      16      86      18
time, rcv:   67      19      19      19      61      19
size of:     348     136     137     136     344     136
-----

Completed profiling protocol 2

```

**Console Listing 3:** Output of profpkge on a Powerbook G4, 1 Ghz PowerPC, 1 GB SDRAM, OS X 10.4.6

- Key roll-over (day change)

The results are shown in Figure 3. This output is extremely interesting as it verifies the relative performance that we were expecting and puts concrete numbers on everything.

#### 4.2.4 Framework

In order to evaluate the viability of particular deployment scenarios, we whipped up a framework consisting of a combination of scripts that provides simulation of any particular workload. It outputs per-message measurements of computation time and message size in tab-delimited form, so that it may be easily analyzed by existing tools. Here is a step-by-step example.

1. Edit `eval_sys.cfg` by hand for desired system parameters.
2. `$ pkgegen setup eval_sys eval_sys.cfg`  
Setup the system
3. `$ ruby genusers.rb 100 > 100users`  
Automatically generate users information
4. `$ pkgegen add eval_sys txt 100users`  
Add them to the system.
5. `$ ruby genscript.rb 100 30 550 0 0.1 0.1`  
Generate a “chat script” for the users the users to act out.
6. `$ ./runscript script-100-30-550-0-0.1-0.1 eval_sys 100`  
Execute the performance, while noting the overhead

Let’s look more closely at the individual steps, so that the interested reader will be able to understand and extend when necessary. The scenario evaluation is intentionally done as a mish-mash of lightweight scripts, since supporting all possible interesting options would be futile and much more effort than simply modifying the scripts to match changing requirements.

**pkgegen setup ...** The first step is to generate a system you wish to use for testing. You will probably want to change the default parameters for forward secrecy — your automata do not need to remain separate from the central authority for very long and it is time-consuming to calculate a large number of unnecessary keys.

**genusers; pkgegen add ...** The next step is to generate users and add them to the system. The `genusers.rb` script was originally done for convenience, since I am a slow typist and have not figured out copy-and-paste yet. However, an important note is that `runscript.c` relies on the system users being called “User#”, so deviate from this step carefully. Once you have generated a number of users once, I cannot think of a reason to ever do so again, so you will likely not have need for `genusers` terribly often.

**genscript** The most interesting part of the medley, this step generates a script that the automated users follow. It supports message payloads of plaintext up to 5 MB, joins, leaves, and dropped messages. The tool options are shown below. Note:  $P(x)$  signifies “probability of  $x$ ” and “round robin?” refers to a boolean option between round robin chatting and random selection.

`genscript.rb` options:

(# users) (# lines) (line length) (round robin?) ( $P(\text{join/leave})$ ) ( $P(\text{message loss})$ )

Example: `$ ruby genscript.rb 5 1000 50 1 0.01 0.1`

Translation: generate a communication script for 5 users, 1000 lines long, per-message length of 50 characters per line using a round-robin communication pattern and a 1% chance (per line) of a recipient joining or leaving the group, and a 10% chance per line of that message getting lost, and store all that in `script-5-1000-50-1-0.01-0.1`.

The generated script is hand-editable to make it easy to insert contrived situations. A snippet of a (hand-generated) script is shown in Figure 4-2 and is hopefully self-explanatory.

```

1    Sup guys!!
2    Hey man, how's it going?
join 3
4    Oh man I owe User3 money...
leave 4
drop 3    Oh, just missed User4, he owes me money
1    What did you say? Missed it...
...

```

Figure 4-2: A snippet of an evaluation script.

**runscript** This step takes the (painstakingly directed and produced) script in the previous step, and runs it against a specified system and user base. In particular the command line for running a script is shown below. The number of users is specified again both for simplicity — no reading parameters from the top of the file, or running through the file to figure it out — and because it allows for a script to be run with varying numbers of users. It only requires as many users as there are speakers, but the command-line used with **runscript** can introduce any number of listeners. The last option is a boolean: should we convert messages to/from base 64 as part of the process? This entails the conversion taking its toll on the computation and message size overheads.

```
./runscript (script) (system) (# users) (base 64?)
```

#### 4.2.5 Results

In order to simulate traffic, I observed some typical chat packets sent using the Jabber protocol. Formatted in XML, the packets are a hefty 500-600 bytes for a typical chat message, without the SSL/TLS encryption option. As a result, I generated and ran a script:

```

ruby genscript.rb 5 30 550 0 0.1 0.1
./runscript script-5-30-550-0-0.1-0.1 eval_sys 5 1

```

This simulates 30 spoken lines in a 5-user chat room with an average payload of 550 bytes. The speaker is selected at random and there is a 10% chance of someone joining or leaving the room, and a 10% chance of a message being dropped. This produced the graphs seen in Figures 4-3, 4-4, 4-5, and 4-6.

#### 4.2.6 Live

The last stage we went through was a live-action script. Using the Gaim plugin, I entered lines from a script at 5-second intervals, capturing the resulting traffic using Ethereal.

#### 4.2.7 Comparison

Well, we've seen that, especially when it's base64 encoded, confidentiality and authentication commands quite a premium in resources over plaintext. However, how does PKGE stack up to existing schemes? Since there is not an equivalent tool trivially at our disposal, let's take one accessible example in particular and extrapolate. PGP-signed-encrypted email,

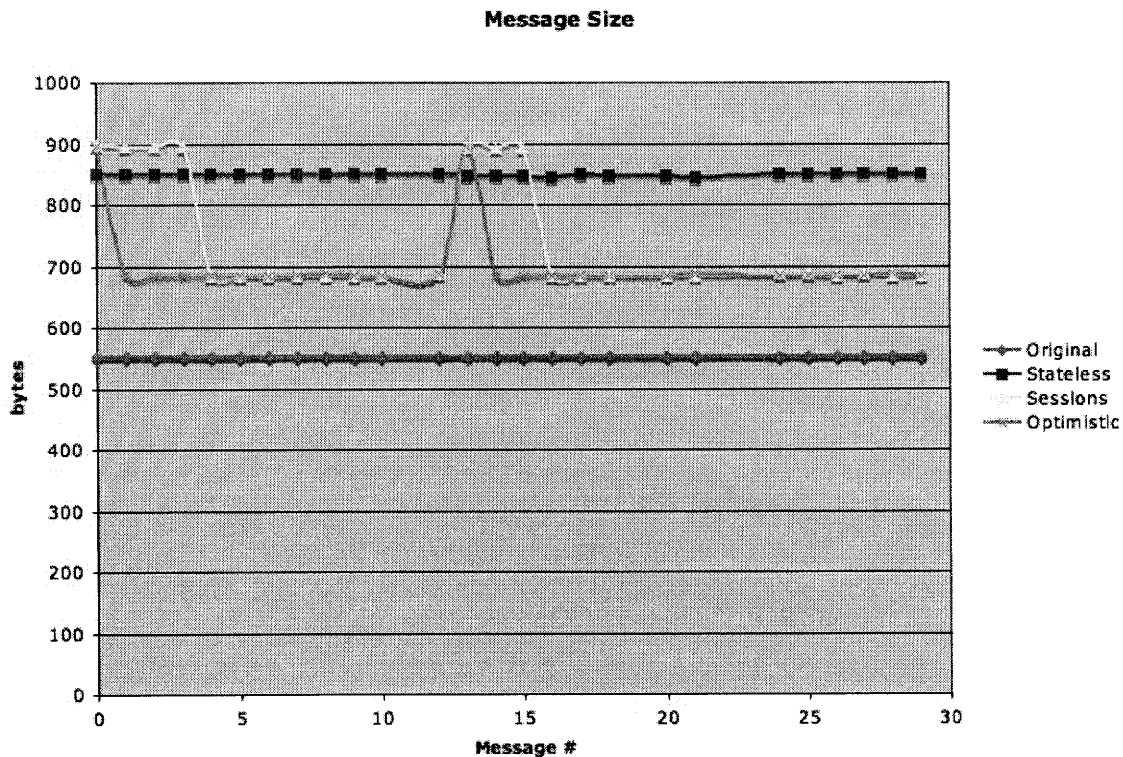


Figure 4-3: Message size overhead.

for example, is one application that provides an equivalent service in one of our target environments.

In order to compare the two, let's consider the same situation of 5 recipients with a payload of 550 bytes. Note that email is the weakest situation that PKGE may be put into, since it disallows all of the stateful optimizations that we have made possible. Regardless, we will use 1024-bit RSA keys for encryption and signing, despite these being significantly less secure than the 224-bit EC curve we are using<sup>1</sup>.

Our test strategy was simple: send a signed, encrypted email to 5 recipients, and measure the overhead as compared to the bare email. One unfortunate item is that the PGP plugin compresses the input before the operations (which is on our todo list), so we simply tested by sending a couple bytes, so that we could measure real overhead. Then we can compare overheads as apples-to-apples, since we do not pull any such sleazy tricks.

The result of encrypting and signing the text "test" to 5 recipients was a (base 64) output of 2,470 bytes. This is enormous.

#### 4.2.8 Evaluation Finale

In this section, we gave hard and fast numbers for computational and space overhead imposed by PKGE. Now we want to connect these numbers back to the very beginning, where we described the environments that we are targetting with this toolkit. What can

<sup>1</sup>A 1024-bit RSA key is roughly equivalent to 160-bit EC key in terms of security.

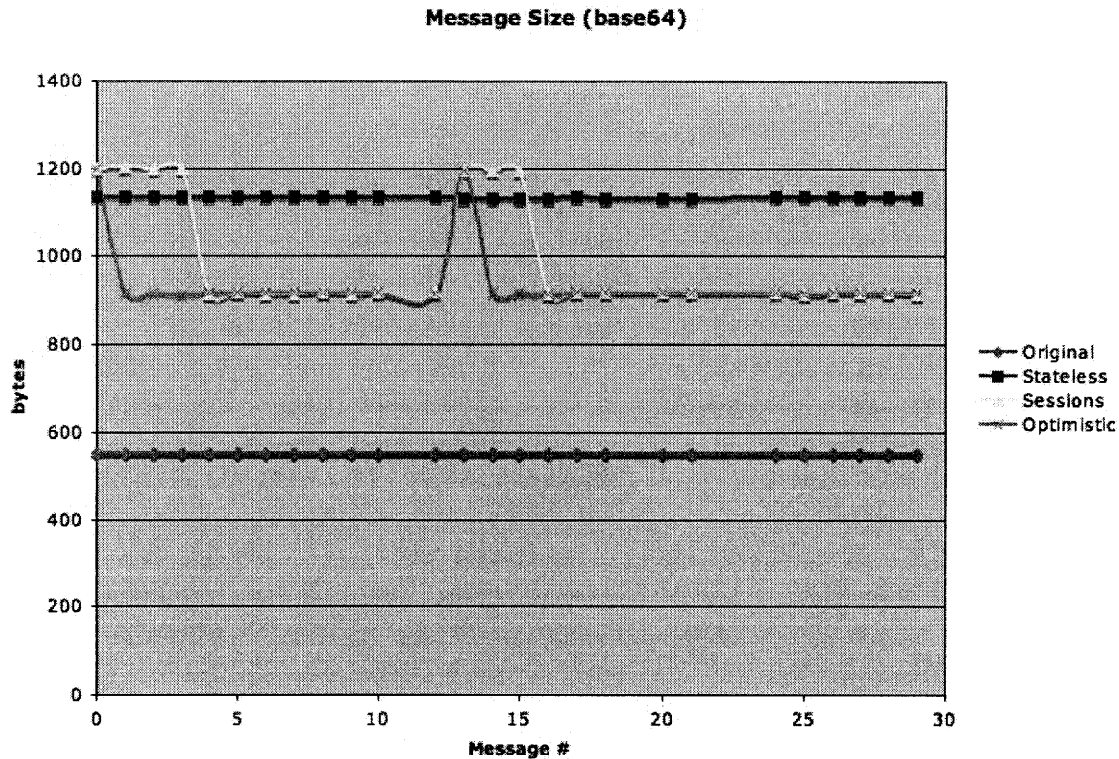


Figure 4-4: Message size overhead.

we take away in particular from this discourse on evaluation procedure and performance?

**CPU** Using the computation times given by the Powerbook (on which I am merrily typing away), can we do some order-of-magnitude calculations to figure out if this library can run on a PDA? Probably nothing meaningful, but let's give it a shot anyway. Assuming a current-level PocketPC (for example, the *Dell Axim X5 Pocket PC*), we are dealing with a 400 MHz Intel XScale CPU, with 64 MB SDRAM. Memory consumption is not an issue, as the largest

Using `top`, we observed that the `RSIZE` was a mere 1.66 MB — `VSIZE` was a more ghastly 31 MB, due to linkage with a number of heavyweight libraries like OpenSSL and GMP. It is completely unclear what performance implications the memory requirements have, and although I do not feel comfortable even giving estimated computation times without a trial run, it is safe to say that header computation requires less than a second, and that is good enough when using any of the stateful protocols.

**Space** Space perhaps offers less of a constraint, as the system space required has already been shown to be reasonable, and the bandwidth hit required by sealing messages has also been shown to be on the order of a couple hundred bytes. The keys to success in this arena are efficient transfer of recipient sets and the incremental public key. With respect to space, PKGE does not step on toes.

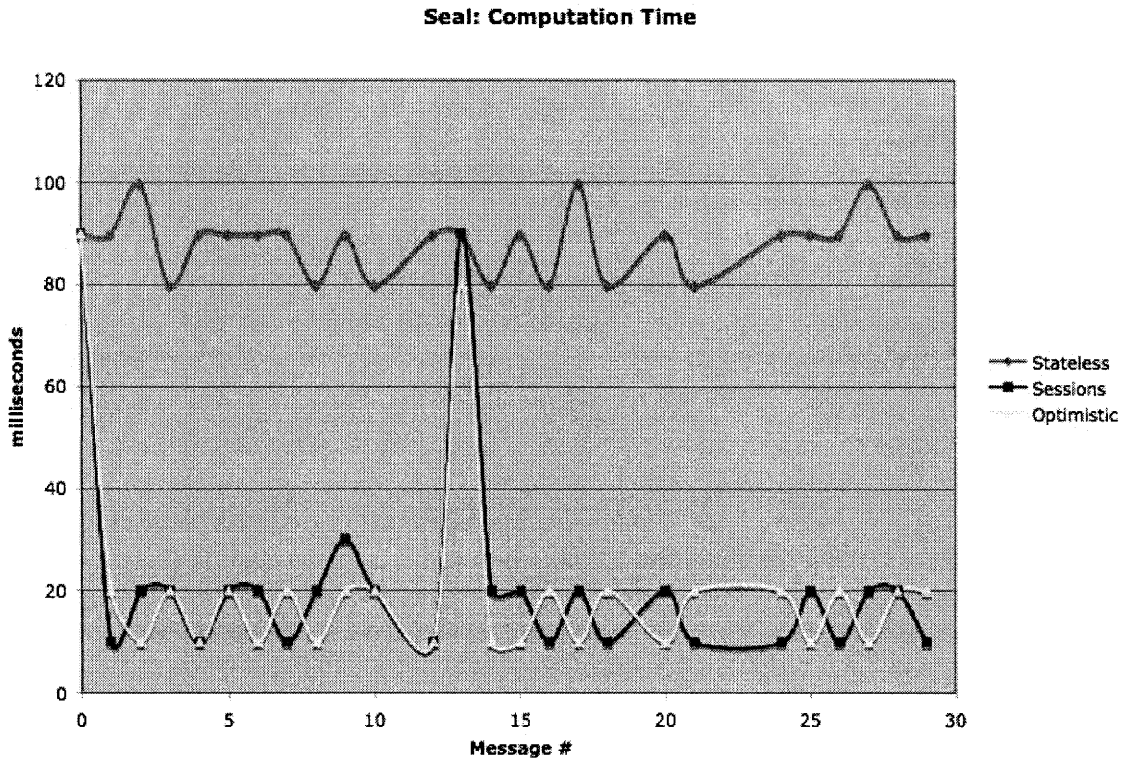


Figure 4-5: Computation required to seal a sequence of messages.

**Conclusion** We are very heartened to find that, at the close of our tortuous journey together, PKGE’s performance comfortably meets our goals of being deployable in a variety of scenarios. Not only that, but there are a number of optimizations I suggest that could make things *even better*. So ends *Evaluation*, on an optimistic note.

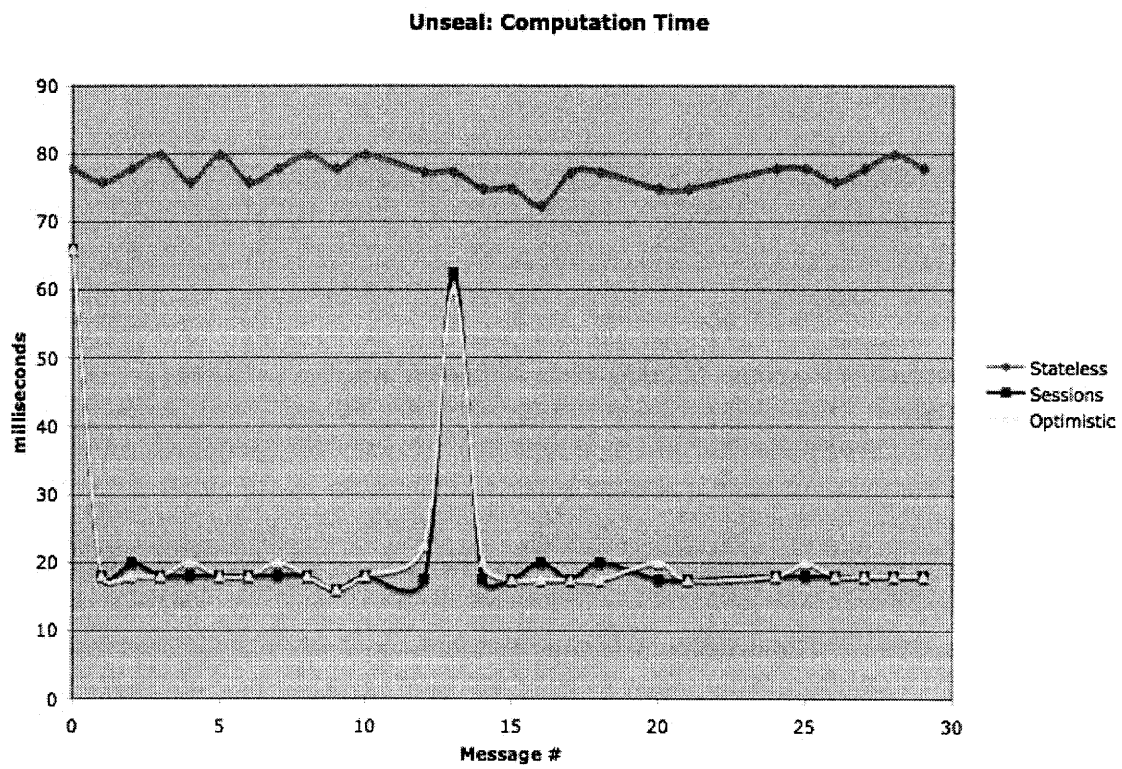


Figure 4-6: Computation required to unseal a sequence of messages.

## 4.3 Conclusion

At long last the sun is setting on this grand adventure. Nearly freed from the shackles that have bound my hands to this keyboard for the past month, my battle-scarred fingertips can scarcely type with any speed or precision. In the past 104 pages we have discovered a terrible void in the state-of-the-art in securing group communication. We have seen situations that sorely lack this capability and we have thought critically about what properties such a solution must have to broadly succeed in these situations.

From these musings sprang forth a design for a Public Key Group Encryption system, offering stateless, disconnected confidentiality and authentication to any number of recipients for a constant price in space overhead. We detailed exactly how this system should be put together, and then we executed those specifications to the best of our ability. Concurrently, we develop sophisticated automated tests to give confidence in the effort. After all, there is a wise adage:

**Trust, but verify.<sup>2</sup>**

With a new beast in the stable, we felt compelled to take him for a ride to see what he could do. We designed a series of trials that would give us an idea of the sort of things he would be good for. Having become intimately familiar with our creation, we provide manuals for the different roles of interaction and this document providing the blueprints. We even provide a list of improvements that may be undertaken by the intrepid developer (Appendix I)..

The synergistic duo of PKGE and Gaim was too tantalizing to ignore. We implemented a plugin that allows PKGE to be used to secure communication across any chat network on any computing platform<sup>3</sup>. It is both an incredibly useful application in its own right and also serves as a paragon of programming using the PKGE API that may be referenced by future developers.

Basically we worked out the service architecture and engineering issues like user registration and initialization, key storage and retrieval, representation of recipient sets, authentication, and encryption. We have developed a prototype, evaluated our implementation, ran experiments, measured overhead, compared ourselves to a more conventional rival, and developed a motivating launch application.

By producing this prototype, we demonstrated feasibility of a solution that has a clean API on top of a straightforward cryptography scheme. It will serve as a foundation for determining bottlenecks of this scheme for real group applications in the targeted environment. It is part of a larger effort at Lincoln Laboratories, and will be the benchmark that future solutions will be measured against.

---

<sup>2</sup>Old Russian proverb popularized by Ronald Reagan in his 1989 farewell address.

<sup>3</sup>Technically, it supports any **known, public** chat networks on any computing platform **that runs Windows, or some flavor of Linux**, but my original statement covers everything except academically pedantic cases.



# Appendix A

## Terminology

This chapter serves as a glossary that offers quick reference for any terms that are frequently used throughout this paper.

**BCE** Broadcast Encryption, the name given typically given in cryptographic publications to any technique that allows (efficient) encryption to large audiences. Used interchangeably with BCE in this paper.

**BGW** Boneh-Gentry-Waters, authors of the cryptographic piece-de-resistance in this thesis. Used interchangeably with BCE in this paper.

**canonical time** Relating to forward secrecy, this is essentially the "floor" function for timestamps. All timestamps within a timestep have the same canonical time, so comparing two messages' canonical time allows easy determination if they can use the same key.

**forward secrecy/security** If an adversary saves secure communications and at some later point compromises a user's system, he should not be able to use any of that user's abilities to read past messages.

**framework** When speaking of the PKGE *framework*, we are referring to the shared C library implementation of the Public Key Group Encryption service as well as the supporting system generation and profiling tools.

**(key) roll-over** The process by which the current private BGW/BCE key is swapped for the next upon expiration of a *security epoch*. Connotes the extra bit of complexity present with the *window*.

**seal** To encrypt and sign a plaintext message

**security epoch** A period of time during which the same keying information may be used and forward secrecy is not guaranteed. This is essentially the discrete unit of forward secrecy — messages are only rendered forward secure after expiration of the security epoch they are sealed to.

**session** The context defined by the group and timestep a message is addressed to. It serves to cache the message header and decryption key to avoid costly computation, and defines an *id* that may be referenced in subsequent messages and allows omission of the vast majority of space overhead from the sealed message.

**store** Upon loading a system, its path and handle are added to the *store*, where it may subsequently be referenced by the path alone.

**timestep** Relating to forward secrecy, this is the domain of time during which a single temporal key is valid. Interchangeable with *security epoch*.

**unseal** To verify the sender's signature and decrypt a ciphertext message

**verified** Used in the context of a session, in the *sessions* protocol. A session becomes verified for a particular user when he receives a message from everyone else in the group, containing that session's header and association with the same private session id. Once a session becomes verified, the user begins sending only optimized messages, with only a private session id in the way of header.

**window** The period of time at the start of a *security epoch* during which the previous epoch's key remains accessible, for the purpose of ameliorating problems inherent in attempting a global synchronous key switch. Once the *window* expires, the previous epoch's keys are irrevocably deleted.

# Appendix B

## Administrator Manual

This section is designed for system administrators. It details how to configure, generate, and add users to a system. Secure handling of sensitive data is also discussed briefly.

### B.1 Configure

PKGE gets a large number of settings and parameters from its configuration file. The recommended method for producing one is to edit the `default.cfg` (see J.1) provided and save it as your own. It is well-commented, so I will simply briefly mention the most relevant parameters.

- pairing configuration — the particular elliptic curve forms the base of the system and determines the basic performance and security properties. PBC must be used to generate alternate configurations.
- N — how many users might you conceivably have sign up for your system during its lifetime?
- timestep — how often do you want keys to "roll-over", that is, cause previous messages to become unreadable. The smaller the timestep, the smaller the window of vulnerability, but the greater the resource requirements.
- window — "grace period" during which messages expiring during the previous timestep can be unsealed in the next
- lookahead — how long should clients be able to go without retrieving updated keys from the central server?
- session\_table\_size — how many different groups might a user form during a single timestep? default of  $2^{10}$  capacity should be fine for all uses, unless you are an aggressive optimizer.

#### B.1.1 An Elliptic Curve to Call Your Own

This section describes how to use PBC to generate your very own elliptic curve. This section is meant for the administrator who took a look at what was provided, sees the general characteristics of the different types of curves, and decides he has his heart set on a "Type F" curve using 123.5 bit keys and won't take "no" for an answer. To get a gist of the characteristics of the curves provided, see Table 1.3.2.

## B.2 Generate

This step takes the configuration file you caringly crafted in the previous step and boils it down into a system file and some boilerplate cryptographic elements. In particular, run the following command:

```
$ pkgegen setup /optional/path/system_name system.cfg
```

This creates the directory `/optional/path/system_name` and gives you everything you need to start the party. Thereafter, and in general, you may reference the system using its path – feel free to slide it around the file system, as long as the internal file structure remains intact. The system generation takes almost no time at all — cryptographic computations are mostly put off until they become absolutely necessary (e.g. users are added to the system).

## B.3 Add Users

Your chariot awaits, now you need to fill the seats. Presumably you have some existing infrastructure that PKGE should fit within. This means that there is a database of everyone for whom you are interested in providing security. Export the list to a tab-delimited text file. The details are negotiable, but the `pkgegen` tool wants very badly to generate X.509 certificates for the user base, and if you provide the following fields it will do just that:

```
CommonName  SurName  GivenName  Title  Position  Group
```

The name that the central authority is known as should be alone on the first line, with user information in the above format taking up all subsequent lines. The user's "Common Name" is the human-readable and human-rememberable textual representation he will be forever known to PKGE as. Once your user information is exported to that format, run `pkgegen`:

```
$ pkgegen add /optional/path/system_name txt users.txt
```

This step may take a while, depending on the number of users. It generates all of the public and private keys needed by the new progeny.

## B.4 Distribute

The system is all set up, and now you need to somehow get it to the eager users. According to the usage vision, this is best accomplished by setting up a server and using the existing infrastructure (there is one, right??) to authenticate users that come looking to download the latest and greatest cryptographic development of the 21<sup>st</sup> century. Once an authenticated user makes a request, the server should require him to select a PIN to be used to protect his keys. Once selected, the server should run the following command:

```
$ pkgegen dist /optional/path/system_name user_name 12345
```

Initial Distribution	Update Distribution
<code>system_name/system_name.pkg</code> <code>system_name/users/Alice.usr</code> <code>system_name/certs/Alice.der</code> <code>system_name/bigpk/*.pk</code>	<code>system_name/users/Alice.usr</code> <code>system_name/bigpk/&lt;any she does not have&gt;.pk</code>

Table B.1: Listings of files that Alice needs on setup and subsequent updates.

Of course, substitute the common name of the user, and probably disallowing 12345 as a PIN would be a good move[37]. Note: valid PIN range is  $[0, 2^32 - 1] = [0, 4294967295]$ , limited to 4 bytes. Therefore, you can require up to a 9-digit PIN. At any rate, once the system has finished encrypting the user's profile, generating, for example, `system_name/Alice.usr`, it is the server's responsibility to package up the files according to Table B.4.

If Alice already has a system and merely wants to update her system (ie. get her key set for the next "lookahead" period), the server has to generate a new `.usr` file for her using the same command, as well as ensure that she receives any new public key elements that have been generated since her last update (see Table B.4. Those are easy to determine by first checking the date that `Alice.usr` was last accessed, and selecting all `.pk` files with creation dates later than that. One more time because it's important and I know you've had a long day: **You are responsible for keeping track of which public key blocks Alice needs and delivering them.**

Note that `$ pkgegen dist` must be run in both cases. Note also that the user files beginning with an underscore ('\_') are **unencrypted** user keys, and must be protected at all costs. There is no reason for any process other than `pkgegen` to access them. Also note that if a user loses or forgets his PIN, he may simply follow the "Update Distribution" protocol, since he is queried each time for a PIN with which to encrypt his `.usr` file.

## B.5 Conclusion

That is all you need to know about administering your very own Public Key Group Encryption system.



# Appendix C

## Client Manual

This manual is for developers that wish to leverage PKGE in their application. It describes the basic API, the different protocols, and the recommended methods for handling certificates and protocol error messages.

### C.1 Client API

So you have an advanced elliptic curve-based system for encryption and authentication of all communication loaded; how do you use it? There are varying degrees of complexity depending on your feature and performance requirements. Here is a simple snippet that lets Alice send a message to Bob.

```
user_t Alice;
pkge_user_load(&Alice, "/path/to/pkge_system", "alice", pin_entry);

group_t alice_and_bob = pkge_group_new(Alice);
pkge_group_add(alice_and_bob, "Bob");

data_t alice_says = { input, input_len };
pkge_seal(alice_and_bob, &alice_says);
```

There you have it. This relatively minimal set of instructions decrypts and loads Alice's private keys, selects a communication protocol, creates a new recipient group to add Bob to (after verifying his identity using a X.509 certificate mind you), calculates a unique key using his public key, and encrypts the provided message using that calculated key. Bob would have to do the following to read that message.

```
user_t Bob;
pkge_load_user(&Bob, "Bob", bobs_pin);

group_t bob_and_alice = pkge_group_new(Bob);
pkge_unseal(bob_and_alice, &alice_says);
```

One observation is that PKGE relies on the `data_t` data type to represent messages. It is a simple structure:

```

struct data_t {
    byte_t* data;
    size_t len;
};

```

A second observation is that no mention was made of the *protocol*. It turns out `pkge_user_load` implicitly compares the given system path to every other currently loaded system. If one matches, the user is loaded using that system, but if not, it calls `pkge_init`, loading the system using the `SESSIONS` protocol. One final observation is that the argument “`const char* sys_filename`” pops up a lot. There had to be some token to identify which system a function is talking about, since this is meant to be a `shared` library, so the path name is overloaded to fill that function. The client must have the path at some point in order to load the system, so he would likely find it convenient to keep the path around, making it a non-onerous token. Note that the path is simply string-compared for equality, so be sure to use a single canonical form, rather than multiple strings that may refer to the same disk entry.

## C.2 Initialization

---

```

int pkge_init      (const char* sys_filename, unsigned long op_mode)
int pkge_user_load(user_t* user, const char* sys_filename,
                  const char* user_name, pin_t pin)

```

---

These functions handle initialization. When called, `pkge_init` checks if `sys_filename` corresponds to any currently-loaded system. If not, it loads the system, initializing it with the protocol given (`op_mode`). You may pass in one of:

```

PKGE_PROTOCOL_{STATELESS, SESSIONS, OPTIMISTIC}

```

to select a protocol. If `sys_filename` corresponds to an already-loaded system, the protocol is set to `op_mode`, and no further action is taken. Calling `pkge_user_load` initializes the system if necessary, using `PKGE_PROTOCOL_SESSIONS` as the default protocol.

An important aside is that a system may be loaded using 3 referentially distinct paths (as opposed to cosmetically distinct). A system residing in `/usr/sys` may be loaded by passing “`/usr/sys`”, “`/usr/sys/sys.pkge`”, or “`/usr/sys/sys.admin`”, assuming that the latter two exist. In the first two cases, `/usr/sys/sys.pkge`, the non-administrative system, will be loaded. In other words, to write a function that works on the administrative system, be sure to load the “`.admin`” system file explicitly!

Example:

```

user_t alice;
pkge_user_load(&alice, "/path/to/system", "Alice", 12345);
// => Alice loaded, system init'd
pkge_init("/path/to/system", PKGE_PROTOCOL_OPTIMISTIC);
// changed protocol to "optimistic"

```



## C.3 Groups

---

```
group_t pkge_group_new    (const user_t u)
int      pkge_group_add   (group_t g, const char* name_to_add)
int      pkge_group_remove(group_t g, const char* name_to_remove)
char*    group_str        (const group_t g)
int      pkge_group_free  (group_t g)
```

---

No surprises here. Straight to examples.

Example:

```
user_t alice;
pkge_user_load(&alice, "/path/to/system", "Alice", 12345);
group_t lover = pkge_group_new(alice);
pkge_group_add(lover, "Bob");

/** flirts **/
/** Bob leaves for work **/

pkge_group_remove(lover, "Bob");
pkge_group_add(lover, "Carl");
char* tryst_description = group_str(lover);

/** Alice has been cheating on poor Bob all along... =[ **/

pkge_group_free(lover);
free(tryst_description);
```

## C.4 Certificates

---

```
int pkge_get_cert (data_t* cert, const char* sys_filename, const char* name)
int pkge_add_cert (const char* sys_filename, const data_t* cert)
int pkge_have_cert(const char* sys_filename, const char* name)
int pkge_get_x509 (X509** x509, const char* sys_filename, const char* name)
```

---

These functions manage user certificates. `pkge_get_cert` returns a DER-encoded certificate as a message ready for transmission, `pkge_add_cert` accepts a DER-encoded certificate and adds it to the PKGE repository, and `pkge_have_cert` tests for a particular certificate's presence, given a user's name.

`pkge_get_x509` is only available if `<openssl/x509.h>` is included previously, and it allows easy extraction of the X509 OpenSSL object so that the application can play with it using the mechanisms provided by OpenSSL.

Example:

```
data_t my_cert = {0,0};
pkge_get_cert(&my_cert, "/path/to/system", "Alice");
```

Error code	Translation
PKGE_ERR_CORRUPT_MSG	Expected data tag was not found
PKGE_ERR_NOT_RCPT	You are not authorized to read this
PKGE_ERR_CERT_MISSING	I don't have the sender's certificate
PKGE_ERR_SESSION	There was a message lost, or you lost state
PKGE_ERR_INVALID_SIG	The sender's signature didn't verify
PKGE_ERR_EXPIRED	The message was sealed to a timestep that has expired

Table C.1: Error codes for `pkge_unseal`.

```
// Alice → my_cert → Bob

pkge_have_cert("/path/to/system", "Alice"); // => 0 / FALSE
pkge_add_cert ("/path/to/system", &alices_cert);
pkge_have_cert("/path/to/system", "Alice"); // => 1 / TRUE
```

## C.5 Cryptographic Operations

---

```
int pkge_seal(const group_t to, data_t* msg)
int pkge_seal_delayed(const group_t to, data_t* msg, int days)
int pkge_unseal(group_t to, data_t* msg)
```

---

We finally made it to the bread 'n butter! Support for forward secrecy requires that messages have a finite lifetime. In particular, to ensure that future key compromise preserves the confidentiality of past messages, keys must be irrevocably deleted once their timestep-in-the-sun has passed. For interactive communication applications, this is not a concern. However, for applications such as email, where there is possibly a lengthy delay between the sending and receiving, it may be necessary to specify a longer length of time during which the message can be unsealed. Thus, `pkge_seal_delayed` offers to preserve a message's unsealability for an integral number of days. `pkge_seal` is simply an alias for `pkge_seal_delayed` with `days=0`.

The responsible developer should always check the return value of `pkge_unseal`, since it has, without doubt, the widest variety of descriptive error codes to return. They are described in Table C.1

Example:

```
pkge_seal_delayed(addressed_to, message, 30);

/** Sits in their mailboxes */
/** They start up their client one day */

user_t bob;
pkge_user_load(&bob, "/path/to/system", "Bob", 12345);
group_t bobs_email = pkge_group_new(bob);
pkge_unseal(bobs_email, msg);
```

## C.6 Utility

---

```
int pkge_to64(data_t*)
int pkge_from64(data_t*)
```

---

These methods encode/decode base64 in place.

Example:

```
pkge_seal(group, msg);
pkge_to64(msg); → text-only channel → pkge_from64(msg);
                                                    pkge_unseal(group, msg);
```

## C.7 Error Handling

---

```
data_t* pkge_get_error_for(const group_t g, data_t* msg)
int pkge_register_sender(const group_t g, int (*send)(data_t*))
```

---

These methods exist to explicitly handle the error condition that occurs when the messages being sent are optimized such that they are sent without the full header. A client may receive such a message and not have a record of the session id that identifies which key should be used, a scenario that may arise when the client experiences a loss of state such as a crash, or when the Optimistic protocol is in use and a group member does not receive the first and only setup message.

These methods provide two different ways to recover. One requires diligent checking of the return code from `pkge_unseal` and the other requires a callback be written that can accept a message to send. In particular, if an invocation of `pkge_unseal` returns `PKG_ERR_SESSION`, `pkge_get_error_msg` may be called, passing in the same message given to `pkge_unseal`, to get an error message that should be relayed to the group to elicit assistance. If a callback is registered, it will be called in this situation with the same error message. An identifier is added to the message by the unseal operation and removed when the error message is retrieved, so if both a callback is registered and `pkge_get_error` is called, the latter will return a null pointer.

In any case, an error message indicates a likelihood that subsequent unseals will fail, until a recovery message is received. The diligent application will save incoming messages that the unseal operation fails for and retry when unseal begins to succeed.

## C.8 `pkge_shutdown()` will set you free

---

```
int pkge_user_free(user_t u)
int pkge_group_free(group_t g)
int pkge_shutdown(const char* sys_filename)
```

---

These methods free resources. All groups allocated with `pkge_group_new` must be freed by the user. However, we are more permissive with the handling of `user_t` objects: a system shutdown causes all loaded users to be freed as well, so if you are unloading PKGE with a number of users loaded, it is perfectly acceptable to call `pkge_shutdown` and assume that everything else is taken care of (except for the groups!).

## C.9 Administrative API

This section describes the functions that may be useful when writing programs to operate on the central authority's system. It contains functionality like creating new users and generating key updates for users. There are few enough functions that we don't bother breaking them into functional groups.

### C.9.1 Initialization

---

```
int  _pkge_setup(pkge_t* sys, const char* sys_filename, const char* cfg)
pkge_t _pkge_init (const char* sys_filename, unsigned long mode)
```

---

A system may be created by the `_pkge_setup` function, using a textual configuration buffer. This `pkge_init` function exists for the dual purpose of initializing the system *à la* `pkge_init` but with the added utility of returning a handle to the system structure. Currently the system handle is only necessary for adding users, serializing the system file, or getting the authority's X.509 signing key (`_pkge_ca_key`).

### C.9.2 Users and Updates

---

```
int  _pkge_user_create (user_t* pu, pkge_t sys, const char* org_name,
                        user_id_t id, char* user_info)
int  _pkge_gen_update  (user_t u, pin_t pin)
int  _pkge_add_user_X509(pkge_t sys, X509* x509, EVP_PKEY* sign_key)
int  _pkge_add_users_txt(pkge_t sys, BIO* user_info)
int  _pkge_user_save   (user_t u, pin_t pin)
```

---

If the single user-creation procedure is too coarse for you, there are also `certs_create` and `user_create`. The former creates a certificate using the given information, signs it with the given CA key, and returns the newly created user key (public and private). The latter takes such a key, plus his PKGE name and id, and returns a new user object, which may be saved using `_pkge_user_save`.

The update functionality is all rolled into this single function call: `pkge_gen.update()`. It deletes stale keys from the user's key file, replaces them with new keys (assuming it hasn't been previously called during the current timestep), and encrypts it using the given pin. If pin is 0, the file is saved unencrypted.

### C.9.3 Serialization

---

```
int  _pkge_save          (pkge_t sys)
int  _pkge_serialize_sanitized(pkge_t sys, BIO* fp)
int  _pkge_serialize     (pkge_t sys, BIO* fp)
```

---

The system serialization procedures have a bit of flexibility. The one-line wonder `_pkge_save` works out all of the disk and file naming issues, saving both administrative and user versions of the system to file. For more flexibility, intermediate functions are provided in `_pkge_serialize` (admin system only) and `_pkge_serialize_sanitized` (user system only). User serialization is accomplished by `_pkge_user_save` — if the given PIN is 0, a fixed-key-encrypted user file (e.g. `r0b.usr`) is generated.

### C.9.4 The Store

---

```
int    _pkge_store_add  (const pkge_t sys, const char* alias)
int    _pkge_store_remove(const char* alias)
pkge_t _pkge_store_get  (const char* alias)
int    _pkge_store_free (void)
```

---

The *store* is where the library stores system object references, mapped to by the path that was used to load them. Systems may be added or removed from this store; it might be desirable to remove a system, for example, if you do not want it to be shared. In that case, when a second instance passes the same path, no system will be found and it will be loaded anew.



## Appendix D

# PBC Library 0.2 Manual

Author: Ben Lynn

### D.1 Overview

Pairing-based cryptography centers around a particular function with interesting properties.

Let  $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$  be cyclic groups of prime order  $r$ . Let  $G_1$  be a generator of  $\mathbb{G}_1$  and  $G_2$  be a generator of  $\mathbb{G}_2$ . Let  $e$  be a function:

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \implies \mathbb{G}_T$$

that is efficiently computable with  $e(G_1, G_2) \neq 1$  (nondegenerate) and  $e(G_1^a, G_2^b) = e(G_1, G_2)^{ab}$  (bilinear) for all integers  $a, b$ . We refer to this function  $e$  as a bilinear map or a pairing. When  $\mathbb{G}_1 = \mathbb{G}_2$  we say that the pairing is *symmetric*, otherwise we say the pairing is *asymmetric*.

With a few more conditions (for example, we may require the discrete log problem to be hard in  $\mathbb{G}_1$  and the existence of an efficiently computable isomorphism from  $\mathbb{G}_2$  to  $\mathbb{G}_1$ ), these pairings can be used to build a wide variety of cryptosystems.

The PBC library builds groups  $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$  and a pairing from given parameters. The underlying mathematics behind the groups are abstracted away in the interface. Routines to generate pairing parameters of are also provided.

### D.2 Testing the Library

Several test programs and curve parameters are bundled with the library.

#### D.2.1 Quick Start

After compilation succeeds, try:

```
test/testpairing < a.param
test/benchmark < c159.param
test/testibe < e.param
```

The \*.param files contain pairing parameters one might typically use in a real cryptosystem. Many of the test programs read the parameters from standard input.

## D.2.2 Test Programs

**listmnt** Searches for discriminants  $D$  that lead to MNT curves with subgroups of prime order.

**genaparam, gencparam, geneparam, genfparam, genbgn** Prints parameters for a curve suitable for computing pairings of a given type. The output can be fed to some of the other test programs. The program `gencparam` should be given a discriminant as the first argument.

**testhilbert** Prints the Hilbert polynomial for a given discriminant.

**testsig, testibe** These programs read curve parameters on standard input and perform computations that would be required in an implementation of the Boneh-Lynn-Shacham short signature scheme and the Boneh-Franklin identity-based encryption scheme. `testbb`, `testbls`, `testbbs`, `testibs`

These test programs are more realistic, and test the implementations of the Boneh-Boyen, Boneh-Lynn-Shacham, Boneh-Boyen-Shacham signature schemes and the Cha-Cheon and Sakai-Kasahara-Schnorr identity-based signature schemes

**benchmark** Reads pairing parameters on standard input and benchmarks the pairing.

## D.3 The PBC API

Programs using the PBC library should include the file `pbh.h`:

```
#include "pbh.h"
```

and linked against the PBC library, e.g.

```
gcc program.c -L. -lpbc
```

The file `pbh.h` includes `gmp.h` thus all GMP functions are available. To use a pairing, first declare and initialize:

```
pairing_t pairing; pairing_init_inp_str(pairing, stdin);
```

In this case, the pairing parameters are fed to the program on standard input<sup>1</sup>. Now `pairing->G1`, `pairing->G2`, `pairing->GT` and `bilinear_map()` can be used as follows. Declare and initialize some group elements:

```
element_t x, y, z;
element_init(x, pairing->G1);
element_init(y, pairing->G2);
element_init(z, pairing->GT);
```

To pick random elements of  $\mathbb{G}_1$ ,  $\mathbb{G}_2$  and compute a pairing:

```
element_random(x);
element_random(y);
bilinear_map(z, x, y, pairing);
```

Now  $z = e(x, y)$ . To raise  $x$  by some random exponent  $r$ :

---

<sup>1</sup>e.g. `./program >a.param`



```

element_t r;
element_init(r, pairing->Zr);
element_random(r);
element_pow_fp(x, x, r);

```

## D.4 Pairing Types

The PBC library implements the pairing using different types of curves. Which curve to use depends on the application. To make it easy to refer to them, I have labelled them with letters. See below for the details.

The library provides pairing parameters useful in typical pairing-based cryptosystems. If you're not interested in what the details are, always use Type A when a fast pairing is desired, and Type F when short group elements are needed. Type C are not as slow as type F, but cannot be quite as short.

### D.4.1 Type A

We use the curve  $y^2 = x^3 + x$  over the field  $\mathbb{F}_q$  for some prime  $q$ . It turns out  $\#E(\mathbb{F}_q) = q + 1$  and  $\#E(\mathbb{F}_q^2) = (q + 1)^2$ . Thus the embedding degree  $k$  is 2.

We set things up so that  $q + 1 = r \cdot h$ , for a particular  $r$ . For speed, we pick  $r$  to be a Solinas prime, that is,  $r$  has the form  $2^a \pm 2^b \pm 1$  for some integers  $0 < b < a$ .

We also choose  $q = -1 \pmod{12}$  so we can implement  $\mathbb{F}_q^2$  as  $\mathbb{F}_q[\iota]$  (where  $\iota = \sqrt{-1}$ ). Also, since  $q = -1 \pmod{3}$ , cube roots in  $\mathbb{F}_q$  are easy to compute. This latter feature may be removed because I have not found a use for it yet (in which case only  $q = -1 \pmod{4}$  is guaranteed).

Pairings on type A curves are the fast and ought to be used where the main concern is efficiency. Typically,  $r$  should be about 160 bits long and  $q$  about 512 bits. In this case, elements of groups  $\mathbb{G}_1$  and  $\mathbb{G}_2$  take 512 bits to represent.

Additionally, the pairing is symmetric, that is  $\mathbb{G}_1$  and  $\mathbb{G}_2$  are in fact the same group.

`a_param` struct fields:

```
exp2, exp1, sign1, sign0, r:
```

```
    r = 2^exp2 + sign1 * 2^exp1 + sign0 * 1 (Solinas prime)
```

```
q, h:
```

```
    r * h = q + 1
```

```
    q is a prime, h is a multiple of 12 (thus q = -1 mod 12)
```

### D.4.2 Type B

This type is reserved for the curve  $y^2 = x^3 + 1$  over  $\mathbb{F}_q$  with  $q = -1 \pmod{12}$ . It has yet to be implemented as I have not seen a compelling reason to use it.

There are advantages unique to this curve however. Since cube roots in  $\mathbb{F}_q$  are fast and guaranteed to exist, for any given value of  $y$ , it is easy to solve for  $x$ . Also, the coefficient of  $x$  is zero in the curve equation, simplifying some equations (e.g. point doubling).

There is also at least one drawback when compared with the similar type A pairing. If symmetry of the pairing is insisted upon, some optimizations are not possible. If ever implemented, perhaps I will divide this case into two subtypes, one symmetric, and the other asymmetric but slightly faster.

### D.4.3 Type C

These are ordinary curves of with embedding degree 6, whose orders are prime or a prime multiplied by a small constant. These are constructed using the method due to MNT.

A type C curve is defined over some field  $\mathbb{F}_q$  and has order  $h \cdot r$  where  $r$  is a prime and  $h$  is a small constant. Over the field  $\mathbb{F}_q^6$  its order is a multiple of  $r^2$ .

Typically the order of the curve  $E$  is around 170 bits, as is  $\mathbb{F}_q$ , the base field, thus  $q^k$  is around the 1024-bit mark which is commonly considered good enough.

Using type C pairings allows elements of group  $\mathbb{G}_1$  to be quite short, typically 170-bits. Because of a certain trick, elements of group  $\mathbb{G}_2$  need only be 3 times longer, that is, about 510 bits rather than 6 times long. They are not quite as short as type F pairings, but much faster.

```
c_param struct fields:
q   F_q is the base field
n   # of points in E(F_q)
r   large prime dividing n
h   n = h * r
a   E is given by y^2 = x^3 + ax + b
b
nk  # of points in E(F_q^k)
hk  nk = hk * r * r
```

I sometimes refer to a type C curve as a triplet of numbers: the discriminant, the number of bits in the prime  $q$ , and the number of bits in the prime  $r$ . The `listmnt` program prints these numbers. The bundled type C curve parameters are the curves 9563-201-181, 62003-159-158 and 496659-224-224.

### D.4.4 Type D

This type is reserved for supersingular curves with embedding degree 6. Pairings are fast as optimizations specific to fields of characteristic 3 may be used. The embedding degree of 6 also means the representations of elements of  $\mathbb{G}_1$  are short.

On the other hand, the low characteristic also makes these curves more susceptible to Coppersmith's attack, so slightly larger fields are needed for security.

### D.4.5 Type E

The CM (Complex Multiplication) method of constructing elliptic curves starts with the Diophantine equation

$$DV^2 = 4q - t^2$$

If  $t = 2$  and  $q = Dr^2h^2 + 1$  for some prime  $r$  (which we choose to be a Solinas prime) and some integer  $h$ , we find that this equation is easily solved with  $V = 2rh$ .

Thus it is easy to find a curve (over the field  $\mathbb{F}_q$ ) with order  $q - 1$ . Note  $r^2$  divides  $q - 1$ , thus we have an embedding degree of 1. Hence all computations necessary for the pairing can be done in  $\mathbb{F}_q$  alone. There is never any need to extend  $\mathbb{F}_q$ . As  $q$  is typically 1024 bits, group elements take a lot of space to represent. Moreover, many optimizations do not apply to this type, resulting in a slower pairing.

This pairing is kept as a reserve, in case some day it is discovered that discrete log in field extensions is easier to solve than previously thought.

#### D.4.6 Type F

Using carefully crafted polynomials,  $k = 12$  pairings can be constructed. Only 160 bits are needed to represent elements of one group, and 320 bits for the other. They should be used when the top priority is to minimize bandwidth (e.g. short signatures). The pairing is slower than the other types. Also,  $k = 12$  allows higher security short signatures. ( $k = 6$  curves cannot be used to scale security from 160-bits to say 256-bits because finite field attacks are subexponential.)

#### D.4.7 BGN Curves

These are curves containing a subgroup of a specified order. Boneh, Goh and Nissim have use for an elliptic curve with subgroup order  $N = pq$  for large primes  $p, q$ .

To implement them, Type A curves can be used, only instead of a Solinas prime, we generate curves with a given group order. Type B can also be used (and that is what is suggested in their paper) but this is less desirable.



## Appendix E

# Cryptographic Fundamentals

In this section I give a brief review of the relevant cryptography. It should be enough to refresh anyone that has seen the concepts/schemes before, and I give references to more complete treatments for the newcomer. Cryptography is broken into functions of confidentiality and authentication. The former is dealt with by the `encrypt` and `decrypt` primitives, the latter with `sign` and `verify`.

### E.1 Symmetric

Symmetric Key Cryptography is the most straightforward way to keep information confidential. The two communicating entities both begin with the same key or sequence of bits. The sender performs an agreed-upon operation on the message, using the key as a parameter, and the receiver reverses the operation with his key. The most obvious (and information-theoretically secure) operation is `xor` (denoted  $\oplus$ ). Equation E.1 shows how an encrypted transmission might occur, for a message  $m$ , key  $k$ , and intermediate cipher text  $c$ .

$$m \oplus k = c \quad \rightsquigarrow \quad c \oplus k = m \tag{E.1}$$

To be secure,  $k$  must be random, and as long as the message (ie.  $|k| = |m|$ ) If one reuses the key  $k$ , it would be possible to gain information about the encrypted messages, since that would imply:

$$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k) = m_1 \oplus m_2 \tag{E.2}$$

due to the identity  $\forall k : k \oplus k = 0$ . If  $|k| < |m|$ , an adversary gains information about  $m$  by observing  $c$ . Due to the non-reusable nature,  $k$  is referred to as a *one-time pad*.

A more advanced method of encryption that also uses symmetric keys, is extremely fast, yet allows key reuse is the AES[56] (Advanced Encryption Standard) algorithm: the current state-of-the-art in symmetric encryption, covered in E.1.1. It is extremely fast (and secure) and can be used to encrypt data at a rate of  $\sim 100$  megabytes/sec on a typical desktop PC. Since we use it in PKGE, let's learn a bit more.

#### E.1.1 AES

Rijndael, an encryption algorithm first published in 1998 by Belgian cryptographers Joan Daemen and Vincent Rijmen, was rechristened the “Advanced Encryption Standard” (AES)[56] by the US Government in November 2001 (actually, a version of Rijndael with restricted block and key sizes). It is the algorithm actually used to encrypt the message bytes in our

Key Size	128, 192, 256 bits
Block Size	128 bits

Key Size	#Rounds
128	10
192	12
256	14

Table E.1: Vital information about AES.

$a$ Alice ( $\implies g^{ab}$ )	$g^b \iff g^a$ Eve $g^a, g^b$ ( $g^{ab} = (g^a)^{\log_g(g^b)}$ )	Bob $b$ ( $\implies g^{ab}$ )
------------------------------------	---	----------------------------------

Table E.2: Diffie-Hellman key agreement.

framework. The statistica vitæ (e.g. batting average, etc) are shown in Table E.1.1. Also note that the OpenSSL 0.9.8 implementation is FIPS 140-2 certified[57].

## E.2 Key Agreement

A problem with symmetric keys is actually establishing the same keys for both parties without out-of-band communication. The revolutionary solution to this problem was first (openly) proposed in the Diffie-Hellman paper of 1976[41]. It allows two parties with no prior knowledge of each other to establish a symmetric key over an insecure channel.

To achieve this feat between Alice and Bob, they first agree on a common finite field  $\mathcal{F}$  with generator  $g$ . Then Alice picks an element  $a$  at random, computes  $g^a$ , and sends it to Bob. Bob does the same with his own random element  $b$ . Now Alice knows  $a$  and  $g^b$ , Bob knows  $b$  and  $g^a$ : they both compute their new shared key,  $g^{ab}$ . The entire procedure is shown in Table E.2.

An eavesdropper Eve only knows  $g^a$  and  $g^b$ , which does not permit her to calculate  $g^{ab}$ . If Eve could solve the discrete logarithm, she could discover the private parameter:  $\log_g(g^a) = a$ .

Diffie-Hellman key agreement does not provide any authentication, and its security relies on the assumed hardness of the discrete logarithm problem.

## E.3 Contributory Keying

Contributory keying schemes - schemes wherein all members contribute to the group key generation - were the first to try solving the group keying problem using new theoretical techniques as opposed to protocols employing around old techniques.

By far the most prolific contributory group work has been relating to Group Diffie-Hellman protocols (GDH), a term given to a DH-style exchange leading to establishment of a key for  $n$  participants. These GDH schemes employ the same mathematics as the original DH, but use different protocols for passing and aggregating group keys between members. If all members perform the protocol correctly, then they have both contributed

a piece of information to form the group key, and they have successfully calculated it. The cost of such a key agreement depends upon the method of combination, and the particular communication patterns between the members during establishment.

One example of the member structure formed by STR, a contributory keying scheme, is shown in Figure 1. In Figure 1, all members are leaf nodes, with the internal nodes being calculated from the two children. The keys are combined pairwise using blinded exponentiation (the DH technique) all the way up the tree. Since transmissions are broadcasts, all nodes manage to learn the final group key by piecing together transmissions they have heard. The fact that they contributed a piece allows them to calculate it, while any eavesdroppers snooping around are stumped.

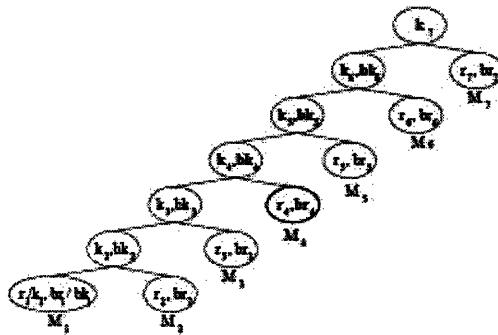


Figure E-1: Example STR Node Tree[49].

## E.4 Public Key

At this time, interested parties are directed to[40] for information on public key encryption.

## E.5 Elliptic Curves

Again, this section is not yet complete and interested parties are referred to an excellent overview of the subject provided by *Certicom* at [7] as well as their online interactive visualization of the geometry. Additionally, elliptic curves enable another class of cryptography referred to as *identity-based*.

## E.6 Authentication Fundamentals

The question of authenticating an entity to be who he claims is a little trickier for various reasons. In order to definitively connect a known identity to a particular stream of messages, there must either be pre-placed data that was obtained from a previous authenticated communication (e.g. in person), or there must be some sort of authority that can be trusted to identify correctly.

This problem is solved by a combination of some *public key infrastructure*(PKI) such as *certificates*, and a cryptographic algorithm (a *digital signature*) that allows possessors of your public key to verify that you indeed have the corresponding private key, without divulging information that could lead to the private key's compromise.

## E.7 DSA

The *Digital Signature Algorithm* is the US Government's standard for digital signatures. It was introduced in 1991 by the National Institute of Science and Technology (NIST) for the comprehensive Digital Signature Standard (DSS) adopted in 1993. Extensions to the standard are ongoing.

The algorithm is a little bit too technical / unenlightening to be worth including in detail, so I simply direct the interested reader to [5].

## E.8 X.509 Certificates

Certificates are the current method of choice to identify others over the Internet. The idea is that a server registers with a *CA* (Certificate Authority, e.g. Verisign[27]) for a fee, and users / software knows that if Verisign says that you are Bob, for example, then that is probably correct. By registering, you are given a *certificate*, which is a data structure containing the public key you wish to use, the identifier you registered under, and a digital signature.

It may seem as if this simply moves the authentication problem one level up the hierarchy since now a user is faced with the problem of having to verify Verisign's signature, requiring a public key. But how does the user know that a public key he finds is actually Verisign's, and not supplied by the phony generator of the certificate? This bootstrapping/rendezvous problem is solved by installing Verisign's public key on all users' machines as part of their operating system or browser installation.

In this way, a certificate chain is formed. All users know what Verisign's public key is via pre-placed data, and these certificates transfer the trust of Verisign to trust that a server's public key is actually what it claims. To extend the example, Verisign might authenticate MIT, which then has the power to authenticate any hosted servers. As such, certificate chains can grow arbitrarily long, and make up one type of *PKI*.

The *X.509*[46] modifier refers to the international standard format for certificate structure — what information goes into a certificate and how it gets encoded/decoded — as well as the certification path verification algorithm, which specifies how certificate chains may be validated. X.509 certificates are used behind the scenes by everyone, whether or not they are aware of it, due to its adoption by SSL[23] as the authentication mechanism of choice, and SSL is the *de facto* way to accomplish secure web transactions.

## E.9 ECDSA

With the advent of elliptic curve cryptography, many older algorithms were due for an upgrade. This algorithm, of course, targets DSA in particular. DSA computes over some finite field defined by the prime modulus chosen as a public system parameter. It relies on the standard hardness assumption of discrete logarithms for its security. ECDSA uses exactly the same algorithm, except it operates over the algebraic group defined by a particular elliptic curve, and then gets to make use of the (believed-to-be) more powerful elliptic curve discrete logarithm hardness assumption. The benefits are smaller element sizes for the same level of security, with possibly faster computation times as a corollary of having to compute using fewer bits.



Symmetric	RSA/DH	EC
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Table E.3: Key sizes (bits) for equivalent security.

Research into elliptic curve cryptography is a hot topic currently, making it difficult to accurately specify an equivalence between key sizes under various algorithms. In its proposal to replace DSA with ECDSA, the NSA prepared a table of their best guesses[19], excerpted in Table E.9.

## E.10 Identity Based Signature

Identity Based Signature (IBS) operates on the premise of allowing a user to pick any arbitrary string as their public key. For example, I could decide that my public key is “r0b@mit.edu”, and if everyone used that convention, then what we have accomplished is an intangible, omnipresent, user-friendly PKI. To verify a signature on a message coming from someone claiming to be “r0b@mit.edu”, you would simply feed that string into the verification procedure, and with no external data (except one-time system parameters) it can spit out a verification or repudiation of the identity.

What it accomplishes is a transfer of responsibility from the receiver to obtain a public key through a PKI, to the sender to obtain a private key from some system authority that can be verified by his identity.



## Appendix F

# Cryptography Options

Here we give brief descriptions for the cryptography schemes that we studied. Although only one ends up being incorporated into our project, it is valuable to understand the alternatives, the justification for our decision, and general ideas harvested from this research. For the interested reader, there are extremely useful surveys of key agreement protocols with an eye toward their applicability to group[59] and mobile communication[36] in particular.

First we detail exactly which properties we desire from the supporting cryptography.

- Confidentiality
- Authentication
- Forward secrecy
- Small bandwidth overhead

We considered a large number of schemes before deciding. We judged on the basis of how well their capabilities matched our requirements as well as their costs in time, space, and implementation complexity. The cryptography can be broken into two parts: confidentiality and authentication. The problem of encrypting group communications has been studied at great length, especially in the past few years. Below we describe particular schemes that vied for use in our system, and include brief evaluations for why they were a better or worse fit. For the reader that does not have a background in cryptography, you may find reading Appendix E helpful in getting quickly up to speed. Presented here are the particular solutions built using those techniques.

All schemes are evaluated for space complexity in terms of how much fixed-size data is required for the scheme to function. The communication complexity is a measure of how much extra data is required to be included with each message sent to the group. A pair of values  $O(a)/O(b)$  signifies  $O(a)$  bytes overhead on the initial message to set up some sort of state, and  $O(b)$  while the state is valid.

We consider here only the *stateless* side of the schemes, since there is no contention that a symmetric key encrypting with AES with intelligent group management to maximize the reuse of state while remaining secure is the best option for the *stateful* part.

The computation required by schemes is usually not symmetric. We differentiate between the computation required of the sender and that of the receive using the notational convention  $O(a)/O(b)$ .

- s - number of users in a group
- r - number of revoked users
- n - total number of users

## F.1 DH Key Agreement

To begin, let's take a look at a homegrown, obvious, quick and dirty reference solution that we affectionately refer to in-house as "Brute Force". We began with the intention of comparing any scheme we came up with to this reference implementation. For this protocol, the user carries out a DH key agreement[41], exchanges a symmetric key, and then uses that to encrypt messages (using AES) to any particular host. To broadcast a message to a group, a random symmetric key  $k$  could be generated to encrypt the message with, and then the user could prefix the message with  $k$  encrypted to all intended recipients. Unfortunately, that scheme requires computation and bandwidth linear in the number of recipients.

Space:  $O(n)$   
Communication:  $O(s)$   
Computation:  $O(s)/O(1)$

## F.2 IPsec

The IPsec[13, 58] (IP Security) standard comprises a transport-layer set of protocols for securing communications by encrypting and/or authenticating all IP packets. I quote a summary given in the reference[13].

IPsec consists of a couple of separate protocols, listed below:

- Authentication Header (AH): provides authenticity guarantee for packets, by attaching strong crypto checksum to packets. If you receive a packet with AH and the checksum operation was successful, you can be sure about two things if you and the peer share a secret key, and no other party knows the key:
  1. The packet was originated by the expected peer. The packet was not generated by impersonator.
  2. The packet was not modified in transit.

Unlike other protocols, AH covers the whole packet, from the IP header to the end of the packet.

- Encapsulating Security Payload (ESP): provides confidentiality guarantee for packets, by encrypting packets with encryption algorithms. If you receive a packet with ESP and successfully decrypted it, you can be sure that the packet was not wiretapped in the middle, if you and the peer share a secret key, and no other party knows the key.
- IP payload compression (IPcomp): ESP provides encryption service to the packets. However, encryption tend to give negative impact to compression on the wire (such as ppp compression). IPcomp provides a way to compress packet before encryption by ESP (Of course, you can use IPcomp alone if you wish to).
- Internet Key Exchange (IKE): As noted above, AH and ESP needs shared secret key between peers. For communication between distant location, we need to provide ways to negotiate keys in secrecy. IKE will make it possible.

IPsec competes most directly with SSL: an informative, succinct comparison of the two was written recently[32]. Since this scheme is simply symmetric key encryption made possible by IKE, it does not scale particularly well. In fact, it is based on the same cryptographic

techniques as DH key exchange. Its implementation at the network layer is attractive from the performance perspective, but leads to difficulty in modification, especially since it partially resides in the kernel.

Space:  $O(n)$   
Communication:  $O(s)$   
Computation:  $O(s)/O(1)$

### F.3 Secure Spread

Based on a version of GDH called STR[49], The Secure Spread[43] toolkit provides pluggable key agreement modules, effectively using the group management layer as a platform for testing various characteristics of the cryptography in use. It is designed to be exactly the type of toolkit that we need; unfortunately, the following quote observes a limitation of the cryptographic techniques they used.

Particularly appropriate for relatively small collaborative peer groups...[43]

Due to the contributory nature of the protocol, it does not scale to many recipients/group members, and requires interactive participation in the protocol, thus making it unusable for non-interactive applications such as email. Additionally, STR (and contributory keying schemes in general) have extremely complex protocols: complexity is often the death knell of reliability.

Space:  $O(1)$   
Communication:  $O(s)^*$   
Computation:  $O(s)/O(s)$   
Requires interactive communication rounds

### F.4 Identity Based Encryption

To introduce IBE, I quote a summary[1]:

Identity-based systems allow any party to generate a public key from a known identity value such as an ASCII string. A trusted third party, called the Private Key Generator (PKG), generates the corresponding private keys. To operate, the PKG first publishes a "master" public key, and retains the corresponding master private key. Given the master public key, any party can compute a public key corresponding to the identity I by combining the master public key with the identity value. To obtain a corresponding private key, the party authorized to use the identity I contacts the PKG, which uses the master private key to generate the private key for identity I.

As a result, parties may encrypt messages (or verify signatures) with no prior distribution of keys between individual participants.

Work has been done recently to achieve forward security and key revocation. Even with these nicer properties, one large problem with these schemes is that ciphertext still grows linearly with the number of recipients. To address this, Hierarchical IBE[48, 53] was developed, which organizes users as leaves in a large tree, and allows each of them to know all keys on the path from them to the root. For example, a message may be encrypted not only to a tree leaf, but also intermediate nodes, allowing any of that node's children to

decrypt. This mitigates the recipient scaling problem to a *set cover* of some set of leaves. This is much more favorable, and the tree can be constructed around existing organizational hierarchy, making an efficient set cover extremely likely.

## F.5 Broadcast Encryption

Another attractive scheme is the Boneh-Gentry-Waters elliptic curve broadcast encryption scheme[35]. It offers flexibility to trade between public key size and per-message overhead on the curve  $|PK| \cdot |Hdr| = n$ . Since we design for low-bandwidth, we choose the public key to scale linearly with number of users, giving us a constant size ( $\approx 80$ – $256$  bytes) header that is decryptable by any number of recipients with their own unique keys.

This is a very unique scheme: it is the only stateless solution we have seen with the ability to produce constant-size ciphertexts, which is an extremely attractive property in a bandwidth-limited environment.

Space:	$O(n)$
Communication:	$O(1)$
Computation:	$O(s)/O(s)$

## F.6 Authentication

Users must be able to prove their identity in an offline manner. Having a set of data that is signed by the trusted *certificate authority*(CA) as correct suffices. Such a signature binds the pieces of information together as unforgeable, avoiding man-in-the-middle attacks by guaranteeing that a given public key belongs to a particular identity. In our case, it binds a PKGE id to a users identity, incorporating any amount of information desirable, for example position, group, location, etc.

Authentication is the smaller piece of the puzzle in this category. The current standard remains X.509 certificates[46], and there appears to be great practical motivation to use them. Besides an extensive support network, the Dept of Defense already has a X.509 certificate system in place to authenticate its personnel. X.509 version 3 also support *extensions*, namely user-defined fields that are included in the signed data, so PKGE could potentially define any desired fields and have them fit nicely inside the existing framework.

There is an alternative also based on elliptic curves that is very attractive from a technical perspective. The ECMQV[51] scheme provides for simultaneous key agreement and authentication. The amount of data transferred and computation required is also significantly less than for a DH key agreement plus X.509 certificate exchange, since the key sizes involved are  $\approx 160$  bits instead of  $\approx 1024$  bits. The creators are not at all shy to observe the quality of their scheme, as evidenced by the first paragraph of their abstract:

The MQV protocol of Law, Menezes, Qu, Solinas and Vanstone is possibly the most efficient of all known authenticated Diffie-Hellman protocols that use public-key authentication. In addition to great performance, the protocol has been designed to achieve a remarkable list of security properties. As a result MQV has been widely standardized, and has recently been chosen by the NSA as the key exchange mechanism underlying “the next generation cryptography to protect US government information.”

## Appendix G

# Group Chat

Cryptography is only one piece of this puzzle. It serves as an enabler for the real motivation behind this thesis: efficient communication within dynamic groups. Thus, we also combed the Internet for applications that claim to solve similar problems to the one we were faced with. The most discriminatory criteria we had was that of requiring the software be decentralized in nature and supporting encrypted group chat. It turns out the vast majority of chat applications rely on a central server or support only secure 1-to-1 chat, leaving groups insecure.

At the outset of this thesis, it appeared that we would implement our own group communication solution should an appropriate application not be found, so we looked more heavily at chat applications with an eye towards their efficiency in shuttling messages like good little peer to peer clients and their ability to handle changes in the communication group. We studied up on technologies like IP Multicast[34] and state of art in p2p techniques[24] in order to implement a super-efficient multicast-using p2p secure chat client.

However, the focus on optimizing a chat client for group communication gradually faded out of scope, due in part to time constraints, but also due to the wonderful advantages offered by Gaim integration: a mature, powerful, well-documented(!) extension environment and a huge existing user base. At any rate, in this section I mention a few of the chat applications and their cryptography abilities that were more relevant to our work.

### G.1 AIM

We began with the current king of IM, AOL Instant Messenger (AIM). Specifically, we looked at the open source client GAIM in conjunction with its `gaim-encryption`[9] module. Its security is based on exchange of RSA public keys, followed by a key agreement and subsequent use of a symmetric session key. Unfortunately, it does not support group encryption, the chat rooms are completely unsecured), its crypto is not applicable to groups, and it has no authentication properties. We moved on.

### G.2 Jabber

The existing open-source chat juggernaut recently publicized by Google Talk[12], Jabber[14] is already in use and in favor by our sponsors. It stands out from the crowd since it is the first (widely-deployed) chat protocol that was designed with security(confidentiality *and*

authentication) as a cornerstone. Also, it is completely open source, well-documented, and designed to be extensible. For these reasons, it was our strong preference *a priori*.

In terms of architecture, all clients maintain SSL connections to their hosting server, and servers manage groups as follows: clients that want to be part of a group `group@hostname` send *join* messages to the server, and subsequent messages addressed to `group@hostname` get reflected by the server through the pre-established secure connections.

In general, to send a message to a user, you must address it `user@hostname` and provide some sort of name lookup service that allows your server to look up and/or forward your message to the recipient's server. Jabber lends itself naturally to secure group communications, since it requires  $n$  secure channels (that already exist in this case) rather than  $n^2$ . Of course, the other side of the coin is that the server is a single choke point, and this architecture results in an extremely centralized solution. Unfortunately, the centralized architectural prevented us from proceeding with a Jabber extension.

### G.3 Skype

We next turned to the state-of-the-art in p2p technology, Skype[25]. This is not an open-source solution, however we studied its architecture for guidance in possibly implementing our own p2p system. Since it is a commercial program, its internals are not documented, but at least one very interesting and thorough reverse-engineering job has been published[24]. The paper covers the setup of the p2p network, specifically how nodes accomplish discovery and routing in a scalable manner. There is a central server to authenticate login, but it does not take part in the steady state operation of the program.

This summary of the security features may be found online[8]:

Since a Skype connection may be routed through an intermediate peer, 256-bit AES encryption actively encodes the data stream of each call, or file transfer. Skype uses 1536-bit RSA to secure the pairwise negotiation of an AES symmetric session key over an untrusted channel. The proprietary session establishment protocol is efficient and prevents both man-in-the-middle and replay attacks. The Skype server certifies each user's public key at log in.

Regarding the centralized login server, it does not appear to be necessary at all. It merely serves to "certify each user's public key at log in", a function served much more efficiently by a simple certificate, signed by the central authority and distributed during the setup phase. This p2p technology is very good, but unfortunately not open source or extensible. Searching for extensible p2p led us to our next destination...

### G.4 JXTA

There is a major Java initiative called JXTA[20]. From its home page:

JXTA technology is a set of open protocols that allow any connected device on the network ranging from cell phones and wireless PDAs to PCs and servers to communicate and collaborate in a P2P manner.

It is a p2p Java API available to any JVM-compatible device that allows peers to form groups and communicate securely across public networks. Its web site provides the development tools and a forum for JXTA developers to host their projects for publicity



and collaboration. These projects take the form of modules that can be easily composed into larger projects, and some of them are extremely relevant to our work. For example, JXTA-rm[22] provides reliable UDP-based multicast.

Another project integrates thin wireless messaging clients running on J2ME<sup>1</sup> with enterprise-scale servers. This allows software modules on the embedded device communicate seamlessly with application or database servers.

A third project, myJXTA, is a general collaboration application that allows secure 1:1 chat, insecure groups, and is designed so that plugins may be easily developed to leverage the pre-existing peer/group formations. For example, development is underway for plugins to do real-time graphs, text-to-speech, shared whiteboard, etc. This choice seemed very appropriate for extension into a launch application but ended up being in too early a stage of development and extremely lacking in any sort of documentation (or even particularly reliable operation).

## G.5 Distributed Center

It is worth a small section to observe that centralized schemes are the most pervasive and effective, so we considered ways to adapt them to fit our needs. One option is to decentralize a centralized scheme: implement the central server as a distributed service. For example, Lehane *et. al.* propose an Ad Hoc Key Management Infrastructure[52] that is based around a distributed key distribution center (KDC). Even more generally, efficient distributed hash tables (DHTs) like Chord[62] that only require per-node state to scale as  $\sqrt{n}$  make maintaining a centralized database accessible by millions of users very feasible.

This approach was considered and discarded due to our unwillingness to rely on connectivity to guarantee functionality — the same reason we consider an online keying service inappropriate. If our security library is to be used in the field, it should allow communication between two allies who possibly rendezvous out of range of network connectivity, without prior arrangement.

---

<sup>1</sup>Java 2 Platform, Micro Edition: JVM for use on embedded processors e.g. cell phones



## Appendix H

# PKGE File Formats

This section details the exact binary format of PKGE files. In order to be most expressive when a number of data sizes are dependent on the parameters chosen, I adopt the following notation:

4	Constant # of bytes
*4	# bytes in my example system, but varies based on configuration
4 <i>T</i>	(#bytes) × <i>lookahead</i> , for quantities linear in the number of temporal keys

Another item of note is that the `EVP_PKEY` OpenSSL type may be stored in conjunction with the parameters that define its host elliptic curve, or separately. The actual key we use in the example has space requirements given in Table H.

Other information may be deduced from the rest of the line. For example, there are many entries representing an array of items, the length of the array depending on the number of timesteps a user is allowed to possess at a time (their *lookahead* #). In my example system, the *lookahead* is 365.

328	<code>i2d_PrivateKey()</code>	Private + Public + Params
302	<code>i2d_PUBKEY()</code>	Public + Params
57	<code>i2d_PublicKey()</code>	Public

Table H.1: OpenSSL ECDSA key serialization properties.

Admin System (e.g. my_system.admin)				
Size	Type	Variable	Sample Value	Code Location
1	byte_t	is_admin	1	_pkge_serialize
4	int	strlen(user_subdir)	5	
*5	char*	user_subdir	"users"	
4	unsigned	user_capacity	1000000	
4	unsigned	num_regd_users	74238	
4	unsigned	session_table_size	10	
4	int	cipher_nid		crypto_save
4	int	hash_nid		
4	int	x509_sig_alg_nid		
4	int	x509_curve_nid_		
4	int	pairing_txt_len_	380	curve_save_sanitized
*380	char*	pairing_txt_	"type a..."	
4	int	strlen(subdir)	5	bigpk_save
*5	char*	subdir	"bigpk"	
4	unsigned	num_users	1000000	
4	unsigned	elements_per_block	64	
4	unsigned	num_taus	365	
*24	mpz_t	alpha		
*65	element_t	g		
*65	element_t	g_gamma		
*65T	element_t*	g_gamma_tau[T]		
4	time_t	fs_T	86400	curve_save_sanitized
4	time_t	fs_w	600	
4	time_t	fs_lookahead	2629744	
4	int	fs_num_keys	365	
4T	time_t*	fs_timestamps[T]		
*24	mpz_t	admin_key ( $\gamma$ )		curve_save
*24T	mpz_t*	tau[T]		
*328	EVP_PKEY*	ca_key		certs_save
4	int	strlen(subdir)+1	5	certs_save_common
*5	char*	subdir	"certs"	
4	int	cache_size	10	

Table H.2: File format for administrative system file (e.g. /path/my\_system/my\_system.admin). The format for the user system file (e.g. /path/my\_system/my\_system.pkge) is nearly identical, only omitting the grayed lines and saving a 0 in place of 1 for the first byte.

User file				
Size	Type	Variable	Sample Value	Code Location
4	user_id_t	id	23483	_pkge_user_save
4	user_id_t	id	23483	privkey_save
4 <i>T</i>	time_t*	timestamps[T]		
*65 <i>T</i>	element_t*	g_i_gamma_tau		
*65	element_t	g_i		
*65	element_t	g_i_gamma		
*328	EVP_PKEY*	sign_key		_pkge_user_save

Table H.3: File format for a .usr file.



# Appendix I

## Technical Notes

This section contains technical miscellany and a laundry list of tasks that I would undertake to improve the library and its applications were I to have the opportunity to spend more time. More importantly, it includes explicit notes where the implementation did not live up to the specification developed in chapter 2, as well as implementation notes that did not fit anywhere else.

### I.1 Development Environment Setup

Development in a Unix-like environment is the preferred route, in which case setup proceeds normally. Development on Windows tends to be a bit trickier. If you go this route, install Cygwin and a separate MinGW (ie. don't use the one Cygwin offers). Set your paths to point to MinGW:

```
$ export PATH="/cygdrive/c/mingw/bin:$PATH"
```

You will have to edit all of your Makefiles to use the MinGW include and library paths. One entertaining nuance is that MinGW provides `gcc`, `ar`, `ld`, but not `install`, so due to the differing path formats, remember that any paths passed to `install` should be Cygwin-esque, and any paths passed to compilation commands should be Windows-esque. While downloading and installing the libraries, you will probably come upon items such as `gettimeofday()` which are unimplemented in MinGW. To solve this, simply implement it yourself in the header file by downloading a code snippet found by googling the error message.

Here are some notes on getting the libraries working.

#### PBC

- Remove the `-Wendif-labels` flag from `CFLAGS`. It is incompatible with MinGW `gcc`.
- Implement `gettimeofday()` in `get_time.h` (available online).
- Change the `install` paths in the Makefile so that the includes and libraries get put into, for example, `/cygdrive/c/mingw/include` and `/cygdrive/c/mingw/lib`.

**GMP, OpenSSL** I'm not convinced it's possible to build these in Cygwin using MinGW. I would suggest downloading the prebuilt libraries for Win32.

**Gaim** For Windows, downloading the `wingaim_build_fetcher.sh` script<sup>1</sup> is the best move. It will automatically fetch Gaim's mountain of dependencies, install them, and point Gaim's Makefiles at them. Mac or Linux can simply use the tarball and autotools.

One note: Gaim's preferences are stored in `/.gaim/prefs.xml`. If Gaim starts to crash immediately after initialization, try deleting that file. Also, once all of these libraries are installed, ensure the Makefile has paths to the include files and libs.

## I.2 Conventions

During development I tried to adhere to a few conventions.

**return 0;** Functions almost always return 0 for success, except in the rare case that it would be confusing. For example:

```
if (pkge_have_cert("sys", "Alice"))
    /** Cool we have Alice's cert! **/
```

In that situation, it would be extremely confusing if 0 was success, so an exception was made. In general, return codes are checked using the pattern:

```
if ((r = user_sign(u, msg))
    /** Sign failed, either recover or return r to caller **/
```

**(sub)dir** The relative directory structure of a PKGE directory is stored and expanded on system load to give absolute paths to resource locations, like the user file directory. This avoids any possible ambiguity related to relative pathnames, and allows for the PKGE directory as a whole to be location independent.

By way of example, the system file specifies that `sys->user_subdir` is "my\_user\_subdir". On system load, `sys->user_dir` is filled in by extracting the path to the system from the initialization path passed in, and appending an OS-dependent separator (e.g. `'\'` on Windows, `'/'` on Unices) plus the subdirectory string, forming "path/to/system/my\_user\_subdir" in our example, and stuffing it in the runtime variable `sys->user_dir`.

**pkge\_store** On initialization, the library adds itself to a static storage structure where it can be looked up by alias, in particular the string used to load it. This has the feature that the system can easily be shared between applications by passing in the same alias to load the system. It does a simple string compare to determine alias equality, so for the testing I have been able to add extra slashes e.g. `"/usr/pkge////my_system"` which the shell doesn't mind at all — it simply discards the extra slashes. However, it allows a loophole for loading multiple instances of the same system. Outside of a test environment, this is probably not desirable behavior.

## I.3 New Features

Throughout the course of development for any project of significant size, flashes of inspiration that dazzle the mind with unrealized possibilities, ripe for the implementing. These

---

<sup>1</sup>Linked from the Gaim homepage.[10]



“wouldn’t that be cool” moments are the lifeblood of any successful project, but I have imparted all the vitality I can in the time I’ve had, and leave these for the daring adventurer that comes after me.

- Plugin for Outlook
- Web server for distribution
- Migrate to X.509 V3
- More secure handling of memory: ensure not paged to disk, and “shred” memory after done using it.
- As part of user setup procedure, add a registry key to make all PKGE-using apps config-free.
- `pkge_unseal` should take a user object and sealed message, and (if successful) should unseal the message in place, and return a pointer to a group representing the group that the message was sealed to. The current interface makes no sense.
- Develop an application that support real-life IP multicast, as that would see huge performance benefits in the scenarios we evaluated.
- NSS - Mozilla Network Security Services is an blossoming open-source solution to managing certificates (and so much more). PKGE certificate management is amateurish, as that was not the focus. Integrating with NSS would be beneficial because it would allow certificates to be easily shared and it provides much more efficient certificate lookup/handling mechanisms.
- The capability to add users that already have X.509 certificates. 90% of the code is in place, the lack of a clear standard for how to obtain the users’ private signing keys stymied the effort.
- Cryptographically speaking, it is possible for someone not in the system to send encrypted messages, since it does not require a personal key. Possibly allow this mode of operation for the case in which a system is available but the user file is not, e.g. his device breaks so he picks up his buddy’s and does not have to use plaintext. These messages would not be signed or authenticated, of course, but still confidential.
- PINs: 4 bytes of Grade A numeric variable allows up to a 9 digit PIN. Extending the PIN to support longer PINs or **alphanumeric** strings may be worthwhile. Additionally, an idea that never materialized was requiring the user to reauthenticate himself to the library every so often. In other words, the user would call `pkge_unseal` and get an error back: `PKGE_ERR_REAUTH` or some such mumbo jumbo. Make a successful call to `pkge_user_load` and continue with normal operation. Implementation would be simple and clean.
- Reconfigurator: there is no technical reason an already-generated system could not be reconfigured, with respect to the algorithms used or the forward secrecy parameters, directory structure, or a million other miscellaneous details.
- Signature algorithms other than ECDSA are supported by OpenSSL but are unimplemented. Support for RSA, DSA, and DH signature algorithms would fit neatly in `crypto.c:crypto_gen_key`.
- Support asymmetric elliptic curves — we didn’t think through it well enough the first time. An asymmetric curve requires 2x the public key, but some of them are  $< \frac{1}{2}$  the size, and they **do not require 2x bandwidth** since you still only send 2 elements.

Bandwidth savings may be had, but in truth it may not be significant considering the session protocols.

- Greater security can be provided by encrypting all user files with a random PIN (on the admin system) and storing all of the random PINs in a file that is encrypted with an administrative PIN.

## I.4 Performance Optimization

This section chronicles observations of sub-optimality that have not been addressed due to opportunity cost/lack of time. I do not expect any of these by itself to make a huge impact, but depending on a particular performance goal, a selection of these may do the trick.

- Compress message input if it's text.
- Looking at 'top', the RSIZE for ./testpkge is 1.66 MB, the VSIZE is 31 MB. I believe this means that due to linking with big libraries like GMP and OpenSSL, we incur a large shared library loading fee if these aren't already loaded when PKGE is requested. We use a tiny tiny fraction of OpenSSL functionality, so perhaps we could cut down on the footprint?
- Add an option to not include the group in the sealed message, but rather rely on the application to tell PKGE what the group is. This may avoid duplicated work, and is one way of solving the "we have constant size ciphertext with linear size headers" embarrassment.
- To encrypt, the C1 piece may be precalculated and used for all time ( $e(g_n, g_1)^t$ ). The  $t$  is random per message, but the bilinear mapping isn't.
- We chose CBC mode for AES encryption by default—do some research to validate the decision or possibly discover a different mode would be better.
- Currently sessions contain a `force_header` field that is never actually used. It supports the passive recovery of errors, and we implement only the active method. The idea was that if an unrecognized session id was received, the `force_header` flag would be turned on, and the next message sent from that host would contain a full header, and by the normal session agreement protocol, everyone else would switch to use that one.
- Currently an unrecognized session id causes the out-of-the-loop user to send a help message to everyone in the group. This causes everyone in the group to respond with a header. Ideally, only one would respond with a header.
- The `sessions` protocol may be further optimized by having the library poke the user to send a message when first starting a new session. This would accomplish fast switching to the optimized header, preserving a lot of bandwidth in the case where a user is present and quietly participating in the group. This could be implemented by using the same channels that session errors traverse. For example, it could use the registered sending function, if available, or return `PKGE_FLAG_SESSION_VERIFY` from `pkge_unseal`.
- When adding users, keep track of users in the same `organizationalUnitName` (an X.509 field). Assign each organizational unit (e.g. Platoon 14) a "group id" that can be used with the API (e.g. `pkge_group_add(my_group, "Infantry 11")`). This identifier can get transformed to a PKGE id and decoded by recipients as the appropriate group of users. Groups can be versioned to support personnel changes. This is

a big win for bandwidth (can represent many recipients with a single id), processing time (easy to store the associated encryption product with a group), and security (the actual recipients are obfuscated by a level of indirection).

- Not save all the EC parameters with each user key. The EC parameter file is already generated (`ec.params`), but never used. Using that, we could drop the size of X.509 private/public keys to  $\approx 60$  bytes down from  $\approx 300$ .
- There are a number of places that caches would speed up operations. Dealing with certificates and doing name $\leftrightarrow$ id lookups would benefit. Caching elements of the public key also makes sense.
- Index and cache for name $\rightarrow$ id lookups instead of loading certificate from disk
- When loading a private key, all  $g_i^{\gamma T_t}$  are loaded into memory. This is wasteful. Loading only the current timestep (or other timesteps on demand) makes more sense.
- Compile separate libraries for client vs administrative functions = $_j$  less code in each, easier to maintain, extend, and test.
- Element serialization: PKGE stores elements prefixed by their size in bytes and a tag indicating that it's an element. All elements are the same size, and in the public key there is nothing but elements, so 5 bytes are wasted per element, especially in the public key. This should be easy to resolve.
- The GAIM plugin should require PINs/load profiles on Account Signon rather than checking on each new conversation (ie. be account-oriented rather than conversation-oriented)
- PKGE uses `ERR_load_crypto_strings()` and `OpenSSL_add_all_algorithms()`. The ERR strings require significant memory and should not be loaded in a production environment. Also, we use relatively little of OpenSSL's capabilities, so using `EVP_add_cipher()` and `EVP_add_digest()` should be done instead, for resource conservation.
- A non-admin system should not contain the code to load an unencrypted user file.
- Currently there are a number of reallocs sprinkled throughout the code. They make it easy to write correct code, but are presumably not efficient. Consider other memory management techniques.
- An alternative scheme to storing X.509 certificates: keep an authenticated table mapping ids to names. Require an X.509 certificate to add to the table. With PKGE name length restriction, this means that you could get away with 20 bytes per user, making an index with a complete list of everyone's name to id mapping realizable. It would allow fast lookups both ways, and it would be easy to trade "address books" since newer indexes are supersets of older ones (as users are added).
- Make PKGE name a function of their email address, for example, to avoid having to look through certificates to do the lookup.
- Currently the message that users pass in to unseal is duplicated in case it has to be restored due to any problems. This is problematic for large buffers—it should not be necessary.

## I.5 Cleanup

Ah, other people cleaning up my mess :o) This section is devoted to imperfections in the implementations, a number of which really do need to be dealt with before PKGE should

be pronounced ready for a production environment.

- PKGE is not ready to be used concurrently, safely: there are few static structures, but locks must be put in place
- GTK2+ preference pane was added to the GAIM plugin late in the gaim. It needs to be double checked for correct resource management (ie. no memory leaks)
- The session protocols were intentionally kept separate to make it easy to modify them independently. It turns out there are only a couple lines difference between sessions and optimistic sessions, and it is badly in need of a refactoring.
- The `set_error_msg` function pushes a probe # on to the message so that the record in the hash table can be found unambiguously (in case there are multiple error messages for the same group) by `get_error_msg`. This setup requires either a user call to `pkge_get_error_msg` or a send by the user's registered sender, but not both
- Base64 encoding is possibly not an appropriate thing to include in this library, as any application that required it would doubtless implement its own. Gaim provides its own, for example.
- C libraries are used extensively throughout the code. `memcpy_s` and `strcpy_s` are supposedly much better form.
- The handling mechanism in case there is a collision of session ids is not implemented. If a new session is created and put into the hash table, and it is discovered that the private id collides, it should be assigned a new since no one else has seen it yet, anyway. However, simply keeping a probe number associated with the session works also, and this is what needs to be done in the case that a colliding private id is received from another user.
- If an error message is received, the private session information is sent in the clear. This is not terrible, since an error message must have a valid signature, and the only thing it gives away is the private id mapping to a recipient set, but it should still be fixed.
- Unit tests for modules would be good.

# Appendix J

## Listings

### J.1 default.cfg

```
# PBC Elliptic curve parameters
#   Requires a symmetric pairing
# Note: These MUST come first.
#       (nothing but blank lines or comments before)

type a
q 878071079966331252243778198475404981580688319941420821102865339926647563...
h 120160122648911460793888213667405342048029544012513118229196151310472072...
r 730750818665451621361119245571504901405976559617
exp2 159
exp1 107
sign1 1
sign0 1
end pairing

# General system parameters (N = capacity)
N 10000000
user_subdir users
cert_subdir certs
pk_subdir  bigpk

# Symmetric encryption settings
cipher AES-256-CBC
hash  SHA256

# X509 key settings (sign/verify keys)
x509_sig_algorithm  ecdsa-with-SHA1
# SN=secp224r1 , nid=713 , comment=NIST/SECG curve over a 224 bit prime field
x509_elliptic_curve 713
x509_key_type        id-ecPublicKey
```

```

# default.cfg continued...

# BigPK settings
elements_per_block 64
g                    RANDOM

# Forward security -- all values in seconds
# 1 day = 86400, 1 month = 2629744, 1 year = 31556926

# Timestep: amount of time one tau is good for.
# Window: how long you can accept messages from the previous timestep
# Lookahead: how far ahead of time are taus generated

timestep    86400
window      3600
lookahead   2629744

# Cache size for session keys (in log_2 format, e.g. 8 => 2^8 = 256)
session_table_size 10

```

## J.2 OpenSSL Cipher Listing

This is a partial list of all ciphers supported under OpenSSL v0.9.8. All algorithms have a fixed key length unless otherwise stated.

```

EVP_enc_null()
    Null cipher: does nothing.

EVP_des_cbc(void), EVP_des_ecb(void), EVP_des_cfb(void),
EVP_des_ofb(void)
    DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede_cbc(void), EVP_des_ede(), EVP_des_ede_ofb(void),
EVP_des_ede_cfb(void)
    Two key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_des_ede3_cbc(void), EVP_des_ede3(), EVP_des_ede3_ofb(void),
EVP_des_ede3_cfb(void)
    Three key triple DES in CBC, ECB, CFB and OFB modes respectively.

EVP_desx_cbc(void)
    DESX algorithm in CBC mode.

EVP_rc4(void)
    RC4 stream cipher. This is a variable key length cipher with
    default key length 128 bits.

```

`EVP_rc4_40(void)`

RC4 stream cipher with 40 bit key length. This is obsolete and new code should use `EVP_rc4()` and the `EVP_CIPHER_CTX_set_key_length()` function.

`EVP_idea_cbc()` `EVP_idea_ecb(void)`, `EVP_idea_cfb(void)`,  
`EVP_idea_ofb(void)`, `EVP_idea_cbc(void)`

IDEA encryption algorithm in CBC, ECB, CFB and OFB modes respectively.

`EVP_rc2_cbc(void)`, `EVP_rc2_ecb(void)`, `EVP_rc2_cfb(void)`,  
`EVP_rc2_ofb(void)`

RC2 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional parameter called "effective key bits" or "effective key length". By default both are set to 128 bits.

`EVP_rc2_40_cbc(void)`, `EVP_rc2_64_cbc(void)`

RC2 algorithm in CBC mode with a default key length and effective key length of 40 and 64 bits. These are obsolete and new code should use `EVP_rc2_cbc()`, `EVP_CIPHER_CTX_set_key_length()` and `EVP_CIPHER_CTX_ctrl()` to set the key length and effective key length.

`EVP_bf_cbc(void)`, `EVP_bf_ecb(void)`, `EVP_bf_cfb(void)`, `EVP_bf_ofb(void)`;

Blowfish encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

`EVP_cast5_cbc(void)`, `EVP_cast5_ecb(void)`, `EVP_cast5_cfb(void)`,  
`EVP_cast5_ofb(void)`

CAST encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher.

`EVP_rc5_32_12_16_cbc(void)`, `EVP_rc5_32_12_16_ecb(void)`,

`EVP_rc5_32_12_16_cfb(void)`, `EVP_rc5_32_12_16_ofb(void)`

RC5 encryption algorithm in CBC, ECB, CFB and OFB modes respectively. This is a variable key length cipher with an additional "number of rounds" parameter. By default the key length is set to 128 bits and 12 rounds.

`EVP_aes_256_ecb(void)`, `EVP_aes_256_cbc(void)`, `EVP_aes_256_cfb1(void)`,

`EVP_aes_256_cfb8(void)`, `EVP_aes_256_cfb128(void)`, `EVP_aes_256_ofb(void)`

AES encryption modes. There are also 128 and 192 bit modes.

### J.3 OpenSSL Curves

Table J.1: Comparison of elliptic curves available for the OpenSSL v0.9.8 ECDSA\_with\_sha1() signature algorithm. Times in milliseconds, sizes in bytes.

Curves description:	NID	key bits	Key/Sig		Disk Size	Time for 500 b		Time for 50 kb	
			pub	priv	sig	sign	verify	sign	verify
0: SECG/WTLS curve over a 112 bit prime field	704	112	29	201	36	0.5	0.1	0.91	0.3
1: SECG curve over a 112 bit prime field	705	110	29	200	36	0.5	0	0.8	0.4
2: SECG curve over a 128 bit prime field	706	128	33	219	40	0.5	0	0.9	0.3
3: SECG curve over a 128 bit prime field	707	126	33	218	40	0.5	0	0.8	0.4
4: SECG curve over a 160 bit prime field	708	161	41	192	50	2.91	2.9	3.3	3.41
5: SECG curve over a 160 bit prime field	709	161	41	255	50	2.81	2.7	3.1	3.11
6: SECG/WTLS curve over a 160 bit prime field	710	161	41	255	50	2.8	2.71	3.1	3.2
7: SECG curve over a 192 bit prime field	711	192	49	222	56	3.8	4.01	4.2	4.31
8: SECG curve over a 224 bit prime field	712	225	57	250	66	4.9	5.21	6.21	5.8
9: NIST/SECG curve over a 224 bit prime field	713	224	57	328	64	3.41	3.5	3.61	3.8
10: SECG curve over a 256 bit prime field	714	256	65	279	72	5.11	5.51	5.51	5.71
11: NIST/SECG curve over a 384 bit prime field	715	384	97	510	104	9.11	10.41	9.41	11.22
12: NIST/SECG curve over a 521 bit prime field	716	521	133	673	141	18.23	21.63	18.73	22.83
13: NIST/X9.62/SECG curve over a 192 bit prime field	409	192	49	292	56	2.6	2.71	3.1	3.01
14: X9.62 curve over a 192 bit prime field	410	192	49	292	56	2.7	2.6	3	3.2
15: X9.62 curve over a 192 bit prime field	411	192	49	292	56	2.71	2.6	3.11	3
16: X9.62 curve over a 239 bit prime field	412	239	61	344	68	4.21	4.3	4.51	4.81
17: X9.62 curve over a 239 bit prime field	413	239	61	344	68	4.11	4.31	4.6	4.81
18: X9.62 curve over a 239 bit prime field	414	239	61	343	68	4.2	4.51	4.5	4.81
19: X9.62/SECG curve over a 256 bit prime field	415	256	65	364	72	4.2	4.51	4.61	5.1
20: SECG curve over a 113 bit binary field	717	113	31	207	38	0.5	0	0.91	0.3
21: SECG curve over a 113 bit binary field	718	113	31	207	38	0.5	0	0.8	0.4
22: SECG/WTLS curve over a 131 bit binary field	719	131	35	235	42	0.5	0	0.9	0.3
23: SECG curve over a 131 bit binary field	720	131	35	235	42	0.5	0	0.9	0.3
24: NIST/SECG/WTLS curve over a 163 bit binary field	721	163	43	202	50	3.1	5.21	3.6	5.51
25: SECG curve over a 163 bit binary field	722	162	43	244	50	3.3	5.61	3.61	6
26: NIST/SECG curve over a 163 bit binary field	723	163	43	223	50	3.41	5.51	3.6	6.01
27: SECG curve over a 193 bit binary field	724	193	51	289	58	4.4	7.61	4.6	8.22
28: SECG curve over a 193 bit binary field	725	193	51	290	58	4.21	7.51	4.7	7.81
29: NIST/SECG/WTLS curve over a 233 bit binary field	726	232	61	249	66	5.51	10.11	5.81	10.52
30: NIST/SECG/WTLS curve over a 233 bit binary field	727	233	61	301	68	6.11	11.21	6.41	11.61
31: SECG curve over a 239 bit binary field	728	238	61	251	68	5.71	10.42	6.01	10.72
32: NIST/SECG curve over a 283 bit binary field	729	281	73	295	80	9.81	18.53	10.11	18.73
33: NIST/SECG curve over a 283 bit binary field	730	282	73	352	80	11.01	20.73	11.21	21.23
34: NIST/SECG curve over a 409 bit binary field	731	407	105	381	110	22.44	43.86	22.94	44.66
35: NIST/SECG curve over a 409 bit binary field	732	409	105	457	112	25.84	50.47	26.14	50.47
36: NIST/SECG curve over a 571 bit binary field	733	570	145	516	153	52.18	103.24	52.88	104.85
37: NIST/SECG curve over a 571 bit binary field	734	570	145	610	153	60.28	119.87	60.89	119.77
38: X9.62 curve over a 163 bit binary field	684	163	43	267	50	3.31	5.51	3.71	5.8
39: X9.62 curve over a 163 bit binary field	685	162	43	268	50	3.5	5.51	3.71	5.91
40: X9.62 curve over a 163 bit binary field	686	162	43	268	50	3.4	5.71	3.7	5.91
41: X9.62 curve over a 176 bit binary field	687	161	45	251	50	3.21	5.5	3.6	5.71
42: X9.62 curve over a 191 bit binary field	688	191	49	284	56	3.71	6.51	3.81	6.41
43: X9.62 curve over a 191 bit binary field	689	190	49	284	56	3.8	5.91	3.9	6.31
44: X9.62 curve over a 191 bit binary field	690	189	49	284	56	3.61	6	4.01	6.41
45: X9.62 curve over a 208 bit binary field	693	193	53	258	58	5.61	10.02	5.9	9.92
46: X9.62 curve over a 239 bit binary field	694	238	61	332	68	6.1	11.42	6.61	11.61
47: X9.62 curve over a 239 bit binary field	695	237	61	332	68	6.21	11.92	6.51	11.72
48: X9.62 curve over a 239 bit binary field	696	236	61	332	68	6.41	11.71	6.61	11.62
49: X9.62 curve over a 272 bit binary field	699	257	69	348	74	10.12	18.92	10.11	18.93
50: X9.62 curve over a 304 bit binary field	700	289	77	380	82	11.42	22.33	11.61	21.63
51: X9.62 curve over a 359 bit binary field	701	353	91	454	98	17.03	33.44	17.62	34.05
52: X9.62 curve over a 368 bit binary field	702	353	93	446	98	18.62	36.25	18.73	35.95
53: X9.62 curve over a 431 bit binary field	703	418	109	501	114	27.44	52.78	26.94	52.37
54: WTLS curve over a 113 bit binary field	735	112	31	156	36	0.5	0.1	0.9	0.3
55: NIST/SECG/WTLS curve over a 163 bit binary field	736	163	43	202	50	3	5.31	3.51	5.61
56: SECG curve over a 113 bit binary field	737	113	31	207	38	0.6	0	0.9	0.3
57: X9.62 curve over a 163 bit binary field	738	163	43	267	50	3.3	5.41	3.7	5.91
58: SECG/WTLS curve over a 112 bit prime field	739	112	29	201	36	0.6	0	0.9	0.3
59: SECG/WTLS curve over a 160 bit prime field	740	161	41	255	50	2.81	2.8	3.1	3.01



Table J.1: (continued)

Curves description:	NID	key bits	Key/Sig pub	Sig priv	Disk Size sig	Time for 500 b sign	Time for 500 b verify	Time for 50 kb sign	Time for 50 kb verify
60: WTLS curve over a 112 bit prime field	741	113	29	150	38	0.6	0	0.8	0.4
61: WTLS curve over a 160 bit prime field	742	161	41	192	50	3.1	2.71	3.2	3.31
62: NIST/SECG/WTLS curve over a 233 bit binary field	743	232	61	249	66	5.71	10.11	5.91	10.51
63: NIST/SECG/WTLS curve over a 233 bit binary field	744	233	61	301	68	6.21	11.11	6.41	11.42
64: WTLS curvs over a 224 bit prime field	745	224	57	305	64	3.41	3.5	3.7	3.91
65:00:00	749	154	41	189	48	2.61	0	2.9	0.3
IPSec/IKE/Oakley curve #3 over a 155 bit binary field. Not suitable for ECDSA.									
Questionable extension field!									
66:00:00	750	184	49	212	54	3.1	0	3.41	5.4
IPSec/IKE/Oakley curve #4 over a 185 bit binary field. Not suitable for ECDSA.									
Questionable extension field!									



# Bibliography

- [1] [http://en.wikipedia.org/wiki/ID-based\\_cryptography](http://en.wikipedia.org/wiki/ID-based_cryptography).
- [2] Aladdin. <http://www.imdb.com/title/tt0103639/quotes>.
- [3] Bob Jenkins' Web Site. <http://www.burtleburtle.net/bob/>.
- [4] Cygwin homepage. <http://www.cygwin.com>.
- [5] Digital Signature Algorithm. <http://www.itl.nist.gov/fipspubs/fip186.htm>.
- [6] "Editor MACroS" aka Emacs: The extensible, customizable, self-documenting real-time display editor. <http://www.gnu.org/software/emacs/>.
- [7] Elliptic Curve tutorial. [http://www.certicom.com/index.php?action=ecc\\_tutorial\\_home](http://www.certicom.com/index.php?action=ecc_tutorial_home).
- [8] Encyclopedia: Skype. <http://experts.about.com/e/s/sk/Skype.htm>.
- [9] gaim-encryption plugin. <http://gaim-encryption.sourceforge.net/>.
- [10] Gaim Project. <http://gaim.sourceforge.net>.
- [11] GNU Multiple Precision Arithmetic Library. <http://www.swox.com/gmp/>.
- [12] Google Talk: IM/Voice chat using the Jabber protocol. <http://www.google.com/talk/>.
- [13] IPsec: Transport-level Security for Internet Protocol. <http://www.netbsd.org/Documentation/network/ipsec/>.
- [14] Jabber home. <http://www.jabber.org>.
- [15] MemCheckDeluxe: memory tracking program. <http://prj.softpixel.com/mcd/>.
- [16] Network Centric Warfare: Dept of Defense Report to Congress. <http://www.defenselink.mil/nii/NCW/>.
- [17] Network Security Services, Mozilla Org. <http://www.mozilla.org/security/nss>.
- [18] NIST Bogosort Specification. <http://www.nist.gov/dads/HTML/bogosort.html>.
- [19] NSA on Elliptic Curve Cryptography. [http://www.nsa.gov/ia/industry/crypto\\_elliptic\\_curve.cfm?MenuID=10.2.7](http://www.nsa.gov/ia/industry/crypto_elliptic_curve.cfm?MenuID=10.2.7).
- [20] Project JXTA. <http://www.jxta.org>.
- [21] RedHat AutoBook: GNU Autotools. <http://sources.redhat.com/autobook/>.
- [22] Reliable Multicast for JXTA. <http://jxta-rm.jxta.org/>.
- [23] Secure Socket Layer. <http://wp.netscape.com/eng/ssl3/>.
- [24] SKYPE: A Security Evaluation. <http://www.skype.com/security/files/2005-031%20security%20evaluation.pdf>.

- [25] Skype Home. <http://www.skype.com>.
- [26] The OpenSSL Project. <http://www.openssl.org>.
- [27] Verisign Inc. <https://www.verisign.com/>.
- [28] Wikipedia: Common Access Card. [http://en.wikipedia.org/wiki/Common\\_Access\\_Card](http://en.wikipedia.org/wiki/Common_Access_Card).
- [29] World Wide Consortium for the Grid. <http://www.w2cog.org/>.
- [30] Fips 186-2: The digital signature standard, 10 2001. <http://csrc.nist.gov/publications/fips/>.
- [31] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. *Digital SRC Research Report*, (125).
- [32] A. Alshamsi and T. Saito. A technical comparison of IPsec and SSL. *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on*, 2, 2005.
- [33] R. Anderson and R. Needham. Robustness Principles for Public Key Protocols. *Lecture Notes in Computer Science*, (963):236–247, 1995.
- [34] T. Ballardie, P. Francis, and J. Crowcroft. Core Based Trees (CBT): An Architecture for Scalable Inter-Domain Multicast Routing. *Proc. ACM SIGCOMM*, 93:85–95, 1993.
- [35] Dan Boneh, Craig Gentry, and Brent Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys, 2005.
- [36] Colin Boyd and Anish Mathuria. Key establishment protocols for secure mobile communications: A selective survey. In *ACISP '98: Proceedings of the Third Australasian Conference on Information Security and Privacy*, pages 344–355, London, UK, 1998. Springer-Verlag.
- [37] Mel Brooks, Thomas Meehan, and Ronny Graham. Spaceballs, 1987. <http://www.imdb.com/title/tt0094012/>.
- [38] M. Burrows, M. Abadi, and R. Needham. A Logic of Authentication. *Proc. R. Soc. Lond.*, (426):233–271, 1989.
- [39] Don Davis. Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 65–78, Berkeley, CA, USA, 2001. USENIX Association.
- [40] W. Diffie. The first ten years of public-key cryptography. pages 510–527, 1988.
- [41] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.
- [42] Tommi Elo. Lessons learned on implementing ECDSA on a Java smart card. In *Proceedings of NordSec2000*, 2000.
- [43] C. Nita-Rotaru et. al. Secure group communication using robust contributory key agreement. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 15(5):468–480, 05 2004.
- [44] T. Gibson. An Architecture for Flexible Multi-Security Domain Networks.
- [45] Paul Hsieh. Hash functions, 2004. <http://www.azillionmonkeys.com/qed/hash.html>.
- [46] Internet Engineering Task Force (IETF). Rfc 2585: Internet x.509 public key infrastructure, 05 1999. <http://www.ietf.org/rfc/rfc2585.txt>.

- [47] Internet Engineering Task Force (IETF). RFC 2898: PKCS 5: Password-Based Cryptography Specification, 09 2000. <http://www.ietf.org/rfc/rfc2585.txt>.
- [48] J. Katz. A forward-secure public-key encryption scheme. Technical report, Cryptology ePrint Archive: Report 2002/060, May 2002, <http://eprint.iacr.org/2002/060>.
- [49] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Communication-efficient group key agreement. In *Sec '01: Proceedings of the 16th international conference on Information security: Trusted information*, pages 229–244, 2001.
- [50] Linda D. Kozaryn. DoD Issues Time-saving Common Access Cards. 10 2000.
- [51] H. Krawczyk. HMQV: A High-Performance Secure Diffie-Hellman Protocol. *Full version of [25]*, available at <http://eprint.iacr.org/2005/176>.
- [52] Brian Lehane, Linda Dolye, and Donal O'Mahony. Ad hoc key management infrastructure. In *ITCC (2)*, pages 540–545, 2005.
- [53] H.W. Lim and K.G. Paterson. Identity-Based Cryptography for Grid Security. *e-Science and Grid Computing, First International Conference on*, pages 395–404, 2005.
- [54] Ben Lynn. Pairing-Based Cryptography Library. <http://rooster.stanford.edu/~ben/abc/>.
- [55] B.P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.
- [56] NIST. FIPS PUB 197: The Advanced Encryption Standard. <http://www.csrc.nist.gov/publications/fips/>.
- [57] F. OpenSSL and O. Module. OpenSSL FIPS 140 2 Security Policy.
- [58] R. Perlman and C. Kaufman. Analysis of the IPsec key exchange standard. *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001. WET ICE 2001. Proceedings. Tenth IEEE International Workshops on*, pages 150–156, 2001.
- [59] Sandro Rafaeli and David Hutchison. A survey of key management for secure group communication. *ACM Comput. Surv.*, 35(3):309–329, 2003.
- [60] Anthony J. Simon. Overview of the Department of Defense Net-Centric Data Strategy. DoD CIO/Information Management Directorate.
- [61] Michael Steiner, Gene Tsudik, and Michael Waidner. Diffie-hellman key distribution extended to group communication. In *CCS '96: Proceedings of the 3rd ACM conference on Computer and communications security*, pages 31–37, New York, NY, USA, 1996. ACM Press.
- [62] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.
- [63] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. 2005.
- [64] L.C. Williams. A Discussion of the Importance of Key Length in Symmetric and Asymmetric Cryptography, 2001.
- [65] Erik De Win, Serge Mister, Bart Preneel, and Michael Wiener. On the performance of signature schemes based on elliptic curves. *Lecture Notes in Computer Science*, 1423:252–??, 1998.

- [66] Adam D. Woodbury, Daniel V. Bailey, and Christof Paar. Elliptic curve cryptography on smart cards without coprocessors. In *Proceedings of the fourth working conference on smart card research and advanced applications*, pages 71–92, Norwell, MA, USA, 2001. Kluwer Academic Publishers.