An Identity and Certificate Manager

by

Brian C. Wu

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

Author ....
Department of Electrical Engineering and Computer Science
September 4, 2007

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

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

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

This research was sponsored by the United States Air Force under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are not necessarily endorced by the US Government.

An Identity and Certificate Manager

by

Brian C. Wu

Submitted to the Department of Electrical Engineering and Computer Science
on September 4, 2007, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We have designed and implemented a software library, called *Identity and Certificate Manager* (ICM), for managing, using, and exchanging application-level usernames, users' digital certificates, and cryptographic username-certificate bindings. ICM can be used in a variety of personal communication applications, such as chat, email, VoIP telephony, and web browsing.

As part of ICM, we designed and implemented a communication-efficient protocol, called *Identity and Certificate Exchange* (ICE), for exchanging certificates, usernames, and bindings within applications. The protocol avoids sending redundant information by remembering what information has been sent to whom; this feature is critical in low-bandwidth networks. The protocol also implements a robust fail-over mechanism for handling out-of-sync situations.

To illustrate the benefits of ICM and ICE, we used ICM in a plugin for a popular chat-client, called *Pidgin.* The plugin allows users to engage in authenticated communication over any of the chat protocols supported by Pidgin, such as Jabber and Oscar (AIM). The plugin relies on ICE to provide assurances about users' identities and to efficiently disseminate users' certificates.

Thesis Supervisor: Roger Khazan
Title: Research Scientist, MIT Lincoln Laboratory

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

Acknowledgments

Many thanks to Roger Khazan for his guidance throughout the life of this project. His creativity, especially during our design discussions, was truly invaluable. Joe Cooley was also very helpful, although his assistance tended to come more from the implementation side. Without his help, I would have spent countless more hours hunting down obscure bugs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

In this thesis project, we have designed and implemented a software library, called *Identity and Certificate Manager* (ICM), for managing, using, and exchanging application-level usernames, users' digital certificates, and cryptographic username-certificate bindings. The three main components of the library are:

- A database back-end for storing and accessing digital certificates and cryptographic keys, application-username-certificate bindings, and other helpful meta-data information about these items;

- A set of cryptographic functions for verifying digital certificates, producing digital signatures for applications' messages, and verifying these signatures using the digital certificates; and

- A communication-efficient protocol, called *Identity and Certificate Exchange* (ICE), for exchanging certificates, usernames, and bindings within applications. The protocol avoids sending redundant information by remembering what information has been sent to whom; this feature is critical in low-bandwidth networks. The protocol also has a robust fail-over mechanism for handling out-of-sync situations.

The term *identity*, in the context of our project, means the application's notion of users' identities, or usernames.

ICM can be used in a variety of personal communication applications, such as chat, email, VoIP telephony, and web browsing. To illustrate the benefits of ICM, we used it in a plug-in for a popular chat-client, called Pidgin [19]. The plug-in allows users to have authenticated communication in Pidgin over any of the chat protocols supported by Pidgin, such as Jabber [11] and AIM [1]. The plug-in uses the ICE protocol to provide assurances about users' identities and to efficiently disseminate users' certificates. The plug-in can also be used as a base for an authenticated Diffie-Hellman key exchange, in order to enable confidential chat.

ICM is implemented using the C++ programming language and three supporting libraries: SQLite [23], OpenSSL [17], and ACE (ADAPTIVE Communication Environment) [25]. We have also implemented a set of C language wrappers around the C++ API provided by ICM; these wrappers can be used for using ICM from C-language applications, like Pidgin. ICM is built to be cross-platform (*nix, Windows-cygwin [3]).

1.2 Motivation

The Internet was originally developed as a tool for academics only. Security was not on the mind of the earliest users, and people assumed that everybody was who they said they were.

Three decades later, the Internet is a vastly different "place." From journalistic bloggers to malicious hackers, from script kiddies to identity thieves, from campaigning politicians to sexual predators, the Internet's user population has changed dramatically. Now, or perhaps several years ago, it is becoming more and more important to know who people are. The act of checking to make certain that a message is from a particular person, and has not been modified in transit, is known as message authentication.

One solution to the problem of authentication is called Public Key Infrastructure

(PKI). A Public Key Infrastructure is based on a hierarchical trust model. The basic concept is that if Alice trusts Bob to trust other people, and Bob trusts Charles, then Alice can trust Charles. See Appendix A for more on PKI.

These days, Internet users have tens or even hundreds of accounts at different websites, along with similar numbers of passwords (or possibly using the same password for all their accounts). Communication has also gotten more complicated, with ever more protocols gaining popularity. For example, IRC (Internet Relay Chat) [10] used to be the popular protocol to use. Since then, AIM (AOL Instant Messenger) [1] has gained widespread usage, starting around when AOL was a popular ISP. However, in the past five or so years, Yahoo! Messenger [31], MSN Messenger [29], and more recently XMPP [30](Jabber [11]) protocols have become popular. Users often have accounts with services implementing these protocols. Our project hopes to help a user manage their account-specific information and corresponding contacts' information.

We want to *bind* usernames from these communication protocols to certificates from a PKI. By creating these cryptographic bindings, we add some level of certainty concerning the user's identity to the communication protocol. Teh bindings and certificates can then be used as a baseis for authenticated communication.

1.3 Related Work

Certificate managers have been designed and realized in the past. For example, there is Mozilla's Network Security Services (NSS) [14] which is used by Mozilla's popular browser Firefox [7]. A plugin for Pidgin [19], an open-source instant messaging client, also has a certificate manager (CertMgr) to enable SSL encrypted conversations. Two other plugins for Pidgin (Pidgin-Encryption and OTR) involve some use of cryptography. Certificate exchange has also been pioneered; a colleague, Joe Cooley, wrote a simple application to perform certificate exchange.

1.3.1 NSS

Mozilla's Network Security Services (NSS) [14] provides cryptography libraries for applications such as Firefox [7], Thunderbird [26], AOL Instant Messenger [1], etc. on many different platforms. NSS has their own certificate database, along with a key database. They have a few commandline tools (certutil, crlutil) to give users a way to manage these databases. They also provide tools to create and verify digital signatures. One major shortcoming of the NSS certificate manager is the inability to search for certificates based on specific attributes. Also, the current certificate and key databases cannot be extended with metadata. There are plans to use SQLite [23], a SQL database, as the backend for Mozilla Firefox 3.0.

1.3.2 Pidgin CertMgr plugin

As a Google-sponsored Summer of Code 2007 project [9], William Ehlhardt wrote a "certificate manager" [32] to enable Pidgin users to have SSL encrypted conversations. His stated motivation reveals a modest goal.

> Pidgin doesn't currently do any certificate verification for SSL. In order to properly do this and ensure security, a certificate manager (something like Mozilla's) needs to be added. [32]

CertMgr uses NSS as a backend, but does nothing as far as managing other cryptographic items and bindings (see Section 3.2.1 and page 26).

1.3.3 Pidgin-Encryption plugin

The Pidgin-Encryption plugin [20] provides encryption and authentication for IM conversations. Its features tend to focus on being user friendly, e.g. public/private key pairs are generated when the plugin is loaded. The user's public key is automatically sent to any contacts. Known contacts' public keys are saved locally, and the user is notified if a contact's public key changes. Keys are stored directly on the file system, so that they can be manipulated easily.

One major downside of this plugin includes lack of support for multi-user chats. Also, it is only available for Pidgin on Windows [28].

1.3.4 Off-the-Record plugin

The Off-the-Record (OTR) plugin [15] seems to be significantly more complex and mature than the Pidgin-Encryption plugin. Not only does it provide encryption and authentication, but it provides other cryptographic properties such as deniability and perfect forward secrecy. OTR messages are *not* digitally signed, unlike messages from Pidgin-Encryption. Also, OTR offers perfect forward secrecy; previous conversations are not compromisable with current keys.

OTR also has extensive documentation on its Authenticated Key Exchange (AKE), which uses unauthenticated Diffie-Hellman key exchange to set up an encrypted channel and then authentication inside that channel. However, like Pidgin-Encryption, OTR is not usable inside multi-user chats.

1.3.5 Certex

Prior to this project, Joe Cooley wrote an application that uses Bonjour and NSS to exchange certificates. Bonjour is a networking technology that allows for computers and other devices to communicate over Ethernet or wireless (802.11) without setup. NSS is described in section 1.3.1. With Certex, when a new machine is detected on the network, its certificate is sent to other machines, and vice versa. Certex also includes a plugin for what used to be Gaim (now Pidgin).

1.4 Accomplishments

What did we actually end up accomplishing? We built a cross-platform identity and certificate manager (ICM) library, using OpenSSL's cryptography and SQLite's database. We wrote a utility that allows users to manage their ICM directly from the commandline. We also designed an Identity and Certificate Exchange (ICE) protocol, whose logic is embedded in our library. The protocol efficiently desseminates users' certificates and username-certificate bindings among users. Finally, we wrote a Pidgin plugin, which utilizes our library, and showed that our exchange protocol works as intended. The plugin authenticates IM and multi-user chat messages using digital signatures. The plugin balances security and usability, using informative notifications and error messages.

1.5 Roadmap

In the next chapter (Chapter 2), we detail why we undertook this project, and what we hoped to achieve. In Chapter 3, we explain the designs for our library, protocol and plugin. In Chapter 4, we describe our implementations for the library, protocol and plugin. In Chapter 5, we evaluate our work measuring the protocol's performance in a number of different situations. Finally, in Chapter 6, we enumerate possible extensions to our work, and then conclude.

Chapter 2

Design Goals

In this chapter, we explore the design goals for the Identity and Certificate Manager (ICM). Our notion of "identity" encompasses email addresses, instant message usernames, or website logins. Any information that indicates an individual person (or possibly even a group of people) is an identity. Our use of the term "certificate" is imprecise; what we really mean is a "cryptographic item" manager. We use the term "certificate" because certificates are the main cryptographic items in ICM. See Section 3.2.1 for more on cryptographic items and identity. We designed the ICM system with four main goals in mind: functionality, security, usability and performance.

2.1 Functionality

Our vision for ICM allows a single user to access from a single repository any and all of the cryptographic items that the user has accumulated. That repository could be a USB drive that the user carries around. Or, perhaps sometime in the future, the repository could be surgically implanted in the user's body. Maybe it is more convenient (and less physically invasive) to have the repository accessible online. The user must be able to access the repository at any time and not worry about keeping it up to date. Applications should be able to rely on our system to handle storage and manipulation of cryptographic objects, message signing and signature verification, as well as secure management of identities.

2.2 Security

We want our system to be secure; we must be sure that malicious individuals cannot tamper with our system. We need to be mindful of SQL-injection attacks and buffer overflows. We rely on well-known cryptography for authentication, such as PKI, CRLs, and OCSP (see Appendix A). These schemes provide us with the security assurances that we seek. We use digital signatures to prevent attackers from tampering with our messages. We use cryptographic bindings to create a chain of trust between the root certificate authority and the application-level username. After creating such a binding, the username can "speak for" the certificate authority, attesting to the user's identity [36].

2.3 Usability

One problem that prevents users from using security features is the lack of usability in security tools. Some say that "security and usability are often inversely proportional" [21], but we aim to make our ICM usable by typical computer users in common situations. Having a clean graphical user interface, as well as informative error messages in layman's terms, are important aspects of designing a usable security application.

2.4 Performance

With technology progressing at an exponential rate [35], application performance seems to be less important in the minds of many. However, in some situations, performance is especially important. For example, if a message must be transmitted via a low-bandwidth radio, e.g. a cell phone, message sizes matter. Or if a message incurs extremely high round-trip-times because it is bounced off a satellite, then the number of overhead messages should be kept to a minimum. In our ICM, we try to minimize both the number and size of messages required to exchange identities and cryptographic items.

Chapter 3

Design

In this chapter we describe the design of the ICM system. The first section summarizes the different modules, and how they relate to each other. The subsequent sections describe the modules themselves, in the order in which they were designed.

3.1 Modules

We designed ICM from the bottom up. That is, we originally wanted an interface with a database that would help users keep track of their cryptographic items (see Section 3.2.1). We also want to keep track of the users' communication contacts, and who has which of their cryptographic items. Naturally we designed a module directly on top of the database (DB). We built two modules and an application on top of the DB layer (see Figure 3-1). The CryptoLib module implements all of the cryptographic functions necessary for secure communications. The Exchange module implements our Identity and Certificate Exchange (ICE) protocol, using the CryptoLib module heavily and storing its own state in the database. The Admin application provides a simple commandline interface for a user to manage cryptographic items stored in the database. A browser-based GUI gives the user a more intuitive way to manage their database. In order to present a single high level API to other applications, we have an Identity and Certificate Manager (ICM) layer sitting atop the Exchange and CryptoLib modules. The c_icm layer consists of C wrappers around the C++

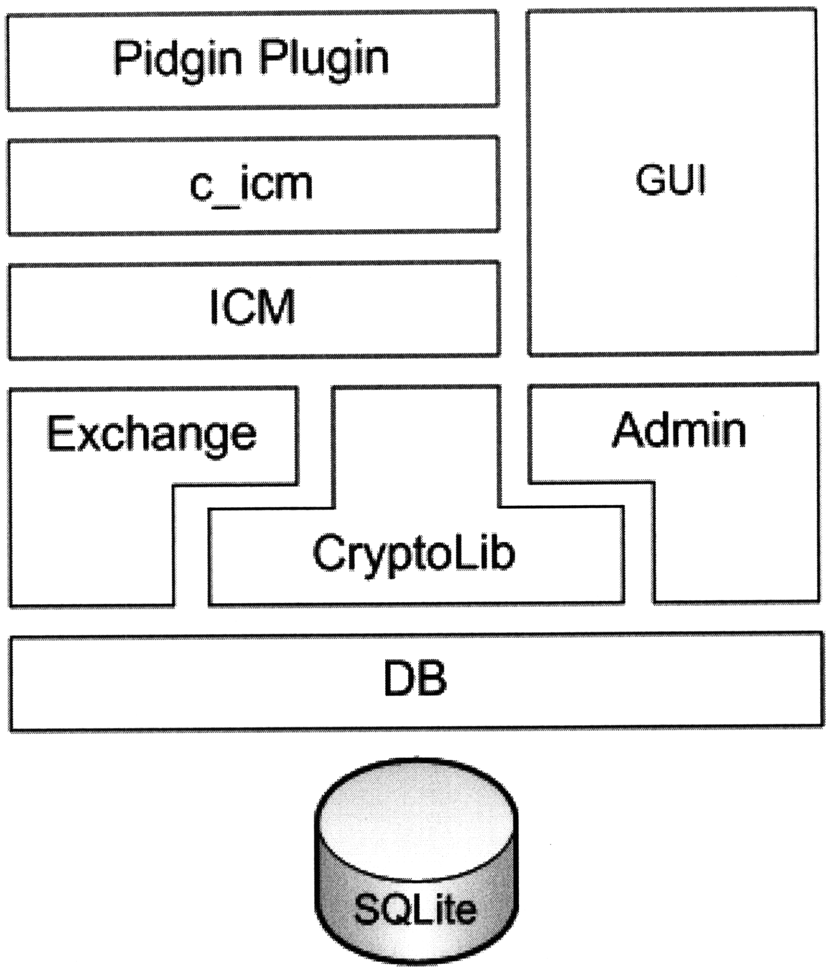


Figure 3-1: ICM Module Diagram

functions of the ICM layer. Finally, we designed a Pidgin plugin (written in C) to use the c_icm API.

3.2 DB

The DB module represents the backend database used to store the user's cryptographic items, and the state associated with the ICE protocol. In this section, we first explore the schema in detail, followed by a discussion of why we chose to use SQLite, and how the database can be maintained.

3.2.1 Schema

The database schema can be split into three main sections. First, there are the tables representing cryptographic items; these include public keys, private keys, certificates, shared keys, and passwords. Then, there is a table containing entities: Jabber [11] usernames and email addresses. Finally, there are several tables relating entities and items.

Cryptographic Items

Why are we storing cryptographic items in a relational database? We could have easily stored all the items as encrypted files directly on the filesystem. Our reasoning behind using a relational database is simple;

- We want to take advantage of the SQL relations. For example, we store metadata with foreign keys referring to items.
- Being able to search for an item based on a particular attribute is very important. Instead of simply storing binary representations of items, we extract useful information that could help the user locate that item again. For example, when storing a public key, we store the algorithm that it was generated for (RSA, or DSA, or maybe ECDSA). We also store what the public key is used for (either encryption or verification).
- We organize our cryptographic items into tables based on their type; certificates and private keys are stored in their own tables. Some items have attributes that others do not.

Cryptographic items all have a `hash` column. Each item has its own definition of a hash; for certificates, it's the hash of the embedded public key. This hash is used by all users to refer to the same item, but the hash is smaller than the item itself. For example, if Alice wants to know if Bob has her certificate, she could send a hash of her certificate to Bob, saving hundreds of bytes worth of communication. To achieve even better performance (communication savings), we can use a prefix of the hash to

```
CREATE TABLE certificates (
        id                          INTEGER PRIMARY KEY,
        trusted                     BOOLEAN DEFAULT 't',
        hash                        BLOB,
        -- useful if trying to construct a chain of trust
        ca_id                       INTEGER,
        -- fields pulled directly from the certificate
        is_ca                       BOOLEAN DEFAULT 'f',
        common_name                 TEXT,
        email                       TEXT,
        organization                TEXT,
        organization_unit           TEXT,
        state                       TEXT,
        country                     TEXT,
        ca_organization             TEXT,
        ca_state                    TEXT,
        ca_country                  TEXT,
        -- the certificate itself
        certificate                 BLOB,
        FOREIGN KEY (ca_id) REFERENCES certificates(id)
);

CREATE TABLE cert_rev_lists (
        id                          INTEGER PRIMARY KEY,
        trusted                     BOOLEAN DEFAULT 't',
        hash                        BLOB,
        ca_organization             TEXT,
        ca_state                    TEXT,
        ca_country                  TEXT,
        cert_rev_list               BLOB
);

CREATE TABLE public_keys (
        id                          INTEGER PRIMARY KEY,
        trusted                     BOOLEAN DEFAULT 't',
        hash                        BLOB,
        algorithm                   TEXT,
        usage                       INTEGER,
        -- PEM or DER
        type                        INTEGER,
        public_key                  BLOB
);

CREATE TABLE private_keys (
        id                          INTEGER PRIMARY KEY,
        trusted                     BOOLEAN DEFAULT 't',
        hash                        BLOB,
        algorithm                   TEXT,
        usage                       INTEGER,
        -- PEM or DER
        type                        INTEGER,
        private_key                 BLOB
);
```

Figure 3-2: Database Schema - Items

identify cryptographic items. We call this predefined-length prefix the item's "MID" (member ID).

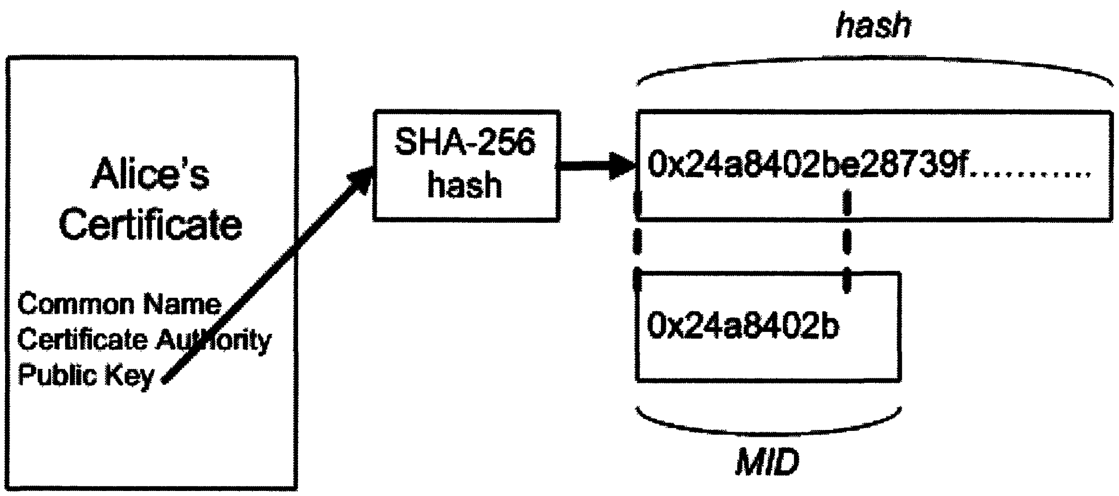


Figure 3-3: Deriving the Hash/MID from a Certificate

The MID is used whenever referring to a particular item in communication. If there are two distinct items which happen to have the same MID, then we iterate through the possible matches to determine which item is being specified. The use of MIDs in our system trades computation time and storage in exchange for less communication. The probabilities of MIDs colliding is discussed later, in Section 3-8. We originally wanted to use the MID as the primary key for every item's table. However, this would have caused a big problem if two distinct cryptographic items had the same MID. The second item would not be able to be inserted into the database, because every primary key in a table must be distinct. Instead, we use *local IDs* as primary keys which can be guaranteed to be unique. When trying to identify an item by its MID, we simply compare to the stored hash column. However, this solution requires us to perform frequent conversions between an item's MID and local unique ID. The ID is used by metadata tables which are explained two sections later.

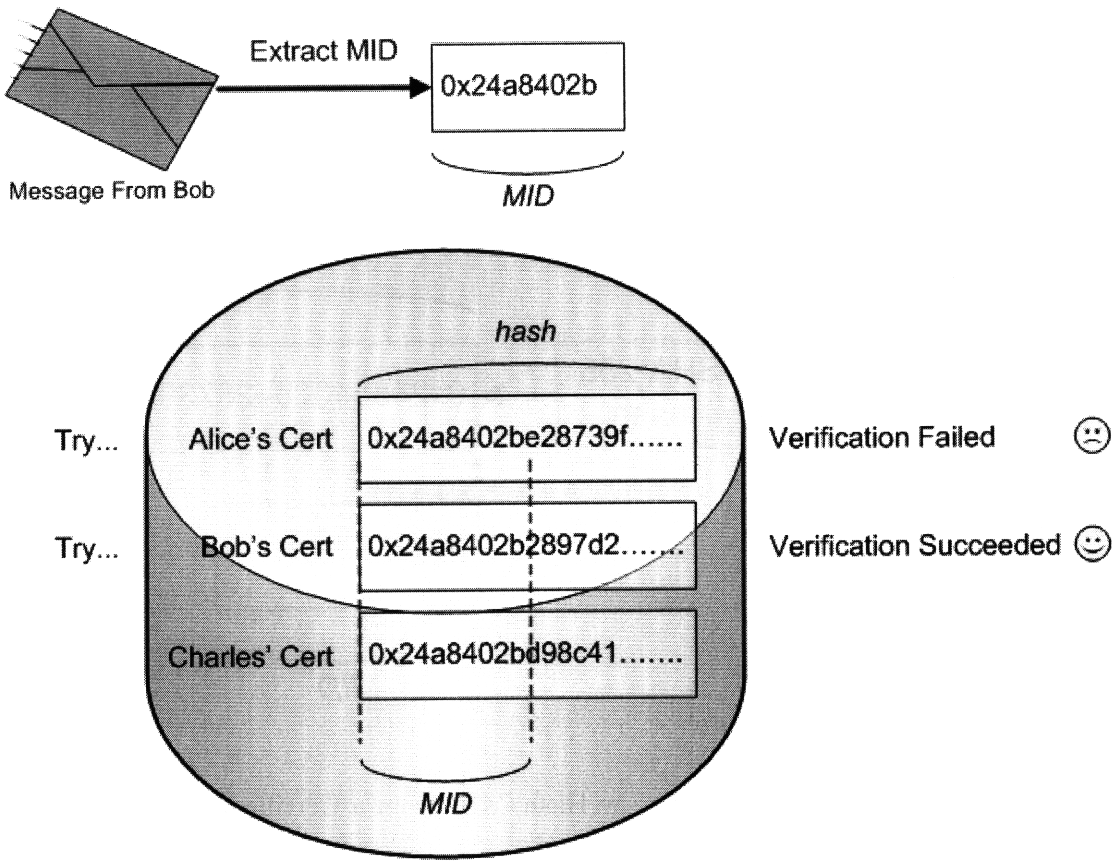


Figure 3-4: Iterating Through MID Collisions (Note: this scenario is very rare, see page 32)

Entities

The database also stores entities. Entities have a few descriptive fields, such as name and email, as well as a hash field. Entities can be described by their MIDs in the same way that cryptographic items are. Again, just like items, entities have their own local unique IDs so that they can be unambiguously referred to by metadata tables (explained below).

Metadata

The ICM database also contains several metadata tables. The most straightforward of the metadata tables is the bindings table. A binding can be described as a relationship in which the entity "speaks for" a cryptographic item by an entity (see Figure 3-5).

```
CREATE TABLE entities (
        id                      INTEGER PRIMARY KEY,
        hash                    BLOB,
        name                    TEXT,
        email                   TEXT
);

-- associate entities with their items
CREATE TABLE bindings (
        id                      INTEGER PRIMARY KEY,
        entity_id               INTEGER NOT NULL,
        -- references one of the item tables
        item_id                 INTEGER,
        item_type               INTEGER,
        principal               BOOLEAN DEFAULT 'f',
        FOREIGN KEY (entity_id) REFERENCES entities(id)
);

-- owners of certs that have a certificate/public key of mine
CREATE TABLE sent_to_cert (
        -- the certificate id (not mine)
        cert_id                 INTEGER,
        -- references one of the item tables
        my_item_id              INTEGER,
        my_item_type            INTEGER,
        my_item_bool            BOOLEAN DEFAULT 'f',
        my_binding_id           INTEGER,
        my_binding_bool         BOOLEAN DEFAULT 'f',
        FOREIGN KEY (cert_id)   REFERENCES certificates(id),
        FOREIGN KEY (my_binding_id) REFERENCES bindings(id)
);

CREATE TABLE pending_binding (
        id                      INTEGER PRIMARY KEY,
        entity_id               INTEGER,
        my_binding_id           INTEGER,
        FOREIGN KEY (entity_id) REFERENCES entities(id),
        FOREIGN KEY (my_binding_id) REFERENCES bindings(id)
);
```

Figure 3-5: Database Schema - Entities/Metadata

For example, "alice@jabber.org" "speaks for" a particular certificate-private key pair. Thus, each binding refers to an entity and an item, both by local IDs. Since there are many different item tables, each binding must also have an item type field. Each binding has its own local ID so that other metadata tables can refer to it. The last field in each binding is a "principal" Boolean. If it is set to `true`, then the item referred is the entity's principal item for that type. Each entity can have only one item of each item type marked as principal. The idea for a principal item for a given entity is similar to a default.

Example: Suppose the item type is a certificate. Alice has a certificate signed by Certificate Authority 1. Then, Alice gets a new certificate signed

> by Certificate Authority 2, and wants this new certificate to be used instead of the first certificate. Instead of deleting the first certificate (which may be useful in the future), the principal flag allows Alice to simply set the second certificate to be the principal.

Bindings allow for a decoupling of a user's certificate and the user's entity. If the user wants to change names (i.e. switch entities), there is no need to get a new certificate. The Certificate Authority does not need to know the user's entity at the time of the certificate generation, so the user gains a lot of flexibility. Also, if the user's certificate expires, a new certificate can easily be bound to the old entity.

Two other metadata tables keep track of the state necessary to achieve performance enhancements in the Identity and Certificate Exchange (ICE) protocol (see Section 3.4). The first is revealingly named `sent_to_cert`. This table keeps track of which items and bindings have been sent to a particular recipient. That recipient is identified by the local ID of his/her certificate. The rest of the columns in the table indicate which item and which binding have been sent. The item ID and item type columns unambiguously identify an item. The binding ID identifies a binding. The remaining two columns are Booleans, which allow the table some flexibility: the first Boolean represents whether the recipient has the user's item, and the second represents whether the recipient has the user's binding.

> **Example:** Suppose Bob has sent Alice his certificate and his binding, but Alice somehow loses Bob's binding. When Bob receives evidence that Alice has his certificate, but not his binding, he marks the first Boolean true (i.e. Alice has received his item), but the second Boolean false (i.e. Alice does not have his binding). Having this information allows for high performance recovery; Bob only needs to resend his binding, and not his certificate.

The second metadata table which stores state for the ICE protocol is called `pending_binding` (again, see Figure 3-5). A row in this table represents the fact that we have sent our binding and certificate to the specified entity.

```
CREATE TABLE applications (
        id                      INTEGER PRIMARY KEY,
        name                    TEXT
);

CREATE TABLE application_bindings (
        id                      INTEGER PRIMARY KEY,
        application_id          INTEGER,
        binding_id              INTEGER,
        FOREIGN KEY (application_id) REFERENCES applications(id),
        FOREIGN KEY (binding_id) REFERENCES bindings(id)
);
```

Figure 3-6: Database Schema - Applications

Example: Suppose Alice enters a chatroom, and sees "bob" and "charles". She immediately sends her certificate to both of them so that they can authenticate her future messages. Before she receives confirmation that "bob" has received her certificate, she receives his certificate. Her `sent_to_cert` table has no indication that she has already sent her certificate to "bob" because she did not have his certificate at the time. The `pending_binding` serves a similar purpose to the `sent_to_cert` table, except it is based on the entity's ID, instead of the certificate's ID. In this example, `pending_binding` would contain a row that indicated that Alice had sent "bob" her certificate and her username "alice".

Application-specific data

Every application using the ICM has its own notion of identity. For example, Pidgin's notion of Alice's identity might be "alice3@jabber.org". Similarly, Microsoft Outlook's [12] notion of identity may be "alice@microsoft.com". In order to represent these different notions, we have the `applications` table and `application_bindings` table (see Figure 3-6). The `applications` table simply maps each application name (and perhaps other information in the future) to an ID. The `application_bindings` table maps each application to a binding. Therefore, when an application initializes ICM, ICM knows which binding to use. Each application can have its own binding, while sharing information about the user's contacts.

3.2.2 SQLite

We chose to use SQLite [23] as the backend database for a number of reasons. SQLite is a lightweight and reasonably high performance database with minimal configuration. SQLite is ACID compliant (atomic, consistent, isolated and durable) and the entire database is a single file. SQLite has a full-featured C/C++ API (well-tested with over 98% coverage), and it is completely free (no licensing at all).

We use a single database for each user. This allows different applications to share their items with each other. We do not, however, want to have multiple people using the same database; we want to keep each user's information isolated for security.

3.2.3 DB Maintenance

We want to allow users to manipulate the database. Users may want to add items manually, delete them, or perform other simple database options. While users may use the sqlite3.exe client, they may not want such a low level view of their database. Applications may choose to allow certain database operations (for an example, see Section 4.8.4). We did, however, design a command line tool (Admin) to allow for simple inserts and deletes. This tool is more user-friendly than the generic sqlite client, but also more powerful than a high-level application's GUI. This tool allows one to insert cryptographic items into the database from files (like certificates in Distinguished Encoding Rules (DER) [4] or Privacy Enhanced Mail (PEM) [18] format). The browser-based GUI allows users to manage their database even more intuitively.

3.3 CryptoLib

This module contains all of the cryptographic operations performed by our system. We purposely designed this module so that multiple versions could be implemented, each using different backend cryptography libraries. This flexibility makes the system easier to maintain, update, and extend. Some of the important operations performed in CryptoLib are certificate validation and creation/verification of digital signatures.

3.4 Identity and Certificate Exchange Protocol

The purpose of the Identity and Certificate Exchange (ICE) protocol is to provide a mechanism for users to engage in authenticated communication. To do this, users must exchange a binding between the application's notion of identity (i.e. username) and a cryptographic notion (i.e. certificate; see Appendix A.1). In our efforts to make communications secure, we must reconcile an application's notion of a user with the cryptosystem's notion of a user. We want the user to have the flexibility to use any valid certificate with any application. Users can exchange only their own certificates, i.e. certificates for which they have private keys.

Depending on the situation, message authentication can be merely comforting or highly mission critical. A digital signature is attached to every single message sent via ICM, including messages containing certificates. Confidentiality (achieved via encryption) can easily be implemented on top of our protocol, so we decided to decouple confidentiality from our authentication goal.

3.4.1 ICE Overview

If performance (bandwidth/latency) were of no concern to us, we could easily attach our identity, certificate and signature to every single message and have a simple authenticated protocol. Sending the same identity and certificate in every message is highly wasteful. This is in fact exactly the way S/MIME (secure email) (see Appendix A.4) works; the certificate is sent in every email. Compare that with Secure Socket Layer (SSL) which initiates every session with a seven-way "handshake" to exchange certificates and set up an encrypted channel through which to send messages. After the SSL connection is established, certificates are no longer exchanged. If the connection ends, and is re-established, the handshake must be accomplished again to initiate a new session.

Our ICE protocol tries to take the idea of a "session" from SSL and go one step further. Once we receive a certificate, we store it in our database so that we can access it in the future. In order to prevent sending our certificate to the same user

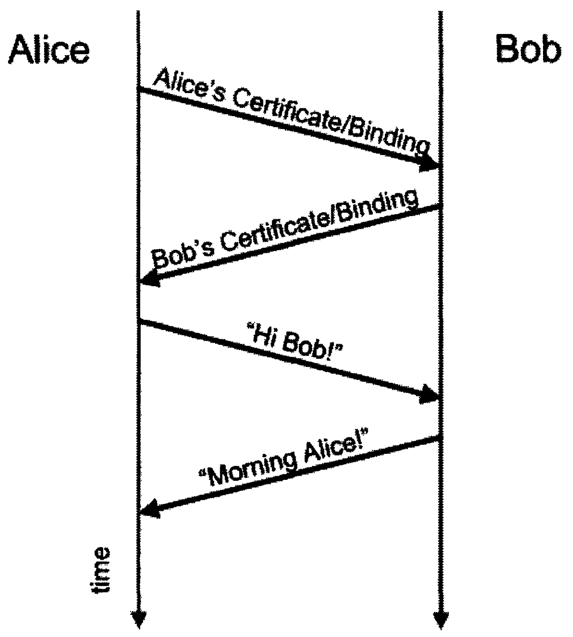


Figure 3-7: ICE Example Timeline (common case: smart push)

more than once, we remember who has received our certificate by storing state in our database. In general, we remember what information other users have about us to eliminate extraneous communications.

In typical usage, ICM exchanges identities and certificates via a "smart push" method. That is, each user's ICM consults its own data to decide whether the other user is missing either his identity or certificate. If one or both are missing, ICM sends the required information. The metadata that we store makes this common case exchange very efficient.

When an unexpected event occurs, then ICM "pulls" (requests) the missing information. These unexpected events, such as failures and other errors, are detected when trying to verify a message's signature. ICM sends the signing certificate's MID (see page 25) or the certificate itself in every message. When verifying a certificate, ICM matches the MID to a certificate in its database, and uses that certificate to verify the signature. The MID is only on the order of 10 bytes, compared to hundreds or even thousands of bytes in a typical certificate. Of course, to achieve this performance gain, we sacrifice simplicity in certain unlikely scenarios such as hash collisions. Collisions are unlikely and their probability only grows with the square root of the number of possible distinct bit strings ($\sqrt{2^{length}}$). For example, if we use 4 bytes (32 bits) to represent items, then we would need on the order of $\sqrt{2^{32}} = 2^{16} = 65536$ items to have a good chance of getting a collision. According to Figure 3-8, we would need to have more than 1000 MIDs to be less than 99.99% sure that we have no collisions.

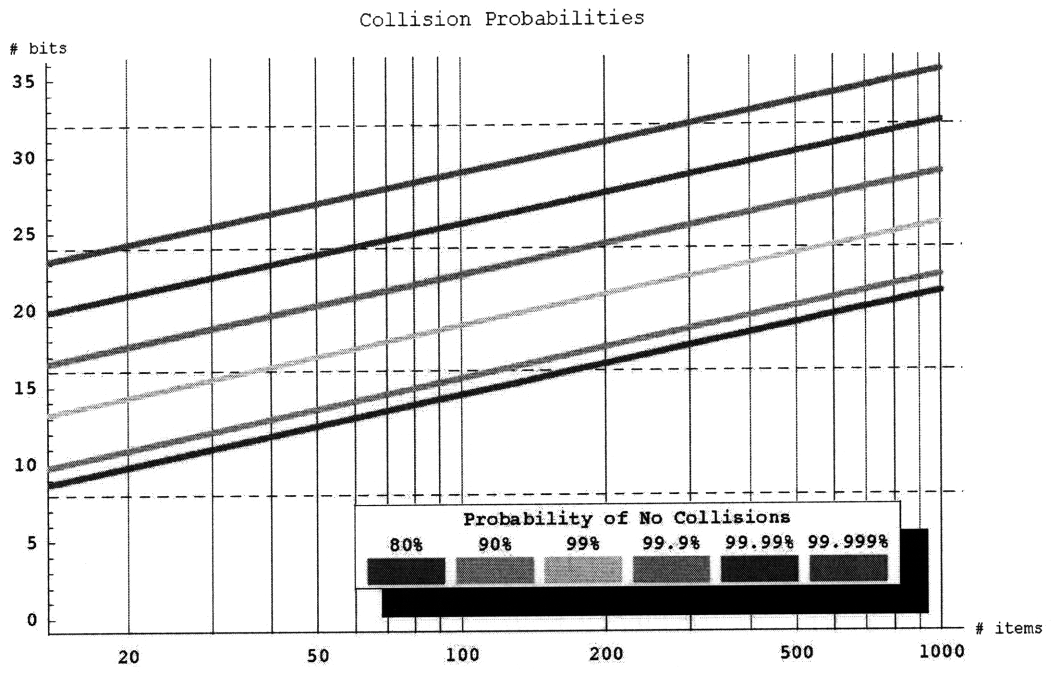


Figure 3-8: Probability of Collisions

3.4.2 Exchange module

The ICE protocol is realized in the functions contained by the Exchange module. All of the protocol logic is contained in two functions CREATECERTIDMSG and RECEIVEMSG (described in pseudocode below). CREATECERTIDMSG contains logic for the "smart push" scenarios, i.e. sending necessary information unrequested. RECEIVEMSG is called by the recipient of any ICE message, which may trigger a "pull" for the sender's identity or certificate.

3.4.3 ICE pseudocode

```
CREATECERTIDMSG($entityMID, certMID, request, cert$)
 1  $result.type \leftarrow$ TYPE_CERTID_MSG
 2  $result.request\_flag \leftarrow request$
 3  $result.stmtLen \leftarrow$ LEN($myEntity.name$)
 4  $result.stmt \leftarrow myEntity.name$
 5  $result.timestamp \leftarrow$ TIME()
 6  // No recipient specified
 7  if $entityMID =$ NIL and $certMID =$ NIL
 8     then $result.recipient \leftarrow$ TYPE_RECIPIENT_BROADCAST
 9          $result.ack \leftarrow$ NIL
10          if $certMID \neq$ NIL
11             then $result.sgnrLen \leftarrow$ LEN($myCert$)
12                  $result.sgnr \leftarrow myCert$
13                  $result.cert \leftarrow$ TYPE_CERT_CERT
14             else $result.sgnrLen \leftarrow 0$
15                  $result.sgnr \leftarrow myCert.mid$
16                  $result.cert \leftarrow$ TYPE_CERT_MID
17     else // We have recipient's cert, or can look it up
18          if $certMID \neq$ NIL or
19             ENTITYMID2CERTMID($entityMID, certMID$) $= 1$
20             then $result.recipient \leftarrow$ TYPE_RECIPIENT_UIDMID
21                  $result.ack \leftarrow (entityMID, certMID)$
22     else // We only have recipient's name
23          $result.recipient =$ TYPE_RECIPIENT_UID
24          $result.ack \leftarrow (entityMID)$
25          $result.sgnrLen \leftarrow$ LEN($myCert$)
26          $result.sgnr \leftarrow myCert$
27          $result.cert \leftarrow$ TYPE_CERT_CERT
```

```
// We know recipient's cert
if $certMID \neq$ NIL
  then if HASMYBINDING($certMID, myCert.id, myBindingID$) and
          $request = 0$
          then return NIL
       if HASMYCERT($certMID, myCert.id, myBindingID$)
          then $result.sgnrLen \leftarrow 0$
               $result.sgnr \leftarrow myCert.mid$
               $result.cert \leftarrow$ TYPE_CERT_MID
          else $result.sgnrLen \leftarrow$ LEN($myCert$)
               $result.sgnr \leftarrow myCert$
               $result.cert \leftarrow$ TYPE_CERT_CERT
       HASMYCERT($certMID, myCert.id, myBindingID$) $\leftarrow$ TRUE
       HASMYBINDING($certMID, myCert.id, myBindingID$) $\leftarrow$ TRUE
       $entityID \leftarrow$ ENTITYMID2ENTITYID($entityMID$)
       PENDINGBINDING($entityID, myBindingID$) $\leftarrow$ FALSE
  else // We know recipient's username
       if $entityMID \neq$ NIL
          then $entityID \leftarrow$ ENTITYMID2ENTITYID($entityMID$)
               if PENDINGBINDING($entityID, myBindingID$) $=$ TRUE
                  and $request = 0$
                  then return NIL
               PENDINGBINDING($entityID, myBindingID$) $\leftarrow$ TRUE
return $result$
```

```
RECEIVEMSG($packedMsg, packedMsgLen, sender, msgText, verifyFunc$)
// Unpack Message
$msg \leftarrow$ MSGUNPACK($packedMsg, packedMsgLen$)
if $msg =$ NIL
  then error "Message could not be unpacked"
```

```
        return NIL
msgText ← msg.stmt
// Obtain Signing Certificate (from database or message)
if msg.cert = TYPE_CERT_MID
  then certMID ← msg.sgnr
        signingCert ← CERTMID2CERTIFICATE(certMID, numCerts)
        if numCerts = 0
          then if msg.request_flag = TYPE_REQUEST_TRUE
                  then HASMYCERT(certMID, myCert.id, myBindingID) ← FALSE
                         HASMYBINDING(certMID, myCert.id, myBindingID) ← FALSE
                e.name ← sender
                ADDENTITY(e)
                return CREATECERTIDMSG(e.mid, certMID, TRUE , FALSE )
  else signingCert ← BUF2X(msg.sgnr, msg.sgnrLen)
        if signingCert = NIL
          then error "Failed to convert attached certificate to X509"
                return NIL
// Check Signature
if MSGVERIFY(msg.sig, msg.sigLen, packedMsg, signingCert) = FALSE
  then error "Unable to verify signature"
        return NIL
// Check to see that this message is for me
if msg.recipient ≠ TYPE_RECIPIENT_BROADCAST
  then if msg.ack ≠ NIL or msg.ack ≠ myEntity.mid
          then error "Message not intended for this user"
                return NIL
        if msg.recipient = TYPE_RECIPIENT_UIDMID and
          msg.ack[ MID_SIZE ] ≠ myCert.mid and certMID ≠ NIL
          then error "Sender has wrong certificate for you"
                HASMYCERT(certMID, myCert.id, myBindingID) ← FALSE
        HasMyBinding($certMID, myCert.id, myBindingID$) $\leftarrow$ FALSE
        $entityMID \leftarrow$ Bits2MID($sender$)
        return CreateCertIDMsg($entityMID, certMID$, FALSE , FALSE )
// Check Message type
if $msg.type \neq$ TYPE_TEXT_MSG and $msg.type \neq$ TYPE_CERTID_MSG
  then error "Unknown message type"
    return NIL
// Done with text messages
if $msg.type =$ TYPE_TEXT_MSG
  then return NIL
// Only CertID messages from here
// Check sender's username and binding match
if VerifyFunc$sender, msg.stmt =$ FALSE
  then error "Sender not verified"
    return NIL
$e.name \leftarrow msg.stmt$
$senderID \leftarrow$ AddEntity($e$)
$entityMID \leftarrow e.mid$
// Add certificate to database
if $msg.cert =$ TYPE_CERT_CERT
  then AddX($signingCert, e$)
    if EntityMID2CertMID($entityMID, certMID$) $> 1$
      then error "Entity MID collision"
        return NIL
    if $msg.recipient =$ TYPE_RECIPIENT_UID
      then error "Sender has no certificate for you"
        if HasMyBinding($certMID, myCert.id, myBindingID$)
          then HasMyCert($certMID, myCert.id, myBindingID$) $\leftarrow$ FALSE
        HasMyBinding($certMID, myCert.id, myBindingID$) $\leftarrow$ FALSE
$numEntities \leftarrow$ EntityMID2CertMID($entityMID, temp$)
```

```
65 if $numEntities > 1$
66   then error "Entity MID collision"
67        return NIL
68   else $c.mid \leftarrow certMID$
69        $certID \leftarrow$ ITEM2ITEMID($c$)
70        INSERTBINDING($senderID, certID$, ITEM_CERTIFICATE )
71        SETPRINCIPAL($senderID, certID$, ITEM_CERTIFICATE , TRUE )
72        ENTITYMID2CERTMID($entityMID, temp$)
73 if $certMID \neq$ NIL and $certMID \neq temp$
74   then error "Username is associated with a different certificate"
75 $certMID \leftarrow temp$
76 if CERTMID2ENTITYMID($certMID, temp$) $= 0$
77   then error "Can't find an entity bound to this certificate"
78        return NIL
79 if $entityMID \neq temp$
80   then error "Username change"
81 if ENTITYMID2ENTITYID($entityMID, entityID$) $\neq 1$
82   then error "Entity MID collision"
83        return NIL
84 if $msg.request\_flag =$ TYPE_REQUEST_TRUE
85   then HASMYCERT($certMID, myCert.id, myBindingID$) $\leftarrow$ FALSE
86        HASMYBINDING($certMID, myCert.id, myBindingID$) $\leftarrow$ FALSE
87   else if PENDINGBINDING($entityID, myBindingID$) $=$ TRUE
88          then HASMYCERT($certMID, myCert.id, myBindingID$) $\leftarrow$ TRUE
89               HASMYBINDING($certMID, myCert.id, myBindingID$) $\leftarrow$ TRUE
90   else if $msg.recipient =$ TYPE_RECIPIENT_UIDMID and
91          $msg.ack = myEntity.mid$ and $msg.ack$[ MID_SIZE ] $= myCert.mid$
92          then HASMYCERT($certMID, myCert.id, myBindingID$) $\leftarrow$ TRUE
93               HASMYBINDING($certMID, myCert.id, myBindingID$) $\leftarrow$ TRUE
94 return CREATECERTIDMSG($entityMID, certMID$, FALSE , FALSE )
```

3.5 Pidgin Plugin

We designed a Pidgin plugin as an example application of ICM and ICE. The plugin will use the ICE protocol in two separate manners depending on the mode of communication that the user chooses. In an instant message (IM) conversation, where the user is only communicating with another single user, the ICE protocol will simply exchange certificates (if necessary!) and cryptographic username-certificate bindings. In a chat conversation, which is often called a "chatroom" because there may be multiple users in the room, the user must exchange certificates with all the other users in the chat. One thing to remember, whenever a user sends a message, IM or chat, that message is packed into the ICE protocol message format (see Section 4.6.1).

As soon as a user tries to send an IM, the ICE protocol is activated. If necessary, the user's plugin sends his/her certificate, before sending the IM. On the other end, as soon as the user receives a message which is tagged as being part of the ICE protocol, that user's plugin may respond with his/her own certificate if necessary.

Chat exchanges are significantly more complicated. When a user (Alice) joins a chat, the plugin first checks the list of users in the chat. If there are any users to whom Alice has not sent her certificate, then the plugin sends a single message containing her certificate and cryptographic binding to the chat. The chat server then distributes this message to the chat's users. The plugins of the other users in the chat may send their bindings and/or certificates to Alice if necessary. If there are n users in the chatroom, Alice sends at most 1 message to the chat, and each of the n users sends at most 1 message to Alice.

Chapter 4

Implementation

In this chapter, we focus on our implementation of the designs described in the preceding chapter. We also describe coding conventions we used, as well as third-party libraries. The format of this chapter is mostly parallel to that of the Design chapter; we discuss the modules from the bottom up.

4.1 Coding Conventions

In our C++ code (the ICM library, Admin tool), we use lowerCamelCase for variable names, and UpperCamelCase for class/struct names [2].

4.2 Libraries

We used 3 open source libraries in this system. First, there's the free SQLite [23] software which has a full-featured C API. For all our cryptography needs, we used OpenSSL (version 0.9.8e) [17]. Finally, we used ADAPTIVE Communication Environment (ACE) [25] for our portable types and data structures

4.3 Doxygen Documentation

We used the open source tool Doxygen to produce documentation for our code. All of our classes are thoroughly documented.

4.4 DB

The purpose of the DB class is to abstract away all of the underlying database API calls. This allows future code maintainers/updaters to easily change the underlying database without changing the higher level code. We specifically did not want any SQL code in any layer higher than the DB. Since only the DB layer would be making any direct calls to the SQLite API, we decided to open the SQLite connection in the DB constructor, and subsequently close it in the destructor. The constructor also checks to see if the database has been initialized with our schema. If the database is empty, then the constructor loads the schema into the database from a file.

4.4.1 Items

Cryptographic items are represented as C/C++ structs. Our implementation supports three types of items: asymmetric keys (public and private keys), certificate revocation lists, and certificates.

We use inheritance and virtual functions (see Figure 4-1) to maximize code reuse and modularity. Our base class, Item, has a few fields common to all cryptographic items. First, all Items have an integer ID which is that Item's database ID. Second, every Item is either trusted or not. This Boolean (which defaults to true) indicates whether the user trusts this Item to be used in cryptographic operations, or whether it should not be used anymore. Third, every Item has a "type" which is simply an enumeration of the different cryptographic item types. Finally, all Items have `MID`, `hash` and `hashSize` fields. The `hash` is a bytestring produced by running the SHA256 hash over some particular part of the Item. For example, SHA256 is applied to the public key of a certificate to produce the certificate's hash. The `hashSize` is the number of bytes in the `hash`. The `MID` is a short prefix of the `hash`, and is explained in further detail in Section 4.4.3.

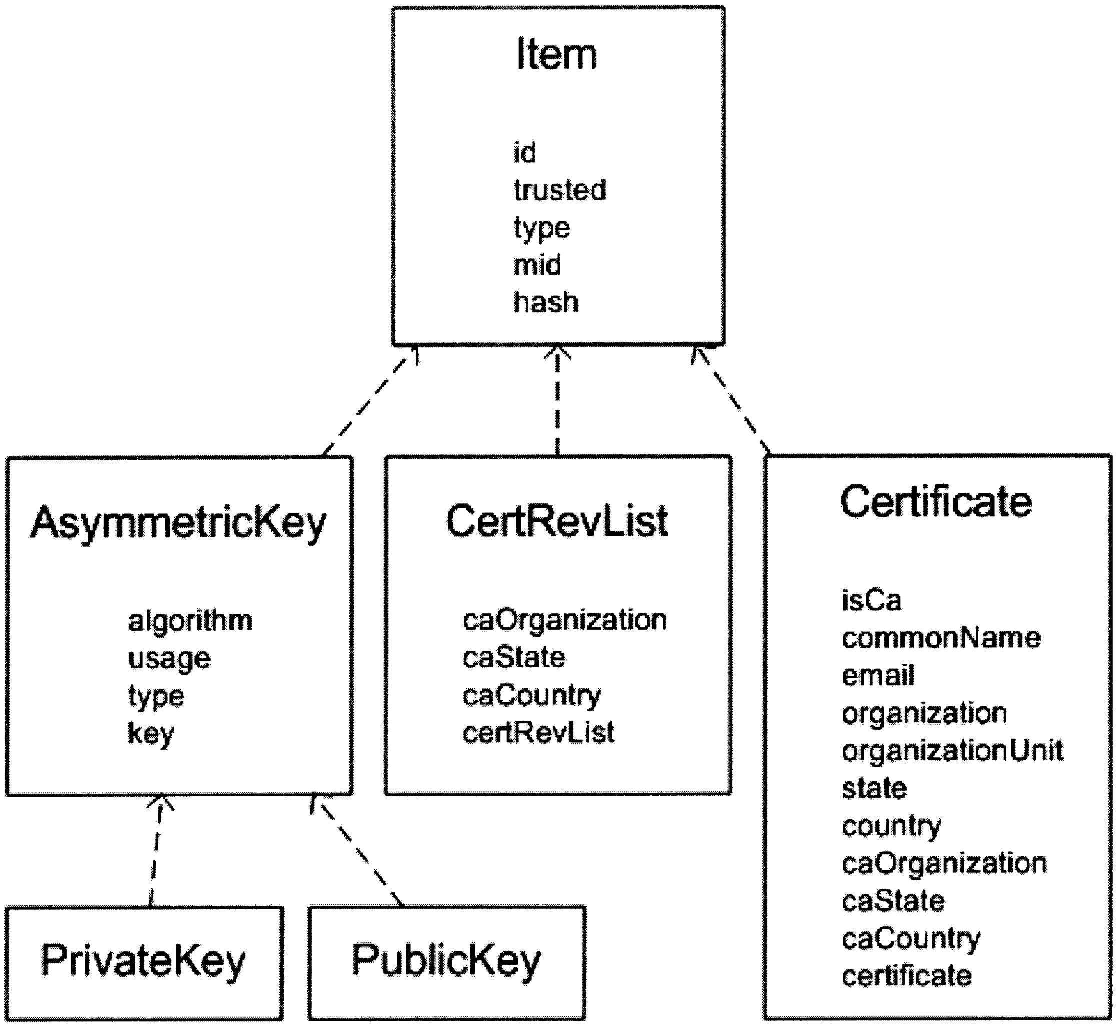


Figure 4-1: Item Inheritance

Items also have a few virtual functions. First, they have constructors to initialize their specific member fields. Second, they have destructors to properly delete the member fields which are allocated on the heap. Finally, FORMSQL, BINDPARAMETERS and SELECTEDITEMS handle SQL statement generation, binding arguments and returning results from a SELECT statement, respectively.

- FORMSQL is called when making a query to the database. It returns a SQL statement based on the member fields of that particular Item. It takes one parameter which tells the function which kind of query is being prepared (INSERT / UPDATE / SELECT / DELETE). For example, if we wanted to generate a SQL statement to select the certificate which is untrusted with ID=5, we would just create a new certificate, set its ID to 5, and set its trusted field to "false". Then we would call FORMSQL on that item, and it would produce a string like "SELECT * FROM certificates WHERE id=? AND trusted=?;". FORMSQL is straightforward, with one exception: handling MIDs (see Section 4.4.3). Since SQLite provides a safe method for constructing SQL statements and all executed SQL comes directly from our code (as opposed to user input), we avoid the risk of SQL-injection attacks.

- BINDPARAMETERS is called to fill in the arguments in the SQL statement that was just created with FORMSQL. In the example detailed above, BINDPARAMETERS would bind the integer 5 to the first variable, and `false` to the second variable. Since the same Item is used as the basis for FORMSQL and BINDPARAMETERS (and presumably left unchanged between the two function calls), the variables and their values are always going to match up.

- SELECTEDITEMS deals with the results of executing a SELECT query. Unlike the other three types of queries (INSERTs, UPDATEs and DELETEs), SELECTs return possibly multiple results. The SELECTEDITEMS packs these results into new appropriate Item objects, and sets these objects' member variables according to the corresponding rows in the Item's database table. If a column is NULL, then that variable does not get set. These new Items are returned in a single ACE_DLList, which is a doubly-linked list implemented by the ACE library (see Section 4.2).

Our design decouples our logic from the actual schema of the database tables. As a result, only a limited amount of code must change to support a change in the database schema. For example, in order to add a column in the table for a private key, a single field needs to be added to the PrivateKey struct, and a couple lines must be added to the PrivateKey implementations of FORMSQL, BINDPARAMETERS, and SELECTEDITEMS. As another example, if we want to add a new kind of cryptographic object, we only have to design the new database table, write a corresponding derived struct of Item, and implement the FORMSQL, BINDPARAMETERS, and SELECTEDITEMS functions.

4.4.2 Entities

Entities are fundamentally different from Items in terms of what each represent. Entities represent individual people, groups of people, or software agents. Still, the Entity objects share many fields in common with the base Item struct. Entities have ID, MID, hash and hashSize fields. They also have name and email fields. However, the database treatment of Entities is very similar to Items, so Entities also have FORMSQL, BINDPARAMETERS and SELECTEDENTITIES methods.

4.4.3 MID

The purpose of the MID is to provide a short name for a particular item (explained in further detail on page 25). The current implementation has the `MID_SIZE` set to 4 bytes. As described earlier, the MID was originally intended to be used as a database ID.

How do we use a MID to identify a particular certificate? We wrote a SQLite extension that compares two BLOBs and returns 1 if one of the BLOBs is a prefix of the other. So, by comparing the MID with the stored full hash, we can find all the potential matches (rarely more than 1 - see page 32).

4.4.4 Bindings

Bindings are primarily used to associate Entities with their Items, and so the `bindings` table is a many-to-many relation. If there were only one type of Item, the bindings table could simply consist of a foreign `entity_id` key and a foreign `item_id` key. However, since the items are spread out amongst many tables, we need an `item_type`

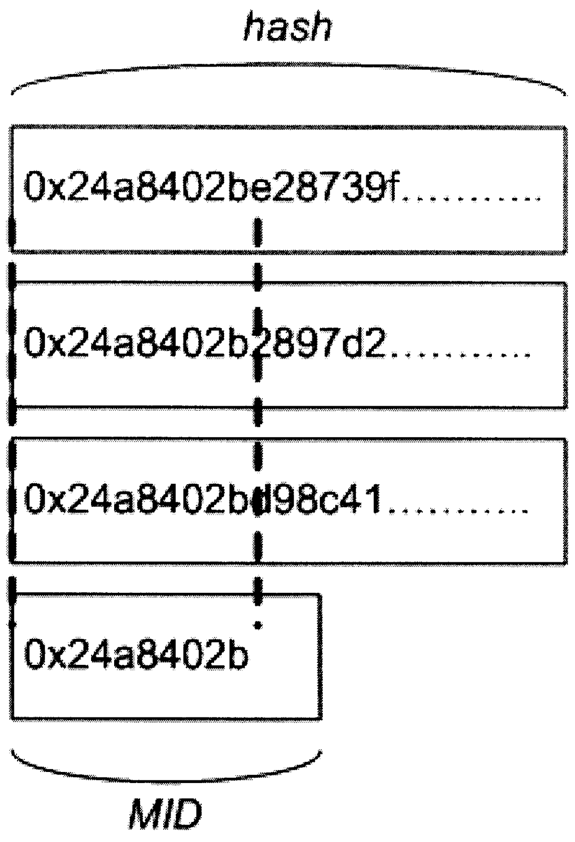

hash
0x24a8402be28739f...........
0x24a8402b?2897d2...........
0x24a8402bd98c41...........
0x24a8402b
MID

Figure 4-2: Colliding MIDs

column to indicate the table to which `item_id` column refers. For example, "alice" could be bound to a certificate, a public key, or a symmetric key.

The DB Bindings API consists of only four functions: an insert function (INSERTBINDING, an update function (SETPRINCIPAL), and two select functions (GETPRINCIPALID and GETBINDINGID).

- INSERTBINDING simply adds a row representing a binding to the bindings table.
- SETPRINCIPAL allows us to designate a particular binding as "principal" (default). If a user has multiple certificates, we use the principal flag to indicate which certificate should be used for that user.
- GETPRINCIPALID gets the principal `item_id` of the given `item_type` for a specified `entity_id`.

- GETBINDINGID simply gets the local database's primary key for the binding. It has several uses: (1) it is used before inserting to check that a binding is not already in the bindings table, (2) it is used to uniquely refer to a particular binding (see Section 4.4.5), and (3) the binding ID is used for initializing any given application (see page 29).

4.4.5 Metadata

Metadata is defined as information about who might have my cryptographic items/bindings: it is stored to allow for significant performance gains in the ICE protocol (see Section 3.4). There are two database tables storing metadata: `sent_to_cert` and `pending_binding`. This metadata contains information on to whom we have sent our cryptographic items (see Section 3.2.1). For each of these tables, there is a pair of functions essentially amounting to accessors and modifiers.

For the `sent_to_cert` table, all of the data for the columns are passed to the NOWHASMY function, which simply stores the information in the database. The underlying logic can be expressed in a simple, though grammatically incorrect sentence. "The owner of certID nowHasMy itemID of type itemType true/false, and nowHasMy bindingID true/false." While the typical case involves sending both the item and binding simultaneously, the Booleans referring to the item and binding allow us flexibility to revise our metadata in the case of unexpected failures. The GETHASMY function retrieves one of the Booleans referring to either the item or the binding.

The `pending_binding` table is much simpler, and the SETPENDINGBINDING either inserts or deletes a row with the given `entityID` and `bindingID`. The GETPENDINGBINDING returns true if there is a row with the given `entityID` and `bindingID`.

4.5 CryptoLib

In order to store and use cryptographic items, we clearly need a library that provides some tools for manipulating them. We chose to use OpenSSL [17] over Mozilla's

Network Security Services (NSS) [14] because OpenSSL makes using elliptic curve cryptography (ECC) easier. Also, the fact that OpenSSL builds on Windows with very little hassle is a big advantage. Not only are OpenSSL's commandline applications easy to use, it has a convenient API to its cryptography library. Finally, ICM will be incorporated into a larger project, which already uses OpenSSL.

In order to limit the effects of our choice of OpenSSL, we perform all the necessary cryptographic operations in this module. Similarly, all the SQLite operations are contained in the DB layer.

The CryptoLib module has three main sets of functions. The first set of functions consists of little more than shallow wrappers of underlying database functions. The second set of functions performs some of the necessary cryptographic operations involved in using X.509 certificates. Finally, the last set of functions performs conversions between various data types. The remaining functions simply set the private member variables `ocspURL`, `myPrivateKey`, and `myPKey`.

4.5.1 DB Operations

The database operations are fairly straightforward. Three functions simply add cryptographic items when given a path (ADDRSAPRIVATEKEYPATH, ADDXPATH, ADDCRLPATH). These are useful for the Pidgin plugin (see Section 4.8) and for the commandline administrative tool. ADDX and ADDENTITY are also fairly similar to their DB counterparts (INSERTITEM and INSERTENTITY), but they also make sure to set their own hashes before calling their respective DB functions. NOWHASMYCERT and GETHASMYCERT are also wrappers of DB functions (NOWHASMY and GETHASMY metadata functions). Finally, the rest of the database operations functions are used only internally by other CryptoLib functions. Instead of making them private though, we made them public to provide a more complete, if slightly redundant API.

4.5.2 Cryptographic Operations

The cryptographic operations are the most important and interesting functions in this API. First, we have the msgSign and msgVerify functions. These calculate and verify signed message digests. Then, we have a couple of functions to calculate hashes and MIDs (bits2hash and bits2MID). Finally, we have two functions to validate X.509 certificates.

Digital Signatures

For the msgSign, we use the SHA-1 hash function to create the message digest, and then we sign the digest with the private key (set with the setMyPrivateKey function). In the msgVerify function, we take in an X.509 object, extract its public key, and use it to verify that the message digest does in fact correspond to the message that we have received.

Hashes and MIDs

bits2hash and bits2MID both calculate the SHA256 hash of their given bitstrings. bits2MID actually calls bits2hash and simply returns the first few bytes of the result.

Certificate Validation

Finally, we have the X.509 validation functions. validateCertificate takes in a `Certificate` and returns true if the `Certificate` is considered valid. First, we start building a "chain of trust" by adding to a data structure all the certificates in the database that claim to be Certificate Authorities (CAs). If the certificate being validated claims to be a CA, we add it as well. OpenSSL takes care of constructing the chain of trust from the certificates we add to its `X509_STORE_CTX`.

If a certificate is self-signed, we do not check any Certificate Revocation Lists (CRLs). Otherwise, we add all CRLs that share the same CA as the certificate we are checking. We search the database for the CRLs that share the same organization, state and country as the certificate we are validating. When we actually run the OpenSSL validation, the function checks the Certificate to make sure that the current date falls between the `notValidBefore` and `notValidAfter` fields.

The second part of validation (VALIDATECERTIFICATEOCSP) involves using the Online Certificate Status Protocol (OCSP) (see Appendix A.3). First, we form an OCSP request using a hash function, the certificate and its certificate authority. Then, we pack and send the OCSP request to the URL set previously (`ocspURL`). If there is no URL set, then OCSP validation is not performed. We then wait for an OCSP response. There are only a few valid responses, and the only ones that we care about are "good" and "revoked". "good" means that the certificate is still valid, while "revoked" means that the certificate used to be valid, but for some reason no longer is. The rest of the responses are all interpreted as indicating that the certificate is not valid. We used the built-in OpenSSL OCSP client function.

If an error occurs in any part of the validation process, the user is notified that the certificate should not be trusted.

4.5.3 Conversions

The conversion functions are mainly focused on manipulating certificates. Certificates are found in three different forms in the system: `X509`, `Certificate`, and DER binary format. `X509` is the C struct used by OpenSSL's cryptographic libraries. Certificate is our struct which is a subclass of Item. It has many of the fields commonly found in certificates like commonName and organization. Finally, DER binary format is sometimes used to store certificates as files, and it's also used to transport certificates over the wire.

BUF2X and X2BUF are more error tolerant forms of OpenSSL's D2I_X509 and I2D_X509 functions. They convert certificates stored in a buffer in DER format into `X509` objects and back, respectively. They are sure to manage memory, and check for errors that may have silently occurred during one of the OpenSSL function calls. C2X converts a `Certificate` to an `X509` struct by simply calling BUF2X on the `Certificate`'s DER buffer.

X2C is the most interesting of the conversion functions. It converts an `X509` struct into a `Certificate` by populating all of the `Certificate`'s fields. The conversion steps are:

- Populate the certificate field by converting the `X509` into a DER buffer;
- Pull out all of the fields (e.g. commonName, email, organization, state, country, etc.);
- Extract the information about the X509's Certificate Authority;
- Set the hash of the public key and corresponding MID.

The last of the conversion functions is CERTMID2CERTIFICATE which is simply a wrapper that tries to find a `Certificate` in the database that matches the specified MID.

4.6 Identity and Certificate Exchange Protocol

In this section, we detail the ICE message format and ICE functions (found in the Exchange module).

4.6.1 Message format

We designed the ICE message format to be as compact as possible to minimize communication overhead (see Figure 4-3). The protocol involves two different types of messages:

- a message containing the sender's certificate and identity/username, and
- a message containing some plain text (email, instant message, etc.).

Our message format accomodates both message types. The format has 13 fields, but is quite compact. The first five fields are combined to use a single byte of space. These are the message type (2 bits), the version (2 bits), certificate-attached (1 bit), recipient (2 bits) and a request flag (1 bit). The message type is named to distinguish among the two types: a certificate-identity message and a text message. We use two bits just so that this format is a little flexible to include future types. The version is the version of the protocol, which again allows for some future flexibility. The certificate-attached bit indicates whether there is a certificate or just a MID from the sender. The recipient field indicates what the sender knows about the recipient. Either the sender knows nothing, or the recipient's username, or both the recipient's username and the recipient's certificate. Finally, the request flag is set to `true` if the sender wants the recipient to send the recipient's certificate to the sender. We crammed all these fields into the first byte of the message so that the recipient can parse this single byte, and have a good idea of what to expect in the rest of the message.

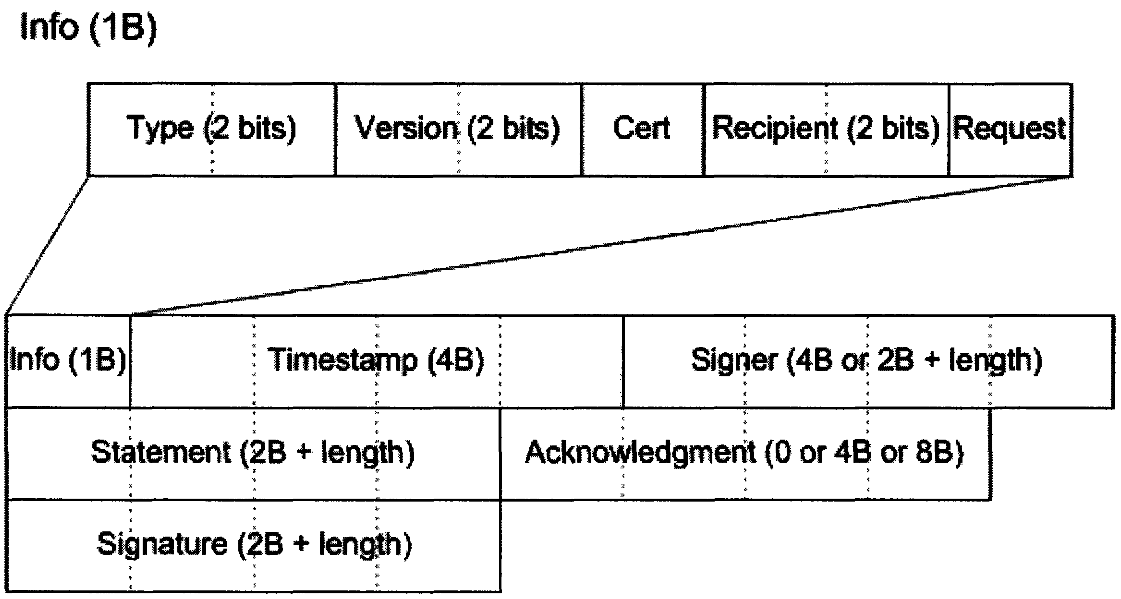


Figure 4-3: Message Format

The next field is simply a 4-byte timestamp which allows the protocol users to protect themselves against replay attacks (assuming their clocks are reasonably synchronized). While our current implementation does not make use of the timestamp, we may defend against replay attacks with the timestamp in the future. The next two fields are the equivalent of an email's "From:" field, so we call them the signer fields. The first signer field is 2 bytes indicating the length in bytes of the next field. The second field is either the sender's certificate in DER format, or the MID of the sender's certificate. Since the MID is fixed at 4 bytes, the size field is completely omitted. The 4 byte MID can be distinguished from a 2 byte size field and the first 2 bytes of a certificate because of the certificate-attached bit in the first byte of the message.

The following two fields are collectively called the statement. The first is again 2 bytes describing in bytes the length of the second field. The statement can be viewed as the payload of the message; in a normal text message, the text would be in the statement. In a certificate-identity message, the statement is the sender's username.

The next field is the equivalent of an email's "To:" field. However, in our protocol, we use it to acknowledge the information that we have about the recipient, so we call it the acknowledgement field. The acknowledgement field is described by the two bit recipient field in the first byte. If the recipient field is 0 (we are broadcasting), then the acknowledgement field is empty. If the recipient field is 1 (we know the recipient's username), then the acknowledgement field is 4 bytes long containing the MID of the username. Finally, if the recipient field is 2 (we know the recipient's username and have the recipient's certificate), then the acknowledgement field is 8 bytes long containing the MID of the username and the MID of the recipient's certificate.

The last two fields contain the cryptographic signature of the message. All of the previous contents are signed, and this signature (consisting of 2 bytes indicating the length of the signature and the signature itself) is appended to the end of the message. Every single message in the protocol is signed. By signing a certificate-identity message, the recipient can verify that the sender does indeed have the private key corresponding to the enclosed certificate. Also, the sender's username is signed, forming a cryptographic binding between the certificate-private key pair and the username.

4.6.2 Exchange module

The exchange module implements functions necessary to perform the ICE protocol. Since the protocol is rather simple, there only six functions in the API. Three of these functions are devoted to constructing messages. There is CREATECERTIDMSG which builds a certificate-identity message (containing the user's certificate and username). While this may sound simple, there is a lot of logic (see Section 3.4.3) to use and store state with the goal of reducing the size and number of communications. If the stored state indicates that the recipient already has this user's certificate, then a null message is returned and nothing will be sent. There is another similar function called CREATECERTIDMSGGroup. This function performs similar functions to CREATE-CERTIDMSGGroup, but does so for multiple intended recipients. It is useful when trying to broadcast out a certificate in a chatroom for example. If all of the recipients are thought to already possess the certificate, then again, nothing will be sent. The last similar function is CREATETEXTMSG which creates a signed text message.

The next two functions are descriptively called MSGPACK and MSGUNPACK. MS-GPACK produces a byte string from a Msg object by simply concatenating the fields in the proper order. MSGUNPACK does the exact opposite, but carefully checks to make sure that the input byte string is long enough to contain the expected fields. These packing functions take into account the local architecture's byte order by using `htonl` and `ntohl` (host to network long and network to host long). These functions make sure that the message's integers are sent in network byte order.

On the recipient's end, RECEIVEMSG handles the byte string by first calling MS-GUNPACK. After a `Msg` object is produced, the signature is verified to ensure that the contents are unaltered and that the message was produced by its proclaimed sender (holder of the corresponding private key). If the recipient realizes that the message was signed by an unknown certificate, then a certificate request with the request flag set is returned to be sent to the original sender. If the message received is a text message, then the contents are returned. Otherwise, the certificate is stored, and the recipient's certificate may be sent if the sender is known not to already possess it.

4.7 Identity and Certificate Manager (ICM)

The Identity and Certificate Manager module presents a single API to applications using the system as the back-end for the Identity and Certificate Exchange protocol. All of the functions are designed to allow an application developer to easily take advantage of the entire system's functionality. With the exception of the portable ACE data types, none of our internal data types are exposed. The functions are mostly thin wrappers around Exchange and CryptoLib functions. The only exceptions are CREATETEXTMSG, its sibling functions; CREATECERTMSG and CREATECERTMSG-GROUP, and their counterpart RECEIVEMSG. These all do packing and signing for the application developer. By abstracting all the lower level details away, we simplify the application developer's job.

4.8 Pidgin Plugin

Pidgin is the new name of the open-source instant messenger client formerly known as Gaim. Pidgin is designed to be easily extended by providing an extensive API available to plugins. We decided to develop a Pidgin plugin to use and test our ICM library and ICE protocol.

We do not allow users to have aliases for other users or chats, because we often use the name of the conversation to retrieve the username. Aliasing prevents those from necessarily being equivalent. In our plugin, we often need to make use of the "name" of a particular IM or chat conversation, which can be masked by a user-specified alias.

4.8.1 C wrappers

Because Pidgin is written in C, we wrote thin C wrappers around our C++ class methods (see Appendix C for an example).

4.8.2 Base64

The MSGPACK function returns a binary string which is not even necessarily null-terminated. In fact, the byte string may have one or more null characters in it. Because XMPP requires text-only strings, we encode the message in Base64 format. Base64 uses lower case and upper case letters of the alphabet, the ten numerals 0-9, and the '+' and '=' characters to encode any binary message. Of course, this encoding increases the size of the message by 33% because three bytes (24 bits) gets encoded as four bytes. Each character takes up one byte, but there are only 64 (2^6) possibilities per character, so only six bits of information can be stored. Base64 is frequently used to encode binary objects when only text characters are allowed.

4.8.3 Signals

Pidgin has an extensive selection of callbacks (Pidgin calls them signals), which makes event-driven programming quite simple. In the plugin initialization function, callback functions of particular types are linked to signals. Signals range from "conversation-created" and "chat-joined" to "quitting" and "playing-sound-event". However, only a handful of these signals is actually useful to the ICE protocol.

IM signals

The signals `sending-im-msg` and `receiving-im-msg` are used for running the ICE protocol over an instant message conversation between two users. The `sending-im-msg` signal is emitted after the user tries to send a message in an IM window. It gives the plugin a chance to replace the outgoing text with whatever the plugin wants to print. In our case, we can take the text and create a text message with a signature, before passing it on to the recipient. Also, if we think the recipient does not have our certificate and/or username, we send our certificate-identity message right before our text message. On the recipient's side, the `receiving-im-msg` is emitted right before displaying the contents to the user. Again, this signal allows the plugin to replace the incoming text with anything. This is where we call RECEIVEMSG, and in the normal case, verify the message's digital signature, extract the message text, and display it in the IM window.

Chat signals

In the design described in Section 3.5, we attach a callback to the `conversation-created` signal which would send out a chat certificate-identity message if any of the users in the chat did not have this user's certificate or username-certificate binding. Our final design ended up being rather different.

ICM is unable to obtain the list of the users in the chat as soon as Alice joins. When the chat is created (`conversation-created`), the Pidgin C-structure contains an empty list of users. Even after the `chat-joined` signal, the list of users is still unpopulated. Soon after, `chat-buddy-joined` signals are emitted, one for each user in the chatroom. There is no way for the plugin to know when it has a complete list of the users in the chat. Therefore, we are forced to choose between processing each user individually (as a callback to `chat-buddy-joined`), or to wait until Alice wants to send a message before initiating the ICE protocol. We chose the latter, because processing each user individually would certainly incur more overhead.

We used a callback for the `sending-chat-msg` signal. When Alice joins a chat, ICM actually does nothing. ICM only sends out Alice's certificate-identity message to the chatroom if at least one user in the chatroom needs it. However, ICM waits to send out Alice's certificate-identity message until right before she is about to send a normal text message to the chat.

On the receiving end, the plugin reacts to the `receiving-chat-msg` signal. If necessary, the plugin automatically replies to a chat message via IM. For example, suppose Dan and Fred are in a chat when Alice joins. Alice's plugin sends her certificate-identity message, as well as a text message. If Dan's plugin needs to reply with his certificate-identity, it IMs Alice directly. There is no need to reply to the chat, because Fred's plugin presumably already has Dan's certificate.

One of the main disadvantages of our design is apparent when using a low-bandwidth link. If Alice wants to send her first message, she must first send a much larger message to the chat containing her certificate. Then, she can send her text message. The overall latency of sending her text message, when preceded by her certificate-identity message, can be increased significantly.

Alternatively, we could have implemented the plugin to wait several seconds after joining before trying to retrieve the list of users. Then, the plugin could determine whether to send the user's certificate to the chat.

There are several Jabber-specific [11, 30] aspects to our implementation. For example, Jabber allows users to choose a "handle" when entering a chat. By default, a user's handle is their username, as registered with the server. When a message is received in a chat, the "sender" is chatname@server/handle. Usually, in an IM message, the format of the "sender" is username@server/resource.

> For example, suppose Alice's and Bob's usernames are Alice@server/Home and Bob@server/Home. Let Alice and Bob be in a chatroom (CoolKids@server), with handles Alice and Bob, respectively. Bob sends a message to CoolKids@server/Alice, the server will translate the recipient and route it to Alice@server, but the "sender" will be CoolKids@server/Bob. However, if Bob sent an message to Alice@server/Home, then the "sender" would be Bob@server/Home.

For these reasons, we currently do not support handles in this format. We assume that a user's handle is their username as well. In the future, we may create a binding between the handle and the certificate.

Another Jabber-specific quirk that we had to deal with is chat history. The Jabber server can be configured to send all or some of the messages from the chat to the user who is just joining. This feature makes certificate exchange very easy, but is a poor model for limited communication. In our tests (Section 5.2), we turned the chat history feature off.

4.8.4 Pidgin GUI/Preferences

Pidgin uses the GIMP Toolkit (GTK) to display its graphical user interface (GUI). Pidgin makes it pretty easy to load and unload plugins, and also to provide a configuration GUI. Preferences, specific to a particular plugin, can be stored in an XML file along with other Pidgin preferences. These allow us to store any state that we want, but we use a database to store most of our state anyway. The two preferences that we do store are a path to the database, and a URL to query for OCSP responses. These preferences can be altered in the ICM configuration window.

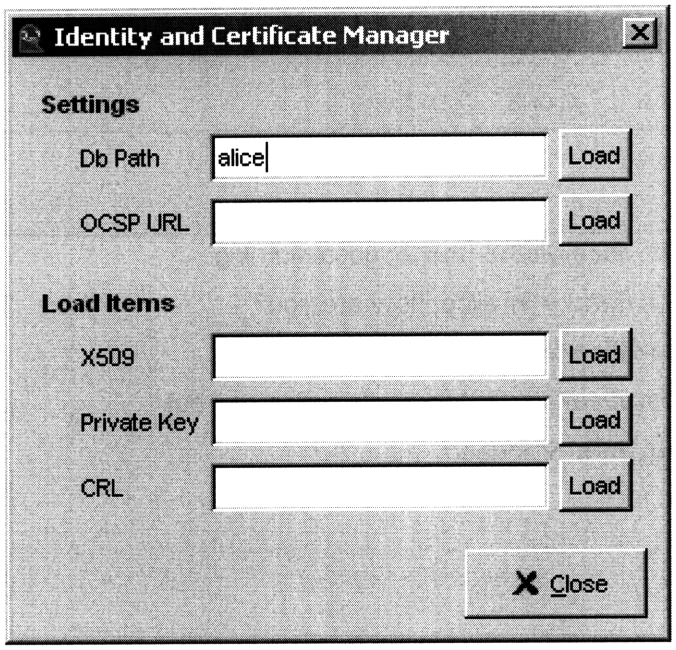
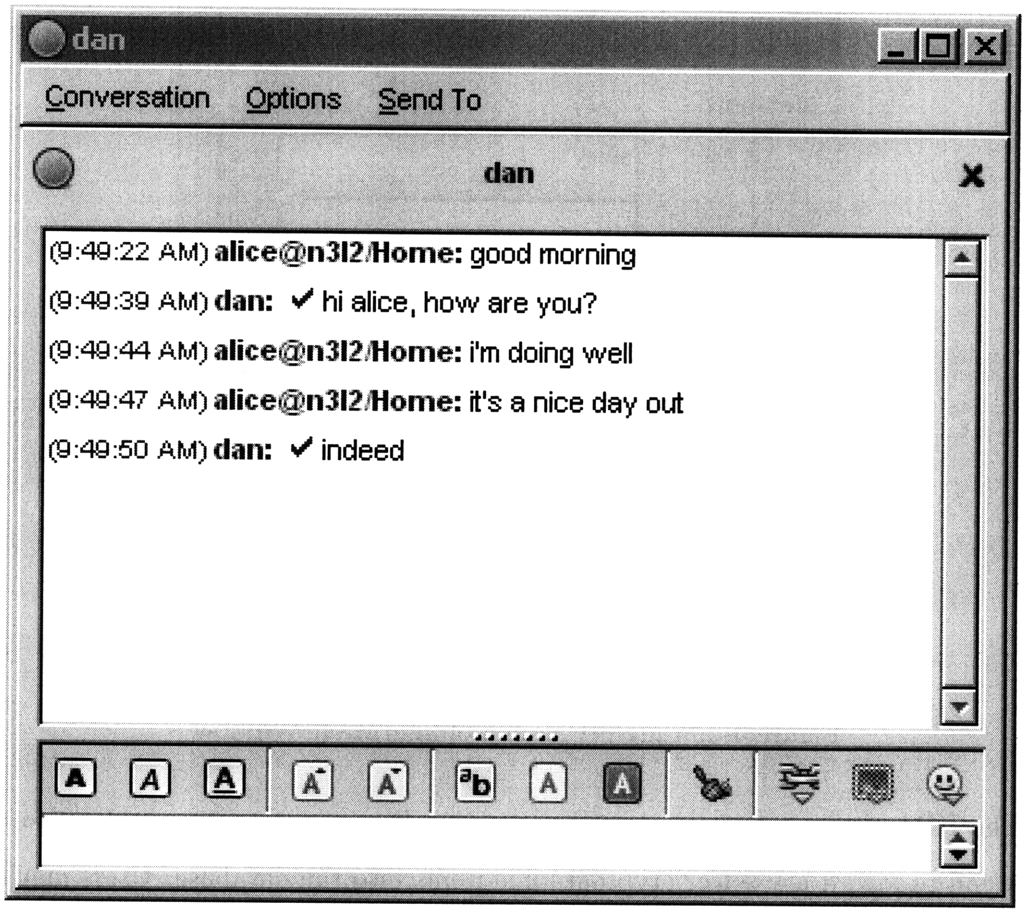


Figure 4-4: Pidgin Plugin Configuration Window

The ICM configuration window (as shown in Figure 4-4) also provides a simple front-end to load a few select cryptographic items into the database. Users may add X509 certificates, RSA private keys and certificate revocation lists via our GUI. These are added by typing in a path to the desired item.

We thought that indicating to the users whether a message has been verified would be very useful. A similar problem is solved by popular web browsers by showing a lock icon when using TLS [27]. At first, we considered simply coloring the text in the IM/chat windows. This would be extremely easy to implement since the text is encoded in HTML. Green text could be verified, while unverified text could be red. The problem with this approach is that these colors could be faked by the user changing their text to green (or red). One alternative that we came up with was to include a small image with verified messages. Images can only be sent between users in Pidgin by establishing a "direct connection". Plugins have another way to insert images, so malicious users would have a lot of difficulty spoofing a verified message. See Figure 4-5 for Alice's conversation with Dan. Further improvements to UI-security notifications will be the subject of future work.



(9:49:22 AM) **alice@n3l2/Home:** good morning
(9:49:39 AM) **dan:** ✔ hi alice, how are you?
(9:49:44 AM) **alice@n3l2/Home:** i'm doing well
(9:49:47 AM) **alice@n3l2/Home:** it's a nice day out
(9:49:50 AM) **dan:** ✔ indeed

Figure 4-5: Sample Pidgin Instant Message Conversation

4.8.5 Initialization

There are a few things that are done by the plugin, and that need to be done by the user before the ICM plugin can be used properly. First, the plugin checks to see whether there is a path to the database in the preferences file. If there is a path, the plugin opens the database. However, if there is no path, the plugin simply creates a database in the current directory named with the account's username by loading the schema from a file. This default behavior is also reflected by setting this path in the preferences file.

The user has to load at least one certificate and one private key, obtained from an appropriate authority. Without these two items, a user cannot participate in authenticated messaging. The items may be loaded in any order, with the exception that a CA-signed certificate should be loaded after its signing CA and the corresponding certificate revocation list. That is, the CA and CRL should be loaded before the other certificate, so that the second certificate can be verified properly. Also, the last certificate to be loaded via the configuration window is the certificate that is used in the ICE protocol. Similarly, the last private key loaded is used to sign messages in the ICE protocol. In one of our tests, we used two certificates, one private key, and one certificate revocation list. One of the certificates was a self-signed certificate authority. The other was a user certificate signed by that CA.

4.8.6 Multiple accounts

Pidgin can support multiple account simultaneously, even using different protocols. However, ICM currently only supports a single application's notion of identity at a time. Therefore, if a user is logged into multiple accounts in the same application, then ICM is unable to know which identity to exchange with other users.

There are two possible solutions to handling multiple accounts in the future. First, an ICM instance can be created for each account, but that would greatly increase the complexity of the plugin. Another option is to modify ICM to handle a list of accounts, and pass information about which account is being used to every function.

Chapter 5

Evaluation

In this chapter, we discuss our testing and evaluation of the ICM system and ICE protocol.

5.1 Testing - Module

We implemented the ICM system from the bottom up, writing and performing unit tests along the way for each of the three major modules (DB, CryptoLib and Exchange). We simply wrote executables that would cover all of the different functions.

5.1.1 `testdb`

The DB test executable (`testdb`) is script-like in that it simply creates a new database and performs different operations on it. All of the DB layer functions perform one or more INSERT, SELECT, UPDATE or DELETE operations. Naturally we call functions to insert rows into the database tables before selecting/updating/deleting them. The tests output debugging feedback to indicate which tests have passed successfully. To determine whether a test was successful, we hardcoded the expected results, and matched them with the output of the SELECT functions. We did not perform any thread safety or multiple access tests.

5.1.2 `testcryptolib`

The CryptoLib test executable (`testcryptolib`) is similar to testdb in that it tests most of the functions individually. There is some set-up required before running `testcryptolib`.

Setup

One of the main purposes of the CryptoLib module is to manipulate and validate certificates. In order to test certificate handling, we generated our own certificates, complete with a certificate authority and certificate revocation list. See Appendix B for OpenSSL commands to generate these test files.

In order to test OCSP validation (see Section 4.5.2), we need to run an OCSP responder. OpenSSL has a simple OCSP responder that can be run as described in Appendix B. This responder uses files created in the process of generating the certificate authority (the CA's certificate, private key and index). The index contains information (including validity status) on all of the certificates signed by the CA.

Testing

The order in which certificates are inserted into the database by the CryptoLib module is important. Every certificate is validated before inserting it into the database. Therefore, we insert the CA's certificate first so that the subsequent certificates' signatures can be verified. Also, we insert the CRL before the CA-signed certificates. The CRL is also used when verifying whether a certificate is valid. We tested inserting a valid certificate, a revoked certificate, a valid certificate while the OCSP responder was unavailable. Then we performed a few sanity checks. For example, we verifyed that the same certificate was the same length in DER format when stored in two different databases.

5.1.3 testexchange

The Exchange test executable mainly checks the logic of the functions implementing the ICE protocol. We tested creating a number of typical common-case messages, as well as all sorts of error-inducing messages. We make sure that MSGPACK can take a Msg and produce a bytestring that the MSGUNPACK function can use to reproduce the original Msg.

Most of the ICE protocol logic resides in the CREATECERTIDMSG and RECEIVEMSG. Therefore, most of the testexchange tests focus on these two functions.

We performed four tests on CREATECERTIDMSG by simply varying the three parameters to cover the different parts of the code. First, we provided no inputs, but had the request flag set to true. This message is the response when we receive an ICE message for which we do not have a certificate. We know neither the sender's certificate nor identity, so both parameters are NULL. The second test involved knowing both the recipient's identity and certificate. This is a common case where the recipient has just sent us a certificate-identity message. The third test involved knowing the only the identity of the recipient. This might happen if the we join a chat and see only one user that does not have our certificate. Finally, the circumstances of the last test were similar to the previous one's. We only had the recipient's identity, but we happened to have a corresponding certificate in our database for that identity. This happens when we see a user in a chatroom for which we have a certificate, but for some reason we have not sent our certificate to them.

5.2 Testing - System

Our system evaluation tests checked for protocol correctness (expected behavior vs. observed behavior), and performance overhead. The performance metrics cover the number and size of the messages that we send, and compare them to the messages produced without using the ICE protocol.

In the following sections, we describe tests for IM scenarios and chat scenarios. We constructed these scenarios as combinations of username changes, certificate changes, and database failures. Often, multiple different scenarios result in the same behavior from the ICE protocol. We also describe how many bytes are saved in each scenario assuming a single text message from each user. We will compare these results to an S/MIME scheme that attaches a certificate to every message (see Appendix A.4).

All of the numbers in the following tests do not include overhead associated with XMPP. All of the messages, both ICE and S/MIME, are Base64 encoded, so in that respect, they are identical. The certificates used contain 1024-bit RSA keys, and when we refer to Alice's certificate, both in the context of ICE and S/MIME, we are actually referring to the same X.509 certificate.

5.2.1 Scenarios - IM

In all of the following scenarios, Alice is trying to send a message to Dan, but one or both of them may not have the other's certificate. In the Table 5.1, we enumerate ten scenarios. Scenario 1 involves both Alice and Dan without each other's information. Scenarios 2-4 involve one or more changes in username. Scenario 5 is the result of a successful exchange of certificates and bindings. Scenarios 6-8 involve old certificates, and finally Scenarios 9 and 10 involve database failures.

In order to clarify which cryptographic item is being specified, "alice", "notalice", "oldalice", alice@ll, and oldalice@ll refer to the current binding, wrong binding, old binding (associated with the wrong certificate), current certificate and old certificates, respectively. Parallel terms are used for Dan and Fred.

For the Savings column in the following tables, we calculated how many Bytes are saved for a "conversation" in which each user sends a single message "hi". For S/MIME, the "hi" message was 2403B, 2383B and 2371B for Alice, Dan and Fred, respectively. Therefore, the maximum possible savings in an instant message conversation between Alice and Dan is 4786 Bytes. The maximum savings for a chat between all three users is 7157 Bytes.

Note that in Table 5.1, a user never has a binding without a certificate, because the binding is a link between a username and a certificate. Without the certificate, the binding cannot be authenticated. Suppose Alice and Dan had each other's bind-

#	Alice has	Dan has	Observed	Savings (B)
1			"alice"/alice@ll → D A ← "dan"/dan@ll	4786 - 3100 **1686**
2*	"notdan" dan@ll	"notalice" alice@ ll	"alice"/alice@ll → D A ← "dan"	4786 - 1940 **2846**
3*	"notdan" dan@ll	"alice" alice@ll	"alice"/alice@ll → D A ← "dan"	4786 - 1940 **2846**
4	"dan" dan@ll	"notalice" alice@ll	"alice" → D	4786 - 604 **4182**
5	"dan" dan@ll	"alice" alice@ll	(nothing)	4786 - 400 **4386**
6	"olddan" olddan@ll	"alice" alice@ll	msg → D A ← "dan"/dan@ll	4786 - 1760 **3026**
7	"dan" dan@ll	"oldalice" oldalice@ll	msg → D A ← "dan"/dan@ll "alice"/alice@ll → D	4786 - 3104 **1682**
8	"olddan" olddan@ll	"oldalice" oldalice@ll	msg → D A ← "dan"/dan@ll "alice"/alice@ll → D	4786 - 3104 **1682**
9		"alice" alice@ll	"alice"/alice@ll → D A ← "dan"/dan@ll	4786 - 3100 **1686**
10	"dan" dan@ll		msg → D A ← "dan"/dan@ll "alice"/alice@ll → D	4786 - 3104 **1682**

Table 5.1: IM Scenarios

ings and certificates, but Alice gets a new certificate. Dan would not have the new certificate, nor the new binding between the name "Alice" and Alice's new certificate. Therefore, the situation is the same as Dan having neither Alice's certificate nor her binding. In practice, the only difference is that a warning is shown to Dan alerting him to the fact that he had previously had a different certificate for Alice's username.

The scenarios marked with an * are special because the observed behavior is different from what we might expect. alice@ll is sent along with "alice", because Alice cannot associate "dan" with dan@ll. She knows that she has sent alice@ll to the owner of dan@ll, but she has a binding for "notdan" and dan@ll. The reverse case does not occur (Scenario 4) because Dan receives "alice", and therefore knows he does not need to send his certificate. In Scenario 9, Alice sends both "alice" and alice@ll because she has no record that dan already has those items.

In Scenario 6, how does Dan know to send Alice his binding and certificate? The answer lies within the acknowledgment field of our message format (see Section 4.6.2). Alice acknowledges the wrong certificate for Dan, so Dan responds appropriately.

5.2.2 Scenarios - Chat

Scenarios for chats are significantly more complicated (and numerous!). There are a few situational aspects that we keep constant. First, there are 3 users in the chat: Alice, Dan and Fred. Second, Alice is always entering the chat last, and sending a message as soon as she arrives. Third, Dan and Fred have already exchanged each other's certificates and bindings.

Scenario 1 is the situation where Alice has never communicated with Dan or Fred. Scenarios 2-9 involve different usernames. Scenario 10 is the result of successful exchange of certificates and bindings among all three users. Scenarios 11-13 involve old certificates. Finally, Scenarios 14-17 involve database failures.

#	Alice has	Dan has	Fred has	Observed	Savings(B)
1		"fred" fred@ll	"dan" dan@ll	"alice"/alice@ll → chat A ← "dan"/dan@ll A ← "fred"/fred@ll	7157 - 4612 **2545**
2	"notdan" dan@ll "notfred" fred@ll	"notalice" alice@ll "fred" fred@ll	"notalice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "dan" A ← "fred"	7157 - 2304 **4853**
3	"notdan" dan@ll "notfred" fred@ll	"notalice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "dan" A ← "fred"	7157 - 2304 **4853**
4	"notdan" dan@ll "notfred" fred@ll	"alice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "dan" A ← "fred"	7157 - 2304 **4853**
5	"dan" dan@ll "notfred" fred@ll	"notalice" alice@ll "fred" fred@ll	"notalice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "fred"	7157 - 2104 **5053**
6	"dan" dan@ll "notfred" fred@ll	"notalice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "fred"	7157 - 2104 **5053**
7	"dan" dan@ll "notfred" fred@ll	"alice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "fred"	7157 - 2104 **5053**
8	"dan" dan@ll "fred" fred@ll	"notalice" alice@ll "fred" fred@ll	"notalice" alice@ll "dan" dan@ll	"alice" → chat	7157 - 756 **6401**
9	"dan" dan@ll "fred" fred@ll	"notalice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	"alice" → chat	7157 - 756 **6401**
10	"dan" alice@ll "fred" fred@ll	"alice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	(nothing)	7157 - 564 **6593**

Table 5.2: Chat Scenarios, part 1

Scenario 1 is probably the most typical scenario; the users already present in the chat (Dan and Fred) have exchanged certificates/bindings, and a new user is joining the chat. Alice recognizes that she has sent her certificate/binding to neither Dan nor Fred, so she sends her certificate-identity message to the chat. Dan and Fred both respond directly to Alice with their respective certificate-identity messages.

Scenario 5 involves two new usernames. Alice has switched from "notalice" to "alice", while Fred has switched from "notfred" to "fred". Alice sees that she has sent her certificate-identity to neither Dan nor Fred, so she sends it to the chat. Fred has not sent his newest binding to alice@ll, so he sends it.

Scenario 7 tests one user changing names (from "notfred" to "fred"). Note that Alice sends her certificate needlessly, but Fred only sends his binding.

Scenario 8 shows off some special case code; Alice recognizes that she has sent her certificate to both Dan and Fred, but has changed her username since then. Therefore, she sends only her new binding to the chat.

Scenario 10 is the simple case of everybody having exchanged information already.

#	Alice has	Dan has	Fred has	Observed	Savings (B)
11	"olddan" olddan@ll "fred" fred@ll	"alice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	A → chat chat ← D "alice"/alice@ll → D A ← "dan"/dan@ll	7157 - 3272 **3885**
12	"olddan" olddan@ll "fred" fred@ll	"oldalice" oldalice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	A → chat A ← "dan"/dan@ll "alice"/alice@ll → D	7157 - 3272 **3885**
13	"dan" dan@ll "oldfred" oldfred@ll	"oldalice" oldalice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	A → chat A ← "dan"/dan@ll "alice"/alice@ll → D chat ← F "alice"/alice@ll → F A ← "fred"/fred@ll	7157 - 5968 **1189**
14	"dan" dan@ll "fred" fred@ll	"oldalice" oldalice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	A → chat A ← "dan"/dan@ll "alice"/alice@ll → D	7157 - 3272 **3885**
15		"alice" alice@ll "fred" fred@ll	"alice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat chat ← D "alice"/alice@ll → D A ← "dan"/dan@ll chat ← F "alice"/alice@ll → F A ← "fred"/fred@ll	7157 - 7316 **-159**
16	"dan" dan@ll "fred" fred@ll		"alice" alice@ll "dan" dan@ll	A → chat A ← "dan"/dan@ll "alice"/alice@ll → D	7157 - 3272 **3885**
17			"alice" alice@ll "dan" dan@ll	"alice"/alice@ll → chat A ← "dan"/dan@ll chat ← F "alice"/alice@ll → F A ← "fred"/fred@ll	7157 - 5968 **1189**

Table 5.3: Chat Scenarios, part 2

Scenario 11 has some strange behavior. Alice thinks she has Dan's and Fred's certificates/bindings, but she actually has Dan's old certificate and binding. However, she sends her message to the chat, without any problems. It is only when Dan sends a message to the chat that Alice realizes she has the wrong certificate for him. She then requests his certificate and binding, and he sends them to her.

Scenario 15 shows how a database failure can cause many extra communications. Alice's database has died, but Dan and Fred believe Alice still has their certificates and bindings. Alice does not know that they already have her information. She sends her certificate/binding to the chat, but receives no responses. When Dan and Fred send messages to the chat, Alice has to request their certificates/bindings individually. If all of the users had universal knowledge, Dan and Fred would have just sent their certificates/bindings to Alice. Instead, there are three extra messages containing Alice's certificate/binding; the initial one to the chat, and the two individual ones to Dan and Fred. The overhead is associated only with the failure-induced "pull" method of exchange; Alice is requesting information directly, instead of Dan/Fred knowing that that she needs their information.

Note how the savings in Table 5.3 are small, and even negative in Scenario 15. These numbers, as well as all the savings numbers for chat scenarios, are misleading when considering a network without true multicast. If Alice, Dan and Fred are all within transmission distance of each other in a wireless network, then they do have true multicast. That is, Alice can send a message to both Dan and Fred for the same price as a message to only Dan (or only Fred). If Alice, Dan and Fred are on a switched network, then at some point, Alice's message to Dan and Fred must be replicated. Therefore, the cost to send to both Dan and Fred is essentially the cost of sending to Dan plus the cost of sending to Fred.

5.3 Performance Evaluation

In the Scenarios discussed earlier in this chapter, we compared the performance of our protocol to S/MIME. Our protocol improves on S/MIME in three ways (two are specific to a wireless network):

- Message Size: we produce smaller messages since certificates are not sent every time;

- Shorter Transmission Latency: smaller messages can be sent faster, usually certificates do not need to be transmitted;
- Lower Power Consumption: fewer/smaller messages require less power to transmit.

It *is* possible to send an S/MIME message without attaching the signing certificate. However, the resulting messages (for text "hi") are all 1197B, as compared to Alice's 2403B, Dan's 2383B and Fred's 2371B. The same message is about 200B in the ICE protocol (no certificate attached).

Calculating the performance gains over a "typical" conversation, or over a given period of time requires some numbers that are not easily obtained. We will assume that a typical conversation involves 25 messages, each averaging 30 characters, transmitted over 5 minutes [33]. Assuming that certificates and bindings are exchanged once in the ICE protocol, a typical conversation between Alice and Dan amounts to 8604B. If Alice and Dan were using S/MIME, the same conversation would be 60525B. If they only exchanged their certificates once, but still used S/MIME for the remaining messages, the conversation size would be 33017B. Applications that do not involve transferring large amounts of data are likely to have certificate exchange constitute a large percentage of overall transmissions.

Chapter 6

Conclusion

In this chapter, we explore possibilities for future work, summarize what I learned over the course of this project, and conclude.

6.1 Future Work

In this section, we discuss areas that could use future development. These are features and ideas that we, in our limited time, did not implement.

6.1.1 Multiple Applications

As explained earlier (page 29), there are tables in the database schema provisioning for allowing multiple applications to use ICM simultaneously. Applications should then be able to initialize themselves properly with their own bindings.

6.1.2 MID/Hash Collisions

We use hashes and especially MIDs to represent different Entities and Items. However, in the unlikely event that there is a collision (e.g. two different Entities with the same MID), we would like our performance to degrade gracefully. Right now, we simply do not handle collisions, and an error is displayed. Handling collisions can be quite simple in some cases, like trying to verify the signature of a message given the signing certificate's MID. All the matching certificates can be tried, until one succeeds. Other cases are likely to be more complex, such as trying to find the certificate associated with a given Entity's MID. If there are two distinct Entities with the same MID, then we may think we have a user's certificate when we actually do not.

6.1.3 Service

We would like a single user's multiple applications to be able to use the same ICM simultaneously. Right now, this is possible by using the same SQLite database back-end. However, we would like to have an ICM service running in the background on the user's machine. Then, applications wishing to use ICM would make requests to and process responses from the ICM service process. The ICM would have to be modified to register and verify application IDs so that application-specific settings are maintained.

Yet another step would be to make the ICM service run on a server, perhaps accessible via the Internet. This would be significantly more involved and dangerous; it would obviously be a major problem if our service gets compromised. A secure scheme to authenticate users to the server would be absolutely necessary. After the authentication problem is solved, an encryption scheme would also be necessary to ensure that eavesdroppers cannot steal items like private keys off the wire. A secure ICM service accessible from anywhere would be useful to users who work on different machines.

6.1.4 OCSP

We allow for optional OCSP validation of certificates. For simplicity, we ask the application for a URL which would handle our OCSP requests. Sometimes, in one of the optional X.509v3 extension fields, there is a URL for the appropriate OCSP responder. A simple extension of ICM would be to parse these fields in search of such URLs.

6.1.5 Thunderbird Extension

We tout ICM as a library that can be used by multiple applications for authenticated communication. However, we only had time to implement one example, a Pidgin plugin (Section 4.8). Writing a Thunderbird extension would be a great way to show ICM's versatility.

6.1.6 Pidgin Plugin

There are several usability features that can be added to the plugin. First, when a user receives a certificate, show that certificate's information in a new GTK window. Allow the user to accept or decline to trust that certificate. Also, when a user is using a "new" certificate, show both the new and old certificates, again with an option to decline to use the new one. However, whether this would improve security/usability depends on the user; many users are accustomed to simply clicking through pop-up dialogs until the application works as expected.

Another feature that might be useful is re-verification of old messages. Suppose Alice sends several messages to Bob before she receives his request for her certificate. Then all of those messages would show up as unverified to Bob. With the new feature, Bob's ICM would be able to verify Alice's older messages.

6.1.7 Browser-based GUI

webpy is a Python-based webserver that would allow us to write a GUI that could be accessed via a user's browser. This GUI would provide a more usable interface to our database than our commandline-based Admin program.

6.1.8 Certificate Retrieval via URL

Suppose Alice uses a personal digital assistant (PDA) and wants to send her identity-certificate to Bob, who is on broadband. One possible extension of our ICM would be to allow users to send URLs in place of their certificates. Then, in our example, Alice could just send a URL for Bob to download her certificate. This would save Alice precious bandwidth and power on her PDA.

6.2 What I Learned

Over the course of this project (designing and implementing ICE and ICM), I learned a lot, from how to use certain developer tools to secure programming practices. I used emacs [8] as my IDE, but I had never used tags [6] to navigate through third-party libaries. This was invaluable, especially when trying to comprehend OpenSSL's cryptography library. Other tools that I became familiar with include Subversion [24] (a versioning system) and Doxygen [5] (a documentation tool).

One of the most important skills that I acquired is writing/editing Makefiles. Using Makefiles made building my project infinitely easier; compiling each source file individually would have been very tedious and time-consuming. I also learned to use GCC from MinGW [13] compiler under the cygwin [3] shell, since we performed most of our development on Windows [28].

The three open-source/free libraries that we used (SQLite [23], OpenSSL [17] and ACE [25]) were also new to me. While the SQLite online documentation was superb, I had to rely on books ([37] and [34]) to get a grasp on the OpenSSL and ACE libraries.

Finally, there were several secure programming practices that I learned to use. For example, to prevent buffer overflow attacks, one must use safe string functions (strncpy, strncat), instead of their unsafe and more common counterparts (strcpy, strcat). Also, when building a SQL query for the database, one must use bind parameters to accept user input, instead of simply concatenating the statement together. Otherwise, the system is vulnerable to SQL-injection attacks.

6.3 Conclusion

In conclusion, we have designed and implemented an Identity and Certificate Manager (ICM) library for storing and using applications' identities and digital certificates. We also designed an efficient protocol, called Identity and Certificate Exchange (ICE), allowing users' ICMs to exchange identities and certificates. We demonstrated the use of both ICM and ICE in by writing a Pidgin plugin.

Appendix A

Public Key Infrastructure (PKI)

Public Key Infrastructure (PKI) is a system designed to simplify authentication (the act of checking to see whether a person is who he says he is, and whether he wrote what he said he wrote). PKI follows a hierarchical trust-based model with each node in the hierarchy representing a certificate. Any certificate that is not a leaf in the hierarchy is called a certificate authority. The certificate authority at the top of the hierarchy is called a root certificate authority.

PKI relies on cryptographic digital signatures. Digital signatures are similar to normal signatures in that they are only easy for one person to create, but anybody can verify them to see that they are authentic. A certificate authority encloses a signature in any certificate that it issues. A root certificate authority signs its own certificate.

The basic concept of a PKI is that trust can be transitive, i.e. if Alice trusts Bob to trust other people, and Bob trusts Charles, then Alice can trust Charles. In this simple example, Bob would be the certificate authority (CA), trusted by Alice. If the CA signs a certificate (like Charles'), that signifies that the CA trusts that Charles is actually Charles (presumably after doing some other kind of authentication). If Alice trusts Bob to verify people's identities, then she trusts that Charles' certificate belongs to Charles.

A.1 Certificates (X.509)

X.509 is an ITU-T standard adopted by the Internet Engineering Task Force (IETF) in RFC3280. There are many useful fields in an X.509 certificate. For example, there's a field called "Subject" which details the person or organization represented by that particular certificate. There are subfields in Subject including common name (often the name of the person), email, organization, state, country, etc. Common names are often used by applications to verify identity. X.509's also have two dates, delimiting the period during which the certificate is considered valid. There is also a field called "Issuer" which has the same subfields as "Subject". The Issuer is the certificate authority which issued, and therefore trusts this certificate. Perhaps most importantly, all certificates enclose a public key. Some public keys are for digital signatures, allowing anyone possessing the certificate to authenticate any message signatures from the owner of the certificate. Therefore, if Charles wants Alice to be able to authenticate her messages, he sends her his own certificate, and signs his messages.

A.2 Certificate Revocation Lists (CRLs)

A certificate whose validity period has not ended yet may need to be invalidated for some reason. An example is if Alice worked for a company which issued her a certificate, but she was fired. One solution is called a certificate revocation list (CRL), which is also specified in RFC3280. A CRL lists certificates that have been "revoked", which means that the certificate authority has invalidated them. Certificate authorities are supposed to generate CRLs periodically. Theoretically, anyone trusting this certificate authority should be checking for new certificate revocation lists often, so that the certificates can be confirmed valid.

CRLs have three main problems. They may become very large files; downloading one could be a lot of overhead if only one certificate needs to be validated. The second problem is that users often forget or do not even bother to check for CRLs.

Finally, for very large CAs, CRLs need to be published very often, since certificates are revoked frequently.

A.3 Online Certificate Status Protocol (OCSP)

Online Certificate Status Protocol (OCSP) is a possible replacement for downloading CRLs, and is specified in RFC2560. An OCSP responder is a server that keeps track of what certificates' statuses are (usually the CA). If Alice wants to find out whether Bob's certificate is still valid, she can send an OCSP request to the OCSP responder, and then receive information only pertaining to Bob's certificate. OCSP requests and responses are very small; only the hash algorithm, issuer's hashed name, issuer's hashed public key, and the certificate's serial number are included for each certificate. The size of an OCSP request is 41B + 64B*n where n is the number of certificates being verified. Two advantages of OCSP over CRLs are that the information gathered is more up-to-date, and no much less extraneous information is transmitted. One disadvantage of OCSP is that it requires the status checking to be done online.

A.4 (S/MIME)

Secure/Multipurpose Internet Mail Extensions is a standard for encapsulating encrypted and signed electronic mail. S/MIME requires the use of private keys and certificates from an appropriate Certificate Authority. Typically, S/MIME is used for generating digital signatures of messages, providing authentication, message integrity, and non-repudiation properties. Authentication gives the recipient information attesting that the message was written by the holder of the private key. Message integrity provides assurance that the message was not tampered with after the signature was generated. Finally, non-repudiation makes it difficult for the signer to deny authoring the message. [22]

Appendix B

OpenSSL

In this chapter, we illustrate the OpenSSL commands used to generate test certificates, keys, CRLs, etc. Please refer to Appendix A for information on any of the aforementioned terms. We used OpenSSL-0.9.8e in this project.

B.1 Creating Certificates

First, we set up our own Certificate Authority (CA). For detailed instructions on how to do this, see [16]. Be sure to download the openssl.cnf file. After we have our CA set up, we go on to create our own private key and certificate pairs:

```
openssl req -newkey rsa:1024 info.txt -keyout key.pem -out req.csr
```

Note: we use RSA 1024-bit keys (yes, admittedly weak). Larger keys can easily be generated, as well as keys using other kinds of cryptography, such as elliptical curve cryptography (ECC).

The above command will produce a certificate request file (req.csr) and a private key file (key.pem). To sign the request and produce a certificate:

```
openssl ca -config openssl.cnf -in req.csr -out cert.pem
```

As might be suspected, the newly signed certificate is cert.pem. In order to verify that a certificate was signed by a given CA, we use the following command:

```
openssl verify -CAfile ca-cert.pem cert.pem
```

This command allows us to establish a chain of trust; i.e. if we trust the CA (ca-cert.pem), then we can trust the certificate (cert.pem).

B.2 Revoking Certificates

Now that we know how to create certificates, we might want to revoke invalid certificates:

```
openssl ca -config openssl.cnf -revoke cert.pem
```

However, just revoking a certificate is insufficient; users need to be notified. One way users can stay up-to-date on the latest revoked certificates is to periodically download certificate revocation lists (CRLs). To generate a CRL:

```
openssl ca -config openssl.cnf -gencrl -out ca-crl.pem
```

Now, we just need to put `ca-crl.pem` in a place where users can easily download it. `ca-crl.pem` contains the serial numbers (unique for a given CA) of revoked certificates.

B.3 S/MIME

S/MIME provides a standard for digital signatures in email, as well as an option for encryption. To create a signed email from a text file named msg.txt:

```
openssl smime -sign -in msg.txt -text -out signed.msg
  -signer cert.pem -inkey key.pem
```

The output is `signed.msg`. Why do we need to include the "`-signer`" certificate? S/MIME includes the signing certificate by default. Of course the private key is needed as well. If we want to explicitly exclude the signing certificate, we simply add "`-nocerts`" as an extra argument to the above command.

On the other end, how do we verify a signed S/MIME email?

```
openssl smime -verify -in signed.msg -CAfile ca-cert.pem
```

The above command only works if the signing certificate is included in the message. If the certificate is not included, then it must be specified by hand, with the "`-certfile`" option.

B.4 OCSP

OCSP is an abbreviation of Open Certificate Status Protocol (see Appendix A.3 for details). To run an OCSP responder, use the following command:

```
openssl ocsp -index index -port 8888 -rsigner ca-certkey.pem
  -CA ca-cert.pem
```

`ca-certkey.pem` is the concatenation of `ca-cert.pem` and `ca-key.pem` (the certificate's private key). The "`-rsigner`" option allows us to sign our OCSP responses, so that the requester can have some assurance that the response was not falsified. The OCSP responder can be run on any port, but in this example, it is running on port 8888.

To send an OCSP request from the commandline, use the following command:

```
openssl ocsp -url http://ocsp.org:8888 -issuer ca-cert.pem
  -cert cert.pem
```

This sends an OCSP request to the OCSP responder running on `ocsp.org` on port 8888 for the certificate `cert.pem`.

Appendix C

Writing C Wrappers for C++ Functions

C++ has classes, while C does not. So then, how do we use C++ libraries from C? The solution involves a few steps, but the main idea is that a void* generic pointer can be used to point to instances of a C++ class. The other important points are to use extern "C" around the C++ function declarations, and to have all the C++ included files in the .cpp file as opposed to in the .h (header) file. Below are a simple C++ class and its header file.

ace_hello.h

```
#include "ace/OS.h"
#include "ace/Log_Msg.h"

class Hello {
 public:
  Hello(ACE_TCHAR* name);
  ~Hello();
  void printHelloName();
 private:
  ACE_TCHAR* myName;
};
```

ace_hello.cpp

```
#include "ace_hello.h"


Hello::Hello(ACE_TCHAR* name) { myName = name; }


Hello::~Hello() {}


void Hello::printHelloName() {
  ACE_DEBUG((LM_INFO, "Hello, %s!\n", myName));
}
```

Note that there is nothing special about these files. Below, we have the C wrappers for this class.

c_hello.h

```
struct hello_st {
  void* hello;
};
typedef struct hello_st cHELLO;


#ifdef __cplusplus
extern "C" {
#endif


  int HELLO_new(cHELLO* cHello, unsigned char* name);
  int HELLO_print(cHELLO cHello);


#ifdef __cplusplus
}
#endif
```

c_hello.cpp

```
#include "c_hello.h"
#include "ace_hello.h"

#define cHELLO2Hello(cHELLO) (Hello*) cHELLO.hello

int HELLO_new(cHELLO* cHello, unsigned char* name) {
  Hello* h;
  ACE_NEW_NORETURN(h, Hello(reinterpret_cast<ACE_TCHAR*> (name)));
  cHELLO result;
  result.hello = h;
  *cHello = result;
  return 0;
}

int HELLO_print(cHELLO cHello) {
  Hello* h = cHELLO2Hello(cHello);
  h->printHelloName();
  return 0;
}
```

Note that we use a struct similarly named to hold a void* pointer to the C++ object. Also, note the `extern "C"` statement in the header file, surrounded by `#ifdef __cplusplus` and `#endif` preprocessor commands. This allows c_hello.h to be included by C++ files (`c_hello.cpp`) as well as C files (`hello.c`) as shown below. The other items to note in `c_hello.cpp` are that it includes another C++-only header (`ace_hello.h`), and the conversion of inputs from C-friendly types to the real C++ types.

hello.c

```
#include "c_hello.h"
```

```
int main(int argc, char* argv[]) {
  cHELLO h;
  if (argv[1])
    HELLO_new(&h, argv[1]);
  else
    HELLO_new(&h, "some guy");
  HELLO_print(h);
}
```

In the Makefile, note that `gcc` is used to compile the `hello.c` file, while g++ is used to compile the rest of the files, including the shared libraries (`libhello.dll` and `libc_hello.dll`). We marked the C wrappers by prepending the wrapped class name with c_.

Makefile

```
ACE_ROOT="C:\ACE_wrappers"
CPPFLAGS=-I${ACE_ROOT}
LDFLAGS=-L${ACE_ROOT}/ace -lACE
CPP=g++
CC=gcc

objects = ace_hello.o

.PHONY: clean all

all: $(objects) libhello.dll libc_hello.dll hello

libhello.dll: $(objects)
$(CPP) -shared \$(objects) $(LDFLAGS) -o libhello.dll
```

```
libc_hello.dll: libhello.dll c_hello.o

$(CPP) -shared c_hello.o -L. -llibhello $(LDFLAGS) -o libc_hello.dll


hello: hello.c

$(CC) -o hello.o -c hello.c

$(CPP) -o hello hello.o -L. -lc_hello


ace_hello.o: ace_hello.h

c_hello.o: c_hello.h


clean:

-rm -f *.exe *.o *~ *.dll
```

Appendix D

Building ICM

This appendix contains build notes ICM, and its dependencies (ACE, SQLite, Pidgin, OpenSSL).

1. **SQLite:** Download the latest version of SQLite from http://sqlite.org/download. When compiling SQLite, be sure to comment out this line in Makefile.in:

   ```
   # TCC += -DSQLITE_OMIT_LOAD_EXTENSION=1
   ```

   Without extensions, all uses of ICM's MIDs (see page 25) will be rendered ineffectual. When we compare the MID to the hashes stored in the database, we use the prefix extension.

2. **ACE:** Download the latest version of ACE from http://download.dre.vanderbilt.edu. Follow the extensive ACE build instructions.

3. **OpenSSL:** Download the latest version of OpenSSL from http://openssl.org/source.

4. **Pidgin:** Download the latest source of Pidgin from http://www.pidgin.im/download/source. If compiling on Windows, be sure to compile in a directory whose path contains no spaces.

5. **ICM:** Download the source from Subversion (LL only: http://subversion/svn/sgc/trunk/certmgr) If debugging, uncomment `-DACE_NTRACE=0` to turn on ACE tracing.

Bibliography

[1] AIM. http:/www.aim.com.

[2] CamelCase. http://en.wikipedia.org/wiki/CamelCase.

[3] Cygwin. http://www.cygwin.com.

[4] Distinguished Encoding Rules (DER). http://asn1.elibel.tm.fr/standards.

[5] Doxygen. http://www.stack.nl/ dmitri/doxygen.

[6] etags. http://www.gnu.org/software/emacs/emacs-lisp-intro/html_node/etags.html.

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

[8] GNU Emacs. http://www.gnu.org/software/emacs.

[9] Google Summer of Code. http://code.google.com/soc/2007.

[10] Internet Relay Chat. http://www.ietf.org/rfc/rfc1459.txt.

[11] Jabber. http://www.jabber.org.

[12] Microsoft Outlook. http://office.microsoft.com/en-us/outlook/default.aspx.

[13] MinGW. http://www.mingw.org.

[14] Network Security Services (NSS). http://www.mozilla.org/projects/security/pki/nss.

[15] Off-the-Record Messaging. http://www.cypherpunks.ca/otr.

[16] OpenSSL Certificate Authority Setup. http://sial.org/howto/openssl/ca.

[17] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org.

[18] PEM. http://www.ietf.org/rfc/rfc1421.org.

[19] Pidgin. http://www.pidgin.im.

[20] Pidgin-Encryption. http://pidgin-encrypt.sourceforge.net.

[21] Security Axioms. http://www.avolio.com/papers/axioms.html.

[22] S/MIME Mail Security (smime) Charter. http://www.ietf.org/html.charters/smime-charter.html.

[23] SQLite. http://sqlite.org.

[24] Subversion. http://subversion.tigris.org.

[25] The ADAPTIVE Communication Environment (ACE). http://www.cs.wustl.edu/ schmidt/ACE.html.

[26] Thunderbird. http://www.mozilla.com/en-US/thunderbird.

[27] Transport Layer Security (tls) Charter. http://www.ietf.org/html.charters/tls-charter.html.

[28] Windows. http://www.microsoft.com/windows/default.mspx.

[29] Windows Live Messenger. http://get.live.com/messenger/overview.

[30] XMPP. http://www.ietf.org/rfc/rfc3920.txt.

[31] Yahoo! Messenger. http://messenger.yahoo.com.

[32] CertMgr - Pidgin, August 2007. http://developer.pidgin.im/wiki/CertMgr.

[33] Daniel Avrahami and Scott E. Hudson. Communication Characteristics of Instant Messaging: Effects and Predictions of Interpersonal Relationships. *CSCW*, November 2006.

[34] Stephen D. Huston, James CE Johnson, and Umar Syyid. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*. Addison-Wesley, 2003.

[35] Raymond Kurzweil. The Law of Accelerating Returns. http://www.kurzweilai.net/articles/art0134.html.

[36] Butler Lampson. Practical Principles for Security. In *Software System Reliability and Security*, 2006.

[37] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL*. O'Reilly, 2002.