Implementation of Elliptic Curve Encryption in the GAIM Instant Messenger Client

by Matthew Estes, Philip Hines, and Michael Lambert, GMU

Abstract-- This paper describes the development of a plug-in for the Instant Messaging client, GAIM, which is used to encrypt instant messages using Elliptic Curve Cryptography (ECC). It covers the project’s design criteria, protocol descriptions, implementation obstacles, test instrumentation, plug-in usage, performance analysis, and specification divergence. In addition, we describe a separate utility developed based on the cryptographic engine of the plug-in which provides general purpose file encryption, decryption and asymmetric key generation.

Index Terms— Elliptic Curve Cryptography, ECC, Instant Messaging, IM, GAIM, OpenSSL

I. INTRODUCTION

Instant Messaging (IM) is becoming increasingly popular and is a convenient form of communication for online and mobile phone users. In a recent study by AOL, 38 percent of U.S. Internet users send as many or more instant messages as emails, and 26 percent of those employed said that they use IM at work [2]. Additionally, 29 percent of IM users say they send files using IM, and, in another study [1], 9 percent of users surveyed admit to sending information that contained sensitive information about their business, themselves, or their clients. IM has become a significant security concern as most clients do not encrypt the messages as they are sent over the internet. It is also a concern since messages that are sent travel over the Internet to the service provider, and back to the recipient. These messages are therefore at risk of compromise on the computers at both ends, in transmission over the Internet each direction, and by system administrators of the service provider. For these reasons and others, IM has been identified as a security danger and has been outlawed by many companies. For the same reasons, there is a need for greater confidentiality of IM users.

To address this concern, this project is focused on developing a practical tool for users of instant messaging for providing confidentiality of their communication. The goal is to enhance the general security options available to the IM community. A secondary goal is to add new functionality to the community in the form of making a new cipher technique available. The form chosen was to use public key Elliptic Curve Cryptography (ECC) in the GAIM IM client. The third goal involved evaluating different cryptographic implementations for use in the plug-in. This paper will discuss the development of the ECC functionality for the GAIM client.

II. OVERVIEW

GAIM is an open source IM system that supports accounts on AIM, ICQ, Yahoo!, MSN, Jabber, IRC, Napster, Gadu-Gadu, Zephyr, and SILC. It is a cross-platform product running on Windows 95/98/NT/2000/XP, BSD Platforms, UNIX, Linux, and others. GAIM supports a plug-in architecture and the plug-in interface was utilized in this project.

In addition to GAIM, a pre-existing plug-in implementing off-the-record (OTR) messaging was used as a baseline for the implementation. The OTR plug-in implemented a Diffie-Hellman key exchange combined with AES symmetric key encryption. The Diffie-Hellman internals from the OTR plug-in were removed and replaced by an ECC engine based on OpenSSL.

III. BACKGROUND

A. Elliptic Curve Cryptography

A full discussion on Elliptic Curves is beyond the scope of this paper but a basic description is provided for those familiar with RSA encryption. ECC comes in two varieties; the first utilizes prime curves defined over $\mathbb{Z}_p$ and the second is binary curves constructed over the Galois-Field $\text{GF}(2^n)$. The first method is most efficient in software, while the second is most efficient in hardware [7]. The two varieties are not interchangeable as they each use different equations:

$$y^2 \mod p = (x^3 + ax + b) \mod p \quad \text{for fields over } \mathbb{Z}_p$$

and

$$y^2 + x \cdot y = x^3 + a \cdot x^2 + b \quad \text{for fields over } \text{GF}(2^n)$$

The public parameters are a curve type (with the chosen coefficients) and a point of the curve. The list of references at the end of paper can be consulted for more detail.

B. Benefits of the proposed ECC plug-in

Currently, time required to solve the elliptic curve discrete logarithm for a particular ECC key (which is required to
obtain the private key) is exponential when compared to RSA’s polynomial factoring time. This allows ECC to provide an equivalent level of security as RSA with a smaller key size [4]. In the future, progress in factoring algorithms may reduce the complexity of attacks (e.g. order notation \(O(.)\)) on ECC keys from exponential complexity to polynomial complexity as is the case with RSA. As a result, ECC can provide superior levels of security with a much smaller key size.

The second benefit of ECC is in the time required for key generation compared to RSA. In the course of the plug-in development, the development team compared the time required for each of the operations required using both RSA and ECC using the OpenSSL library. Unfortunately, performance results for encryption and decryption using RSA could not be obtained due to a problem with OpenSSL. In the case of key generation, though, RSA was found to be significantly slower than ECC. Additionally, levels of security equivalent to AES 128, 192, or 256 according to NIST [10], RSA key generation was unduly slow to a degree that would negatively affect the user experience. For the same levels of security, however, ECC could generate keys fast enough to go unnoticed by the user. The figure below shows the time required for generating ECC keys of each of the three main types and for generating RSA keys, based on the levels of security provided by symmetric encryption of 80 bits, 112 bits, 128 bits, and 256 bits.

Additionally, it stands as an example of using OpenSSL’s ECC capabilities, and thereby increasing the usability of OpenSSL.

IV. DESIGN

A. ECC Implementation Methodology

The plug-in developed by this project uses an ECC/AES hybrid encryption scheme. Elliptic Curve Diffie-Helman (ECDH) key agreement scheme is used to agree on an AES session key. AES is then used for data encryption. This approach was chosen both because it is secure and because of its simplicity. Since ECC by itself is not an actual encryption scheme (like RSA), but rather a foundation for a family of cryptographic algorithms, a specific algorithm must also be selected. Many of these algorithms, including El-Gamal, Menses-Vanstone, and ECDH encryption (which is similar to, but not the same as ECDH key agreement) all use keys that are generated for use with each message. In this plug-in, however, generation of a message key was an unnecessary cost in both performance and complexity. For this reason, ECDH key agreement was chosen as the best approach.

Since the public and private keys are stored and reused for multiple sessions, two users will consistently agree upon the same AES key. For this reason, the AES key which is the output of ECDH is used only to encrypt a session key for use in a single IM session. This thereby reduces the amount of data available for use in an attack against the public key cryptosystem and ensures that any compromise of the key for single session will affect only that session.

B. Library Selection

The evaluation of existing cryptographic libraries was made early on in the process and no formal evaluation criteria were involved. The selection involved listing the strengths and weaknesses of each of the libraries, giving consideration to importance. This involved reading documentation, examining supported algorithms, compiling the source code, judging the strength of the underlying implementations, and evaluating ease of use. The candidates were ecc, libecc, pegwit, Crypto++, borZoi, and OpenSSL.

<table>
<thead>
<tr>
<th>Library</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecc</td>
<td>Is not written in ANSI C and requires the Objective-C compiler. The GAIM and OTR plug-in required different compilers thus the implementation was deemed too risky to use.</td>
</tr>
<tr>
<td>libecc</td>
<td>Library compilation failed at the configure script. The library status is developmental and has no stable build release.</td>
</tr>
<tr>
<td>Pegwit</td>
<td>Builds an executable file and was not packaged as a library.</td>
</tr>
</tbody>
</table>

Incorporating the ECC features into this project’s plug-in makes GAIM the first application of its kind to use ECC in everyday communication. It is also the first GAIM plug-in that integrates with the OpenSSL library, opening the door to allow GAIM to integrate other OpenSSL encryption packages into its feature set. In these respects, this project is quite original as it can pave the way for future improvements in IM security using these tools.

Figure 1 – Key Generation Time For Equivalent Levels of Security
These candidates were quickly eliminated and those remaining examined in more detail.

<table>
<thead>
<tr>
<th>Table 2 - Library Selection Second Round</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Library</strong></td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>borZoi</td>
</tr>
<tr>
<td>Crypto++</td>
</tr>
</tbody>
</table>

A consensus was reached to attempt to use OpenSSL for the project and if significant problems were encountered then the project would be attempted with the borZoi implementation.

**C. ECC Curve Selection**

In its current form, the plug-in provides a constant level of security, which is consistent with the level provided by AES 128. Therefore since the symmetric cipher is constant, it made sense for the asymmetric cipher to be constant as well and for it to provide a level of security consistent with the level of security provided by the symmetric cipher. If the asymmetric cipher is ever changed to provide more levels of security, the symmetric cipher should be changed likewise to maintain consistency. For this reason, we chose to use the NIST recommended Prime Curve P-256 [8]. This is a curve with coefficients that have been chosen and verified to provide adequate security consistent with 128 bit symmetric encryption. Moreover, of the three curves which NIST claims provide this level of security, the NIST P-256 curve performed substantially better than the other two.

**D. Key Agreement**

Key agreement is implemented at initiation of an encrypted session using the Elliptic Curve Key Agreement Scheme, Diffie-Hellman 1 (ECKAS-DH1) in accordance with the Chapter 9 of IEEE Standard 1363-2000. When two instant messaging peers wish to begin an encrypted session, each will generate and exchange parameters to initialize the session.

**E. GAIM-OTR Adaptation**

The GAIM-OTR plug-in was selected to serve as an example for implementing a GAIM plug-in and to provide some of the basic user-interface components and plug-in logic. GAIM-OTR was separated into two components. The GAIM-OTR plug-in code manages the user interface and plug-in logic.

The encryption code was handled by a separate library called libotr. Since the plug-in logic and user-interface will not change significantly with GAIM-ECC, the GAIM-OTR plug-in code did not require major modifications.

However, libotr required almost a complete rewrite. The libotr library relies heavily on a GNU encryption library called libgcrypt to manage cryptographic functions. But, libgcrypt does not support any Elliptic Curve functionality. Because the data types differ between libgcrypt and OpenSSL, few of the libgcrypt encryption functions could even be reused.

**F. Plug-in Key Generation**

The plug-in will generate an ECC private/public key-pair from the preferences menu of the plug-in. The plug-in will maintain independent key-pairs for each account (such as Yahoo or AOL). These keys are stored in a file that is line-delimited. The only way to remove a key is to manually edit the file or delete the file.

**G. Plug-in Protocol**

The plug-in must be able to support all the functionality required to complete the key exchange and to encrypt messages. It must also be able to handle exceptional situations such as when a user becomes disconnected or a user does not support GAIM-ECC. Therefore, there is a protocol that must be developed to both send and interpret messages.

There are a few protocols that must be outlined. There is the message type, the handshake (or authentication) protocol, and there is the encryption protocol.

Each of these handshake steps sends a message with certain fields. These message fields are described below:

<table>
<thead>
<tr>
<th>Table 3 - Handshake Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Field Name</strong></td>
</tr>
<tr>
<td>Message type</td>
</tr>
<tr>
<td>Header</td>
</tr>
<tr>
<td>Reply</td>
</tr>
<tr>
<td>UKeyLen</td>
</tr>
<tr>
<td>UKey</td>
</tr>
<tr>
<td>AES Len</td>
</tr>
<tr>
<td>ecc_encsession_key</td>
</tr>
<tr>
<td>siglen</td>
</tr>
<tr>
<td>signature</td>
</tr>
</tbody>
</table>

The following table summarizes the steps in the handshake protocol used to establish the encrypted session. The steps are described in detail below the table:

<table>
<thead>
<tr>
<th>Table 4 - Handshake Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Participant</strong></td>
</tr>
<tr>
<td>1. A sends to B</td>
</tr>
</tbody>
</table>
2. B sends to A

| 2. B sends to A | Responds with B’s public key. |

3. A


4. A send to B

| 4. A send to B | Sends A’s public key and encrypted session key to B. |

5. B


6. Begin encrypted conversation by storing off the AES Session key from the authentication structure to the context structure (see below)

Step 1: The state of GAIM-ECC always starts with both clients in an unencrypted state. The handshake process begins when one user initiates a QUERY, shown as step 1 in the “Handshake Steps” figure above. This is accomplished by User A sending a message to User B that has the GAIM-ECC message type of “MSGTYPE_QUERY”, which is represented by the text “?ECC?”. If User B does not have a plug-in, then he will see an informative message, starting with “?ECC?”, telling him that the other user was attempting to initiate a conversation with him.

Step 2: If User B’s plug-in interprets this message, it will call auth_start_ecc(). This function will create an authentication message type of MSGTYPE_KEYEXCH, represented by the text “?ECC:AAEK?”. It will set Reply to 0. This means that User B is not replying to a message sent by User A and this variable is used to control the state of the key exchange. It will then populate its public key into the field and create the message by calling create_ecc_key_exchange_message(). The other fields will be blank at this time.

Step 3: User A will receive the message and because it is of type MSGTYPE_KEYEXCH, it will call handle_ecc_key_exchange(). This function will see that the reply is 0 and know that it has to generate the random AES session key, derive an AES agreed key from the public key of User B (that he just received), and encrypt that session key. This session key is populated, along with User A’s public key. This message is now signed with a SHA-1 hashing algorithm and Elliptic Curve DSA based on User A’s private key.

Step 4: The reply is set to 1 to indicate that User A is replying to a key exchange message. The message is sent to User B by calling create_ecc_key_exchange_message(). At this point, User A activates encrypted communication, since he knows the session key, by calling go_encrypted().

Step 5: User B receives the message with type of MSGTYPE_KEYEXCH and calls handle_ecc_key_exchange(). However, since reply is now 1 in the received handshake data, this function responds differently. It now calculates the AES agreed key, as before, and then uses this key to decrypt the AES Encrypted Session Key. It verifies the signature, and then calls go_encrypted() to activate encrypted communication.

Throughout this process, data for the handshake is temporarily stored in the authentication structure. This structure is described below:

### Table 5 - Authentication Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>authstate</td>
<td>State of authentication (encrypted or not)</td>
</tr>
<tr>
<td>their_UKeyLen</td>
<td>Length of their public key</td>
</tr>
<tr>
<td>their_UKeyBuf</td>
<td>Their public key</td>
</tr>
<tr>
<td>ecc_agreed_key</td>
<td>The derived key from the ECDH key derivation process.</td>
</tr>
<tr>
<td>ecc_session_key</td>
<td>Unencrypted AES session key, randomly generated initially.</td>
</tr>
<tr>
<td>ecc_encsession_key</td>
<td>The ecc_session_key encrypted by AES using the ecc_agreed_key.</td>
</tr>
<tr>
<td>lastauthmessage</td>
<td>Buffer that stores the next message that needs to be sent to the other user.</td>
</tr>
<tr>
<td>protocol_version</td>
<td>Currently, only 1 is supported. For future use.</td>
</tr>
<tr>
<td>their_fingerprint</td>
<td>The SHA1 hash of their public key. This is stored for future reference and also displayed in the preferences menu for offline verification.</td>
</tr>
</tbody>
</table>

At this point, both sides know the session key and the handshake protocol is complete. Part of the job of the go_encrypted() function mentioned earlier is to move the data stored in the authentication structure to the context structure shown in the figure below. It remains in the context structure for the duration of the conversation. This context data is maintained until the client is shut-down completely. The AES session key stored in the context structure is also the only one used for encrypted communication.

### Table 6 - Context Structure

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>The username for the account.</td>
</tr>
<tr>
<td>protocol</td>
<td>The type of service (Yahoo, AOL, MSN, etc.)</td>
</tr>
<tr>
<td>auth</td>
<td>The auth structure used during handshake.</td>
</tr>
<tr>
<td>msgstate</td>
<td>The state of the message.</td>
</tr>
<tr>
<td>protocol_version</td>
<td>Currently, only 1 is supported. For future use.</td>
</tr>
<tr>
<td>their_UKeyLen</td>
<td>Length of their public key</td>
</tr>
<tr>
<td>their_UKeyBuf</td>
<td>Their public key</td>
</tr>
</tbody>
</table>
The current implementation is vulnerable to three types of attacks. The local machine has two known vulnerabilities in the implementation.

For encrypted communication with data packets, the message fields become drastically simpler as shown in the following figure:

Table 7 - Encrypted Data Fields

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Type</td>
<td>?ECC:AAID?</td>
<td>10</td>
</tr>
<tr>
<td>Data</td>
<td>(AES Encrypted Data)</td>
<td>variable</td>
</tr>
</tbody>
</table>

The message type tells the plug-in that the data following the header is message data to be decrypted and displayed with the AES Session Key established by the handshake protocol.

V. IMPLEMENTATION

A. Implementation Attack Surface

This section covers the known attacks against the plug-in implementation. The goal of this section is not to eliminate all possible attacks but to illustrate possible attacks against the components used in the design e.g. files, sockets, etc. The level of risk must be within the bounds of the project’s scope and function. Any work needed could be considered for future work within the implementation.

1) Network

The current implementation is vulnerable to three types of attacks.

a) Key Negotiation: This vulnerability involves man-in-the-middle attacks where an attacker could use a network sniffer to detect the transmitted public key and replace it with their own public key. They could then automate the decoding, storing, and forwarding of the attack stream.

b) Data Encryption: ECC is a new encryption scheme that may not hold up over time. In comparison factorization of prime numbers as in RSA has been studied for hundreds of years.

c) Keys Strength: An attack vector against GAIM, OTR, OpenSSL ECC, or this plug-in could be used to dramatically reduce the key strength. An example would be the discovery of a buffer overflow vulnerability that would allow an attacker to take control of the application.

2) Local Machine

The local machine has two known vulnerabilities in the implementation.

a) Key Storage: Should a user’s local machine become compromised their secret key files could be stolen and/or the public key files replaced. This places the security of future IM sessions at risk.

b) Plug Vulnerability: The plug-in itself is vulnerable to a tampering attack and has no way to defend against it. It is possible that a rogue plug-in could use the GAIM plug-in architecture to listen to GAIM internal data which contains unencrypted chat session data. The rogue then simply forwards the desired data to a server of their choosing for later retrieval.

3) Source

The plug-in is vulnerable to a three types of source or vendor channel attacks.

a) Binary Integrity: A skillful attacker with a hex editor could hobble the binary to prevent or reduce the strength of encryption.

b) Vendor Integrity: The binary files posted to the download server for the plug-in are vulnerable to web server attack that replaces the OEM image.

c) Source Integrity: The source code for the project if posted online could be modified by a later ‘contributor’ to weaken the cryptographic strength of the implementation. Inserting a buffer overflow or unsigned integer vulnerability into the source may be hard to detect if skillfully performed on part of the system that has a low level of observable activity.

B. Implementation Testing Regimen

The following testing regimen was developed to verify various aspects of the system. Success of the project is based on the correct operation of the ECC encryption and decryption algorithms, sound design of the software plug-in, and tested operation of ECC within the GAIM messenger client. The following metrics are designed to validate this approach.

1) ECC engine tests: One or more command line programs were designed to test the exercising of the ECC engine. The programs tested the following aspects:

- ECC library initialization,
- AES Encryption/Decryption,
- ECDH Public/Private Key Generation, and
- ECC Signing and Verification.

2) Key and IV truncation test: A software trace of the source code was made to verify that the encryption key and initialization vector are not truncated.

3) Verify encryption and decryption operation in plug-in: Verify that once the user has successfully initiated an encrypted session that message text and file data transferred between users are encrypted in transit and readable by the remote side. Users should not need to perform special actions to type and send encrypted text.

4) Verify generation of ‘one time’ keys: The tester verifies that keys generated are random and/or have a long period before repetition. The test will also verify that the recipient’s private key generated for decryption of data is the only key created for that purpose. The tester verifies that the ECC library does not make unintended back-up keys. Decryption of encrypted test data shall be attempted with several keys created during the test.

5) Verify installation of plug-in: Verify that the plug-in may be successfully installed into an existing GAIM installation without disrupting any existing settings or configurations. Verify that the plug-in is recognized and
can be loaded by GAIM. Verify that installation of the ECC plug-in does not change normal unencrypted messaging.

6) **Verify user management of keys in the plug-in**: Verify that the ECC plug-in provides a means for users to exchange keys. Verify that the key exchange does not compromise any of the user’s private keys.

7) **Verify ECDH operation with no prior key exchange**: Test the start a new encrypted messaging session between two users with no prior key exchange. Test that plug-in can perform all steps in the ECDH operation to securely create an encrypted session using only the GAIM and plug-in UI.

8) **Fuzz Testing**: Tester verifies that during normal operation of the plug-in will cause neither GAIM nor the operating system to crash.

9) **Compatibility**: The tester will verify that messages may be sent to other clients that do not have the plug-in and that these messages are unencrypted.

10) **UI Feedback**: The tester will verify that the UI has a mechanism for informing the user that a message and/or conversation is encrypted. The tester will verify that the indicated message and/or conversation is encrypted.

11) **Removable**: The tester will verify that the encryption may be enabled and disabled by the user. The test will verify that the plug-in can be uninstalled without damaging GAIM or the operating system.

## C. Development Challenges

1) **Windows Build**: Attempts to build GAIM and OTR on the Windows platform were not as successful as initially planned. After several attempts to resolve the dependencies between the components the project moved to the Linux platform. OpenSSL is available on both platforms and as a consequence none of the progress made in flushing out the source code between GAIM-OTR and OpenSSL was sacrificed.

2) **GAIM-OTR libgcrypt**: GAIM-OTR’s Linux implementation relied heavily on the libgcrypt library. As a consequence significant portions of the OTR implementation had to be rewritten to support the ECC functionality. This required rewrites of the key storage routines for storing a catalog of accounts, protocols, and arbitrary length key values for those accounts needed to be rewritten.

3) **OpenSSL**: The OpenSSL implementation suffered due to poor documentation of the ECC specific pieces. Additionally, the library itself makes use of minimal source code comments. With some effort the project was able to work with the library and did not have to fall back to alternative ECC library implementations. This did, however, require significant effort in reverse engineering the source code of OpenSSL to determine its interface. It is noted that for OpenSSL to be useful to future developers of cryptosystems, it will need much improved documentation, especially regarding interfaces and usage.

4) **Real-Time Chat**: Implementing cryptography over real-time chat with multiple accounts and multiple states is a painful exercise. It required tracking of multiple:

- States - Online, Offline, Negotiating, Encrypted, Finished
- Accounts – AOL, Yahoo, MSN…
- Conversations
- Unpredictable session states - User is “away”

Since an encryption stream maintains state careful coordination and debugging was required.

5) **Debugging using IM user accounts**: Debugging is complicated since many service providers will lock out a user if that user regularly connects and disconnects, which was required for debugging the GAIM plug-in.

5) **Key Storage With Multiple Accounts**: Key storage routines had to be completely rewritten including storing a catalog of accounts, protocols and keys of arbitrary lengths.

## VI. ECCPG Utility

### A. Usage

In addition to developing the plug-in as the primary effort, the development team has developed a utility for encrypting and decrypting files using ECC, called the Elliptic Curve Cryptography Privacy Guard. This utility has been developed for the Windows PC platform based on the cryptographic engine of the plug-in. It allows a user to generate ECC public and private keys, encrypt and compress files, and decrypt and decompress files. The command line interface is documented in the help screen displayed when no command line parameters are given:

![ECCPG Help Screen](image)

### B. Operation

The ECCPG utility also uses ECDH, but the agreement is between the intended user’s key and a message key, rather than between the keys of two users. When a file is encrypted, the utility:

- Deserializes the public key file of the recipient
- Generates a message public and private key
• Calculates the ECDH key agreed between the key of the recipient and the message key and uses this key to encrypt a generated AES session key
• Writes a header to the beginning of the file including the message public key
• Compresses and encrypts the file (all in one step) using the gzip algorithm and AES in the cipher feedback mode
• Discards the message private key

Similarly, to decrypt a file, the recipient’s utility:
• Deserializes his or her private key
• Reads the message public key
• Calculates the ECDH key agreed between his or her key and the message public key
• Decrypts the session key
• Decrypts and decompresses the file (all in one step) using AES in the cipher feedback mode and the gzip algorithm

VII. RESULTS

A. GAIM Plug-in Usage

The client needs to install the plug-in in the GAIM plug-in installation directory. From the GAIM Preferences menu the plug-in dialog allows the user to activate the ECC plug-in shown in Figure 2.

![Figure 3 - ECC Plug-in Installation](image)

Once enabled the user can go in and configure their default settings. They can also generate and read the fingerprint of their public key as shown in Figure 3.

![Figure 4 - Private Key Configuration](image)

Once the plug-in is configured and the private key is generated, the user must open a dialog to the intended user. In this session shown below, user “mattaestes” speaks to “ecctestuser” in the clear at first, and then attempts to initiate encrypted communication by pressing the “ECC: Not Private” button.

![Message exchange](image)

Now, the GAIM-ECC plug-in initiates the Elliptic Curve Diffie-Hellman key agreement process. Since this is the first time “mattaestes” has talked with “ecctestuser”, the plug-in reports an “Unverified conversation”.

![Unverified conversation](image)
Figure 6 - User A Desires to Encrypt Conversation

An “Unverified” conversation with “ecctestuser” begins. This means that both users have properly generated and installed private keys and that the key exchange process was successful, but that “mattaestes” does not have any trust for the provided public key for “ecctestuser”. Note that both the text and the button in the lower-right indicate “Unverified”. It is perfectly possible for the user to continue communicating at this point, but without verifying the fingerprint through an alternate channel you leave yourself open to man-in-the-middle attacks that falsify public-keys and fingerprints.

Figure 7 – Unverified Fingerprint in Preferences

“mattaestes” must now verify the fingerprint for “ecctestuser”. To do this, he must compare the fingerprint reported by the remote user under the “Config” tab of the preferences with the fingerprint shown in the “Known Fingerprints” tab. Presumably, this is done through an alternate channel such as email or telephone. To verify the fingerprint, the user clicks on the fingerprint and clicks “Verify Fingerprint”. Since GAIM-ECC stores the fingerprint for future communication, this step need be completed only once.

Figure 8 – Verified Fingerprint in Preferences

Now, the figure above shows the fingerprint is verified.

Figure 9 – New Verified Conversation

At this point, GAIM-ECC will report “ECC: Private” for all conversations with this user. The above figure shows an entirely new conversation with the “Private” conversation. Note the “Private” indicator in both the text and the button in the lower-right.

At any point, the session can be refreshed (generating a new session key) by pushing the “ECC: Private” button. The conversation is unaffected by the refresh, it simply continues on with a new dynamically generated AES session key.
**B. Performance Results**

In the process of developing the plug-in, the development team has done performance analysis between ECC and RSA. All of the performance analysis was performed on a 600MHz Pentium III computer with 256 MB of RAM running Windows 2000. While this is admittedly a below average computer by current standards, it is a good choice for performance analysis since it represents the lower end of likely users of the system. Additionally, all performance data was done based on the encryption and decryption of a single block of data, assuming that this is all that is necessary to transmit or agree upon a symmetric key for further encryption. Unfortunately, the RSA tests were unsuccessful due to a problem with OpenSSL and as a result, we were unable to include RSA encryption and decryption data. As a result, we include only a comparison of the three types of ECC curves based on equivalent levels of security according to NIST.

![Figure 10 - Encryption Performance of Equivalent Levels of Security Using OpenSSL](image)

![Figure 11 - Decryption Performance of Equivalent Levels of Security Using OpenSSL](image)

Even though we were unable to confirm any performance improvement of ECC compared to RSA, several papers [11] [12] [13] have made such a comparison, finding ECC to be significantly faster than RSA (although the magnitude of the difference varies). Overall, these comparisons find ECC to provide a small improvement for lower levels of security, but find a much larger improvement for higher levels of security. This conclusion is also confirmed in a report by the NSA [14].

**VIII. Agreement with Original Objectives**

Overall the project has excellent agreement with the original design goals. The core objectives have been and will be discussed in the conclusions of the paper. This section guides the reader through the project’s design goals and options that were presented as option in the original project specification.

**A. Certificate Authority Option**

This was an option in the original project specification. As the size and scope of the work for the project’s main goals grew the project team decided not to execute this option. The working implementation provides the key fingerprint mechanism to allow users to verify public key identities.

**B. Revising Key Exchange Algorithm**

This was an option in the original project specification. As the size and scope of the project grew the team decided not to execute this option due to its complexity and the limited time available.

**C. Additional User Interface Components**

User interface components were developed as necessary for the plug-in, but no additional user interface enhancements were developed.

**D. Separate ECC Library**

The option for a separate ECC library was not developed as a library for use by developers. Thus a standalone utility providing ECC encryption capabilities was created. This utility is documented as the ECCPG utility.

**IX. Conclusions**

This paper has covered the architecture of the GAIM and proposed a novel new plug-in for encrypting instant messages. In making the case for this application, a brief overview of the candidate ECC and OpenSSL technologies has been provided. Survey results for available ECC libraries were discussed. The security strengths and weaknesses of the implementation were mapped out and the test criteria described. Finally, a walk through of the available working application was provided.

A working plug-in implementing ECC for the GAIM IM client now exists. It runs on the Linux platform. It gives the IM community the option of a strong public key crypto system paired with a strong symmetric cipher algorithm for protection. This system enables the user to have more options in authenticating the sender of IM messages and...
prevent viewing by third-parties. It also serves as an example for other ECC applications, such as ECCPG.

These features raise the overall level of security options available to the IM community. It is the first time that ECC has been made available in an everyday application for the majority of the population. Thus, it is upon the preponderance of these accomplishments that the goals for this project are considered complete.

X. LIST OF REFERENCES


XI. APPENDIX A – PREREQUISITES

The GAIM ECC Module is written in C and the following baseline is needed to do direct builds of the plug-in source files.

- Linux Redhat Fedora Core 3, http://fedora.redhat.com/download/
- GNU C compiler, make environment, and Linux development libraries and headers
- OpenSSL 0.9.8a.tar.gz, http://www.openssl.org/source/
- gaim-1.5.0.tar.bz2, http://sourceforge.net/projects/gaim/
- gaim-otr-3.0.0.tar.gz, http://www.cypherpunks.ca/otr/#downloads
- libotr-3.0.0.tar.gz, http://www.cypherpunks.ca/otr/#downloads

In addition, to the above files the core packages for GAIM and GAIM-OTR have dependency lists which are not reproduced here. The individual packages should be consulted for the specific list of sub-packages. The OpenSSL version listed above has the proper support for ECC and builds cleanly on the Fedora Core platform. The build instructions differ between win32 and Linux. As mentioned in the Issues and Workaround section, the project moved to the Linux platform during development hence only the Linux build instructions are listed in these sections.