PGP Messaging: Elliptic Curve PGP Implementation for SMS

David E. Stoner

Abstract—PGP Messaging is an application for Android that provides secure short message service messaging using encryption, digital signature and a key management system based on Pretty Good Privacy. The application uses a hybrid encryption scheme consisting of elliptic curve keys for key generation, elliptic curve signature algorithms and advanced encryption standard for secret key encryption.

Index Terms—AES, advanced encryption standard, cryptography, digital signatures, elliptic curve cryptography, encryption, PGP, pretty good privacy, public key, short message service, SMS

I. INTRODUCTION

SHORT MESSAGE SERVICE, or SMS, is a protocol used for sending 160 character messages from one device to another. Sent messages are transmitted to a nearby cellular base station and routed to a short message service center (SMSC). From here, messages are routed to the base station that the receiving mobile device is connected to [1].

Pretty Good Privacy (PGP) is a computer program designed by Phil Zimmerman in 1991. This program was designed to exchange email securely using encryption and certificate verification. PGP uses a combination of digital signatures, encryption, compression and message encoding to provide these services. Additionally, PGP provides key management and certificate services [2].

II. MOTIVATION

The confidentiality of SMS messages may be compromised via many different avenues. Some products attempt to protect against attacks but the protection mechanisms provided can be improved.

A. Security Risks of SMS

SMS messages can be intercepted during transmission to and from a mobile device. SMS messages are also vulnerable while they are stored on short message service center servers awaiting download [3]. Messages may also be intercepted by a mobile device masquerading as another user’s device via subscriber identity module (SIM) card duplication. The security features of Global System for Mobile Communications (GSM) are not sufficient to protect a message from simple message interception [4].

SMS messages are vulnerable while residing on a mobile device. Typically, SMS messages are stored in unencrypted databases on a mobile platform. If the physical security of the device is compromised, any messages may be read. If any messages are sent while the phone is compromised, the receiver may not be aware of the received message’s fraudulent nature. Other vulnerabilities include applications/viruses that forward messages to undisclosed users or upload messages to online destinations [5].

B. Alternative Products

Many Android applications are available to combat these losses of confidentiality. The most common SMS encryption applications perform symmetric or secret key encryption. While effective, these applications suffer from key distribution problems as well as not providing any authentication services.

Other android applications have attempted to solve this problem using asymmetric or public key encryption. One such application, SMS PGP, uses RSA to encrypt messages for transmission. However, the RSA key sizes needed for secure encryption means multiple SMS messages are required to send messages of any length. Additionally, despite its name, PGP SMS does not provide the user with PGP services such as key rings, trust algorithms and digital signatures. SMS ECC, another application by the same developer of SMS PGP, provides symmetric encryption via elliptic curve cryptography but does not perform digital signatures of the message.

Another application, TextSecure, performs asymmetric encryption but relies on third party servers to provide additional keys during the encryption process. This application provides authentication but does not provide the non-repudiation and trust services of PGP.

III. IMPLEMENTATION

PGP Messaging was written for the Android platform to provide secure SMS messaging using the same methods set forth by PGP. This application uses a combination of elliptic curve cryptography and the Advanced Encryption Standard (AES) to provide confidentiality and non-repudiation services.

The application also uses digital signatures for sharing public...
keys with other users. None of the capabilities provided require a centralized infrastructure to function.

A. Cryptologic Provider

PGP uses RSA for public key cryptography and supports several different secret key encryption algorithms. RFC-4880 recommends that RSA keys are at least 1024 bit keys and 2048 bit keys are preferable to maintain message confidentiality [2]. SMS messages are limited to 160 characters or 1280 bits. However, because SMS messages are sent in normal text strings, some form of encoding (like Base64) is required. This means that a maximum of 120 bytes or 960 bits (prior to Base64 conversion) can be transmitted. Therefore, to send a single RSA encrypted message using 1024 bit keys would require two SMS messages. Due to this message length requirement, an alternative public key algorithm is preferable. Elliptic curve cryptography was chosen because it can provide the same level of security with smaller public key size. A 160 bit elliptic curve key provides the same security as a 1024 bit RSA key [6]. These smaller key sizes are much more conducive to the 160 character SMS limit.

Three cryptologic providers were considered for this application. The first provider considered was gnucrypto [7]. Gnu-crypto provides AES algorithms but does not support elliptic curve cryptography. The second provider considered was Bouncy Castle [8]. Bouncy Castle provides several symmetric encryption algorithms, elliptic curve digital signature algorithms and supports multiple elliptic curves. Bouncy Castle is written to interface with Java’s security provider which allows developers to request cipher objects with specific algorithms from various providers using generic Java classes. However, the Android SDK uses a stripped down version of Bouncy Castle as its default cryptologic provider. This stripped down version does not provide any elliptic curve cryptography functions beyond simple digital signature algorithms. Because of this inclusion, a developer cannot request advanced algorithms from imported Bouncy Castle jars because Java cannot differentiate between the imported provider and the default. An alternative to Bouncy Castle is Spongy Castle [9]. Spongy Castle is a repackage of Bouncy Castle for Android which was created to specifically address this problem.

B. Key Generation

All keys are generated using the National Institute of Standards and Technology (NIST) recommended P-192 elliptic curve [10]. A random number generator provided by the Android Software Development Kit (SDK) is used to seed the key generator.

When the elliptic curve key pair is initially generated, the user is prompted for a password. A SHA-256 digest of the entered password is created and is used as the key for the AES encryption of the private key. The encrypted private key and the unencrypted public key are Base64 encoded and stored in the private key database. The Base64 encoded public key is stored in the public key database.

C. Message Sending

To transmit encrypted messages, the user selects a User ID/Key Id pair from the private key ring, the desired recipient’s User ID/Key ID pair from the public key ring, enters a password (used to decrypt the private key) and enters the message to be encrypted. A SHA-256 digest is computed from the message and the digest is signed using an elliptic curve digital signing algorithm. The signature and message are concatenated together along with a single byte that indicates the length of the signature. The sender’s private key and the receiver’s public key are used to generate a 256 bit key via an Elliptic Curve Diffie-Hellman key derivation function (KDF). This key is used to encrypt the message and signature using AES-256. The key IDs selected are attached to the encrypted message. The entire byte array is then Base64 encoded and transmitted using the Android SMS interface. Fig. III-1 is a block diagram of the entire message encryption process.

D. Message Receiving

Received SMS messages are stored in the local Android SMS database on the receiving device. All SMS messages sent by PGP Messaging remain encrypted within the database. The application searches the SMS database for SMS messages originating from the application (messages starting with PGP). All conversations that contain at least one instance of this string are displayed for further viewing and decrypting. When a conversation is chosen, all SMS messages within that conversation are displayed.

The user is presented with a view of all SMS messages within a given conversation. This view includes the text of the SMS, the time received and telephone number (user ID) of the sending user. When a password is entered and a button is pressed, decryption is attempted on every message with the PGP header. The entire message is Base64 decoded and the sender and receiver key IDs are removed from the message and used to retrieve the corresponding public and private keys. The private key is retrieved from the private key database and decrypted using the entered password. The sender’s public key is retrieved from the public key database. Both keys are used in an Elliptic Curve Diffie-Hellman key derivation function to create the same 256-bit key used in encryption. The resulting key is used to decrypt the message and signature. Signature verification occurs and if it succeeds, the plain text of the message is displayed to the user. If the
verification fails, the encrypted data remains displayed. In addition to the unencrypted message, an indicator shows the associated level of trust in the Public Key used to encrypt the message. Fig. III-2 is a block diagram of the entire message decryption process.

![Fig. III-2. Implemented Message Receiving/Decryption](image)

E. Private-Key Ring

The Private-Key Ring contains the Base64 encoded AES-256 encrypted private key, the Base64 encoded public key, the associated key ID, a user ID and a time stamp. The key ID is the last 64 bits of the Y coordinate of the public key. This ID is used as a unique identifier for the key pair and is included in messages for the purpose of identifying the public key used for encryption and signature generation. The user ID is an 11 digit telephone number and is used to send the SMS message. The time stamp indicates when the key pair was generated.

From the private key ring, the user is able to transmit the public key to any user. The public key is Base64 encoded and packaged in an SMS message and transmitted unencrypted.

F. Public-Key Ring

The Public-Key Ring contains the Base64 encoded public key, the key ID, the user ID, time stamp, key legitimacy and owner trust. The key ID and user ID are of the same format as the private key database. The owner trust is a user definable owner trust. The key ID and user ID are of the same format as the private key database. The owner trust value is an integer that ranges from 1 to 4 (Table 1). The key legitimacy value is an integer that ranges from 1 to 4 (Table 2).

Table 1

<table>
<thead>
<tr>
<th>Trust Level</th>
<th>Meaning</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ultimate Trust</td>
<td>Public key can be found in Private-Key Ring</td>
<td>1</td>
</tr>
<tr>
<td>Complete Trust</td>
<td>Public key is always trusted to sign other public keys</td>
<td>2</td>
</tr>
<tr>
<td>Marginal Trust</td>
<td>Public key is usually trusted to sign other public keys</td>
<td>3</td>
</tr>
<tr>
<td>Not Trusted</td>
<td>Public key is not trusted to sign other public keys</td>
<td>4</td>
</tr>
<tr>
<td>Unknown User</td>
<td>Public key belongs to an unknown user</td>
<td>5</td>
</tr>
<tr>
<td>Undefined User</td>
<td>Public key belongs to an undefined user</td>
<td>6</td>
</tr>
</tbody>
</table>

Table contains all owner trust levels, their meanings and the integer value used in the public key database.

Table 2

<table>
<thead>
<tr>
<th>Trust Level</th>
<th>Meaning</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete Trust</td>
<td>Complete trust in key ownership</td>
<td>1</td>
</tr>
<tr>
<td>Marginal Trust</td>
<td>Marginal trust in key ownership</td>
<td>2</td>
</tr>
<tr>
<td>Not Trusted</td>
<td>Key ownership not trusted</td>
<td>3</td>
</tr>
<tr>
<td>Unknown Public Key</td>
<td>Unknown or undefined trust.</td>
<td>4</td>
</tr>
</tbody>
</table>

Table contains all key legitimacy levels, their meanings and the integer value used in the public key database.

The user can perform several different operations on public keys located within the Public-Key Ring. These operations include signing a given public key, setting a public key’s owner trust level, transmitting the public key and all associated signatures to another user and deleting the public key from the public key database.

G. Signatures

In PGP Messaging, signatures for the public key ring are generated by passing the bytes of the public key into the elliptic curve digital signature algorithm along with the signing private key. These signatures are then stored in the signature database. The signature database contains the Base64 encoded signature, the key ID of the signature creator, the key ID of the signed public key and signature trust. Signature trust is the same value as the signature creator’s owner trust value in the public key database. The signature trust values are updated on a regular basis to match the owner trust values in the Public-Key Ring.

Key legitimacy is set to “Complete Trust” if the public key was signed by a private key in the Private-Key Ring. Otherwise, key legitimacy in the Public-Key Ring is calculated based on the composite trust of the signatures signing the public key. “(Equation 1)” is used for calculating key legitimacy where $X$ is the number of always trusted signatures required for complete trust and $Y$ is the number of usually trusted signatures required for complete trust.

$$\frac{1}{X} \cdot n_{always trusted} + \frac{1}{Y} \cdot n_{usually trusted}$$

Equation 1

If the resulting value is greater than or equal to 1, the key legitimacy value in the Public-Key Ring is set to “Complete Trust”. If the value is between 0 and 1, the key legitimacy value is set to “Marginal Trust”. If the value is 0, the key legitimacy is set to “Not Trusted”.

Public keys and associated signatures can be transmitted to other users via SMS messaging. Received public keys and signatures can be imported into the Public-Key Ring. Once a public key is imported, the user may set the owner trust of the public key by manually choosing one of the available trust levels. Key legitimacy is automatically calculated.

H. Message Types

All SMS messages must be 160 characters or less. PGP Messaging uses three message types to exchange data.

The first message type is used for exchanging encrypted data between two users. All data in this message, with the exception of the PGPM header, is Base64 encoded. Once the header is taken into account (along with a semi-colon used as a delimiter), the total number of Base64 characters (or bytes) able to be transmitted is 114. This message contains the sending and receiving key ID, each of which are 8 bytes. This reduces the maximum number of transmittable bytes to 98. AES-256 encrypts in 32 byte chunks, therefore, with 98 available characters, only 96 bytes of data can be encrypted per message. This encrypted data contains a signature (which
can range from 54 to 56 bytes), one byte describing the size of the signature and the message itself. When all overhead has been accounted for, the total number of plaintext characters that can be sent is 39. Table 3 shows each field in the message and any associated encryption and encoding.

<table>
<thead>
<tr>
<th>Field</th>
<th>Number of Characters</th>
<th>Base64 Encoded?</th>
<th>AES-256 Encrypted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>4</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Receiver Key ID</td>
<td>8</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Sender Key ID</td>
<td>8</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Signature Size</td>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Signature</td>
<td>56 (max)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Message</td>
<td>39</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Message format for a message used to exchange encrypted data.

The second message type is used for sharing public keys. It uses the string “PGPPub” as the header followed by a colon. The remaining data consists of a user ID and Base64 encoded public key. Table 4 shows each field in the message and any associated encryption and encoding.

<table>
<thead>
<tr>
<th>Field</th>
<th>Number of Characters</th>
<th>Base64 Encoded?</th>
<th>AES-256 Encrypted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>6</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>User ID of Public Key Owner</td>
<td>11</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Public Key</td>
<td>75</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Message format for a message used to exchange a public key.

The third message type is used for sharing public-key signatures. It uses the string “PGPSig” as the header followed by a colon. Signature messages contain the signature creator’s key ID, the key ID of the signed public key, the user ID of the public key and the signature itself. Only the public key is Base64 encoded. Key IDs are sent as decimal numbers. Table 5 shows each field in the message and any associated encryption and encoding.

<table>
<thead>
<tr>
<th>Field</th>
<th>Number of Characters</th>
<th>Base64 Encoded?</th>
<th>AES-256 Encrypted?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>6</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Key ID of Signature</td>
<td>19</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Signature</td>
<td>56 (max)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Delimiter</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Key ID of signature creator</td>
<td>19</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Message format for a message used to exchange a signed public key.

IV. Java Classes

PGP Messaging is made up of 12 Java classes. The classes create the user interface, allow user input, perform the cryptographic functions or perform the key databases.

A. MainMenu

The MainMenu class is the starting point of the program. Upon creation, MainMenu parses the SMS database for all conversations (series of SMS messages between two users) that contain at least one message starting with the string “PGP”. This filters out any conversations that do not include messages created by PGP Messaging.

In addition to displaying SMS conversations, MainMenu provides the functionality for generating the elliptic curve key pairs, encrypting the private key and storing keys in the public and private key databases. MainMenu creates the KeyRingDBHelper class and provides the launch points for MessageComposer, MessageReader, PrivateKeyRing and PublicKeyRing.

B. KeyRingDBHelper

The KeyRingDBHelper class creates the private key, public key and signature databases which is SQLite. This class also provides an interface to the databases.

The private key database is used to store data used by the Private Key Ring. This data includes Base64 encoded public key (String), encrypted and Base64 encoded private key (String), timestamp of generation (long), user ID of private key owner (String) and key ID of key pair (long).

The public key database is used to store data used by the Public Key Ring. This data includes Base64 encoded public key (String), , timestamp of when key was added to database (long), user ID of public key owner (String), key ID of key pair (long), owner trust (integer), and key legitimacy (integer).

The signature database is used to store all signed public keys. The data includes Base64 encoded signature (String), key ID of signed public key (long), key ID of signature creator (long) and signature trust (integer).

C. MessageComposer

The MessageComposer class provides all fields necessary for sending an encrypted message. Spinners are populated with all key ID/user ID pairs from the public and private key databases. The password field is for decrypting the selected private key and the message field contains the plaintext message.

When a user pushes the send button, the encryption process is performed on the plaintext, the message to be transmitted is built and transmitted to the selected user via SMS.

D. MessageReader

The MessageReader class displays all received SMS messages in a chosen conversation. Messages originating from the PGP Messaging application remain encrypted within the SMS database until the password is entered and the “Decrypt” button is pressed. When that occurs, all encrypted messages displayed on the screen are decrypted. Messages that are successfully decrypted and verified are displayed to the user. The key legitimacy of the user on the other side of the conversation is looked up in the public key database and
an indicator is displayed based on the value retrieved. Regardless of the key legitimacy value, the message is always displayed if the signature verification succeeds.

In addition to decryption functions, the MessageReader class enables a user to import messages containing public keys or signatures into the database tables. In either case, a message is selected by long pressing the row. The user is given the option to import the row into the database. The software determines whether the row is a signature or public key and then imports the values into the corresponding database table.

E. PublicKeyRing

The PublicKeyRing class retrieves all public keys from the public key database and formats the retrieved values for the user. Integer values for owner trust and key legitimacy are translated into strings that describe the meanings of the integer values. The number of signatures that are associated with a public key are looked up in the signature database and displayed.

PublicKeyRing allows users to set owner trust levels on individual public key entries by long pressing the row and selecting the desired trust level. Whenever the key ring is viewed or owner trust is updated, PublicKeyRing causes the key legitimacy values for all public keys in the database to be updated.

PublicKeyRing provides the functionality for creating and sharing signatures. Public keys and signatures are shared by selecting the public key and providing the user ID to receive the public key and signatures. The software creates the necessary messages and transmits them using the SMS interface. Signatures are created by selecting the public key, choosing a private key from the private key database and entering the necessary password to decrypt the private key for signature creation. The resulting signature is stored in the signature database.

F. PublicKeyRingEntry

The PublicKeyRingEntry class is a simple container class used for moving single entries from the public key database. The PublicKeyRingEntry performs regular occurring tasks such as key ID creation from public key values and re-creating PublicKey objects from the Bae64 encoded public keys in the database.

G. PrivateKeyRing

The PrivateKeyRing class retrieves all key pairs from the private key database and formats the retrieved values for the user (with the exception of the private key). Public keys may be transmitted to other users (via SMS) from this class. When public keys are transmitted from the Private Key Ring, no signatures are included.

H. PrivateKeyRingEntry

The PrivateKeyRingEntry class is a simple container class used for moving single entries from the private key database. The PrivateKeyRingEntry performs regular occurring tasks such as key ID creation from public key values and re-creating PublicKey objects from the Bae64 encoded public keys in the database. The class does not perform any encryption or decryption of the private key.

I. ECSignature

The ECSignature class performs all digital signing processes in the application. The class uses the default Java signature interfaces but specifically calls out the SHA256withECDSA signature algorithm from the Spongy Castle crypto provider. SHA256withECDSA is an elliptic key digital signature algorithm that signs a SHA-256 digest of the provided bytes.

ECSignature provides two sets of functions. The first set is for creating and verifying public key signatures. The createPKSignature function takes in a private key and public key as arguments and uses the private key to sign the public key. The matching verifyPKSignature takes in a public key signature, the matching public key and the signer’s public key to verify the signature.

The second set of functions is for creating message signatures. The signature creation function takes a byte array and creates the signature using the provided private key. The verification function takes a signature, the matching byte array and the public key used to create the signature.

J. AESEncoder

The AESEncoder class performs all AES-256 encryption. The encoder implements AES as a padded block cipher.

K. ECIESwithAESandDSA

Pre-existing algorithms within the Spongy Castle crypto provider have two shortcomings. First, Bouncy Castle provides an ECIESwithAES algorithm but this class is not available in the current version of Spongy Castle. The closest algorithm to ECIESwithAES in Spongy Castle is the significantly weaker ECIES algorithm. This algorithm uses an elliptic curve Diffie-Hellman key derivation function to generate a key and XORs the key against the byte stream. This leaves the entire system vulnerable to a known plain text attack. The second shortcoming is that default ECIES algorithms create message authentication codes (MAC) instead of digital signatures. The MAC does not meet the PGP requirements of non-repudiation, therefore, a custom algorithm is required.

The ECIESwithAESandDSA class is an elliptic curve integrated encryption scheme. When encrypt is called, a signature is created by the ECSignature class of the passed in data. The resulting signature and original data are concatenated together. The provided keys are fed into an elliptic curve Diffie-Hellman key derivation function provided by Spongy Castle. The resulting 256 bit key is used to encrypt the data plus signature.

The decrypt function works in a similar manner; the provided keys are fed into the same key derivation function, the resulting key is used to decrypt the cipher text and the signature is verified using the ECSignature class.

L. SmsReceiver

The SMSReceiver class simply runs and listens for incoming SMS messages. When an SMS message is received,
the listener provides an alert to the user regarding message reception.

V. CONCLUSION

While PGP Messaging begins to solve the issue of secure SMS messaging, additional work and research can be performed to improve it. For instance, random numbers should be introduced into the key derivation function so messages containing the same plaintext never look the same. One repercussion of implementing this is further reducing the amount of transmittable characters in a message. PGP allows users to specify which symmetrical encryption algorithms to use. Currently this application does not provide that ability but could be enhanced to do so. PGP Messaging also does not support certificate revocation which should be implemented in the future to further increase security.

REFERENCES

VI. APPENDIX

The following screen shots were taken while the application was running via an emulator from within the Eclipse development suite.

Figure VI-1: Initial application screen with options to generate key pair and view the Public and Private Key Rings.

Figure VI-2: Key pair generation screen. Password is entered to perform encryption of the private key.

Figure VI-3: Private Key Ring.
Figure VI-4: Sending public key from the Private Key Ring to another user.

Figure VI-5: Public Key Ring.

Figure VI-6: Options available from the Public Key Ring.

Figure VI-7: Sending public key and associated signatures from the Public Key Ring to another user.
Figure VI-8: Signing a public key with a selected private key.

Figure VI-9: Setting owner trust from the Public Key Ring.

Figure VI-10: Received public keys and signatures.

Figure VI-11: Message composition.
Figure VI-12: Received messages, encrypted.

Figure VI-13: Received message, unencrypted. Yellow triangles indicate marginal trust in public key.

Figure VI-14: Received message, unencrypted. Green checkmarks indicate complete trust in public key.