A Secure Decentralized Messaging Platform Using Encryption and Social Networks

Earl Lee, Peter Nguyen, and Marvin Qian

Yale University, CPSC 426

Introduction

Our original Peerster has good functionality. It can broadcast messages across a network of nodes, allowing connected nodes to be involved in an open chat room of some sort. It also allows for direct peer-to-peer messaging using a decentralized message transfer scheme as well as file sharing in a similar manner. However, in order for our Peerster to meet industry standards and be a viable tool in the real world, we must incorporate some form of encryption to secure direct messages. Note that open messages spread across the network using rumor mongering do not need encryption, since they are meant for every receiving node to read. However, direct messages serve a different use case. They are meant to be private and should not be exposed to anyone but the two parties involved.

Thus, we propose to adapt a version of Pretty Good Privacy (PGP) to encrypt direct messages. The primary components of PGP include public-key and symmetric-key encryption, digital signatures, and a web of trust, each of which we will implement in our Peerster.

In addition to this basic level of security, we want to add an additional layer of security in the form of friendships. PGP will allow us to establish a baseline of trust and then distribute keys through our web of trust, but we may not want to automatically trust everyone within that web. To allow for a more exclusive form of

trust, users will need to send encrypted and signed friend requests and responses to each other. Users will be able to transmit direct messages only once a friendship is established. With the level of trust that a friendship represents, users can feel more secure when sharing files if they can share it with only their friends.

Key Transfer

One of the most critical tasks in securing point-to-point messages in Peerster is to establish a secure way to advertise public keys. The system will use a simple web of trust mechanism that relies on direct connections and manually selecting trusted peers. Peerster will display all peers (IP:port pairs) that are directly connected to the user's node and the user can select those that it trusts and directly notifies those nodes that it wants to merge their respective webs. If both peers agree to the merge, they will trade information about each other's webs (a set of origin names and their public keys) and then propagate it through their web of trust via a broadcast message. Thus, everyone in the web of trust will automatically be connected by transitive, direct trust links.

To walk through this process, let's suppose that Alice wants to add Bob to her web of trust. She will first send a message to Bob with two QVariantLists, "Keys" which contains QStrings (origin names) and "Names" which contains QByteArrays (public keys). Since Alice isn't already a trusted link, Bob will inspect the lists and decide if he wants to accept the connection. If he chooses to accept, he will send Alice a message with the origin names and public keys form his web and then propagate all the new keys to the rest of his web through his other direct links (and his links, since they are part of Bob's web of trust, will pass it on to their links and so on). Alice, when she

receives Bob's reply, will pass on all of Bob's keys to her direct links which will propagate through her web. As nodes relay the keys, they only pass on the ones that they have not yet encountered to prevent loops (since if they have encountered the key before they should have passed it on to all of their links before). At the end of this process, every node in Alice's and Bob's webs will have each other's public keys and thus the two webs will be merged. If Bob chooses not to connect with Alice, he simply does not respond—in the UI this is done by closing a request dialogue box.

While this approach is simple, it has a few drawbacks. One is that broadcasting one node's public key and origin name pair will have to span that entire connected component of the web of trust. Furthermore, it will trigger responses from all newly trusted nodes which will also increase network traffic. Instead of doing this any time a node wants to connect to a new, untrusted node, the nodes could trigger this process periodically (e.g., once per minute) and discover new, trustable nodes that way. Another way to resolve this is to try to accomplish the transfer through direct messages instead of broadcast along the web of trust. The next hop entry for each origin is not a feasible approach since there is no guarantee that that hop is a trusted peer and, even if it were trusted, there is no guarantee that the closest trusted hop has a path to the target peer. One possible solution is to keep a list of all next hops that are trusted regardless of latency. This could significantly reduce network traffic, but may not be part of the initial implementation. In the end, we decided to go with the simple broadcasting method The final issue is that it might still not be enough privacy. The web of trust will likely weed out any legitimately malicious nodes, but just being

not-malicious may not be enough to be willing to accept private messages from them. To remedy this, we propose a simple social-network to establish friendships.

Social Network

Friendships establish reciprocal trust at the level of sending private messages and sharing specific files. A friend request will appear much like an ordinary direct message except that, instead of a “ChatText” field, there will be a “FriendRequest” field with an encrypted, randomly generated QString. To accept the friend request, the node will respond with a similar message except the “FriendRequest” will become a “FriendResponse,” and it will contain the QString that was received in “FriendRequest” but encrypted for the initiator to read and verify. To deny the friend request, the node simply does not respond.

Before friendship is established, direct messages from non-friends will be automatically dropped. After two nodes become friends, then direct messages will be properly passed between them. Furthermore, nodes will be able to share files that are only visible to their friends. The user will be able to specify if a file should only be shared to friends when sharing as well as see a visual distinction between search results from friends and those that are just within the same web of trust (search results from those outside of your web of trust are intentionally dropped). All point-to-point messages are encrypted so other user’s will not be able to garner any information about what search results the user received or what files the user decided to download.

Sending and Receiving Messages

Once two parties are friends, they can begin to send secure private messages. Suppose Alice is trying to send a private message to Bob. She will first generate a random key for symmetric cryptography. She will then use the random key to encrypt her message to Bob. After that, she'll encrypt the random key with Bob's public key. Next, she'll hash the original, unencrypted message to create a digest. Finally, she'll encrypt the digest with her private key to create a signature. The encrypted digest and encrypted random key will be additional fields in the message datagram that is sent to Bob.

When Bob receives the message, he will first decrypt the random key with his private key. Then he will use the random key to decrypt the message. Following that, he'll verify the signature by comparing the signature and decrypted message using Alice's public key. If signature verification succeeds, then the message from Alice reached him intact. If not, he'll assume that someone tampered with the message and discard it.

Suppose a malicious third party, Mallory tries to intercept the message along the path from Alice to Bob. There is no way for her to discover or modify the contents of the message. She can't decrypt the random key since she does not have Bob's private key so she cannot read the message. If she tries to modify the message, the digest won't match. If she tries to modify the digest too, it still won't match since it needs to be signed by Alice's private key which she also does not have. Thus, she cannot create a man-in-the-middle attack except for throwing away all or some of the messages.

To support potentially encrypted messages in addition to regular messages, the project restructures the message format. Since routing information is needed regardless of whether the message is encrypted, the new message format separates actual message content from routing info. This is achieved by having the top-level QVariantMap that represents a datagram contain a “Message” key that points to a QByteArray value which is a representation of another QVariantMap that contains the actual message content relevant to the receiving node. The “Message” value is a QByteArray instead of a QVariantMap so that it can be encrypted if necessary. The top-level QVariantMap will also contain any relevant routing info (e.g., Origin, Dest, Budget, HopLimit, LastIP, and LastPort) as well as a “Type” key that will map to a QString specifying how the message should be processed (e.g., private, rumor, search, etc.). Finally, if the message is encrypted, then the top-level QVariantMap will also contain a “Key” key that maps to the encrypted random key and a “Signature” key that maps to the signature for the encrypted message.

Results

We managed to achieve all of the functionality that we set out to implement, and our improved Peerster runs well, having tested up to 10 nodes at a time.

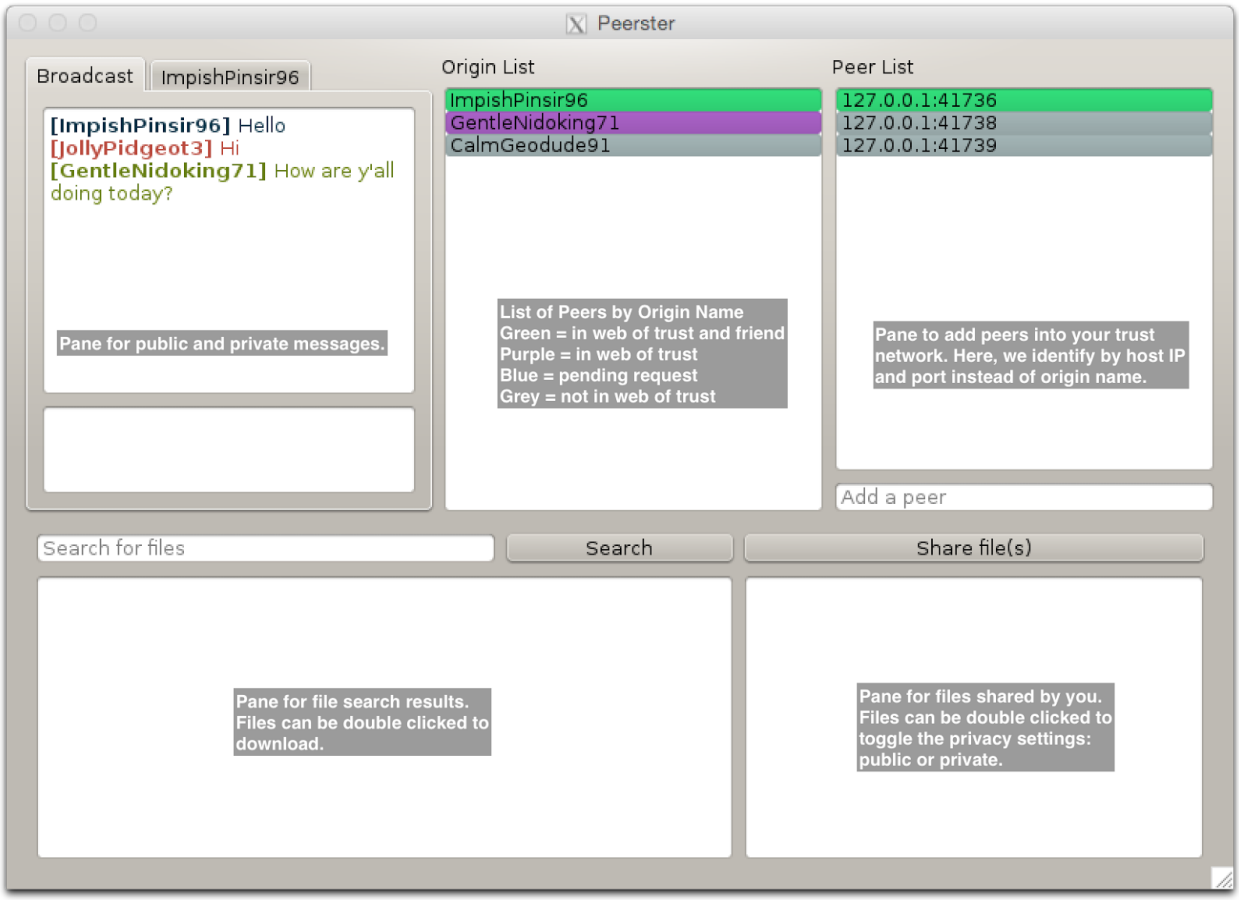

Fig. 1: The UI for our improved Peerster app with annotations.

With our Peerster, you can add new peers in the Peer List pane. Grey indicates peers that have are not direct trust links while green indicates that they are direct trust links. To send a trust request that upon acceptance will merge the current node and the requested node's web of trust, the user can simply double click that peer (the peer will be blue to mark it as pending until the requested peer accepts). The Origin List pane lists named origins that you are connected to throughout the network (whereas peers lists only direct connections). Each origin is highlighted a different color depending on the state. Green indicates a friend and in your trust network, purple indicates a node in your trust network, blue indicates a pending request, and

grey indicates that the node is not in your trust network, implying it is not your friend. The user can send a friend request by double clicking on an origin. In the search results pane, files in green are from friends, while those in purple are from nodes just in your web of trust. The shared files pane follows a similar color scheme in which purple files are shared publicly in the user's web of trust whereas green files are shared only to friends. The user can double click a shared file to change its privacy setting.

Work Report

Marvin worked on implementing the friend network module, selective file sharing, and UI. Marvin changed ChatDialog.cc/hh for the ability to send friend and trust requests as well as see a list of peers and their states—in your trust network, friends, or neither. He refactored and added new code to parts of NetSocket.cc/hh and PeerList.cc/hh for handling friend requests/replies, search requests/replies, and message forwarding. Marvin also modified Origin.cc/hh and SharedFile.cc/hh to delineate difference between public and private files.

Peter worked on implementing the web of trust module. Peter made changes to NetSocket.cc/hh and PeerList.cc/hh to create trust requests and integrate the encryption/decryption module into messages.

Earl worked on the encryption/decryption module. He created PBP.cc/hh to serve as a plug-and-play module that can be used to encrypt and decrypt message. He made sure the module could easily be added in as a layer in the message processing/routing pipeline.

In order to complete our project, we had to get new QT cryptographic libraries installed to support the use of ciphers in encryption/decryption. Luckily, we were able to do so and planned out our project well enough so that work could be done independently on different modules and easily integrated together.

Conclusion

In our naive implementation of PGP, we can successfully prevent third-party nodes from reading or tampering with messages sent between two nodes engaging in direct messaging. Hostile nodes will not be able to make sense of encrypted data, and any tampering would be evident upon decryption—so long as the hostile nodes do not have access to private keys. The friend request extension on top of that provides another layer of privacy and security, and it even allows for more restricted sharing of files.

Further improvements to our project can be made by finding a better, more streamlined way to quickly build genuine networks of trust. Currently, we rely on manually connecting users, but there is little real verification behind this. Messages could also be tampered with and not sent to the proper destination or sent at all if intercepted, since routing information is not encrypted. Lastly, we would like to add some convenience features such as unfriending people or removing nodes from a web of trust.

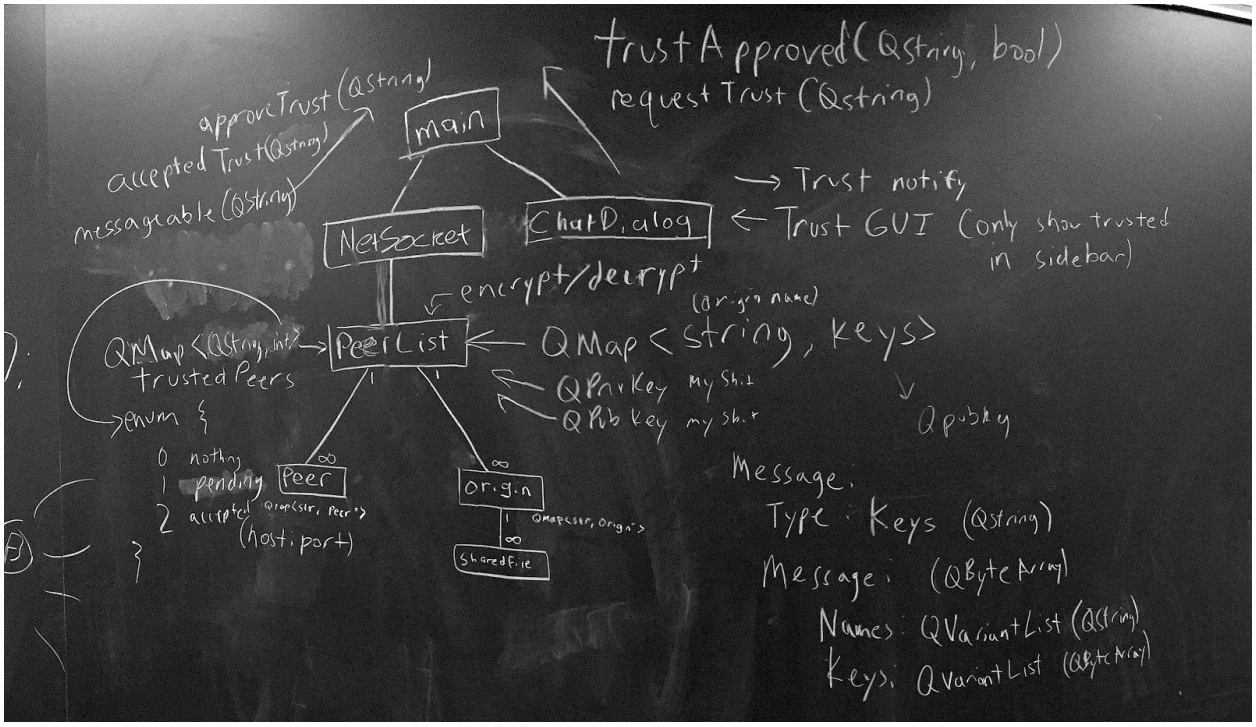

Fig. 2: A draft of our software architecture.

Detailed Specifications

The general message format has been changed to have two levels. The main level (regular keys in the QVariantMap) include all the routing data and then a “Message” key with a QByteArray value that contains the actual message content and may or may not be encrypted (which is why it is a QByteArray representation of a QVariantMap instead of just the QVariantMap). Thus, the datagram QVariantMap contains the following keys (with associated types).

- Message - QByteArray (serialized version of a QVariantMap of the actual data)

    - actual message contents which includes all fields that are not listed in the “Possible routing information” section below
    - represented as a byte array since it is possibly encrypted
- Type - QString (possible values listed below; SearchReply, Private, BlockRequest, BlockReply, FriendRequest, and FriendReply are point-to-point so they have encrypted Message QByteArrays)
    - SearchRequest
    - SearchReply
    - Private
    - Rumor
    - BlockRequest
    - BlockReply
    - Status
    - Trust
    - FriendRequest
    - FriendReply
- Possible routing information (follows original specification)
    - Origin - QString
    - Dest - QString
    - Budget - quint32
    - HopLimit - quint32
    - LastIP - quint32
    - LastPort - quint16
- If encrypted, includes

    - Signature - QByteArray
        - Signature resulting from signing the Message QByteArray with the sender's private key (uses QCA::EMSA3_MD5 for hashing)
    - Key - QByteArray
        - Symmetric key for AES128 in CBC mode encrypted by RSA (QCA::EME_PKCS1_OAEP) using the receiver's public key

Below are the new message types that we have added/modified and some specifications on how to handle them. All other message types are the same except that they follow the new message structure described above, have a "Type" key, and may have their "Message" value encrypted.

- Private (dropped if not from a friend)
    - Dest - QString
    - Origin - QString
    - HopLimit - quint32
    - Signature - QByteArray
    - Key - QByteArray
    - Type - QString
    - Message - QByteArray
        - ChatText - QString
- Trust (Initial direct trust request if from an untrusted node, key propagating message if from a trusted node)
    - Type - QString
    - Message - QByteArray

        - each QString key is a name of an origin with the associated value being that origin's public key as a QByteArray
- FriendRequest/FriendReply (need to track random QStrings that FriendRequests are sent with to verify that the person responded with an appropriate FriendReply)
    - Dest - QString
    - Origin - QString
    - HopLimit - quint32
    - Signature - QByteArray
    - Key - QByteArray
    - Type - Trust
    - Message - QByteArray
        - FriendRequest/FriendReply - QString (random) which is sent back to verify