

CSU34031 Advanced Telecommunications

Project #2

”Matador”

Multi-User Secure Messaging System

Jack Harley jharley@tcd.ie — Student No. 16317123

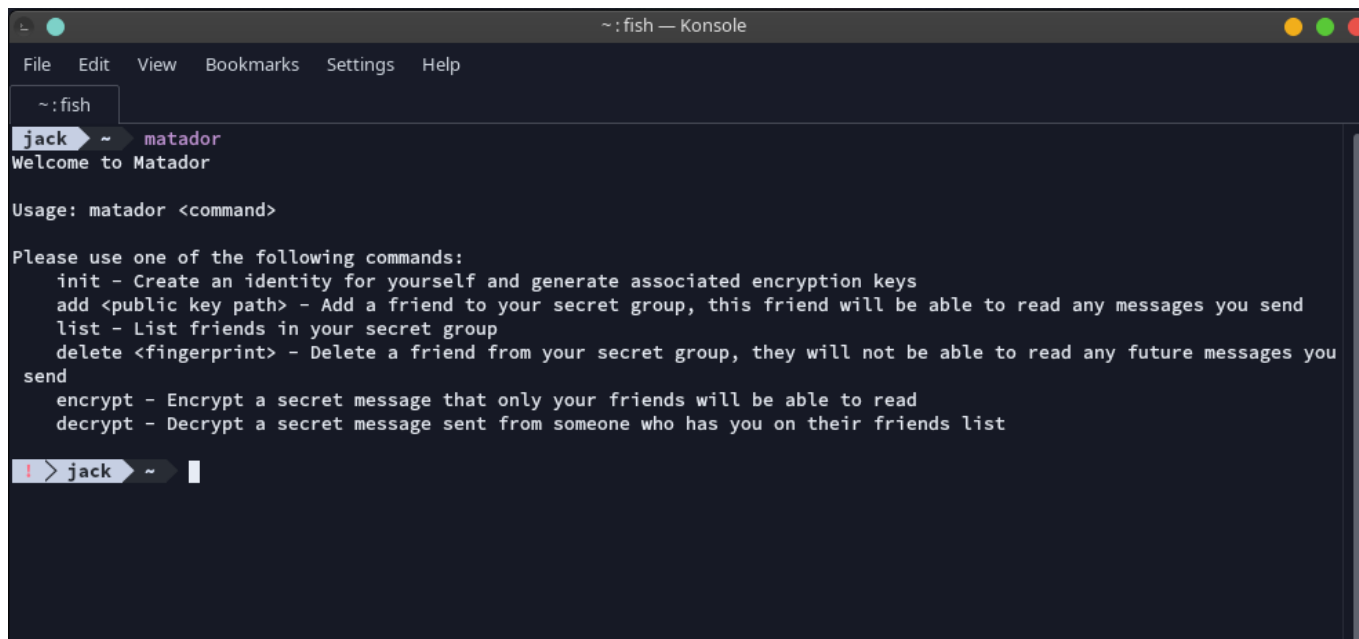
April 2020

YouTube Demo: <https://www.youtube.com/watch?v=c0bWx81WbeI> (watch in 1080p HD)
GitHub: <https://github.com/JackHarley/Matador>
Binaries for Download: <https://github.com/JackHarley/Matador/releases>

Contents

1	Introduction	2
1.1	Codebase Structure	3
1.2	Encryption Technologies/Libraries	3
2	Usage/Demo	3
3	Data Formats/Protocol	4
3.1	Encrypted Message Format	4
3.2	Public and Private Keys	5
4	Processes/Functionality	6
4.1	Message Encryption	6
4.2	Message Decryption	6
4.3	Public Key Fingerprint Calculation	7
4.4	Key Management	7
4.4.1	Local User	7
4.4.2	Friends/Recipients	7
5	Appendix: Code Listings	8
5.1	main.go	8
5.2	identity.go	9
5.3	friends.go	11
5.4	message.go	13

1 Introduction



```
~: fish — Konsole
File Edit View Bookmarks Settings Help
~: fish
jack ~ matador
Welcome to Matador

Usage: matador <command>

Please use one of the following commands:
  init - Create an identity for yourself and generate associated encryption keys
  add <public key path> - Add a friend to your secret group, this friend will be able to read any messages you send
  list - List friends in your secret group
  delete <fingerprint> - Delete a friend from your secret group, they will not be able to read any future messages you
send
  encrypt - Encrypt a secret message that only your friends will be able to read
  decrypt - Decrypt a secret message sent from someone who has you on their friends list

! > jack ~
```

I have built my solution in GoLang. GoLang is a statically typed, compiled programming language designed at Google, it is syntactically similar to C, but with extra features including memory safety, garbage collection, structural typing.

My system is named Matador, in reference to a programme that is something of its opposite- the NSA's BULLRUN global cryptanalysis programme. Provided the NSA has not in fact secretly developed a quantum computer with enough qubits to break RSA-2048, Matador should hopefully be resistant to their abilities. Alas we have no way of knowing if they may have discovered another vulnerability in RSA/AES but I personally do not believe they have.

Matador is a simple system which allows you to generate an "identity" for yourself (an RSA-2048 public-private keypair) and then add "friends" public keys to your database. You can encrypt messages via the command line interface, and your friends will be able to decrypt the ciphertext using their private keys. You can remove "friends" from your database at any time, and any future messages you encrypt will not be able to be decrypted by them.

Matador is a fully production ready application. It could be used in a scenario where a group wishes to transmit messages to each other through an existing social media platform such as Facebook. Each user would generate an identity, post their public key on the social media site or transmit it in some other way, and then after importing all their friends' public keys they could then encrypt messages to post publicly, safe in the knowledge that they are unreadable to unauthorized viewers.

The full source code is included as listings in the appendix, and is also available on GitHub here: <https://github.com/JackHarley/Matador>.

1.1 Codebase Structure

The project consists of 4 Go source files all of which are in the main package, as is typical when structuring small GoLang programs. It would in fact be possible to merge all the code into a single file with little to no changes but this would make the project more difficult to navigate and comprehend.

The 4 main source files are:

1. **main.go** - This is home to the entrance routine of the program, it has very little functionality. It can display help text when no command is supplied, and when a command is supplied it calls the relevant function from one of the other files.
2. **identity.go** - Implements functionality relating to the local user's identity. It can generate a new identity with the init command (generates an RSA-2048 public-private keypair) and retrieve the user's private and public keys, and the special fingerprint of the public key which we use to identify it quickly.
3. **friends.go** - Implements functionality relating to the public keys of the user's friend group. Handles storing the friendly public keys in a simple JSON file database along with their fingerprints and the user-entered name for each of them. Allows the user to add, delete and list the entries in the database.
4. **message.go** - Implements the actual encryption/decryption of messages for transmission.

1.2 Encryption Technologies/Libraries

Matador utilizes GoLang's built-in encryption libraries to perform all of its cryptography. These libraries are one of GoLang's strong points and part of the reason I chose the language for this assignment. By using built-in, well-vetted libraries we can be certain there are no amateurish errors that could open us to an attack. This is in stark contrast to something like the Javascript ecosystem, where developers tend to use whatever NPM library comes up first on their Google search, many of which have dependency chains too long to properly analyse, with possible security-critical hidden bugs beneath the depths.

We utilize both symmetric and asymmetric encryption in Matador. For symmetric we use the well-known Advanced Encryption Standard (AES) with a 256 bit key-size. We use the Galois/Counter Mode of operation when operating the cipher. Together this combination is commonly referred to as AES-GCM 256.

For asymmetric encryption we use RSA-2048. RSA has a couple of notable drawbacks, for us the principal one is the size of the outputs it generates, which are always 2048 bits (256 bytes). It will become apparent later on why this can be an issue when a user has a large friends list.

2 Usage/Demo

Please watch the following short YouTube video to see a full demonstration of Matador's functionality: <https://www.youtube.com/watch?v=c0bWx8lWbeI>

You can also try out the program yourself! I have published it on GitHub here: <https://github.com/JackHarley/Matador>

There are Windows x64 and Linux x64 binaries available on the releases page here: <https://github.com/JackHarley/Matador/releases>. Simply download the appropriate binary and execute it in a command line environment.

3 Data Formats/Protocol

3.1 Encrypted Message Format

A Matador encrypted message looks like the following:

```
1 -----BEGIN MATADOR ENCRYPTED MESSAGE-----
2 MIICzAQMv5uZlq8UXVxMK1q8BCRGelhIvqA7vAvetX1tsKgPs3y4B/7VvYpC3GOL
3 5vq0vegi2twggKUMIIBRhNAMzJhODEwYjgOMTdmZDE3NDUxZDRiMzQxNGYwMzI3
4 ODNhZjI5NzI4NTAwMDJhOGIyM2Q3OTRlYWw1YjY2N2JjZASCAQAATL8u+OrtPDE3
5 0W+09FrX+hhhXxt0cx1EAaxyduIoK0h2Lt1l1M4PJPccsQUHh5tpuA2oVi5qU+1X
6 LoF4SEd1AMbJeJu5MMINn17kxQV3iyD0TvLyM4RZuYC9PLTBLQmlmwFaUStjZgDb
7 2TnJXM+1dXYOB2VzLcx15ecW8dB+tpB0kaXQyUkAay8Fe8824nyf2ep7sTTCkaTs
8 quDVsmzDlIBfg94t4BFHnu/MdK3KDN2X3xmWzoB7iIffvgBZQK/nqTpAG/CdvBvO
9 MZHwx+67Rg6ssIPFGCVP2YI3hvUF4FXInncbSYSOJ3iDzEjLJPj2Ca+IRT+zSct
10 6uazym1NMIIBRhNANTgzZjFhODA2MDU3YTYOM2M2NzQxYzd1OTI5Y2QyNzQOMjVj
11 NjRhNmIzYjBiYmU1NTgyY2Y3NzMyNWY1N2FhNwSCAQCC60whPvgJxwLruQ/xbx88
12 Rz6KsChmBB8b0x7/RrTwAcnJDbSFUQYJRjNzSfbK+A35RHZtrr7itmfoIOwrVuK
13 vdBQIKclZ5aut9ezI6d3jgIR2EKgd96ZM+jY1VbHrFKcG83fU00WMeRT8TDTdPZN
14 7aF0bnMdfLTOEMqCwFfJRSr6L4T9dTxdLLeZGZZrj8dDX71YUv/BR3pVR4EWxow
15 UadpTB0ZM2EWqtUite/DUL+Lik4SoAlRsQQJn9gruMSCBH0+ZraDxSnLhUuShrpP
16 DRV2sClyxXZCbbaxi2Yx3pJMGOr0McfF0JzmUR7ErdTz/Hrvhpo/Ga+fLSi/0Dd
17 -----END MATADOR ENCRYPTED MESSAGE-----
```

The format will instantly be familiar to anyone used to dealing with cryptography. The message is in a PEM encoded format, with ASN.1 data inside. Let's take a look inside the ASN.1 data. We can dump the message into an ASN.1 decoder like that available here to start: <https://lapo.it/asn1js/>:

```
SEQUENCE (3 elem)
  OCTET STRING (12 byte) BF9B9996AF145D5C4C2B5ABC
  OCTET STRING (36 byte) 467A5848BEA03B8C0BDEB57D6D0A80FB37CB807FED58D8A42DC6D0BE6FA8EBDE822DA...
  SEQUENCE (2 elem)
    SEQUENCE (2 elem)
      PrintableString 32a810b8417fd17451d4b3414f032783af2972850002a8b23d794eaceb667bcd
      OCTET STRING (256 byte) 004CBF2EF8EAED3C3137D16F8EF45AF1FA18615F1B7473194401AC7276E2282B48762...
    SEQUENCE (2 elem)
      PrintableString 583f1a806057a643c6741c7e929cd274425c64a6b3b0bbe5582cf77325f57aa7
      OCTET STRING (256 byte) 9CEB4C213EF809C702EBB90FF16F1F3C473E8AB1C1E6041F1BD31EFF4684F001C9C90...
```

Here's what you're seeing above:

- The first octet string (12 bytes) is the unique nonce used for the AES-GCM 256 encryption.
- The second octet string (36 bytes) is the ciphertext of the encrypted message.
- We then have a list of objects each containing two elements, each object represents a user with permission to decrypt the message:
 - The printable string is a hex encoded SHA256 fingerprint of the public key (further detail on how this is calculated will be discussed later).
 - The octet string is the ciphertext of the RSA encrypted session key.

If we were the user in receipt of this encrypted message we would need to fingerprint our own public key and compare it against each of the session key objects until we find a match. At that point we can use our RSA private key to decrypt the AES-256 session key. We can then use the session key in combination with the plaintext nonce to decrypt the main message ciphertext.

Earlier I mentioned a disadvantage of RSA-2048 being that the ciphertext is always 2048 bits (256 bytes) long (when used securely with correct padding). This disadvantage rears its head here. For every user that has access to the message, there is an extra 256+32 bytes added to the transmission. This can result in small single sentence messages becoming extremely long even when encoded with ASN.1/PEM (which results in base64 encoding), and with enough users involved you may start hitting character limits on social media services. Using an elliptic curve based asymmetric solution could shorten these outputs however the libraries to do so are not as mature in GoLang, and so I opted against it.

3.2 Public and Private Keys

RSA public and private keys are stored in PKCS1 format in a PEM file. When generating an identity, two files are created for the user (RSA public and RSA private key-files). Examples are shown below:

private_key.pem:

```
1 -----BEGIN RSA PRIVATE KEY-----
2 MIIEpAIBAAKCAQEAvmKQ8/MbCc7bny93uDZjkIUc8vGNQrf9BZkoovLse0Un1TZ4
3 1dWYEcyIHgcrxgHRRtnPrWr3DuW7IGUFQAZ+DZYbyh16BNEPyY0hIG9TlCKL+1P
4 IFqY5iDV5nHx0jDKQjR5DBoM9jVwVyxRe5UKhmxZzWVW1BZ1UbdKCi7NsOp3AjEN
5 7tw0BwvkbHYEELIfy0wu5Y1+6+b46a271/rzmIqz40t8UpMD/sb5evyPnNtYtEGk
6 mKOE20zd+5V/1ywkjT2DUN4v9R30nLkxiGTFc9RxBcuwbp8fhqRWhUZJn/cvUyGi
7 gLYvIxJPZ5139T+1UFSb1eFavDG4CxbejoA6vwIDAQABAoIBACCX2tblNCTpi4yi
8 wwpzxHZNuZOCWfqhI1xi6eF5Uz8fGMIUoHl4JJ5nMqRw9bzdcsN9Y116g027w1q3
9 7z4V3lzU48yjXs6UON3vG+3RhmRwDWPl4sh9ijsVa8qsQgX8YWDFXRMwQsh+ifx
10 HcDYsQRwZq4QOdD+RDjH9ZPojqwNxx+JyDUn7wfVT6oxtZ7AERWUxE2R5Hy/1WQ
11 zP/KJzvGpY2jjh02e0LbKCz7Z+0Eg3hq87aEGjoRo3BjEA71I3USx01p5qYf+61g
12 R7H4YdCo5t90y70ygs8tUKIOczL0XraDE61Uipso6AzMEaAlBk6E1NbEtC+g31tUJ
13 Fo2LfoECgYEA0ZUJd1xgcKushe7uC4CrPpMLnpFvB4y/MLXAHf8qWhclS4LkSXZ/
14 yTbHtqBcVfcfwpDVkiBhh0mJFP6HVgOURakgiUB54PqLxTINP+030CUPzPdGxf0X
15 U7iUREscSYksX3adLeMxieJCoBDrFn10by62he8B5ZukyA+hcSZzq2UCgYEA6KCg
16 SOVssdU/bRv+TDAEAfTGB1mkvBCxultAqBpdHxU8ZdW3Kgc5XpuByRQITatJnWg
17 ygxifWumTpp0D9oMT94gxALoDDBLr8fmbU17uSeH6nMMhzYeyWHCRIBro0qSC6g
18 r+mMcTKUCODuYp3ra2LJBVKKzKc4mYDBfPVu9VMCgYBt9R0R1cWDV46cdk70Gi0+
19 IMfn2cw24FEu7SxukIFX4uzeQT4kjNdeai890cSgFZfELNip0XNYmG4jZab7qcUR
20 Bzrax8kqEELknG33LC4sNkWj89hd5/tlIWIosn+U1TNCqi0eo5zHHuAH2A33HpK
21 GugIP01x2ccKb6j0BHrnQKKBgQC/fdn1nESJbv0zF+QRW0UxRwjrnF8ffd+6VdSC
22 YQHt/Zr+9JFw0oLu9AU2o9HDlpJUImglamR40fCoq6Sur90KulNFhw716vMCXBnp
23 2x7KtC7l6aRjBc1X0eCCuWFGGuPlX1yu8hVxoz5wwJf7Xfu4/ixz0SqopM6WrcKY
24 1oBsOQKBgQDP1Ypfgm/1qiEPxYGMJhV0wXaW1FY1JmWAnDPHx6GT3YIPugtKXPGM
25 3xJCFteK8VA8e1J299fuLAudOy3Be5U17wLyBWLjW74YfAdz4VJEKrymjKfbXy4M
26 jGQTM+tyCnj/wkrCNg9RMvOAOSccfNFibZihMWvXGcm1vzeBkeeoJw==
27 -----END RSA PRIVATE KEY-----
```

alice.pub.pem:

```
1 -----BEGIN RSA PUBLIC KEY-----
2 MIIBGgKCAQEAvmKQ8/MbCc7bny93uDZjkIUc8vGNQrf9BZkoovLse0Un1TZ41dWY
3 EcyIHgcrxgHRRtnPrWr3DuW7IGUFQAZ+DZYbyh16BNEPyY0hIG9TlCKL+1PIFqY
4 5iDV5nHx0jDKQjR5DBoM9jVwVyxRe5UKhmxZzWVW1BZ1UbdKCi7NsOp3AjEN7tw0
5 BwvkbHYEELIfy0wu5Y1+6+b46a271/rzmIqz40t8UpMD/sb5evyPnNtYtEGkmKOE
6 20zd+5V/1ywkjT2DUN4v9R30nLkxiGTFc9RxBcuwbp8fhqRWhUZJn/cvUyGigLYv
7 IxJPZ5139T+1UFSb1eFavDG4CxbejoA6vwIDAQAB
8 -----END RSA PUBLIC KEY-----
```

4 Processes/Functionality

4.1 Message Encryption

This is the precise process followed to encrypt a message, the code to do so can be found in the `encryptMessage()` function in `message.go`. This process is executed when the command **matador encrypt** is issued.

1. Read an input message from the user, terminate of the message is indication by a single `.` on a new line, this is a common way of indicating termination used in Linux mail programs such as postfix.
2. Generate 256 random bits, this will be the session key
3. Generate 96 random bits, this will be the nonce. **N.B.** Technically a unique nonce is not required since we never reuse the session key (we could just use a hard-coded static nonce/zero nonce), however we will use one anyway since they are standard practice. When attempting to use a zero nonce the Go implementation refused to encrypt the data, so the safest thing to do is simply do what the developers expect you to. The additional data that has to be transmitted with the message is negligible compared to the size of the multiple copies of the asymmetrically encrypted session key.
4. Encrypt the input message using AES in Galois/Counter-Mode with the 256 bit key and 96 bit nonce we generated.
5. Retrieve the RSA public keys of all friends in the database, for each key:
6. Encrypt the session key with the RSA public key.
7. Append the public key's fingerprint and the asymmetrically encrypted session key to a list in the encrypted message structure.
8. Encode the encrypted message structure containing the nonce, encrypted message and encrypted session keys with fingerprints into ASN.1 and further encode that into the PEM format.
9. Output the PEM text to the console for the user to send.

4.2 Message Decryption

This is the precise process followed to decrypt a message, the code to do so can be found in the `decryptMessage()` function in `message.go`. This process is executed when the command **matador decrypt** is issued.

1. Read an input encrypted message PEM from the user, the program understands termination by watching for the PEM `—END—` line.
2. Decode the PEM, giving an ASN.1 encoded byte stream
3. Decode the ASN.1 into our `EncryptedMessage` structure
4. Load the user's private key and calculate its public fingerprint.
5. Compare the fingerprint to the fingerprints attached to each copy of the encrypted session key. If a match is found, we can proceed with decryption using that copy. If no match is found, the user has not been authorized to decrypt the message and the program will abort.
6. Decrypt the session key using the RSA private key.
7. Use the session key and the nonce from the input encrypted message to decrypt the message.
8. Output the decrypted message to the console.

4.3 Public Key Fingerprint Calculation

The public key fingerprints are not important for maintaining security, they are simply a convenience factor allowing the receiving user to immediately know if there is a copy of the session key available that they should be able to decrypt.

The fingerprinting could in fact be entirely removed and the program could instead attempt to decrypt every bundled session key, however this would be computationally wasteful, particularly when a message has been sent for use by a large number of users.

This is the code used to perform the fingerprinting:

```
1 func getPublicFingerprint() string {
2     publicKey := getPublicKey()
3     fingerprint := sha256.Sum256(x509.MarshalPKCS1PublicKey(publicKey))
4     return hex.EncodeToString(fingerprint[:])
5 }
```

As you can see it is quite a simple process.

We simply encode the public key as a PKCS1 byte array, calculate a SHA256 hash of the bytes and then encode that SHA256 hash as a hexadecimal string. The resulting hexadecimal string is the fingerprint.

4.4 Key Management

4.4.1 Local User

The local user's keys are held in files named `private_key.pem` and `name.pub.pem`. They are generated by issuing the command **matador init** which generates and stores the keys in the current directory. Functionality in `identity.go` allows other parts of the program to retrieve the key data from the filesystem whenever required.

4.4.2 Friends/Recipients

The program maintains a database of the public keys of as we will refer to them: "friends". Friends are user's with a public key on file who are authorized to view the messages we encrypt. The command **matador add**, **matador list** and **matador delete** allow the user to respectively add, list and delete friends from the database.

The friends database is maintained in a human/machine readable `friends.json` file. It contains an array of friend objects, each of which having a name assigned to it by the user, its fingerprint (which is used to refer to it if deletion is required) and the full public key which is used during message encryption.

5 Appendix: Code Listings

Listings are optimized for PDF readability by limiting line length to 110 characters.

5.1 main.go

```
1 package main
2
3 /**
4  * Matador Multi-User Messaging Encryption System
5  *
6  * Copyright (c) 2020, Jack Harley, jackpharley.com
7  * All Rights Reserved
8  */
9
10 import (
11     "fmt"
12     "os"
13 )
14
15 func main() {
16     if len(os.Args) < 2 {
17         helpText()
18         os.Exit(1)
19     }
20
21     friendsSetup()
22
23     command := os.Args[1]
24
25     switch command {
26     case "init":
27         initIdentity()
28     case "add":
29         addFriend()
30     case "list":
31         listFriends()
32     case "delete":
33         deleteFriend()
34     case "encrypt":
35         encryptMessage()
36     case "decrypt":
37         decryptMessage()
38     }
39 }
40
41
42 func helpText() {
43     fmt.Println("Welcome to Matador")
44     fmt.Println("")
45     fmt.Println("Usage: matador <command>")
46     fmt.Println("")
47     fmt.Println("Please use one of the following commands:")
48     fmt.Println("    init - Create an identity for yourself and generate associated encryption keys")
49     fmt.Println("    add <public key path> - Add a friend to your secret group, this friend will be " +
50         "able to read any messages you send")
51     fmt.Println("    list - List friends in your secret group")
52     fmt.Println("    delete <fingerprint> - Delete a friend from your secret group, they will not be " +
53         "able to read any future messages you send")
54     fmt.Println("    encrypt - Encrypt a secret message that only your friends will be able to read")
55     fmt.Println("    decrypt - Decrypt a secret message sent from someone who has you on their " +
56         "friends list")
57     fmt.Println()
58 }
```


5.2 identity.go

```
1 package main
2
3 /**
4  * Matador Multi-User Messaging Encryption System
5  *
6  * Copyright (c) 2020, Jack Harley, jackpharley.com
7  * All Rights Reserved
8  */
9
10 import (
11     "bufio"
12     "crypto/rand"
13     "crypto/rsa"
14     "crypto/sha256"
15     "crypto/x509"
16     "encoding/hex"
17     "encoding/pem"
18     "fmt"
19     "io/ioutil"
20     "os"
21     "strings"
22 )
23
24 // initIdentify initializes a new identity, prompting for a name on the CLI and generating a
25 // public private keypair
26 func initIdentify() {
27     if _, err := os.Stat("private_key.pem"); err == nil {
28         fmt.Println("You already have an identity generated, please delete your private_key.pem " +
29             "file and run the init command again if you would like to create a fresh identity.")
30         os.Exit(1)
31     }
32
33     stdin := bufio.NewReader(os.Stdin)
34
35     fmt.Printf("Please enter the name you would like to be known as and press enter: ")
36     name, err := stdin.ReadString('\n')
37     if err != nil {
38         fmt.Println("Failed to read your name, exiting, please restart the program and try again.")
39         os.Exit(1)
40     }
41
42     name = strings.TrimSpace(name)
43     nameForFile := strings.Replace(name, " ", "", -1)
44     nameForFile = strings.ToLower(nameForFile)
45     keySize := 2048
46     randSource := rand.Reader
47
48     // generate key
49     fmt.Printf("Generating key...")
50     key, err := rsa.GenerateKey(randSource, keySize)
51     if err != nil {
52         fmt.Println("Failed to generate keys, please restart the program and try again.")
53         os.Exit(1)
54     }
55     fmt.Printf("done!\n")
56
57     fmt.Printf("Saving keys...")
58
59     // save private key
60     privateOut, err := os.Create("private_key.pem")
61     defer privateOut.Close()
62     if err != nil {
63         fmt.Println("Unable to save private key, please restart the program and try again.")
64         os.Exit(1)
65     }
66     privateKeyPem := &pem.Block{
67         Type:  "RSA PRIVATE KEY",
68         Bytes: x509.MarshalPKCS1PrivateKey(key),
69     }
70     err = pem.Encode(privateOut, privateKeyPem)
```

```

71     if err != nil {
72         fmt.Println("Unable to save private key, please restart the program and try again.")
73         os.Exit(1)
74     }
75
76     // save public key
77     publicOut, err := os.Create(nameForFile + ".pub.pem")
78     defer publicOut.Close()
79     if err != nil {
80         fmt.Println("Unable to save public key, please restart the program and try again.")
81         os.Exit(1)
82     }
83     publicKeyPem := &pem.Block{
84         Type:  "RSA PUBLIC KEY",
85         Bytes: x509.MarshalPKCS1PublicKey(&key.PublicKey),
86     }
87     err = pem.Encode(publicOut, publicKeyPem)
88     if err != nil {
89         fmt.Println("Unable to save public key, please restart the program and try again.")
90         os.Exit(1)
91     }
92     fmt.Printf("done!\n\n")
93
94     fmt.Println("Identity generated succesfully, please make sure to restrict access to this " +
95         "directory from unauthorized users.")
96     fmt.Printf("You should send your %s.pub.pem key to all other members of your group.\n", nameForFile)
97 }
98
99 func getPrivateKey() *rsa.PrivateKey {
100     privateIn, err := ioutil.ReadFile("private_key.pem")
101     if err != nil {
102         fmt.Println("No private key found, or unable to read it. Please run matador init if you " +
103             "need to generate an identity.")
104         os.Exit(1)
105     }
106
107     block, _ := pem.Decode(privateIn)
108     if block == nil || block.Type != "RSA PRIVATE KEY" {
109         fmt.Println("Private key file is corrupt, please restore it from a backup or " +
110             "generate a new identity.")
111         os.Exit(1)
112     }
113
114     key, err := x509.ParsePKCS1PrivateKey(block.Bytes)
115     if err != nil {
116         fmt.Println("Private key file is corrupt, please restore it from a backup or " +
117             "generate a new identity.")
118         os.Exit(1)
119     }
120
121     return key
122 }
123
124 func getPublicKey() *rsa.PublicKey {
125     privateKey := getPrivateKey()
126     publicKey := privateKey.Public()
127     rsaPublic := publicKey.(*rsa.PublicKey)
128     return rsaPublic
129 }
130
131 func getPublicFingerprint() string {
132     publicKey := getPublicKey()
133     fingerprint := sha256.Sum256(x509.MarshalPKCS1PublicKey(publicKey))
134     return hex.EncodeToString(fingerprint[:])
135 }

```

5.3 friends.go

```
1 package main
2
3 /**
4  * Matador Multi-User Messaging Encryption System
5  *
6  * Copyright (c) 2020, Jack Harley, jackpharley.com
7  * All Rights Reserved
8  */
9
10 import (
11     "bufio"
12     "crypto/sha256"
13     "crypto/x509"
14     "encoding/base64"
15     "encoding/hex"
16     "encoding/json"
17     "encoding/pem"
18     "fmt"
19     "io/ioutil"
20     "log"
21     "os"
22     "strings"
23 )
24
25 type friendsContainer map[string]friend
26 type friend struct {
27     Name           string
28     PKCS1PublicKey string
29     Fingerprint    string
30 }
31
32 var fCont friendsContainer
33
34 func init() {
35     fCont = make(map[string]friend)
36 }
37
38 func friendsSetup() {
39     attemptLoadFriends()
40 }
41
42 func saveFriends() {
43     f, _ := json.Marshal(fCont)
44     e := ioutil.WriteFile("friends.json", f, 0644)
45     if e != nil {
46         log.Fatal("Failed to save friends to file: " + e.Error())
47     }
48 }
49
50 func attemptLoadFriends() {
51     f, _ := ioutil.ReadFile("friends.json")
52     json.Unmarshal(f, &fCont)
53 }
54
55 func addFriend() {
56     if len(os.Args) < 3 {
57         fmt.Println("You must provide a path to the public key file for the friend you are adding, " +
58             "please try again")
59         os.Exit(1)
60     }
61
62     publicIn, err := ioutil.ReadFile(os.Args[2])
63     if err != nil {
64         fmt.Println("Unable to find public key at the specified path, please try again.")
65         os.Exit(1)
66     }
67
68     block, _ := pem.Decode(publicIn)
69     if block == nil || block.Type != "RSA PUBLIC KEY" {
70         fmt.Println("Public key file provided is not valid and/or is corrupt, please try again.")
71     }
72 }
```

```

71     os.Exit(1)
72 }
73
74 publicKey, err := x509.ParsePKCS1PublicKey(block.Bytes)
75 if err != nil {
76     fmt.Println("Public key file provided is not valid and/or is corrupt, please try again.")
77     os.Exit(1)
78 }
79
80 fmt.Println("Key opened successfully.")
81
82 stdin := bufio.NewReader(os.Stdin)
83 fmt.Printf("Please enter the name of the person who owns this key: ")
84 name, err := stdin.ReadString('\n')
85 name = strings.TrimSpace(name)
86
87 fingerprint := sha256.Sum256(x509.MarshalPKCS1PublicKey(publicKey))
88 friend := friend{
89     Name:      name,
90     PKCS1PublicKey: base64.StdEncoding.EncodeToString(x509.MarshalPKCS1PublicKey(publicKey)),
91     Fingerprint: hex.EncodeToString(fingerprint[:]),
92 }
93
94 fCont[friend.Fingerprint] = friend
95 saveFriends()
96 }
97
98 func listFriends() {
99     if len(fCont) == 0 {
100         fmt.Println("You currently have no friends added, please use the \"add <public key path>\" " +
101             "command to add one")
102         os.Exit(0)
103     }
104
105     fmt.Println("The following people are able to decrypt any messages that you currently create:")
106     for _, f := range fCont {
107         fmt.Printf("- \"%s\", Fingerprint: %s\n", f.Name, f.Fingerprint)
108     }
109 }
110
111 func deleteFriend() {
112     if len(os.Args) < 3 {
113         fmt.Println("You must provide a fingerprint or fingerprint prefix to delete, please try again")
114         os.Exit(1)
115     }
116
117     fingerprint := os.Args[2]
118     if f, ok := fCont[fingerprint]; ok {
119         delete(fCont, fingerprint)
120         saveFriends()
121         fmt.Printf("Deleted friend named %s with fingerprint: %s\n", f.Name, f.Fingerprint)
122         return
123     }
124
125     for _, f := range fCont {
126         if strings.HasPrefix(f.Fingerprint, fingerprint) {
127             delete(fCont, f.Fingerprint)
128             saveFriends()
129             fmt.Printf("Deleted friend named %s with fingerprint: %s\n", f.Name, f.Fingerprint)
130             return
131         }
132     }
133
134     fmt.Printf("Failed to find friend with fingerprint matching %s, please try again\n", fingerprint)
135 }

```

5.4 message.go

```
1 package main
2
3 import (
4     "bufio"
5     "crypto/aes"
6     "crypto/cipher"
7     "crypto/rand"
8     "crypto/rsa"
9     "crypto/x509"
10    "encoding/asn1"
11    "encoding/base64"
12    "encoding/pem"
13    "fmt"
14    "os"
15    "strings"
16 )
17
18 /**
19  * Matador Multi-User Messaging Encryption System
20  *
21  * Copyright (c) 2020, Jack Harley, jackpharley.com
22  * All Rights Reserved
23  */
24
25 type EncryptedMessage struct {
26     Nonce          []byte
27     Ciphertext     []byte
28     EncryptedSessionKeys []EncryptedSessionKey
29 }
30
31 type EncryptedSessionKey struct {
32     PublicKeyFingerprint string
33     Ciphertext           []byte
34 }
35
36 func encryptMessage() {
37     if len(fCont) == 0 {
38         fmt.Println("You currently have no friends added, you must add at least one friend before " +
39             "encrypting a message, please use the \"add <public key path>\" command to add one")
40         os.Exit(0)
41     }
42
43     fmt.Printf("Please start typing your message to encrypt, you can use as many lines as you need. " +
44         "End your message with a line with nothing but a single . on it:\n\n")
45     scanner := bufio.NewScanner(os.Stdin)
46     message := ""
47     for scanner.Scan() {
48         if scanner.Text() == "." {
49             break
50         }
51         message += scanner.Text() + "\n"
52     }
53
54     fmt.Printf("\nEncrypting message... (this may take some time if your friends list is large)\n")
55
56     // generate a session key
57     sessionKey := make([]byte, 32) // 32 bytes = 256 bit (AES256)
58     rand.Read(sessionKey)
59
60     // prepare cipher
61     block, err := aes.NewCipher(sessionKey)
62     if err != nil {
63         fmt.Println("Failed to create AES blockcipher to encrypt message, please try again.")
64         os.Exit(1)
65     }
66     aesgcm, err := cipher.NewGCM(block)
67     if err != nil {
68         fmt.Println("Failed to create AES-GCM cipher to encrypt message, please try again.")
69         os.Exit(1)
70     }
71     nonce := make([]byte, 12)
```

```

71     rand.Read(nonce)
72
73     // encrypt the message
74     em := EncryptedMessage{
75         Nonce:      nonce,
76         Ciphertext:  aesgcm.Seal(nil, nonce, []byte(message), nil),
77         EncryptedSessionKeys: make([]EncryptedSessionKey, 0, len(fCont)),
78     }
79
80     // encrypt the session key with rsa for each friend's public key
81     randSource := rand.Reader
82     for _, friend := range fCont {
83         pkcs1PubKeyBytes, _ := base64.StdEncoding.DecodeString(friend.PKCS1PublicKey)
84         publicKey, _ := x509.ParsePKCS1PublicKey(pkcs1PubKeyBytes)
85         ciphertext, _ := rsa.EncryptPKCS1v15(randSource, publicKey, sessionKey)
86         esk := EncryptedSessionKey{
87             PublicKeyFingerprint: friend.Fingerprint,
88             Ciphertext:             ciphertext,
89         }
90         em.EncryptedSessionKeys = append(em.EncryptedSessionKeys, esk)
91     }
92
93     asn1OutBytes, _ := asn1.Marshal(em)
94     outPem := &pem.Block{
95         Type: "MATADOR ENCRYPTED MESSAGE",
96         Bytes: asn1OutBytes,
97     }
98
99     fmt.Printf("Encryption complete, send the following text to your friends:\n\n")
100    pem.Encode(os.Stdout, outPem)
101 }
102
103 func decryptMessage() {
104     fmt.Printf("Please paste the message to decrypt below and press Enter. You should include " +
105         "the BEGIN and END lines:\n\n")
106
107     // read input
108     scanner := bufio.NewScanner(os.Stdin)
109     data := ""
110     for scanner.Scan() {
111         data += scanner.Text() + "\n"
112         if strings.Contains(scanner.Text(), "-----END MATADOR ENCRYPTED MESSAGE-----") {
113             break
114         }
115     }
116
117     fmt.Printf("\nAttempting to decrypt message...\n")
118
119     // parse
120     inPem, _ := pem.Decode([]byte(data))
121     if inPem == nil || inPem.Type != "MATADOR ENCRYPTED MESSAGE" {
122         fmt.Printf("Unable to recognise the input message, please try again and make sure to " +
123             "include the full message including the BEGIN and END lines.")
124         os.Exit(1)
125     }
126
127     em := EncryptedMessage{}
128     _, err := asn1.Unmarshal(inPem.Bytes, &em)
129     if err != nil {
130         fmt.Printf("The message appears to be corrupted, please ask the sender to re-encrypt " +
131             "it and send it again.")
132         os.Exit(1)
133     }
134
135     // load our private key
136     fingerprint := getPublicFingerprint()
137     for _, esk := range em.EncryptedSessionKeys {
138         if fingerprint == esk.PublicKeyFingerprint {
139             privateKey := getPrivateKey()
140
141             // decrypt session key
142             sessionKey := make([]byte, 32)

```

```

143     randSource := rand.Reader
144     err := rsa.DecryptPKCS1v15SessionKey(randSource, privateKey, esk.Ciphertext, sessionKey)
145     if err != nil {
146         fmt.Printf("Failed to decrypt session key, please try again.")
147         os.Exit(1)
148     }
149
150     // prepare cipher for decrypting message
151     block, err := aes.NewCipher(sessionKey)
152     if err != nil {
153         fmt.Println("Failed to create AES blockcipher to decrypt message, please try again.")
154         os.Exit(1)
155     }
156     aesgcm, err := cipher.NewGCM(block)
157     if err != nil {
158         fmt.Println("Failed to create AES-GCM cipher to decrypt message, please try again.")
159         os.Exit(1)
160     }
161
162     // decrypt the message
163     message, err := aesgcm.Open(nil, em.Nonce, em.Ciphertext, nil)
164     if err != nil {
165         fmt.Println("Failed to decrypt message, please try again.")
166         os.Exit(1)
167     }
168
169     fmt.Printf("Message decrypted successfully, printing to console:\n\n%s", string(message))
170     return
171 }
172 }
173
174 fmt.Printf("Your private key has not been granted access to this message, make sure the sender " +
175     "has you added as a friend with your public key.")
176 os.Exit(1)
177 }

```