OTR PROTOCOL IMPLEMENTATION IN GO

ISA 763 – Security Protocol Analysis

Project Report

Date: 02/05/2018

Team Members:
Jaivendra Singh
Jonathan Torchia
Rutwij Kulkarni

Contents

Table of Figure

1. Introduction

Off-the-record protocol or more commonly known as OTR, is a cryptographic protocol that provides encryption for instant messaging conversation. OTR combines several cryptographic schemes: AES symmetric-key algorithm with 128 bits key length, the Diffie–Hellman key exchange with 1536 bits group size, and the SHA-1 hash function.

The motivation behind this protocol was to provide "deniable authentication" for the participants in a conversation while at the same time keeping their conversations "private". We can think of using PGP (instead of OTR) for communicating between two entities via Internet, however, PGP comes with the following drawbacks:

- The compromise of Bob's secret allows Trudy (attacker) to read not only future messages protected with that key, but past messages as well, and,
- Alice uses her Digital Signature to prove to Bob that she was the initiator of the message, which also proves it to Trudy or any other third entity.

To overcome these drawbacks, OTR was designed by Nikita Borisov, Ian Goldberg and Eric Brewer.

2. OTR Cryptographic Primitives

2.1. Perfect Forward Secrecy

The most obvious feature that we need from OTR is "confidentiality", which means, only Alice and Bob should be able to read messages that make up their online conversation. For this to work, we need to use encryption.

OTR achieves PFS by using short lived encryption keys that are generated as and when needed and discarded after use. These keys also contain the property that make it impossible for anyone to rederive them from any long-term material.

Thus, neither Trudy nor Alice or Bob would be able to reconstruct the key to read those past messages.

2.2. Digital Signatures

Digital Signatures are used to authenticate the author of the message. In short, a digital signature provides non-repudiation. However, OTR desires "repudiation". That is, no one should be able to prove that Alice sent any message. For this reason, OTR does not use Digital Signature to prove Alice's authorship of any message.

2.3. Message Authentication Codes (MAC)

OTR provides repudiation however, it still requires authentication to get security. That is, Bob needs to be assured that Alice is the one sending him the messages. For this reason, OTR uses MAC.

MAC provides repudiation, because Trudy cannot look at the MAC appended message and determine that Alice sent it because Trudy does not know the MAC key. Furthermore, Bob cannot prove to any other third entity that Alice sent him a message. All he can prove is that “someone” with the MAC key sent him the message and that “someone” could be Bob, himself.

Thus, by using a MAC, Bob can be assured that Alice sent the message, yet no one (not even Bob) can prove this fact to any other third entity.

2.4. Malleable Encryption

A more stronger property than repudiation is “forgeability”. This means that OTR desires to have a property that essentially says, “anyone could have modified the message or sent the message”.

To achieve this, after Alice knows all the messages she has sent to Bob that were MAC’d with the given MAC key have been received, Alice publishes the MAC key as part of her next message. Key things to note here are:

1) Bob has already checked all the messages authenticated by this MAC key and

2) Anyone can now create arbitrary messages with that MAC key and no one will be able to determine the author of the message.

Any other third-party entities can now modify past messages and recompute a correct MAC for them. However, this will not result in successful decryption because modifying an encrypted message will result in garbage once it is decrypted.

Thus, OTR uses Malleable encryption in the form of stream cipher, to encrypt the messages. In a malleable encryption scheme, someone can easily make changes to Ciphertext to make meaningful changes to Plaintext, even when someone does not know the key.

OTR derives the MAC key from Diffie-Hellman shared secret: MAC key is the hash of the encryption key.

3. Code Design

1) **Specific functions have been used to perform the following functions:**
    i) Generate Diffie-hellman parameters p,g
    ii) Generate x and y for Alice and Bob respectively. These are secret values selected to compute $g^x \text{mod}(p)$, $g^y \text{mod}(p)$, $g^{xy} \text{mod}(p)$
    iii) To generate public-private RSA key pair for the two users
    iv) Generating RSA signature on $g^x \text{mod}(p)$, $g^y \text{mod}(p)$ and performing verification
    v) Generation of SHA256 digest, HMAC, EK and MK keys.
    vi) Encryption and decryption of messages using AES
    vii) Re-keying operations

2) **The communicating users have been defined as structs to store various values and properties associated to them.**
3) **Used various crypto libraries available in GO to implement:**
    a) Diffie-hellman key exchange
    b) SHA256 digest generation
    c) HMAC generation
    d) RSA key generation
    e) RSA Signature generation and verification
    f) Encryption and decryption of messages using AES

4) **Channels used to communicate between processes**
    a) Channels allow synchronization without explicit locks
    b) Channels exists between the various processes specially the user functions

5) **Used WaitGroups to coordinate between the different actors.**
    a) Example: Bob can store Alice's public exponent only after she generates in Step 1 first.

4. Results

```
osboxes@osboxes:~/project4$ go run project4.go
##### RSA Initialization being done for users #####
*** Successfully generated RSA key pair for Alice  ***
*** Successfully generated RSA key pair for  Bob   ***
*** RSA Public key exchange successfully completed between Alice <<--->>  Bob  ***

##### DH (p,g) being generated by Alice #####
--> g =  2
--> p = 150519981562080791910336154121745892087816776085064806124029518832688300300
938057810621462159709403071884364719712459690135040656650882822365603740960150445256
##### DH (p,g) successfully shared with   Bob  #####
```

Figure 1: Generation of RSA key pairs for users and DH parameters (p,g)

```
##### DH (Public,Secret) exponents being generated by Alice #####
--> Alice 's picked secret:  10396408402002632484820933611265013599148418103262457767
924616299423410523579469092538048765160805320801622674055228061021849893131845803008 40
*** Generated SHA256 Digest ***
*** Generated RSA Signature ***

>>>>> Sharing Alice 's public exponent with   Bob  <<<<<
*** Generated SHA256 Digest ***
*** RSA Signature verification successful. Accepting Alice 's public exponent ***


##### DH (Public,Secret) exponents being generated by  Bob  #####
-->  Bob  's picked secret:  11952014496573863053363362312497309329411921388336413755 1
515831480258243077679113972893157005603830336278254416600350110531663619605209661871 94
*** Generated SHA256 Digest ***
*** Generated RSA Signature ***

>>>>> Sharing  Bob  's public exponent with  Alice <<<<<
*** Generated SHA256 Digest ***
*** RSA Signature verification successful. Accepting  Bob  's public exponent ***
##### Generating DH shared secret #####
||  Bob  || Generated Shared key:  150022248432210514716610828729664380371153357784880
955599225650732777284662324406412299980869612457458678806403606814262745264341820117 75
|| Alice || Generated Shared key:  150022248432210514716610828729664380371153357784880
955599225650732777284662324406412299980869612457458678806403606814262745264341820117 75
##### Generating EK and MK #####
```

Figure 2: Exchanging DH parameters with RSA signature verification

```
**********************************************************************
                        MESSAGE CONSOLE
**********************************************************************

----------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
----------------------------------------
|| Alice : " Lights on " ||

++++++++++++++++ Encrypting using AES ++++++++++++++++
|| Encrypted Message: 6YQ6MraSYLg26nsELJcgqFpvHalPNzM5iQPmv3ElcHY ||
++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

----------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
----------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by  Bob  : " Lights on " ||
++++++++++++++++++++++++++++++++++++++++++++++++++++
**********************************************************************
```

Figure 3: First message exchange between Alice-Bob with AES encryption and MAC verification

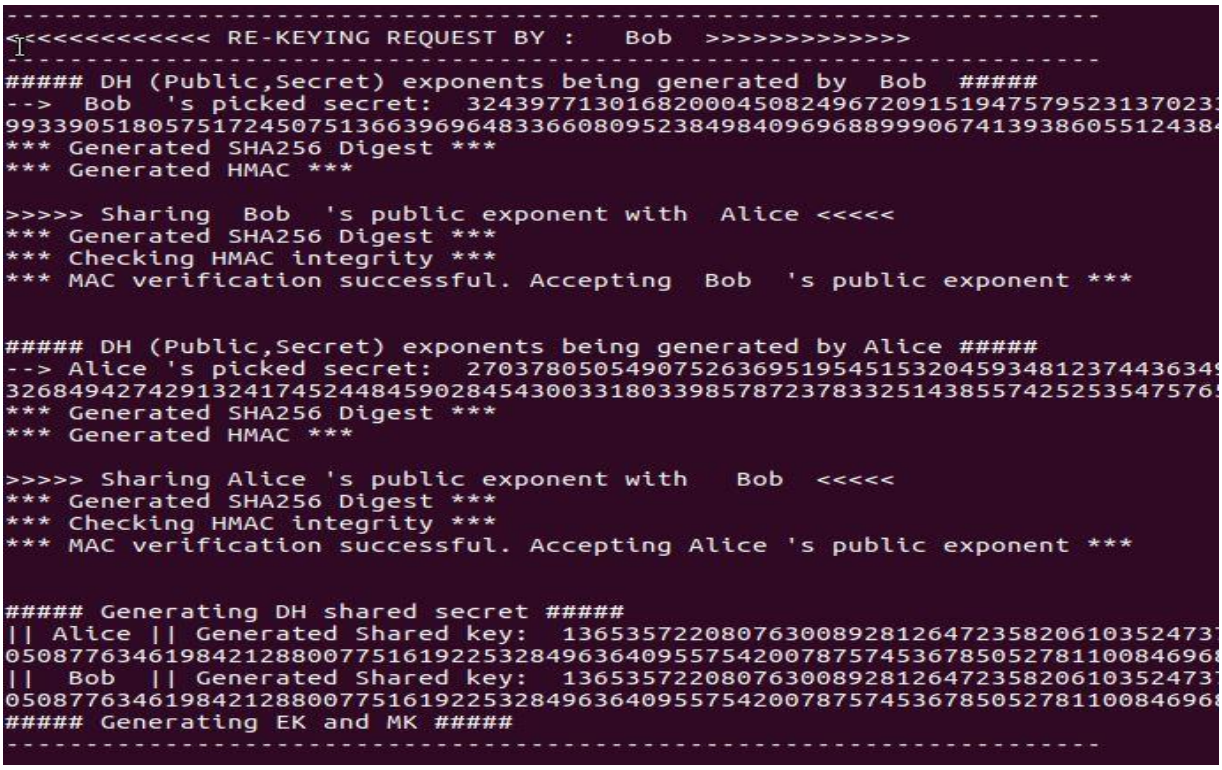

Figure 4: First Re-keying request by Bob to Alice

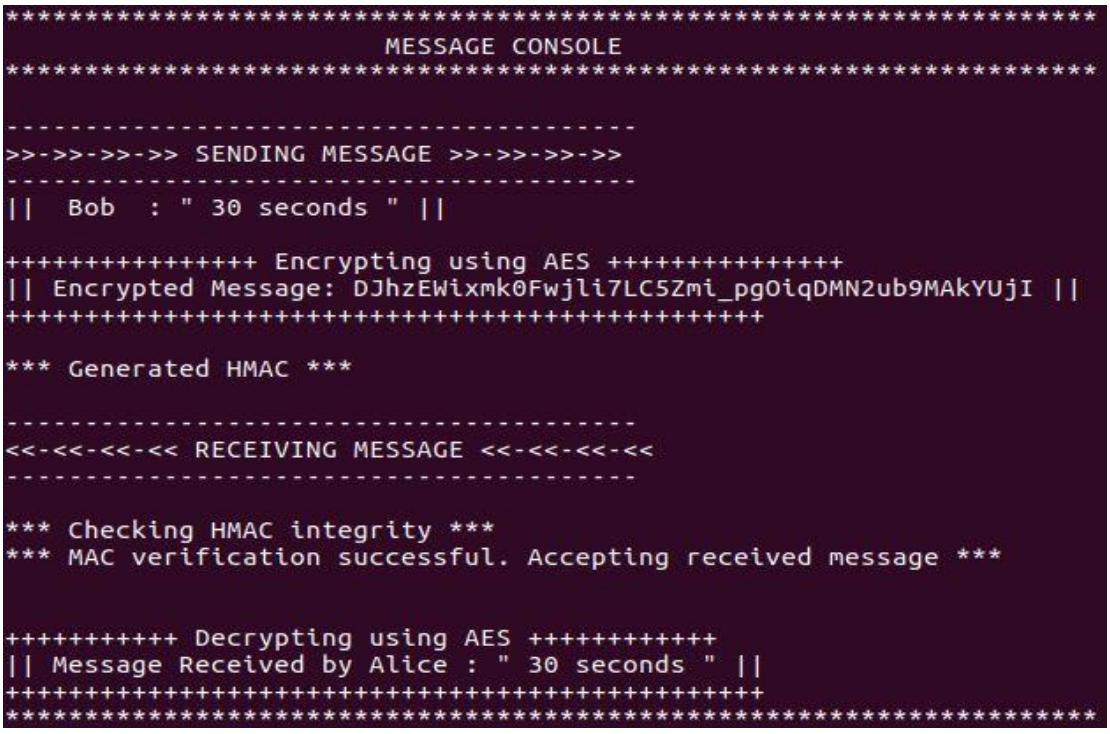

Figure 5: Second Message from Bob to Alice after Re-keying

```
**********************************************************************
                        MESSAGE CONSOLE
**********************************************************************

----------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
----------------------------------------
|| Alice : " Forward drift? " ||

++++++++++++++++ Encrypting using AES +++++++++++++++
|| Encrypted Message: BSXDNgJn6jU8Mb-tydA9WT7iF3MyFOqfnyKfsI2I7aU ||
++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

----------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
----------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by  Bob  : " Forward drift? " ||
++++++++++++++++++++++++++++++++++++++++++++++++++
**********************************************************************
```

Figure 6: Third Message from Alice to Bob after Re-keying

```
**********************************************************************
                        MESSAGE CONSOLE
**********************************************************************

----------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
----------------------------------------
||  Bob  : " Yes " ||

++++++++++++++++ Encrypting using AES +++++++++++++++
|| Encrypted Message: JwF7Nv3gfI6iAdS1NV--cxqTXz-yQERXKlejr0K4sS8 ||
++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

----------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
----------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by Alice : " Yes " ||
++++++++++++++++++++++++++++++++++++++++++++++++++
**********************************************************************
```

Figure 7: Fourth Message from Bob to Alice after Re-keying

```
**************************************************************************
                          MESSAGE CONSOLE
**************************************************************************

----------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
----------------------------------------
|| Alice : " 413 is in " ||

++++++++++++++++ Encrypting using AES +++++++++++++++
|| Encrypted Message: GcqGOQpmKM9Bd9HlLarNz1yjrmZgp33Irjj7uzacxB0 ||
++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

----------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
----------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by  Bob  : " 413 is in " ||
++++++++++++++++++++++++++++++++++++++++++++++++++
**************************************************************************
```

Figure 8: Fifth Message from Alice to Bob after Re-keying

```
**************************************************************
                      MESSAGE CONSOLE
**************************************************************

--------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
--------------------------------------
||  Bob  : " Houston, Tranquility base here " ||

++++++++++++++++ Encrypting using AES +++++++++++++++
|| Encrypted Message: McF_5wt9CIhk8BbuNyX9Yuc2jx9tLW9uGtw4dXKzHpmDCzaNIgPRCCfFB8vVQBzk ||
++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

--------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
--------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by Alice : " Houston, Tranquility base here " ||
++++++++++++++++++++++++++++++++++++++++++++++++++
**************************************************************
```

Figure 9: Sixth Message from Bob to Alice after Re-keying

```
**********************************************************************
                        MESSAGE CONSOLE
**********************************************************************

---------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
---------------------------------------
|| Alice : " The Eagle has landed " ||

++++++++++++++++ Encrypting using AES ++++++++++++++++
|| Encrypted Message: y-jw9zxfqxyy1HtgGh1htC7LYQHL64ibWdRPHb_QAP44IlRZ5bhFdK4DjEPXqngH ||
++++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

---------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
---------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by  Bob  : " The Eagle has landed " ||
++++++++++++++++++++++++++++++++++++++++++++++++++++++
**********************************************************************
```

Figure 10: Seventh Message from Alice to Bob after Re-keying

```
**********************************************************************
                        MESSAGE CONSOLE
**********************************************************************

---------------------------------------
>>->>->>->> SENDING MESSAGE >>->>->>->>
---------------------------------------
||  Bob  : " A small step for a student, a giant leap for the group " ||

++++++++++++++++ Encrypting using AES ++++++++++++++++
|| Encrypted Message: N_90tqZnSvlz76D-afrtatBvsc5dVtcPiNsQp_r9zDeqrZHPtMr6uQMh6o0D0Ca3PludCm-zWIgRrP_vpDpL4u1y7_oATfsPN5bJKGYQp5M ||
++++++++++++++++++++++++++++++++++++++++++++++++++++++

*** Generated HMAC ***

---------------------------------------
<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<
---------------------------------------

*** Checking HMAC integrity ***
*** MAC verification successful. Accepting received message ***


++++++++++++ Decrypting using AES ++++++++++++
|| Message Received by Alice : " A small step for a student, a giant leap for the group " ||
++++++++++++++++++++++++++++++++++++++++++++++++++++++
**********************************************************************
```

Figure 11: Final Message from Bob to Alice after Re-keying

5. Conclusion

The results prove that OTR protocol provides encryption, authentication, and perfect forward secrecy. Even though it provides authentication, it also provides plausible deniability. During the protocol, Alice and Bob can be confident that the messages are authentic, but after the protocol finishes, it is not possible to prove that someone sent any given message (repudiation).

References

[1] Nikita Borisov, Ian Goldberg, Eric Brewer "Off-the-Record Communication, or, Why Not to Use PGP" https://otr.cypherpunks.ca/otr-wpes-present.pdf

[2] Cypherpunks "Off-the-Record Messaging Protocol version 3" https://otr.cypherpunks.ca/Protocol-v3-4.0.0.html

[3] Online GO Documentation: http://golang.org

[3] TutorialsPoint GO Language: https://www.tutorialspoint.com/go/index.htm

Appendix

I. Code:

```
package main

import (
    "bytes"
    "fmt"
    "strings"
    "time"
    "sync"
    "io"
    "crypto"
    "crypto/aes"
    "crypto/cipher"
    "crypto/rsa"
    "crypto/rand"
    "crypto/hmac"
    "crypto/sha256"
    "math/big"
    mathr "math/rand"
    "os/exec"
    "log"
    "encoding/base64"
)

/********************************** Diffie-Hellman
*******************************************************/

func (u *User) genDHparam(ch_bigint chan *big.Int, ch_int chan int) {
    out, err := exec.Command("bash", "-c", "openssl gendh -2 1024 | openssl dh -noout -text |
sed 's/^[ ^t]*//;s/://g;' | sed -n 3,11p | tr -d '\\n'").Output()
    if err != nil {
        log.Fatal(err)
    }
    output := string(out[:])
    p := new(big.Int)
    p.SetString(output, 16)
    u.dh_p=p
    u.dh_g=2
    fmt.Println("--> g = ",u.dh_g,"\n--> p =",u.dh_p)
    ch_bigint <- u.dh_p
```

```
    ch_int <- u.dh_g
    wg.Done()
}

func (u *User) genDHkeys (ch_bigint chan *big.Int) {
    secret := new(big.Int)
    public := new(big.Int)
    var err error

    mathr.Seed(time.Now().UTC().UnixNano())

    secret, err = rand.Int(rand.Reader, u.dh_p)
    if err != nil {
            fmt.Println(err)
            }
    u.dh_secret = secret
    public.Exp(big.NewInt(int64(u.dh_g)), u.dh_secret, u.dh_p)
    u.dh_public = public
    fmt.Println("-->",u.identity,"'s picked secret: ",secret)//,"\n-->",u.identity,"'s Public
Exponent: ", u.dh_public)
    ch_bigint <- u.dh_public
    wg.Done()
}

func (u *User) genSharedkey() {
    shared := new(big.Int)

    shared.Exp(u.dh_public_partner, u.dh_secret, u.dh_p)
    u.dh_shared = shared
    fmt.Println("||",u.identity,"||","Generated Shared key: ", u.dh_shared)
    wg.Done()
}

func initializeDHparams (u1 *User, u2 *User){
    channel_bigint := make(chan *big.Int,1)
    channel_int := make(chan int,1)
    wg.Add(1)
    fmt.Println("\n##### DH (p,g) being generated by",u1.identity,"#####")
    go u1.genDHparam(channel_bigint, channel_int)
    wg.Wait()
    u2.dh_p = <-channel_bigint
    u2.dh_g = <-channel_int
    fmt.Println("##### DH (p,g) successfully shared with ",u2.identity,"#####")
}

func initializeDH (u1 *User, u2 *User) {
    channel_digest := make(chan []byte,1)
    channel_sig := make(chan []byte,1)
    channel_bool := make(chan bool,1)
    channel_bigint := make(chan *big.Int,1)

    wg.Add(3)
    fmt.Println("\n\n##### DH (Public,Secret) exponents being generated
by",u1.identity,"#####")
    go u1.genDHkeys(channel_bigint)
    go u1.genSHA256(<-channel_bigint, channel_digest, channel_bigint)
    go u1.genRSAsig(<-channel_digest, channel_sig)
    wg.Wait()
    fmt.Println("\n\n>>>>> Sharing",u1.identity,"'s public exponent with
",u2.identity,"<<<<<")
    wg.Add(2)
    go u2.genSHA256(<-channel_bigint, channel_digest, channel_bigint)
```

```
    go u2.verifyRSAsig(<-channel_digest,<-channel_sig, channel_bool)
    wg.Wait()
    if (<-channel_bool){
            fmt.Println("\n*** RSA Signature verification successful.
Accepting",u1.identity,"'s public exponent ***")
            u2.dh_public_partner = <-channel_bigint
    } else {
            fmt.Println("\n*** RSA Signature verification successful.
Rejecting",u1.identity,"'s public exponent ***")
    }
}

/*************************************************************************************
***/

/********************************** RSA
*****************************************************/

func (u *User) genRSAkeys (bits int, channel chan *rsa.PublicKey) {
    private_key, err := rsa.GenerateKey(rand.Reader, bits)
    if err != nil {
            fmt.Println(err)
            }
    fmt.Println("*** Successfully generated RSA key pair for", u.identity," ***")
    public_key := private_key.PublicKey
    u.rsaKey = private_key
    u.rsaPubKey = &public_key
    channel <- u.rsaPubKey
    wg.Done()
}

func (u *User) genRSAsig (digest []byte, channel chan []byte) {
    signature, err := rsa.SignPKCS1v15(rand.Reader, u.rsaKey, crypto.SHA256, digest)
    if err != nil {
            fmt.Println(err)
            }
    channel <- signature
    fmt.Printf("\n*** Generated RSA Signature ***")
    wg.Done()
}

func (u *User) verifyRSAsig (digest []byte, signature []byte, channel chan bool) {
    err := rsa.VerifyPKCS1v15(u.rsaPubKey_partner, crypto.SHA256, digest, signature)
    if err != nil {
            fmt.Println(err)
            channel <- false
            }
    channel <- true
    wg.Done()
}

func initializeRSA(u1 *User, u2 *User){
    channel_rsaPubKey := make(chan *rsa.PublicKey, 2)
    fmt.Println("##### RSA Initialization being done for users #####")
    wg.Add(1)
    go u1.genRSAkeys(2048,channel_rsaPubKey)
    wg.Wait()
    wg.Add(1)
    go u2.genRSAkeys(2048,channel_rsaPubKey)
    wg.Wait()
    u2.rsaPubKey_partner = <-channel_rsaPubKey
        u1.rsaPubKey_partner = <-channel_rsaPubKey
```

```
    fmt.Println("*** RSA Public key exchange successfully completed between",u1.identity,"<<-
-->>",u2.identity,"***")
}


/*************************************************************************************************
***/

/********************************** SHA256 Digest
*****************************************/

func (u *User) genSHA256 (b *big.Int, channel chan []byte, ch_int chan *big.Int) {

         digest := sha256.Sum256([]byte(b.String()))
     fmt.Printf("*** Generated SHA256 Digest ***")
     channel <- digest[:]
     ch_int <- b
     wg.Done()
}

/*************************************************************************************************/

/********************************** HMAC ********************************************/

func (u *User) genMACkey (message []byte, channel chan []byte) {
    fmt.Printf("\n*** Generated HMAC ***")
    mac := hmac.New(sha256.New, u.mk)
    mac.Write(message)
    messageMAC := mac.Sum(nil)
    channel <- messageMAC[:]
    wg.Done()
}

func (u *User) checkMACkey(message []byte, messageMAC []byte, channel chan bool) {
    fmt.Printf("\n*** Checking HMAC integrity ***")
    mac := hmac.New(sha256.New, u.mk)
    mac.Write(message)
    expectedMAC := mac.Sum(nil)
    channel <- hmac.Equal(messageMAC, expectedMAC)
    wg.Done()
}

func (u *User) genMACmsg (message string, channel chan []byte, channel_string chan string) {
    fmt.Printf("\n*** Generated HMAC ***")
    mac := hmac.New(sha256.New, u.mk)
    mac.Write([]byte(message))
    messageMAC := mac.Sum(nil)
    channel <- messageMAC[:]
    channel_string <- message
    wg.Done()
}

func (u *User) checkMACmsg(message string, messageMAC []byte, channel chan bool,
channel_string chan string) {
    fmt.Printf("\n*** Checking HMAC integrity ***")
    mac := hmac.New(sha256.New, u.mk)
    mac.Write([]byte(message))
    expectedMAC := mac.Sum(nil)
    channel <- hmac.Equal(messageMAC, expectedMAC)
    channel_string <- message
    wg.Done()
}
```

```
/*************************************************************************************************/

/********************************** AES ***********************************************/

func addBase64Padding(value string) string {
    m := len(value) % 4
    if m != 0 {
        value += strings.Repeat("=", 4-m)
    }

    return value
}

func removeBase64Padding(value string) string {
    return strings.Replace(value, "=", "", -1)
}

func Pad(src []byte) []byte {
    padding := aes.BlockSize - len(src)%aes.BlockSize
    padtext := bytes.Repeat([]byte{byte(padding)}, padding)
    return append(src, padtext...)
}

func Unpad(src []byte) ([]byte) {
    length := len(src)
    unpadding := int(src[length-1])

    if unpadding > length {
        fmt.Println("Unpadding Error !!")
    }

    return src[:(length - unpadding)]
}

func (u *User) encrypt(text string, channel chan string) {
    fmt.Println("+++++++++++++++++ Encrypting using AES ++++++++++++++++")
    block, err := aes.NewCipher(u.ek)
    if err != nil {
        fmt.Println(err)
    }

    msg := Pad([]byte(text))
    ciphertext := make([]byte, aes.BlockSize+len(msg))
    iv := ciphertext[:aes.BlockSize]
    if _, err := io.ReadFull(rand.Reader, iv); err != nil {
        fmt.Println(err)
    }

    cfb := cipher.NewCFBEncrypter(block, iv)
    cfb.XORKeyStream(ciphertext[aes.BlockSize:], []byte(msg))
    finalMsg := removeBase64Padding(base64.URLEncoding.EncodeToString(ciphertext))
    channel <- finalMsg
    fmt.Println("|| Encrypted Message:", finalMsg,"||")
    fmt.Println("++++++++++++++++++++++++++++++++++++++++++++++++")
    wg.Done()
}

func (u *User) decrypt(text string) {
    fmt.Println("++++++++++++ Decrypting using AES ++++++++++++")
    block, err := aes.NewCipher(u.ek)
    if err != nil {
        fmt.Println(err)
```

```
    }

    decodedMsg, err := base64.URLEncoding.DecodeString(addBase64Padding(text))
    if err != nil {
        fmt.Println(err)
    }

    if (len(decodedMsg) % aes.BlockSize) != 0 {
        fmt.Println("Blocksize must be multiple of decoded message length")
    }

    iv := decodedMsg[:aes.BlockSize]
    msg := decodedMsg[aes.BlockSize:]

    cfb := cipher.NewCFBDecrypter(block, iv)
    cfb.XORKeyStream(msg, msg)

    unpadMsg := Unpad(msg)

    fmt.Println("|| Message Received by",u.identity,": \"", string(unpadMsg[:]),"\" ||")
    fmt.Println("++++++++++++++++++++++++++++++++++++++++++++++++++")
    wg.Done()
}
/*****************************************************************************************
***/

/********************************** OTR ***********************************************/

func (u *User) genEKMK () {
    ek := sha256.Sum256([]byte(u.dh_shared.String()))
    mk := sha256.Sum256(ek[:])
    u.ek = ek[:]
    u.mk = mk[:]
    wg.Done()
}

func genKeys(u1 *User, u2 *User){
    fmt.Println("##### Generating DH shared secret #####")
    wg.Add(2)
    go u1.genSharedkey()
    go u2.genSharedkey()
    wg.Wait()

    fmt.Println("##### Generating EK and MK #####")
    wg.Add(2)
    go u1.genEKMK()
    go u2.genEKMK()
    wg.Wait()
}

func reKeying (u1 *User, u2 *User) {
    channel_bigint := make(chan *big.Int,1)
    channel_digest := make(chan []byte,1)
    channel_mac := make(chan []byte,1)
    channel_bool := make(chan bool,1)
    wg.Add(3)
    fmt.Println("##### DH (Public,Secret) exponents being generated by",u1.identity,"#####")
    go u1.genDHkeys(channel_bigint)
    go u1.genSHA256(<-channel_bigint, channel_digest, channel_bigint)
    go u1.genMACkey(<-channel_digest, channel_mac)
    wg.Wait()
```

```
    fmt.Println("\n\n>>>>> Sharing",u1.identity,"'s public exponent with
",u2.identity,"<<<<<")
    wg.Add(2)
    go u2.genSHA256(<-channel_bigint, channel_digest, channel_bigint)
    go u1.checkMACkey(<-channel_digest, <-channel_mac, channel_bool)
    wg.Wait()
    if (<-channel_bool){
            fmt.Println("\n*** MAC verification successful. Accepting",u1.identity,"'s public
exponent ***\n\n")
            u2.dh_public_partner = <-channel_bigint
    } else {
            fmt.Println("\n*** MAC verification failed. Rejecting",u1.identity,"'s public
exponent ***\n\n")
    }
}

func reKey (u1 *User, u2 *User) {
    fmt.Println("\n-------------------------------------------------------------------------")
    fmt.Println("<<<<<<<<<<<<< RE-KEYING REQUEST BY : ",u1.identity,">>>>>>>>>>>>>")
    fmt.Println("-------------------------------------------------------------------------")
    reKeying(u1, u2)
    reKeying(u2, u1)
    genKeys(u1, u2)
    fmt.Println("-------------------------------------------------------------------------")
}

func commMsg(u1 *User, u2 *User, i int) {
    fmt.Println("\n\n*************************************************************************")
    fmt.Println("                          MESSAGE CONSOLE                                ")
    fmt.Println("*************************************************************************")
    fmt.Println("\n----------------------------------------")
    fmt.Println(">>->>->>->> SENDING MESSAGE >>->>->>->>")
    fmt.Println("----------------------------------------")
    fmt.Println("||",u1.identity,": \"",u1.messages[i],"\" ||\n")
    channel_string := make(chan string, 1)
    channel_mac := make(chan []byte, 1)
    channel_bool := make(chan bool,1)
    wg.Add(2)
    go u1.encrypt(u1.messages[i],channel_string)
    go u1.genMACmsg(<-channel_string, channel_mac, channel_string)
    wg.Wait()
    fmt.Println("\n\n----------------------------------------")
    fmt.Println("<<-<<-<<-<< RECEIVING MESSAGE <<-<<-<<-<<")
    fmt.Println("----------------------------------------")
    wg.Add(1)
    go u2.checkMACmsg(<-channel_string,<-channel_mac,channel_bool,channel_string)
    wg.Wait()
    if (<-channel_bool){
            fmt.Println("\n*** MAC verification successful. Accepting received message
***\n\n")
            wg.Add(1)
            go u2.decrypt(<-channel_string)
            wg.Wait()
    } else {
            fmt.Println("\n*** MAC verification failed. Rejecting received message ***\n\n")
    }
    fmt.Println("*************************************************************************")
}

/*****************************************************************************************
***/
```

```
type User struct {
    identity string   // User's Name
    messages []string // User's Messages to send
    rsaPubKey *rsa.PublicKey // RSA Public key of user
    rsaPubKey_partner *rsa.PublicKey // RSA Public key of partner
    rsaKey *rsa.PrivateKey // RSA private key of user
    dh_p *big.Int // Value of DH prime
    dh_g int // Value of DH group
    dh_secret *big.Int // Value of user's selected secret DH exponent e.g. x
    dh_public *big.Int // Value of user's public key i.e. (g)^x mod p
    dh_public_partner *big.Int // Value of partner's public key i.e. (g)^y mod p
    dh_shared *big.Int // Value of shared key (g)^xy mod p
    ek []byte // Encryption Key
    mk []byte // Key for HMAC
}

var wg sync.WaitGroup

func main() {
    mathr.Seed(time.Now().UTC().UnixNano())

    u1 := User{identity: "Alice", messages: []string {"Lights on","Forward drift?","413 is
in","The Eagle has landed"}}
    u2 := User{identity: " Bob ", messages: []string {"30 seconds", "Yes", "Houston,
Tranquility base here", "A small step for a student, a giant leap for the group"}}

    initializeRSA(&u1,&u2)

    initializeDHparams(&u1,&u2)

    initializeDH(&u1,&u2)
    initializeDH(&u2,&u1)
    genKeys(&u1, &u2)
    j:= 0
    for i:=0;i<8; i++ {
            if (i%2==0) {
                    commMsg(&u1,&u2,j)
                    reKey(&u2,&u1)
            } else {
                    commMsg(&u2,&u1,j)
                    reKey(&u2,&u1)
                    j++
            }
    }
}
```