The Integration of Symmetric and Asymmetric Cryptography to Form a Secure Encryption Scheme

Quillen Flanigan

Conjure-2

6/10/2023

-In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper

Abstract:

Sharing information through digital communication forces us to develop cryptographic schemes to be able to keep our information and data secure. Our encryption protocol integrates the benefits and design schemes of symmetric and asymmetric encryption, while incorporating integrity verification and authentication in the forms of message authentication codes and digital signatures, respectively. We combined the Diffie-Hellman Key Exchange for asymmetric encryption and the Fernet algorithm for symmetric encryption. We then used the HMAC library to to verify the encrypted message's integrity and used an RSA key pair to create digital signatures to authenticate the sender of the information. Integrating these schemes into one large encryption protocol allows us to send encrypted messages over a network connection in a secure manner and verify both the integrity and authenticity of the message.

1. Introduction:

Cryptography has always been an integral part of our society, dating back all the way to the inscriptions carved into the walls of tombs in ancient Egypt. As we march through history, we find that cryptography is forever evolving, constantly trying to catch up with the growing security demands around the world. Modern society and technology is only further increasing the need to grow in the field of information security, as with every technological advancement comes a new vulnerability to exploit. Billions of messages sent each day by hundreds of millions of people. Secure communication has never been more essential to the world as we know it.

The field of cyber security and cyber warfare is especially focused on this goal of developing secure communication. Cyber warfare is one of the most critical and intense fields on the planet, as millions of lives are dependent on our ability to facilitate secure communication and keep our sensitive data protected. Whether we need to communicate about the location of an adversary, intelligence among allies, or share sensitive personal data, we must be able to communicate in a way that satisfies the main criteria of security.

We must keep our information confidential, maintain it’s integrity, and make sure the information we need is always available at a moments notice. These three branches help form a general idea of what standards of security must be met by modern encryption. My team and I have acted on this need to produce an encryption protocol that incorporates various cryptographic libraries, algorithms, and all the present suggested standards to build a secure scheme of encryption. This scheme will then allow us to facilitate secure communication of any message needed over a network connection for any specified pair of users.

1.1 Scope and Limitations:

The design we have created is a combination of several algorithms that all have their own unique limitations. Given these limitations, the message being encrypted is limited to text files/ASCII text for now and has a maximum size of 2^{39} bytes given the Fernet algorithm limitations.

2. Background:

The concepts of security and cryptography are very popular, but often understood in a watered-down manner to the average individual. In this context, security of communications refers to the idea that if someone is looking at encrypted data over the network, they have no ability to gain any information from observing the encrypted traffic. If they can gain any knowledge about the encrypted data, whether that's a whole sentence or just realizing the same few characters are being used in the same place, then that message is not considered secure. We have designed our encryption protocol to limit the ability of an adversary to gain any knowledge of the encrypted data.

2.1 CIA Triad:

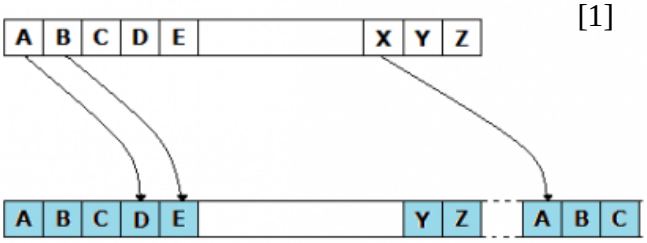
Secondly, there are three major concepts discussed relating to our standards of securing data: confidentiality, integrity, and availability. Together these form what is known as the CIA triad, serving as the foundation of our modern standards of security. Confidentiality refers to the idea that we want our data to remain disguised from anyone looking at it from outside. Integrity refers to the idea that when we send encrypted data, we want to ensure our data is correct and not being altered in any way while across the network. And lastly, we want our information to be

available. This means at all times, we want to be able to access the correct information, whenever we may need it, at a moments notice. These three standards are required to be able to establish secure communication of sensitive data.

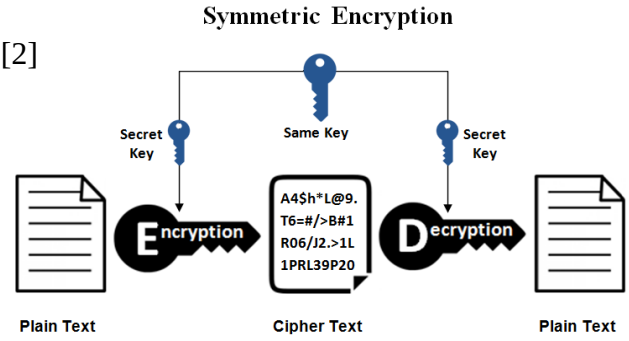
2.2 Symmetric Encryption:

We designed an encryption scheme using various methods, with the most basic being symmetric encryption. Very simply, symmetric encryption relies on a secret key to encrypt the data you want to send, and anyone who wants to encrypt or decrypt the data needs this specific key to "unlock" to data. This form of encryption served as one of the first real forms of cryptography, used in methods such as the Caesar Cipher, Vigenere Cipher, and more famously the Enigma machine, used by the Germans during World War II. These basic ciphers rely on the production of a "key" to be able to transform the plain text (original message), into cipher text (encrypted message). This image portrays the basics of original ciphers, using a specific variant to shift each character by a constant amount, forming encrypted text.

[1]



[2]



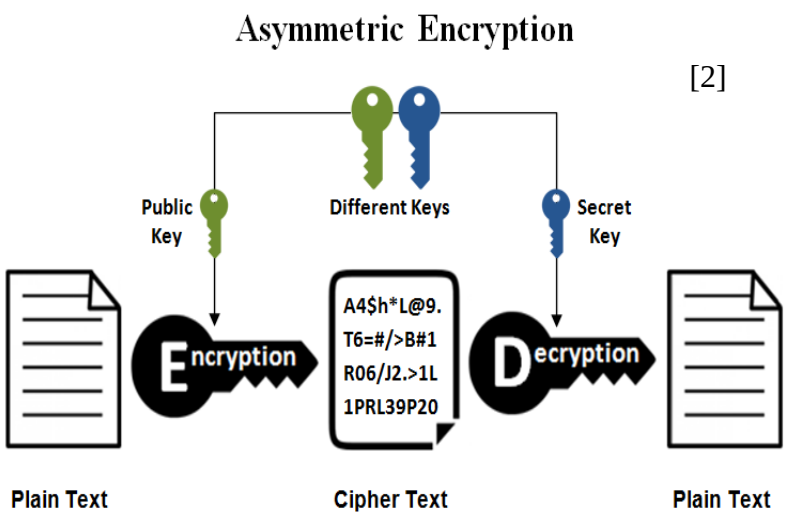
The word "hello" would be encrypted as "khoor" with this cipher. As you can see with this cipher, we can now tell that the two middle letters "oo" are the same letter in the original plain text. We have let the adversary discover information about our plain text, violating the most basic rule of secure cryptography. The modern use of symmetric encryption must be able to produce unique cipher text even if the plain text remains the same. This image represents the process that symmetric key encryption uses, as it produces a secret key that both encrypts and decrypts a

message, hopefully producing unique output each time we create cipher text. This method is simple, efficient and effective, but relies solely on the secrecy of the symmetric key and forces us to now have to worry about securely sending both the key and encrypted text across a network. Symmetric encryption has its drawbacks, but is an effective, efficient way to establish confidentiality of our information[1].

2.3 Asymmetric Encryption:

As we have established, symmetric encryption can be quick and easily implemented, but it requires us to now have to worry about securely sending a secret key along with the encrypted text. Losing this secret key means all of the encrypted messages that have been sent using that key are now immediately compromised. To counter this vulnerability, asymmetric encryption poses we use two keys used in conjunction to form a secure line of communication. The first key is a public key that is responsible for the encryption of the data we want to send. We don't have to worry about this key being stolen or discovered, as asymmetric encryption and decryption relies on having both keys to be able to encrypt or decrypt the data. The second key is the known as the private key, which is owned by both the sender and receiver and used to decrypt messages. The image below illustrates this process.

The sender of the plain text uses the public key of the recipient to encrypt the data, and the receiver then uses their private key to decrypt, and vice versa. Although this method lacks the speed and simplicity of symmetric encryption, it performs very well in terms of security as it depends on a less vulnerable method using the two keys[2].

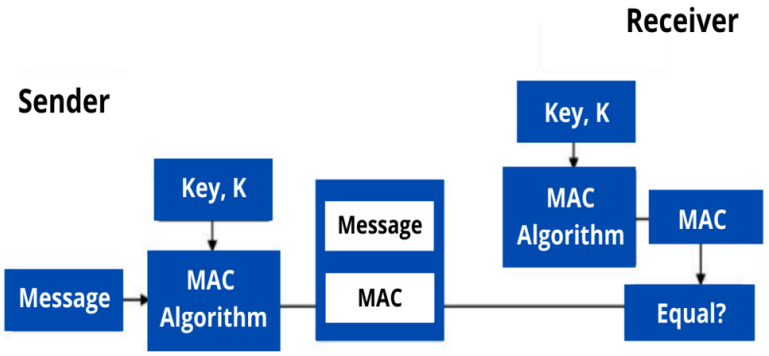


[2]

2.4 Message Authentication Codes and Digital Signatures

Now that we have covered the confidentiality of our information with both asymmetric and symmetric encryption, we must focus on ensuring its' integrity and availability. As for the integrity of our message, Message Authentication Codes (MAC's) are used to verify that the encrypted data was not altered by any adversary while in transit across a network. MAC's are "personal secrets" used in symmetric cryptography that get hashed using the symmetric key and plain text and then appended to a message after the message is encrypted. This removes the ability to alter the message while over a network without being discovered, as the receiver of the message can use the private symmetric key and the same hashing algorithm to see if the MAC is the same. If the MAC is different than the original, we know the message has been altered since it was sent.

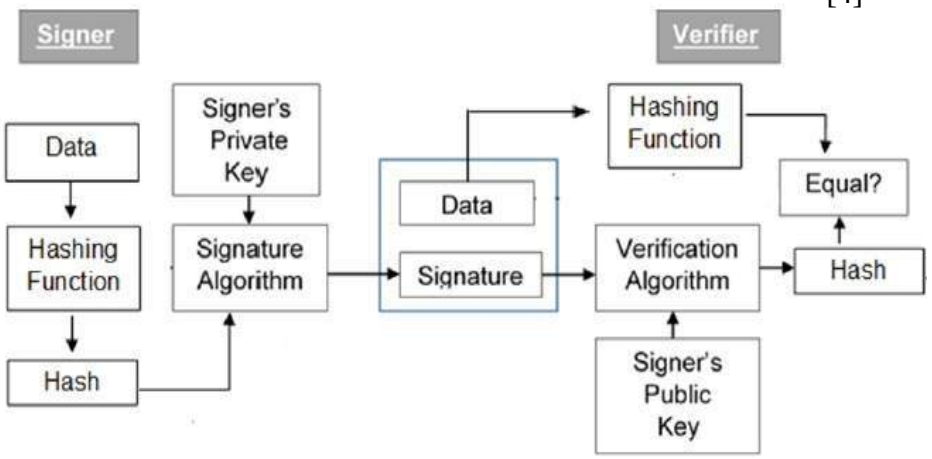
As we can see, the sender uses the symmetric key and the plain text to generate a MAC, and the receiver can check with their key and the same MAC algorithm to see if they get the same MAC, thus verifying the integrity of the message.



[3]

Deepening our verification of integrity and providing a layer of authentication to our encryption, digital signatures allow the sender of a message to “sign” the encrypted message so the receiver can make sure it was sent by an authenticated individual[3]. While MAC’s are often paired with symmetric encryption, digital signatures often go hand in hand with asymmetric encryption. As this diagram illustrates, the sender uses a hashing function and their private key to create a signature that is sent along with the encrypted data. The verifier then uses this same hashing function and the sender’s public key to see if the hashes are equal, which then proves that the correct person is who encrypted and sent the message. This functionality provides us authentication of the sender and further increases the security of the encryption scheme[4].

[4]



3. Methods:

So far we have discussed four main tools used in modern encryption: symmetric encryption, asymmetric encryption, message authentication codes and digital signatures. We have also seen that these four tools are often used in pairs with very limited overlap. Symmetric encryption is paired with message authentication and asymmetric encryption is used with digital signatures. The encryption scheme we have developed aims to incorporate all these tools into one protocol working in tandem, providing the users with confidentiality, integrity, availability and authentication of their sensitive data.

Our first step was to build the first method of encryption using asymmetric encryption. There are several useful asymmetric algorithms including RSA and elliptic curve schemes, but we decided to implement the Diffie-Hellman (DH) Key Exchange algorithm. Though these are the three most popular techniques with RSA being the most commonly used, Diffie-Hellman is currently understood to be the most cryptographically secure method for asymmetric encryption. The primary benefit of DH is the speed and ease in which it can produce new, unique keys and remain effective. This is important when discussing the concept of forward secrecy: the discovery of an encryption key leading to the breakdown of our encryption scheme. When using keys to encrypt and decrypt data, if an adversary were to discover this key, it would instantly break all current encryption based on those keys. DH however has the unique property of limited damage in terms of a key being exposed. When we lose a key in DH, there is usually only one session of encryption being exposed as the keys shift with every exchange. This means the damage done by an adversary discovering our hidden keys is far less severe than it would be if using the RSA algorithm.

The second main advantage we focused on with when choosing Diffie-Hellman was it's advantage in mathematical complexity. Essentially every modern encryption scheme is developed with highly complex mathematical computation and these computations are often stored in matrices. The DH algorithm is special as instead of storing single bits in these enormous matrices like RSA does, it utilizes the *modulo* function of very large prime numbers and stores those in the matrices while keeping the space complexity the same as an RSA algorithm of equal size. This means we can produce much more mathematically complex encryption using DH over RSA in the same amount of bits/bytes. For example, RSA 1024 and DH 1024 are technically the same space complexity but the DH algorithm is considered more mathematically strenuous and cryptographically stronger[5].

Through the Diffie-Hellman exchange, we produce a pair of keys for both users: a private key and a public key.

```
dhAlice = pyDH.DiffieHellman()
alicePK = dhAlice.gen_public_key()
```

We assign the variable dhAlice with an instance of the python Diffie-Hellman library, then use this instance to generate a public key for Alice. The code is similar for the other user's program, just replacing the 'Alice' phrase with 'Bob'. Now that the public keys are generated, we must establish an exchange between both users.

Our next step was to set up the initial network connection that would be used to send encrypted data over. This design choice helps to replicate more real-world usage and applications, as we generally are worried most about encrypting our data during transmission over a network. To accomplish this, we imported several python libraries in our program. This program consists of two main python files/"users", named Bob and Alice. The network connection and initial key exchange below is completed in our asymmetric encryption key exchange function, deemed

```
def asymKE():
```

This function is present in both user programs and establishes the first encrypted transmission of data.

```
import socket, pyDH, base64, sys, os, hashlib, binascii, time
```

This first import line allows us to import several python libraries, most importantly the socket library, allowing us to open a network connection from Bob and Alice, with Alice serving as the host/server and Bob serving as the client. We also see the pyDH library which allowed us to use the statements above to generate key pairs for both Bob and Alice.

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
s.bind((HOST, PORT))
s.listen()
```

We created this instance of the socket library in the Alice python file to bind the host IP address to the port number of the computer, thus creating a connected network for Bob and Alice to send messages to each other. In the Bob file, we used the command

```
s.connect((HOST, PORT))
```

to connect to this instance of the socket connection. For this program, we initially set the HOST (IP address) to be 127.0.0.1, representing the local computer. This means that the network will be looping back from the local system back on itself. For real-world applications, the HOST would just need to be changed to which ever IP addresses the users would like to communicate to and from, whether that be a local computer, server, or another remote system. The PORT variable can just be set to any available port; we used port 65432 in both files as the network will just be looping back onto the local host[6].

The respective programs for Bob and Alice send each other their public keys, which they will each use to encrypt messages. The two statements below accomplish this:

```
conn.sendall(str(alicePK).encode('utf8'))
s.sendall(str(bobPK).encode('utf8'))
```

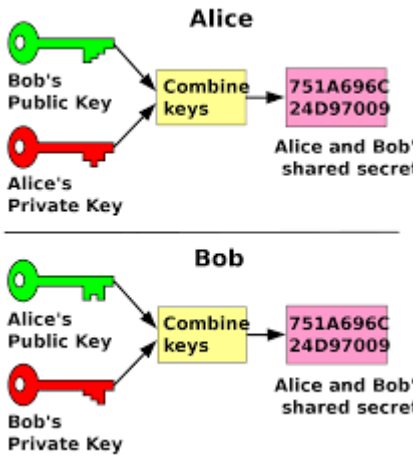
We can see Alice sending her public key, deemed ‘alicePK’ to Bob, and Bob sending his public key, ‘bobPK” to Alice. This establishes an initial connection and notion of security among both parties, and also allows us to now encrypt messages as we use the receiver’s public key for encryption under the DH algorithm. We also see each of Bob and Alice receive the other’s public keys and save them:

```
bobPK = int(conn.recv(1024))
alicePK = int(s.recv(1024))
```

We have enabled Alice to receive Bob’s public key before sending hers as Bob is the initial sender, and Alice must receive Bob’s key before sending her own. Bob then receives Alice’s public key in the next like shown above. The recv(1024) statement ensures we have the

correct size of our message, and we convert each public key into an integer value using the int() function for future key generation.

This initial public key exchange sets the foundation for our scheme using asymmetric encryption. Alice and Bob have both exchanged each other's public keys, and are now moving on to accessing their private keys. This is where we see the first incorporation of symmetric encryption into the scheme. Our design focuses on having a secret symmetric key used for encryption and decryption that is nested within an asymmetric system facilitated by the Diffie-Hellman algorithm. This gives us the benefits of speed and simplicity that symmetric keys provide while also limiting the risk of violating forward secrecy and having the layered protection of asymmetric encryption[7].



[5]

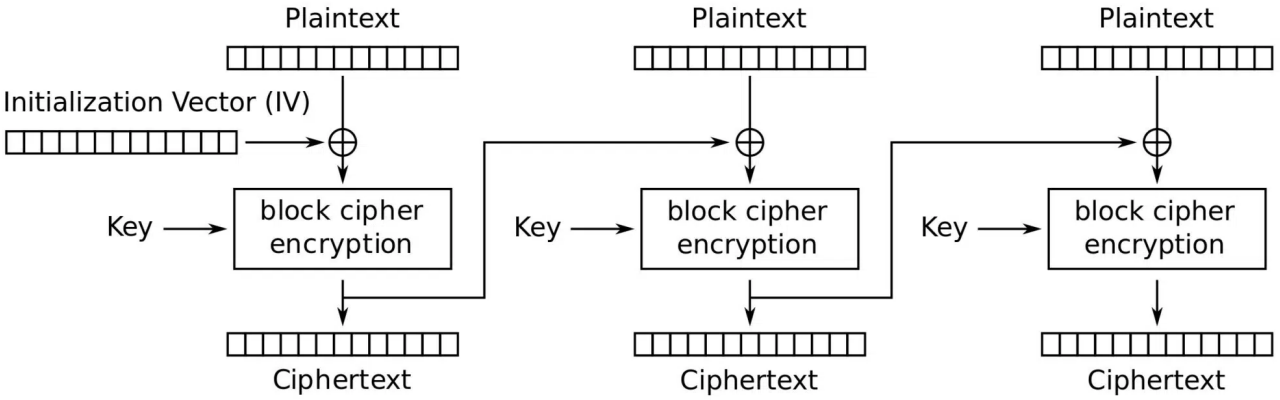
To accomplish this, we first used Alice's private key and Bob's public key to form a shared key that both parties would use. This process was identical on Bob's side (Bob's private key and Alice's public key), but I will focus on using the Alice side as the example to avoid redundancy.

```
shared = bytes(dhAlice.gen_shared_key(bobPK)[:-21]+"=",'utf8')
```

This statement reflects the syntax for generating the shared key that both parties will use, as Bob's key was generated with the same form just using Alice's public key. We then use this shared key to generate the first symmetric encryption, using the Fernet algorithm. We were able to import this from the python cryptography library. Using the Fernet structure, we are able to symmetrically encrypt messages inside of an asymmetric scheme.

The asymKE() function returns with this statement: `return Fernet(shared)`
We can see the shared key produced by both Alice and Bob is being used with the Fernet algorithm to create a symmetric key.

The Fernet algorithm has various advantages that were appealing in the design process of this system. Fernet is built on the foundation of the Advanced Encryption Standard, or AES. This is the most widespread and trusted system of symmetric encryption in current use. However, one fatal flaw of AES encryption is that it doesn't produce unique cipher text if the plain text stays the same. Therefore, someone trying to crack the encryption could study the cipher text long enough and start to find patterns in the cipher leading to them gaining knowledge on the plain text. As this obviously disrupts on our most basic concept of secure communication, we must find a solution to this *'Electronic Phone Book'* problem. The solution to this is choosing a mode of AES known as Cipher Block Chaining (CBC), which is utilized by the Fernet algorithm. CBC relies on the XOR logic function and constant randomization within a block chaining process to constantly produce unique cipher text even if the plain text is the same.



Cipher Block Chaining (CBC) mode encryption [6]

This image reflects the complexity of the CBC encryption mode. We see the XOR symbol being used heavily, as this function supports our idea of security within communication. The XOR function also has a special property in that you cannot predict with any better odds than a coin flip what the inputs are for any 1 bit of output. For example, if my two input bits into an XOR gate are 1 and 1, the output is 0, but if my input bits are 0 and 0, the output is also 0. The adversary has no way of telling with any certainty what any bit of plain text is based on viewing

the cipher text, providing us a solid foundation of security within CBC. Within this AES mode of encryption, we are chaining together small chunks (blocks) of plain text XOR with cipher text from the previous block. This forms a compacting chain of encrypted blocks as the encryption keeps building on top of itself, getting ever more so complex as it goes further down the chain; hence, cipher *block chaining.*

Continuing with CBC, we can see the process starts with an initialization vector (IV) as there is no cipher text to XOR with any plain text in the beginning. The Fernet algorithm chooses to build this IV in a rather unique way, and provided a clear advantage when designing the encryption scheme. Fernet records the amount of seconds passed from January 1st, 1970 to the time the Fernet key is created and uses the os.random library from python to form a 64 bit integer used to randomize the IV. This way, the keys are constantly being randomized by factors independent of our plain text which only improves our ability to produce unique keys despite similar input, enhancing our security as a whole[8].

After generating the shared key and using it to create a symmetric Fernet key, we have now integrated symmetric and asymmetric encryption together. We now are able to use this Fernet key to encrypt and decrypt messages and send them across our socket network.

```
temp = f.encrypt(toSend)
conn.sendall(temp)
```

The next step was to design the functions for sending and receiving messages across the network: Alice serving as the sender this time and Bob as the receiver. Alice runs the first command to encrypt the message toSend, which is just a message read in from a text file given by the user who wants to encrypt and send information. The variable ‘f’ is an instance of the Fernet library as we are accessing its’ ability to encrypt messages. This encrypted message ‘temp’ is then sent using the socket function ‘sendall()’. Bob then has a recMsg() function in which he is able to use the socket recv() function seen earlier to receive the encrypted message

sent by Alice from the socket network. He then can use the same Fernet key ‘f’, as this Fernet key was generated using their shared key, and use it’s decrypt() method to decrypt the message. We can see this process in the code:

```
toDec = s.recv(1024)

s.close()

decryptedMsg = f.decrypt(toDec)
```

One of our next design choices was the incorporation of adding file system functionality to the encryption system. We wanted to emulate real-world applications as much as possible, which is why we chose using socket to form a real-world view of encrypted network traffic. However, we recognized that users may want to encrypt or decrypt files on their own systems, and may want to save/download their keys and encrypted messages within files. The following lines of code show the syntax used to provide this functionality:

First with writing the decrypted message to a text file

```
decMsgFile = open("decrypted.txt", 'wb')

decMsgFile.write(decryptedMsg)
decMsgFile.close()
```

Next, writing the encrypted message to its own text file

```
encryptedMessageFile = open("encrypted.txt", 'wb')

encryptedMessageFile.write(temp)
encryptedMessageFile.close()
```

And lastly, writing the shared key to its own text file

```
sharedKeyFile = open("sharedKey.txt", 'wb')

sharedKeyFile.write(shared)
sharedKeyFile.close()
```

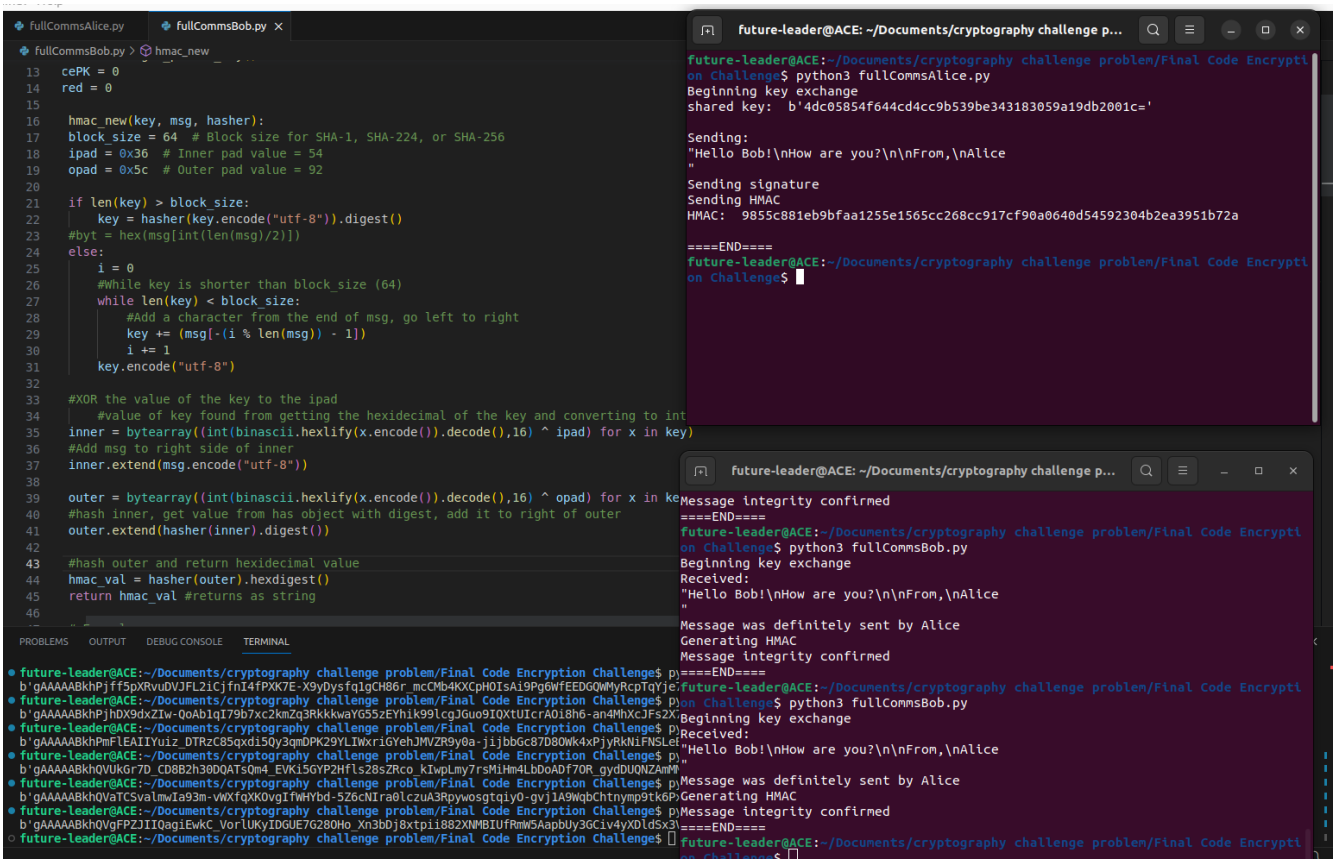
We also chose to use this same file functionality to read in the message that the user wanted to be encrypted. This way, we can provide a small layer of abstraction for the user so they won't have to interact with the code directly; they can simply put their code in the 'message.txt' file and let the program accomplish the rest.

We have now seen the encryption scheme combine the technologies of asymmetric encryption and symmetric encryption, accomplishing a solid form of providing confidentiality to the user. Our next steps are to try and knock down the pillars of integrity and authentication. As for integrity, we needed a form of a message authentication code to be able to verify the information wasn't altered as it traveled across a network. We implemented an hmac_new() function to construct a message authentication code system. We chose to follow the hmac algorithm as it allows for more secure encryption of both the key and the message itself, and allowed us to use a popular hashing scheme known as SHA256. We can see this inclusion in the function declaration and call below:

```
def hmac_new(key, msg, hasher):
```

```
HMAC = hmac_new(str(shared), msg.decode('utf8'), hashlib.sha256)
```

As shown above, we are using the shared key, user message and SHA256 hashing scheme to create a MAC that will allow us to verify the integrity of our message. This hmac_new() function is present in both python programs as we use it to both encode messages with a MAC and check to see if a decrypted message has remained unchanged. The main functionality of this hmac_new() function is built upon using the XOR logic function with two set hexadecimal values recommended by the current SHA256 hashing library. This XOR logic works to create a MAC using the shared key and desired message while also checking to make sure the hash and MAC is consistent with the size of the message[9].

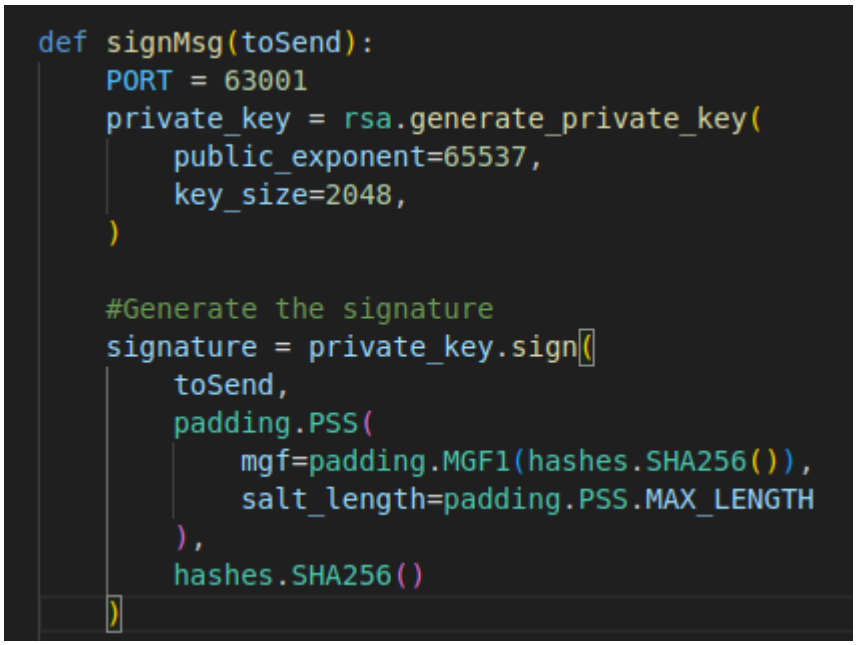


The image above displays this message sending and HMAC creation in action as we can see the main program logic for the hmac_new() function, shared by both Bob and Alice, on the left. The two terminals on the right show Alice on the top generating her shared key, sending a message read in from a file and sending her HMAC to Bob. The HMAC is also printed to the screen for visualization purposes but is removed from the final rendition of the program to avoid security concerns. We then see in Bob's terminal that he receives the message, checks and verifies that the message integrity is confirmed and has not been altered[10].

We can also see the inclusion of digital signature statements being sent along with the HMAC. This is our last step in designing our wholistic encryption scheme. These digital signatures help provide authentication to the receiver that they got the message from the desired sender, and further establishes a stronger connection between the two users.

In the code we have Alice acting as the signing party, as she actually creates the signature and appends it to the message, while Bob has a function to verify the signature to make sure the message was truly sent by Alice. I previously mentioned that there are multiple versions of asymmetric encryption that are in use today, with RSA being the most popular asymmetric key generation algorithm. With the idea of incorporating more encryption within our scheme, we chose to use RSA to build another set of keys we could then sign messages with. The python RSA importable library allows us to create a private key and then use it to directly sign a message with the RSA attribute class .sign().

This code shows the function within the Alice program using the RSA library to generate a private key and use it to sign the message 'toSend'. We can also see that this .sign() method includes both another instance of SHA256 and the added security of padding which both ensures that the signature is the correct size and helps to remove any predictability of the cipher text or signature itself. This quality of interweaving cryptographic techniques together was our main priority within designing this scheme, adding layers of security throughout every step of the protocol[11].



The signMsg() function then opens up another socket connection to send the signature and signed message over to Bob, who then has a verifyMsg() function to help authenticate the sender. This verifying function receives the signature, signed message and original message. It then loads in a serialized version of the sender's public key that it received from the RSA

generation to be able to authenticate the sender. We see the receiver, in this case Bob, use the key's attribute '.verify()':

```
try:
    publicKey.verify(
        signedDoc,      #verifying the signed message
        msg,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return True
except:
    return False
```

We pass the signed message, original message, padding and hashing algorithms used in the signature creation step to the verify() function and if it returns True, we know the sender is authenticated. We can see Alice sending the signature, Bob receiving it, running it through verification, and verifies that the message was sent by Alice.

4. Results:

We have now seen the integration of the four main tools of modern encryption into one single scheme to produce secure communication across a socket network. This encryption protocol is able to use asymmetric encryption to create a key pair using the Diffie-Hellman key exchange algorithm and initialize a form of encrypted communication over the network. It then uses these asymmetric keys to generate a shared key, which is then passed through a Fernet algorithm to produce a symmetric key used for encryption and decryption. The program reads in information from a file, abstracting the code into a more user-friendly manner, and processes that

information in a secure fashion. We then were able to create our own hashing Message Authentication Code system to produce secret keys that we can append onto messages after they're encrypted. This allows for the receiver to verify the integrity of the message after it has been sent over the network. And finally, we were able to use an asymmetric set of keys generated by the RSA algorithm to produce digital signatures of messages being sent across the network. Using the public key of this RSA pair, the receiver can then authenticate the sender of the message. These tools together make up a functioning, wholistic encryption scheme.

5. Discussion:

We have developed a program that can read in a message from a file, use asymmetric and symmetric encryption to provide confidentiality, generate hashed message authentication codes to verify the integrity of the message, and create a digital signature using RSA key pairs to authenticate the sender of the message.

5.1. Limitations:

With the inclusion of file system functionality, being able to read the message in from an external text file and being able to write to/save information on external files eliminated one of our preliminary limitations. For now, I would argue the biggest limitation on our scheme is we don't know how widespread the network can be to maintain functionality as it has only been tested on a short-distanced socket network connection. Also, with the use of the Fernet algorithm, the size of our files are restrained to be slightly smaller than if we would've chosen a less complex algorithm. However, the security properties and block chaining process of the Fernet key was worth the sacrifice in performance in our opinions.

5.2. Impact:

This program highlights the concept that integration of multiple tools of encryption can be combined into one in a manner that increases the security of our information by ensuring all of confidentiality, integrity, availability and authentication. This scheme is by no means perfect, but its' real world applications seem to be on the horizon as it is able to securely communicate over a network, as is often used in real-world applications. The capability of interacting with the file system on the local computer also can allow it to be used by single systems working independently to encrypt their own data. The abstraction created by the program reading in the information from a file means the user can just enter their data in a text file and run a simple command to get encrypted data. Whether it's passwords, sensitive files, or any other personal data, this program can easily be extended to encrypt information on one's own system.

As the world of cyber warfare continues to evolve and grow even more complex, encryption of information both while in transit and within one's own system has never been so vital.

5.3. Future Work:

Interestingly, the future of cryptography will look incredibly different than the solution we have posed in this report. With the introduction of quantum computing, the heightened computing power will most likely render this type of binary encryption useless. However, we seem to be at least a few years off from this exciting advancement. In the meantime, this program desperately needs widespread testing and adaptations to a larger network. Connections among systems in the field of cyber security operations aren't getting any simpler as networks continue to grow more and more complex across the world. Being able to incorporate this integrated

system of encryption tools across a large, widespread network could have some very impactful implications for real world use.

If given more time on this project I think our next steps would follow in this path of building the infrastructure from within the code to support multiple users across a network. As we look farther down the road I would suggest we try and adapt this encryption with the growing standards of quantum computing. This program could hopefully be an influence in trying to incorporate both quantum and binary encryption schemes into one for the future, as we endlessly pursue the frightening future of cryptography.

[1] “Symmetric-Key Cryptography,” *www.cs.cornell.edu*.

https://www.cs.cornell.edu/courses/cs5430/2010sp/TL03.symmetric.html

(accessed Jun. 11, 2023).

[2] Cloudflare, “What is asymmetric encryption? | Asymmetric vs. symmetric encryption | Cloudflare,” *Cloudflare*. Available: https://www.cloudflare.com/learning/ssl/what-is-asymmetric-encryption/

[3] Fortinet, “What Is a Message Authentication Code (MAC)?,” *Fortinet*. https://www.fortinet.com/resources/cyberglossary/message-authentication-code

[4] “Understanding Digital Signatures | CISA,” *Cybersecurity and Infrastructure Security Agency CISA*. https://www.cisa.gov/news-events/news/understanding-digital-signatures

[5] “Is there any particular reason to use Diffie-Hellman over RSA for key exchange?,” *Information Security Stack Exchange*. https://security.stackexchange.com/questions/35471/is-there-any-particular-reason-to-use-diffie-hellman-over-rsa-for-key-exchange (accessed Jun. 11, 2023).

[6] Real Python, “Socket Programming in Python (Guide),” *Realpython.com*, Aug. 2018. https://realpython.com/python-sockets/

[7] “How to combine symmetric and asymmetric encryption to encrypt large files?,” *Information Security Stack Exchange*. https://security.stackexchange.com/questions/203286/how-to-combine-symmetric-and-asymmetric-encryption-to-encrypt-large-files (accessed Jun. 11, 2023).

[8] “What is fernet and when should you use it?,” *Comparitech.com*, 2022. https://www.comparitech.com/blog/information-security/what-is-fernet/

[9] “hmac — Keyed-Hashing for Message Authentication,” *Python documentation*. https://docs.python.org/3/library/hmac.html (accessed Jun. 11, 2023).

[10] “Python generate HMAC-SHA-256 from string,” *browse-tutorials.com*, Feb. 02, 2016. https://browse-tutorials.com/snippet/python-generate-hmac-sha-256-string (accessed Jun. 11, 2023).

[11] “RSA Signatures - Practical Cryptography for Developers,” *Nakov.com*, 2022. https://cryptobook.nakov.com/digital-signatures/rsa-signatures

Image Citations:

[1] H. Sidhpurwala, “A Brief History of Cryptography,” *www.redhat.com*, Aug. 14, 2013. https://www.redhat.com/en/blog/brief-history-cryptography

[2] SSL2BUY, “Symmetric vs. Asymmetric Encryption – What Are differences?,” *SSL2BUY Wiki - Get Solution for SSL Certificate Queries*, Feb. 07, 2019. https://www.ssl2buy.com/wiki/symmetric-vs-asymmetric-encryption-what-are-differences

[3] A. Miya, “Message Authentication Code (MAC) - Use My Notes,” Sep. 17, 2021. https://usemynotes.com/message-authentication-code-mac/

[4] tutorialspoint, *Model of Digital Signature*. 2019. Accessed: Jun. 11, 2023. [Online]. Available: https://www.tutorialspoint.com/cryptography/cryptography_digital_signatures.htm

[5] Wikipedia Contributors, “Public-key cryptography,” *Wikipedia*, Jul. 19, 2019. https://en.wikipedia.org/wiki/Public-key_cryptography

[6] “What is CBC?,” *Educative: Interactive Courses for Software Developers*. https://www.educative.io/answers/what-is-cbc