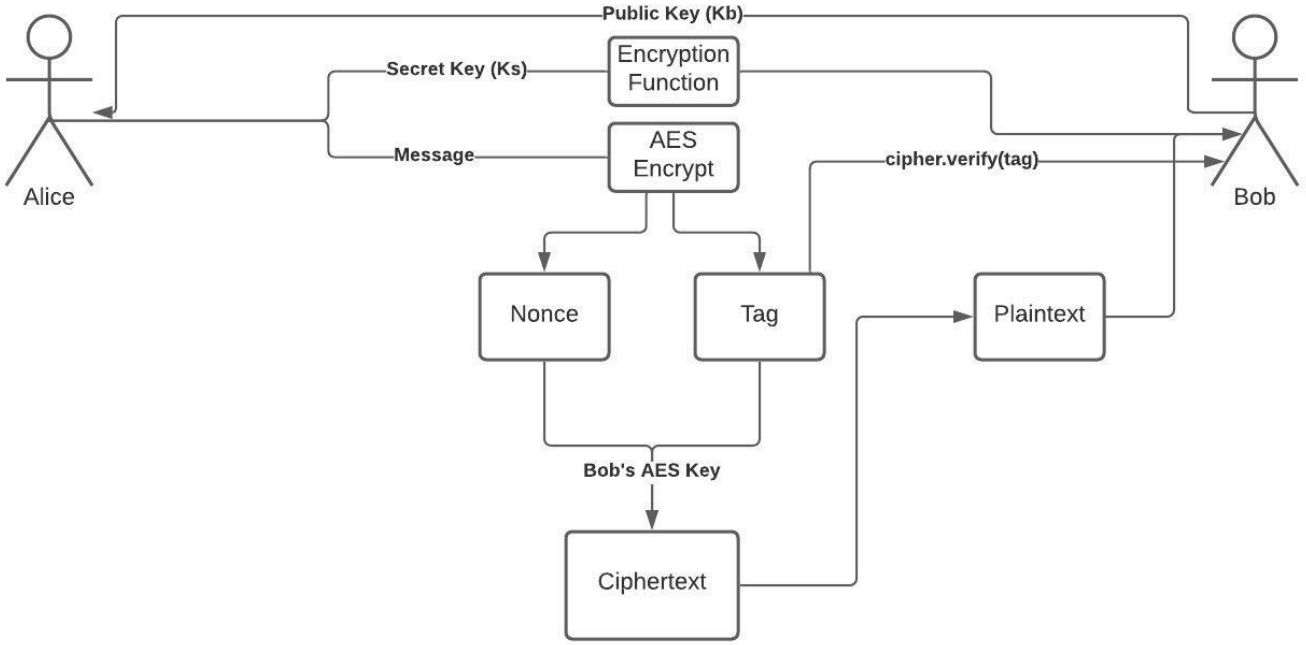
End-to-End Cryptography for Messaging

ZEXING LI & SIMON FEDOTOV

Security Analysis

- Man-in-the-middle attacks
  - 2048-bit AES encryption rules out some MitM attacks
    - Mitigates attacks on integrity of messages
- Eavesdropping
  - Our encryption functions make it hard for an attacker to eavesdrop on messages being sent between two parties
  - Ensures privacy of any messages sent
- Alteration for messages in transit
  - Message tag verification will check the authenticity of a message
    - Our program will let you know if the integrity of the message has been compromised



Design Validity

Implementation Validity

- Private and public keys are generated, and public key is exported to 'RSA_publickey.pem' file to be accessed later
- Private key is generated with 2048 bits, the recommended length for better security

```python
def Alice_RSA_start():
    # Generate public/private key for key sharing
    return RSA_generate_keys()
```

```python
def RSA_generate_keys():
    # Generate the private key
    # https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html
    private_key = RSA.generate(2048)

    # Generate the public key
    public_key = private_key.publickey()

    # Export public key
    f1 = open('RSA_publickey.pem', 'wb')
    f1.write(public_key.export_key('PEM'))
    f1.close()

    return private_key
```

Implementation Validity (cont.)

- os.urandom(32) is used because it is a truly random number generator that is hard to predict, unlike the native Python random number generator
- Bob encrypts his AES key using the public key generated by Alice

```python
def Bob_AES_key_gen_send():
    # Generate AES key
    aes_key = AES_keygen()

    # Share AES key to Alice using RSA
    RSA_encrypt_public_key(aes_key)

    # Return AES key for testing
    return aes_key
```

```python
def AES_keygen():
    # https://stackoverflow.com
    # 256-bit key
    return os.urandom(32)
```

```python
def RSA_encrypt_public_key(a_message):
    # Read the received public key file
    f1 = open('RSA_publickey.pem', 'rb')
    public_key = RSA.import_key(f1.read())
    f1.close()

    # Encrypt with public key
    encryptor = PKCS1_OAEP.new(public_key)
    encrypted_msg = encryptor.encrypt(a_message)

    # Encode as base64
    encoded_encrypted_msg = base64.b64encode(encrypted_msg)

    # Save the encrypted message as file
    f2 = open('Encrypted_AESkey.txt', 'wb')
    f2.write(encoded_encrypted_msg)
    f2.close()
    return encoded_encrypted_msg
```

```python
def Alice_get_AES_key(private_key):
    # decrypt the AES key sent by Bob
    return RSA_decrypt_private_key(private_key)
```

```python
def RSA_decrypt_private_key(private_key):
    # Read encrypted message from file
    f1 = open('Encrypted_AESkey.txt', 'rb')
    encoded_encrypted_msg = f1.read()
    f1.close()

    # Decrypt received message
    encryptor = PKCS1_OAEP.new(private_key)
    decoded_encrypted_msg = base64.b64decode(encoded_encrypted_msg)

    # Decode received message
    decoded_decrypted_msg = encryptor.decrypt(decoded_encrypted_msg)

    return decoded_decrypted_msg
```

Implementation Validity (cont.

DECRYPTS AND DECODES THE RECEIVED AES KEY USING THE PRIVATE KEY AND RETURNS IT.

Implementation Validity (cont.)

Nonce, tag, and ciphertext variables are created and read into separate files

Tag is used to verify the authenticity of the message

```
def AES_encrypt(aes_key):
    # https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html
    # Encrypt data with AES
    cipher = AES.new(aes_key, AES.MODE_EAX)
    # Nonce is a number that can be only used once!
    nonce = cipher.nonce
    data = b"Hello world!"
    ciphertext, tag = cipher.encrypt_and_digest(data)
    print(ciphertext)
    # Export AES ciphertext
    f1 = open('AES_ciphertext.txt', 'wb')
    f1.write(ciphertext)
    f1.close()

    # Export AES tag
    f2 = open('AES_tag.txt', 'wb')
    f2.write(tag)
    f2.close()

    # Export Nonce
    f3 = open('AES_nonce.txt', 'wb')
    f3.write(nonce)
    f3.close()
```

Implementation Validity (cont.)

Files are opened and read to decrypt and verify the authenticity of the message

If the key or the message were tampered with in transit, the program throws a value error

Otherwise, the message is printed out and verified as authentic

```python
def AES_decrypt(aes_key):
    # https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html
    # Decrypt data with AES

    # Read AES ciphertext
    f1 = open('AES_ciphertext.txt', 'rb')
    ciphertext = f1.read()
    f1.close()

    # Read AES tag
    f2 = open('AES_tag.txt', 'rb')
    tag = f2.read()
    f2.close()

    # Read Nonce
    f3 = open('AES_nonce.txt', 'rb')
    nonce = f3.read()
    f3.close()

    cipher = AES.new(aes_key, AES.MODE_EAX, nonce=nonce)
    plaintext = cipher.decrypt(ciphertext)
    try:
        cipher.verify(tag)
        print("The message is authentic:", plaintext)
    except ValueError:
        print("Key incorrect or message corrupted")
```