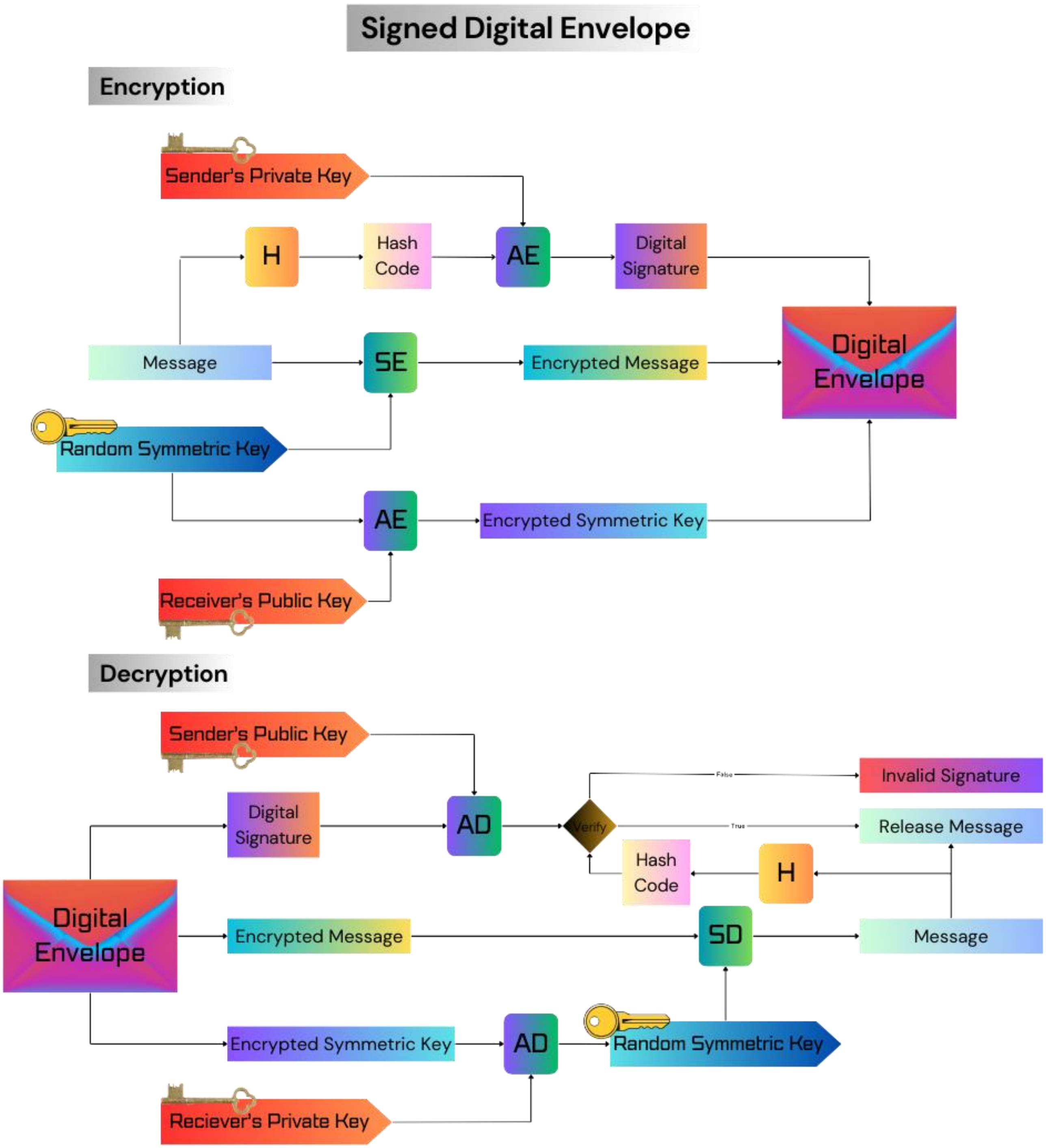
COSC 3364 – Principles of Cybersecurity
Lab 03
Signed Digital Envelope

1. Develop a program to generate a signed digital envelope utilizing RSA for the asymmetric encryption of the symmetric key & digital signature, AES in CBC mode for the symmetric encryption of the message, and SHA256 for the hash function.



AES

class **cryptography.hazmat.primitives.ciphers.algorithms.AES**(*key*) [source]

AES (Advanced Encryption Standard) is a block cipher standardized by NIST. AES is both fast, and cryptographically strong. It is a good default choice for encryption.

Parameters: **key** (bytes-like) – The secret key. This must be kept secret. Either `128`, `192`, or `256` bits long.

```
>>> import os
>>> from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
>>> key = os.urandom(32)
>>> iv = os.urandom(16)
>>> cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
>>> encryptor = cipher.encryptor()
>>> ct = encryptor.update(b"a secret message") + encryptor.finalize()
>>> decryptor = cipher.decryptor()
>>> decryptor.update(ct) + decryptor.finalize()
b'a secret message'
```

RSA

RSA is a public-key algorithm for encrypting and signing messages.

Generation

Unlike symmetric cryptography, where the key is typically just a random series of bytes, RSA keys have a complex internal structure with specific mathematical properties.

cryptography.hazmat.primitives.asymmetric.rsa.generate_private_key(*public_exponent*, *key_size*) [source]

Added in version 0.5.

Changed in version 3.0: Tightened restrictions on `public_exponent`.

Generates a new RSA private key. `key_size` describes how many bits long the key should be. Larger keys provide more security; currently `1024` and below are considered breakable while `2048` or `4096` are reasonable default key sizes for new keys. The `public_exponent` indicates what one mathematical property of the key generation will be. Unless you have a specific reason to do otherwise, you should always use 65537.

```
>>> from cryptography.hazmat.primitives.asymmetric import rsa
>>> private_key = rsa.generate_private_key(
...     public_exponent=65537,
...     key_size=2048,
... )
```

Parameters:

- **public_exponent** (*int*) – The public exponent of the new key. Either 65537 or 3 (for legacy purposes). Almost everyone should use 65537.
- **key_size** (*int*) – The length of the modulus in bits. For keys generated in 2015 it is strongly recommended to be at least 2048 (See page 41). It must not be less than 512.

Returns: An instance of `RSAPrivateKey`.

RSA Encryption

RSA encryption is interesting because encryption is performed using the **public** key, meaning anyone can encrypt data. The data is then decrypted using the **private** key.

Like signatures, RSA supports encryption with several different padding options. Here's an example using a secure padding and hash function:

```
>>> message = b"encrypted data"
>>> ciphertext = public_key.encrypt(
...     message,
...     padding.OAEP(
...         mgf=padding.MGF1(algorithm=hashes.SHA256()),
...         algorithm=hashes.SHA256(),
...         label=None
...     )
... )
```

Valid paddings for encryption are `OAEP` and `PKCS1v15`. `OAEP` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

RSA Decryption

Once you have an encrypted message, it can be decrypted using the private key:

```
>>> plaintext = private_key.decrypt(
...     ciphertext,
...     padding.OAEP(
...         mgf=padding.MGF1(algorithm=hashes.SHA256()),
...         algorithm=hashes.SHA256(),
...         label=None
...     )
... )
>>> plaintext == message
True
```

RSA Signing

A private key can be used to sign a message. This allows anyone with the public key to verify that the message was created by someone who possesses the corresponding private key. RSA signatures require a specific hash function, and padding to be used. Here is an example of signing `message` using RSA, with a secure hash function and padding:

```
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import padding
>>> message = b"A message I want to sign"
>>> signature = private_key.sign(
...     message,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     hashes.SHA256()
... )
```

Valid paddings for signatures are `PSS` and `PKCS1v15`. `PSS` is the recommended choice for any new protocols or applications, `PKCS1v15` should only be used to support legacy protocols.

RSA Verification

The previous section describes what to do if you have a private key and want to sign something. If you have a public key, a message, a signature, and the signing algorithm that was used you can check that the private key associated with a given public key was used to sign that specific message. You can obtain a public key to use in verification using `load_pem_public_key()`, `load_der_public_key()`, `public_key()`, or `public_key()`.

```
>>> public_key = private_key.public_key()
>>> public_key.verify(
...     signature,
...     message,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     hashes.SHA256()
... )
```

If the signature does not match, `verify()` will raise an `InvalidSignature` exception.

SHA-256

class `cryptography.hazmat.primitives.hashes.SHA256` [source]

SHA-256 is a cryptographic hash function from the SHA-2 family and is standardized by NIST. It produces a 256-bit message digest.

CBC

class `cryptography.hazmat.primitives.ciphers.modes.CBC`(*initialization_vector*) [source]

CBC (Cipher Block Chaining) is a mode of operation for block ciphers. It is considered cryptographically strong.

Padding is required when using this mode.

Parameters: **initialization_vector** (bytes-like) – Must be random bytes. They do not need to be kept secret and they can be included in a transmitted message. Must be the same number of bytes as the `block_size` of the cipher. Each time something is encrypted a new `initialization_vector` should be generated. Do not reuse an `initialization_vector` with a given `key`, and particularly do not use a constant `initialization_vector`.

PSS

class `cryptography.hazmat.primitives.asymmetric.padding.PSS`(*mgf, salt_length*) [source]

Added in version 0.3.

Changed in version 0.4: Added `salt_length` parameter.

PSS (Probabilistic Signature Scheme) is a signature scheme defined in **RFC 3447**. It is more complex than PKCS1 but possesses a security proof. This is the recommended padding algorithm for RSA signatures. It cannot be used with RSA encryption.

Parameters:
- **mgf** – A mask generation function object. At this time the only supported MGF is `MGF1`.
- **salt_length** (*int*) – The length of the salt. It is recommended that this be set to `PSS.DIGEST_LENGTH` or `PSS.MAX_LENGTH`.

`MAX_LENGTH`

Pass this attribute to `salt_length` to get the maximum salt length available.

`DIGEST_LENGTH`

Added in version 37.0.0.

Pass this attribute to `salt_length` to set the salt length to the byte length of the digest passed when calling `sign`. Note that this is **not** the length of the digest passed to `MGF1`.

`AUTO`

Added in version 37.0.0.

Pass this attribute to `salt_length` to automatically determine the salt length when verifying. Raises `ValueError` if used when signing.

`mgf`

Type: `MGF`

Added in version 42.0.0.

The padding's mask generation function (MGF).

MGF1

class `cryptography.hazmat.primitives.asymmetric.padding.MGF1`(*algorithm*) [source]

Added in version 0.3.

Changed in version 0.6: Removed the deprecated `salt_length` parameter.

MGF1 (Mask Generation Function 1) is used as the mask generation function in `PSS` and `OAEP` padding. It takes a hash algorithm.

Parameters: **algorithm** – An instance of `HashAlgorithm`.

OAEP

class **cryptography.hazmat.primitives.asymmetric.padding.OAEP**(*mgf*, *algorithm*, *label*) [source]

Added in version 0.4.

OAEP (Optimal Asymmetric Encryption Padding) is a padding scheme defined in **RFC 3447**. It provides probabilistic encryption and is proven secure against several attack types. This is the recommended padding algorithm for RSA encryption. It cannot be used with RSA signing.

Parameters:

- **mgf** – A mask generation function object. At this time the only supported MGF is `MGF1`.
- **algorithm** – An instance of `HashAlgorithm`.
- **label** (*bytes*) – A label to apply. This is a rarely used field and should typically be set to `None` or `b""`, which are equivalent.

PKCS7

class **cryptography.hazmat.primitives.padding.PKCS7**(*block_size*) [source]

PKCS7 padding is a generalization of PKCS5 padding (also known as standard padding). PKCS7 padding works by appending `N` bytes with the value of `chr(N)`, where `N` is the number of bytes required to make the final block of data the same size as the block size. A simple example of padding is:

```
>>> from cryptography.hazmat.primitives import padding
>>> padder = padding.PKCS7(128).padder()
>>> padded_data = padder.update(b"11111111111111112222222222")
>>> padded_data += padder.finalize()
>>> padded_data
b'11111111111111112222222222\x06\x06\x06\x06\x06\x06'
>>> unpadder = padding.PKCS7(128).unpadder()
>>> data = unpadder.update(padded_data)
>>> data += unpadder.finalize()
>>> data
b'11111111111111112222222222'
```

Parameters: **block_size** – The size of the block in bits that the data is being padded to.

Raises: ValueError – Raised if block size is not a multiple of 8 or is not between 0 and 2040 inclusive.

padder() [source]

Returns: A padding `PaddingContext` instance.

unpadder() [source]

Returns: An unpadding `PaddingContext` instance.

InvalidSignature Exception

class **cryptography.exceptions.InvalidSignature** [source]

This is raised when signature verification fails. This can occur with HMAC or asymmetric key signature validation.

Results:

Source Code:

```python
###########################CODE HERE#############################################
    def encrypt_digital_env(self, plaintext):
        plaintext = plaintext.encode()
        key, iv, ciphertext = self.symmetric_encryption(plaintext)
        key_iv = key + iv
        encrypted_key = self.asymmetric_encrypt(key_iv)
        signature = self.sign(plaintext)
        digital_envelope = self.package_envelope(encrypted_key, ciphertext, signature)
        digital_envelope = digital_envelope.hex()
        return digital_envelope

    def decrypt_digital_env(self, digital_envelope):
        digital_envelope = bytes.fromhex(digital_envelope)
        encrypted_key, ciphertext, signature = self.unpackage_envelope(digital_envelope)
        decrypted_key = self.asymmetric_decrypt(encrypted_key)
        plaintext = self.symmetric_decryption(ciphertext, decrypted_key)

        if (self.verify_signature(signature, plaintext)): #####{Verified}#####
            plaintext = plaintext.decode()
            return plaintext #####{Plaintext}#####
        else:
            return "Invalid Signature"

    def gen_senders_keys(self):
        self.senders_private_key = rsa.generate_private_key(public_exponent = 65537, key_size = 2048)
        self.senders_public_key = self.senders_private_key.public_key()


    def gen_receivers_keys(self):
        self.receivers = rsa.generate_private_key(public_exponent = 65537, key_size = 2048)
        self.receivers_public_key = self.senders_private_key.public_key()

    def symmetric_encryption(self, plaintext):
        key = os.urandom(32)
        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
        encryptor = cipher.encryptor()
        padder = PKCS7(algorithms.AES.block_size).padder()
        #cipher.algorithm
        padded_message = padder.update(plaintext) + padder.finalize()
        ciphertext = encryptor.update(padded_message) + encryptor.finalize()
        return (key, iv, ciphertext)
```

```python
def symmetric_decryption(self, ciphertext, decrypted_key):
    key = decrypted_key[:32]
    iv = decrypted_key[32:]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    plaintext = decryptor.update(ciphertext) + decryptor.finalize()
    unpadder = PKCS7(algorithms.AES.block_size).unpadder()
    plaintext = unpadder.update(plaintext) + unpadder.finalize()
    return plaintext

def asymmetric_encrypt(self, key_iv):
    encrypted_key = self.senders_public_key.encrypt(
        key_iv,
        padding.OAEP(
            mgf = padding.MGF1(algorithm = hashes.SHA256()),
            algorithm = hashes.SHA256(),
            label = None
        )
    )
    return encrypted_key

def asymmetric_decrypt(self, encrypted_key):
    decrypted_key = self.senders_private_key.decrypt(
        encrypted_key,
        padding.OAEP(
            mgf = padding.MGF1(algorithm = hashes.SHA256()),
            algorithm = hashes.SHA256(),
            label = None
        )
    )
    return decrypted_key

def sign(self, plaintext):
    signature = self.senders_private_key.sign(
        plaintext,
        padding.PSS(
            mgf = padding.MGF1(hashes.SHA256()),
            salt_length = padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature

def verify_signature(self, signature, plaintext):
    try:
        self.senders_public_key.verify(
            signature,
            plaintext,
            padding.PSS(
                mgf = padding.MGF1(hashes.SHA256()),
```

```python
    def verify_signature(self, signature, plaintext):
        try:
            self.senders_public_key.verify(
                signature,
                plaintext,
                padding.PSS(
                    mgf = padding.MGF1(hashes.SHA256()),
                    salt_length = padding.PSS.MAX_LENGTH
                ),
                hashes.SHA256()
            )
            return True
        except InvalidSignature:
            return False

    def package_envelope(self, encrypted_key, ciphertext, signature):
        digital_envelope = encrypted_key + ciphertext + signature
        return digital_envelope

    def unpackage_envelope(self, digital_envelope):
        encrypted_key = digital_envelope[:256]

        ciphertext = digital_envelope[256:-256]
        #ciphertext = digital_envelope[256:272]

        signature = digital_envelope[-256:]
        #signature = digital_envelope[272:]
        return (encrypted_key, ciphertext, signature)
##################################################################################
```