Department of Computer Science and Engineering - University of New South Wales

Lab01 Report

COMP4337 - Securing Fixed and Wireless Network

Group T16B 2

Quynh Boi Phan (z5205690)

Jack Vuong (z5164149)

Python Code Implemented

tempdes.py

Figure 1.1 are the libraries imported into Python to implement the Cipher Block Chaining DES mode of operation:

```
import binascii
from Crypto.Cipher import DES
from Crypto import Random
import sys
```

Figure 1.1: Libraries used to implement DES-CBC.

The system first checks if the user enters the correct number of arguments in the main function. If not, it sends an error message and exits. This implementation is observed in *Figure 1.2*.

```
if (len(sys.argv) != 5):
    print("ArgumentError: Use tempdes.py as follow: python3 tempdes.py [iv] [key] [inputfile] [outputfile]")
    exit(1)
```

Figure 1.2: Argument checking before continuing.

The system then uses the functions `get_IV` and `get_key` to get the IV and key. In `get_IV` (*Figure 1.3*), the function checks if the string consists of only hexadecimal digits. The system then returns the binary string of the hexadecimal string.

```
def get_IV(iv):
    try:
        test_iv = int(iv, 16) # Checks if string consists of only hexadecimal digits.
    except ValueError:
        print("Error: iv must be only contain hexadecimal digits")
        exit(1)
    iv_bytestr = binascii.unhexlify(iv) # Converting to binary string.
    return iv_bytestr
```

Figure 1.3: `get_IV` *function.*

The `get_key` function follows the same structure; however, if the key does not match the requirements, a different error message appears.

After opening the file to read using the function `get_file` (*Figure 1.4*), the system begins the encryption process in `des_encrypt` (*Figure 1.5*). The message is converted to a binary string before encrypting. If the message is not a multiple of 8 bytes, the null character '/0' is appended to the original message until the encoded message is a multiple of 8 bytes.

```
def get_file(inputFile):
    with open(inputFile) as f:
        a = f.read() # Read input file.
    return a
```

Figure 1.4: `get_file` *function.*

```
def des_encrypt(msg, cbc_key, iv):
    new_msg = msg.encode('utf-8') # Converting to binary string.
    while len(new_msg) % 8 != 0: # While encoded message length is not multiple of 8
        msg += '\0' # Add null character
        new_msg = msg.encode('utf-8') # Converting to binary string.

    cipher = msg.DES.new(cbc_key, DES.MODE_CBC, iv)
    cipher_text = cipher.encrypt(new_msg) # Encrypt message.
    return cipher_text
```

Figure 1.5: `des_encrypt` *function.*

The ciphered text is then written into the output file in *Figure 1.6*.

```
with open(dest_file, "wb") as external_file:
    external_file.write(cipher_text) # Write cipher text to destination file.
    external_file.close()
```

Figure 1.6: Writing to the output file.

The message must first be decrypted using the function `des_decrypt` (*Figure 1.7*) and converted into a string again to return to its original state.

```
def des_decrypt(cipher_text, cbc_key, iv):
    cipher = msg.DES.new(cbc_key, DES.MODE_CBC, iv)
    decrypted_text = cipher.decrypt(cipher_text) # Decrypt message.
    decoded_text = decrypted_text.decode('utf-8') # Converting to string.
    return decoded_text
```

Figure 1.7: `des_decrypt` *function.*

tempaes.py

tempaes.py follows a similar structure to *tempdes.py*.

Figure 1.8 are the libraries imported into Python to implement the Cipher Block Chaining AES mode of operation:

```
import Crypto
from Crypto.PublicKey import AES
from Crypto import Random
import sys
```

Figure 1.8: Libraries used to implement AES-CBC.

Instead of specifying a *key* and *iv* used in the command line argument, it uses a random byte string generator for the *Crypto.Random* package to generate a byte string of length 16. The system then opens the reading file specified by the command line argument (*Figure 1.4*) and pads the text until the message is a multiple of 16. The message is then converted into a binary string before encrypting.

After decrypting a message, it is converted back into a string to return to its original state.

The complete code is observed in *Figure 1.9*.

```
cbc_key = Random.get_random_bytes(16) # Generating key
iv = Random.get_random_bytes(16) # Generating IV
aes1 = AES.new(cbc_key, AES.MODE_CBC, iv)
aes2 = AES.new(cbc_key, AES.MODE_CBC, iv)
```

```
plain_text = get_file(sys.argv[1]) # Opening file to read text (Figure 1.4)

enc_text = plain_text.encode('utf-8') # Converting to byte string
while len(enc_text) % 16 != 0: # Padding string to length of multiple 16.
    plain_text += '\0'
    enc_text = plain_text.encode('utf-8')

cipher_text = aes1.encrypt(enc_text) # Encrypt text

msg = aes2.decrypt(cipher_text) # Decrypt text
msg = msg.decode('utf-8') # Converting to normal string
```

Figure 1.9: tempaes.py implementation.

temprsa.py

Figure 1.10 are the libraries imported into Python to implement the RSA encryption and decryption:

```
import Crypto
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
from Crypto import Random
import ast
import sys
```

Figure 1.10: Libraries used to implement RSA encryption and decryption.

A random bytes generator from the *Crypto.Random* package generates the public and private keys. The system then opens the reading file specified by the command line argument (*Figure 1.4*) then convert the text into a binary string. The binary string is then encrypted.

After decrypting a message, it is converted back into a string to return to its original state.

The complete code is observed in Figure 1.11.

```
random_generator = Random.new().read
key = RSA.generate(1024, random_generator)  # generate pub and priv key

publickey = key.publickey()  # pub key export for exchange
plaint_text = get_file(sys.argv[1]) # Opening file to read text (Figure 1.4)
enc_text = plain_text.encode('utf-8') # Converting to byte string
```

```
encryptor = PKCS1_OAEP.new(publickey)
cipher_text = encryptor.encrypt(enc_text) # Encrypting text

decryptor = PKCS1_OAEP.new(key)
decrypted = decryptor.decrypt(ast.literal_eval(ast(cipher_text))) # Decrypting text
decode_text = decrypted.decode('utf-8') # Converting to normal string
```

Figure 1.11: temprsa.py implementation.

tempsha1.py

Figure 1.12 are the libraries imported into Python to implement the SHA-1 hash generation:

```
import hashlib
import sys
```

Figure 1.12: Libraries used to implement SHA-1 hash generation.

The system opens the reading file specified by the command line argument (*Figure 1.4*) then convert the text into a binary string. The binary string is used for the SHA-1 digest.

The hexadecimal equivalent of the SHA-1 digest can be read using `result.hexdigest()`, `result` being the SHA-1 digest.

The complete code is observed in Figure 1.11.

```
str = get_file(sys.argv[1]) # Opening file to read text (Figure 1.4)
result = hashlib.sha1(str.encode()) # Creating the SHA-1 digest.
print(result.hexdigest()) # Printing the hexadecimal equivalent of SHA-1 digest
```

Figure 1.13: tempsha1.py implementation.

temphmac.py

Figure 1.14 are the libraries imported into Python to implement the RSA encryption and decryption:

```
import hashlib
import hmac
import sys
```

Figure 1.14: Libraries used to implement HMAC signature generation.

The system creates a secret key and the digest maker before generating the HMAC signature. Upon opening the file, the system reads and update the digest maker 1024 bytes at a time.

The HMAC digest is then created, generating a string containing only hexadecimal digits.

The complete code is observed in Figure 1.15.

```
secret_key = 'secret-shared-key-goes-here'.encode('utf-8')
digest_maker = hmac.new(secret_key, digestmode=hashlib.md5)

f = open(sys.argv[1], 'rb')
try:
    while True:
    block = f.read(1024)
    if not block:
        break
    digest_maker.update(block)
finally:
      f.close()

digest = digest_maker.hexdigest()
```

Figure 1.15: temphmac.py implementation.

Timing the Performance of each Algorithm

The `time` library from Python was used for every algorithm to measure the performance time. *Figure 1.16* depicts the usage of the library in each code.

```
import time

... # Code before timing is needed
start = time.time() # Begin timer of area to measure
... # Code needed to be measured
time_taken = time.time() - start # Get time difference from start for performance time
time_taken = time_taken * 1000000 # Converting to microseconds
```

```
... # Code after timing is needed
```

Figure 1.15: Using Python `time` *library.*

Results

DES Encryption/Decryption Time

Figure 2.1 below is the graph of the DES encryption/decryption time.

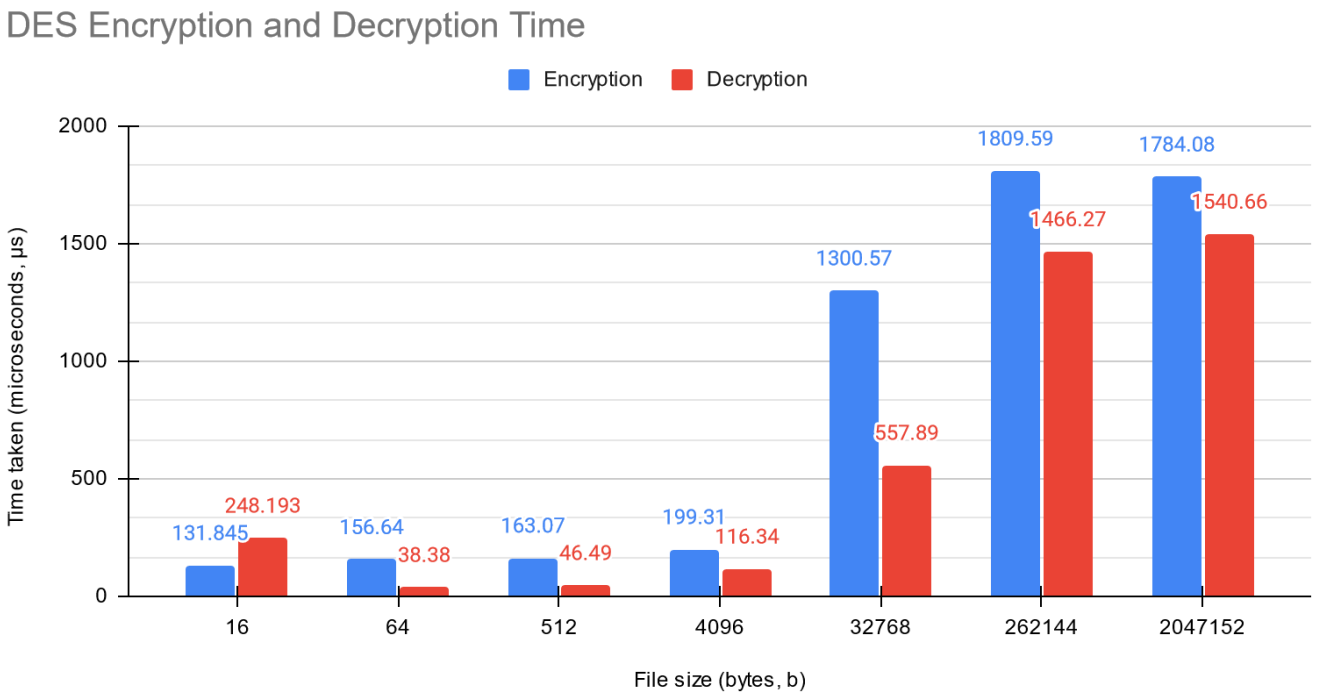

Figure 2.1: DES Encryption and Decryption Time.

AES Encryption/Decryption Time

Figure 2.2 below is the graph of the AES encryption/decryption time.

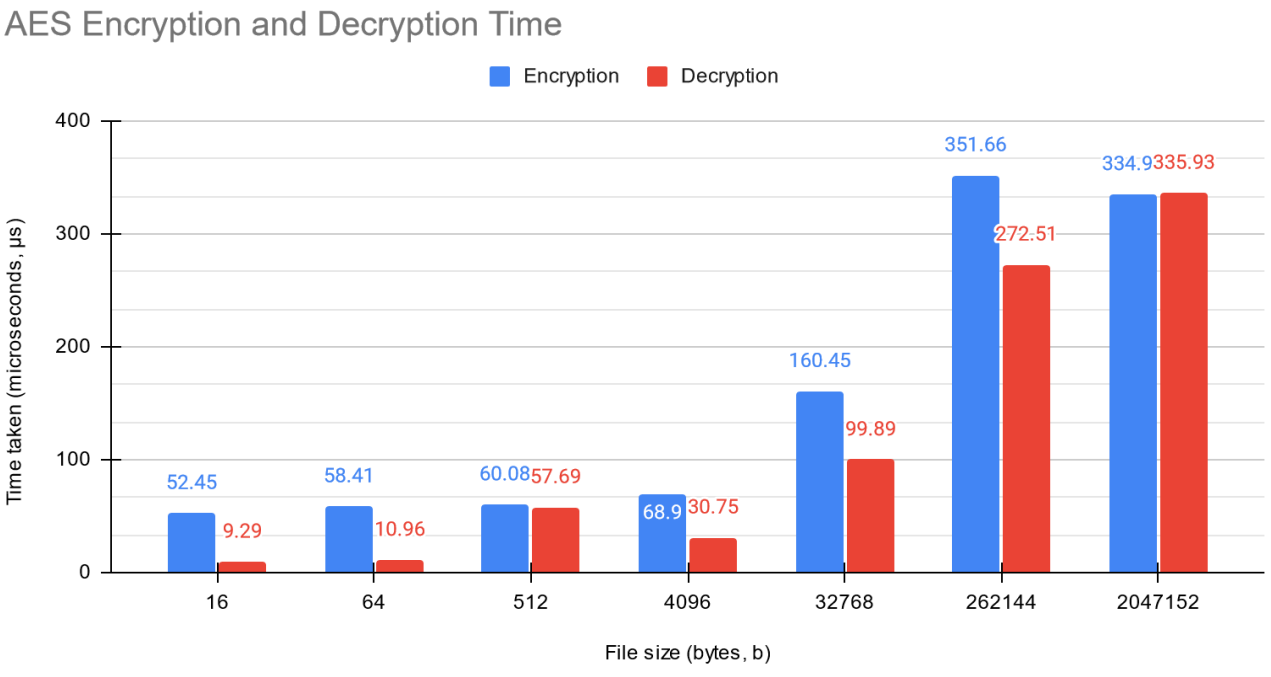

Figure 2.2: AES Encryption and Decryption Time.

RSA Encryption/Decryption Time

Figure 2.3 below is the graph of the RSA encryption/decryption time.

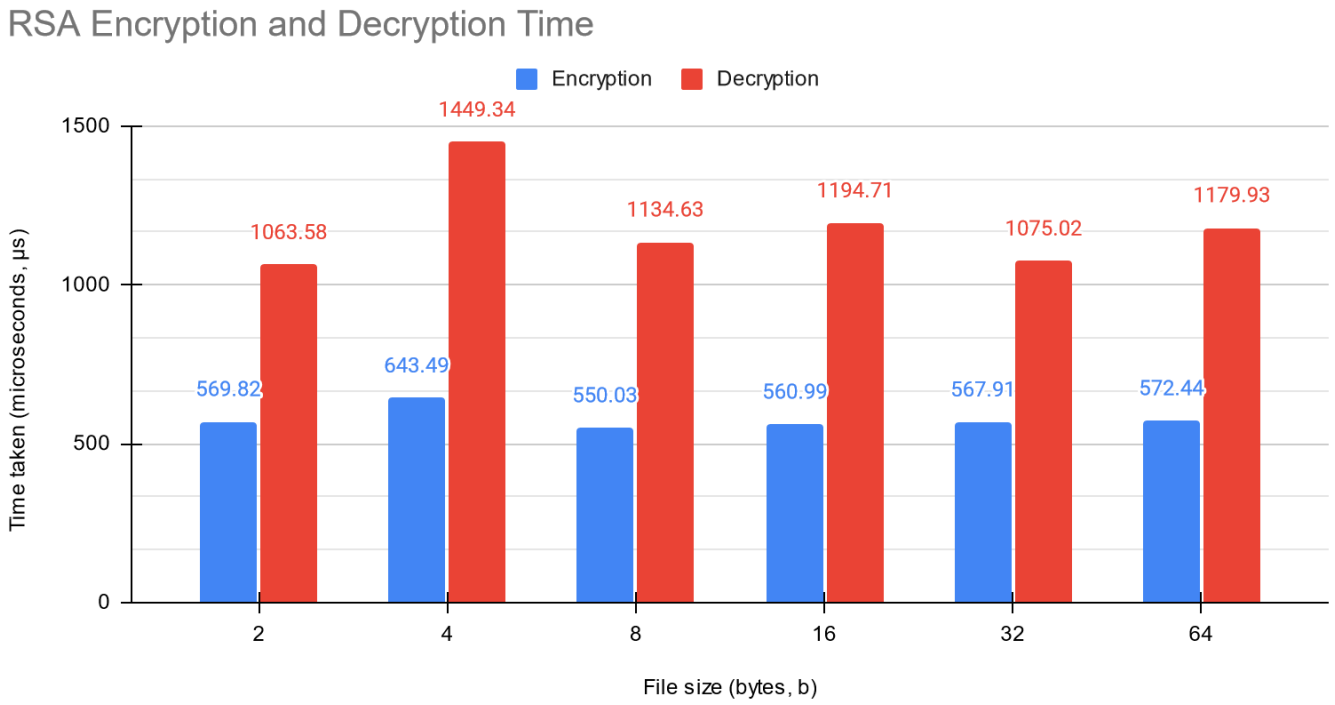

Figure 2.3: RSA Encryption and Decryption Time.

SHA-1 Digestion Generation Time

Figure 2.4 below is the graph of the SHA-1 digestion generation time.

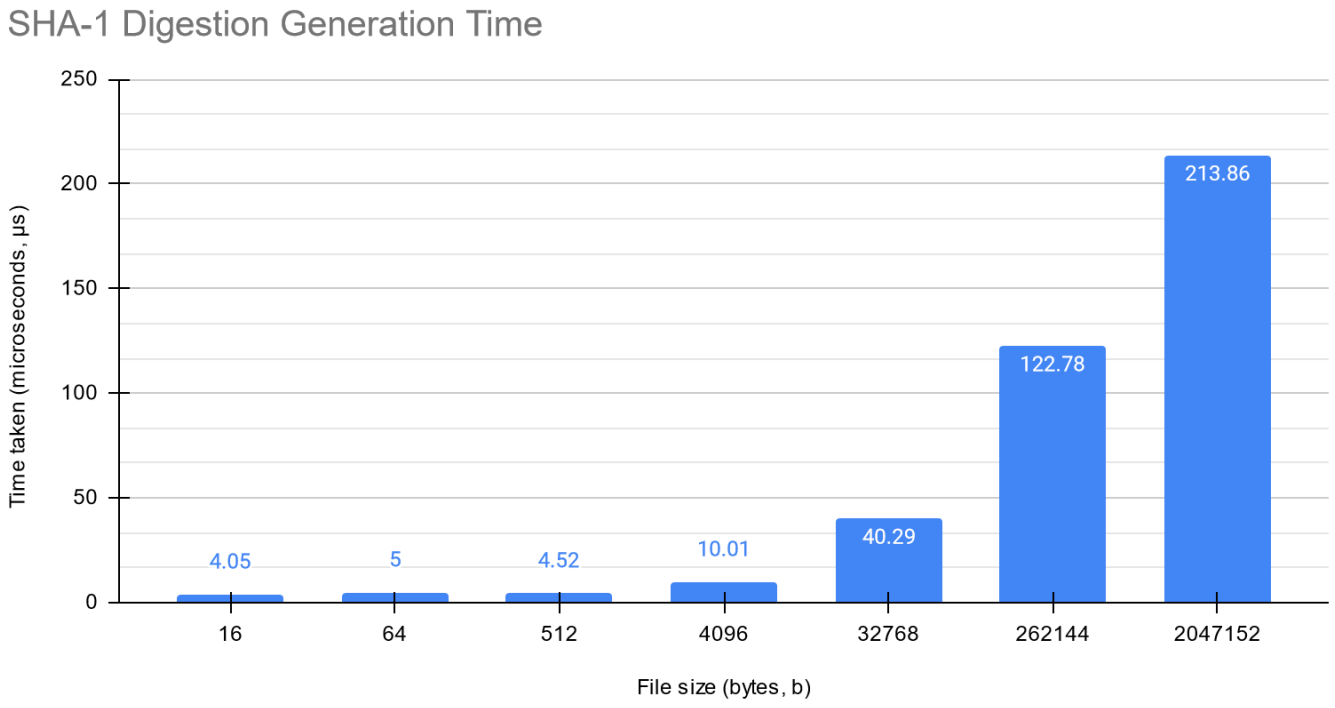

Figure 2.4: SHA-1 Digestion Generation Time.

HMAC Signature Generation Time

Figure 2.5 below is the graph of the HMAC signature generation time.

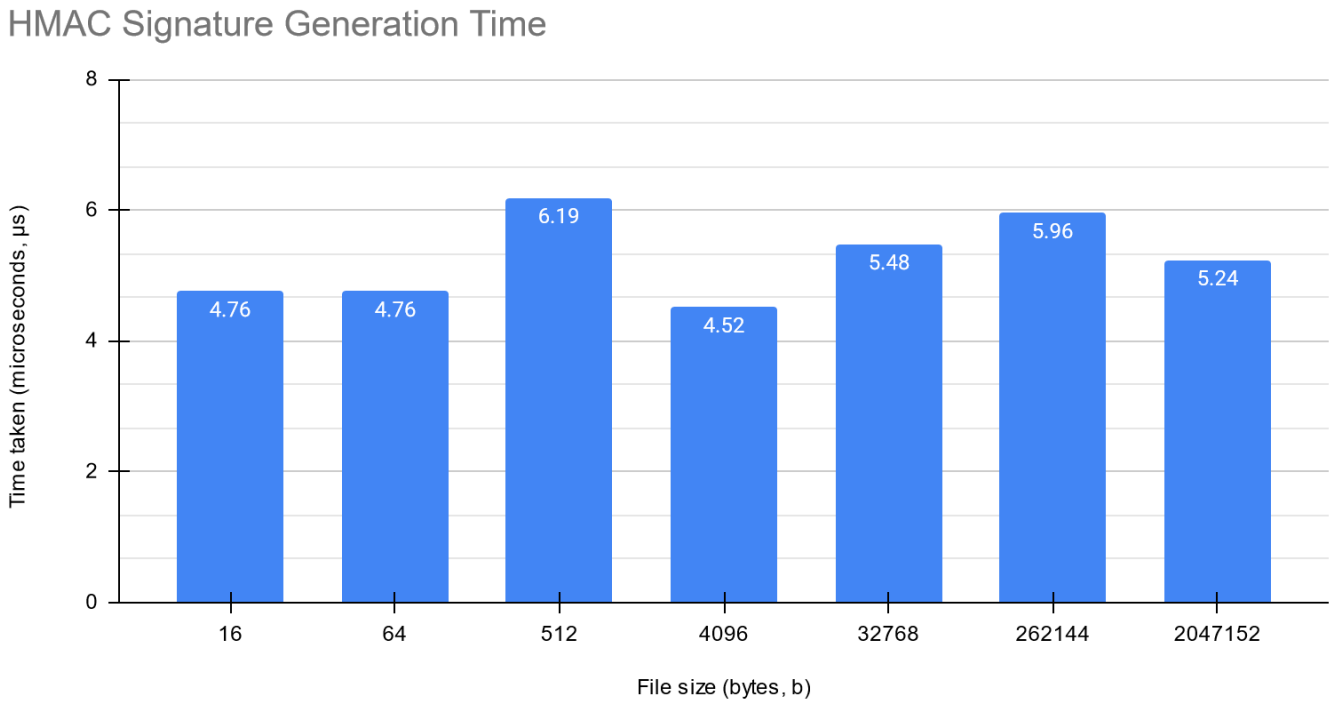

Figure 2.5: HMAC Signature Generation Time.

Discussion

Comparison between DES and AES

From the graphs, it is clear that DES is comparatively slower than AES. As observed, the encryption process of file size 2047152 bytes for DES was five times slower than AES. This is because DES uses a Feistel network with permutation encryption steps, whereas AES uses substitution permutation to encrypt.

Comparison between DES and RSA

Since the test files were not all the same between DES and RSA, file sizes 16 bytes and 64 bytes will be used to make time comparisons.

DES took 131.845 microseconds to encrypt a 16-byte file, whereas RSA took 560.99 microseconds.

DES took 156.64 microseconds to encrypt a 64-byte file, whereas RSA took 572.44 microseconds.

Thus DES is faster than RSA as DES is a symmetric key algorithm whereas RSA is an asymmetric key algorithm.

Comparison between DES and SHA-1

DES is comparatively slower than SHA-1 digest. When SHA-1 was used to encrypt the file of size 2047152 bytes, it took 213.86 microseconds, compared to DES, which took 1784 microseconds. This is because SHA-1 is a hash digest algorithm, which is generally faster than symmetric key algorithms.

Comparison between HMAC and SHA-1

In my test results, HMAC was slower than SHA-1 for files less than 4096 bytes but was considerably faster than SHA-1 for large file sizes up to 2047152 bytes. This might be because the padding is required if the input string is smaller than the block size, thus would increase time consumption.

Comparison between RSA Encryption/Decryption Time

Throughout my tests, RSA was considerably faster at encrypting than decrypting, which was twice as slow. This is because RSA decryption relies on prime number decomposition. RSA encryption is quickly done with prime numbers, but decrypting requires decomposing extremely large prime numbers.