Cryptography - Spring 2024
Lamport's One-Time Signature

Nada Ismail 202001387
Sama Yousef 202000819

Communications and Information Engineering
Zewail City of Science and Technology

May 23, 2024

Contents

1 Objective I: Implementing The Algorithm

1.1 GenerateKey

This code will create a key pair, a public key (pkey) by hashing SHA-256 skey items and a secret key (skey) within it. The aim is to ensure that the pkey may be made shared to anyone.

```
def GenerateKey(key_length):
    skey = [[0] * key_length for _ in range(2)] #containing two sublists 256 '1' and '0'
    pkey = [[0] * key_length for _ in range(2)]
    for i in range(len(skey)):
        for j in range(len(skey[i])):
            skey[i][j] = bin(secrets.randbits(key_length))[2:]
    for i in range(len(pkey)):
        for j in range(len(pkey[i])):
          pkey[i][j] = bin(int(hashlib.sha256(skey[i][j].encode()).hexdigest(), 16))
    keypair = [skey, pkey]
    return keypair
```

1.2 Sign

The following code calculates the digital signature of text using a secret key. First, the function converts the text message into a 256-binary digit representation by hashing with SHA-256. After that, it uses the secret key to generate an applicable signature for each bit of the binary hash. That way the message uniqueness is guaranteed since getting a message from its hash is hard.

```
def Sign(m, private_key):
    l=len(private_key[0])
    mesgh = int(hashlib.sha256(m.encode()).hexdigest(), 16)
    mesgh = bin(mesgh)[2:]
    sign = []
    for j, i in enumerate(mesgh):
      j=int(j)
      i=int(i)
      sign.append(private_key[i][j])
    return sign
```

1.3 Verify

A public key and a signature are used to verify whether a message is authentic by the following code. First we hash the message using SHA-256, then we convert the hash to binary and then check each bit against the public key. If any discrepancies are found during this verification process, meaning they don't fit together properly or match up at all this will be a failed verification attempt.

```
def Verify(m,public_key, signature):
    mhash = int(hashlib.sha256(m.encode()).hexdigest(), 16)
    mhash = bin(mhash)[2:]
    for i in range(len(signature)):
        j=int(mhash[i])
        verify = bin(int(hashlib.sha256(str(signature[i]).encode()).hexdigest(), 16))
        if public_key[j][i] != verify:
            return False
    return True
```

2 Objective II: Comparison with DSA

Please note that the time taken for the key generation in the DSA algorithm is so high that we had to remove it to see the other results.

2.1 Time Comparison of different file sizes by Signature Algorithms

Given that the time for DSA key generation so high we know that DSA algorithm takes longer, but if disregard this the lamport algorithm takes longer time.

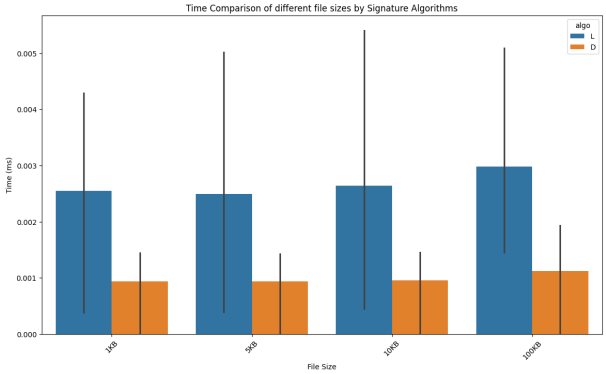

Figure 1: file sizes vs Signature Algorithms

2.2 Time Comparison of Operations by File Size

key generation and verification is not affected by file size but signing is takes longer for bigger files

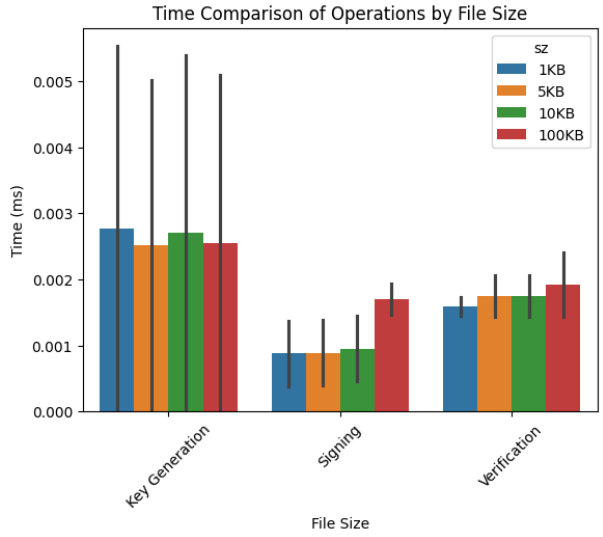

Figure 2: Operations vs File Size

2.3 Time Comparison of Operations by Algorithm

keep in mind that we removed the the key generation time for DSA from graphs. The time taken for key generation in much higher in DSA than Lamport. The time taken for signing in higher in DSA than Lamport. The time taken for verification in lower in DSA than Lamport.

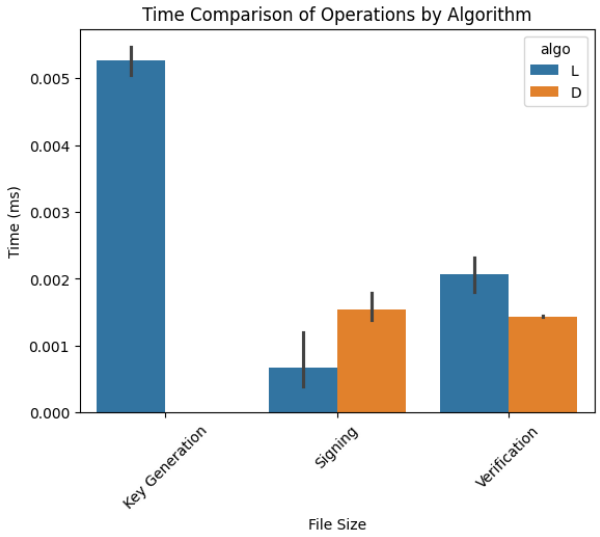

Figure 3: Operations vs Algorithm

2.4 Complete graph

This is a complete graph with all the values except DSA Key generation.

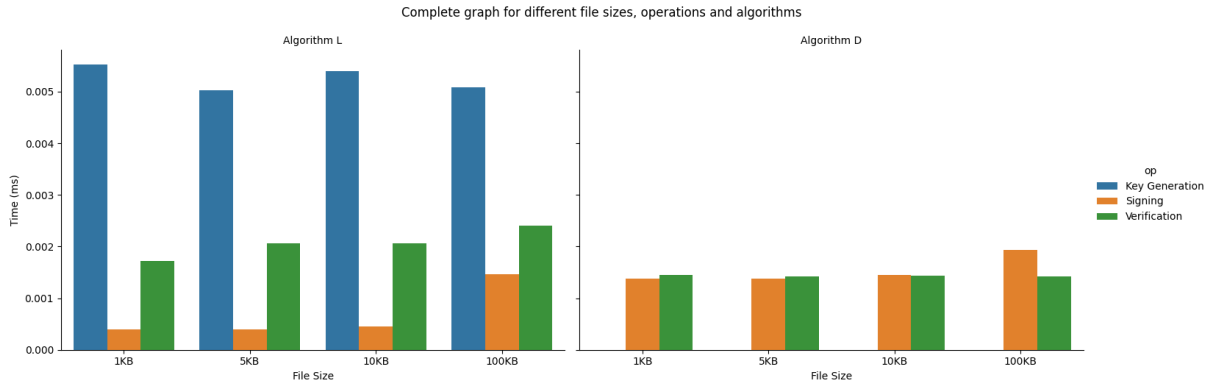

Figure 4: Complete Graph

3 Bonus: Signature Forging Implementation

3.1 Sign

The sign function of forging message is different because the message is not hashed but only encoded which make forging easier then there is loop that loop over the encoded message and for each element checked is is it 1 or 0 then pick the corresponding part from the private key

```
def Sign(message, skey):
    l=len(skey[0])
    mesgh = message.encode('utf-8')
    mesgh=bin(int.from_bytes(mesgh, byteorder='big'))[2:]
    sign = []
    for j, i in enumerate(mesgh):
        j = int(j)
        i = int(i)

        # Check if the index 'i' is within the range of skey
        if i < len(skey) and j < len(skey[i]):
            sign.append(skey[i][j])

    return sign
```

3.2 verification

The same as sign function the only different thing in the verification function is the dealing with the message, as the message is only encoded without hashing

```
def verification(message, pkey, signature):
    mhash = bin(int.from_bytes(message.encode('utf-8'), byteorder='big'))[2:]

    for i in range(len(signature)):

        j=int(mhash[i])

        verify = bin(int(hashlib.sha256(str(signature[i]).encode()).hexdigest(), 16))

        if pkey[j][i] != verify:
            return False
    return True
```

3.3 designgen

the use of this function is to get forged key through collecting number of signed messages then encode them after looping over them do another loop over each message to collect the parts of secret key

```
def designgen(messages, signatures, l):

    design = [[0] * l for _ in range(2)]
    for k,st in enumerate(messages):
      mesgh = st.encode('utf-8')
      mesgh=bin(int.from_bytes(mesgh, byteorder='big'))[2:]
      for j, i in enumerate(mesgh):
          j = int(j)
          i = int(i)
          # Check if the index 'i' is within the range of skey

          if i < 2 and j < l:

              design[i][j]= signatures[k][j]

    return design
```

4 appendix

Table 1: Time Taken by each Operation

Algorithm	File Size	Operation	Time(sec)
Lamport	1KB	Key Generation	0.00552
Lamport	1KB	Signature	0.00038
Lamport	1KB	Verification	0.00172
Lamport	5KB	Key Generation	0.00502
Lamport	5KB	Signature	0.00039
Lamport	5KB	Verification	0.00206
Lamport	10KB	Key Generation	0.00540
Lamport	10KB	Signature	0.00206
Lamport	10KB	Verification	0.00508
Lamport	100KB	Key Generation	0.00145
Lamport	100KB	Signature	0.00240
Lamport	100KB	Verification	7.99481
DSA	1KB	Key Generation	0.00137
DSA	1KB	Signature	0.00144
DSA	1KB	Verification	0.95192
DSA	5KB	Key Generation	0.00138
DSA	5KB	Signature	0.00142
DSA	5KB	Verification	8.30094
DSA	10KB	Key Generation	0.00145
DSA	10KB	Signature	0.00143
DSA	10KB	Verification	0.00143
DSA	100KB	Key Generation	1.39252
DSA	100KB	Signature	0.00192
DSA	100KB	Verification	0.00142