DAT510 - Assignment 2

Frøydis Jørgensen

October 29, 2020

Abstract

A one-paragraph summary of the entire assignment - your choices of cryptographic primitives and their parameters, procedure, test results, and analysis.

Introduction

A description of the scientific background for your project, including previous work that your project builds on. (Remember to cite your sources!) The final sentence (analogous to the thesis statement in a term paper) is the objective of your experiment.

Design and Implementation

I have created both a digital signature scenario in the main with a predefined message where we can validate the signature on the message and two applications where Alice and Bob can send messages to each other with a signature and validate that the messages are correct. Both the main and the applications uses the same methods and I will therefore first explain all the methods before I go into the applications.

When implementing RSA, I think the most crucial thing is to find some good keys. We need two random large primes, **p** and **q**. By multiplying **p** and **q** we get **n**. The keys are generated as shown in Figure 1. To decide **p** and **q** I use a package called Crypto and uses their method *getPrime*, and to get a random primes I also uses Crypto's function *get_random_bytes*. Although the chances for p and q to be the same is very small, we should make sure that's not the case. I take in the bit length as an input to the function. There are major runtime differences that depend on the bit length, but I will get back to that.

```python
def generatePrimes(bitLength):
    p, q = 0, 0
    # Todays standard is 1024 bits and larger
    while p == q:
        p = Crypto.Util.number.getPrime(bitLength, randfunc=get_random_bytes)
        q = Crypto.Util.number.getPrime(bitLength, randfunc=get_random_bytes)
    n = p*q
    return p, q, n
```

Figure 1: Generates two large primes and multiplies them

The next step is to find the totient of **n**, ϕ phi. As shown in Figure 2 that's an easy calculation using **p** and **q**.

```python
def totient(p, q):
    return (p-1) * (q-1)
```

Figure 2: Calculated the totient of n from p and q

Together with **n** we have another public exponent **e**. This is a prime number that should be the greatest common divisor of 1 with $\phi(n)$. But it is often chosen to be the Fermat number 65537. That is the largest known prime in the form of $2^{2^n} + 1, (n = 4)$. It is large enough to avoid attacks and quick on binary calculations. But we should always check that e is not a factor of the totient ϕ. If that is the case, we choose the next largest prime number, [1]. As shown in Figure 3 I use *gcd* imported from the package math to calculate the greatest common divisor.

```python
# e is the public exponent, Fn = 2^2^n + 1 med n = 4
e = 65537
# e can not be a factor of phi, it is unlikely, but must be checked
while (phi % e == 0):
    e = gcd(2, phi-1)
```

Figure 3: e is F4, 65537, if it is a factor of the totient use the gcd

Using the public exponent **e** and ϕ we can create the private exponent **d**. The private exponent **d** is the inverse of the public exponent with respect to

the totient ϕ. To calculate the inverse you can use the *Extended Euclidean Algorithm*. I have taken an implementation of that from geeksforgeeks, ref [2]. This method works when **e** and ϕ is coprimes, which we already know they are. The function to find **d** is shown in Figure 4

```
def modInverse(a, m) :
    m0 = m
    y = 0
    x = 1
    if (m == 1) :
        return 0
    while (a > 1) :
        q = a // m
        t = m
        m = a % m
        a = t
        t = y
        y = x - q * y
        x = t
    if (x < 0) :
        x = x + m0
    return x
    # This code is contributed by Nikita tiwari.
```

Figure 4: Finding the inverse. Function from GeeksforGeeks

Now that we have the public keypair of **e** and **n** and the private keypair of **d** and **n**, we can start creating the digital signature on the message. Before we can run the RSA encryption calculation to create the signature we have to hash the original data (the message). I have imported *hashlib* and used SHA256 for hashing the message. So that means that no matter how long the message is, we will get a 256 bit block. Using the hash we can now create the signature by using RSA's encryption function. We will use our private key to create the signature as shown here in the mathematical expression.

$$hash^d \ mod \ n$$

In the code, as shown in Figure 5, I use the function *pow* which does the same as the mathematical expression. As you can also see in the figure is that I convert the hash into an integer, and throughout the program I only work with integers instead of bits.

```
hashedMessage = int.from_bytes(hashlib.sha256(message).digest(), byteorder='big')
signature = pow(hashedMessage, d, n)
```

Figure 5: Hashing the message and create signature

Now that we have the message and the digital signature we can send it and verify that the message is correct according to the signature and it have not been tampered by some man in the middle. To verify the signature we take the message and creates a hash using the same hashing method SHA256. This hash should match what we decrypt from the signature. To decrypt the signature we do the same mathematical operation as when it was created but instead of the message, we use the signature, and instead of the private key **d** we use the public key **e**. This works because both **e** and **n** is public keys. So in this case everyone can verify that the message was sent from f.ex Alice. Normally we would use some encryption on top of that, since the message now is sent in plaintext. As you can see in Figure 6, we check that the hash we got from the message is the same that we got after decrypting the signature. If they are the same, we have verified that the message from Alice is the same as at the point Alice signed it and sent it.

```
def verifySignature(signature, message, e, n):
    ourHash = int.from_bytes(hashlib.sha256(message).digest(), byteorder='big')
    hashFromSignature = pow(signature, e, n)
    print('The hash the signature is created from: ', hashFromSignature)
    print('Our recreated hash from the message: ', ourHash)
    if ourHash == hashFromSignature:
        print('The signature is verified')
        return True
    else:
        print('Something went wrong, maybe there is a man in the middle? Do not trust this')
        return False
```

Figure 6: Verifying that the signature matches the message

In the application we have two url's. The start url, where we can send a message and sign it. And /getMsg where we can see the message from the other part and see if the signature is correct. Opening the application will automatically run the generation of the public and private keys, in the same way as explained previously.

When a message is written and it's clicked send the signature will be created as shown in Figure 7. The fetching point for people that want to

receive the message is /sentMsg. And the message is sent as a dictionary including the message, the signature, **e** and **n**.

```python
@app.route('/', methods=['POST', 'GET'])
def sendMessage():
    text = request.form.get('message')
    if text:
        global signature
        global message
        # Hash the message and use RSA encryption to create a digital signature
        hashedMessage = int.from_bytes(hashlib.sha256(text.encode()).digest(), byteorder='big')
        signature = pow(hashedMessage, d, n)
        message = text
    return render_template('sendMsg.html', title="Send message")
```

Figure 7: Running a template with input field and submit button to send message with a signature

To fetch messages you go to /getMsg. If there is no message it will be displayed "There is no message". If there is a message we have to verify the signature. We use the *verifySignature* method as shown in Figure 6. If the message and the signature corresponds the message will be shown with a note that the signature is correct. If the signature is wrong there is a warning that someone might have tampered the message.

```python
@app.route('/getMsg', methods=['GET'])
def getMessage():
    fullMessage = {}
    fullMessage = requests.get("http://127.0.0.1:3000/sentMsg")
    messageObject = json.loads(fullMessage.content.decode())
    print(messageObject)
    if fullMessage.ok:
        if messageObject['message'] == '' or messageObject['signature'] == 0:
            return "There is no message."
        # If there is a message, check that the signature matches to the message
        if verifySignature(messageObject['signature'], messageObject['message'].encode(), messageObject['e'], messageObject['n']):
            return '''The message from Bob is: <h3>{}</h3> <div>This message has Bob's
            digital signature</div>'''.format(messageObject['message'])
        return '''The message from Bob is: <h3>{}</h3> <div> But something went wrong.
        There might be a man in the middle here.</div>'''.format(messageObject['message'])
    else:
        return "No response"
```

Figure 8: Fetching the message and verifying the signature

The applications has a very very similar set up as in assignment 2, and one is run on port 3000 and one on port 5000.

Test results

Running the main applications will print out the information we get while doing a digital signature and verifying it. As you can see in Figure 9, the first thing printed is the public and private keys. Then we can see the message sent from Alice to Bob and the signature sent with it. The recreated hash is created by Bob to see that it matches the hash we got by encrypting Alice's signature. And in this case they are the same and therefore the signature is verified.

```
The public key pair is e = 65537 and n = 190827891933115370632542459388764730348402087319844
725817597969933732644074757291104856903530586603313796580233762418387936196691048265101320499
795862496012020816634513950968586665776785369160518755581300728990246540706213403149498584819
942925235132730020388685703039741922989616169800052593486281049226860015522168285270799975746
596910495136207802025297216685814058314025538747876772972722898358444696245731472634887100747
445817539290910862810854061016476243628416108451527932985993376039195967018368934225204103623
598866302792100580907564401103688642348721697025598322015225453555894843765457959866791193143
The private key pair is d = 103402354369726918258431844878676367611157894394042059863471708974
821354907362677712088694685166881525989829280751540504111019762407877380758669694602906281312
523967229578092173832334026049438174288448482584669543946704206159121910447437998587553841265290
612148928096626863169155144802816637117505965809573436792887346865458111488049877289293022442
282357086555329582974878643865559849830213154615242419162684721677605730156382163061063334353
637579360253712521143053257529364310090333624992691659693647169178062009610700060816869964484
722626674688997795883348911424970566748332879579520054968130492409124980583264704 1 and n = 1908
278919331153706325424593887647303484020873198449672581759796993373264407475729110485690353058
603313796580233762418387936196691048265101320499479586249601202081663451395096858666577678536
916051875558130072899024654070621340314949858481959429252351327300203886857030397419229896161
698005259348628104922686001552216828527079997574675969104951362078020252972166858140583140255
387478767729727228983584446962457314726348871007475445817539290910862810854061016476243628416
108451527932985993376039195967018368934225204103623159886630279210058090756440110368864234872
169702559832201522545355589484376545795986679119314331
The message is:  b'This is a message from Alice to Bob'
The hash the signature is created from:  552534748110604274816924181575477296092375192478768487
2344346694818674384522 1
Our recreated hash from the message:  55253474811060427481692418157547729609237519247876848723
44346694818674384522 1
The signature is verified
```

Figure 9: The print statements from the RSA.py main

Using this on the two applications will look as shown in Figure 10, when the signature is verified. If the signature does not match the message, the note under the message will tell you that.

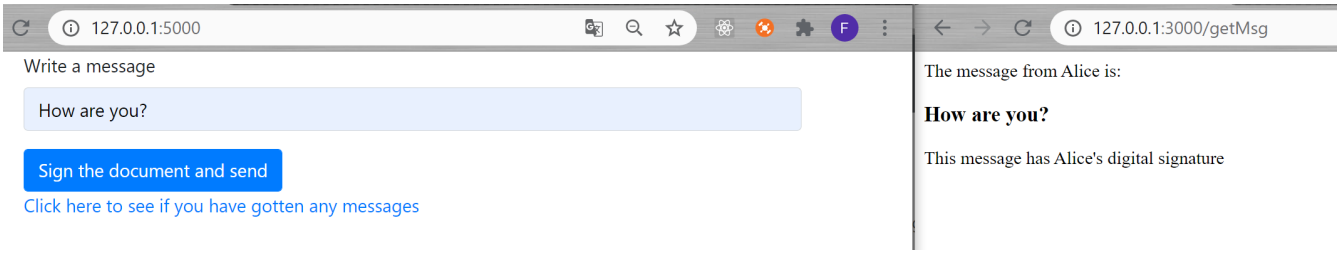

Figure 10: Alice sends a message and sign it. Bob receives it and verifies the signature

It seems like RSA usually uses key of 2048 bits. Creating and working with that big numbers goes slow in python and to beware of computational problems I have used the package *timeit* to find out the different execution times depending on the bit length. You can see the result in the table here.

Bit length	Execution time
128	0.060
256	0.186
512	0.738
1024	4.894
2048	35.01

As you can see in the table, the execution time increases exponentially as we increases the bit length. But the important thing here is that if the product of the

Discussion

Your analysis of what your testing results mean, and your analysis.

I could have, and was thinking about splitting up the message, to get a lower runtime, but did not see that as a must in this case, since the messages was not that big anyway. I tried sending a string with 5000 characters, and it was slower, but nothing crucial.

Conclusion

A short paragraph that restates the objective from your introduction and relates it to your results and discussion, and describes any future improve-

ments that you would recommend. Works Cited A bibliography of all of the sources you got information from in your report.

References

[1] 65,537, *Wikipedia - The number 65537*, https://en.wikipedia.org/wiki/65,537.

[2] GeeksforGeeks, *Modular inverse*, https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/.