# DAT510 - Assignment 3

Eirik Sakariassen

November 3, 2020

## Contents

## 1 Abstract

We will in this assignment implement a digital signature scheme DSS[1] such that two communicating parties can sign and validate the data that is being transferred between them. The signature scheme consists of several important building blocks; a key exchange protocol, a hash function and a per-user private and public key. We will in detail explain how these blocks can be implemented and how they are put together to form a system that can sign and validate data. We will in the end develop a proof-of-concept service named CryptoBank where a user can transfer money to other bank accounts. The service enables a user to sign its transaction data which then will be stored securely on CryptoBanks servers. A fictive VISA server can then fetch and validate transaction data from CryptoBank before transferring it to the correct bank account.

## 2 Introduction

### 2.1 Digital Signature Standard

The Digital Signature Standard (DSS) specifies algorithms to be used for generating digital signatures[2]. Among these algorithms is the Digital Signature Algorithm (DSA)[3] which was first published by NIST in 1991, and in 1994 adopted as Federal Information Processing Standard FIPS 186. The newest specifications of this algorithm was issued in July 2013 and is named FIPS 186-4. Digital signatures can be used to authenticate the ID of a user who signed a document or to detect modifications to data.

The signature is a string of bits that is computed by a set of rules and parameters that allows us to verify the integrity of the data and ID of the signatory. A per-user private key is used to generate the digital signature and a per-user public key is used to verify it. The public key can be known by the public and anyone can verify the signature by using it.

An appropriate hash function must be used while generating a signature. The hash function generates a message digest which is a condensed version of the data that will be used as input in the DSA. The verifying part can then use the same hash function together with the public key to verify it.

### 2.2 Secure Hash Standard

The Secure Hash Standard (SHS) specifies hash algorithms to be used to generate digests of messages[4]. Secure Hash Algorithms (SHA) are typically used together with other cryptographic algorithms. One example is digital signature algorithms as described in section 2.1. The hash algorithms is secure because it is computationally infeasible to guess the message corresponding to a message digest. It is also computationally infeasible to find two messages that produces the same message digest. Even just a minor change to the original message will result in a totally different message digest.

## 2.3 PKI certificates

a) **What are the different types of SSL's and how different they are in aspect of security? Why ?** There are three types of TLS/SSL certificates[1]:

1. **Domain Validation** (DV): Used to ensure that the part requesting a certificate is authorized to do so for the given domain. They will be checked against a domain registry that can prove the ownership of the site. Domain Validated certificates does however not identify organizational information, so client users cannot validate whether it is legitimate via the certificate alone. Among the three types of TLS/SSL certificates, DV contains the lowest level of authentication used to validate a certificate.

2. **Organization Validation** (OV): More credible certificate than DV. The Certification authority (CA) will check organizations against business registry databases that is being hosted by governments. The certificate will activate https and client users can be assured that the site is being run by a legitimate company.

3. **Extended Validation** (EV): Offers the highest level of authentication by adding extra validation steps. Organizations must provide documents to the CA so that they can confirm and prove its legal, rights to the domain and physical and operational existence. The certificate is often being used by big organizations, for instance within ecommerce, as these sites must gain customer confidence. Websites with EV certificate triggers a green bar in the URL section in several browsers.

b) **Research about the the Certificate Authority Security concerns and explain.** A Certificate Authority (CA) is an organization that validates identities of online entities and issues their digital certificates[5]. Digital certificates provides authentication, encryption and integrity.

There exists several systematic problems for TLS/SSL and the CA model. Many of these problems are discussed in research articles such as [6], but we have chosen one particular problem which is being discussed at the security blog "securityboulevard"[7].

Certificate Authorities can be targeted by cybercriminals who wants to obtain verfified and signed certificates. Attackers can use techniques like social engineering[8] to accomplish this, and then sell it on various black markets. Whoever buys the stolen certificate can then use it for illegal purposes. It is common to use it to sign malicious files such that they can bypass security systems of different organizations. The reason they bypass these systems is simply because the signed certificate is recognized by the CA. Files without such a certificate looks more suspicious and are more likely to be detected.

Public CAs performs strict checks against government databases etc. before issuing certificates. Private CAs however do not have to follow these industry regulations. Private vendors are the clear choice for network authentication, but one should ensure

[1]https://www.digicert.com/blog/how-to-choose-the-right-type-of-tls-ssl-certificate/

that they follow industry trends and regularly updates based on security patches. If not, the network may be at big risk.

c) **How does browsers identify secure CA's From another CA's and how is it measured?**

Browsers do only trust publicly trusted PKIs that conforms to RFC 5280[9] and is in X.509 v3 format[2]. This format allows a certificate to include additional data such as policy information or constraints for usage. Additional data extensions are either critical or non-critical. Critical extenstions are required to be processed and validated by the browser.

CAs signs all issued certificates with a private key that proves that it came from the CA and was not modified later on. Ownership of their signing key is established by holding a self-issued certificated for a corresponding public key. This self-issued certificate is reffered to as the *root*. A *root* is tightly controlled and used to issue *intermediate* certificates that can be used to issue customers certificates. Browsers have a list of trusted roots built in.

A browser will obtain a sequence of certificates (*certification path*) where each of them have signed the next item, connecting the CA's root (*trust anchor*) to the servers certificate (*leaf certificate*). Browser will often consider mulitple certification paths before a valid one is found for a given certificate. They validate the paths by using information inside the certificates. It must be cryptographically proven that from the certificate signed by the trust anchor, each next certificate in the chain was issued by the previous certificate's private key.

## 2.4 CryptoBank

CryptoBank is a proof-of-concept service where a user securely can sign its transaction data before it is transmitted to another bank account through a third party. The third party will verify the signed document based on the digital signature. Figure 1 illustrates the system design. Let's say Alice wants to send money to Bob. Alice will then login to her CryptoBank account and create a new transaction to Bob. The transaction data is sent and stored at CryptoBanks server once Alice signs the transaction. A third-party service such as VISA will then periodically through the day fetch signed transactions from CryptoBanks servers. VISA servers will verify every detail for each transaction based on the digital signature. If someone modifies Alice's transaction, VISA will detect and stop it. VISA can send the transaction to Bob's bank if it is verified.

[2]https://tools.ietf.org/html/rfc5280#section-3.1

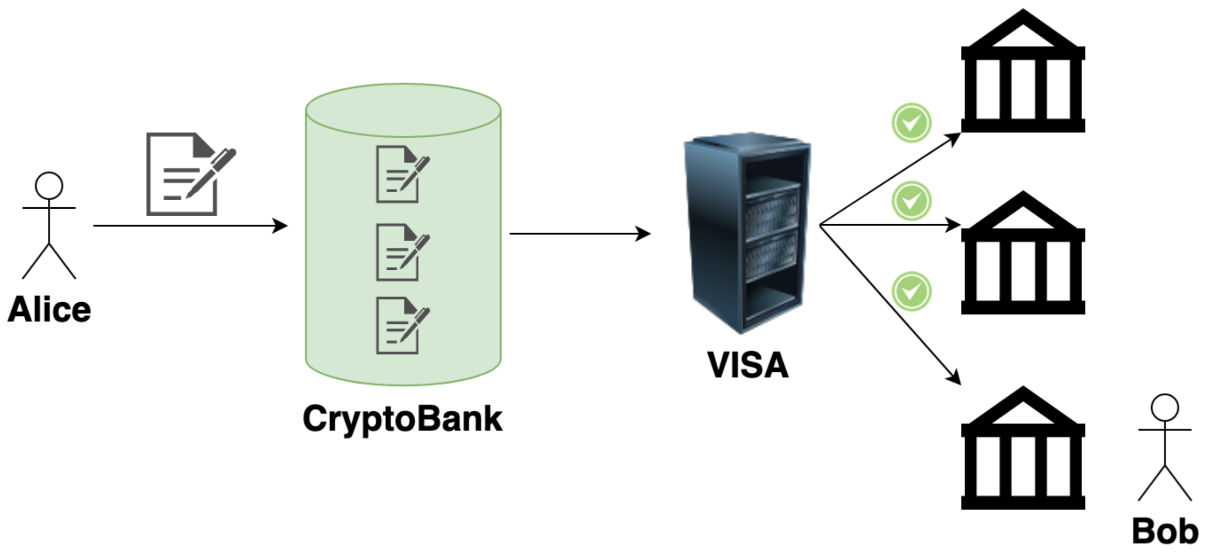


Figure 1: CryptoBank system

## 3 Design and Implementation

### 3.1 Digital Signature Algorithm

#### 3.1.1 Key Generation

The first step of the algorithm is to generate the shared public key components ($p$,$q$,$g$). This set of parameters will be known to the group that communicates. We choose N=160 and generates a 160-bit prime number $q$ such that it is a prime divisor of ($p$-1), where $2^{N-1} < q < 2^N$. We then pick $p$ where $2^{L-1} < p < 2^L$ for 512 $<= L <=$ 1024. L must be a multiple of 64 and ($p$-1) must divide $q$. Finally, $g = h^{(p-1)/q} mod p$ where $h$ is any integer $1 < h < (p-1)$ such that $g$ is greater than 1. Code block 1 shows the implementation of this. The method utilizes a python library named sympy[3] to validate and generate prime numbers in the format we described above. If some of the criterions aren't met, it will raise error messages and return.

```python
def gen_pubkeys(L, N=160):
    if L not in range(512,1025) or L%64 != 0:
        raise ValueError("Choose L: 512 <= L <= 1024 and L is a multiple of 64")
    p, g = 1, 1
    while not sympy.isprime(p):
        m = random.randrange(2**(L-N-1), 2**(L-N))
```

[3]https://www.sympy.org/en/index.html

```
        q = sympy.randprime(2**(N-1), 2**N)
        p = m * q + 1
    assert (p-1)%q == 0
    while g <= 1:
        h = random.randrange(1, p-1)
        g =  pow(h, (p-1)//q, p)
    return p,q,g
```

Listing 1: Generation of public keys

Once $(p,q,g)$ are generated, the users choose a random private key $x < q$ and computes a public key $y = g^x mod p$. A method named *gen_userkeys* takes care of this part and is shown in code block 2.

```
def gen_userkeys(p, q, g):
    x = random.randrange(0, q)
    y = pow(g, x, p)
    return x, y
```

Listing 2: Generation of user keys

### 3.1.2 Signing

For each signature of a message M, the sender needs to generate a new random signature key $k$ where $k < q$. The signature of M consists of a pair of numbers $r$ and $s$ which are computed by the signature key, user's private key, public key components and the hash of the message H(M). We picked SHA256 as the hashing algorithm since it is secure and supported by DSS. We compute $r$ and $s$ in the following way:

$$r = (g^k mod p) mod q \tag{1}$$

$$s = [k^{-1}(H(M) + xr)] mod q \tag{2}$$

Implementation of the signature creation is shown in code block 3. Calculating the modular inverse of k in equation 2 naively did not work in our case since we worked with big integers. We used extended Euclidean algorithm[4] to fix this computational problem. The method *mod_inv* is called in line 5 in 3.

```
1 def sign_message(message, p, q, g, x):
2     k = random.randrange(0, q)
3     r = pow(g, k, p)%q
4     HM = hashlib.sha256(message.encode('utf-8')).hexdigest()
5     k_inv = mod_inv(k, q)
6     s = ((k_inv*(int(HM,16) + x*r))%q
7     return r, s, message
```

Listing 3: DSA signature

[4]https://www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/

### 3.1.3 Verifying

We denote received versions of M, $r$ and $s$ as M, $r$' and $s$', respectively. The receiving part will verify the signature by computing a quantity $v$ that should match $r$'. Computations in the verification part can be expressed as:

$$w = s^{-1} mod q \tag{3}$$

$$u1 = [H(M)w] mod q \tag{4}$$

$$u2 = (rw) mod q \tag{5}$$

$$v = [(g^{u1} y^{u2}) mod p] mod q \tag{6}$$

H(M) is the same hash function as in the signature creation; SHA256. We have to use extended Euclidean algorithm to compute equation 3. Code implementation is shown in block 4.

```python
def verify_message(message, p, q, g, r, s, y):
    w = mod_inv(s,q)
    HM = hashlib.sha256(message.encode('utf-8')).hexdigest()
    u1 = (int(HM, 16)*w)%q
    u2 = (r*w)%q
    _gu1 = pow(g, u1, p)
    _yu2 = pow(y, u2, p)
    v = ((_gu1*_yu2)%p)%q
    if v == r:
        return True
    return False
```

Listing 4: DSA verification

## 3.2 SHA256

We tried to implement the SHA256 algorithm from scratch by following the guidelines described in [4]. We encountered some bugs in the later steps of the algorithm, so we decided to use hashlib [5] in our DSA instead. We will however describe the most important steps of our implementation here.

## 3.3 Preprocessing

Preprocessing consisted of three steps:

1. Message padding
2. Parsing message into blocks
3. Setting initial hash value $H^0$

[5]https://docs.python.org/3/library/hashlib.html

### 3.3.1 Message Padding

We pad the message to ensure it is a multiple of 512. A message with a length of l bits will first be padded with a "1" followed by k zero bits, where k is the smallest non-negative solution to:

$$l + 1 + k = 448 mod 512 \tag{7}$$

We will then append a 64-bit block that is equal to the number l. We defined helper functions *pad_len* and *block_bits* which generated these paddings. Code block 5 shows how we performed this by first taking in a message (string), converting it to a bit string (based on ASCII) and then adding the paddings in the end. We ensure it is a multiple of 512 before returning.

```python
def pad_message(message):
    bitstring = ''.join([char_to_bitstring(c) for c in message])
    bitlen = len(bitstring)
    padding = pad_len(bitlen)
    len_bits = block_bits(bitlen)
    # Add padding and 64-bit message block
    bitstring = bitstring + padding + len_bits
    assert len(bitstring)%512==0
    return bitstring
```

Listing 5: SHA256 padding

### 3.3.2 Parse message into block

The padded message was parsed into N m-bit blocks. For SHA256, m=512, and each 512-bit block is then expressed as sixteen 32-bit blocks. The method *parse_block* in 6 converts a 512-bit block into the right format.

```python
def parse_block(bitstring):
    M = [bitstring[i:i + 32] for i in range(0, 512, 32)]
    assert len(M) == 16
    return M
```

Listing 6: SHA256 message parsing

### 3.3.3 Initial hash value

The original hash value $H^{(0)}$ consisted of eight 32-bit words in hex format. They were obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. We was given these values beforehand.

## 3.4 SHA256 Algorithm

The algorithm can be used to hash any message of length $0 <= l <= 2^{64}$ into a 256-bit message digest. The key parts of the algorithm is:

1. A message schedule of 64 32-bit words

2. Eight working variables of 32-bit each
3. A hash value of 8 32-bit words

We will not describe more details of this algorithm, but it can be found in [4]. The algorithm used integers, strings, hex digits etc. to run different types of operations on w-bit words or bit strings. A *word* is a w-bit string represented as a sequence of hex digits. We defined a BitString class which would handle all these convertions and helped us create and manipulate binary data.

A BitString object could be initiated from a list of bits, hex digits or a bitstring. We found this feature very helpful as for instance initial hash values was hex digits, but padded message was bitstring. Code block 7 shows the classmethod for BitString initialization from hex digits and code block 8 shows the classmethod for initialization from a bitstring. Both blocks contains an example in the button on how it can be used.

The BitString class contained methods that would represent the object in different formats; *to_int*, *to_string* and *to_hex*. BitString did also contain all necessary bitwise operations; *add*, *extend*, *unshift*, *rotate_right*, *shift_right*, *bitwise_xor*, *bitwise_and* and *bitwise_not*. The algorithm used this class for all bit manipulation.

```python
@classmethod
def from_hex(cls, hexstring, n):
    bits = bin(hexstring)[2:].rjust(n,'0')
    bits = [int(i) for i in bits]
    return cls(bits)
b = BitString.from_hex(h0, 32)
```

Listing 7: Initiate from hex digits

```python
@classmethod
def from_bitstring(cls, bitstring):
    bits = [int(i) for i in bitstring]
    return cls(bits)
b = BitString.from_bitstring(padded_message)
```

Listing 8: Initiate from bitstring

## 3.5 CryptoBank

The cryptobank system consisted of three files:

1. **app.py**: Flask application running a user interface where new transactions can be created and signed.
2. **cryptobank.py** CryptoBank server running and listening for third-party clients to request new transaction records
3. **visa.py** Third-party client service that periodically requests and verifies new transaction records from CryptoBank servers.

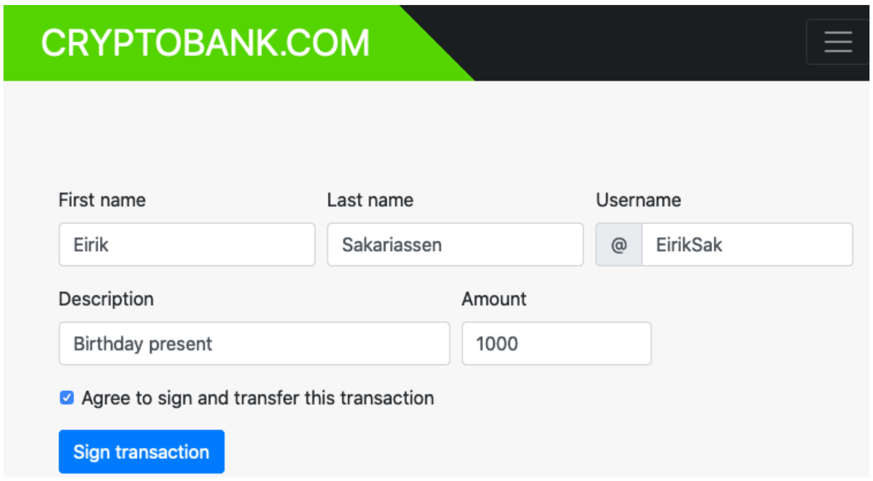
The user interfae contained simple Bootstrap 4 template parts[6] with a form field where you could specify information about the receiving end as well as extra transaction description and the amount you wish to send. The user would then have to agree to sign and transfer the transaction before it could sign it by clicking a button. Clicking the button would issue a POST request to the server on */sign* where a method on the flask server would handle the data.

The *sign* method generated public key components and private and public keys. It would then go through data from each form field and generate a signature using the DSA functionality from 3.1. A random hash string was used to create a unique transaction ID and we stored the signed transaction object under a transactions folder. Figure 2 shows the UI of the application. Listing 9 shows the transaction object that got created after the user signed the transaction in the UI figure. We can see that each form field has different $r$ and $s$ values (signatures). This was done so we could detect which exact parts of the transactions was modified in case of an attack.

```
{
    "firstname": {
        "value": "Eirik",
        "r": 777856560967354927322759927764236078701354341189,
        "s": 10687088348007477271163692769742942022552529262
    },
    "lastname": {
        "value": "Sakariassen",
        "r": 510463974266816073539515393974834288673248902554,
        "s": 191503133389151464963909691715824724886569223078
    },
    "username": {
        "value": "EirikSak",
        "r": 261587787392906974206679254036332129281532776604,
        "s": 675323445639989545907879840570173195666025777253
    },
    "description": {
        "value": "Birthday present",
        "r": 940811770806795214995206370900253551084071358033,
        "s": 179151498943126774628435820470192612779806077766
    },
    "amount": {
        "value": "1000",
        "r": 630558831439060542090180970591399022783773664702,
        "s": 651171098350288295225049203952605497184407422570
    },
    "public_key": 1002910404582948756786310613362610168218704006931..,
    "p": 12178488682831843235062573635122013493679476...,
    "q": 13382048771094285284210559276052905440581744...,
    "g": 17342527160526573781612378684577210600617902...
}
```

Listing 9: Transaction object

[6]https://bootsnipp.com/snippets/Q04ZX



| | | |
|---|---|---|
| First name | Last name | Username |
| Eirik | Sakariassen | @ EirikSak |
| Description | Amount | |
| Birthday present | 1000 | |

☑ Agree to sign and transfer this transaction

Figure 2: CryptoBank UI

We used pythons socket module[7] to create the CryptoBank and VISA servers. The VISA server would connect to the CryptoBank server at port 4000 and request transaction records. CryptoBank used a method named *get_transactions* to iterate over all generated transaction objects in the transactions folder. All available transaction objects would then be sent over to VISA who then would issue a verification method named *verify_transactions*. The method logged all transaction and field verifications in the command line interface and prompted whether or not they was verified. We used the DSA *verify_message* method from subsection 3.1.3 to do this.

## 4 Test Results

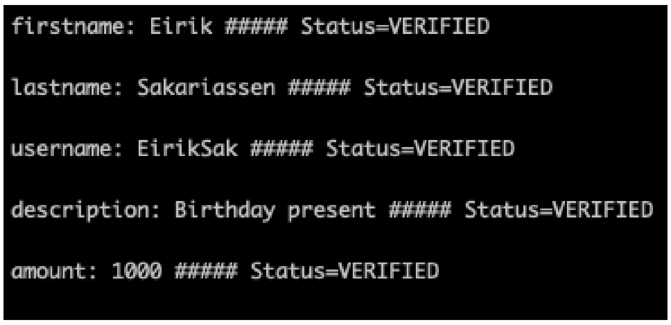
### 4.1 DSA Implementation

We created a test script *test_dsa* that would verify the implementation of the DSA. The test cases would generate keys, ask the user to write a message in the command line, and then sign it using DSA. The generated $r$ and $s$ values would be prompted to the user before it was used to verify the message. The program displayed whether or not it suceeded. We could run this test in two modes; *normal* and *attacker*. Normal mode was just as described above, while attacker mode would enable the user to step inbetween as an attacker and generate a fake message. The test would then check whether or not it got verified based on the signature of the orignal user. The implementations worked fine for all tests conducted, meaning it verified everything in user mode and did not verify in attacker mode.

We used the CryptoBank service to test our implementation in a more advanced fashion. We copied the transaction object listed in 9 and changed the amount field only.

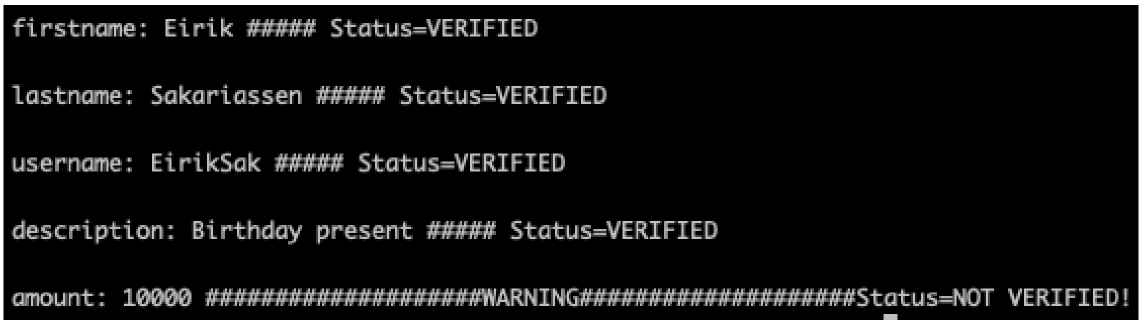
[7]https://docs.python.org/3/library/socket.html

We could then start the VISA server and fetch both the legit and modified transaction objects to check whether or not the DSA verification would detect it. The VISA server verified the original message and detected the modification of the amount field in the scamming transaction. Figure 3 shows logged output of the verification of the original version and figure 4 shows logged output on the modified version. We can see that the the amount had been changed from 1000 to 10000, something that VISA discovered through digital signature verification.



```
firstname: Eirik ##### Status=VERIFIED

lastname: Sakariassen ##### Status=VERIFIED

username: EirikSak ##### Status=VERIFIED

description: Birthday present ##### Status=VERIFIED

amount: 1000 ##### Status=VERIFIED
```

Figure 3: Original transaction



```
firstname: Eirik ##### Status=VERIFIED

lastname: Sakariassen ##### Status=VERIFIED

username: EirikSak ##### Status=VERIFIED

description: Birthday present ##### Status=VERIFIED

amount: 10000 ###################WARNING###################Status=NOT VERIFIED!
```

Figure 4: Modified transaction

## 4.2 Execution Times

We measured execution time for key generation using different bit-sizes as shown in table 4.2. We did also measure execution time for the complete DSA algorithm using message lengths from 1 to 10000 as shown in figure 5.

| Key generation | | |
|---|---|---|
| parameters | mean | std |
| L=512,N=160 | 121ms | 25.4ms |
| L=1024,N=160 | 261ms | 66.1ms |
| L=512,N=10 | 9.27ms | 651us |
| L=1024,N=10 | 56.3ms | 14.8ms |

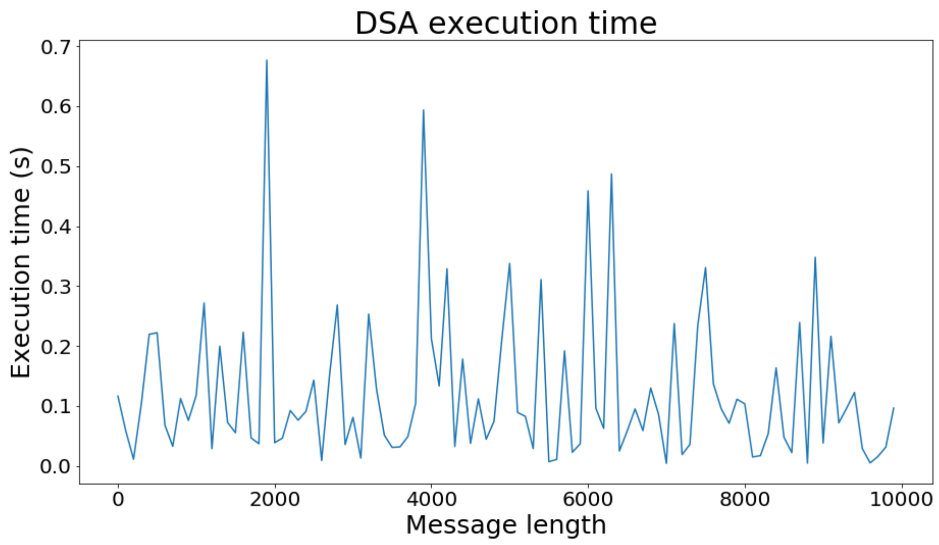


Figure 5: DSA execution time

## 5 Discussion

The DSA implementation worked as expected and executed all steps fairly fast. We did not try to optimize any parts more to speed up execution time other than using extended Euclidean algorithm. Key generation did, as expected, run faster with shorter lengths on the keys (small L or N). Very small N had bigger impact on the execution time than a minimum L of 512. We used L=512 and N=160 as default values in our program as these were pretty secure and ran fast.

The message length did not have any impact on the execution time, as shown in figure 5. The y-axis displays execution time in terms of seconds while the x-axis is the message length. We can see that the total time of DSA (including signing and validating) takes between 0.1 to 0.3 seconds normally. There are also spikes where it takes up to 0.7 seconds, but this does not depend on the size of the message.

Our SHA256 implementation did not work as expected and we did not fix the final bugs. The function hashed messages into 256-bit message digests and even a character change in the original message would produce a totally different hash. It did however not produce the same hash values as hashlib, so we could not use it in the system. We think the use of BitString objects for all bit manipulation caused problems as these objects were stored in memory and probably was used wrong within the loops of the algorithm. We tried to copy instances to avoid this issue, but did not suceed. Most of the implementation seems to work fine, so we would in the future try to handle the objects more carefully or just represent everything as hex digits.

## 6 Conclusion

We have in this assignment implemented a Digital Signature Scheme (DSS) using the Digital Signature Algorithm (DSA). In order to demonstrate this scheme, a online banking service coined CryptoBank was developed. The service enabled users to sign private transactions which a third party later on could verify. Everything was accomplished by using the implemented DSA which relied on a secure hash algorithm; SHA256. We tried to implement SHA256 from scratch, but encountered some bugs in the last steps which was'nt resolved. We did therefore use *hashlib* implementation of SHA256 in the final system to ensure consistency. We would never use our own implementations in a real world application anyways since crypto libraries are maintained by trusted cryptography experts. Small bugs in a system like this could could cause major problems for both users and the organization running it.

## References

[1] Wikipedia contributors. Digital signature standard — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Digital_Signature_Standard&oldid=972298776, 2020. [Online; accessed 2-November-2020].

[2] C.F. Kerry and P.D. Gallagher. Digital signature standars (dss. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf, 2013. National Institute of Standards and Technology, U.S. Dept. Commerce, Rep. FIPS PUB 186-4.

[3] Wikipedia contributors. Digital signature algorithm — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Digital_Signature_Algorithm&oldid=981973635, 2020. [Online; accessed 2-November-2020].

[4] Quynh H. Dang. Secure hash standard (shs). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, 2015. Federal Inf. Process. Stds. (NIST FIPS) - 180-4.

[5] Aaron Russell. What is a certificate authority (ca)? https://www.ssl.com/faqs/what-is-a-certificate-authority/, 2019.

[6] J. A. Berkowsky and T. Hayajneh. Security issues with certificate authorities. In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 449–455, 2017.

[7] Jake Ludin. Risks of a public certificate authority. `https://securityboulevard.com/2020/01/risks-of-a-public-certificate-authority/`, 2020. [Online; accessed 2-November-2020].

[8] Wikipedia contributors. Social engineering (security) — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Social_engineering_(security)&oldid=986491612`, 2020. [Online; accessed 2-November-2020].

[9] Nick Naziridis. Browsers and certificate validation. `https://www.ssl.com/article/browsers-and-certificate-validation/`, 2019.