Department of Computer Engineering and Science,
Florida Institute of Technology, Melbourne, Florida

DIGITAL SIGNATURE ALGORITHM

By

Muskan Jain

Pooja Choudhary

Sheetal Ghodake

Venkatesh Munaga

Professor

Dr. William Shoaff

CSE5211 Analysis of Algorithms

Group Project (Fall 2021)

TABLE OF CONTENTS

1. Problem Statement..... 3

2. Solution..... 3

3. What is Digital Signature 4

4.The Working of Digital Signature Algorithm 4

5. Digital Signature implemented with DSA in real life..... 5

6. DSA Algorithm (Pseudocode) 6

7. Theoretical Explanation of Algorithm..... 7

8. Analysis..... 8

9. Statistics..... 9

9.1CProfiler Screenshots..... 9

9.2 Comparison Graphs 12

10.Conclusion..... 13

11. Reference..... 13

12. Appendix..... 14

12.1 Python Code..... 14

1. Problem statement

Nowadays, because of the lack of face-to-face interaction and standard distance constraints have had to digitally sign some official documents over the past couple of years. Thus, all vital data must be signed by its owner before being sent safely inside the network. We must ensure that only authorized people have access to the data. Because only the endpoints have the authority to sign or validate the data throughout the verification process.

However, as cyber attackers get more competent at detecting holes in information systems, this is becoming increasingly difficult. They've figured out how to launch an attack against e-signature validations. There are several validation attacks that allow hackers to exploit certain vulnerabilities that may exist in various electronic signatures.

2. Solution

Therefore DSA Algorithm was proposed to find the optimal algorithm for digital signature is to deal with few issues that can provide security services which need to be implemented in network. This application mainly deals with security threats that are seen in network and to provide a end to end secured communication for user without any middle hackings or threats to messages and deal with key areas like, authentication, non repudiation and Data integrity, confidentiality.
Digital signature Algorithm is one of the techniques used in the field of Computer Science to generate signatures and verify to authenticate any financial, legal documents. It employs asymmetric cryptography. If the algorithm is properly implemented then the digital signature received by the receiver provides a valid reason that the message was sent from the right user.

Before understanding what is Digital Signature Algorithm first we need to understand what is digital Signature.

3. What is Digital signature?

In today's modern world, digital signatures are necessary to verify the identity and Integrity of a document. In order to authenticate any financial, legal documents, there should be a presence of handwritten signature over that document. The receiver of the signed document can make use of the signature to validate the documents received even if the sender disowns the contents of the document. With the advancement of computerized technology in sending or receiving the crucial documents, Digital signatures is the best option to do it among various other methods that have evolved in years. Digital signature means that we should be able to verify the author, time of the signature and authenticate message contents which can also be verified by the third parties to resolve disputes. It is intended to use in electronic mails, electronic fund transfer, software distribution and other type of applications which requires integrity and authentication of data.

These signatures are created using certain algorithms. In this project we are using Digital Signature Algorithm. DSA is a type of public-key encryption algorithm, and it is used to generate an electronic signature.

4. The Working of Digital Signature Algorithm

The DSA algorithm is a digital signature standard that is based on the public-key cryptosystems principle and is based on the algebraic features of discrete logarithm problems and modular exponentiations. The National Institute of Standards and Technology suggested it in 1991, and it was universally standardized in 1994. (NIST).
Digital signatures are based on the concept of two cryptographic keys that are mutually authenticating. Public/private key pairs are used to create signatures. A mathematically connected private key and public key can be created using a public-key algorithm. With his private key, one can sign a digital communication. A private key can be used to encrypt data relating to signatures. A person who wishes to make a digital signature should always have their private key with them. Because the public and private keys are mathematically connected, they can always be deduced from one another. The only method to decode this data is to use the signer's public key. Anyone who requires verification of the signer's signature can be given the public key. It is critical to keep your private key confidential since it can be used to generate your signature on a document. The authentication digital signature is completed in this manner. Only public and private keys ensure the validity of a digital signature.

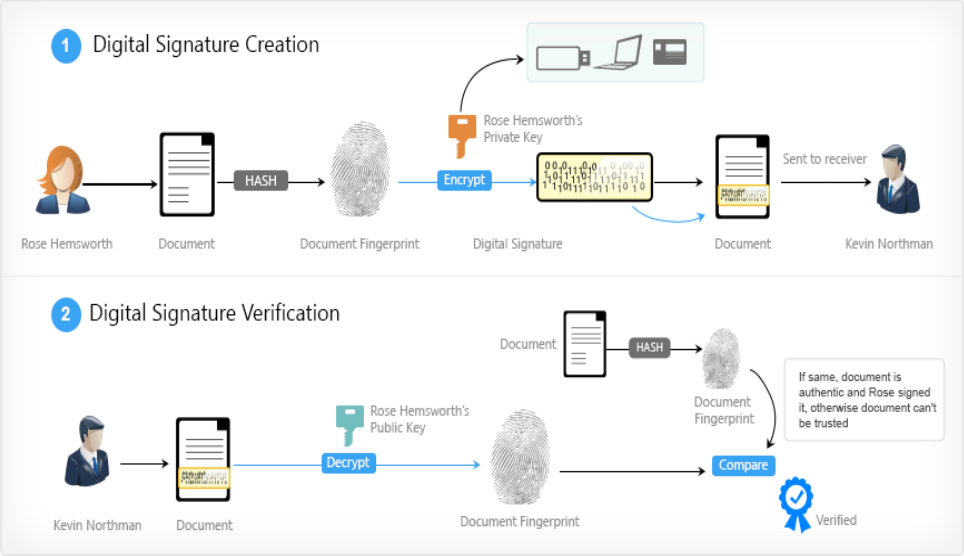
On the other hand, the digital signature algorithm does not encrypt data with a private key. A public key is also used to decode the data in a digital signature process. DSA uses the notion of a unique mathematical function to construct a digital signature with two 160-bit values. The private key and the message digest are used to generate these two

integers. The verification method is complicated since the public key is not utilized to authenticate the signature. For further protection, both keys are utilized to safeguard data in a particular digital signature algorithm.

The message digest is now created using a hash function. The digital signature is created using the produced message digest and the DSA algorithm. The message is then sent with this signature attached. The same hash algorithm is used to authenticate the source and data on the receiving end.

5. Digital Signature implemented with DSA in real life

Scenario to explain how documents will be share between Rose and Kevin using digital signature algorithm in real life.



1. First, Rose generates two keys: Public key and Private key.
2. She has a document she wants to send to Kevin.
3. She creates the hash of the document and encrypts just the hash of the document using her private key.
4. She attaches the hash with the original document and then sends it to Kevin.
5. On the receiving side, Kevin decrypts the hash that was sent by Rose.
6. He then calculates the hash of the document using the same algorithm that Rose used and then compares the hash of the document that he created along with the decrypt hash and therefore if both of them are same, then the document is as it is and not been forged or corrupted.

6. DSA Algorithm (Pseudocode)

Three step process is required to accomplish this algorithm:-

1. DSA Key Generation

The key generation process generates two keys, one is public and one is private. The Key Generation phase calculates several values like p, q, g, x and y and packs them all as public key {p, q, g, y} and private key {p, q, g, x}. The detailed steps are given below.

Steps to generate public and private keys:

- Select a prime number q, also known as the prime divisor.
- Choose a different primer number, p, so that p-1 mod q = 0. This is known as the prime modulus.
- Choose an integer g such that 1 g p, g**q mod p = 1, and g = h**((p–1)/q) mod p are all equal to 1. G's multiplicative order modulo p is also known as q.
- Choose an integer that equals 0 x q.
- Substitute g**x mod p for y.
- Use the letters p, q, g, and y to package the public key.
- Use the letters p, q, g, and x to package the private key.

2. DSA Signature Generation

- Using the SHA-2 hash method, generate the message digest h.
- Pick an integer k at random, such that 0 k q.
- Substitute (gk mod p) mod q for r. Choose a different k if r = 0.
- Determine I so that k * I mod q = 1. The modular multiplicative inverse of k modulo q is termed i.
- Substitute s = i*(h + r*x) mod q for s. Choose a different k if s = 0.
- The signature is valid (r, s).

3. DSA Signature Verification

- To create the digest h, you utilize the same hash function (H#).
- You then provide this digest to the verification function, which requires additional arguments.
- If 0rq or 0sq are not fulfilled, reject the signature.
- Work out w = s-1 mod q.
- Substitute u1 = H(m).w mod q into the equation.
- Work out u2 = r. w mod q.
- Substitute v = (gu1 yu2 mod p) mod q into the equation.
- Unless v = r, the signature is invalid.

7. Theoretical explanation of the Algorithm

1. Different senders can send messages to the same user. Some may use a digital signature, while others may not. The user accepts a message without verifying it if it is not digitally signed. In the event of a digitally signed communication, however, he may validate the message using the sender's public key.

2. The received message is put into the SHA for the message digest to be generated. The SHA used by the receiver and the sender must be same. As a result, SHA will generate the same message digest if the message content remains unchanged throughout transmission.

3. The DSA verify unit takes the SHA message digest as well as the receiver's public key. The received digital signature is confirmed using the DSA parameters, public key, and message digest. If the signature is validated, the message's integrity is assured. as well as the sender's identity being verified If it doesn't get confirmed, it's either because the message was damaged during transmission or because the private key used doesn't match the public key. The message is invalid in either instance and should be discarded by the recipient.

8. Analysis of the Digital Signature Algorithm

1. A Signature Generation Time complexity

We calculated r and s for signature generation , based on that given below is the time complexity of signature generation

- r value complexity
  r= (g^k mod p) mod q
  O(r) = O(log n) according to fast power algorithm**.**

- s value complexity
  s = i*(h + r*x) mod q
  O(s) = O(log n) according to Extended Euclid's Algorithm.

There is one power, one inverse operation, one addition operation, two multiplication operations, and three Mod operations in the signature individually. When compared to power and inverse operations, addition, multiplication, and Mod operations come at a little or no cost. Therefore, O(Signature Generation)=O(2log n)

2. A Signature Generation Time complexity

We calculated w,u1,u2 and v for signature verification, based on that given below is the time complexity of signature verification

- w value complexity
  w = s^-1 mod q
  O(w) = $O(log\ n)$ according to Extended Euclid's Algorithm.

- u1 value complexity
  u1 = H(m).w mod q =
  O(u1) = $O(1)$ one multiplication operation of h(m) with w.

- u2 value complexity
  u2 = r. w mod q
  O(u2) = $O(1)$ one multiplication operation of r and w.

- v value complexity
  v = (g^u1 y^u2 mod p) mod q
  O(v) = $O(log\ n)$ + $O(log\ n)$ + $O(1)$ according to fast power Algorithm.

Therefore , O(Signature Verification) = O(3logn).

So **O(DSA)=logn**

9. Statistics

We used CProfiler tool to analyze the code and calculate the time consumed.

9.1 CProfiler Screenshots

```
>>>
= RESTART: C:\Users\Pooja\Desktop\Study_FirstTerm\Computer Science\Analysis Of Algorithms\Group_Pro
Prime divisor (q):  29
Prime modulus (p):  523
Enter integer between 2 and p-1(h): 67
Value of g is :  280
Randomly chosen x(Private key) is:  4
Randomly chosen y(Public key) is:  408

Enter the name of document to sign: Test.txt
Hash of document sent is:  a9303f775b41b9e05ae5d56715704f17516f5f6c
         240 function calls in 0.005 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.005    0.005 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 _bootlocale.py:11(getpreferredencoding)
        1    0.000    0.000    0.000    0.000 codecs.py:260(__init__)
        1    0.000    0.000    0.000    0.000 cp1252.py:22(decode)
        1    0.000    0.000    0.000    0.000 digital_signature.py:15(mod_inverse)
        1    0.000    0.000    0.005    0.005 digital_signature.py:73(signature)
        1    0.000    0.000    0.000    0.000 digital_signature.py:9(hash_function)
        1    0.000    0.000    0.000    0.000 random.py:237(_randbelow_with_getrandbits)
        1    0.000    0.000    0.000    0.000 random.py:290(randrange)
        1    0.000    0.000    0.000    0.000 random.py:334(randint)
       28    0.000    0.000    0.000    0.000 rpc.py:153(debug)
        4    0.000    0.000    0.004    0.001 rpc.py:216(remotecall)
        4    0.000    0.000    0.000    0.000 rpc.py:226(asynccall)
        4    0.000    0.000    0.004    0.001 rpc.py:246(asyncreturn)
        4    0.000    0.000    0.000    0.000 rpc.py:252(decoderesponse)
        4    0.000    0.000    0.004    0.001 rpc.py:290(getresponse)
        4    0.000    0.000    0.000    0.000 rpc.py:298(_proxify)
        4    0.000    0.000    0.004    0.001 rpc.py:306(_getresponse)
        4    0.000    0.000    0.000    0.000 rpc.py:328(newseq)
        4    0.000    0.000    0.000    0.000 rpc.py:332(putmessage)
        4    0.000    0.000    0.000    0.000 rpc.py:559(__getattr__)
        4    0.000    0.000    0.000    0.000 rpc.py:57(dumps)
        4    0.000    0.000    0.000    0.000 rpc.py:601(__init__)
        4    0.000    0.000    0.004    0.001 rpc.py:606(__call__)
        8    0.000    0.000    0.000    0.000 run.py:420(encoding)
        8    0.000    0.000    0.000    0.000 run.py:424(errors)
        4    0.000    0.000    0.004    0.001 run.py:441(write)
        8    0.000    0.000    0.000    0.000 threading.py:1338(current_thread)
        4    0.000    0.000    0.000    0.000 threading.py:228(__init__)
        4    0.000    0.000    0.004    0.001 threading.py:280(wait)
        4    0.000    0.000    0.000    0.000 threading.py:82(RLock)
        1    0.000    0.000    0.000    0.000 {built-in method _codecs.charmap_decode}
        1    0.000    0.000    0.000    0.000 {built-in method _hashlib.openssl_sha1}
        1    0.000    0.000    0.000    0.000 {built-in method _locale._getdefaultlocale}
        4    0.000    0.000    0.000    0.000 {built-in method _struct.pack}
        4    0.000    0.000    0.000    0.000 {built-in method _thread.allocate_lock}
        8    0.000    0.000    0.000    0.000 {built-in method _thread.get_ident}
```

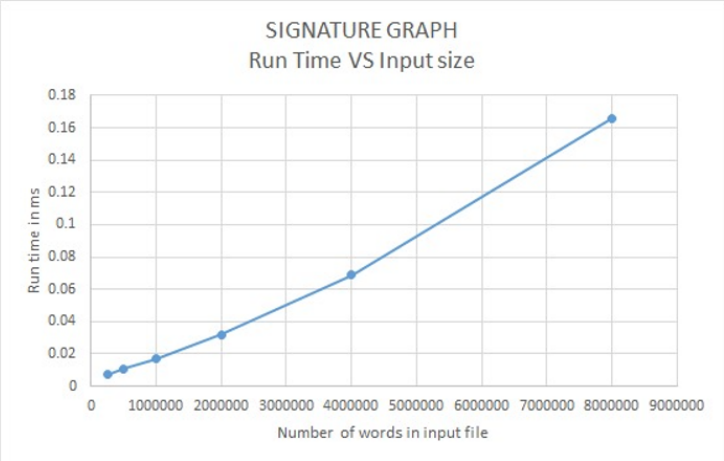
```
  8    0.000    0.000    0.000    0.000 {built-in method _thread.get_ident}
  1    0.000    0.000    0.005    0.005 {built-in method builtins.exec}
  8    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
 12    0.000    0.000    0.000    0.000 {built-in method builtins.len}
  1    0.000    0.000    0.000    0.000 {built-in method builtins.pow}
  1    0.000    0.000    0.004    0.004 {built-in method builtins.print}
  1    0.000    0.000    0.000    0.000 {built-in method io.open}
  4    0.000    0.000    0.000    0.000 {built-in method select.select}
  1    0.000    0.000    0.000    0.000 {method '__exit__' of '_io._IOBase' objects}
  4    0.000    0.000    0.000    0.000 {method '_acquire_restore' of '_thread.RLock' objects}
  4    0.000    0.000    0.000    0.000 {method '_is_owned' of '_thread.RLock' objects}
  4    0.000    0.000    0.000    0.000 {method '_release_save' of '_thread.RLock' objects}
  4    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.RLock' objects}
  8    0.004    0.000    0.004    0.000 {method 'acquire' of '_thread.lock' objects}
  4    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
  1    0.000    0.000    0.000    0.000 {method 'bit_length' of 'int' objects}
  4    0.000    0.000    0.000    0.000 {method 'decode' of 'bytes' objects}
  1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
  4    0.000    0.000    0.000    0.000 {method 'dump' of '_pickle.Pickler' objects}
  5    0.000    0.000    0.000    0.000 {method 'encode' of 'str' objects}
  4    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
  1    0.000    0.000    0.000    0.000 {method 'getrandbits' of '_random.Random' objects}
  4    0.000    0.000    0.000    0.000 {method 'getvalue' of '_io.BytesIO' objects}
  1    0.000    0.000    0.000    0.000 {method 'hexdigest' of '_hashlib.HASH' objects}
  1    0.000    0.000    0.001    0.001 {method 'read' of '_io.TextIOWrapper' objects}
  4    0.000    0.000    0.000    0.000 {method 'release' of '_thread.RLock' objects}
  4    0.000    0.000    0.000    0.000 {method 'send' of '_socket.socket' objects}
```

```
r(Component of signature) is:  3
k(Randomly chosen number) is:  7
s(Component of signature) is:  3

Enter the name of document to verify: Test.txt
Hash of document received is:  a9303f775b41b9e05ae5d56715704f17516f5f6c
Value of w is :  10
Value of u1 is:  3
Value of u2 is:  1
Value of v is :  3
The signature is valid!
         1213 function calls in 0.034 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.034    0.034 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 _bootlocale.py:11(getpreferredencoding)
        1    0.000    0.000    0.000    0.000 codecs.py:260(__init__)
        1    0.000    0.000    0.001    0.001 cp1252.py:22(decode)
        1    0.000    0.000    0.000    0.000 digital_signature.py:15(mod_inverse)
        1    0.000    0.000    0.000    0.000 digital_signature.py:9(hash_function)
        1    0.000    0.000    0.034    0.034 digital_signature.py:94(verification)
      154    0.000    0.000    0.000    0.000 rpc.py:153(debug)
       22    0.000    0.000    0.032    0.001 rpc.py:216(remotecall)
       22    0.000    0.000    0.001    0.000 rpc.py:226(asynccall)
       22    0.000    0.000    0.030    0.001 rpc.py:246(asyncreturn)
       22    0.000    0.000    0.000    0.000 rpc.py:252(decoderesponse)
       22    0.000    0.000    0.030    0.001 rpc.py:290(getresponse)
       22    0.000    0.000    0.000    0.000 rpc.py:298(_proxify)
       22    0.000    0.000    0.030    0.001 rpc.py:306(_getresponse)
       22    0.000    0.000    0.000    0.000 rpc.py:328(newseq)
       22    0.000    0.000    0.001    0.000 rpc.py:332(putmessage)
       22    0.000    0.000    0.000    0.000 rpc.py:559(__getattr__)
       22    0.000    0.000    0.000    0.000 rpc.py:57(dumps)
       22    0.000    0.000    0.000    0.000 rpc.py:601(__init__)
       22    0.000    0.000    0.032    0.001 rpc.py:606(__call__)
       44    0.000    0.000    0.000    0.000 run.py:420(encoding)
       44    0.000    0.000    0.000    0.000 run.py:424(errors)
       22    0.000    0.000    0.032    0.001 run.py:441(write)
       44    0.000    0.000    0.000    0.000 threading.py:1338(current_thread)
       22    0.000    0.000    0.000    0.000 threading.py:228(__init__)
       22    0.000    0.000    0.030    0.001 threading.py:280(wait)
       22    0.000    0.000    0.000    0.000 threading.py:82(RLock)
        1    0.001    0.001    0.001    0.001 {built-in method _codecs.charmap_decode}
        1    0.000    0.000    0.000    0.000 {built-in method _hashlib.openssl_sha1}
        1    0.000    0.000    0.000    0.000 {built-in method _locale._getdefaultlocale}
       22    0.000    0.000    0.000    0.000 {built-in method _struct.pack}
       22    0.000    0.000    0.000    0.000 {built-in method _thread.allocate_lock}
       44    0.000    0.000    0.000    0.000 {built-in method _thread.get_ident}
```

```
44    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
66    0.000    0.000    0.000    0.000 {built-in method builtins.len}
 2    0.000    0.000    0.000    0.000 {built-in method builtins.pow}
 6    0.000    0.000    0.032    0.005 {built-in method builtins.print}
 1    0.000    0.000    0.000    0.000 {built-in method io.open}
22    0.000    0.000    0.000    0.000 {built-in method select.select}
 1    0.000    0.000    0.000    0.000 {method '__exit__' of '_io._IOBase' objects}
22    0.000    0.000    0.000    0.000 {method '_acquire_restore' of '_thread.RLock' objects}
22    0.000    0.000    0.000    0.000 {method '_is_owned' of '_thread.RLock' objects}
22    0.000    0.000    0.000    0.000 {method '_release_save' of '_thread.RLock' objects}
22    0.000    0.000    0.000    0.000 {method 'acquire' of '_thread.RLock' objects}
44    0.030    0.001    0.030    0.001 {method 'acquire' of '_thread.lock' objects}
22    0.000    0.000    0.000    0.000 {method 'append' of 'collections.deque' objects}
22    0.000    0.000    0.000    0.000 {method 'decode' of 'bytes' objects}
 1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
22    0.000    0.000    0.000    0.000 {method 'dump' of '_pickle.Pickler' objects}
23    0.000    0.000    0.000    0.000 {method 'encode' of 'str' objects}
22    0.000    0.000    0.000    0.000 {method 'get' of 'dict' objects}
22    0.000    0.000    0.000    0.000 {method 'getvalue' of '_io.BytesIO' objects}
 1    0.000    0.000    0.000    0.000 {method 'hexdigest' of '_hashlib.HASH' objects}
 1    0.001    0.001    0.001    0.001 {method 'read' of '_io.TextIOWrapper' objects}
22    0.000    0.000    0.000    0.000 {method 'release' of '_thread.RLock' objects}
22    0.001    0.000    0.001    0.000 {method 'send' of '_socket.socket' objects}
```

9.2 Comparison Graph

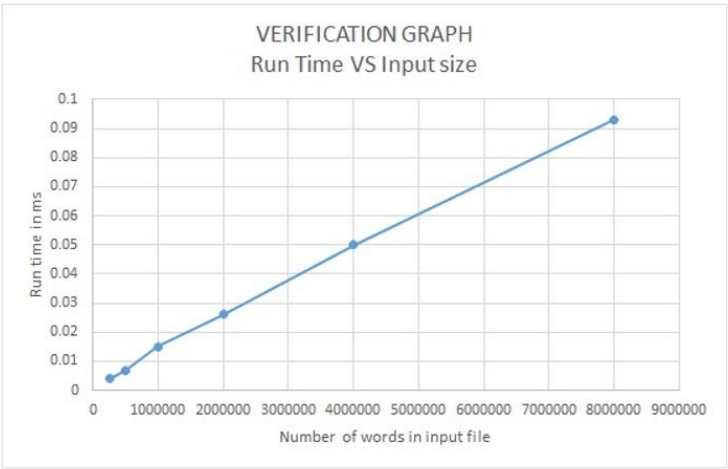
1. A Signature Generation Time complexity Graph



- We started giving input with 250,000 words and doubled in each iteration till we reached 8,000,000 words.We observed that as the number of words in the input increases, the amount of time for signature generation is also increases.

- **R-Squared value is 0.9926**

2. A Signature Verification Time complexity Graph



- We started giving input with 250,000 words and doubled in each iteration till we reached 8,000,000 words.We noticed that as the number of words in the input rises, so does the verification time long it takes.

- **R-Squared value is 0.9985**

10. Conclusion

In today's world, a digital signature algorithm is an early and extremely simple implementation of what we term security. On the basis of cost, security, time, and speed, the digital signature algorithm is one of the finest authentication algorithms. DSA was designed to be safe in the past, but now that we're dealing with sensitive data and content, we need a stronger algorithm that can describe nearly every restriction that exists and therefore be secure enough for everyone. There are a number of algorithms for generating and verifying digital signatures on documents and messages, although not all of them are well-known or extensively used. DSA is still the most popular digital signature algorithm. There are many more powerful and secure algorithms available now, like as RSA, but they were created through DSA. As a result, DSA is the most advanced and safe algorithm available today.

11. References

[1] "FIPS PUB 186]: Digital Signature Standard (DSS), 1994-05-19". csrc.nist.gov.

[2] Bendel, Mike (2010-12-29). "Hackers Describe PS3 Security As Epic Fail, Gain Unrestricted Access". Exophase.com. Retrieved 2011-01-05.

[3] "FIPS PUB 180-4: Secure Hash Standard (SHS), March 2012" (PDF). csrc.nist.gov.

[4] "NIST Special Publication 800-57" (PDF). csrc.nist.gov.

[5] "The Debian PGP disaster that almost was". root labs rdist.

[6]"https://meu.edu.jo/librarytheses/5a15379df21ea_1.pdf. (n.d.). Retrieved December 10, 2021, from https://www.researchgate.net/publication/339746817_httpsmeuedujolibraryTheses5a15379df21ea_1pdf " (PDF)

[7]"Crypto-systems/digital_signature.py at master .... https://github.com/Abhiramborige/Crypto-systems/blob/master/digital_signature.py"

12. Appendix

We have attached source codes below.
This is code for user input where the user can enter inputs manually.
Therefore, according to user inputs, code will get executed and give suitable output.

12.1 Python code

```
from Crypto.Util.number import *
from random import *
from hashlib import sha1
import cProfile

# create Hash of message in SHA1

def hash_function(message):
    hashed = sha1(message.encode("UTF-8")).hexdigest()
    return hashed

# Modular Multiplicative Inverse

def mod_inverse(a, m):
    a = a % m;
    for x in range(1, m):
        if ((a * x) % m == 1):
            return (x)
    return (1)

# create q,p, and g as Global parameters

def parameter_generation():
    # primes of 8 bits in length in binary
    # q is prime divisor
    q = getPrime(5)
    # p is prime modulus
    p = getPrime(10)

    # Always p should be greater than q
    # because p-1 must be a multiple of q

    # to make sure that p not equal to q while generating randomly
    # and q is prime divisor of p-1
    while ((p - 1) % q != 0):
        p = getPrime(10)
        q = getPrime(5)
```

```
    print("Prime divisor (q): ", q)
    print("Prime modulus (p): ", p)

    flag = True
    while (flag):
        h = int(input("Enter integer between 2 and p-1(h): "))
        # h must be in between 1 and p-1
        if (1 < h < (p - 1)):
            g = 1
            while (g == 1):
                g = pow(h, int((p - 1) / q)) % p
            flag = False
        else:
            print("Wrong entry")
    print("Value of g is : ", g)

    # returning them as they are public globally
    return (p, q, g)

def per_user_key(p, q, g):
    # User private key:
    x = randint(1, q - 1)
    print("Randomly chosen x(Private key) is: ", x)

    # User public key:
    y = pow(g, x) % p
    print("Randomly chosen y(Public key) is: ", y)

    # returning private and public components
    return (x, y)

def signature(name, p, q, g, x):
    with open(name) as file:
        text = file.read()
        hash_component = hash_function(text)
        print("Hash of document sent is: ", hash_component)
    r = 0
    s = 0
    while (s == 0 or r == 0):
        k = randint(1, q - 1)
        r = ((pow(g, k)) % p) % q
        i = mod_inverse(k, q)

        # converting hexa decimal to binary
        hashed = int(hash_component, 16)
```

```
        s = (i * (hashed + (x * r))) % q

    # returning the signature components
    return (r, s, k)
# call for verification

def verification(name, p, q, g, r, s, y):
    with open(name) as file:
        text = file.read()
        hash_component = hash_function(text)
        print("Hash of document received is: ", hash_component)

    # computing w
    w = mod_inverse(s, q)
    print("Value of w is : ", w)

    hashed = int(hash_component, 16)

    # computing u1, u2 and v
    u1 = (hashed * w) % q
    u2 = (r * w) % q
    v = ((pow(g, u1) * pow(y, u2)) % p) % q

    print("Value of u1 is: ", u1)
    print("Value of u2 is: ", u2)
    print("Value of v is : ", v)

    if (v == r):
        print("The signature is valid!")
    else:
        print("The signature is invalid!")


global_var = parameter_generation()
keys = per_user_key(global_var[0], global_var[1], global_var[2])

# Sender's side (signing the document):
print()
file_name = input("Enter the name of document to sign: ")
#components = signature(file_name, global_var[0], global_var[1], global_var[2],
keys[0])
cProfile.run('components = signature(file_name, global_var[0], global_var[1],
global_var[2], keys[0])')


print("r(Component of signature) is: ", components[0])
```

```
print("k(Randomly chosen number) is: ", components[2])
print("s(Component of signature) is: ", components[1])


# Receiver's side (verifying the sign):

print()
file_name = input("Enter the name of document to verify: ")
#verification(file_name, global_var[0], global_var[1], global_var[2], components[0], components[1], keys[1])
cProfile.run('verification(file_name, global_var[0], global_var[1], global_var[2], components[0], components[1], keys[1])')
```