File Security System for COVID-19 Test Results Using Steganography and Digital Signature

Vincent Budianto - 13517137
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517137@std.stei.itb.ac.id

***Abstract*—In early 2020, the world was caught off guard by the outbreak of unknown pneumonia that began in Wuhan, Hubei Province. It spread rapidly throughout more than 190 countries and territories. This outbreak is named coronavirus disease 2019 (COVID-19), caused by severe acute respiratory syndrome coronavirus-2 (SARS-CoV-2). Entering holiday seasons, many cities in Indonesia require visitors to have COVID-19 test result file to enter the city. However, at this time the file does not have a sufficient level of security therefore it can be misused by irresponsible parties. To overcome this lack of security, digital signature techniques and steganography can be applied to the COVID-19 test result file. This paper will explain the use of the steganography using Bit-Plane Complexity Segmentation Steganography (BPCS-steganography) and digital signature using Elliptic Curve Digital Signature Algorithm (ECDSA) to secure the COVID-19 test result file.**

***Keywords*—BPCS, Cryptography, Digital signature, ECDSA, Health, Steganography.**

I. Introduction

Communication has become a part of human life. Especially in the information age like now, communication is very crucial. There are times when the information is important and confidential. Therefore, the communication method used must be made in such a way that no other party knows about the information.

For this reason, cryptography was born, namely a method of processing information with a certain algorithm so that it becomes cryptic and its meaning is difficult to understand. However, this method often raises the suspicion of third parties, because messages that are difficult to understand must have been processed and show that the message is important information.

To avoid this problem, steganography was born, which is a method of hiding information in a medium, which can be in the form of image, sound or video media. The most important aspect of steganography is the level of security with which the information is hidden, which refers to the extent to which third parties are unable to detect the presence of hidden information.

Steganography is commonly used for hiding information on image media, where text information is inserted into image pixel bits. However, the frequently used methods are still simple enough that third parties can still find hidden information.

Therefore this paper discusses an implementation that makes text steganography on image media stronger and safer. This implementation signed a message with a digital signature using the Elliptic Curve Digital Signature Algorithm (ECDSA). The digital signed message is then entered into the image using Bit-Plane Complexity Segmentation Steganography (BPCS-steganography) algorithm.

II. Theory

A. COVID-19

COVID-19 is a contagious disease caused by severe acute respiratory syndrome with the first case identified in Wuhan, China, in December 2019. It has since spread worldwide, leading to an ongoing pandemic.

Symptoms of COVID-19 are variable, but often include fever, cough, fatigue, breathing difficulties, and loss of smell and taste. Symptoms begin one to fourteen days after exposure to the virus. Around one in five infected individuals do not develop any symptoms. While most people have mild symptoms, some people develop acute respiratory distress syndrome. Acute respiratory distress syndrome (ARDS) can be precipitated by cytokine storms, multi-organ failure, septic shock, and blood clots. Longer-term damage to organs has been observed. There is concern about a significant number of patients who have recovered from the acute phase of the disease but continue to experience a range of effects—known as long COVID—for months afterwards. These effects include severe fatigue, memory loss and other cognitive issues, low-grade fever, muscle weakness, and breathlessness.

The virus that causes COVID-19 spreads mainly when an infected person is in close contact with another person. Small droplets and aerosols containing the virus can spread from an infected person's nose and mouth as they breathe, cough, sneeze, sing, or speak. Other people are infected if the virus gets into their mouth, nose or eyes. The virus may also spread via contaminated surfaces, although this is not thought to be the main route of transmission. but infection mainly happens when people are near each other for long enough. It can spread as early as two days before infected persons show symptoms, and from individuals who never experience symptoms. People remain infectious for up to ten days in moderate cases, and two weeks in severe cases. Various testing methods have been

developed to diagnose the disease. The standard diagnosis method is by real-time reverse transcription polymerase chain reaction from a nasopharyngeal swab (SWAB test).

Preventive measures include physical or social distancing, quarantining, ventilation of indoor spaces, covering coughs and sneezes, hand washing, and keeping unwashed hands away from the face. The use of face masks or coverings has been recommended in public settings to minimise the risk of transmissions. Several vaccines have been developed and various countries have initiated mass vaccination campaigns.

Although work is underway to develop drugs that inhibit the virus, the primary treatment is currently symptomatic. Management involves the treatment of symptoms, supportive care, isolation, and experimental measures.

B. Cryptography

Cryptography, or cryptology, is the practice and study of techniques for secure communication in the presence of third parties called adversaries. More generally, cryptography is about constructing and analyzing protocols that prevent third parties or the public from reading private messages; various aspects in information security such as data confidentiality, data integrity, authentication, and non-repudiation are central to modern cryptography. Modern cryptography exists at the intersection of the disciplines of mathematics, computer science, electrical engineering, communication science, and physics. Applications of cryptography include electronic commerce, chip-based payment cards, digital currencies, computer passwords, and military communications.

Cryptography prior to the modern age was effectively synonymous with encryption, converting information from a readable state to unintelligible nonsense. The sender of an encrypted message shares the decoding technique only with intended recipients to preclude access from adversaries. The cryptography literature often uses the names Alice for the sender, Bob for the intended recipient, and Eve for the adversary. Since the development of rotor cipher machines in World War I and the advent of computers in World War II, cryptography methods have become increasingly complex and its applications more varied.

Modern cryptography is heavily based on mathematical theory and computer science practice; cryptographic algorithms are designed around computational hardness assumptions, making such algorithms hard to break in actual practice by any adversary. While it is theoretically possible to break into a well-designed such system, it is infeasible in actual practice to do so. Such schemes, if well designed, are therefore termed "computationally secure"; theoretical advances, e.g., improvements in integer factorization algorithms, and faster computing technology require these designs to be continually reevaluated, and if necessary, adapted. There exist information-theoretically secure schemes that cannot be broken even with unlimited computing power — an example is the one-time pad —- but these schemes are much more difficult to use in practice than the best theoretically breakable but computationally secure schemes.

The growth of cryptographic technology has raised a number of legal issues in the information age. Cryptography's potential for use as a tool for espionage and sedition has led many governments to classify it as a weapon and to limit or even prohibit its use and export. Cryptography also plays a major role in digital rights management and copyright infringement disputes in regard to digital media.

Until modern times, cryptography referred almost exclusively to encryption, which is the process of converting ordinary information into unintelligible form . Examples of asymmetric systems include RSA, and ECC . Quality symmetric algorithms include the commonly used AES which replaced the older DES . Not very high quality symmetric algorithms include the assorted children's language tangling schemes such as Pig Latin or other cant, and indeed effectively all cryptographic schemes, however seriously intended, from any source prior to the invention of the one-time pad early in the 20th century.

C. Steganography

Steganography is the practice of concealing a message within another message or a physical object. In computing/electronic contexts, a computer file, message, image, or video is concealed within another file, message, image, or video. The word steganography comes from Greek steganographia, which combines the words steganós, meaning "covered or concealed", and -graphia meaning "writing".

The first recorded use of the term was in 1499 by Johannes Trithemius in his Steganographia, a treatise on cryptography and steganography, disguised as a book on magic. Generally, the hidden messages appear to be something else: images, articles, shopping lists, or some other cover text. For example, the hidden message may be in invisible ink between the visible lines of a private letter. Some implementations of steganography that lack a shared secret are forms of security through obscurity, and key-dependent steganographic schemes adhere to Kerckhoffs's principle.

The advantage of steganography over cryptography alone is that the intended secret message does not attract attention to itself as an object of scrutiny. Plainly visible encrypted messages, no matter how unbreakable they are, arouse interest and may in themselves be incriminating in countries in which encryption is illegal.

Whereas cryptography is the practice of protecting the contents of a message alone, steganography is concerned both with concealing the fact that a secret message is being sent and its contents.

Steganography includes the concealment of information within computer files. In digital steganography, electronic communications may include steganographic coding inside of a transport layer, such as a document file, image file, program or protocol. Media files are ideal for steganographic transmission because of their large size. For example, a sender might start with an innocuous image file and adjust the color of every hundredth pixel to correspond to a letter in the alphabet. The change is so subtle that someone who is not specifically looking for it is unlikely to notice the change.

D. *Bit-Plane Complexity Segmentation Steganography*

The Bit-Plane Complexity Segmentation Steganography (BPCS-steganography) is a type of digital steganography. Digital steganography can hide confidential data securely by embedding them into some media data called "vessel data". The vessel data is also referred to as "carrier, cover, or dummy data". In BPCS-steganography, true color images are mostly used for vessel data. The embedding operation in practice is to replace the "complex areas" on the bit planes of the vessel image with the confidential data. The most important aspect of BPCS-steganography is that the embedding capacity is very large. In comparison to simple image based steganography which uses solely the least important bit of data, and thus can only embed data equivalent to 1/8 of the total size, BPCS-steganography uses multiple bit-planes, and so can embed a much higher amount of data, though this is dependent on the individual image. For a 'normal' image, roughly 50% of the data might be replaceable with secret data before image degradation becomes apparent.

BPCS-steganography makes use of the special features of human visual systems which have special properties so that visual patterns that are too complex cannot be considered "informative forms". For example, on a very flat beach shore every single square-foot area looks the same - it is just a sandy area, no shape is observed. However, if you look carefully, two same-looking areas are entirely different in their sand particle shapes. BPCS-steganography makes use of this property. It replaces complex areas on the bit-planes of the vessel image with other complex data patterns . This replacing operation is called "embedding". No one can see any difference between the two vessel images of before and after the embedding operation.

However BPCS-steganography has an issue when the data to be embedded appears visually as simple information, if this simple information replaces the complex information in the original image it may create spurious 'real image information'. In this case the data is conjugated with a binary image, in order to create a reciprocal complex representation.

E. *Digital Signature*

A digital signature is a mathematical scheme for verifying the authenticity of digital messages or documents. A valid digital signature, where the prerequisites are satisfied, gives a recipient a very strong reason to believe that the message was created by a known sender, and that the message was not altered in transit .

Digital signatures are a standard element of most cryptographic protocol suites, and are commonly used for software distribution, financial transactions, contract management software, and in other cases where it is important to detect forgery or tampering.

Digital signatures are often used to implement electronic signatures, which includes any electronic data that carries the intent of a signature, but not all electronic signatures use digital signatures. In some countries, including Canada, South Africa, the United States, Algeria, Turkey, India, Brazil, Indonesia, Mexico, Saudi Arabia, Uruguay, Switzerland, Chile and the countries of the European Union, electronic signatures have legal significance.

Digital signatures employ asymmetric cryptography. In many instances they provide a layer of validation and security to messages sent through a non-secure channel: Properly implemented, a digital signature gives the receiver reason to believe the message was sent by the claimed sender. Digital signatures are equivalent to traditional handwritten signatures in many respects, but properly implemented digital signatures are more difficult to forge than the handwritten type. Digital signature schemes, in the sense used here, are cryptographically based, and must be implemented properly to be effective. Digital signatures can also provide non-repudiation, meaning that the signer cannot successfully claim they did not sign a message, while also claiming their private key remains secret. Further, some non-repudiation schemes offer a timestamp for the digital signature, so that even if the private key is exposed, the signature is valid. Digitally signed messages may be anything representable as a bitstring: examples include electronic mail, contracts, or a message sent via some other cryptographic protocol.

A digital signature is an authentication mechanism that enables the creator of the message to attach a code that acts as a signature. Typically, a digital signature consists of three algorithms;

1. A key generation algorithm that selects a private key uniformly at random from a set of possible private keys. The algorithm outputs the private key and a corresponding public key.
2. A signing algorithm that, given a message and a private key, produces a signature.
3. A signature verifying algorithm that, given the message, public key and signature, either accepts or rejects the message's claim to authenticity.

Two main properties are required. First, the authenticity of a signature generated from a fixed message and fixed private key can be verified by using the corresponding public key. Secondly, it should be computationally infeasible to generate a valid signature for a party without knowing that party's private key.

The Elliptic Curve Digital Signature Algorithm (ECDSA), proposed by Scott Vanstone in 1992 in response to the National Institute of Standards and Technology (NIST) request for public comments on their first proposal for Digital Signature Standard (DSS), is one of many examples of a signing algorithm.

F. *Elliptic Curve Digital Signature Algorithm*

The Elliptic Curve Digital Signature Algorithm is a variant of the Digital Signature Algorithm which uses Elliptic Curve Cryptography.

As with Elliptic Curve Cryptography in general, the bit size of the public key believed to be needed for ECDSA is about twice the size of the security level, in bits. For example, at a security level of 80 bits the size of an ECDSA private key would be 160 bits, whereas the size of a DSA private key is at least 1024 bits. On the other hand, the signature size is the same for both DSA and ECDSA: approximately 4 t bits, where

t is the security level measured in bits, that is, about 320 bits for a security level of 80 bits.

III. Implementation

A. *BPCS Algorithm*

Here is an implementation of the BPCS algorithm in Python.

messageBPCS.py

```
import math
import numpy as np
import os
import random

Wc = np.array([[0, 1, 0, 1, 0, 1, 0, 1],
               [1, 0, 1, 0, 1, 0, 1, 0],
               [0, 1, 0, 1, 0, 1, 0, 1],
               [1, 0, 1, 0, 1, 0, 1, 0],
               [0, 1, 0, 1, 0, 1, 0, 1],
               [1, 0, 1, 0, 1, 0, 1, 0],
               [0, 1, 0, 1, 0, 1, 0, 1],
               [1, 0, 1, 0, 1, 0, 1, 0]])

class messageBPCS():
  def __init__(self, filename = None, content = None, key = None, threshold = 0.3, encrypted = False, randomized = False, block_size = 8):
    self.filename = filename
    self.content = content
    self.key = key
    self.threshold = threshold
    self.encrypted = encrypted
    self.randomized = randomized
    self.block_size = block_size
    self.bitplane = []

    if (self.filename != None):
      self.filedata = len(self.filename)
    else:
      self.filedata = None

    if (self.content != None):
      self.data = len(self.content)
    else:
      self.data = None

    if (self.key != None):
      self.seed = sum(ord(k) for k in key)

    self.header = None
    self.header_bitplane = []
    self.content_bitplane = []
    self.conjugation_map = []

  def shuffle(self, bitplane):
    random.seed(self.seed)
    random.shuffle(bitplane)

    return bitplane

  def unshuffle(self, bitplane):
    n = len(bitplane)
    perm = [i for i in range(1, (n + 1))]
    shuffled_perm = self.shuffle(perm)
    unshuffled = list(zip(bitplane, shuffled_perm))
    unshuffled.sort(key = lambda x: x[1])

    return [a for (a, b) in unshuffled]

  def to_binary(self, message):
    binary = [format(byte, '08b') for byte in message]

    while((len(binary) % self.block_size) != 0):
      binary.append('01010101')

    return binary

  def to_bitplane(self, binary):
    block = np.array([list(bit) for bit in binary])
    height, width = block.shape
    bitplane = []

    for h in range(0, (height - self.block_size + 1), self.block_size):
      for w in range(0, (width - self.block_size + 1), self.block_size):
        bitplane.append(block[h:(h + self.block_size), w:(w + self.block_size)].astype(int))

    return bitplane

  def complexity(self, bitplane):
    count = 0

    for h in range(self.block_size):
      for w in range(self.block_size):
        if (h != (self.block_size - 1)):
          if(bitplane[h][w] != bitplane[h + 1][w]):
            count += 1

        if (w != (self.block_size - 1)):
          if(bitplane[h][w] != bitplane[h][w + 1]):
            count += 1

    return count / 112

  def conjugate(self, plane):
    return plane ^ Wc

  def int_bitplane(self, x):
    bitplane = []

    byte = format(x, '064b')
    plane = np.array([list(byte[b:(b + self.block_size)]) for b in range(0, len(byte), self.block_size)])
    plane = self.conjugate(plane.astype(int))
    bitplane.append(plane)

    return bitplane

  def get_int(self, bitplane):
    bitplane = self.conjugate(bitplane)
    binary = ''.join([''.join(bit) for bit in bitplane.astype(str)])
    result = int(binary, 2)

    return result

  def set_header(self):
    header = ''

    if (self.encrypted):
      header += '22|'
    else:
      header += '11|'

    if (self.randomized):
      header += '22|'
    else:
      header += '11|'

    header += str(self.filedata) + '|'
    header += str(self.data) + '|'
    header += self.filename

    self.header = header.encode('utf-8')
    binary = self.to_binary(self.header)
    self.header_bitplane = self.to_bitplane(binary)
```

```
        return self.header_bitplane

    def get_header(self):
        self.header = self.get_byte(self.header_bitplane)
        self.header = self.header.decode('utf-8', errors='ignore')

        headers = self.header.split('|')

        if (int(headers[0]) == 22):
            self.encrypted = True
        elif (int(headers[0]) == 11):
            self.encrypted = False

        if (int(headers[1]) == 22):
            self.randomized = True
        elif (int(headers[1]) == 11):
            self.randomized = False

        self.filedata = int(headers[2])
        self.data = int(headers[3])

        return headers[4][:self.filedata]

    def set_content(self):
        content = self.to_binary(self.content)
        self.content_bitplane = self.to_bitplane(content)

        if (self.randomized):
            for i in range(len(self.content_bitplane)):
                binary = []
                old = self.get_byte(self.content_bitplane[i])
                new = self.shuffle(old)

                for z in range(0, len(new), self.block_size):
                    byte = new[z:(z + self.block_size)]
                    byte = [format(bit, '01b') for bit in byte]
                    binary.append(''.join([''.join(bit) for bit in byte]))

                self.content_bitplane[i] =
np.asarray(self.to_bitplane(binary))[0]

    def get_content(self):
        if (self.randomized):
            for i in range(len(self.content_bitplane)):
                binary = []
                old = self.get_byte(self.content_bitplane[i])
                new = self.unshuffle(old)

                for z in range(0, len(new), self.block_size):
                    byte = new[z:(z + self.block_size)]
                    byte = [format(bit, '01b') for bit in byte]
                    binary.append(''.join([''.join(bit) for bit in byte]))

                self.content_bitplane[i] = self.to_bitplane(binary)[0]

            content = self.get_byte(self.content_bitplane)[:self.data]
        else:
            content = self.get_byte(self.content_bitplane)[:self.data]

        return content

    def conjugate_content(self):
        i = 0
        j = 0

        while (i < len(self.header_bitplane)):
            if (self.complexity(self.header_bitplane[i]) < self.threshold):
                self.header_bitplane[i] =
self.conjugate(self.header_bitplane[i])
                self.conjugation_map.append(i)

            i += 1

        while (j < len(self.content_bitplane)):
            if (self.complexity(self.content_bitplane[j]) < self.threshold):
                self.content_bitplane[j] =
self.conjugate(self.content_bitplane[j])
                self.conjugation_map.append(i + j)

            j += 1

    def unconjugate_content(self, bitplane):
        cmap = []

        for i in range(len(self.conjugation_map)):
            self.conjugation_map[i] =
self.conjugate(self.conjugation_map[i])

        for plane in self.conjugation_map:
            cmap.append(''.join([''.join(bit) for bit in plane.astype(str)]))

        cmap = ''.join(cmap)

        for i in range(len(bitplane)):
            if (cmap[i] == '1'):
                bitplane[i] = self.conjugate(bitplane[i])

        return bitplane

    def conjugation_mapping(self):
        conjugation_map = ['0' for i in range(len(self.bitplane))]

        for i in self.conjugation_map:
            conjugation_map[i] = '1'

        while((len(conjugation_map) % self.block_size) != 0):
            conjugation_map.append('0')

        binary = [''.join(conjugation_map[j:(j + self.block_size)]) for j in
range(0, len(conjugation_map), self.block_size)]

        while((len(binary) % self.block_size) != 0):
            binary.append('01010101')

        self.conjugation_map = self.to_bitplane(binary)

        for j in range(len(self.conjugation_map)):
            self.conjugation_map[j] =
self.conjugate(self.conjugation_map[j])

        return self.conjugation_map

    def get_byte(self, bitplane):
        result = bytearray()

        for plane in bitplane:
            for bit in plane:
                byte = int(''.join(bit.astype(str)), 2)
                result.append(byte)

        return result

    def set_message(self):
        self.set_header()
        self.set_content()
        self.conjugate_content()

        self.bitplane += self.int_bitplane(len(self.header_bitplane))
        self.bitplane += self.int_bitplane(len(self.content_bitplane))

        self.bitplane += self.header_bitplane
        self.bitplane += self.content_bitplane

        conjugation_bitplane = self.conjugation_mapping()
        self.bitplane += self.int_bitplane(len(conjugation_bitplane))
        self.bitplane += conjugation_bitplane

        return self.bitplane

    def get_message(self, bitplane):
```

```
    header_length = self.get_int(bitplane.pop(0))
    content_length = self.get_int(bitplane.pop(0))
    temp = []

    for i in range(header_length + content_length):
      temp.append(bitplane.pop(0))

    cmap_length = self.get_int(bitplane.pop(0))

    for j in range(cmap_length):
      self.conjugation_map.append(bitplane.pop(0))

    temp = self.unconjugate_content(temp)

    for k in range(header_length):
      self.header_bitplane.append(temp.pop(0))

    for l in range(content_length):
      self.content_bitplane.append(temp.pop(0))

    self.filename = self.get_header()
    self.content = self.get_content()

    return self.filename, self.content, self.encrypted
```

Algorithm 1. messageBPCS Implementation

imageBPCS.py

```
import cv2
import math
import numpy as np
import random

from messageBPCS import messageBPCS
from vigenere import Vigenere

class imageBPCS():
  def __init__(self):
    self.block_size = 8

  def readImage(self, filename):
    try:
      image = cv2.imread(filename)

      self.image = image
      self.height, self.width, self.channels = image.shape
      self.size = self.width * self.height
      self.map = list(range(self.size))
    except Exception as exception:
      print(exception)
      print("Error while reading image file")

  def writeImage(self, filename):
    cv2.imwrite(filename, self.image)

  def complexity(self, block):
    count = 0
    height, width = block.shape

    for h in range(height - 1):
      for w in range(width - 1):
        if (block[h][w] != block[h + 1][w]):
          count += 1

        if (block[h][w] != block[h][w + 1]):
          count += 1

    return count / 112

  def to_bitplane(self, block):
    result = []

    for i in reversed(range(8)):
      bit = (block / (2 ** i)).astype(int) % 2
      result.append(bit)

    return result

  def to_byte(self, bit, plane):
    if (plane == 0):
      result = bit[plane]
    else:
      result = 2 * self.to_byte(bit, (plane - 1)) + bit[plane]

    return result

  def from_bitplane(self, bitplane):
    return self.to_byte(bitplane, (len(bitplane) - 1))

  def embed(self, path, key, threshold, encrypted, randomized):
    if (encrypted):
      vig = Vigenere(key)
      content = vig.encryptFile(path)
    else:
      content = open(path, "rb").read()

    filename = path.split('/')[-1]

    msg = messageBPCS(filename = filename, content = content,
key = key, threshold = threshold, encrypted = encrypted, randomized
= randomized, block_size = self.block_size)

    message = msg.set_message()

    if ((self.width * self.height * self.channels) < len(message)):
      raise Exception('Image is smaller than payload')

    i = 0

    while (i < len(message)):
      h = 0

      while ((h < (self.height - self.block_size + 1)) and (i <
len(message))):
        w = 0

        while ((w < (self.width - self.block_size + 1)) and (i <
len(message))):
          block = self.image[h:(h + self.block_size), w:(w +
self.block_size)]
          blocks = cv2.split(block)
          bitplane = [self.to_bitplane(block) for block in blocks]
          j = 0

          while ((j < len(bitplane)) and (i < len(message))):
            k = 0

            while ((k < len(bitplane[j])) and (i < len(message))):
              if (self.complexity(bitplane[j][k]) >= threshold):
                bitplane[j][k] = message[i]
                i += 1

              k += 1

            j += 1

          channel = [self.from_bitplane(plane) for plane in
bitplane]
          new_blocks = cv2.merge(channel)
          self.image[h:(h + self.block_size), w:(w + self.block_size)]
= new_blocks
          w += self.block_size

        h += self.block_size

    return self.image

  def extract(self, key, threshold):
    msg = messageBPCS(key = key, threshold = threshold,
block_size = self.block_size)
```

```
    message = []

    h = 0

    while (h < (self.height - self.block_size + 1)):
      w = 0

      while (w < (self.width - self.block_size + 1)):
        block = self.image[h:(h + self.block_size), w:(w +
self.block_size)]
        blocks = cv2.split(block)
        bitplane = [self.to_bitplane(block) for block in blocks]
        j = 0

        while (j < len(bitplane)):
          k = 0

          while (k < len(bitplane[j])):
            if (self.complexity(bitplane[j][k]) >= threshold):
              message.append(bitplane[j][k])

            k += 1

          j += 1

        w += self.block_size

      h += self.block_size

    filename, content, encrypted = msg.get_message(message)

    with open('result/image/' + filename, "wb") as f:
      f.write(content)

    if (encrypted):
      vig = Vigenere(key)
      vig.decryptFile('result/image/' + filename,
('result/image/decrypted_' + filename))

    return filename, content

  @staticmethod
  def psnr(image_one, image_two):
    mse = np.mean((image_one - image_two) ** 2)

    if (mse == 0):
      return 100

    max_pixel = 256.0
    psnr = 20 * math.log10(max_pixel / math.sqrt(mse))

    return psnr
```

Algorithm 2. imageBPCS Implementation

B. ECDSA Algorithm

Here is an implementation of the ECDSA algorithm in Javascript.

ECDSA.js

```
/* All elements of array must be in value 0-255 */
/**
 * Requirements:
 * ECMAScript 2015
 */

let string = require('./util/string')
let utils = require('./util/utils')
let keccak = require('./Keccak')
let ec = require('./EllipticCurve')

const byteReduction = 0n
const binReduction = 1n

module.exports = {
  ECDSA: class {
    /**
     *
     * @param {BigInt} a
     * @param {BigInt} b
     * @param {BigInt} p
     * @param {Array} basePointNum
     * @param {BigInt} n
     */
    constructor(a, b, p, basePointNum, n) {
      // DEFAULT NIST-192
      if (a === undefined || b === undefined || p === undefined) {
        let p =
BigInt(6277101735386680763835789423207666416083908700390324961279n)
        let n =
BigInt(6277101735386680763835789423176059013767194773182842284081n)
        let a = BigInt(-3n)
        let b =
BigInt(0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1)
        let g =
[BigInt(0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012),

BigInt(0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811)]
        this.curve = new ec.EllipticCurve(a, b, p, g)
        this.curve.setOrderExplicit(n)
      }
      // BASE POINT NUM EXPLICIT
      else if (isNaN(basePointNum)) {
        this.curve = new ec.EllipticCurve(a, b, p, basePointNum)
        if (n === undefined) {
          this.curve.setOrder();
        } else {
          this.curve.setOrderExplicit(n)
        }
      }
      // ONLY A, B, P, and NUMBER G
      else {
        this.curve = new ec.EllipticCurve(a, b, p)
        this.curve.setBasePointNumber(basePointNum)
      }
    }

    setKeyRandom() {
      let binCount = this.curve.binSize - binReduction
      this.privateKey = utils.getRandomIntRange(binCount, 2n **
binCount + 2n, (this.curve.n - 2n))
      this.privateKeyHex = this.privateKey.toString(16)
      this.publicKey =
this.curve.multiplyGraphPoint(this.curve.base, this.privateKey)
      this.publicKeyHex = this.publicKey[0].toString(16) + '' +
this.publicKey[1].toString(16)
    }

    /**
     *
     * @param {BigInt} privateKey
     */
    setPrivateKey(privateKey) {
      this.privateKey = privateKey
      this.privateKeyHex = this.privateKey.toString(16)
    }

    /**
     *
     * @param {Array} publicKey
     */
    setPublicKey(publicKey) {
      this.publicKey = publicKey
      this.publicKeyHex = this.publicKey[0].toString(16) + '' +
this.publicKey[1].toString(16)
    }
```

```
    /**
     *
     * @param {String} privateKeyHex
     */
    setPrivatekeyHex(privateKeyHex) {
      this.privateKeyHex = privateKeyHex
      this.privateKey = BigInt('0x' + privateKeyHex)
    }

    /**
     *
     * @param {String} publicKeyHex
     */
    setPublicKeyHex(publicKeyHex) {
      this.publicKeyHex = publicKeyHex
      let splittedPublic = publicKeyHex.split(" ")
      this.publicKey = [BigInt('0x' + splittedPublic[0]), BigInt('0x' +
splittedPublic[1])]
    }

    setKeyHex(privateKeyHex) {
      this.privateKeyHex = privateKeyHex
      this.privateKey = BigInt('0x' + privateKeyHex)
      this.publicKey =
this.curve.multiplyGraphPoint(this.curve.base, this.privateKey)
      this.publicKeyHex = this.publicKey[0].toString(16) + ' ' +
this.publicKey[1].toString(16)
    }

    initiateK(privateKey) {
      let binCount = this.curve.binSize - binReduction
      let k = utils.getRandomIntRange(binCount, 2n ** binCount,
privateKey)

      while (k >= privateKey) {
        k = utils.getRandomIntRange(binCount, 2n ** binCount,
privateKey)
      }

      let qa = this.curve.multiplyGraphPoint(this.curve.base, k)
      return [k, qa]
    }

    validatePublicKey(publicKey) {
      if (publicKey[0] === -1 && publicKey[1] === -1) {
        return false;
      }
      let mulPub = this.curve.multiplyGraphPoint(publicKey,
this.curve.N)

      if (mulPub[0] === -1 && mulPub[1] === -1) {
        return false;
      }
      return true;
    }

    sign(message, privateKey, hexedKey = false, hexedOutput =
false) {
      let s = 0n
      let r = 0n

      if (hexedKey) {
        privateKey = BigInt('0x' + privateKey)
      }

      while (s === 0n) {
        let initiate = this.initiateK(privateKey)
        let k = initiate[0]
        let qa = initiate[1]

        r = utils.mod(qa[0], this.curve.n)

        while (r === 0n) {
          initiate = this.initiateK()
          k = initiate[0]
          qa = initiate[1]
          r = utils.mod(qa[0], this.curve.n)
        }

        let invK = utils.modInverse(k, this.curve.n)

        if (!isNaN(Number(invK))) {
          let hashed = keccak.hash(message,
Number(this.curve.binSize / 8n))
          let e = BigInt(utils.strToHex(hashed))
          let sumEDR = utils.mod(e + privateKey * r, this.curve.n)
          s = utils.mod((invK * sumEDR), this.curve.n)

          if (isNaN(Number(utils.modInverse(s, this.curve.n)))) {
            s = 0n
          }
        }
      }

      if (!hexedOutput) {
        return [r, s]
      } else {
        return (r.toString(16) + '|' + s.toString(16))
      }
    }

    verify(message, signature, publicKey, hexedKey = false,
hexedSign = false) {
      let r = 0
      let s = 0

      if (hexedKey) {
        let splitted = publicKey.split(" ")
        publicKey = [BigInt('0x' + splitted[0]), BigInt('0x' +
splitted[1])]
      }

      if (!hexedSign) {
        r = signature[0]
        s = signature[1]
      } else {
        let splitted = signature.split("|")
        r = BigInt('0x' + splitted[0])
        s = BigInt('0x' + splitted[1])
      }

      if (r >= this.curve.n || r <= 0 || s >= this.curve.n || s <= 0) {
        return false
      }

      let hashed = keccak.hash(message,
Number(this.curve.binSize / 8n))
      let e = BigInt(utils.strToHex(hashed))
      let w = utils.modInverse(s, this.curve.n)
      let u1 = utils.mod((e * w), this.curve.n)
      let u2 = utils.mod((r * w), this.curve.n)
      let u1g = this.curve.multiplyGraphPoint(this.curve.base, u1)
      let u2q = this.curve.multiplyGraphPoint(publicKey, u2)
      let x = this.curve.sumGraphPoint(u1g, u2q)

      if (x[0] === -1 || x[1] === -1) {
        return false
      }

      let x1 = x[0]
      let v = utils.mod(x1, this.curve.n)

      return (v === r)
    }
  }
}
```

Algorithm 3. ECDSA Implementation

IV. Proposed File Security System

First of all, the COVID-19 test result file will be digitally signed using the ECDSA algorithm using the public and private keys of the party or institution that issued the file. After being signed, the COVID-19 test result file is inserted into the logo image of the institution that issued the file with a stego-key known only to the authorities.

Private key	d2144a6e3ef71a1e5fef25c51696c07dda1d6f641ba1ef34
Public key	2197b5e2bb8eee1ea8ebc122417612ffc9b2f957aab4a3c9 bb14232ecaa0020bc899ee8ee4e85b44bf0a9e3bb1da781a

Table 1. ECDSA Public-Private Key Example

When the examiner wants to check the file, the examiner will extract the file from the image using a stego key and then verify it using the public key of the institution that issued the file.

V. Conclusion

The use of steganography and digital signatures on the covid test result files has advantages in the security aspect, but along with that there are challenges in implementation and supporting elements to ensure that this file security system can work properly.

VI. Acknowledgment

First of all, the authors express gratitude and gratitude to God Almighty for His love and blessings so that the author can still take lessons in Informatics Engineering, Bandung Institute of Technology.

The author would also like to thank both parents and families who always pray for and support what the author is doing. The author also wants to thank Mr. Rinaldi Munir as a lecturer in the Cryptography course who helped the author understand the material presented and the preparation of this paper.

Finally, the author would also like to say many thanks to friends, especially from the Jiwa-Jiwa Yang Tersiksa group who always support and encourage the author in his studies at Bandung Institute of Technology, including in the preparation of this paper.

References

[1] Johnson, Don, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). https://www.cs.miami.edu/home/burt/learning/Csc609.142/ecdsa-cert.pdf (diakses tanggal 20 Desember 2020)

[2] Limbong, Sepriani T. 2020. Virus Corona (COVID-19). https://www.klikdokter.com/penyakit/coronavirus (diakses tanggal 20 Desember 2020)

[3] Munir, Rinaldi. 2020. Bitplane Complexity Segmentation (BPCS). http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/BPCS-2020.pdf (diakses tanggal 20 Desember 2020)

[4] Munir, Rinaldi. 2020. Digital Signature Standard (DSS). http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/DSS-2020.pdf (diakses tanggal 20 Desember 2020)

[5] Munir, Rinaldi. 2020. Steganografi (Bagian 1). http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Steganografi-Bagian1-2020.pdf (diakses tanggal 20 Desember 2020)

[6] Munir, Rinaldi. 2020. Steganografi (Bagian 2). http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Steganografi-Bagian2-2020.pdf (diakses tanggal 20 Desember 2020)

[7] Munir, Rinaldi. 2020. Steganografi (Bagian 3). http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Steganografi-Bagian3-2020.pdf (diakses tanggal 20 Desember 2020)

[8] Munir, Rinaldi. 2020. Tanda-tangan digital (digital signature). http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Tanda-tangan-digital-2020.pdf (diakses tanggal 20 Desember 2020)

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 December 2020

Vincent Budianto 13517137