

# **Real Estate Blockchain Documentation**

## Table of contents

5.0 Implementation Technique .....	1
5.1 Blockchain Concept .....	1
5.2 Hashing Algorithm.....	6
5.3 Cryptography Algorithm.....	10
5.3.1 Symmetric .....	10
5.3.2 Asymmetric.....	14
5.4 Digital Signature Algorithm .....	21
5.5 Immutability Technique.....	26

## 5.0 Implementation Technique

The solution was developed utilizing blockchain principles and chaining techniques. This comprises cryptography algorithms (both symmetric and asymmetric), hashing algorithms, digital signatures and also immutability techniques.

### 5.1 Blockchain Concept

The blockchain is composed of a few blocks, each of which contains transactions records, in which when each block is added to the blockchain, a ledger is created (Hayes, 2022). Furthermore, there additionally exists a timestamp for each block, as well as prior and current hash values. The genesis block is the very first block that holds the first header and usually without any transaction in it. The utilization of Distributed Ledger Technology (DLT) is a key component of blockchain (Frankenfield, 2023). It is the location where every network participant is connected to the distributed ledger. By using a shared ledger, the transaction which is recorded inside the ledger is only recorded once, which prevents data duplication. The use of DLT in blockchain assists in eliminating difficulties like unidentified participants between transactions, which supervises the transactions that are being performed. DLT can keep track of the whole transaction and is approved by the entire peer network. Below is the comparison between traditional ledger and blockchain ledger (TutorialsPoint, n.d.).

Attribute for Ledger	Traditional	Blockchain
Encryption	Less security mechanism implemented	More secure since it is encrypted with cryptographic encryption
Data Integrity	Data can be untrustworthy since it can be manually added or modified	Data cannot be modified, so it should have high integrity
Transparency	It is only partial, or even no transparency at all	Everyone on network has full transparency
Authority	Usually centralized	Distributed and decentralized on the nodes
Immutability	No	Yes

```

package blockchain;

import ...5 lines

/**
 *
 * @author Brian
 */
public class Block implements Serializable{
    public Header header;
    public PropertyData record;

    //constructor
    public Block( String prevHash, int index ) {
        header = new Header();
        header.setPreviousHash(prevHash);
        header.setIndex(index + 1);
        header.setTimestamp( new Timestamp(
            System.currentTimeMillis() ).getTime() );
        byte[] blockByte = this.getBytes();
        String currHash = new String(blockByte);
        String blockHash = Hasher.hash( currHash, "SHA-256" );
        header.setCurrentHash( blockHash );
    }

    public Header getHeader() {
        return header;
    }

    public PropertyData getData() {
        return record;
    }

    public void setData(PropertyData realEstateList) {
        this.record = realEstateList;
    }

    //composition relations between block and header
    public class Header implements Serializable{

        private int index;
        private String currentHash;
        private String previousHash;
        private long timestamp;

        public String getCurrentHash() {
            return currentHash;
        }

        public void setCurrentHash(String currentHash) {
            this.currentHash = currentHash;
        }

        public String getPreviousHash() {
            return previousHash;
        }

        public void setPreviousHash(String previousHash) {
            this.previousHash = previousHash;
        }

        public long getTimestamp() {
            return timestamp;
        }

        public void setTimestamp(long timestamp) {
            this.timestamp = timestamp;
        }

        public int getIndex() {
            return index;
        }

        public void setIndex(int index) {
            this.index = index;
        }

        @Override
        public String toString() {
            StringBuilder sb = new StringBuilder("Header [");
            sb.append( "index="+this.index+", " );
            sb.append( "currentHash="+this.currentHash+", " );
            sb.append( "previousHash="+this.previousHash+", " );
            sb.append( "timestamp="+this.timestamp+", " );
            return sb.append("]").toString();
        }
    }
}

```

*Figure 5.1.1 Block.java Part 1*

```

@Override
public String toString() {
    return "Block [header="+ this.header +", transaction="+ this.record +"]";
}

private byte[] getBytes() {
    try(
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(baos);
    ) {
        out.writeObject( this );
        return baos.toByteArray();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

```

Figure 5.1.2 Block.java Part 2

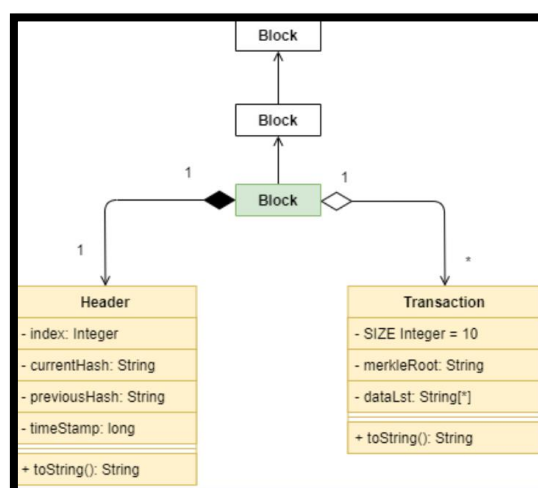


Figure 5.1.3 Block UML Concept (Kimapu, 2022)

The first thing to do when a developer wants to develop blockchain system is to create the block first. The first two figures above show the Block.java class. On the class declaration, there is “Serializable” keyword, which means that the class can be converted into a stream of bytes and stored in a file or transmitted over a network, which allows objects of the class to be written to an object output stream and read back from an object input stream. The block contains a header and body. The header contains index, current and previous hash address, and a timestamp at the creation of the block. As seen on figure 5.1.1, the current hash is created from the previous hash which is hashed with SHA-256. The following code is only a get set method for the header class. Header is defined here because it has a composition relationship with the block as seen on figure 5.1.3.

```

package blockchain;

import ...9 lines

/**
 *
 * @author Brian
 */
public class Blockchain {
    //create the blockchain file - unreadable by people
    private static final String CHAIN_FILE = "master/chain.bin";

    private static LinkedList<Block> DB = new LinkedList<>();

    //create ledger file for user to read
    private static final String LEDGER_FILE = "ledger.txt";

    public static void genesis() {
        Block genesis = new Block("0", -1);
        DB.add(genesis);
        Blockchain.persist();
        Blockchain.distribute();
    }

    public static void nextBlock(Block newBlock) {
        DB = get();
        DB.add(newBlock);
        Blockchain.persist();
    }

    private static void persist() {
        try {
            FileOutputStream fout = new FileOutputStream(CHAIN_FILE);
            ObjectOutputStream out = new ObjectOutputStream(fout); {
                out.writeObject(DB);
                System.out.println(">>> Master file updated!");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    public static LinkedList<Block> get() {
        try {
            FileInputStream fin = new FileInputStream(CHAIN_FILE);
            ObjectInputStream in = new ObjectInputStream(fin); {
                return (LinkedList<Block>) in.readObject();
            } catch (Exception e) {
                e.printStackTrace();
                return null;
            }
        }
    }

    public static void distribute() {
        String chain = new GsonBuilder().setPrettyPrinting().create().toJson(DB);
        System.out.println(chain);
        try {
            Files.write(Paths.get(LEDGER_FILE), chain.getBytes(), StandardOpenOption.CREATE);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Figure 5.1.4 Blockchain.java

Blockchain.java class is created in order to create the chain of blocks. First, it is important to create a bin file for the machine to read, and txt file for developer to read, which basically contains the same thing with different format. The data structure chosen by the developer is a linked list, since it is the data structure that has the most resemblance to blockchain (has next and previous address). The function genesis() is created for the first block of the system. The function nextBlock() is used to create next block on the chain. The function persist() has the job to write in the bin file, while distribute() function used to write on txt file.

```
package blockchain;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author Brian
 */
public class PropertyData implements Serializable{
    private final int SIZE = 8;
    //MerkleRoot is a hash value according to dataLst
    private String merkleRoot = "helloworld";
    private List<String> dataLst;

    //constructor
    public PropertyData() {
        dataLst = new ArrayList<>( SIZE );
    }

    public void setMerkleTree(String root) {
        this.merkleRoot = root;
    }

    //add-operation
    public void add( String tranx ) {
        dataLst.add(tranx);
    }

    @Override
    public String toString() {
        return "Transaction ["+ dataLst +"]";
    }

    public List<String> getDataLst() {
        return dataLst;
    }
}
```

*Figure 5.1.5 PropertyData.java*

This class is the body of the block, which contains size, Merkle Root, and also the data list of the transaction, such as buyer name, seller name, property type, transaction date, price, location, author name, and author's digital signature.

```

{
  {
    "header": {
      "index": 0,
      "currentHash": "745cd9bbf55dd41b81a6647dcefe5b794ae3e6cb0516b3337042a5c1bf647db59",
      "previousHash": "0",
      "timestamp": 1679800355656
    }
  },
  {
    "header": {
      "index": 1,
      "currentHash": "ef76306058808ec652986a963406eb8be5b21af5cd94ab5ee6dd4f89d8bf0db7",
      "previousHash": "745cd9bbf55dd41b81a6647dcefe5b794ae3e6cb0516b3337042a5c1bf647db59",
      "timestamp": 1679800355748
    },
    "record": {
      "SIZE": 8,
      "merkleRoot": "57ba4a829108eed3cbl7a886c61c9c8e2748ccd9a44c069793e5549bb92fdcce",
      "dataLst": [
        "QzH/xYlh7tEWxiRj1NL14eT1j9Vy/FvucKskFGJm2XGWUeAcB7gv3da+q22XKLyPz1lLuM1+np2JbxOVvbJLsT2FkoulbeofKxuQEfHEc4cRUodSA9TWxtUMiTpDi0",
        "TDcPIJGg05ynTMyHBQ2HqPPNAy617Q15GbTBC3VaTSVdqOVRfnvv+mQFnfjuYDcL99DEzcZUnWJ8HSuOOnOmF4mNYRVMn8hPrUIZuS3I0exzvBgPJ4BpEPT+jUMsfX1",
        "house",
        "1000",
        "200000",
        "KY7Nvgb/iUG+r1l4KU8oO3736NalZ7nznqdsLmUrlM6ENZCcTQ/X2WVmr4/t2nRd4VnRI4A2wvs3xuGbVag1aj140szF9xSchjDe5g5SposgP5S21EoO4qoyg4S2HD",
        "brian",
        "GKRvxT5ktMLQa8u3zpoGvxeZe+H3xquGGzqnqs4DhdM7UgXzY/sj7TjLZaEbzmculzmQ8owotiQx3heTUPtHSeqgp7HkmyMc4JRAKR9FtA7mh2pqLvVOZjckL9B2L1"
      ]
    }
  }
},

```

*Figure 5.1.6 Blockchain Output*

As for the output of the system, the one highlighted on blue is the genesis block with index 0 and previous hash 0. Genesis block does not contain anything other than header. The next block's previous hash reference to the genesis block's hash and contains the record with some encrypted data.

## 5.2 Hashing Algorithm

Hashing is the process of converting any size message or data into a fixed-length string of characters that reflects the original message uniquely and irrevocably (Narrative, n.d.). Hashing is commonly used in computer science for data security applications such as password storage, message integrity, and digital signatures. The hash value may be used to check data integrity by comparing hash values before and after transmission or storage. Hashing is a critical data security method because it allows sensitive information to be stored or conveyed in a secure and irreversible manner, preventing unauthorized data access or change. For example, there are md5 algorithm, SHA-0, SHA-1, and SHA2 algorithms (256, 512). Hashing is actually not secure enough, so salt should be implemented as well (Arias, 2021). A salt is a random value added to the input data before hashing, which ensures that even if two users have the same password, their hash values will be different due to the different salts added. For a quick comparison, below is a table about different hashing algorithm.



Attribute	MD5	SHA-0	SHA-1	SHA-2
Benefit	Fast and efficient hash computation	None	Widely adopted and compatible with existing systems	Offers stronger security and resistance to collision attacks
Security Level	Broken, considered insecure	Broken, considered insecure	Broken, considered insecure	Currently considered secure
Created Date	Introduced in 1992, widely used in the 2000s	Introduced in 1993, withdrawn shortly after	Introduced in 1995, standardized in 1999	Introduced in 2001, standardized in 2002

SHA-256 was selected to be utilized given that it takes a smaller RAM value and so improves block speed. Furthermore, SHA-256 is currently a trustworthy method since it is free of flaws when compared to other hashing algorithms which have been broken by hackers and currently considered as insecure algorithms.

```

package hashing;

import java.security.MessageDigest;
import org.apache.commons.codec.binary.Hex;

/**
 *
 * @author Brian
 */
public class Hasher {
    public static String sha256(String data) {
        return hash(data, "SHA-256");
    }

    public static String hash(String data, String algo)
    {
        String hash = null;
        try {
            MessageDigest md = MessageDigest.getInstance( algo );
            md.update( data.getBytes() );

            //use salt to add security
            md.update( Salt.generate() );

            //get the byte array value
            byte[] hashByte = md.digest();

            hash = String.valueOf( Hex.encodeHex(hashByte) );
        } catch (Exception e) {
            e.printStackTrace();
        }
        return hash;
    }
}

```

*Figure 5.2.1 Salt.java*

```

package hashing;

import java.security.SecureRandom;

/**
 *
 * @author Brian
 */
public class Salt {
    public static byte[] generate() {
        //creating rand value...
        SecureRandom sr = new SecureRandom();
        byte[] b = new byte[16];
        sr.nextBytes(b);
        return b;
    }
}

```

*Figure 5.2.2 Hasher.java*

Two figures above show the hashing system that is used for this system. The Hasher class is a Java class that provides a method for hashing data using the SHA-256 algorithm. The function sha256() returns a SHA-256 hash value. The hash() function takes string and hashing algorithm and then returns the hash value with the added salt to make it more secure.

```

//constructor
public Block( String prevHash, int index ) {
    header = new Header();
    header.setPreviousHash(prevHash);
    header.setIndex(index + 1);
    header.setTimestamp( new Timestamp(
        System.currentTimeMillis() ).getTime() );
    byte[] blockByte = this.getBytes();
    String currHash = new String(blockByte);
    String blockHash = Hasher.hash( currHash, "SHA-256" );
    header.setCurrentHash( blockHash );
}

```

*Figure 5.2.3 Implementation of Hashing Part 1*

```

[
  {
    "header": {
      "index": 0,
      "currentHash": "745cdbbf55dd41b81a6647dcefe5b794ae3e6cb0516b3337042a5c1bf647db59",
      "previousHash": "0",
      "timestamp": 1679800355656
    }
  },
  {
    "header": {
      "index": 1,
      "currentHash": "ef76306058808ec652986a963406eb8be5b21af5cd94ab5ee6dd4f89d8bf0db7",
      "previousHash": "745cdbbf55dd41b81a6647dcefe5b794ae3e6cb0516b3337042a5c1bf647db59",
      "timestamp": 1679800355748
    }
  }
]

```

*Figure 5.2.4 Output Hashing Part 1*

Based on figure 5.2.3, the block's current hash is generated from block's previous hash that has been hashed with the SHA-256 algorithm. Figure 5.2.4 shows the output of the hashing algorithm.

```
private List<String> genTranxHashLst(List<String> tranxLst) {
    List<String> hashLst = new ArrayList<>();
    int i = 0;
    while( i < tranxLst.size() ) {
        String left = tranxLst.get(i);
        i++;

        String right = "";
        if( i != tranxLst.size() ) right = tranxLst.get(i);

        String hash = Hasher.sha256(left.concat(right));
        hashLst.add(hash);
        i++;
    }
    return hashLst;
}
```

*Figure 5.2.5 Implementation of Hashing Part 2*

```
{
  "header": {
    "index": 1,
    "currentHash": "ef76306058808ec652986a963406eb8be5b21af5cd94ab5ee6dd4f89d8bf0db7",
    "previousHash": "745cdbbf55dd41b81a6647dcefe5b794ae3e6cb0516b3337042a5c1bf647db59",
    "timestamp": 1679800355748
  },
  "record": {
    "SIZE": 8,
    "merkleRoot": "57ba4a829108eed3cb17a886c61c9c8e2748ccd9a44c069793e5549bb92fdcce",
    "dataLst": [
      "QzH/xYlh7tEWxiRj1NL14eTlj9Vy/FvucKskFGJm2XGWUeAcB7gv3da+qZ2XKLyPzI1LuM1+np2JbxOVvbJLs",
      "TDcPIJGg05ynTMyHBQ2HqPPNAy6l7Q15GbTBC3VaTSVdqOVRFnvv+mQFnfjuYDcL99DEzcZUnWJ8HSuOOOnOmP",
      "house",
      "1000",
      "200000",
      "KY7Nvgb/iUG+r1l4KU8oO3736NalZ7nrzNqdsLmUrlM6ENZCcTQ/X2WVmr4/t2nRd4VnRI4A2wvs3xuGbVag1",
      "brian",
      "GKRvxT5ktMLQa8u3zpoGvxeZe+H3xquGGzqnqs4DhdM7UgXzY/sj7TjLZaEbzmcuLzmQ8owotiQx3heTUPtHS"
    ]
  }
}
```

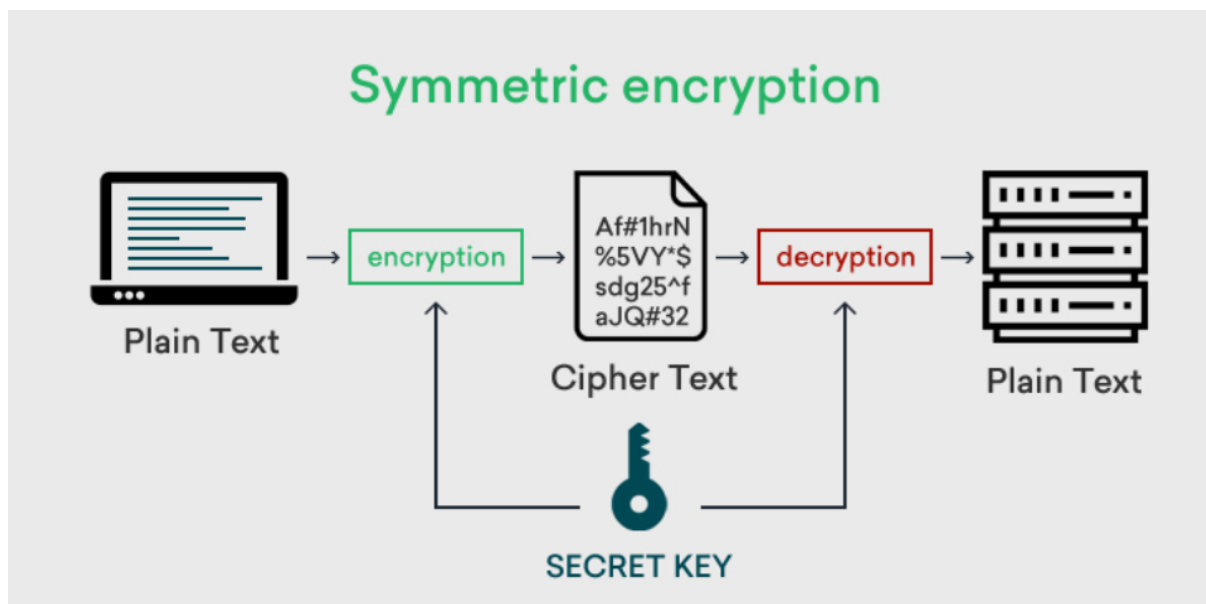
*Figure 5.2.6 Output Hashing Part 2*

In the 2 figures above, hashing algorithm is used in the Merkle root, which will be explained more thoroughly on chapter 5.5 immutability technique. Here, the Merkle root is also hashed with SHA-256 algorithms.

## 5.3 Cryptography Algorithm

Cryptographic algorithms are collections of mathematical rules and methods that are utilized to safeguard and secure data sent across networks or kept on devices (Fruhlinger, 2022). These algorithms are crucial in cryptography because they ensure that confidential data stays confidential. There are 2 example algorithms, such as symmetric-key algorithms and asymmetric-key algorithms. Cryptographic algorithms have become indispensable in modern communication, with applications including online banking, e-commerce, encrypted texting, and others. The difference between hashing a cryptography is that one is reversible while the other is not. Cryptography is reversible because it serves as a safe communication way, so that people who do not have access cannot read the encrypted plain text (Economic Times, 2023).

### 5.3.1 Symmetric



*Figure 5.3.1.1 Symmetric Encryption (ClickSSL, 2022)*

Based on the figure above, symmetric encryption only uses 1 key for encrypting and decrypting process, which means that sender and receiver must possess the same key to do this kind of encryption (Burnett & Foster, 2004). Symmetric encryption algorithms typically fall into two categories, which are stream cipher and block cipher. Block cipher means that the data is divided into a fixed-size blocks and then encrypt it one by one using the key, while stream cipher encrypt the data in the form of continuous stream bits, and this kind of technique is useful for real time data transmission (GeeksforGeeks, 2023).

As everything stands, this encryption system has its advantages and disadvantages. The advantages are that it has a quick speed and efficiency in processing the encryption and decryption, because it only uses a single key to do both tasks. As for the disadvantage, users need to think more about key management (Developer, 2022). This key must be put into safety because if it leaks, then attackers can get the data easily. One way to go around this problem is to use a key derivation function, which means that the key is generated on a session, and when it closes, the key is no longer usable.

```
package cryptography;

import java.security.Key;
import java.util.Base64;
import javax.crypto.Cipher;

/**
 *
 * @author Brian
 */
public class Symmetric extends Cryptography{
    public Symmetric(String algorithm) {
        super(algorithm);
    }

    @Override
    public String encrypt(String data, Key key) throws Exception {
        String cipherText = null;
        cipher.init(Cipher.ENCRYPT_MODE, key);

        //encrypt
        byte[] cipherBytes = cipher.doFinal(data.getBytes());
        cipherText = Base64.getEncoder().encodeToString(cipherBytes);
        return cipherText;
    }

    @Override
    public String decrypt(String cipherText, Key key) throws Exception {
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] cipherBytes = Base64.getDecoder().decode(cipherText);

        //decrypt
        byte[] dataBytes = cipher.doFinal(cipherBytes);
        return new String(dataBytes);
    }
}
```

*Figure 5.3.1.2 Symmetric.java*

As seen above, the encryption and decryption process is mostly done with the help of javax.crypto.Cipher API from Java Cryptography Architecture package. It is an essential component of Java's encryption and decryption capability, allowing developers to conduct symmetric and asymmetric encryption, decryption, and key creation. In encrypt() function,

the string data is encrypted based on the key parameters that are passed on the function calling with byte array value. Similarly, decrypt() function is also decrypted based on the key parameters that are passed on the function calling with byte array value. Below is the implementation of symmetric encryption in the solution.

```
public boolean registerAccount() throws Exception {
    Key key = new SecretKeySpec(Arrays.copyOf>Password.getBytes(), 16), "AES");
    Cryptography encryptor = new Symmetric("AES");
    String encryptedText = encryptor.encrypt(Username, key);

    if (!Files.exists(filePath)) {
        Files.createFile(filePath);
    }
    if (!checkUsername(Username)) {
        Files.write(filePath, (encryptedText + ":" + Username + "\n").getBytes(), StandardOpenOption.APPEND);
        return true;
    }
    return false;
}
```

*Figure 5.3.1.3 Register Function*

```
public boolean login() {
    String accountdata = "";

    Boolean access = false;

    Cryptography decryptor = new Symmetric("AES");

    Key key = new SecretKeySpec(Arrays.copyOf>Password.getBytes(), 16), "AES");

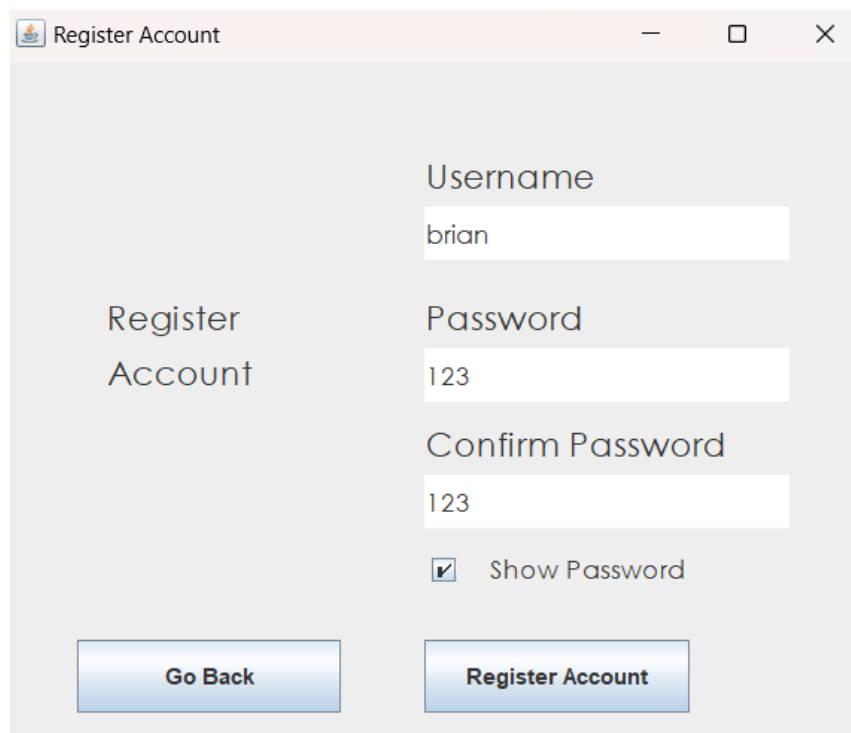
    try {
        Scanner scanner = new Scanner(filePath);
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            if (line.contains(Username)) {
                accountdata = line;
            }
        }
        scanner.close();
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Inccorect Username Password");
        return false;
    }

    String[] encrpytedData = accountdata.split(":");

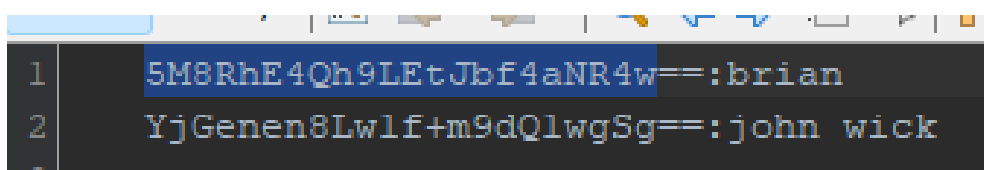
    if (encrpytedData.length > 0) {
        String decryptedUsername = "";
        try {
            decryptedUsername = decryptor.decrypt(encrpytedData[0], key);
        } catch (Exception e) {
            return false;
        }
        if (decryptedUsername.equals(Username)) {
            return true;
        }
    }
    return false;
}
```

*Figure 5.3.1.4 Login Function*

As seen in the code above on figure 5.3.1.3, when user registers a new account, it is being encrypted with AES on symmetric encryption. The thing that is being encrypted is the username, and the password acts as the secret key to encrypt the username. Then the username and encrypted username is stored on a txt file. On figure 5.3.1.4, the login function also uses the same AES algorithm to decrypt the username. If a match is found, then the user is allowed to login to the system.

A screenshot of a web application window titled "Register Account". The window has a light gray background. On the left side, the text "Register Account" is displayed in a dark blue font. On the right side, there are three input fields: "Username" with the value "brian", "Password" with the value "123", and "Confirm Password" with the value "123". Below the "Confirm Password" field is a checkbox labeled "Show Password" which is checked. At the bottom of the form, there are two buttons: "Go Back" and "Register Account".

*Figure 5.3.1.5 Login UI*

A screenshot of a text file named "Accounts.txt". The file contains two lines of text. The first line is "5M8RhE4Qh9LEtJbf4aNR4w==:brian" and the second line is "YjGenen8Lwlf+m9dQlwgSg==:john wick". The first line is highlighted in blue.

*Figure 5.3.1.6 Accounts TXT file*

As demonstration example on figure 5.3.1.5, the username is “brian”, and the password, which acts as the key is “123”. When the system registers the user, it will encrypt “brian” with AES using “123” as the secret key. When user wants to login, it will check if the value of “5M8RhE4Qh9LEtJbf4aNR4w==” is equal to “brian” after it is decrypted with “123” as the key.

### 5.3.2 Asymmetric

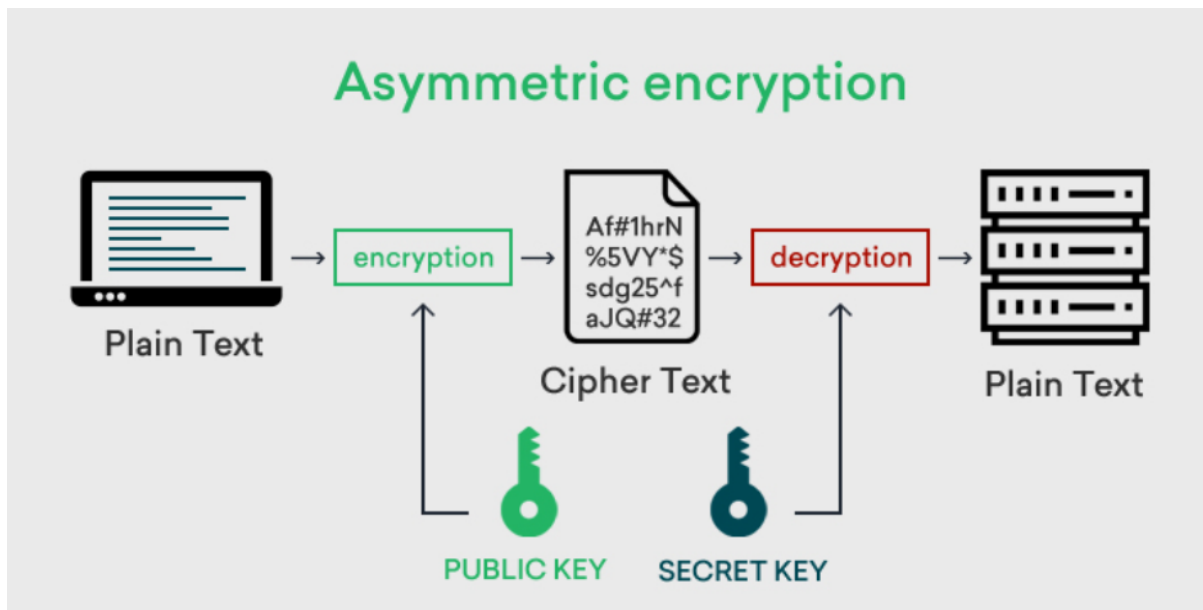


Figure 5.3.2.1 Asymmetric Encryption (ClickSSL, 2022)

Based on the figure above, it is clear what differentiates symmetric with asymmetric encryption. On figure 5.3.1.1, symmetric encryption only uses 1 same key for encryption and decryption, but on figure above, asymmetric use a pair of keys, public key for encryption, and private key for decryption (Security, 2021). The algorithm used in asymmetric is also different. Some popular choices are Rivest-Shamir-Adleman (RSA), Elliptic Curve Cryptography (ECC), and Diffie-Hellmann algorithm. As a result, the receiver of this procedure will decrypt the data to its original condition using the pair of public key (private key) that was used for encryption. Furthermore, asymmetric encrypts only the data being sent and the public key may be viewed by anyone, but symmetric encrypts the data and both public and private keys are communicated.

Just like everything else, there are pros and cons to this cryptography technique as well. The advantage of asymmetric encryption is that it is more secure in a way, since private key does not need to be distributed among people, unlike in the case of symmetric encryption. But in exchange for this benefit, the drawback is that it needs more computational capability and taking more resource since there are 2 keys, so it maybe not a great choice for some applications (Miller, 2017). There are also some ways to attack this, which is called “man-in-the-middle” attack, which means that invader will change the public key used for encryption with their own public key, which resulting in compromised data since invader now can access



the data with their own private key. Below is the implementation of asymmetric encryption on the system, which is used when adding lists in the block body into the blockchain.

```
package cryptography;

import java.security.Key;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.util.Base64;
import javax.crypto.Cipher;

/**
 *
 * @author Brian
 */
public class Asymmetric extends Cryptography{
    public Asymmetric(String algorithm) {
        super(algorithm);
    }

    @Override
    public String encrypt(String data, Key key) throws Exception {
        String cipherText = null;
        cipher.init(Cipher.ENCRYPT_MODE, (PublicKey) key);

        //encrypt
        byte[] cipherBytes = cipher.doFinal( data.getBytes() );
        cipherText = Base64.getEncoder().encodeToString(cipherBytes);
        return cipherText;
    }

    @Override
    public String decrypt(String cipherText, Key key) throws Exception {
        cipher.init(Cipher.DECRYPT_MODE, (PrivateKey) key);
        byte[] cipherBytes = Base64.getDecoder().decode( cipherText );

        //decrypt
        byte[] dataBytes = cipher.doFinal( cipherBytes );
        return new String( dataBytes );
    }
}
```

*Figure 5.3.2.2 Asymmetric.java*

As previously stated, the encryption and decryption processes are largely carried out via the javax.crypto.Cipher API from the Java Cryptography Architecture package. It is an important part of Java's encryption and decryption capabilities, allowing developers to perform symmetric and asymmetric encryption, decryption, and key generation. The string data is encrypted in the encrypt() method with public key. Similarly, the decrypt() method is decrypted depending on the private key.

```

package cryptography;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;

/**
 *
 * @author Brian
 */
public class KeyGenerator {
    private KeyPairGenerator keygen;
    private KeyPair keyPair;

    public KeyGenerator() {
        try {
            keygen = KeyPairGenerator.getInstance("RSA");
            keygen.initialize(1024);
        } catch (NoSuchAlgorithmException e) {
        }
    }

    private static void placeDirectory(byte[] keyBytes, String path) {
        File file = new File(path);
        file.getParentFile().mkdirs();
        try {
            Files.write(Paths.get(path), keyBytes, StandardOpenOption.CREATE);
        } catch (IOException e) {
        }
    }

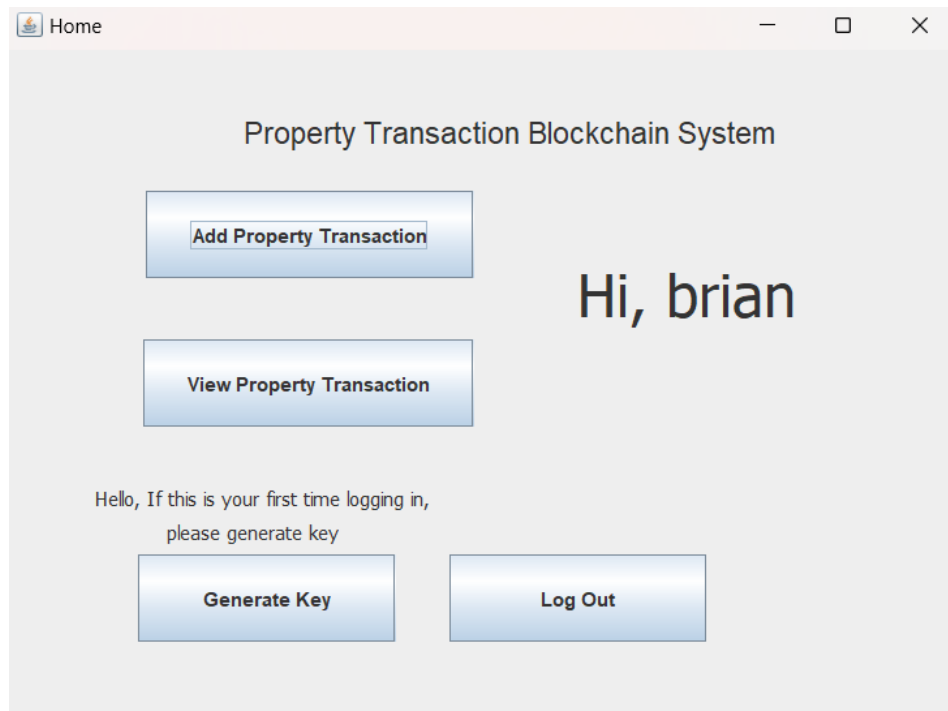
    public static void create(String path) {
        KeyGenerator myKeyMaker = new KeyGenerator();
        // generate keypair
        myKeyMaker.keyPair = myKeyMaker.keygen.generateKeyPair();
        // generate public key
        PublicKey pubKey = myKeyMaker.keyPair.getPublic();
        // generate private key
        PrivateKey prvKey = myKeyMaker.keyPair.getPrivate();

        //place it into the folder
        placeDirectory(pubKey.getEncoded(), "KeyPair/" + path + "/PublicKey");
        placeDirectory(prvKey.getEncoded(), "KeyPair/" + path + "/PrivateKey");
    }
}

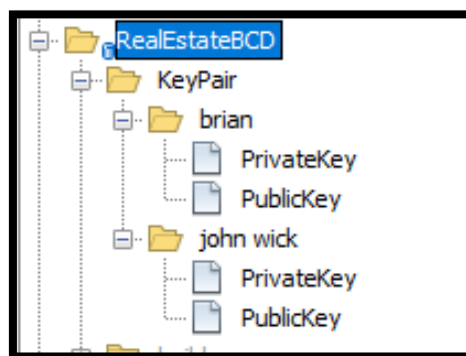
```

*Figure 5.3.2.3 KeyGenerator.java*

The code above is used for generating key for each user. It is also being mostly helped by the API, such as KeyPair, KeyGenerator, PrivateKey, and PublicKey, so that developer can easily develop the system. The key is generated with RSA algorithms. The placeDirectory() function is used to create a new KeyPair folder based on the current logged in user. The create() function is used to generate public and private key, and the place the keys in corresponding folder to be saved.



*Figure 5.3.2.4 Generate Key UI*



*Figure 5.3.2.5 Generated Files*

Based on figure 5.3.2.4 and 5.3.2.5, the key is generated successfully. When user, in this case “brian” click on the “Generate Key” button, it will directly create a KeyPair folder (if not exist yet), and create folder “brian” under it, with the private and public key saved in it. The file saved here is not plain text, not readable by human, so it is secure.

*Figure 5.3.2.6 Add Data UI*

On figure above, the UI for adding data into the blockchain can be seen. Users just need to key in buyer name, seller name, property type, price, location, and transaction date on the corresponding boxes, and the system will process it to add the block into the chain.

```
private KeyPair getKeyPair(String data) throws IOException, NoSuchAlgorithmException, InvalidKeySpecException {
    // Read Public Key.
    File filePublicKey = new File("KeyPair/" + data + "/PublicKey");
    FileInputStream fis = new FileInputStream("KeyPair/" + data + "/PublicKey");
    byte[] encodedPublicKey = new byte[(int) filePublicKey.length()];
    fis.read(encodedPublicKey);
    fis.close();

    // Read Private Key.
    File filePrivateKey = new File("KeyPair/" + data + "/PrivateKey");
    fis = new FileInputStream("KeyPair/" + data + "/PrivateKey");
    byte[] encodedPrivateKey = new byte[(int) filePrivateKey.length()];
    fis.read(encodedPrivateKey);
    fis.close();

    // Generate KeyPair.
    KeyFactory keyFactory = KeyFactory.getInstance("RSA");
    X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(
        encodedPublicKey);
    PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);

    PKCS8EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(
        encodedPrivateKey);
    PrivateKey privateKey = keyFactory.generatePrivate(privateKeySpec);

    return new KeyPair(publicKey, privateKey);
}
```

*Figure 5.3.2.7 Get Key Pair for Encryption and Decryption*

```

private void addBlockchain() {
    String buyerName = buyerlbl.getText();
    String sellerName = sellerlbl.getText();
    String propType = (String) cmbtype.getSelectedItem();

    String D=String.valueOf(cmbDate.getSelectedItem());
    int Month=cmbMonth.getSelectedIndex();
    String M=String.valueOf(Month+1);
    String Y=String.valueOf(cmbYear.getSelectedItem());
    String app_date= D+"/"+M+"/"+Y; //get data from combo box to form a date and combine it

    String propPrice = pricelbl.getText();
    String propLoc = locationlbl.getText();

    //Asymmetric encryption
    KeyPair keyPair = null;
    try {
        keyPair = getKeyPair(username);
    } catch (Exception e) {
        System.out.println("Key Pair Not Found");
    }
    PublicKey pubKey = keyPair.getPublic();

    Asymmetric crypto = new Asymmetric("RSA");

    //store all the value into an array
    List<String> propertyList = new ArrayList<>();

    try {
        propertyList.add(crypto.encrypt(buyerName, pubKey));
        propertyList.add(crypto.encrypt(sellerName, pubKey));
        propertyList.add(propType);
        propertyList.add(crypto.encrypt(app_date, pubKey));
        propertyList.add(propPrice);
        propertyList.add(crypto.encrypt(propLoc, pubKey));
        propertyList.add(username);

    } catch (Exception e) {
        System.out.println("Cannot Add Data");
    }
}

```

*Figure 5.3.2.8 Encrypt Data*

The function on figure 5.3.2.7 is created to read the encoded bytes of the public key and the private key from their respective files using `FileInputStream`, and then uses these bytes to generate a `KeyPair` object using the RSA algorithm. The function returns `KeyPair` object containing both the public and private keys. Moving on to the figure 5.3.2.8, the `KeyPair` object is created, and then the current logged in username is used as parameter to get the key pair. In encrypting, only public key is used, so that is why `getPublic()` function is used. `Asymmetric` object is created with RSA algorithm parameter. The user input value is then used to encrypt the data (can be seen as highlighted blue on the figure).

```

"header": {
  "index": 1,
  "currentHash": "6666f8fa478e2d1a71e7a05aa28ef0e05abb0691f810f815033126ac7a92f1ba",
  "previousHash": "0b90f9d2c2cba6d95cde577362b379db69190971eed1402f725e6d9453715a18",
  "timestamp": 1679888533627
},
"record": {
  "SIZE": 8,
  "merkleRoot": "3a293986f8cla71ld440f4d0416eada9a231clld6b4e418e27bff6b9e6938c4",
  "dataLst": [
    "HuV2QuWRTlZJCCXazk2Cq1tW6T7Xc0JK/f9cUNfv/F1qcwp898LIjzSlX/Pk2osrw+xA8sSAAgSdO8sd42eR5Dx/",
    "e1TM13mPDgYgkltldirKpkxJ+BAoP4kCG4guRTK8STQoQicw+Xfflp9Ti++qpqxg0wxP4gSouo2W7Sf1ZHHIRC+",
    "Land",
    "UIpPsVEHqtUmb5CpfaWnNtegdsMJUl+euI65XwthLNnziD0e0u/4jgZa5yICZSWc8H6gmcKb7uW5OKuTf/aaM5Y9",
    "$450000",
    "XIMZC45zVs47KnX8vVN7DmNmCsfgEIyy0SITbjaDtImkb6IrVyVgxFBEZSlwanZ98qoAE0DX6D213qyesMIDttJT",
    "brian",
    "GKRvxT5ktMLQa8u3zpoGvxze+H3xquGGzqnqs4DhdM7UgXzY/sj7TjLZaEbzmculzmQ8owotiQx3heTUPtHSegg"
  ]
}

```

*Figure 5.3.2.9 Encrypted Result*

Based on figure 5.3.2.8, the data that are encrypted are buyer name, seller name, transaction date, and location, so as seen on the figure 5.3.2.9 highlighted on blue, some of the data on the list is encrypted, while some can be seen as plain text.

```

private void decryptAndShow(int index) {
    List<String> Blocks = Blockchain.get().get(index).getData().getDataLst();

    Asymmetric crypto = new Asymmetric("RSA");

    String buyer = "";
    String seller = "";
    String propLocation = "";
    String date = "";

    KeyPair keyPair = null;
    try {
        //get author's key pair
        keyPair = getKeyPair(Blocks.get(6));
    } catch (Exception e) {
        JOptionPane.showMessageDialog(this, "Key Pair Not Found");
    }

    PrivateKey privateKey = keyPair.getPrivate();

    //decrypt encrypted data in the blockchain
    try {
        buyer = crypto.decrypt(Blocks.get(0), privateKey);
        seller = crypto.decrypt(Blocks.get(1), privateKey);
        date = crypto.decrypt(Blocks.get(3), privateKey);
        propLocation = crypto.decrypt(Blocks.get(5), privateKey);
    } catch (Exception e) {
        e.printStackTrace();
    }

    buyerlbl.setText(buyer);
    sellerlbl.setText(seller);
    typelbl.setText(Blocks.get(2));
    datelbl.setText(date);
    pricelbl.setText(Blocks.get(4));
    locationlbl.setText(propLocation);
    authorlbl.setText(Blocks.get(6));
    digsigta.setText(Blocks.get(7));
}

```

*Figure 5.3.2.10 Decrypt Data*

Based on figure 5.3.2.10, the one highlighted on blue is the decrypting process. The encrypted data is then decrypted with the private key, making it visible as plain text. The figure above is the result of decrypting so it can be shown on the user interface.

The screenshot shows a web application window titled "View Property Transaction". It contains several input fields for transaction details and four buttons on the right side. The data entered in the fields is as follows:

Field	Value
Buyer Name:	jason
Seller Name:	derulo
Property Type:	Land
Date of Transaction:	1/3/2023
Price:	\$450000
Location:	New York
Author:	brian
Digital Signature:	EbzmcuLzmQ8owotiQx3heTUPtHSe qgp7HkmnyMc4JRAkR9FtA7mh2pqL vVOZjoKL9B2LIMLnjM/+tc4zj5k3D4/ wPb1wR+IOTarpBkYDXuWY/J3wQ=

The buttons on the right are "Next", "Previous", "Verify", and "Go Back Home".

Figure 5.3.2.11 Decrypted Result

## 5.4 Digital Signature Algorithm



Figure 5.4.1 Digital Signature Process (Pedamkar, 2023)

So contrary to asymmetric cryptography, how digital signature cryptography works is the data encrypted using private key and the public key then used for decrypting the data. Digital signatures are utilized to construct blocks in blockchains and to validate transactions (Google, n.d.). This implies that whenever someone makes a transaction to the blockchain, only the permitted individual may allow the node in the network to make use of the funds, prohibiting outsiders from doing so. The usage of digital signature is to verify the authenticity and integrity of the data (Google, n.d.). With the help of combination of hashing and asymmetric encryption, it creates a unique identifier for the data. To use this, below are the procedures.

According to Turner (2015), the user must first sign the communication using a private key as the first step to implement digital signature. It will be used to encrypt the hash after it has been signed. The reason for encrypting the hash value rather than the entire message is that the hash function helps to transform random input into a fixed length value that takes less time to convert. The hashed message must then be confirmed via a public key in the next step. It consists of two phases, which is hash generation and signature deciphering. We can decode the hash using the signer's public key. If the hash value matches the encrypted data, it is safe to say there has not been any alteration on the data since it's been digitally signed. However, if the hash values fail to be identical, then the data could have been altered with, or there may be problems with the signature formed, in which the private key is not compatible with the public key. The table below shows the advantages and disadvantages of using this technology (Indodax, 2023).

<b>Advantage</b>	<b>Disadvantage</b>
Increases security and authenticity of digital documents and transactions	Require more computational resources to authenticate data
Reduces the risk of fraud and tampering with digital documents	Not universally accepted or recognized, which can cause legal challenges in some jurisdictions
Allows for secure and legally binding electronic transactions	May require additional software or hardware to generate and verify digital signatures
Provides non-repudiation, meaning the signer cannot deny their signature	Key management and revocation can be complex and challenging



```

public class SignatureSign {
    private Signature sig;

    // Testing section (KeyPair)
    private KeyPair keyPair;

    // Get KeyPair from txt file
    public KeyPair getKeyPair(String data) throws IOException, NoSuchAlgorithmException, InvalidKeySpecException {
        // Read Public Key.
        File filePublicKey = new File("KeyPair/" + data + "/PublicKey");
        FileInputStream fis = new FileInputStream("KeyPair/" + data + "/PublicKey");
        byte[] encodedPublicKey = new byte[(int) filePublicKey.length()];
        fis.read(encodedPublicKey);
        fis.close();

        // Read Private Key.
        File filePrivateKey = new File("KeyPair/" + data + "/PrivateKey");
        fis = new FileInputStream("KeyPair/" + data + "/PrivateKey");
        byte[] encodedPrivateKey = new byte[(int) filePrivateKey.length()];
        fis.read(encodedPrivateKey);
        fis.close();

        // Generate KeyPair.
        KeyFactory keyFactory = KeyFactory.getInstance("RSA");
        X509EncodedKeySpec publicKeySpec = new X509EncodedKeySpec(
            encodedPublicKey);
        PublicKey publicKey = keyFactory.generatePublic(publicKeySpec);

        PKCS8EncodedKeySpec privateKeySpec = new PKCS8EncodedKeySpec(
            encodedPrivateKey);
        PrivateKey privateKey = keyFactory.generatePrivate(privateKeySpec);

        return new KeyPair(publicKey, privateKey);
    }
    // =====

    public SignatureSign(String data) {
        super();
        try {
            sig = Signature.getInstance("SHA256WithRSA");
            keyPair = getKeyPair(data);
        } catch (Exception e) {
            System.out.println("KeyPair Not Found");
        }
    }

    public String sign(String data) throws Exception {
        try {
            sig.initSign(keyPair.getPrivate());
            sig.update(data.getBytes());
        } catch (Exception e) {
            System.out.println("KeyPair Not Found");
        }
        return Base64.getEncoder().encodeToString(sig.sign());
    }

    public boolean verify(String data, String signature){
        boolean access = false;
        try {
            sig.initVerify(keyPair.getPublic());
            sig.update(data.getBytes());
            sig.verify(Base64.getDecoder().decode(signature));
            access = true;
        } catch (Exception e) {
            System.out.println("Not Found");
        }
        return access;
    }
}

```

Figure 5.4.2 SignatureSign.java

Figure 5.4.2 represents the SignatureSign.java class. For the getKeyPair() function, it is actually similar to what has been explained on figure 5.3.2.7, which is to read the encoded bytes of the public key and the private key from their respective files, and then uses these bytes to generate a KeyPair object using the RSA algorithm, and returns KeyPair object containing both the public and private keys. The sign() function is used to sign the data. First it initializes the sig object with the private key, then updates the sig object which generates the hash of the data, and returns a string which is the digital signature. The verify() function initializes the sig object with the public key, which then updates the sig object that generates the hash of the data. The function then decodes digital signature string and verifies it using the sig.verify build in API method.




```
//DIGITAL SIGNATURE
SignatureSign signature = new SignatureSign(username);

String encrypt = null;
try {
    encrypt = signature.sign(username);
} catch (Exception e) {
    e.printStackTrace();
}

btnAdd.setActionCommand(encrypt);
```

*Figure 5.4.3 Calling Sign Function in Add Record*



```
private void verifyBtnActionPerformed(java.awt.event.ActionEvent evt) {
    // TODO add your handling code here:
    //DIGITAL SIGNATURE
    SignatureSign sign = new SignatureSign(username);

    boolean isValid = sign.verify(username, digsigta.getText());
    if (isValid) {
        JOptionPane.showMessageDialog(this, "Signature is Verified");
    } else {
        JOptionPane.showMessageDialog(this, "Signature is Not Recognized");
    }
}
```

*Figure 5.4.4 Calling Verify Function in View Record*

The two figures above are how the function being called. For signing, it is used when the user wants to add transaction record to the chain. The result can be seen on figure 5.4.5 on the blue highlighted part. As for the verify function, it is called on the click button. If the signature is valid or invalid, it will show the corresponding user interface pop up message which can be seen in figure 5.4.6.

```
[
  {
    "header": {
      "index": 0,
      "currentHash": "0b90f9d2c2cba6d95cde577362b379db69190971eed1402f725e6d9453715a18",
      "previousHash": "0",
      "timestamp": 1679888533519
    }
  },
  {
    "header": {
      "index": 1,
      "currentHash": "6666f8fa478e2d1a71e7a05aa28ef0e05abb0691f810f815033126ac7a92f1ba",
      "previousHash": "0b90f9d2c2cba6d95cde577362b379db69190971eed1402f725e6d9453715a18",
      "timestamp": 1679888533627
    },
    "record": {
      "SIZE": 8,
      "merkleRoot": "3a293986f8c1a711d440f4d0416eada9a231c1d1d6b4e418e27bff6b9e6938c4",
      "dataList": [
        "HuV2QuWRT1zJCCXazk2CqltW6T7Xc0JK/f9cUNfv/Flqcwp898LIjzS1X/Pk2osrw+xA8sSAAgSdO8sd42eR5Dx/d/TMxWXV7ybw5RbZ1",
        "e1TM13mPDgYgk1tldirKpkxJ+BAoP4kCG4guRTK8STQoqQicw+Xfflp9Ti++gpqXg0wxP4gSouo2W7SF1ZHHIRC+KAb0uxrKTANpnrH39",
        "Land",
        "UIpPsVEHqtUmb5CpfaWnNTegdsmJU1+euI65XwthLNnziD0e0u/4jg2a5yICZSWc8H6gmcKb7uW5OKuTf/aaM5Y96EedbgDCw+1o4+WNL",
        "$450000",
        "XIMZC45zVs47KnX8vVN7DmNMCSfgEIyy0SITbjaDtImkb6IrVyVgxFBEZS1wanZ98qoAE0DX6D213qyesMIDttJT9919D4y8QBBOVeOtF",
        "brian",
        "GKRvxT5ktMLQa8u3zpoGvxeZe+H3xquGGzqnqs4DhdM7UgXzY/sj7TjLZaEbzmCuLzmQ8owotiQx3heTUPtHSeqqp7HkmnyMc4JRAkR9F"
      ]
    }
  }
]
```

Figure 5.4.5 Digital Signature on Ledger

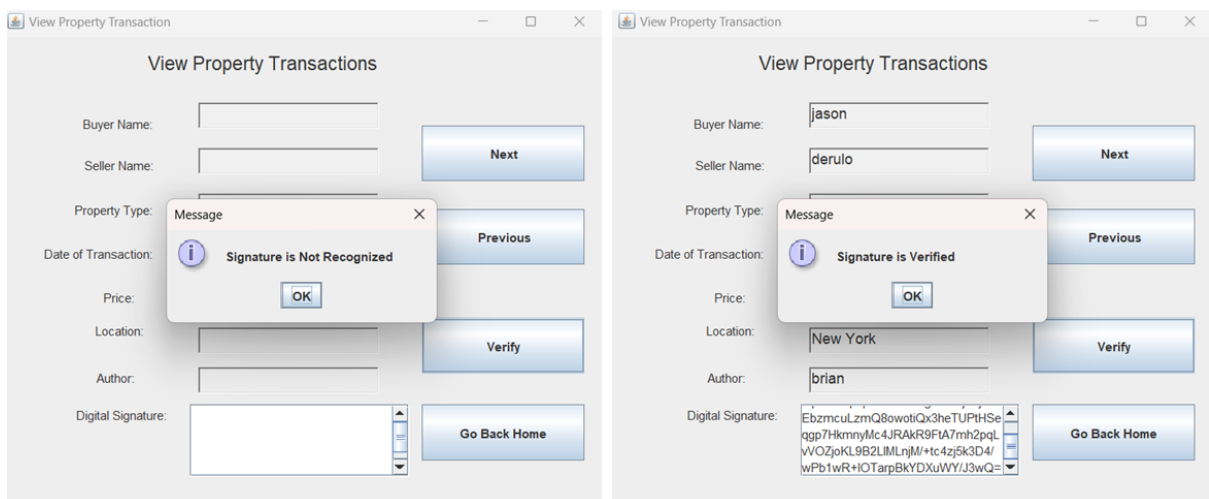


Figure 5.4.6 Verification of Digital Signature Visualization

## 5.5 Immutability Technique

On blockchain technology, immutability is one of the strongest characteristics, which means that once the data is inside the chain it becomes impossible to change or alter the data anymore (Doubleday, 2021). If a transaction record includes an error, a new transaction must be added to reverse the error, and both transactions are then visible. In this Real Estate transaction record system, each block has Merkle root. Merkle root and immutability concept are inextricably linked to one another (Zhou, 2022).

The Merkle root is a compact and fast approach to verify the integrity of the whole block. It symbolizes a cryptographic summary of all the transactions included in the block (Ravikiran, 2023). The immutability of a block in the blockchain is ensured by the use of the Merkle root. Because the Merkle root is dependent on the hashes of all the transactions in the block, modifying a single transaction in the block results in a different Merkle root. If an invader tries to make any modification to the Merkle root would be immediately identifiable, and the block would be deemed invalid.

```
{
  "header": {
    "index": 1,
    "currentHash": "6666f8fa478e2dla71e7a05aa28ef0e05abb0691f810f815033126ac7a92flba",
    "previousHash": "0b90f9d2c2cba6d95cde577362b379db69190971eed1402f725e6d9453715a18",
    "timestamp": 1679888533627
  },
  "record": {
    "SIZE": 8,
    "merkleRoot": "3a293986f8c1a711d440f4d0416eada9a231c1d1d6b4e418e27bff6b9e6938c4",
    "dataLst": [
      "HuV2QuWRT1zJCCXazk2Cq1tW6T7Xc0JK/f9cUNfv/Flqcwp898LIjzS1X/Pk2osrw+xA8sSAAgSdO8sd42eR",
      "elTM13mPDgYgkltldirKpkxJ+BAoP4kCG4guRTK8STQoqQicw+Xfflp9Ti++qpqxg0wxP4gSouo2W7SFlZHh",
      "Land",
      "UIpPsVEHqtUmb5CpfaWnNTegdsmJU1+euI65XwthLNnziD0e0u/4jgZa5yICZSWc8H6gmcKb7uW5OKuIf/aa",
      "$450000",
      "XIMZC45zVs47KnX8vVN7DmNMCSfgEIyyOSITbjaDtImkb6IrVyVgxFBEZS1wanZ98qoAE0DX6D213qyesMID",
      "brian",
      "GKRvxT5ktMLQa8u3zpoGvxeZe+H3xquGGzqnqs4DhdM7UgXzY/sj7TjLZaEbzmculzmQ8owotiQx3heTUPtH"
    ]
  }
}
```

*Figure 5.5.1 Merkle Root Result on Ledger*

```

public class MerkleTree {
    private List<String> tranxLst;
    private String root = "0";

    public String getRoot() {
        return root;
    }

    private MerkleTree(List<String> tranxLst) {
        super();
        this.tranxLst = tranxLst;
    }

    private static MerkleTree instance;
    public static MerkleTree getInstance( List<String> tranxLst ) {
        if( instance == null ) {
            return new MerkleTree(tranxLst);
        }
        return instance;
    }

    private List<String> genTranxHashLst(List<String> tranxLst) {
        List<String> hashLst = new ArrayList<>();
        int i = 0;
        while( i < tranxLst.size() ) {
            String left = tranxLst.get(i);
            i++;

            String right = "";
            if( i != tranxLst.size() ) right = tranxLst.get(i);

            String hash = Hasher.sha256(left.concat(right));
            hashLst.add(hash);
            i++;
        }
        return hashLst;
    }

    public void build() {
        List<String> tempLst = new ArrayList<>();

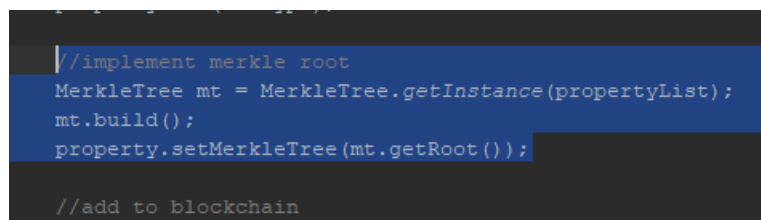
        for (String tranx : this.tranxLst) {
            tempLst.add(tranx);
        }

        List<String> hashes = genTranxHashLst( tempLst );
        while( hashes.size() != 1 ) {
            hashes = genTranxHashLst( hashes );
        }
        this.root = hashes.get(0);
    }
}

```

*Figure 5.5.2 MerkleTree.Java*

In this class, the `tranxLst` variable is a list of transactions, which is used to build the Merkle tree. The `root` variable is a string representing the root node of the Merkle tree, which is initially set to "0" but is updated when the tree is built. The `getInstance()` function returns a singleton instance of the `MerkleTree` class. This ensures that only one instance of the class is created throughout the lifetime of the program. The `genTranxHashLst()` function generates the hash values of the transactions in a given list. It iterates over the list in pairs, concatenates the left and right values, and computes the SHA-256 hash of the concatenated string. The hash values are stored in a list and returned. The `build()` function builds the Merkle tree using the transactions in `tranxLst`. It starts by generating the hash values of all the transactions and then repeatedly computes the hash values of adjacent pairs of nodes until there is only one hash value left, which becomes the root value.



```
//implement merkle root
MerkleTree mt = MerkleTree.getInstance(propertyList);
mt.build();
property.setMerkleTree(mt.getRoot());

//add to blockchain
```

*Figure 5.5.3 Merkle Root Implementation on Add Record*

The code snippet above is the command that generates a Merkle root. It first creates the Merkle tree object with the property data list as the parameter, and then it calls the `build()` function which will generate the Merkle root hash value, and then add it into the property list. The result of this operation can be seen on figure 5.5.1 on the blue highlighted part.