Implementing Hybrid Blockchain for Vehicle Fine Records Management System

1st Akib Ahmed Fahad
B.Sc. in Computer Science and Engineering
ID:011162083

2nd Md. Hasibul Huq Shanto
B.Sc. in Computer Science and Engineering
ID:011191066

3rd Ali Ahammed Khan
B.Sc. in Computer Science and Engineering
ID:011162039

4th Md Ishtiaque Ferdous Emon
B.Sc. in Computer Science and Engineering
ID:011181132

Dr. Mohammad Shahriar Rahman
Associate Professor
Department of Computer Science and Engineering
United International University

***Abstract*—Sharing the vehicle fine records poses some challenges regarding concealing the sensitive information as well as secured, transparent, and efficient distribution of the non-sensitive information. Thus, we propose a management system of vehicle fine records that implement a hybrid blockchain paired with security features that ensures decentralization, transparency, immutability, security, and improved performance.**

***Index Terms*—Decentralization, Hybrid Blockchain, Cryptography, Digital Signatures, Security Model, Privacy, Traffic Offense Records**

I. Introduction

A. Problem Statement

The records of a traffic offense need to be shared among various stakeholders such as insurance companies, government officials, and employers, outside the law enforcement agencies that are in charge of these records. These records contain many sensitive information like National Identification (NID) Number, Vehicle Registration number, etc. should not be publicly available to avoid identity theft or the danger of malicious stalking. From Figure 1 we can see how serious of a problem identity theft is, affecting over a million of lives.

On the other hand, a unique identifier such as vehicle fine notice ID is used to communicate between the database and blockchain in their corresponding fields to obtain the specific vehicle fine records requested by the system user.

The implementation of the hybrid blockchain system into the vehicle fine system would ensure that all sensitive data are managed securely whilst allowing decentralization of non-sensitive data. The raw data with sensitive content can be kept in a secure database with access restricted to the private network, while the non-sensitive data is shared with the public network as a public ledger.

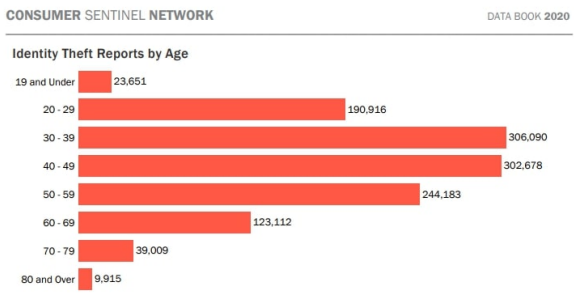

Fig. 1: Reported Cases of Identity Theft by Age in the US in 2020

B. Motivation

We want to develop a system that can hold personal information in an encrypted way so that no one can leak information from a database, where we can store all kinds of data. For example, public access data and private access data are stored together in the same system but there will be filtering for users who can access only public data and who can access all data. In Vehicle Fine system it's rare to find hybrid blockchain technology security model.which is convenient and more secure. By using this blockchain technology, our public or private data will be safe. Moreover, traditional architectures designed for sharing the records have a feasible risk of the occurrence of server traffic congestion if they use centralized servers. Also, these servers require high power to handle the huge amount of client requests, rendering this approach less cost-effective.

The implementation of hybrid blockchain in the road department system can bring many benefits such as transparency, decentralization, immutability, and provide better performance.

II. Literature Review

In this section we describe the relevant literature works we have studied for this project.

[1] Node identity authentication is an important means to ensure its security in wireless sensor networks(WSN). The analysis of security and performance shows that the scheme has comprehensive security and better performance.

[2] With the rapid development and wide application of crowdsourcing, the shortcomings of the traditional crowdsourcing systems are gradually exposed. The traditional centralized crowdsourcing systems are vulnerable to a single point of failure. For example, users could not access Wikipedia data from the server due to eight service outages in 2018. In such a centralized system where a server controls all the transactions, the issue of controller's silent misbehavior is likely to occur without effective detection. When a conflict of opinions exists between the task requesters and the workers, the issues of free riding (workers receive rewards without making real efforts) and false reporting (requesters try to repudiate the payment) could happen in the system. During the procedure of task assignment, the sensitive information (e.g., location and preference) of crowdsourcing participants may be revealed to the public. For example, in November 2017, 57 million Uber drivers' information was hacked.

[3] Decentralization In a blockchain-based decentralized system, the rights and obligations of all nodes are equal. The operation of the system will not be affected even though a node stops working. Collective maintenance The system is maintained by all nodes with maintenance functions. Everyone in the system participates in the maintenance work. Automation With the help of smart contracts, the resource and data sharing services could be automatically executed without human intervention.

[5] Contract management or rights management is important for monopolizing resources. To prevent an attacker from monopolizing resources, blockchain technology is giving better security.

III. Tools and Validation

A. Description of Building Blocks

1) Hashing: Hashing is used to generate a fixed-length character value from a string of text using a mathematical algorithm. Hashing works as a one-way cryptographic function where the hash cannot be decrypted to retrieve the original data compared to the encryption concept. Some of the popular hashing libraries include MD5, SHA-1, SHA-2, SHA-256, and others.

2) Blockchain: Blockchain is a growing list of records, called blocks, that are linked and secured using cryptography. As illustrated in Figure 2, blocks are cryptographically linked through hash; i.e., hash of a block is same as the previous hash of the block succeeding it in the blockchain. The first block in a blockchain is called "Genesis Block" and its previous hash is zero as there is no block preceding it. Matching of a block's hash with the previous hash value of the next block is mandatory for the blockchain to be considered valid.

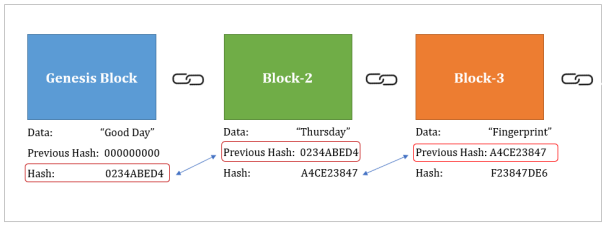

Fig. 2: Visual Representation of an Example of a Blockchain

3) Elliptic Curve Cryptography (ECC): [6] An elliptic curve is the set of points that satisfy the mathematical equation:

$$y^2 = x^3 + ax + b$$

Figure 3 shows a graphical representation of an elliptic curve.

Cryptography is the ciphering or scrambling of an ordinary text into ciphertext using a key that is deciphered later upon arrival of the desired destination or recipient using the same key (symmetric) or a different one (asymmetric). The Elliptic Curve Cryptography (ECC) is modern family of public-key cryptosystems based on the algebraic structures of the elliptic curves over a finite field and on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP).

For cryptographic purposes, rather than allowing any value for the points on the curve, the domain is restricted to integers in a fixed range as shown in Fig. 4. An elliptic curve cryptosystem can be defined by picking a prime number as a maximum, a curve equation and a public point on the curve. A private key is a number *priv*, and a public key is the public point that undergoes some mathematical operations with itself *priv* times. Computing the private key from the public key in this kind of cryptosystem is called the elliptic curve discrete logarithm function.

ECC implements all major capabilities of the asymmetric cryptosystems: encryption, signatures and key exchange. Also, it uses smaller keys and signatures than RSA for the same level of security thus providing very fast key generation, fast key agreement and fast signatures.

4) Digital Signatures: A digital signature is a cryptographic value calculated from the author's desired data and private key. In this process, The recipient would use the public key given by the author to decrypt the digital signature and then use the same hashing algorithm as the author to obtain the hash value of the digital message and data. Afterwards, the recipient would check whether the digital signature is the same or not using the same hashing algorithm. If the hash value produced is the same as the decrypted digital signature, then the digital message or data has not been tampered with. With digital signature, users could be identified as each digital signature is unique to one person or an entity, ensuring that the information can be securely protected.

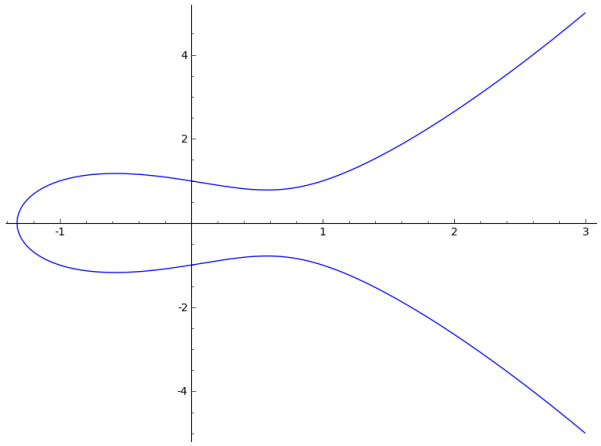

Fig. 3: An Elliptic Curve

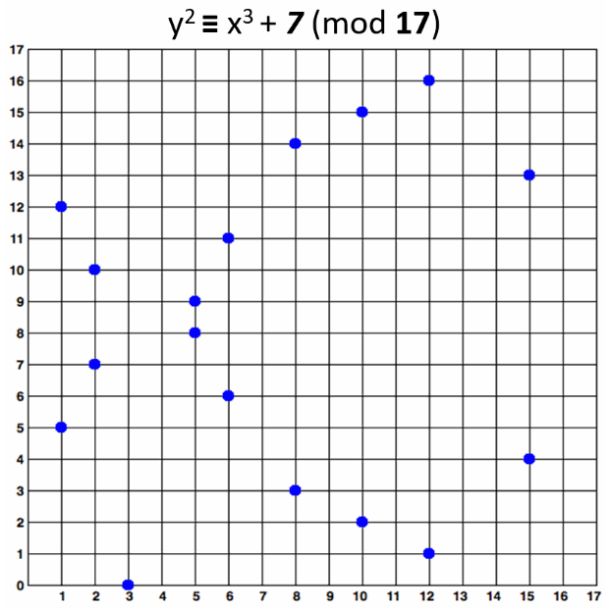

Fig. 4: An elliptic curve over a finite field

B. Technologies Used with Justification

In Table I the tools, software, and platform used are mentioned along with their justification.

IV. Plan and Design

A. Project Description

The aim of our project is to decentralize the vehicle fine records information without disclosing sensitive information of the user and without compromising security. Figure 5 is a high level representation of the integration of blockchain for distribution of data. The hybrid blockchain consists of a private and a public network. The private network is accessible to the primary stakeholders, which are the police department officials and database administrators who are in charge of handling the database and hosting the public ledger. The public network is accessible to rest of the stakeholders, which may include insurance companies, recruitment agencies, government officials, etc, that can only retrieve the data. Thus, our secure and reliable system encourages collaboration among various stakeholders.

Technology	Justification
Java(JDK 11)	In-built modules for security features, access modifiers, cross-platform support, object-oriented programming. JDK 11 is LTS so get support for a long time.
IntelliJ IDEA	Popular IDE for Java development with many features enabling to maintain and modularize code easily.
GitHub	Version Controlling using Git. Secured. Easy to use. More familiarity.
SHA-256 (hashing)	Very secured. Recommended by US government agencies to protect sensitive information.
ECC (encryption), ECDSA (digital signature)	Use smaller keys to get the same levels of security as RSA. Suitable for cryptography done on less powerful devices like mobile phones.

TABLE I: Engineering Technologies Used with Justification.

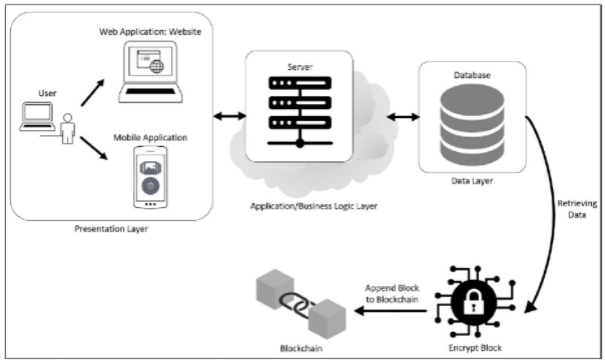

Fig. 5: A high level representation of the integration of blockchain with security components

B. System Design

Figure 22 shows the algorithmic view of our proposed system. When the user first launches the system, they are given two options: either select the Administrator user account or a regular user account. When the administrator presses the administrator selection, the system will check whether any administrator credentials are stored in the database. If there are no database entries containing admin credentials, the administrator would be redirected to the Registration page. Once the administrator has registered an account or the database entry containing admin credentials exists, the administrator is redirected to the login page. In order to log into the vehicle fine system, the administrator would need to enter their Admin ID, password then press the “Login” button to verify the typed credentials matches with the administrator credentials stored in the database. If the database entry does

not match with the typed credentials, a prompt as shown in the figure above will pop out letting the administrator know the administrator credentials do not match with the administrator credentials in the database and let them try again to log into the system.

Once the administrator has successfully logged in to the system, the administrator is redirected to the administrator dashboard. In the Administrator Dashboard, the administrator can add the vehicle fine record in "Add Fine Record" tab after filling up the add fine record form for the new record. The administrator can view the database by selecting the "View Fine Record" tab and has permission to view sensitive data such as the NID number and vehicle number as well.

The login process of the user is the same as the administrator. After logging in to the system, the user can search for records by providing the NID number. Then the system will search the matched vehicle fine records in the public ledger and display them in a tabular fashion. A failed search attempt prompt would be displayed when the inputted NID number of the user does not have any match in the ledger. The user can further filter the current displayed matched vehicle fine records in the table by one's vehicle number to specify the desired vehicle fine records. A failed filter attempt alert would be shown when the inputted vehicle number of the user does not have any match in the ledger.

C. Implementation

```
//    Hashing the block with salt
    public static String hash(byte[] blockBytes, String ALGO)
    {
        String hashOutput = null;
        byte[] inputBytes = blockBytes;

        try
        {
            md = MessageDigest.getInstance(ALGO); //Hash using SHA-256.
            md.update(inputBytes); //feed in input bytearray to md instance

//            Adding salt.
            md.update(Salt.gen());

//            Generate the hash output as byte[]
            byte[] hashBytes = md.digest();

//            Byte[] not much readability hence convert to Hex format.
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < hashBytes.length; i++)
            {
                sb. append(Integer.toHexString(0xFF & hashBytes[i]));
            }
            hashOutput = sb.toString();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }

        return hashOutput; //Return SHA-256 hashed Block data.
    }
```

Fig. 6: BlockHasher.java code snippet

1) Hashing: Fig. 6 showcases the hash() method used to hash the block data that is used to append to the Blockchain. First, it grabs the blockBytes that were converted in the getBytes() method in Block.java. Then, a MessageDigest is initiated with the SHA-256 hashing algorithm. In the MessageDigest, the blockBytes and a salt value are added to complicate the hash generation further to produce a unique value. After that, the md.digest() method is used to generate the MessageDigest as a byte array. After that, a for loop is then used to convert the byte array into a string value by parsing the hex value in the byte array to change it to a String value representation to show the SHA-256 block hash in plain-text, which is returned into a string called hashOutput.

```
public class Block implements Serializable
{
//    Block Properties
    public static int counter;

    private int index;
    private String NoticeID, IDCategory, NoticeType,
           OffenseType, OffenseLocation, OffenseDate, PaymentDate; //data
    private String hash, previoushash;
    private long timestamp;

//    Constructor
    public Block(String NoticeID, String IDCategory, String NoticeType, String OffenseType,
                 String OffenseLocation, String OffenseDate, String PaymentDate, String previoushash) throws Exception
    {
        this.index = counter;
        counter++;
        this.NoticeID = NoticeID;
        this.IDCategory = IDCategory;
        this.NoticeType = NoticeType;
        this.OffenseType = OffenseType;
        this.OffenseLocation = OffenseLocation;
        this.OffenseDate = OffenseDate;
        this.PaymentDate = PaymentDate;
        this.previoushash = previoushash;
        this.timestamp = new Timestamp(System.currentTimeMillis()).getTime();

//        Generate current hash for the block
        byte[] blockBytes = Block.getBytes(this);

        if(blockBytes != null)
        {
//            Complicate the blockBytes by combining timestamp, data, and previoushash.
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            baos.write(blockBytes);
            baos.write(previoushash.getBytes());
            baos.write(Long.toString(timestamp).getBytes());
            this.hash = BlockHasher.hash(baos.toByteArray(), "SHA-256");
        }

        else
        {
            throw new Exception("Unable to generate current block hash");
        }
    }
```

Fig. 7: Block.java code snippet

2) Blockchain: As shown in Fig. 7, the Block.java code snippet explains the data contained by the block via defining various block variables snd properties to be implemented in the block, such as the index number, different vehicle fine record variables, and a timetable. It also depicts how the Block hash is being created by getting the bytes of a block that is then stored in a byte array. After the byte array is created, the blockBytes is further complicated with the block index, block timestamp, and vehicle fine records stored in the block to generate a unique hash for the block to be appended into the blockchain.

```
private static byte[] getBytes(Block blk)
{
    try(
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            ObjectOutputStream out = new ObjectOutputStream(baos); //getBytes returns in object form.
            )
    {
        out.writeObject(blk);
        return baos.toByteArray(); //Return byte[]

    } catch (Exception e)
    {
        e.printStackTrace();
        return null;
    }
}
```

Fig. 8: Block.java code snippet

In Fig 8, the getBytes() method is used to convert a Block into a byte array. The method first creates a ByteArrayOutputStream that implements an output stream where the data is written into a byte array. Then, an ObjectOutputStream is created to write primitive data types of Java objects to an OutputStream. In this case, it passes the ByteArrayOutputStream object called “baos”. An ObjectOutputStream object called “out” is used with the combination of the writeObject method, which writes the “blk” into a ByteArrayOutputStream. After that, the ByteArrayOutputStream of the block is converted into a byte array in the return method.

```
public class BlockFunctions
{
    //    Add Blocks to the Blockchain - validation whether 1st block or next block.
    public static void inputBlock(String NoticeID, String IDCategory, String NoticeType,
                                  String OffenseType, String OffenseLocation,
                                  String OffenseDate, String PaymentDate)
    {
        File file = new File(Paths.get("ledger.txt").toString());
        boolean exists = file.exists();

//        Check if file created or file contains something.
        if (exists == false || file.length() == 0)
            genesisBlock(NoticeID, IDCategory, NoticeType, OffenseType,
                    OffenseLocation, OffenseDate, PaymentDate);
        else
            nextBlock(NoticeID, IDCategory, NoticeType, OffenseType,
                    OffenseLocation, OffenseDate, PaymentDate);
    }

//    Create the first Block in the Blockchain
    static void genesisBlock(String NoticeID, String IDCategory, String NoticeType,
                             String OffenseType, String OffenseLocation,
                             String OffenseDate, String PaymentDate)
    {
        try
        {
            Block genesis = new Block(NoticeID, IDCategory, NoticeType, OffenseType,
                    OffenseLocation, OffenseDate, PaymentDate, "0");
            Blockchain.nextBlock(genesis);
        }
        catch(Exception ex)
        {
            Logger.getLogger(BlockFunctions.class.getName()).log(Level.SEVERE, null, ex);
        }
        finally
        {
            Blockchain.distribute();
        }
    }
```

Fig. 9: BlockFunctions.java code snippet

As shown in Fig. 9, the BlockFunctions class contains various methods interacting with the Block. One of those methods includes the inputBlock() method used to append new data to the ledger. The administrator uses this method when a vehicle fine record is added, which will trigger to append the block data to the ledger. First, the inputBlock() method checks whether the ledger file is empty or not created. If it is not, a genesis block, also known as a first block, is created in the blockchain and inserted into the ledger.

The BlockFunctions class shown in Fig 9 also contains the genesisBlock() method used to create the first block in the Blockchain, which manually sets the previous hash to zero as there are no previous blocks before it since it is the first block to be created.

The BlockFunction class also contains the nextBlock() method shown in Fig. 10 to generate the next block in the Blockchain, which would store the hash value from the previous block.

As shown in Fig. 11, the Blockchain.java file contains a

```
//    Add NewBlock to the Blockchain.
    static void nextBlock(String NoticeID, String IDCategory, String NoticeType,
                          String OffenseType, String OffenseLocation,
                          String OffenseDate, String PaymentDate)
    {
        try
        {
            String previousHash = Blockchain.get().getLast().getHash();

            Block b = new Block(NoticeID, IDCategory, NoticeType, OffenseType,
                    OffenseLocation, OffenseDate, PaymentDate, "0");
            Blockchain.nextBlock(b);
        }
        catch(Exception ex)
        {
            Logger.getLogger(BlockFunctions.class.getName()).log(Level.SEVERE, null, ex);
        }
        finally
        {
            Blockchain.distributeAppend();
        }
    }
```

Fig. 10: BlockFunctions.java code snippet

```
public class Blockchain
{
//    Stores the object of the linked list, since block is stored in chronological order,
//    it would keep the change in a binary format.
//    Access the file to get the previous hash to add the block to the chain.
    private static final String BCHAIN_FILE = "master/chain.dat";

//    source - gets it from the file stored in the memory
    private static LinkedList<Block> db = new LinkedList<>();

//    operations: persist, get, distribute blockchain
//    write to the masterchain file (persist)

    private static void persist()
    {
//        helpers: FileOutputStream & ObjectOutputStream
        try
                (
                        FileOutputStream fos = new FileOutputStream(BCHAIN_FILE);
                        ObjectOutputStream out = new ObjectOutputStream(fos);
                        )

        {
//            write db object to the file
            out.writeObject(db);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
```

Fig. 11: Blockchain.java code snippet

directory to the masterchain file called chain.dat in the master folder that stores the entire linked list object, and a LinkedList compromising of Blocks is created to be referred later. There are also several methods available in the Blockchain class that would be discussed below.

The get() method in Fig 12 is used to obtain the existing chain of Blocks in the Blockchain by referring to the masterchain file that stores the entire LinkedList object.

In Fig.13 the distribute() method would convert the Block java object into a JSON format that would be stored into a file named “ledger.txt”. The ledger file only contains non-sensitive data. Therefore, it provides some transparency of data available in the system to the users and includes decentralization by allowing the ledger to be hosted by other nodes that would like to participate in the peer-to-peer blockchain system.

In Fig 13 the distributeAppend() method is used to append new Block data into the JSON file while ensuring that the JSON file formatting for the blocks remains intact. The nextBlock() method is used to execute the add() method and

```
//    obtain the existing chain
    public static LinkedList<Block>get()
    {
//        helpers: FileInputStream and ObjectInputStream
        try(
                FileInputStream fis = new FileInputStream(BCHAIN_FILE);
                ObjectInputStream in = new ObjectInputStream(fis);
                )
        {
            return (LinkedList<Block>)in.readObject();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            return null;
        }
    }
```

Fig. 12: Blockchain.java code snippet

```
//    display/write the Blockchain
    public static void distribute()
    {
//        display the Blockchain
        String chain = new GsonBuilder()
                        .setPrettyPrinting()
                        .create()
                        .toJson(db);
        try
        {
            Files.write(Paths.get("ledger.txt"), chain.getBytes(), StandardOpenOption.CREATE);
        }
        catch (IOException ex)
        {
            Logger.getLogger(Blockchain.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

//    Append new data to the Ledger.
    public static void distributeAppend()
    {
//        display the Blockchain
        String chain = new GsonBuilder()
                        .setPrettyPrinting()
                        .create()
                        .toJson(db);
        try
        {
            Files.write(Paths.get("ledger.txt"), chain.getBytes(), StandardOpenOption.SPARSE);
        }
        catch (IOException ex)
        {
            Logger.getLogger(Blockchain.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
//    add a new block to the existing Blockchain
    public static void nextBlock(Block newBlock)
    {
        Blockchain.db.add(newBlock);
        Blockchain.persist();
    }
```

Fig. 13: Blockchain.java code snippet

the persist() method in the Blockchain class to add a new block to the existing blockchain.

The readLedger() method, shown in Fig. 14 is used to read the block data stored in the ledger. The ledger is accessed using a FileReader that is passed to the BufferedReader to be processed by the Block array. The Block array retrieves the ledger file from the BufferedReader and stores the JSON data of the ledger. An enhanced for loop (for-each loop) is then used to iterate through the block and store each Block data variable by adding the specific data into a list comprising of Strings. After the for loop concludes, the list comprising of Strings that contain Block data is returned.

3) Cryptography: Fig 15 shows the class and method used to implement asymmetric encryption in the vehicle fine records

```
//    Read the ledger.txt file.
    public static List<String> readLedger(String DBNoticeID)
    {
        Gson gsonRead = new Gson();
        List<String> ledgerData = new ArrayList<>();
        try
        {
            FileReader fileReader = new FileReader("ledger.txt");
            BufferedReader buffered = new BufferedReader(fileReader);
            Block[] userArray = gsonRead.fromJson(buffered, Block[].class);

            for (Block block : userArray)
            {
                if (DBNoticeID.equals(block.getNoticeID()))
                {
                    ledgerData.add(block.getNoticeID());
                    ledgerData.add(block.getIDCategory());
                    ledgerData.add(block.getNoticeType());
                    ledgerData.add(block.getOffenseType());
                    ledgerData.add(block.getOffenseLocation());
                    ledgerData.add(block.getOffenseDate());
                    ledgerData.add(block.getPaymentDate());
                }
            }
        }
        catch (Exception ex)
        {
        }

        return ledgerData;
    }

}
```

Fig. 14: Blockchain.java code snippet

```
public class ASymmCrypto
{
    private Cipher cipher;

//    Constructor
    public ASymmCrypto()
    {
        try
        {
            cipher = Cipher.getInstance(Config.ALGORITHM);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

//    Encryption using public key
    public String encrypt(String data, PublicKey key) throws Exception
    {
//        Convert to byteArray for encryption
        String cipherText = null;
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] cipherBytes = cipher.doFinal(data.getBytes());

        cipherText = Base64.getEncoder().encodeToString(cipherBytes);
        return cipherText;
    }

//    decryption using private key
    public String decrypt(String cipherText, PrivateKey key) throws Exception
    {
//        Convert to string for decryption.
        String data = null;
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] dataBytes = cipher.doFinal(Base64.getDecoder().decode(cipherText));

        data = new String(dataBytes);
        return data;
    }
}
```

Fig. 15: AsymmCrypto.java code snippet

system. First, the ASymmCrypto class is initialized with a Cipher instance with the Elliptic Curve Integrated Encryption Scheme (ECIES) algorithm. Then, in the class, there are two main methods which are the encrypt() method and decrypt() method.

In the encrypt() method in Fig 15, a string called CipherText is initialised, and a Cipher instance is also initialized and set to ENCRYPT_MODE that has a public key passed into it to encrypt the data. Then, the data is converted into bytes using the getBytes() method and then encrypted with the ECIES algorithm using the public key via the Crypto.doFinal() method, which is then stored into a byte array. Then, the byte array is converted into a hashed string form by using the Base64 encoder to convert the cipherBytes into cipherText.

On the other hand, in the decrypt method shown in the same figure, a string called data is initialized, and a Cipher instance is also initialized and set to DECRYPT_MODE that has a private key is passed into it to decrypt the data. Then, the cipher.doFinal() method decrypts the data using the ECIES algorithm using the private key and converts the cipherText into a plain text form using the Base64 decoder. Then, the decrypted data is stored in a variable called data that is returned.

```
public class KeyAccess
{
    public static PublicKey getPublicKey() throws Exception
    {
//        content is stored in byte[] array, need to convert back to original form (keyspec).
        byte[] keyBytes = Files.readAllBytes(Paths.get(Config.PUBLICKEY_FILE));

//        convert to proper keyspec
        X509EncodedKeySpec spec = new X509EncodedKeySpec(keyBytes);
        return KeyFactory.getInstance(Config.ALGORITHM).generatePublic(spec);
    }

    public static PrivateKey getPrivateKey() throws Exception
    {
//        content is stored in byte[] array, need to convert back to original form (keyspec).
        byte[] keyBytes = Files.readAllBytes(Paths.get(Config.PRIVATEKEY_FILE));

//        convert to proper keyspec
        PKCS8EncodedKeySpec spec = new PKCS8EncodedKeySpec(keyBytes);
        return KeyFactory.getInstance(Config.ALGORITHM).generatePrivate(spec);
    }
}
```

Fig. 16: KeyAccess.java code snippet

As shown in Fig 16, there are two main methods in the KeyAccess class, which are the getPublicKey() method and the getPrivateKey() method. The getPublicKey() method obtained the public key bytes stored in the PublicKey file and saved them to a byte array called keyBytes. The byte array is then converted to the proper key specification under the X509 standard, and the public key is returned to the system. The private key is also obtained similarly, but instead of passing the public key in the method, the private key is passed instead, as shown in the getPrivateKey() method.

In the KeyMaker class shown in Fig. 17, the KeyMaker is initialized with the ECIES algorithm with a size of 256 for the keypair. Then, the mkKeyPair() method is used to generate the public key and private key for the use of asymmetric encryption in the system. First, a Keymaker called is initialized, and a keypair object is called to run the generateKeyPair() to generate the public key and the corresponding private key.

```
public class Keymaker
{
    KeyPairGenerator keygen;
    KeyPair keypair;

//    Constructor
    public Keymaker() throws Exception
    {
        keygen = KeyPairGenerator.getInstance(Config.ALGORITHM); //ECIES
        keygen.initialize(256); //This small keysize for ECC is more effective than 1024 keysize of RSA
    }

//    Make a keypair
    public static void mkKeyPair()
    {
        try
        {
//            Generate Key Pairs
            Keymaker keymaker = new Keymaker();
            keymaker.keypair = keymaker.keygen.generateKeyPair();

//            Get public key
            PublicKey publicKey = keymaker.keypair.getPublic();

//            Get private key
            PrivateKey privateKey = keymaker.keypair.getPrivate();

//            Store public key and private key into a file
            store(Config.PUBLICKEY_FILE, publicKey.getEncoded());
            store(Config.PRIVATEKEY_FILE, privateKey.getEncoded());
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
```

Fig. 17: Keymaker.java code snippet

The keymaker.keypair.getPublic() method is used to obtain the public key for the system that is stored under the public key object. Then, the keymaker.keypair.getPrivate() method is used to obtain the private key for the system stored that is stored under the private key object. After obtaining the keys, the public key and private key are saved into the file locations defined in Config.java class.

```
//    Constructor
    public DigitalSignature()
    {
        try
        {
            cipher = Cipher.getInstance(CRYPTO_ALGO);
            keygen = KeyPairGenerator.getInstance(CRYPTO_ALGO);
            keypair = keygen.generateKeyPair();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

//    Encryption Cipher
    public byte[] hash(String data)
    {
        byte[] hashBytes = null;
//        API-MessageDigest
        try
        {
            MessageDigest md = MessageDigest.getInstance(HASHING_ALGO);
            hashBytes = md.digest(data.getBytes());
            return hashBytes;
        }
        catch (Exception e)
        {
            e.printStackTrace();
            return null;
        }
    }
```

Fig. 18: DigitalSignature.java code snippet

4) Digital Signatures: Fig 18 above shows the class and methods used to implement digital signature in the vehicle fine records system. First, the DigitalSignature class is initialized with several instances, such as the Cipher instance with the SHA256withECDSA algorithm and a KeyPairGenerator with the ECDSA algorithm. It also executes the keygen.genKeyPair() method. Other than that, there are several methods in the class, such as the hash() method used to hash the MessageDigest data using the SHA-256 hashing algorithm and returned as a byte array.

```
//    encrypt; verify (decrypt)
    public String encrypt(byte[] hash)
    {
        byte[] dsBytes = null;
        try
        {
//            init
            cipher.init(Cipher.ENCRYPT_MODE, keypair.getPrivate());
            dsBytes = cipher.doFinal(hash);
        }
        catch (Exception e)
        {
        }

        if (dsBytes == null)
        {
            return Base64.getEncoder().encodeToString(dsBytes);
        }

        else
        {
            return null;
        }
    }

    public boolean verify(String data, String ds)
    {
        byte[] dataHash = hash(data);
        byte[] dsBytes = null;

        try
        {
            cipher.init(Cipher.DECRYPT_MODE, keypair.getPublic());
            dsBytes = cipher.doFinal(Base64.getDecoder().decode(ds));
        }

        catch(Exception e)
        {
            e.printStackTrace();
        }

        return Arrays.equals(dataHash, dsBytes);
    }
}
```

Fig. 19: DigitalSignature.java code snippet

In Fig. 19, the encrypt() method in the digital signature is used to encrypt the byte array of the MessageDigest obtained from the hash() method explained previously. The byte array is encrypted using the private key, and the dsBytes stores the byte array after it has been encrypted using the ECDSA algorithm. There is also null checking to verify if there is any data from the byte array generated. If there is, then the Base64 Encoder is used to convert the hashed byte array data into a String format.

On the other hand, the verify() method is used to verify whether the digital signature is correct to ensure that the data was not tampered. In order to do so, the verify() method obtains the data and the hashed digital signature. The data is hashed with the same SHA-256 algorithm to compare the hash between the original data and the generated digital signature. The digital signature itself is decrypted using the public key and stored in a byte array called dsBytes. Then, in the return function, it checks whether the hash in the digital signature called dsBytes is the same as the hash generated from the current data called dataHash. If there are the same, a Boolean with the true result is produced. Otherwise, the result would be false.

D. Project Plan

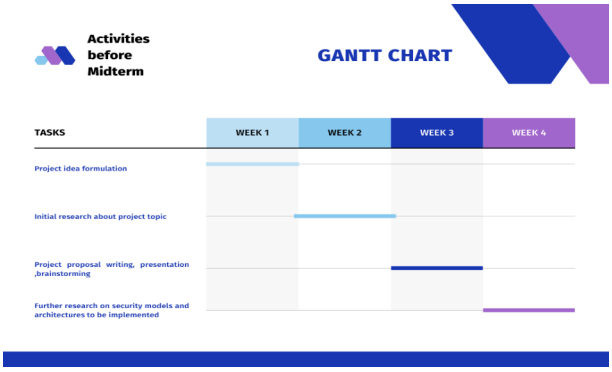

Fig. 20: Gantt Chart of Activities Before Midterm

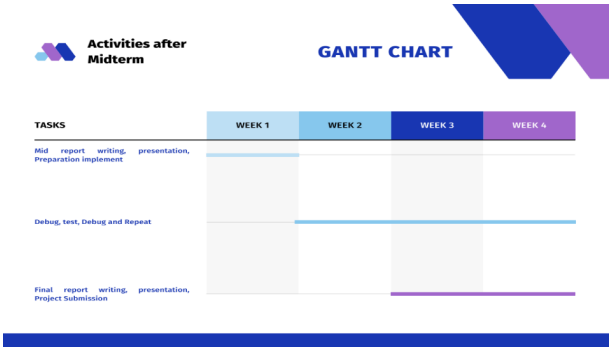

Fig. 21: Gantt Chart of Activities After Midterm

Figure 20 depicts the activities that have been done in the first half of the project duration while Figure 21 illustrates the activities that will be conducted in the rest of the duration of the project.

V. Complex Engineering Problems and Activities

A. Problems and Activities to Address Them

The application of security features along with blockchain technology would give us many benefits such as transparency, decentralisation, immutability, and security as well as improve the performance of the system.

The **transparency** of the proposed Vehicle Fine Records System would be achieved by distributing the ledger that stores some information about the vehicle fine records that is not

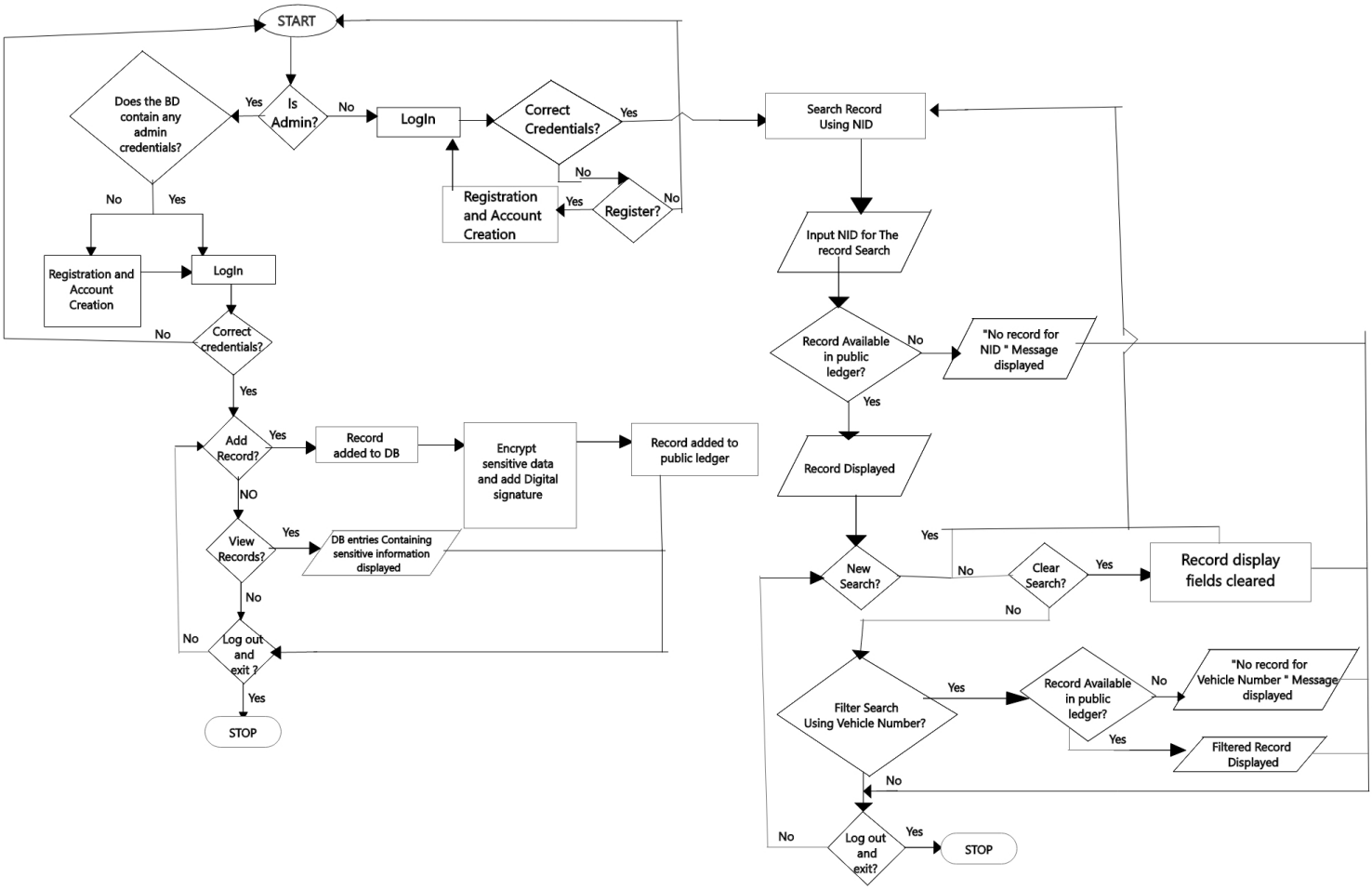

Fig. 22: Algorithmic View of the Vehicle Fine Records Management System

sensitive across participating nodes in the blockchain to the system users.

The data stored in the ledger and the data stored in the database are different as the database would only be accessed by the primary stakeholders of the proposed system, whereas the ledger would be accessed by the other public stakeholders. In other words, the **decentralisation** of the proposed Vehicle Fine Records System would be attained by limiting the blockchain network to "trusted" nodes to host the ledger containing non-sensitive data of the vehicle fine records.

Aside from that, **immutability** ensures the ledger's data has not been tampered with by hashing the block that stores the vehicle fine records. The block hash would then be used to append to a new block that references the hash of the previous block, which causes a chain of blocks to form. Blockchain is designed to be an append-only structure that only allows read and write operations.

Besides that, **immutability** in blockchain could be achieved so that the ledger's data would not be tampered by implementing Digital Signature to obtain the digest of the vehicle fine records in the requested block. This is because there would be chances of a personnel with malicious thoughts being successful in modifying the original data of the ledger during the transmission in the network.

Hence, with the implementation of Digital Signature, any modification done to the data would reflect in the unmatched digest when compared to the digest from the system (sender). This works because the digest is obtained from the same hashing algorithm as the sender, which generates the digest generated from the modified data. Henceforth, it is suitable to perceive that the implementation of Digital Signature could provide **security** in the proposed system as it prevents the users from obtaining modified records of their vehicle fine.

Other than that, integrating blockchain in the proposed system enhances **security** with the utilization of asymmetric cryptography. The implementation of asymmetric cryptography uses the ECC encryption algorithm that would encrypt and decrypt the National ID number and the Vehicle Number entered in the proposed system by the system user. This allows only the proposed system (receiver) and the system user (sender) to know the content of the transmitted sensitive data.

In terms of **performance**, the proposed system improves upon a three-tier client server architecture designed to provide

vehicle fine records checking service as the public ledger can be distributed across a number of nodes from which the users can perform searching records. This drastically reduces the load on the central servers and eliminates the risk of disruption of business operations that might occur from a single point of failure.

B. Engineering Practices

In Table II we outline the engineering practices we adhered to with description.

Practices	**Description**
Modularity	We have divided the project into modules for easier maintenance of code. For each feature implementation, we have divided the code to at least one class file. This have enabled us to reuse the codes and maintain a proper structure.
Clean code	We tried to follow clean code conventions but choosing appropriate variable names and division of methods for handling each sub task. Additionally we have included comments to enhance readability of our codes.
Version Controlling	Since the development and testing was distributed among the team members, we used version controlling for effective collaboration. The repository was updated and maintained and made public for review.
Project Management	We divided the tasks of the project and distributed handling the aspects of our project according our individual strengths. This promoted synergy among the team members. We also used tools like Gantt Charts to schedule the activities as effectively as we could.

TABLE II: Engineering Practices Followed with Description.

VI. Conclusion

With our proposed security model we can provide benefits to the user in terms of transparency, immutability, security, decentralization, improved performance. It will make collaboration between various stakeholder secured and smooth. This will put more trust in the system which will be beneficial especially for the government organizations should they implement it. Without any drawbacks, the data can be used by various agencies that can analyze the data and gain valuable insights that might be useful for shaping policies, reducing rates of traffic offenses, etc.

References

[1] Z. Cui et al., "A Hybrid BlockChain-Based Identity Authentication Scheme for Multi-WSN," in IEEE Transactions on Services Computing, vol. 13, no. 2, pp. 241-251, 1 March-April 2020, doi: 10.1109/TSC.2020.2964537.

[2] S. Zhu, Z. Cai, H. Hu, Y. Li, W. Li, zk Crowd: a hybrid blockchain-based crowdsourcing platform. IEEE Trans. Ind. Inf. 16(6), 4196–4205 (2020)

[3] Wang, C., Cheng, X., Li, J. et al. A survey: applications of blockchain in the Internet of Vehicles. J Wireless Com Network 2021, 77 (2021). https://doi.org/10.1186/s13638-021-01958-8

[4] S. King and S. Nadal, "PPCoin: Peer-to-Peer Crypto-Currency with Proof-of-Stake", 2012, [online] Available: http://peercoin.net/assets/paper/peercoin-paper.pdf.

[5] H. Watanabe, S. Fujimura, A. Nakadaira, Y. Miyazaki, A. Akutsu and J. Kishigami, "Blockchain contract: Securing a blockchain applied to smart contracts," 2016 IEEE International Conference on Consumer Electronics (ICCE), 2016, pp. 467-468, doi: 10.1109/ICCE.2016.7430693

[6] Sullivan, Nick. "A (Relatively Easy to Understand) Primer on Elliptic Curve Cryptography." The Cloudflare Blog, The Cloudflare Blog, 15 Feb. 2019, blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/.