BlockMatrix User Manual

National Institute of Standards and Technology

Description

This java package serves as implementation of a transaction system implemented similarly to a blockchain, but which uses a "blockmatrix" data structure to allow deletion. It allows local use of a blockchain which uses a blockmatrix.

Users can:

- Create wallets
- Pass funds along with a message to other user's wallets
- Modify messages passed along with transactions after blocks have been published

For any questions regarding the package, please contact:

Arsen Klyuev: akliouev1@gmail.com

Structure and Concept

The idea behind this package came from the concern that with increasing privacy regulations, such as the EU General Data Protection Regulation, traditional blockchains will no longer be suitable for organizations use. This is because the GDPR requires that organizations make it possible to remove all personal information of an individual at their request. Due to immutability of traditional blockchains, this is not possible (or, at least, very difficult) to do.

To combat this, I worked on using a newly designed data structure called a block matrix to replace the traditional "chain" in blockchains to allow some level of mutability in blockchains. I have included a link to the paper for reference at the bottom of this section.

In this block matrix "blockchain," each transaction consists of 4 key parts: A sender, a recipient, a value, and a message. The transaction is stored in a block in the blockmatrix. For each row and column in the blockmatrix, the blockmatrix stores a hash code made from the hashes of each block in that row or column.

The hash in a block is generated from the hash of the transactions in that block using a merkleroot.

In a traditional blockchain, we could not modify any of the info in a transaction/any info that has been placed in a block without compromising the integrity of the entire blockchain. This is because traditional blockchains maintain integrity by storing a hash of the contents of the previous blocks header, so if any info in any block changes, the hashes of all subsequent blocks will be invalidated. Using a blockmatrix prevents this from occurring, as each block has 2 hashes ensuring its integrity. For example, below, the block marked X is represented in both hashes $H_{,2}$ and $H_{3,}$.

	0	1	2	3	4	
0						$H_{0,-}$
1						$H_{1,-}$
2						$H_{2,-}$
3			X			$H_{3,-}$
4						$H_{4,-}$
	$H_{-,0}$	$H_{-,1}$	$H_{-,2}$	$H_{-,3}$	$H_{-,4}$	

Figure 1. Block matrix

If the data in block in X was modified, as long as only the hashes $H_{-,2}$ and $H_{3,-}$ were modified, we could be assured that every other block would have at least one row or column hash unchanged, meaning that block has not been modified. This allows us to modify the data in one block without being worried that all the info that has been placed in blocks since that block has been put at risk.

Only the message passed along with transactions can be modified, however, due to the nature of transactions in blockchains. Security is assured by not using a system where wallets have a traditional basic "balance." Instead, users can only pass assets from one wallet to another by "proving" they own an asset they are sending by providing a reference to the transaction in which they received this asset. Just as in traditional blockchains, it is impossible to modify or delete the sender, recipient, or value of a transaction, because if these are modified or deleted, and someone needs to reference that transaction in a transaction they are trying to complete, it would prevent them from doing so.

Nevertheless, this provides use for blockchains where providing messages is promoted or required (similar to Venmo, a mobile application where users can pay each other but must include a message as to what the transaction is for.) It also provides the high level of security and the lack of need for trust that blockchains do, with the additional benefit of being able to modify messages passed along with transactions after they have been sent.

Block Matrix NIST Cybersecurity White Paper:

https://csrc.nist.gov/CSRC/media/Publications/white-paper/2018/05/31/data-structure-for-integrity-protection-with-erasure-capability/draft/documents/data-structure-for-integrity-with-erasure-draft.pdf

Setup

Installation

To install the BlockMatrix package, navigate to out/artifacts/blockmatrix_jar/ and download the JAR file blockmatrix.jar. Include this jar file as an external library for your project.

Dependencies
You will need to import Bouncy Castle:

- bouncy castle: https://www.bouncycastle.org/download/bcprov-jdk15on-159.jar

Once you have done this, import the package with

```
import blockmatrix.*;
```

at the top of your Java file to be able to use the program.

First Steps

1. Create a BlockMatrix blockchain, initializing it with the constructor:

   ```
   public BlockMatrix(int dimension)
   ```

   The **dimension** parameter is where you can specify the size of the BlockMatrix you would like. A **dimension** of N will create an NxN BlockMatrix that will be able to hold NxN – N blocks total (due to the empty diagonal). This value cannot be changed, so it is recommended to make sure the BlockMatrix has more than enough space when you initialize it.

2. Set up the security provider. Suppose the name of the BlockMatrix object we initialized is bm. Set up the security provider like so:

   ```
   bm.setUpSecurity();
   ```

   This will add Bouncy Castle as a provider to the Java Security API we will be using, allowing us to create Elliptic-Curve KeyPairs for our wallets for the wallets Public and Private keys.

3. Create your initial wallet, initializing it with the constructor:

   ```
   public Wallet();
   ```

4. Start up our blockchain by creating the genesis transaction which will give our initial wallet a specified amount of funds.

   ```
   public void generate(Wallet wallet, float value)
   ```

   The **wallet** parameter is the wallet we want to be the recipient of our genesis transaction to be. This wallet will receive **value**, a prespecified amount of the asset (or "coin") you want this wallet to start with.

Once this is done, you have completed all the initial steps of setting up your blockmatrix blockchain! You are free to create wallets, create transactions, create blocks, add transactions to blocks, and add blocks to the block matrix at your own discretion. **Note:** The steps in the above setup should only

be completed once. It is not possible to have multiple blockchains in one program using the package.

Demo

The following is brief example of how this package could be used to send funds between different wallets:

```
bm = new BlockMatrix(5);
bm.setUpSecurity();

//Create wallets:
walletA = new Wallet();

bm.generate(walletA, 200f);
walletB = new Wallet();


Block block2 = new Block();
System.out.println("\nWalletA's balance is: " + walletA.getBalance());
System.out.println("\nWalletA is Attempting to send funds (40) to WalletB...");
block2.addTransaction(walletA.sendFunds(walletB.publicKey, 40f, "Here is 40 coins!"));
bm.addBlock(block2);
System.out.println("\nWalletA's balance is: " + walletA.getBalance());
System.out.println("WalletB's balance is: " + walletB.getBalance());
bm.clearInfoInTransaction(2,0);

System.out.println("\nMatrix is Valid: " + bm.isMatrixValid());
```

What this does is run through the setup, then create a new wallet and send funds to that wallet. Adding the transaction to the new block creates the changes in the wallets, and the block is then added to the matrix.

This program produces the following output:

```
Creating and Mining Genesis block...
Transaction Successfully added to Block
Block Mined: 52da29193e5b9d5c4a64d4af4deed46c854d7d67fc585ba4968e0c226f00e7f0

WalletA's balance is: 200.0

WalletA is Attempting to send funds (40) to WalletB...
Transaction Successfully added to Block
Block Mined: 9b16a5ce6115d6ae9409b19e8473e11db1a5e24719d0494e09875ec7e1e7b529

WalletA's balance is: 160.0
WalletB's balance is: 40.0

Matrix is Valid: true
```

Classes and Functions

The following are the classes of the package with accessible methods.

BlockMatrix

- `public BlockMatrix(int dimension)`
  - Constructor
  - Creates a BlockMatrix of the specified **dimension**. The BlockMatrix can hold (**dimension***dimension**) – **dimension** blocks.

- `public void setUpSecurity()`
  - Sets up Bouncy Castle as our security provider in order to use the Java Security API.

- `public void generate(Wallet wallet, float value)`
  - Creates a genesis block, making a transaction that transfers **value** to **wallet.**

- `public void addBlock(Block newBlock)`
  - Adds **newBlock** to a BlockMatrix

- `public Block getBlock(int blockNumber)`
  - Returns the block in the BlockMatrix specified by **blockNumber**, which is the number of the block in terms of when it was inserted (e.g. 1st block, 2nd block, etc.)
  - Block numbers begin with 1. They are not 0-indexed.

- `public ArrayList<Transaction> getBlockTransactions(int blockNumber)`
  - Returns a list of all transactions in the block specified by **blockNumber**.

- `public void clearInfoInTransaction(int blockNumber, int transactionNumber)`
  - Clears the info in the transaction specified by **transactionNumber** in the block specified by **blockNumber**.
    - For example, to clear the 1st transaction in the second block, type `clearInfoInTransaction(2, 1)`

- `public int getInputCount()`
  - Returns the number of blocks that have been added to the BlockMatrix.

- `public String[] getRowHashes()`
  - Returns an array of all the row hashes of the BlockMatrix.

- `public String[] getColumnHashes()`
  - Returns an array of all the column hashes of the BlockMatrix.

- `public float getMinimumTransaction()`
  - Returns the value of the minimum transaction of the BlockMatrix

- `public void setMinimumTransaction(float num)`
  - Changes the value of the minimum transaction of the BlockMatrix to **num**.

- `public int getDimension()`
  - Returns the dimension of the BlockMatrix.

- `public ArrayList<Integer> getBlocksWithModifiedData()`
  - Returns a list of the blocks (by number) that have had their data cleared after being added to the BlockMatrix.

- `public void printRowHashes()`
  - Prints the row hashes of the BlockMatrix.

- `public void printColumnHashes()`
  - Prints the row hashes of the BlockMatrix.

- `public void printHashes()`
  - Prints all hashes of the BlockMatrix.

- `public Boolean isMatrixValid()`
  - Returns true or false depending on whether the BlockMatrix has been tampered with and if it is or is not still secure.

Block

- `public Block()`
  - Constructor
  - Creates a Block

- `public boolean addTransaction(Transaction transaction)`
  - Adds a transaction to the block. Returns true if the transaction is added successfully, and false if not.

- `public ArrayList<Transaction> getTransactions()`
  - Returns a list of all transactions in the Block.

- `public String getHash()`
  - Returns the hash of the Block.

- `public void printBlockTransactions()`
  - Prints transaction details of each transaction in the Block.

Transaction

- `public int getBlockNumber()`
  - Returns the number of the block this Transaction is stored in.

- `public String getTransactionId()`
  - Returns the id of this Transaction. The id is also the hash of the Transaction.

- `public PublicKey getSender()`
  - Returns the PublicKey of the sender of this Transaction.
- `public PublicKey getRecipient()`
  - Returns the PublicKey of the recipient of this Transaction.
- `public float getValue()`
  - Returns the value of this Transaction (the amount being sent.)
- `public String getInfo()`
  - Returns the info/message being passed along with this Transaction.
- `public byte[] getSignature()`
  - Returns the signature of this Transaction.
- `public ArrayList<TransactionInput> getInputs()`
  - Returns a list of all inputs of this Transaction.
- `public ArrayList<TransactionOutput> getOutputs()`
  - Returns a list of all outputs of this Transaction.

TransactionInput

- `public String getTransactionOutputId()`
  - Returns the id of the TransactionOutput this TransactionInput is referencing.
- `public TransactionOutput getUTXO()`
  - Returns the unspent TransactionOutput this TransactionInput is using.

TransactionOutput

- `public String getId()`
  - Returns the id of this TransactionOutput. This id is the hash of the TransactionOutput.
- `public PublicKey getRecipient()`
  - Returns the PublicKey of the recipient, the new owner of the coins from this TransactionOutput.
- `public float getValue()`
  - Returns the amount of the asset in this TransactionOutput.
- `public String getParentTransactionId()`
  - Returns the id of the Transaction this output was created in.

Wallet

- `public Wallet()`
  - Constructor
  - Creates a Wallet.

- `public float getBalance()`
  - Returns the balance of this wallet.

- `public Transaction sendFunds(PublicKey recipient, float value, String info)`
  - Returns a Transaction which sends **value** funds from this wallet to the wallet specified by **recipient**, along with the message **info**.

- `public PublicKey getPublicKey()`
  - Returns the PublicKey of this wallet.

- `public HashMap<String, TransactionOutput> getUTXOs()`
  - Returns a HashMap of the unspent TransactionOutputs owned by this wallet. Keys are the TransactionOutput ids, whereas the values are the TransactionOutputs themselves.