
TrustChain Android Documentation

Release 0.0

Wilko Meijer, Rico Tubbing, Jonathan Raes

Jun 28, 2018

1	Introduction	1
2	What is TrustChain?	3
3	Creating a block	5
3.1	Structure of blocks	5
3.2	Create block	6
3.3	Validate block	6
3.4	Sending a block	7
3.5	Links to code	7
4	Connection between peers	9
4.1	Connections overview	10
4.2	Background handling of peers	10
4.3	Background handling of messages	11
4.4	Message transmission	11
4.5	Networking classes and their responsibilities	12
4.6	Stress Testing	12
4.7	Statistics	12
4.8	Links to code	13
5	Inbox	15
5.1	Why an inbox?	15
5.2	How does it work in the app?	15
6	Chain Explorer	19
6.1	Links to code	23
7	Wallet	25
7.1	Tokens	25
7.2	Import of tokens from PC	25
7.3	Import/Export of tokens between phones	25
7.4	QR code	26
7.5	Example	26
7.6	Links to code	26
8	Passport	29

8.1	Implementation	30
8.2	Links to code	33
9	Message structure (Protocolbuffers)	35
9.1	Making changes	35
9.2	Complete structure	35
9.3	Links to code	37
10	Local chain storage (database)	39
10.1	Database structure	39
10.2	Links to code	40
11	Crypto	41
11.1	(De)serialization	41
11.2	Links to code	41
12	Claims, attestation and zero knowledge proofs	43
12.1	Zero knowledge proofs	43
12.2	Implementation	43
13	Maturity of Code	45
13.1	Code coverage	45
14	Installation Instructions	47
14.1	Installing TrustChainAndroid APK	47
14.2	Setting up the Android Project	47
15	Contact and links	49
15.1	Contact	49
15.2	Useful links	49

CHAPTER 1

Introduction

TrustChain Android is a native Android app implementing the TU Delft style blockchain, called *TrustChain*. This app provides an accessible way to understand and to use TrustChain. The app is build as part of a Blockchain Engineering course of the TU Delft. It is meant as a basic building block to experiment with blockchain technology. The app also demonstrates a *network overlay* for peer to peer communication, and a way to *communicate* with the chip in the Dutch ID-card/passport. This documentation should get you started in the workings of the app, however for thorough understanding, reading other documentation and looking at the source code is a necessity.

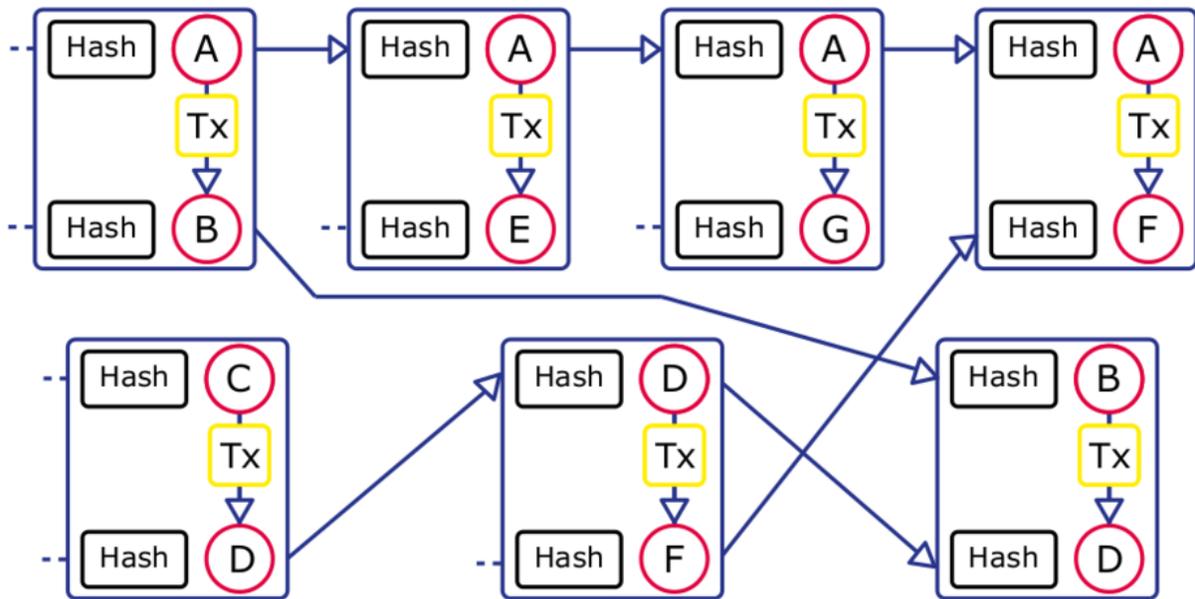
We have tried to make the code clear. However, this app was not build by Android experts so please don't hold any mistakes or weird structures for Android against us. Instead, please let us know what could be improved, or provide a fix yourself by submitting a pull request on [GitHub](#).

What is TrustChain?

In this section a short explanation of TrustChain is given. For a more extensive and complete overview please refer to the publications that can be found on the [blockchain lab website](#).

TrustChain is designed to deliver scalable distributed trust. This is achieved by letting two parties agree to a deal or transaction and storing the proof of the transaction in an immutable way. The transaction is stored in a block and a series of blocks are chained together to form a blockchain. To achieve scalability, each peer keeps track of their own blockchain which holds all the transactions they participated in. So when two peers want to agree on a transaction peer A creates a block with the transaction, signs it, adds it to their chain and sends it to peer B. Then peer B creates a block based on the block it received from A, signs it, adds it to their chain and sends it back to peer A. Both parties now have a proof of their transaction in their chain and they also have a copy of the block from the other party. For a more detailed description of how a block is created see *Creating a block*.

Many transactions by many peers thus create a complicated network of entangled chains. However, the size of the chain of one peer is entirely dependent on the amount of transactions they participate in, so the whole system scales very well. The following picture tries to make the entanglement more clear. Each unique letter in the image represents a unique peer.



Creating a block

In order to complete a transaction between two parties a block needs to be created. A block in TrustChain is a little different than in bitcoin-style blockchains. In bitcoin-style blockchains, a block is a collection of transactions that happened in the network. A block is created by a node and is propagated through the network. All connected nodes validate the block and the transactions. In TrustChain a block is formed by two peers whom wish to agree on a transaction. Therefore a TrustChainBlock only has one transaction.

Both parties need to agree on a transaction, so there has to be some interaction. The way this is done in TrustChain is to first create an incomplete block, called a block proposal. This block proposal is send to the second party, whom completes the block and sends it back to the first party. Now both parties have proof that the other party agrees to the transaction. This process is explained in more detail below.

3.1 Structure of blocks

A block has the following attributes:

- `public_key` - The public key of the peer that created this block
- `sequence_number` - Represents the position this block has in the chain of the creating peer
- `link_public_key` - The public key of the other party
- `link_sequence_number` - The position the connected block has in the chain of the other party
- `previous_hash` - A sha256 hash of the previous block in the chain
- `signature` - The signature of the first peer on the sha256 hash of this block
- `transaction` - The data that both parties need to agree on, the input is simply a series of bytes so this can be anything, from text to documents to monetary transactions

Note that `link_sequence_number` will be unknown for the created block proposal because peer A won't be sure when peer B inserts the linked block in his chain. This will stay unknown, as updating a block already in the chain is not desirable, since it might invalidate later blocks. When the block is completed peer A will have the block of peer B in its database as well, so it can always find out the position of the linked block in peer B's chain.

3.2 Create block

There are two situations that require creating a block. Initiating the creation of a transaction with another peer and completing a block that was sent to you by another peer. This is both done using the `TrustChainBlockHelper`. This class contains methods for creating, signing, and validating blocks.

3.2.1 Initiating a transaction

To initiate a transaction some information is needed: the bytes of the transaction, the initiating party's public key, the public key of the other party, and a link to the database containing your chain to `createBlock`. The latest block in your chain will be retrieved from the database, to be able to set `sequence_number` and `prev_hash`. `link_sequence_number` will remain empty and the hash of this block is calculated and signed in order to complete the block proposal. The block proposal can now be added to the local chain and sent to the other party.

3.2.2 Responding to a block proposal

When a block proposal is received the receiving party B can decide whether or not they agree with the transaction in the block and choose to either sign the block or ignore it. After the block is validated, party B knows that this proposal is consistent with what is known of the sending party A. In order to complete the block, a new block is created. `public_key` and `sequence_number` from the block proposal will now be `link_public_key` and `link_sequence_number`. `public_key`, `sequence_number`, and `prev_hash` will be set according to the current state of party B's chain. `transaction` will remain the same and a new hash is calculated that will be signed by party B.

3.3 Validate block

Block validation is the most important step here, as this ensures the validity of the blockchain. There are 6 different validation results:

- `VALID`
- `PARTIAL` - There are gaps between this block and the previous and next
- `PARTIAL_NEXT` - There is a gap between this block and the next
- `PARTIAL_PREVIOUS` - There is a gap between this block and the previous
- `NO_INFO` - We know nothing about this block, it is from an unknown peer, so we can say nothing about its validity
- `INVALID` - It violates some rules

The validation function starts off with a valid result and will update the validity result according to whether the rules hold for the block. Validation consists of six steps:

- Step 1: Retrieving all the relevant blocks from the database if they exist (previous, next, linked, this block)
- Step 2: Determine the maximum validity level according to the blocks retrieved in the previous step
- Step 3: Check whether the block is created correctly, e.g. whether it has a sequence number that comes after the sequence number of the genesis block
- Step 4: Check if we already know this block, if so it should be the same as we have in our database
- Step 5: Check if we know the linked block and check if their relation is correct

- Step 6: Check the validity of the previous and next block

For a more detailed explanation of the validation function, please take a look in the code and try to understand what happens there.

3.4 Sending a block

There are *two methods* for sending a block to another party:

- Via Internet, using the network overlay
- Offline, using either QR codes or Android beam

Note that offline sending will add block to the regular chain, so they will get propagated through the network when the peer is online again.

3.5 Links to code

- Block structure in ProtocolBuffers ([Message.proto](#))
- All block related methods ([TrustChainBlockHelper.java](#))

Also see the [readme on the ipv8 github](#)

Connection between peers

In order to be part of the network, you need to be connected to other people (peers). The underlying protocol for finding peers was taken from [App-To-App Communicator](#). This network overlay enables a peer to discover more peers through peers they are already connected with. By default firewalls will block incoming connections from unknown sources. The protocol makes use of [UDP hole punching](#) to circumvent this and setup direct connections between peers. Basically this uses a mutual friend to initiate a connection between two peers who are unknown to each other.

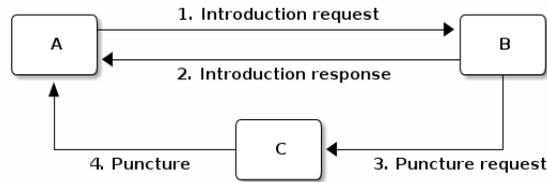
The protocol consists of the following messages:

- `IntroductionRequest` - Request for an introduction to some new peer.
- `IntroductionResponse` - Response with a new peer that could be connected to.
- `PunctureRequest` - Request to send a puncture to some peer.
- `Puncture` - A basically empty message, used to punch a hole in the NAT/firewall of the sender.

When a peer initially starts up the app, it needs to find other peers to connect to. For this initial peer discovery a bootstrap server is used. This bootstrap server resides on a fixed IP and introduces new peers to each other when they initially connect to the network.

The following steps explain how a typical introduction goes down:

1. Peer A knows of some peer B and asks for a new connection to be setup.
2. Upon connection peer B chooses a peer from its active connections, peer C, and sends the address of peer C to peer A as an introduction response message.
3. Peer B sends peer C a puncture request for peer A.
4. Peer C sends a puncture message to peer A to punch a hole in its own NAT/firewall, allowing the incoming connection from peer A.



4.1 Connections overview

After starting the app and choosing a username a screen is shown with an overview of all current connections. The screen is divided into two columns. The left column shows peers which are actively connected with you. The right screen shows peers, which you have heard about, but have yet to respond to you. They have been introduced to you through the `IntroductionRequest` message.

In order to prevent sending messages indefinitely to peers who aren't responding peers can time-out, the bootstrap server will never time-out. Every five seconds a message is sent to 10 random peers asking for an introduction to more peers. This is limited to 10, because otherwise a large network will be sending a lot of traffic. (imagine 100 peers sending about 1kb size message to 100 other peers every 5 seconds for an hour) These introduction requests also act as a heartbeat timer keeping track of which peers are still alive. If no messages were received from a peer for 25 seconds the connection is deemed to be lost and the peer will be removed from the list. A new introduction would have to take place to reconnect to the peer.

For each peer the following information is displayed and updated every second:

- Status light
 - green when a message was received in the last 15 seconds
 - orange when peer is new, but hasn't responded yet
 - red when no message was received in the last 15 seconds, a strong indicator that the connection is lost
- Username - The username the peer chose when first starting the app
- Connection indicator - Information about the type of connection, e.g. WiFi or mobile. In case the peer is set as a bootstrap server next to the username the word "Server" is displayed
- Ip address - public ip address of the peer, plus the port of the connection
- Last received and sent - In the bottom row two timers can be found indicating the time since the last message was received from and when the last message was sent to the peer.
- Received and sent indicators - The UI shows when messages are sent to a peer by changing the color of the last received and last sent timers when a message is received in the past 500 ms.

4.2 Background handling of peers

All operation that are done on peers are done through or make use of the `PeerHandler` class. This class holds an arraylist of peers currently known and can split these peers into two lists of active peers and new peers. Peers are kept in memory for now, so each time the app is closed all peers have to be rediscovered through the bootstrap server.

There are two ways the host can hear about new peers:

- When a message is received, the sending peer becomes known
- An `introductionResponse` message, which contains a list of peers that the sending peer is connected with

Note that in the latter case a puncture request is still only sent to one peer (the invitee).

Each time a peer is 'introduced' through either method the `getOrCreatePeer` function of the `PeerHandler` is called. This function does one of four things:

- The peer is already known, so the object associated with this known peer is returned
- The peer is already known based on its id, but with a different `InetAddress`, the associated object is updated and returned
- The peer is already known based on its address, but with a different id, the associated object is updated and returned
- The peer is unknown, a new `Peer` object is created, added to the `PeerList` and returned

This way a host keeps track of all peers that ever interacted with them and keeps the information up-to-date.

For each peer the following information is stored:

- `address` - the `InetSocketAddress` where this peer can be reached
- `peerId` - the identifier of this peer
- `lastSendTime` - the last time a message was send to this peer
- `lastReceiveTime` - the last time a message was received from this peer
- `long creationTime` - the time this peer was first introduced to this host

Every second the peer list is checked for dead peers. Dead peers are peers from which no message was received in the last 25 seconds. These dead peers are removed from the peerlist.

4.3 Background handling of messages

Since all messages are created using `protocolbuffers`, it is easy to rebuild them on reception. When a message is received, the message type is checked and the appropriate functions are called to further handle the message. Messages not build with (the correct) `protocolbuffers` will simply be discarded.

4.4 Message transmission

4.4.1 Network

Messages are sent over the network using UDP datagrams. Currently, one message is sent in one datagram, putting an upper limit on the message size of 65KB, the maximum UDP datagram size. The message is sent in the `Network` class' `sendMessage` method.

4.4.2 Offline transmission

The app also provides functionality to transmit blocks offline. A checkbox in the `PeerSummaryActivity` activates this functionality. Offline sending can be performed in two ways, one is using QR codes and the other is using *Android Beam*.

QR code transmission uses the QR generator located in the `funfs.qr` package. The maximum size of data contained in a QR code is limited to about 3 kilobytes, so this has an even stricter maximum size then the network's UDP transmissions.

Android Beam transmits data using either NFC or Bluetooth, allowing a practically infinite maximum message size (no physical maximum has been identified), although anything above 65KB will create problems when announcing the chain to other peers. Of course both the sending and receiving party need to support the required technologies.

As hinted above, the blocks exchanged offline do not stay offline, but are automatically shared with all connected peers if there is an internet connection. Therefore the offline send feature should only be used when there is no network connection available, not for keeping the exchanged block secret from other peers.

4.5 Networking classes and their responsibilities

There are two main classes which have to do with networking. [Network](#) and [MessageHandler](#).

The Network class is a singleton class and is responsible for sending and receiving messages. It has a datagram channel which has a socket bound to a local port (default 1873). Through this channel messages are sent and received to and from peers. The network class has methods to build the different messages of the protocol.

The MessageHandler is responsible for handling the messages after they have been deserialized. It decides on how to respond to a received message.

4.6 Stress Testing

The stress testing feature allows to spin up any desired number of nodes. These nodes start in the *StressTestNode* [<TODO_add_link>](#) class, they use their own instances of the *Network* class and provide the *StressTestNode* instance as a PeerListener, instead of the normal *OverviewconnectionActivity*. They act just like a normal node, except having no visual displays and generating a new temporary keypair.

4.7 Statistics

The Network class logs all its sent and received messages into the singleton *StatisticsServer* [<TODO_add_link>](#) class. Messages are logged by type and separate statistics are kept for all running nodes. Additional logged data is the number of sent and received bytes and the number of active and new connections. All this data is tallied and shown on the StressTestActivity, updated periodically.

- Statistics are NOT displayed on phones running API 23 or less, as tallying uses a reduce function not available on API 23 or less.
- Note that running many nodes may slow the phone down dramatically and statistic updates may come in extremely slowly or not at all.

4.7.1 Logging and graph generation

The StatisticsServer class runs a logging task that prints all statistics, in csv format, to the console at a fixed time interval. This csv data can be used to generate graphs to visually show the data using `generategraph.py` from the python folder, using the following steps:

1. Get data of the desired node by filtering the log output on 'Statistics-<desired username>'
2. Copy the data to a text file in the python folder (the timestamps can be left in) and change the filename in the python script
3. Run 'python generategraph.py <column name>' to generate a graph for the given node and column

example: 'python generategraph.py messagesSent messagesReceived'

In order to take all nodes into account generategraph_aggregate.py is provided. This script takes a data file that contains the logs of an arbitrary number of nodes, and calculates the averages and standard deviations of the requested columns. - Please note that this script is more hacky than generategraph.py, may contain bugs and does not offer custom ticks on the x and y axis. - Please also note that this script has 'stress_test_user_0' hardcoded as node name and (ab)uses this on order to fill the array of x values. In order for this script to work reliably, the hardcoded username value should always be the first log in each series of log updates. - Make sure that the resulting file used to create the graph contains only one header line containing the column names.

The steps to create graphs are the same as above, except that the recommended filter is 'Statistics-stress_test' in order to filter out the main node (including it will cause some issues since this node is started much earlier than the rest)

Conclusion

A problem of the current implementation is that each connected peer sends around one kilobyte of data every 10 seconds. Suppose we have 100 clients which are all connected, then this would mean we would send $100 * 1 * 6 = 600$ kilobytes every minute. If a device is on a 4G connection this would consume too much data.

A solution to this problem could be to send less messages. For example, it is possible to send every minute a message to a peer instead of every 10 seconds. This could work since it is not necessary to check if a peer is alive every 10 seconds. Another solution is to send smaller messages over the network. However, protocol buffers is used to generate, send, and receive the messages, removing protocol buffers means a lot of new code has to be added to generate, send, and receive the messages.

Another problem is that messages are send to 10 random peers. A better solution is to send messages to peers we have not send a message in a while. A further improvement could be to send a message to more than 10 peers every 10 seconds, but ultimately this does not scale well.

4.8 Links to code

- [Network class \(Network.java\)](#)
- [Message handling \(MessageHandler.java\)](#)
- [Offline sending and receiving](#)

The inbox is the place where you can interact with peers. The inbox gives an overview of all the peers that have sent blocks to you and you can add peers manually by clicking on them in the connection overview. When a peer is added to the inbox you can click on it to sign blocks sent to you, send new blocks, and inspect their chain using the *Chain Explorer*.

5.1 Why an inbox?

In the first version of this app, it was only possible for the user to send/sign a block when they had a live connection with the peer that they wished to communicate with. The drawback of this design was that both the user and the peer needed to be online at the same time. This was cumbersome for the users, as they would have to tell each other to come online each time that they wanted to send a block to each other. With the inbox, no live connection is needed between users. All incoming blocks are stored locally and users can review them later at their convenience.

5.2 How does it work in the app?

In the main screen of the app, the user can add a peer to their inbox by clicking on an active peer. This is shown in [Fig. 5.1](#).

After a peer has been added to the inbox, the user can open their inbox by clicking on the button that is located on the bottom of the screen. The user can see the peers that they have added to their inbox and the peers that have sent blocks. This is shown in [Fig. 5.2](#).

Each peer in the inbox is displayed in a list item (see [Fig. 5.3](#)) with the following properties:

- Colored circle, indicating peer status
- Peer username
- Peer IP address combined with port
- Mail icon with indicator if there are unread blocks

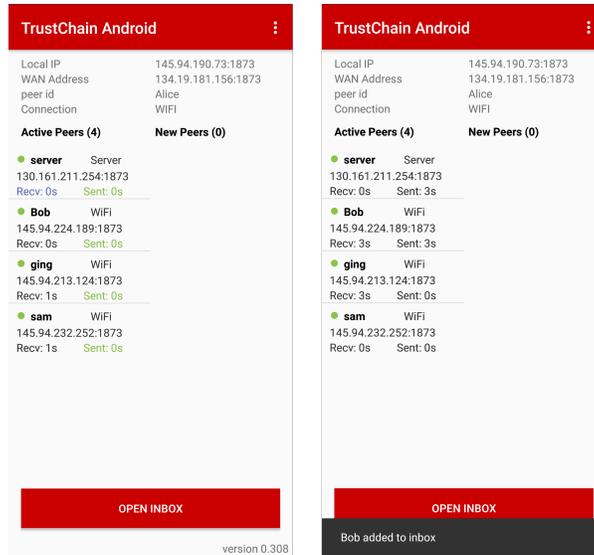


Fig. 5.1: Adding a peer to the inbox

The user can then click on the peer to send a block as shown in Fig. 5.4.

The user can also see all the blocks that they have sent to a peer and blocks that they have received from a peer. The user can sign blocks that have been received as shown in Fig. 5.5.

5.2.1 Links to code

- [The inbox package](#)

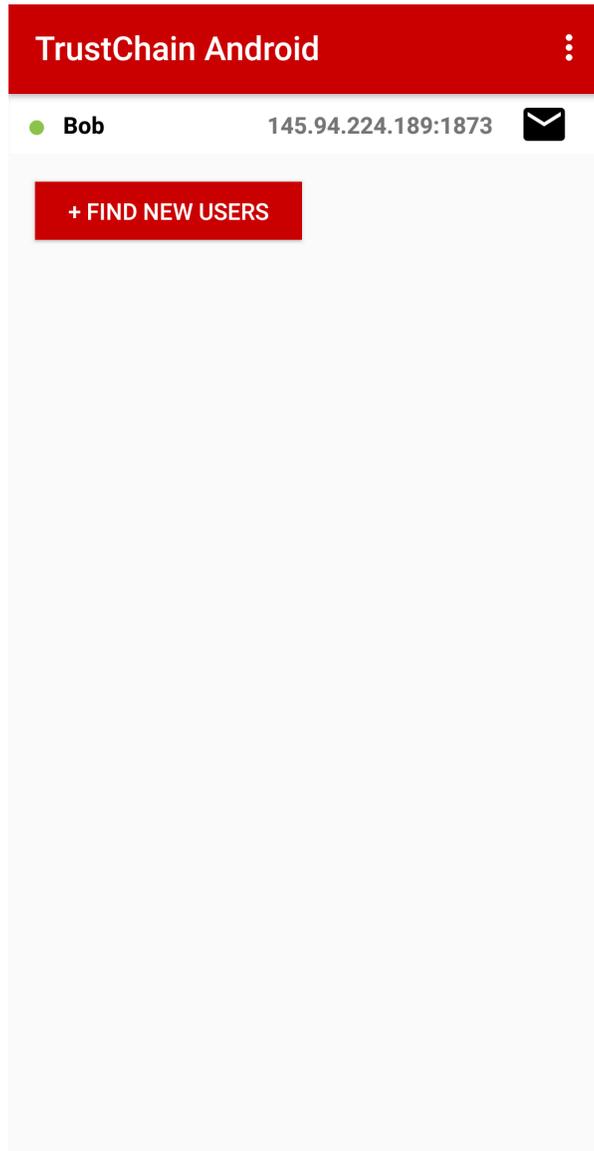


Fig. 5.2: Inbox of the user



Fig. 5.3: Example inbox item

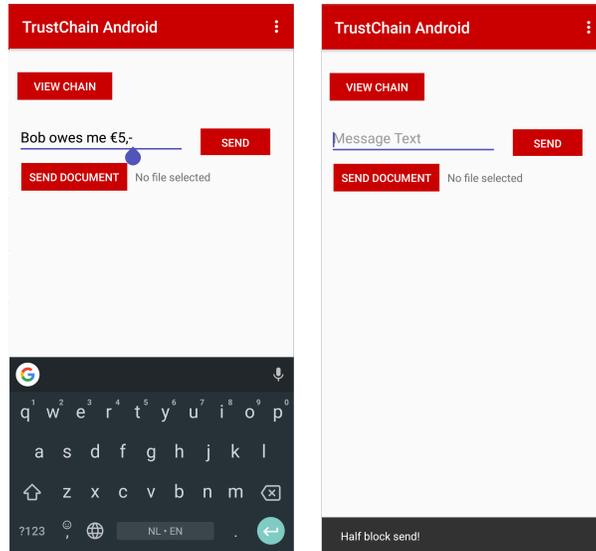


Fig. 5.4: Sending a block to a peer

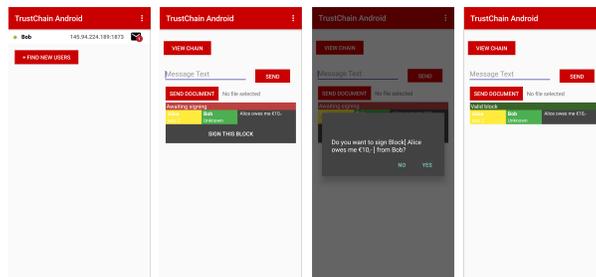
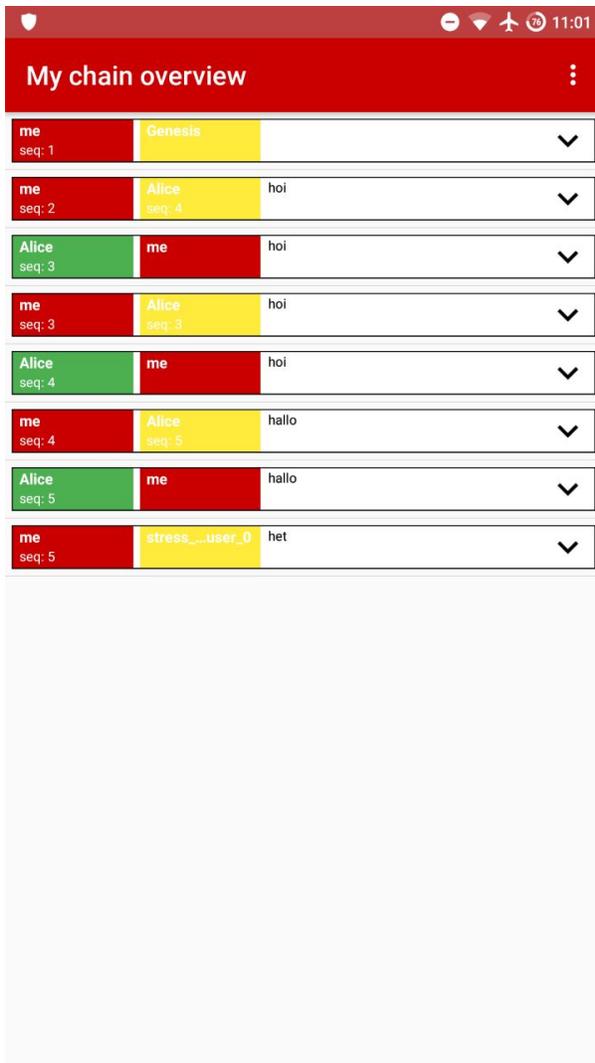


Fig. 5.5: Viewing received blocks and signing of a block

The chain explorer has several functionalities:

- Show all (known) blocks in a chain
- Show the content of a block such as public key, link public key and the transaction
- Click on other public keys to explore other chains
- Open files from a block with a file

Viewing another chain can be done by clicking on the arrow on the right side of a block. This will expand the view and show information about the block as shown in Figure 2. If one of the public keys is pressed the chain explorer will load the chain of this public key as shown in Figure 3. Files can be opened by pressing the text 'Click to open' as shown in Figure 4.



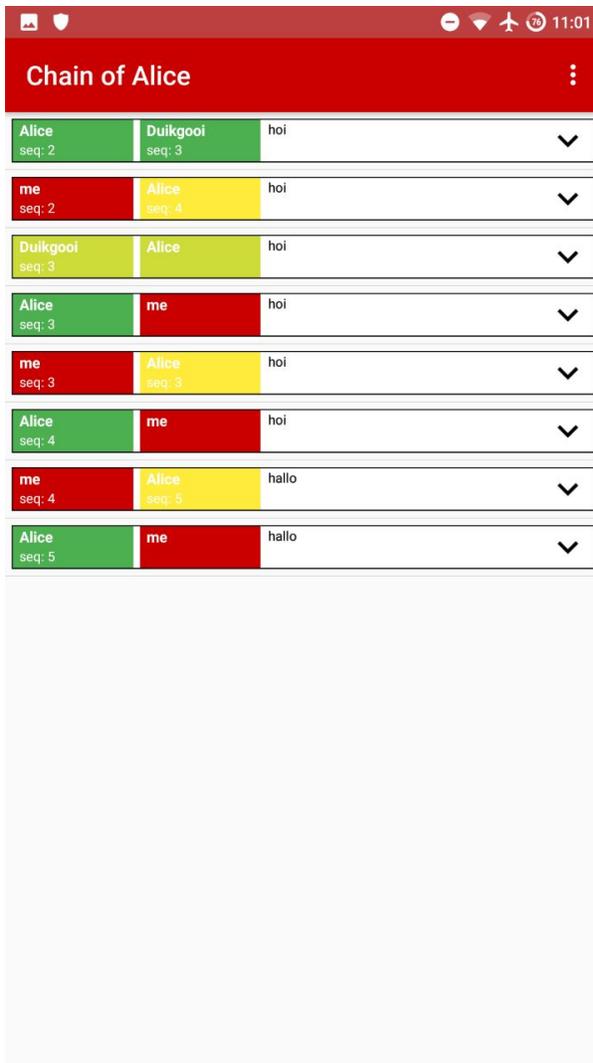
My chain overview

me seq: 1	Genesis		▼
me seq: 2	Alice seq: 4	hoi	▼
Alice seq: 3	me	hoi	▼
me seq: 3	Alice seq: 3	hoi	▼
Alice seq: 4	me	hoi	▼
me seq: 4	Alice seq: 5	hallo	▲

Public key: 4C69624E61434C506B3A2B64F5225E1C8F2849984527E2150EEA
A009C3A3D9A218C0A36F603FB4D124513838958CEE782E556B08
056ADABC1A1AF210E739A146704B2E9176566983D975
Link public key: 4C69624E61434C506B3A8C5A59D1EAE2AAD95E8D379655837B
F3E8B923072E9F6E2F44692DDE966C856B19B660FE358F98A4C5
A89500EB281AA5610BC0E63D530CFC04F42A25072846D
Previous hash: 9D1CD03A045433ECF1924876649DBC24AD572CFC960B30289A
511B4E3468C2B
Signature: 167CA7F1009BC566154BAA34CD5827D7360D10C5E1CCE63AB93
448AFF97528A04216AEF640BA060314E19ECBB721D8541B8FA21
7791DCBF568EC1EF7881B8D0B
Transaction: hallo

Alice seq: 5	me	hallo	▼
me seq: 5	stress...user_D	het	▼

The left image shows all the block in the chain, the right image shows the content of block.





The left image shows the chain of a different user than the device owner, the right image shows a block which contains a file.

6.1 Links to code

- Chain explorer activity
- Chain explorer adapter

The wallet management works with a Tribler concept, called the reputation of each user. This reputation is based on the amount of data uploaded and downloaded, and the simple subtraction of these two quantities provides a number representing it. This number gives a positive value when the user is uploading more content than downloading, therefore contributing positively to the overall system. It will be referred in terms of *tokens*, that account for the reputation of the user, and can be transferred.

7.1 Tokens

Each transaction holds the values up, down, total_up and total_down. When receiving the first half of a block, the receiver flips up and down, and sets the total_up and total_down for itself based on the content of the transaction.

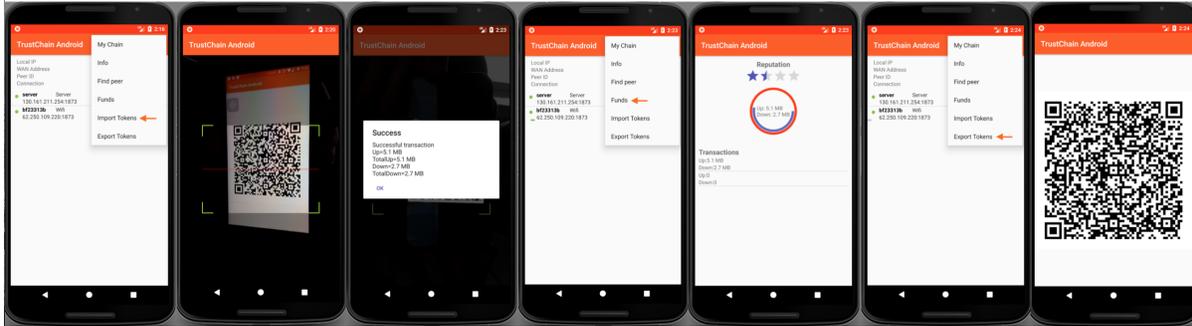
7.2 Import of tokens from PC

This feature is done by generating a throw-away identity in the PC, to which a chosen amount of Tribler tokens are transferred. Then, this identity is exported to the phone by printing a QR code with the necessary information in the screen, and scanning it with the TrustChain Android app. To this mean, an option called “Import tokens” is implemented in the app’s menu. Once both identities are settled in the phone, the final transfer is performed and only the phone identity outlasts. Once this process has been completed, the phone has successfully received the tokens from Tribler and they can be checked in the “Funds” menu option. However it must be clarified that, as it is an offline process, there are no guarantees for preventing the double-spending problem.

7.3 Import/Export of tokens between phones

The functioning is exactly the same as the import of tokens PC-phone, only that now it is performed between two phones using the Trustchain Android app. In order to go through with it, the sender uses the menu option “Export tokens”, which displays in the screen a QR code with all the necessary to export the total amount of tokens. Then,

the receiver should make use of the “Import tokens” option to scan it and complete the transaction. As it is an offline process, same security concerns as before apply here.



7.4 QR code

The QR code has to be able to transfer the throw-away identity, while having a total size small enough to make it readable. To that effect, only the essential information is included in it:

- Private key of the throw-away identity, which includes the private key special crypto format (explained in the *Crypto* section).
- Transaction object, with the up and down quantities.
- Block hash and sequence number belonging to the half block of the transaction between the sender identity and the throw away one.

This information is encoded in a JSON string before being put into the QR code.

Once the QR code has been read, the receiver uses this information to reconstruct the transaction and throw-away identity. The QR code used is the [version 13](#), which has a capacity between 1440-3424 data bits depending on the ECC level.

7.5 Example

An example of the data in the QR code is the following:

```
{
  "private_key":
  ↪ "TGLiTmFDTFNL0vHyazzyYvb00cdAIb+xmDUzflFOsnzYTm3vbFcRV0FfuxWh827LrDLxYljG5+ga\n/
  ↪ m0SUKDYcDiHRnuf5BQ1HAI=\n",
  "transaction": {"down": 0, "up": 11114175918},
  "block": {"block_hash": "7Jh0+S93fbtoqWwKQlYmsPMjC8eU7Bzo91NaKy/0d0w=\n",
  ↪ "sequence_number": 1}
}
```

Which will result in the following QR code:

7.6 Links to code

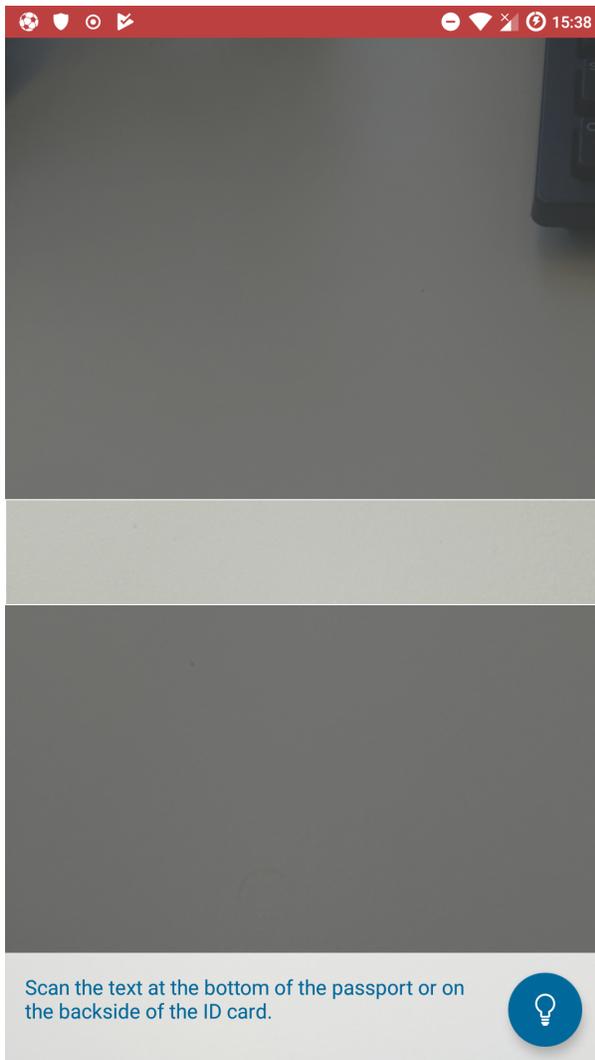
- [The funds package](#) - holds the main wallet classes



-
- [Classes for importing and exporting funds using QR codes](#)

8.1 Implementation

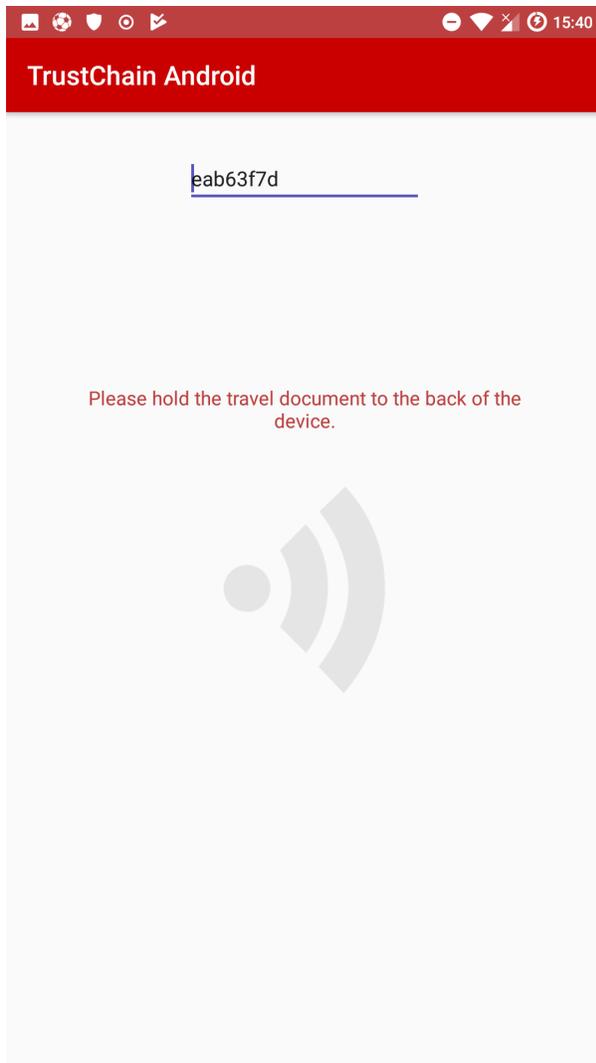
A connection with the passport is setup by the open source library [jMRTD](#). The logic for opening the connection, performing BAC and AA are mainly implemented in the class `PassportConnection`, which is used by the class `PassportConActivity`. Reading the MRZ can be done in two different ways: OCR or manual input. The library that is used for OCR is called Tessarect. The package `ocr` (in the passport package) is responsible for handling all OCR related work, such as opening the camera and providing images to perform OCR. The class `ManualInputActivity` is responsible for handling the manual input of data for BAC. The `ManualInputActivity` can only be reached from the `CameraActivity` and when no MRZ has been successfully read by the OCR. The data of a passport holder is passed around via intents. The `DocumentData` class contains the data to perform BAC and is passed from either the `CameraActivity` or the `ManualInputActivity` to the `PassportConActivity`.

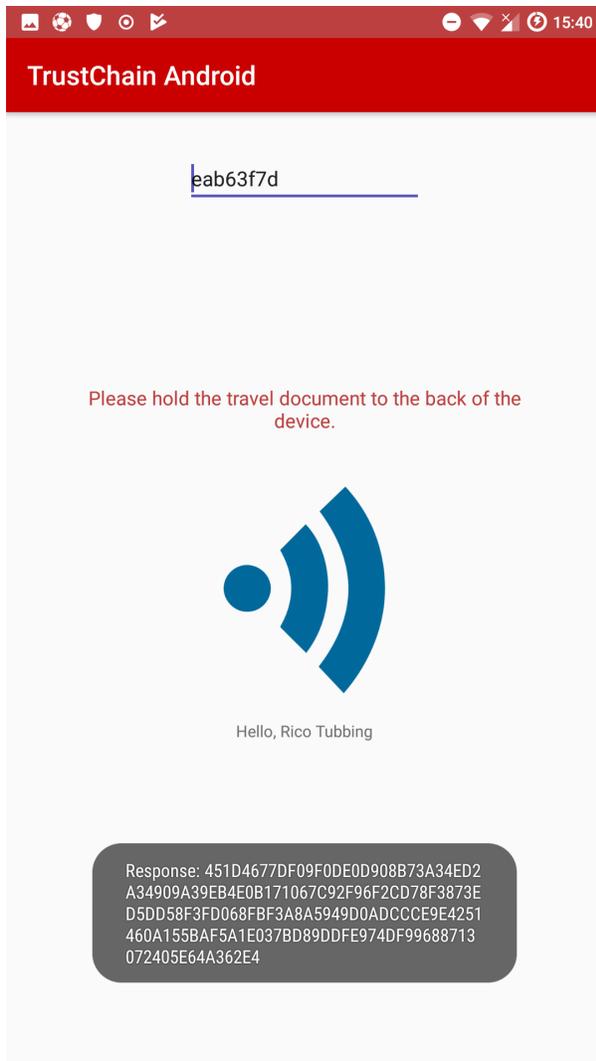


The screenshot shows the TrustChain Android app interface. At the top, there is a red header bar with the text "TrustChain Android". Below the header, the form contains the following elements:

- A label "Document number" followed by a text input field containing "ABC1234".
- A label "Expiry date" followed by three dropdown menus: the first contains "1", the second contains "January", and the third contains "2018".
- A label "Date of birth" followed by three dropdown menus: the first contains "1", the second contains "January", and the third contains "1960".
- A red button at the bottom with the text "SUBMIT".

The image on the left is the CameraActivity and the image on the right is the ManualActivity.





The image on the left shows the PassportConActivity with the hex number 'eab63f7d' is going to be signed. The right image shows the same Activity with the resulting signature in hexadecimal as response from the passport.

8.2 Links to code

- [OCR scanner \(TesseractOCR.java\)](#)
- [Passport nfc connection](#)

Message structure (Protocolbuffers)

Creating a network of TrustChain peers which only run a Java version of TrustChain is not very useful. Therefore the TrustChain blocks and messages should be compatible with many platforms, so cross-platform connection is possible. For the storage of the chain this is achieved by using SQLite, which has implementation for many platforms. For sending messages (blocks and crawlrequests) this compatibility can be achieved by using [Google's Protocolbuffers](#), which is a cross-platform data serialization mechanism. When implementations across platforms use the same Protocolbuffers setup, messaging will across these platforms will be possible.

9.1 Making changes

Protocolbuffers is used to create the structure of messages. This includes all the messages used by the network overlay (*Connection between peers*) and the structures of the Trustchain related objects like `TrustChainBlock` and `CrawlRequest`. A complete overview of the message structure can be found in *Complete structure*. With Protocolbuffers the corresponding Java classes can be compiled, so it is possible to call the structures as objects. Making changes and recompiling the Java classes is quite easy, just follow the [tutorial of ProtocolBuffers](#) and you should be fine. When making changes, don't forget to also update the database structure if necessary.

9.2 Complete structure

The complete structure of all parsable objects as defined in protocolbuffers.

9.2.1 Message

- `bytes source_public_key`
- `string source_name`
- `bytes destination_address`
- `int32 destination_port`

- int32 type
- Payload payload

9.2.2 Payload

- IntroductionRequest introductionRequest
- IntroductionResponse introductionResponse
- Puncture puncture
- PunctureRequest punctureRequest
- TrustChainBlock block
- CrawlRequest crawlRequest

9.2.3 TrustChainBlock

- bytes public_key
- int32 sequence_number
- bytes link_public_key
- int32 link_sequence_number
- bytes previous_hash
- bytes signature
- Transaction transaction
- google.protobuf.Timestamp insert_time

9.2.4 Transaction

- bytes unformatted
- string format
- Claim claim

9.2.5 Claim

- bytes name
- google.protobuf.Timestamp timestamp
- int32 validity_term
- bytes proof_format

9.2.6 CrawlRequest

- bytes public_key
- int32 requested_sequence_number
- int32 limit

9.2.7 IntroductionRequest

- int64 connection_type

9.2.8 IntroductionResponse

- int64 connection_type
- string internal_source_socket
- Peer invitee
- repeated Peer peers

9.2.9 Puncture

- string sourceSocket

9.2.10 PunctureRequest

- string source_socket
- Peer puncture_peer

9.2.11 Peer

- bytes address
- int32 port
- bytes public_key
- string name
- int32 connectionType

9.3 Links to code

- [Structure of message \(Message.proto\)](#)

Local chain storage (database)

Every valid block proposal created is saved locally on the device. Additionally, all of the incoming blocks of other peers, either as a response to a crawlrequest or in other ways, are saved locally when validated correctly. The blocks are saved using an SQLite database. Android has code in place to handle all the complicated parts, so using the database after setup consists mainly of writing queries. Please refer to the [Android tutorials](#) for an explanation on how to use SQLite databases in Android.

The database is set up in a similar way as in the [ipv8 python code](#). The only difference is the added column `TX_FORMAT`. So the database from the `ipv8` implementation in python can be imported trivially into android. The columns correspond to the *Message structure (Protocolbuffers)*, so for inserting it simply needs to parse relevant data from the block. Note that it when receiving raw bytes it always has to be passed to a Protocol Buffers object first before it is added to the database, to ensure that data was received correctly.

10.1 Database structure

The table has the following columns:

- TX - Transaction
- TX_FORMAT - Transaction format e.g. pdf
- PUBLIC_KEY - Base64 encoding of the raw public key pair bytes (see *Crypto*)
- SEQUENCE_NUMBER - sequence number of the block
- LINK_PUBLIC_KEY - Base64 encoding of the public key pair of the linked block
- LINK_SEQUENCE_NUMBER - sequence number of the linked block
- PREVIOUS_HASH - Base64 encoding of the hash of the previous block in the chain
- SIGNATURE - Base64 encoding of the signature
- INSERT_TIME - Time at which the block was inserted into the database
- BLOCK_HASH - Base64 encoding of the hash

The primary keys are the public key and the sequence number.

10.2 Links to code

- Creation of database, inserting blocks (TrustChainDBHelper.java)
- IPv8 (database.py)

The app maintains cryptographic compatibility with [Tribler](#), using [Libsodium](#). To achieve this, [Java JNI bindings](#) has been used to allow the Trustchain Android app to use native libsodium method invocations.

Libsodium has been chosen due to the fact that it is used by Tribler, and because it provides all of the core operations needed to build high-level cryptographic tools. It is a portable, cross-compileable, installable, packageable fork of [NaCL](#) with a compatible extended API. Libsodium supports a variety of compilers and operating systems, including Windows (with MinGW or Visual Studio, x86 and x64), iOS, Android, as well as Javascript and Web Assembly.

Libsodium supports the notion of [Dual Secrets](#), an object that supports both encryption and signing. The key formats used in the app match the keys used in Tribler, using `xsa1sa20poly1305` for identity management and `ed25519` for signing.

11.1 (De)serialization

During transmission and storage, keys are serialized and deserialized in the following manner:

- Public key pair: `LibNaCLPk`: + public key bytes + verify key bytes
- Private key pair: `LibNaCLSk`: + encryption seed + signing seed

The signing key is then generated using the signing seed. The private key can be generated from the encryption seed.

11.2 Links to code

- [The crypto package](#)

Claims, attestation and zero knowledge proofs

The idea of trustchain app is to implement claims, attestation and zero knowledge proofs as in the ipv8 python implementation, see <https://tools.ietf.org/html/draft-pouwelse-trustchain-01#section-4.2> for more details. Claims and attestation have been implemented in the current trustchain app, however zero knowledge proofs have not because of several problems, which will be explained in the following sections.

12.1 Zero knowledge proofs

Zero knowledge proofs can be used to prove a claim without revealing the true value in this claim. A small example: suppose Alice wants to buy liquor at the liquor store, which can only be bought when you are 18 years or older. However, Alice does not want to reveal her age, but still wants to buy the liquor. This is a situation where a zero knowledge proof is extremely useful. A zero knowledge proof allows Alice to show that her age is in a certain range (this is also known as a range proof), which does not reveal her age.

12.2 Implementation

Three different implementations of zero knowledge proofs have been tried or

12.2.1 Bulletproof

#A well known rule of crypto is: don't implement crypto yourself if you are not an expert. Bulletproofs are zero knowledge proofs which require no trusted setup (<https://crypto.stanford.edu/bulletproofs/>). There exists a [Java implementation](#), which has been used for this project. However, there were some difficulties to get it to work on Android. The source code of the Bulletproof library is written in Java 1.9, which is not supported by Android. To overcome this issue several parts of the code have successfully been rewritten to Java 1.8. However, Java 1.8 on Android is only supported by Android 8.0 or higher. There exists some tools/libraries to port Java 1.8 byte and/or source code to Java 1.8 byte and/or source code, for example [retrostreams](#). Unfortunately, these tools/libraries could not port all Java 1.8 features to a lower version of Java. It would be possible to rewrite the Bulletproof library to Java 1.7, but it depends heavily on the [SimpleReact](#) library which uses Java 1.8 as well. This makes the Bulletproof only suitable

for Android devices that run Android Oreo or higher. This is around 5% of all Android devices at the time of writing (<https://www.digitaltrends.com/mobile/android-distribution-news/>) , so it was chosen to not use the Bulletproof library.

Next to the Java implementation of the Bulletproof library, there exists a c++ implementation. With JNI (Java Native Interface) it would be possible to use this implementation in the app. However, since all of us have little to no experience with this. little knowledge of the Bulletproof library, and there was not much documentation it was decided to invest time in other methods.

12.2.2 Ipv8 Android Application

The `ipv8` app provides a REST interface for zero knowledge proofs. If this would be used, it means that users should install an additional app (the `ipv8` app) to run the trustchain app, which is not desirable. Next to this, building the `ipv8` app is not straightforward. The buildscripts use an old gradle experimental format (the plugin `com.android.model.application`). Building the app without modifying the buildscripts results in build errors on the latest version of Android Studio. These errors can be fixed by changing some things in the buildscript and source code, but ultimately did not build a fully functional app. Because of these two reasons it was decided to not use the `ipv8` app.

12.2.3 Other implementations

Since the above options were not viable it was decided to look to other implementations of zero knowledge proofs. However, there are not much zero knowledge proof libraries written in Java. The ones that are written in Java miss proper documentation and are not used by others (to our knowledge). Since not the libraries are not used, the security of these libraries is questioned.

It was decided to leave zero knowledge proofs out of the application for now, until a better library comes around.

Maturity of Code

This project has had quite a few developers contributing to it. This has caused the codebase to become a mess in the past. After a few major refactors the codebase is much more structured now. This has the added advantage of easier identifying bugs and generally being much easier to understand. Because of the issues lined out in *Code coverage* and because writing tests for Android is generally quite time consuming. The code isn't automatically tested as well as would be desired. However, due to pull-based development the quality of the code and the operation of the app is tested by hand quite often by the developers.

So to put a label on it the code (*kinda*) works. It is definitely not production quality code, however it does work quite well and can demonstrate the possibilities on a small scale. The major bottleneck at the moment is the network overlay. While the TrustChain is scalable, the network overlay has some problems with scalability, limiting the scalability of the app. There are also some problems with the UI when the network is under load, as it can't update the network information.

13.1 Code coverage

It is quite hard to get a good idea of the code coverage for Android projects. This is due to the fact that there are two types of tests for Android. The instrumented (Android) tests and regular Unit tests. The instrumented tests are run on devices and emulators and can make use of the Android framework. Therefore they can be used to tests the UI and other parts which require a device. In our case the usage of the LibSodium cryptographic library requires us to run most cryptography related tests as an instrumented test, because the *library only get's loaded when the app is run on an actual device*. This is appears to be a bug/feature in Android. In some cases this can be solved by mocking the crypto related objects, however generally it severely limits the ability to write unit tests.

Unfortunately we haven't been able to get codecov to work with the AndroidTests, due to a combination of build errors and difficulties with getting AndroidTests to properly run on Travis. Therefore the codecov report below only reflects part of the coverage that can be done with unit tests. However, it must be said that adding the coverage of the AndroidTests wouldn't spectacularly increase the coverage.

Fig. 13.1: Coverage Grid, click on the grid ([external link](#)) and hover on a block to see which file it is

Files	lines	hit	partial	missed	coverage
block	218	65	4	149	29.82%
chainExplorer	165	0	0	165	0.00%
crypto	106	0	0	106	0.00%
funds	323	0	0	323	0.00%
inbox	117	17	3	97	14.53%
main	462	5	0	457	1.08%
network	267	0	0	267	0.00%
offline	316	0	0	316	0.00%
passport	979	80	6	893	8.17%
peer	203	55	6	142	27.09%
peersummary	289	0	0	289	0.00%
storage	291	0	0	291	0.00%
util	208	69	4	135	33.17%
message/MessageProto.java	3,637	207	64	3,366	5.69%
Project Totals (70 files)	7,581	498	87	6,996	6.57%

Latest coverage table can be found at [codecov](#)

Installation Instructions

If you want to quickly test the basic version of TrustChain Android follow *Installing TrustChainAndroid APK*, if you want to create a version of TrustChain Android with your own features follow *Setting up the Android Project*.

14.1 Installing TrustChainAndroid APK

The easiest way to install the app is through [Google Play](#). If you want to install the [latest release](#) from github, download the apk attached to the release to your phone and open it to install. The current minimum version of Android needed is 21 (5.0), however not all features will be available to this version. The recommended version is 26 (8.0) or higher. To reset the apps settings (refreshing the public key) and removing the local chain, simply clear all the data of the app in the app settings.

14.2 Setting up the Android Project

Follow the steps below if you want to make alterations to the project and add your own functions. If you are already familiar with developing Android native apps and GitHub, this will be trivial.

- [Download and install](#) Android Studio
- Make yourself familiar with how Android projects are set up, by reading the [Android Getting Started Guide](#)
 - Note that the guide makes use of the Layout Editor, however writing the xml files directly will give you much better control
- Clone the repository to your work station `git clone https://github.com/wkmeijer/CS4160-trustchain-android.git`
- Import the project in Android Studio by `File>Open` and search for the cloned repository
- Start editing

Note that connecting to an emulator will often not work, so for proper testing you will need two phones.

15.1 Contact

For questions about the code please use Github.

15.2 Useful links

- [App source code on GitHub](#)
- [TrustChain source code on IPv8 GitHub](#)
- [Dispersy ReadTheDocs](#)
- [Tribler GitHub](#)
- [Blockchain Lab TU Delft](#)
- [Paper | TrustChain: A Sybil-resistant scalable blockchain](#)