Digital voting pass - Final report

Wilko Meijer Daan Middendorp Jonathan Raes Rico Tubbing

May 2017

Contents

Chapter 1

Introduction

1.1 Problem definition

1.2 Problem analysis

Chapter 2

Initial Prototype

The first sprint after the research phase was used for creating an initial prototype. This sprint had a duration of 1.5 weeks so the next sprints could start on a Monday.

The prototype consists of two parts:

- Creating an app that can interact with a passport
- Altering a blockchain implementation so it works with a passport

The focus of this sprint was to make it clear if our proposed solution would be viable. Getting Multichain to work with the passport was a concern and had to be implemented in this first sprint, otherwise another (less preferred) blockchain implementation had to be chosen to ensure the viability of the project.

2.1 OCR

There are a few options to choose from when it comes to doing OCR on an android device. Two popular approaches that were considered are using the Mobile Vision API [1] provided by Google and using Tesseract [2], a popular OCR library that has a history of being an accurate OCR solution, it has an android solution Tesseract Tools [3]. We chose to implement Tesseract and not Android Vision because first of all, looking at the vision example project, there was little support as issues were not being addressed. Next to that it seemed that the Android Vision library is much less accurate than Tesseract and also has no support for putting contrains on the detected data. People online seem to switch to Tesseract after not reaching their goal with the Vision API. Therefore we decided to go with Tesseract. More specifically, tess-two [4], a fork of Tesseract that is expanded and maintained. Because of our API 21+ requirement, we could go with only supporting the improved "android.hardware.camera2" camera API classes that were introduced by Google in API 21. [5] To make implementation easier and not reinvent the wheel, a sample app made by Google was used to get the code related to using the camera2 API. [6] To get a feel for the Tesseract API, a tutorial was used [**?**].

During development of the OCR feature we found that the speed of detection and the ability to run multiple scanning threads simultaneously is greatly dependent on the kind of phone used. For example the Samsung galaxy S7 that is owned by one member of the team had no difficulty running at least 4 threads and is able to scan a passport or ID card almost instant. However the LG G4 that we used in testing had a lot more difficulty, and started visually lagging when we deploy multiple scanning threads. Of course optimization of the OCR feature is still possible, which could improve detection speed and accuracy some. since OCR is a CPU intensive operation we expect this problem only to diminish in the future.

Surprisingly we also found that the accuracy of the OCR is greatly dependant on the used document. One passport seems to be much easier to detect than another. This may be because of a slightly different font used on the older passports or the particular order of letters in the MRZ zone. In particular characters

like 0, *O* and *D* or 8 and *B* are often wrongly interpreted. It is possible to write code that swaps known wrongly-interpreted characters, but this would mean checking for many combinations as the list of possibly wrongly interpreted characters grows. Also this would increase the chance on getting a result that passes checksums but doesn't open a connection to the passport. In another attempt to increase the accuracy, some image processing was done. Attempts included changing the contrast, filtering colors, and converting to monochrome bitmap. This did not seem to have any significant, if any at all, effect on accuracy. Part of this was expected since Tesseract does image processing by itself using the leptonica library. [7]

In order to increase detection efficiency ad accuracy further we could make specialized trained data for reading MRZ data. Someone already created this data once [8], and we tried to use it, sadly this traineddata seems to cause crashes when performing the scan on some devices.

To ensure that the information we obtained from the OCR scan is correct, we check it with a checksum using several check digits provided in the MRZ data. [9–11]

2.2 Blockchain implementation

2.2.1 Multichain

MultiChain is a blockchain which is based on the Bitcoin core, but the developers of MultiChain have added more features. It features a customizable permission system, where each node can have the following permissions: connect, send, receive, issue, create, mine, activate and admin [12]. The permissions send, receive, issue and mine are the most valuable to the initial prototype. The following list elaborates what a wallet is allowed to do:

- send - The wallet is allowed to send a token
- receive - The wallet is allowed to receive a token
- issue - The wallet is allowed to create a new token
- mine - The wallet is allowed to mine

Sending and receiving are valuable permissions since every voter has a wallet where they must receive their voting right, and claiming this persons voting right by sending a digital token. The mining permissions is valuable, since a third party can perform a monopoly attack (51% attack) and MultiChain relies on nodes that mine in order to verify transactions. Besides this, it is possible to alter the time it takes to mine a block, which is useful since transactions should be verified in at least five seconds (see requirement 3 in section A.1).

Travel document signing

MultiChain uses the secp256k1 elliptic curve to verify and sign transactions and since travel documents use the BrainpoolP320r1 elliptic curve a library that implements the BrainpoolP320r1 elliptic curve is desired. CryptoPP [1] has implemented this curve and has been used this sprint to alter MultiChain to support BrainpoolP320r1.

To be able to verify a transaction using a travel document, MultiChain must use the accept BrainpoolP320r1 elliptic curve signatures. The initial step to implement this, was to find where the private and public keys are stored. After some searching in the code we found that the private and public key are stored in the classes *CKey* and *CPubKey* respectively. These classes have been altered to use BrainpoolP320r1 elliptic curve instead of the secp256k1 elliptic curve. Now we had to find out how the keys were stored in these classes. After some debugging, we find that in both classes the keys were stored as a byte array called *vch*. The next step was to implement CryptoPP keys in the classes CKey and CPubKey and retrieve the byte arrays from these keys and store this in the variable *vch*. After this, the reverse has also been implemented: create a CryptoPP key from a byte array *vch*. This took some time since the keys were loaded and stored

[1] `https://www.cryptopp.com/`

from a few different functions in these classes. These functions have been successfully altered and mining and sending transactions to the blockchain is now possible with the BrainpoolP320r1 elliptic curve.

Since a travel document can only sign eight bytes of data and the hash of a transaction is a SHA256 hash (32 bytes), the hash is split up in four different parts and are signed individually by a travel document. In Figure 2.1 a visualization of the signing by the passport is given, where the 32 byte block is the hash of a transaction.

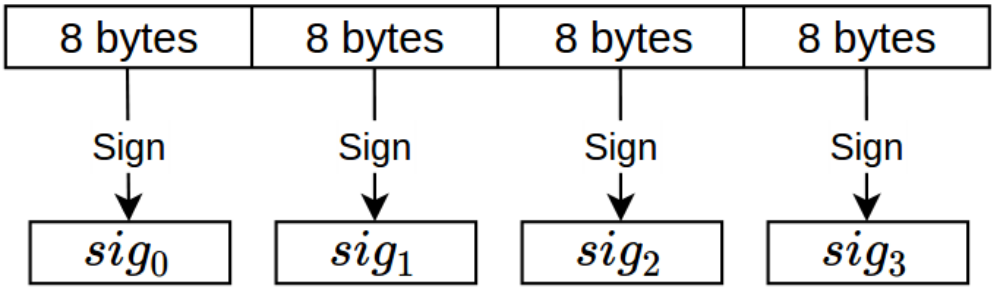


Figure 2.1: Signing a transaction with a travel document

Now the blockchain must verify a transaction signed by a travel document, so the verify function must be altered so it accepts four signatures. This was caused some trouble since MultiChain implemented a maximum signature length of 255 bytes while one signature from a travel document is always 80 bytes long (so $4 * 80 \geq 255$). The 255 byte limited has been altered to support signatures of maximum 320 bytes long. This makes is possible to store the signatures after each other in a byte array, as has been visualized in Figure 2.2. So for example, the signature of the second part of a hash, sig_1, is stored at index 80 till 160 of the byte array containing all signatures. The verify function now verifies each sig_i and if it isn't correct the whole transaction is rejected.

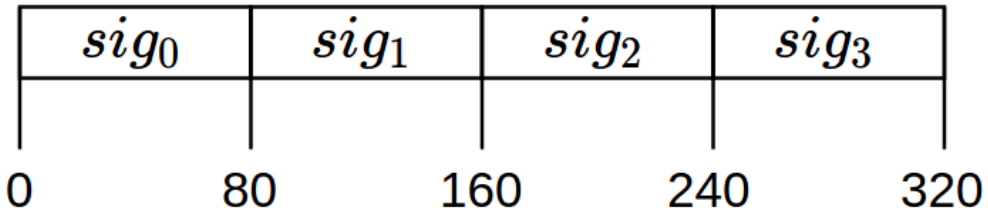


Figure 2.2: Signature storage

The problem now is that mined blocks create one signature of the hash, while the verify expects four signatures. Two options were considered:

- Only verify a transaction with four signatures
- Sign the mined blocks also with four signatures

The first option is chosen, since it is easier to implement. The hash is again split up in four parts, as displayed in Figure 2.1. These signatures are then stored as in Figure 2.2 so the verify function accepts these signatures. The second option is more secure and will be implemented in one of the following sprints.

Public key to blockchain address

To ensure that a machine readable travel document can act as a wallet, the public key of the document needs to be translated to an address on the blockchain. MultiChain uses an address system similar to the ones

used by Bitcoin. The small differences consist of the addition of an identifier generated by the blockchain instance itself. This value is hashed into the final address. Due to this, it is impossible to use an address which is intended for a different blockchain.

The steps leading to a valid address are well documented by in the developer reference. This ensured a smooth integration of this process which lead to a function to translate the public key of a passport to a valid address on the blockchain.

Distributing voting tokens

At every election, there needs to be a new batch of voting tokens sent to the citizens. For this first prototype, a simple python CLI script is built. This script follows the following steps:

1. Load CSV with public keys
2. Convert public keys to valid MultiChain addresses
3. Create new assets corresponding to the amount of addresses
4. Grant send and receive permissions to the generated addresses
5. Distribute the assets to the generated addresses

This script can be found in the *digital-voting-pass-util* repository.

2.2.2 Alternatives

Due to minor struggles during the implementation of the BrainpoolP320r1 Elliptic Curve in MultiChain, other solutions are also considered during this sprint. One of the drawbacks of MultiChain is that it is built on the original Bitcoin core which is written in C++. This language makes it harder to understand the complex structure of a blockchain. The other possibilities that are considered are listed below.

Dragonchain

Dragonchain is a blockchain platform which is actively developed by the Walt Disney company, but does not seem to be entirely finished. It is written in Python which makes the code really accessible. The structure of a transaction relies on an embedded public key in PEM format. This means that the curve parameters are shipped with every transaction. Due to this, it does not matter which Elliptic Curve is used for the signing of a transaction. It literally took 5 minutes to come up with a proof of concept by modifying the unit test to a BrainpoolP320r1 curve.

Unfortunately, Dragonchain is more some sort of blockchain platform, which only has support for raw transactions and basic Python based smart contracts. There does not seem to be any implementation of a wallet or any other way to store any value in the blockchain, just pure signed transactions.

Due to this, and the lack of documentation, this platform doesn't seem to be suitable for the purpose of implementing a digital voting pass at this moment.

Openchain

Another possible implementation is Openchain, which acts more like blockchain as a service. Due to this, there can only be one validation node, as discussed earlier. The code is written in C# which makes it quite accessible. The only drawback of C# is that it is Microsoft-minded and you need all the Dotnet stuff to get it working.

The implementation makes use of the Bouncycastle library for signing and validation of transaction signatures. Due to this, it was relatively easy to transform the codebase to BrainpoolP320r1. This transformation consists out of changing the curve properties which are sent to Bouncycastle.

Different experiments turned out that is is fairly easy to issue a new assets as an admin and distribute it to an address. It was also quite simple to transform the public key of a passport to a valid Openchain address.

Further investigation of this solution was stopped because significant process was booked with the first blockchain solution (Multichain).

2.3 Alternative signing

2.3.1 Block ciphers modes of operation

As discussed in the research report, there is an issue with the capabilities of travel document standard. Due to this, it is not possible to sign a message larger than 8 bytes. This makes is more difficult to sign a transaction which is hashed with a SHA256 hash, which has a length of 32 bytes. This problem is also relevant in other fields of cryptography, for example the 3DES encryption is only able to encrypt blocks of 8 bytes. To tackle this problem there are mainly two methods of operation to avoid this issue.

The first one is ECB (Electronic Code Book), named after the analog lookup books for encrypting and decrypting texts [13]. With this method, every single block of 8 bytes is encrypted and decrypted separately. Because 3DES is deterministic, which means that encrypting the same data with the same key results in the same cyphertext, block which contain the same data will also have the same cyphertext. Due to this, it becomes easy to find patterns in the encrypted data. It makes it also possible by an attacker to combine different blocks and create a complete new signed/encrypted message using blocks from other encrypted messages. This is not that big of a problem, because the only thing that will be signed in this prototype is an SHA256 hash, which makes it difficult to combine different parts.

The second one is a method so called CBC (Cipher Block Chaining). Just like the blockchains in the rest of the prototype, this operation method consists of using previous blocks to sign or encrypt the upcoming blocks. This makes it more difficult to see patterns in the data. This seems to be a really smart idea, because it isn't even possible to determine the signature of a block if the signature of the previous block is unknown. Unfortunately, this methods uses the previous signature to XOR the following block. The signatures created by a Dutch travel document are signed using the ECDSA standard. Due to the possible leak of the private key, these signature cannot be deterministic. This means that signing a block with the same travel document multiple times, results in a different signature every time. So encrypting an decrypting the next block using XOR will not work.

2.4 Coupling android app and passport

For both the Voting Station App (VSA) as the Voter Helper App (VHA) a connection with the passport needs to be made. This connection needs to be able to do two things:

- Read the Active Authentication (AA) public key (located in Data-group 15 [14])
- Sign a blockchain transaction using the 'private key' located in the passport

The AA public key serves as a identifier for the account of the voter on the blockchain. For every interaction (e.g. sending tokens, retrieving current balance) with the account of a voter, this public key is needed. To ensure that a transaction is performed with the authorization of the voter, the transaction needs to be signed with the voters' passport.

The signing is done by making use of the AA protocol of the passport. This protocol serves as a verification of the uniqueness of the passport. It accepts an 8-byte input and signs this with the private key of the passport located in a non-readable part of the chip. The passport returns a byte array. It can be verified that this byte array was signed with the passport belonging to a certain public key, confirming the authorization of the voter.

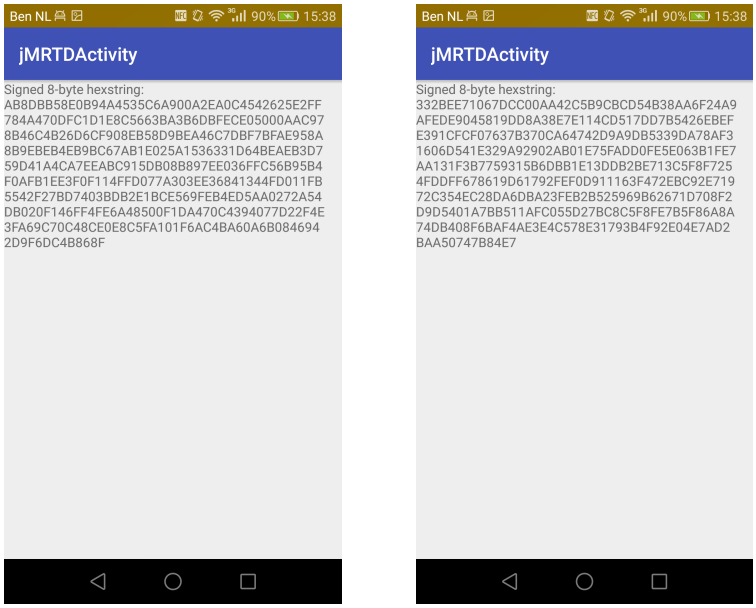
For creating the connection with the passport the JMRTD library[2] is used. JMRTD is an open-source java library that implements the MRTD (Machine Readable Travel Documents) standards as defined by the International Civil Aviation Organization (ICAO). The use of this library makes the connection in an Android app much easier since there is no need for a custom built interpretation layer between two programming languages.

In the Google Play Store a few apps can be found that make a connection with the passport and retrieve information from it. These apps give a clear example of a working concept. A well-known app is ReadId[3] which also makes use of the JMRTD library, this app unfortunatily is not open source. As a starting point the epassportreader app by github user Glamdring[4] was used, which has a very basic implementation of the JMRTD library.

The activity that reads the data from the passport waits until a NFC chip is detected, if this NFC chip belongs to a passport it tries to make a connection. For this connection three bits of information are needed:

- Document number
- Date of birth
- Expiry date of document

This is needed for the Basic Access Control (BAC), which ensures a person trying to read the passport also has physical connection to it. This information can be read from the Machine Readable Zone on the passport by using OCR as explained in 2.1 or can be filled in manually when OCR fails or if OCR is not preferred by the user. Once the BAC succeeds, a connection is established and the contents of the chip are unlocked and can be read.



(a) Output 1 (b) Output 2

Figure 2.3: Results of signing the hex string '0a1b3c4d5e6faabb' with a passport

Depending on what functionality is needed one of two functions can be called:

- *getAAPublicKey* - which returns the public key located in datagroup 15
- *signData* - which accepts an 8-byte array as input and returns a byte array signed by the passport

In Figure 2.3 the resulting byte array represented as a hex string can be seen, since the signing algorithm of the passport is non-deterministic, the same input will yield different outputs every time.

These two functions are essential for signing transactions and getting information from the blockchain associated with the voter. The implementation of this functionality is described in chapter 3.

[2]http://jmrtd.org/
[3]https://www.readid.com/
[4]https://github.com/Glamdring/epassport-reader

Chapter 3

Clickable + Technical Demo

The goal of sprint 2 is a clickable demo of the app's UX and a technical demo consisting of the connection between the blockchain and the app. During this sprint testing was set-up for the various parts of the application. This chapter gives an overview of the work that was done and the problems that were encountered.

3.1 User experience design

As stated in the project plan, the client offered a budget to use an external party to create a UX design for the application. At the end of sprint 1, a professional UX designer was hired at Milvum. To make clear the purpose of the first application, a wireframe on was sketched on paper. During the second sprint, there were several meetings set up with this professional.

The first prototype was based on a sketch drawn with pencil and paper, to make clear which screens were needed. This can be seen in figure B.1, this emerged to a basic colorized workflow.

When discussing this design, there were some things that we thought were still sub-optimal and resulted in new insights. There was for example no way to select the current elections. Beside these minor changes, there were also some small detailed changes we came up with. E.g. the amount of ballot-papers that should be handles was not very obvious, the process of scanning the MRZ isn't really a process which should be visible in a process bar. The design was also not fully compliant to the Angular Material Design styleguide as we wanted to see. The styleguide states that the app bar should be used for branding, navigation, search, and actions, not for notifications.

The feedback above resulted in a second iteration where these issues are improved. This result is visible in figure B.2.

Besides design choices made by the designer, we made some of our own. One of these is about the status bar, we decided to make it transparent according to Material design guidelines. We added a top-padding to the Action bar and enabled the transparent status bar. This way the color of the Action bar will shine trough the status bar giving it the same kind of color but darker. In the camera view the transparent status bar causes the camera feed to shine through he status bar, which is a nice effect in our opinion.

As part of the clickable demo, the design was implemented. During the implementation Android Material Design [15] was used as reference to get a uniform design which would feel familar and logical for users.

3.2 App Permissions

The adnroid app needs several permissions in order to perform its basic functions. These include:

- Camera permissions to scan the MRZ zone.
- NFC permissions to communicate with the document.

- Internet permissions to communicate with the blockchain. Storage permissions to store and read both the blockchain data and trained data for the OCR scanning library.

Internet and NFC permission are not considered dangerous permissions by Android **??**, so these permissions are granted when installing the app and we can use them with no problem. The others, storage access and camera access, are considered dangerous permissions. This means that starting from API level 23 these permissions needs to be granted at runtime when the app requires them. For the camera permission, this occurs when the CameraActivity is opened after the user clicks the button to scan a document. The need for camera permissions in this case if quite clear to the user. For storage permissions the situation is different however, these permissions are needed right when the app is launched in order to store the downloaded blockchain data. The need for this permission is not very clear to the user and may be confusing when the app asks to write to the storage right after launch. For this reason we included an ErrorDialog in our app. When the app is denied one of the dangerous permissions, it will display a dialog explaining why the permission is needed, after which the activity is closed.

3.3 Connecting to the blockchain

The app which is used at the voting station needs to check the balance amount of votes spendable by the owner of the passport. In order to know this balance, something needs to look inside the block chain and check unspent outputs. Clearly we need a connection to the blockchain in some fashion. There are three ways of connecting to the blockchain. In this section, the pros and cons of these choices will be taken into consideration.

3.3.1 JSON RPC

This is the easiest solution, because the app doesn't need to know anything about the blockchain. Downloading all the blocks will not be necessary and all the logic happens at the server. Only the signing of the transaction occurs in the app. Due to this, the system relies on one working webservice, which is a single point of failure.

The standard RPC service which is shipped with Multichain is also not designed to act in this way. For example, it relies on its own wallet which is attached to the node. So, before we can use this RPC service, we need to fork it and implement some authentication rules in it, or place some kind of proxy service between the app and the node.

3.3.2 Light wallet

In this case, the wallet functionality is embedded into the app. So the app is able to calculate the balances and is directly connected to the nodes in the network. The download of the entire blockchain can be a really time and data consuming process. To fix this, light wallets rely on only the headers of every block with only the transactions related to the wallet included.

There are two problems with this approach: Because it is desirable to have near instant verifications, the parameters are set to mine really small blocks in short intervals. Downloading only the headers will still be inefficient. The main Bitcoin network is set to mine a block every 10 minutes on average. With the parameters used in blockchain it is 15 seconds.

Experiments turned out that downloading headers with a size of 80 bytes runs at 1 kilobyte per second. The amount of headers produced in one day with a rate of 4 blocks per minute is $4 * 60 * 24 * 80 = 460800$ bytes, or 461 kilobytes. So downloading one entire day of headers will take around 7 minutes, which is not acceptable.

Another problem with this approach is that the headers are not enough to calculate the balance of voting tokens of every citizen. So, either the blockchain needs to be synchronized every time a passport is scanned, or the entire blockchain still needs to be downloaded.

3.3.3 Full wallet

Tackles all of the problems above, but it will be slow and will also be slower as the blockchain grows. This will also not be suitable for running on mobile devices.

3.3.4 Hybrid

To have the best of both worlds (light wallet and JSON RPC), the following approach might work. If the app doesn't have a local blockchain at all, but is connected to a node. Then it is still able to broadcast a transaction to the network. A rejection of a transaction it also broadcasted back.

Why not broadcast 3 transactions and see how many of them are rejected? This could be supported by a simple web service which returns the amount of voting tokens available in a wallet before the actual transaction happens.

3.3.5 Choice

The last option seems to be the most obvious solution. Unfortunately, during the implementation, it turned out that this is not possible due to the way transactions are built.

3.4 Refactor MultiChain

CreateBlockSignature from miner.cpp calls sign from class *CKey*
TransactionSignatureChecker::VerifySignature calls verify

3.5 Permission analysis

MultiChain has implemented a permissions system, which enables permissions on address level. These permissions are stored on the blockchain and changes to them are done through transactions. The following permissions [12] are currently available:

- *connect* – to connect to other nodes and see the blockchain's contents.
- *send* – to send funds, i.e. sign inputs of transactions.
- *receive* – to receive funds, i.e. appear in the outputs of transactions.
- *issue* – to issue assets, i.e. sign inputs of transactions which create new native assets.
- *create* – to create streams, i.e. sign inputs of transactions which create new streams.
- *mine* – to mine blocks, i.e. to sign the metadata of coinbase transactions.
- *activate* – to change connect, send and receive permissions for other users, i.e. sign transactions which change those permissions.
- *admin* – to change all permissions for other users, including issue, mine, activate and admin.

As will be explained in section 3.6, anyone is able to connect to the network. This poses a minimal risk to the network since nodes can only see what is happening in the blockchain. It is thus not possible to alter the blockchain with this right.

If a node has the *send* permission, it is possible to send funds to another node. When this node is a corrupt node, it can flood the network with transactions. To prevent this, only wallet addresses of travel documents of voters will receive this right. Since the private keys of these travel documents are not accessible, there is now way to send more than three transactions (maximum of votes through proxy voting) and thus

flood the network. The *receive* permissions poses minimal risk since receiving requires a node to send some funds.

Creating an asset or stream is considered as a transaction in MultiChain [16]. This means that rogue nodes can flood the network by, for example, creating many. assets. To prevent this only government nodes should have permissions to *issue* and *create*. However, the government nodes can be hacked and hackers can flood the network. This can be prevented by revoking these rights of these nodes after the vote tokens are distributed. A disadvantage of this solutions is that it is not possible to reuse the blockchain for a future election.

The mining right allows a node to generate a block. If there is monopoly of rogue nodes with mining permissions it is possible that fraud is committed. Without the mining right no blocks will be generated and no transactions will be verified. There need to be thus at least one mining node and this mining node should be owned by the government. Again, this mining node can be hacked, which is why it is desired to have several mining nodes owned by the government. Now the non-hacked nodes can reject a mined block from a rogue node.

With admin right it is possible to change permissions, either revoking or granting, of users. Revoking is useful when a person dies just before an election and this person has an account on the blockchain with a token. The government is now able to revoke the permissions of this person, so no other person can 'steal' the token by proxy voting.

A disadvantage of an admin account is that this account can be hacked, and can be used to tamper with the elections by granting himself mining rights for example. This can be prevented by revoking the permissions of the admin (the admin can do this himself). The problem with this solution is that revoking the permissions of other nodes is not possible any more, so a solution to revoke proxy votes of dead persons should be thought of.

3.6 Blockchain parameters

With MultiChain it is possible to alter parameters of the chain. In this section the most important parameters, values of these parameters and a justification of these values will be discussed. A complete list of the parameters, values and explanations can be found in Appendix C.

3.6.1 Permissions

The two parameters that will be discussed in this section are *anyone-can-connect* and *anyone-can-mine*, which are permission parameters. These parameters control if any node can connect to the blockchain and if any node can mine a block. Parameter *anyone-can-connect* is set to *true*, so any person can see what is going on in the blockchain and increase the trust of a person. The second parameters, *anyone-can-mine*, is set to false. This is done so only government nodes can mine a block, which ensures a monopoly attack is not possible.

3.6.2 Blocks

A transaction is accepted when it is contained in a block, so in order to verify an transaction swiftly a block should also be generated swiftly. MultiChain has two parameters which can influence the block time: *target-block-time* and *pow-minimum-bits*. The parameter *target-block-time* which is the target average time in seconds between two consecutive blocks. The other parameter, *pow-minimum-bits*, is the proof-of-work difficulty which must contain a value between one and 32. One is the most easy and 32 the most difficult.

If we assume that there are 13 million persons who cast their vote on a single day, where it is possible to cast a vote from 6am to 9pm, we calculated that there are approximately 240 transactions per second ($13000000/15/60/60 \approx 240$). This is the average and since we want to handle more than the average, we assume 480 transactions per second. A transaction is about 200 bytes and we have blocks of one megabyte,

there fit about 5000 transactions in a block. Now it is possible to calculate the target block time: $5000/480 \approx 10.4$. The *target-block-time* is set to 10 since only natural numbers are allowed.

To make sure that a miner is indeed able to mine a block in 10 seconds, the parameters *pow-minimum-bits* must be low enough. After some experimentation we concluded that a proof-of-work difficulty of four is suited to mine blocks in ten seconds.

3.7 Testing MultiChain

MultiChain test framework has been removed, since Bitcoin has a test framework - private add link Is hard, since all files need to be compiled and MultiChain removed all tests from the repo. Compiling with pre-compiled libraries does not work since, it gives us a mc_gstate unkown exception Works if you pass a flag UNIT_TESTS_MULTICHAIN to the compiler

3.8 Testing NFC related methods (passportconnection)

The passportconnection related methods all work with the Passportservice from JMRTD. This Passportservice handles all requests and responses to and from the passport and makes use of an Inputstream. Because in Unittests the passport (hardware) can't be used, this Passportservice is mocked, just as the Inputstream.

The Inputstream is given data that represents a ECDSA public key, which should get handled similarly to data from the passport. However, this raises an exception in the method that reads the public key from the data. Even when the data from datagroup 15 from an actual passport is used as input, it still raises an exception.

Because a lot of time was spent on trying to get these tests to work without any result, it was decided to not test these methods by using Unit tests. Instead AndroidTests or a testing plan could be set up to manually test the functionality of the passportconnection before each release. Even though this means Travis CI can't be used for continuous integration tests, this proposed solution is a way to ensure no functionality is broken by new releases.

AndroidTests (instrumented tests) are used to test on dedicated hardware. The problem with this project is that several user actions are required for testing (scanning the OCR and hold passport to the phone). This makes it very hard to test for specific cases. Only the case of a successful passport connection can be tested, but this is the same as testing the functionality of the app. It was decided that creating separate tests for this is not worth the time. Testing for how the app handles the failed cases, e.g. passport is held to the phone too short, is possible by using regular Unit Tests. Since the tests will handle the null cases, there is no need to mock any Inputstreams, which makes testing easier. Testing for the handling of the NFC tag discovery was not successful, because Mocking NFC tag intents is not possible, methods to do so are not available in newer android versions. [**?**]

For each release to master a testing plan should be walked through in order to ensure the functionality that is not tested by unit or integration tests is still working. The test plan for the passport connection can be found in **??**. Using a test plan instead of unit tests or integration tests has serious limitations. The amount of test cases is very limited and it is much harder to test for rare bugs. Furthermore it is hard to maintain a consistent test environment, which would make it harder to debug. Although this set-up is far from ideal, it is sufficient to ensure a working prototype for releases.

3.9 Testing and improving OCR

In order to test the Tesseract OCR UnitTests were written. Because tesseract uses a number of dependencies that cannot be mocked, AndroidTests were used instead of regular tests. AndroidTests allow for executing tests in an emulated environment or on actual hardware.

Using AndroidTest and images of traveling documents it was possible to verify the accuracy of the OCR scanning. During the use of the application the OCR scanner scans a series of pictures until it recognizes

a correct MRZ (verified by checksums). Because of this it is not needed that the OCR scanner has 100% accuracy for each picture. Since the images used for testing are not perfectly focused, the OCR scanner won't have 100% accuracy. To account for this a threshold was set for accuracy in the tests. This accuracy was calculated using the Levenshtein distance. The resulting MRZ code from the OCR scanning of the test images should at least have a reasonable accuracy, like 90%. By setting the accuracy threshold to just below the lowest performing test, any drop in performance of the OCR scanner can be detected by continuous integration, since test will start failing.

These tests are also essential in order to test for improvements of the OCR. Using the accuracy improvements to the OCR scanner can easily be detected. After creating these tests several traineddata files were tested to see if any caused an improvement in scanning the MRZ. Eventually a traineddata file was created based on the OCR-B ttf file. This resulted in a 5 percent increase in accuracy. In the real world, however, it resulted in much faster recognition of the MRZ. On passports where reading the MRZ used to be impossible, now the scanner recognized the MRZ immediately after the camera had a good focus. This improvement is very important for the ease-of-use requirement of the system.

Chapter 4

Full prototype

Chapter 5

Sprint 4

Chapter 6

Conclusions

Bibliography

[1] Android. Mobile vision. [Online]. Available: https://developers.google.com/vision/

[2] Tesseract. Tesseract ocr. [Online]. Available: https://github.com/tesseract-ocr/tesseract

[3] ——. Tesseract android tools. [Online]. Available: https://code.google.com/archive/p/tesseract-android-tools/

[4] Tess two. [Online]. Available: https://github.com/rmtheis/tess-two

[5] Android. Android developer docs. [Online]. Available: https://developer.android.com/reference/android/hardware/camera2/package-summary.html

[6] Google. Camera2 google samples. [Online]. Available: https://github.com/googlesamples/android-Camera2Basic

[7] C. Colglazier. Improving the quality of the output. Accessed: 2017-05-19. [Online]. Available: https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality

[8] Tesseract addons. [Online]. Available: https://github.com/tesseract-ocr/tesseract/wiki/AddOns#community-training-projects

[9] Wikipedia. Machine-readable passport. [Online]. Available: https://en.wikipedia.org/wiki/Machine-readable_passport#Nationality_codes_and_checksum_calculation

[10] A. de Smet. Machine readable passport zone. [Online]. Available: http://www.highprogrammer.com/alan/numbers/mrp.html

[11] ICAO. Doc 9303: Machine readable travel documents. [Online]. Available: https://www.icao.int/publications/Documents/9303_p3_cons_en.pdf

[12] MultiChain. Multichain permissions managment. Accessed: 2017-05-22. [Online]. Available: http://www.multichain.com/developers/permissions-management/

[13] M. Dworkin, "Recommendation for block cipher modes of operation. methods and techniques," DTIC Document, Tech. Rep., 2001.

[14] "Machine readable travel documents part 10," International Civil Aviation Organization, Montréal, Quebec, Canada H3C 5H7, Tech. Rep. 9303, 2015.

[15] Google Inc. Material design for android. [Online]. Available: https://developer.android.com/design/material/index.html

[16] G. Greenspan. Multichain private blockchain. Accessed: 2017-05-26. [Online]. Available: http://www.multichain.com/download/MultiChain-White-Paper.pdf

Appendix A

Requirements

A.1 Blockchain

A.1.1 Must Have

1. The blockchain must verify transactions with the ECDSA curve BrainpoolP320r1.
2. A voter must be able to claim his vote right only one time and no more, either with a digital voting pass or physical voting pass.
3. The government must be in control of the blockchain.
   (a) The government must be able to transfer votes to wallets of eligible persons
   (b) The government must be able to control which nodes are able mine
4. The blockchain must accept a transaction when the wallet has sufficient funds and the transaction was signed by a passport
5. The blockchain technology must not have a single point of failure

A.1.2 Should Have

1. A voter should be able to transfer a vote to a different voter who is eligible to vote (proxy-vote)
2. The government should be able to freeze accounts
3. A transaction should be contained in a block (confirmed) within twenty seconds

A.1.3 Could Have

1. A voter could be able to use all types of identification documents without specifying which one this person is voting with

A.2 Voting station app

A.2.1 Must Have

1. There must be an option to manually input the data from the machine readable zone.
2. Reading public key from passport using NFC

3. The app must support of transaction with ePassport using NFC
4. The app must show wether the signing was succesful and if the transaction was acepted by the network.
5. The app must send a signed transaction to the blockchain.

A.2.2 Should Have

1. The app should be able to show a list of transactions made with the scanned passport.
2. The app should be able to use OCR to read the ePassport's MRZ zone

A.2.3 Could Have

1. The app could have manual control for flash and focus in OCR scanning mode.

A.3 Voter app

A.3.1 Must Have

1. See current balance of voting tokens
2. Transfer tokens to another voter (proxy-voting)
3. See transaction history

A.3.2 Should Have

1. Reclaim voting token from proxy-voting

A.3.3 Could Have

1. Verify if traveling document chip is working correctly
2. Explanation of how the digital voting process works
3. Information about voter turnout
4. Information about nearest voting stations

A.3.4 Won't Have

1. Registration for digital voting
2. Local storage of voter information

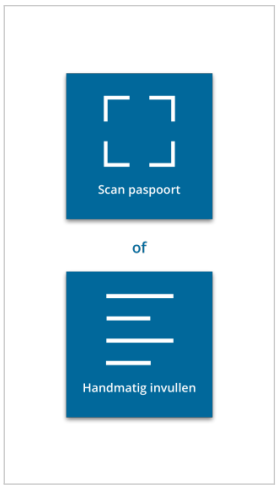
Appendix B

UX

B.1 Polling station app - iteration I

Note: The following user experience design emerged from a collaboration with Angelo Croes (angelo@milvum.com), UX designer at Milvum.

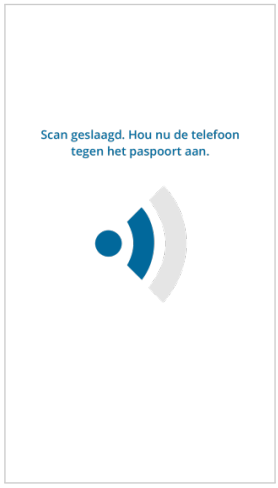
(a) Splash screen



(b) Main menu

(c) OCR I

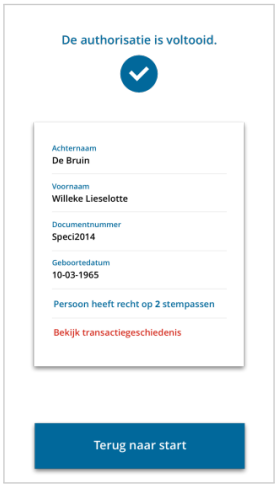
(d) OCR II



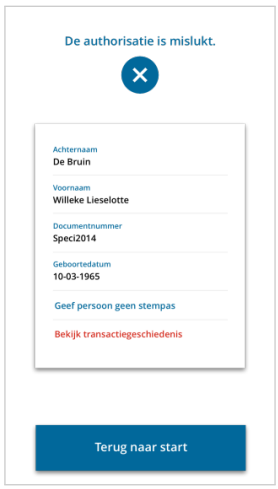
(e) NFC

(f) OCR error

(g) Manual input



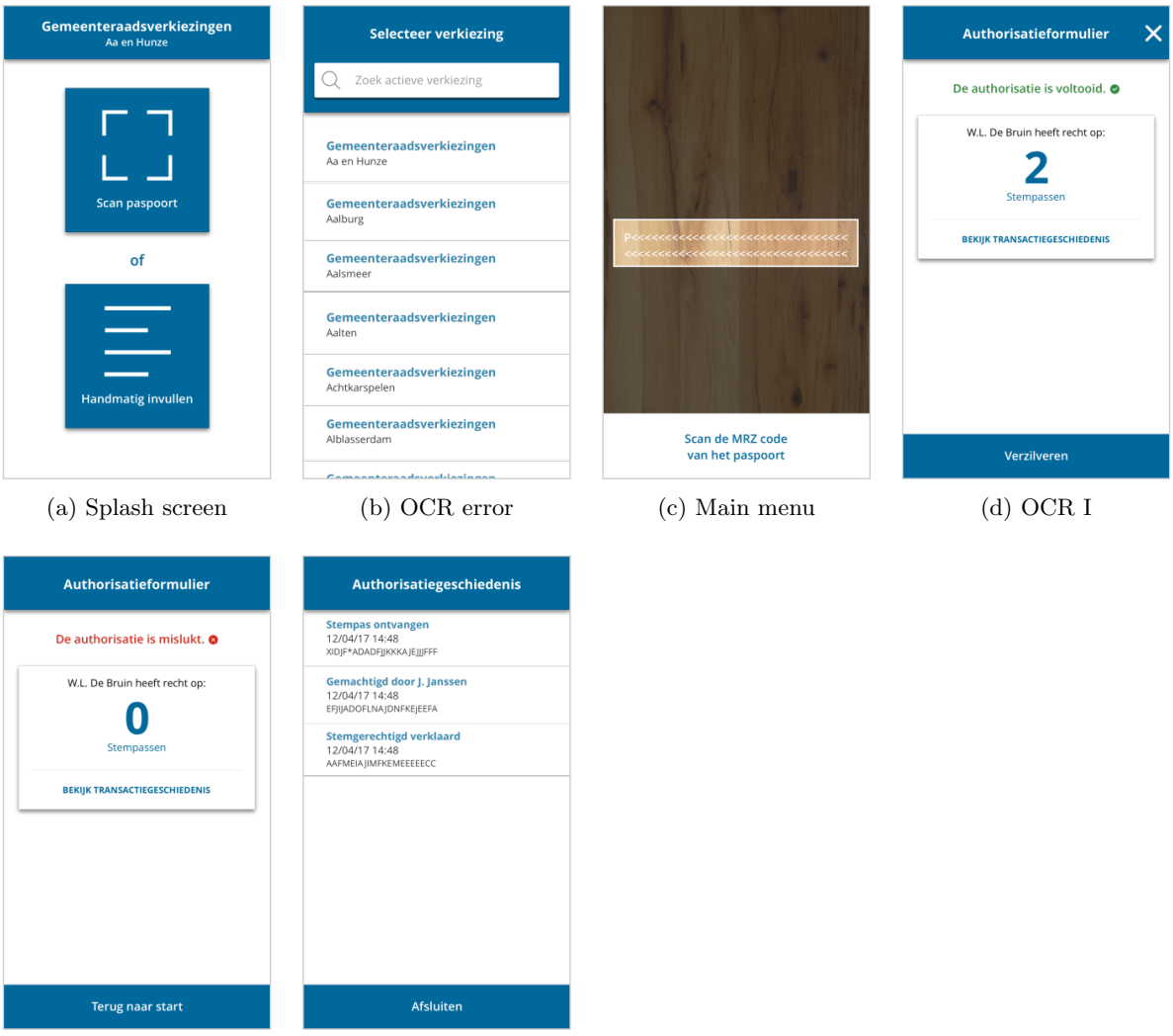
(h) Authorized



(i) Not authorized

(j) Transaction history

Figure B.1: Iteration 1



(a) Splash screen (b) OCR error (c) Main menu (d) OCR I

(e) OCR II (f) NFC

Figure B.2: Iteration 2

Appendix C

Parameters

Parameter	Description	Value	Explanation
chain-protocol	Use MultiChain for a MultiChain blockchain or bitcoin for a bitcoin-style blockchain with no permissions, native assets or streams.	MultiChain	Necessary to have permissions in the blockchain
chain-description	Textual description of the blockchain for display to users.	Elections X	
root-stream-name	Name of the root stream for general data storage (leave blank for none).	root	Default
root-stream-open	Allow anyone with send permissions to write to the root stream.	true	Default
chain-is-testnet	Whether to set testnet to true in the output of various JSON-RPC API calls. This is for compatibility with Bitcoin Core and does not affect any other testnet-like behavior.	false	The blockchain should not be a test net
target-block-time	Target average number of seconds between blocks, i.e. delay for confirming transactions. If this is below 10 seconds, it is recommended to set mining-turnover low, to minimize the number of forks.	10	See subsection 3.6.2
maximum-block-size	Maximum number of bytes in each block, to prevent network flooding by a rogue miner.	1MB	Default, should be able to hold 5k transactions, see final report

Table C.1: Basis chain parameters

Parameter	Description	Value	Explanation
anyone-can-connect	Apply no restriction to connecting to the network, i.e. nodes do not require connect permissions.	true	For transparency reasons
anyone-can-send	Apply no restriction to sending transactions, i.e. signing transaction inputs.	false	Government can freeze wallets if person in question loses his right to vote, (e.g in case of incarceration or death)
anyone-can-receive	Apply no restriction to receiving transactions, i.e. appearing in transaction outputs.	true	Ability to proxy-vote
anyone-can-receive-empty	Apply no restriction to addresses which appear in transaction outputs containing no native currency, assets or other metadata. Only relevant if anyone-can-receive=false. This allows addresses without receive permission to include a change output in non-asset transactions, e.g. to publish to streams.	false	Default
anyone-can-create	Apply no restriction to creating new streams.	false	No streams, except the root stream, are used so unnecessary
anyone-can-issue	Apply no restriction to issuing (creating) new native assets.	false	Issuing means creating new election which is only reserved for the government
anyone-can-mine	Apply no restriction to mining blocks for the chain, i.e. confirming transactions.	false	Mining reserved for government nodes to protect against monopoly attack
anyone-can-activate	Apply no restriction to changing connect, send and receive permissions of other users.	false	Only admin is allowed to do this
anyone-can-admin	Apply no restriction to changing all permissions of other users.	false	Obvious
support-miner-precheck	Support advanced miner permission checks by caching the inputs spent by an administrator when setting admin or mine permissions – see permissions management for more information.	true	Default
allow-p2sh-outputs	Allow pay to scripthash outputs, where the redeem script is only revealed when an output is spent. See permissions management for more information about permissions and P2SH addresses.	false	p2sh is not needed to create and send/verify transactions. This is thus an unnecessary risk (when turned on), which can be avoided.
allow-multisig-outputs	Allow multisignature outputs, where more than one address is explicitly listed in a transaction output, and a given number of these addresses are required to sign in order to spend that output. See permissions management for more information about permissions and multisig outputs.	false	Isn't needed

Table C.2: Global permissions parameters

Parameter	Description	Value	Explanation
setup-first-blocks	Length of initial setup phase in blocks. During the setup phase, the constraints specified by the other parameters in this section are not applied.	60	This does not matter since the government will send about 13 million transactions, which will create a lot of blocks
mining-diversity	Minimum proportion of permitted miners required to participate in round-robin mining to render a valid blockchain, between 0.0 (no constraint) and 1.0 (every permitter miner must participate). Unlike mining-turnover, this is a hard rule which determines whether a blockchain is valid or not.	0.75	Default
admin-consensus-admin	Proportion of permitted administrators who must agree to modify the admin privileges for an address, between 0 (no consensus required) and 1 (every admin must agree).	1	All admins for safety
admin-consensus-activate	Proportion of permitted administrators who must agree to modify the activate privileges for an address, between 0 and 1.		Value
admin-consensus-mine	Proportion of permitted administrators who must agree to modify mining privileges for an address, between 0 and 1.	1	All admins agree for safety reasons
admin-consensus-create	Proportion of permitted administrators who must agree to modify stream creation privileges for an address, between 0 and 1.	1	There is only stream, and no more. So default one
admin-consensus-issue	Proportion of permitted administrators who must agree to modify asset issuing privileges for an address, between 0 and 1.		Value

Table C.3: Consensus parameters

Parameter	Description	Value	Explanation
lock-admin-mine-rounds	Ignore forks that reverse changes in admin or mine permissions after this many (integer) mining rounds have passed. A mining round is defined as mining-diversity multiplied by the number of permitted miners, rounded up. This prevents changes in the blockchain's governance model from being reversed and can be overridden by each node using the lockadminminerounds runtime parameter.	10	Default value
mining-requires-peers	A node will only mine if it is connected to at least one other node. This is ignored during the setup phase or if only one address has mine permissions, and can be overridden by each node using the miningrequirespeers runtime parameter.	true	An isolated node mining is useless to the network
mine-empty-rounds	If there are no new transactions, stop mining after this many rounds of empty blocks. A mining round is defined as mining-diversity multiplied by the number of permitted miners, rounded up. This reduces disk usage in blockchains with periods of low activity. If negative, continue mining indefinitely. This is ignored during the setup phase or if target-adjust-freq>0, and can be overridden by each node using the mineemptyrounds runtime parameter.	5	No mining needed when there is no election in progress or no passes being cashed,
mining-turnover	A value of 0.0 prefers pure round robin mining between an automatically-discovered subset of the permitted miners, with others stepping in only if a miner fails. In this case the number of active miners will be mining-diversity multiplied by the number of permitted miners, rounded up. A value of 1.0 prefers pure random mining between all permitted miners. Intermediate values set the balance between these two behaviors. Lower values reduce the number of forks, making the blockchain more efficient, but increase the level of mining concentration. Unlike mining-diversity, this is a recommendation rather than a consensus rule, and can be overridden by each node using the mining-turnover runtime parameter.	Value	Must be low to reduce forks

Table C.4: Mining runtime parameters

Parameter	Description	Value	Explanation
skip-pow-check	Skip checking whether block hashes demonstrate proof of work.	false	Obvious
pow-minimum-bits	Initial and minimum proof of work difficulty, in leading zero bits. (1 - 32)	4	Should be low so it easy to mine a block and thus conform transactions
target-adjust-freq	Interval between proof of work difficulty adjustments, in seconds, if negative - never adjusted. (-1 - 4294967295)	-1	Difficulty should not increase, to ensure blocks are still mined fast enough
allow-min-difficulty-blocks	Allow lower difficulty blocks if none after 2*.	false	Difficulty is already low, so this is not necessary

Table C.5: Advanced mining parameters

Parameter	Description	Value	Explanation
only-accept-std-txs	Only accept and relay transactions which qualify as 'standard'.	true	Default
max-std-tx-size	Maximum size of standard transactions, in bytes. (1024 - 100000000)	Value	
max-std-op-returns-count	Maximum number of OP_RETURN metadata outputs in standard transactions. (0 - 1024)	10	Default
max-std-op-return-size	Maximum size of an OP_RETURN metadata output in a standard transaction, in bytes.	2097152	Default
max-std-op-drops-count	Maximum number of inline OP_DROP metadata elements in a single output in standard transactions.	5	Default
max-std-element-size	Maximum size of data elements in standard transactions, in bytes.	8192	Default

Table C.6: Standard transaction definitions

Appendix D

Testplan

D.1 Passport connection

For each release to the master branch the following steps must be taken to ensure the connection with the passport still works as intended.

Do all of the below for a Dutch passport and a Dutch ID card, both issued after March 9th 2014. The whole test suite should be done on two different devices in order to uncover device specific issues.

Successful connection

1. Start up the app
2. Wait for the app to be initialized
3. Choose an election
4. Press manual input button
5. Put in the document details of the travel document and submit
6. Start connection
7. Hold travel document to back of phone
8. Wait until the passport connection is completed
9. **The authorization screen should be showed**

D.1.1 Bad BAC Key

1. Start up the app
2. Wait for the app to be initialized
3. Choose an election
4. Press manual input button
5. Put in wrong document details and submit
6. Start connection
7. Hold travel document to back of phone

8. **The passport connection should fail and display an error indicating the document details are wrong**

D.1.2 Bad NFC connection

1. Start up the app
2. Wait for the app to be initialized
3. Choose an election
4. Press manual input button
5. Put in the document details of the travel document and submit
6. Start connection
7. Hold travel document to back of phone for a very short time failing the passport connection
8. **An error indicating to retry holding the travel document to the phone should be displayed.**