Digital voting pass - Final report

Wilko Meijer Daan Middendorp Jonathan Raes Rico Tubbing

May 2017

Contents

1 Initial Prototype **2**
1.1 OCR . . . 2
1.2 Blockchain implementation . . . 3
1.2.1 Multichain . . . 3
1.2.2 Alternatives . . . 5
1.3 Alternative signing . . . 6
1.3.1 Block ciphers modes of operation . . . 6
1.4 Coupling android app and passport . . . 6

A Requirements **9**
A.1 Blockchain . . . 9
A.1.1 Must Have . . . 9
A.1.2 Should Have . . . 9
A.1.3 Could Have . . . 9
A.2 Voting station app . . . 9
A.2.1 Must Have . . . 9
A.2.2 Should Have . . . 10
A.2.3 Could Have . . . 10
A.3 Voter app . . . 10
A.3.1 Must Have . . . 10
A.3.2 Should Have . . . 10
A.3.3 Could Have . . . 10
A.3.4 Won't Have . . . 10

Chapter 1

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.

1.1 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]

1.2 Blockchain implementation

1.2.1 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 1.1 a visualization of the signing by the passport is given, where the 32 byte block is the hash of a transaction.

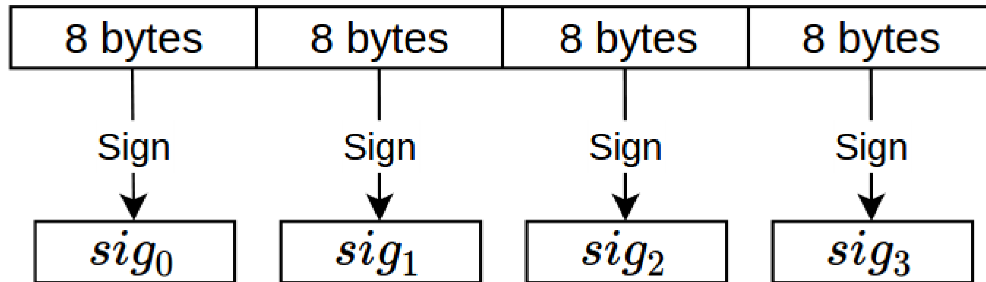


Figure 1.1: Signing a transaction with a travel document

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 1.2. 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.

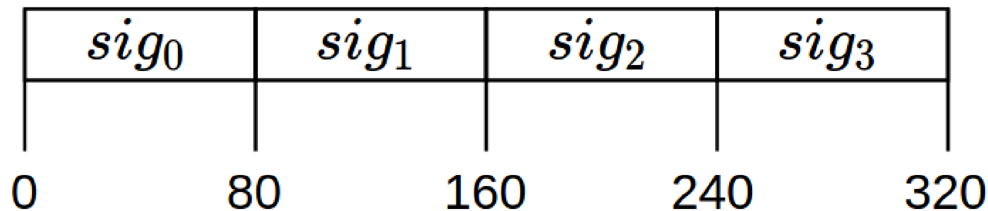


Figure 1.2: Signature storage

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 1.1. These signatures are then stored as in Figure 1.2 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.

1.2.2 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).

1.3 Alternative signing

1.3.1 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.

1.4 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:

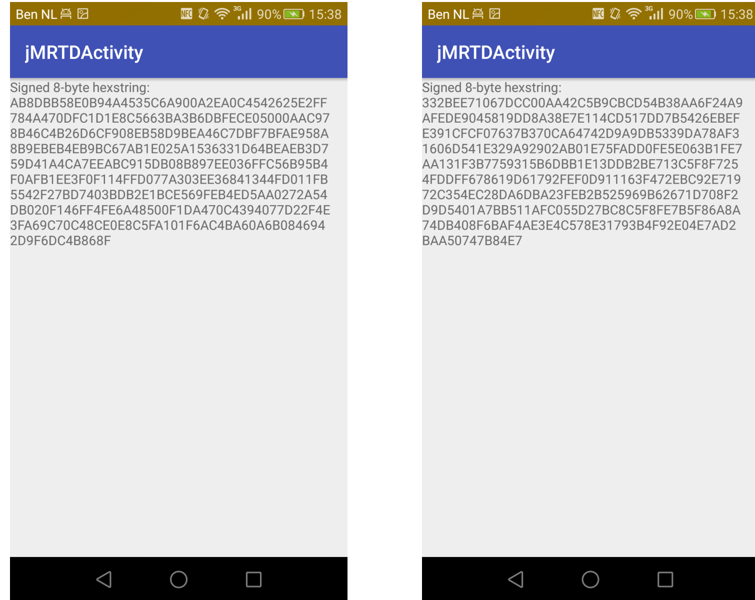
- Read the Active Authentication (AA) public key (located in Data-group 15 [14])
- Sign a blockchain transaction using the 'private key' located in the passport

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 unfortunatly 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.



(a) Output 1 (b) Output 2

Figure 1.3: Results of signing the hex string '0a1b3c4d5e6faabb' with a passport

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 1.1 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.

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 1.3 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 **??**.

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

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.

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, if the wallet has sufficient funds, which was signed by a passport

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 within at least five 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. Manual input of MRZ zone
2. Reading public key from passport using jMRTD
3. Signing of transaction with ePassport using jMRTD
4. Sending signed transaction to blockchain

A.2.2 Should Have

1. OCR Reading of MRZ zone
2. Message suggesting to go to manual input mode after OCR mode is active for dome time
3. UX designed by external party

A.2.3 Could Have

1. 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. Verify if traveling document chip is working correctly
2. Reclaim voting token from proxy-voting

A.3.3 Could Have

1. Explanation of how the digital voting process works
2. Information about voter turnout
3. Information about nearest voting stations

A.3.4 Won't Have

1. Registration for digital voting
2. Storage of account information