Cross-Platform Mobile Blockchain

Sacha Kozma

School of Computer and Communication Sciences

Decentralized and Distributed Systems lab

Semester Project

June 2018

Responsible
Prof. Bryan Ford
EPFL / DEDIS

Supervisor
Linux Gasser
EPFL / DEDIS

Acknowledgments

I would like to thank Linus Gasser who gave me the chance to do this project. I would also like to thank him for the help he provided me all along the semester, which gave me the confidence to go through the different challenges this project imposed. Beside giving me a lot of new knowledge on cryptography subjects , this project strengthens my belief of continuing my studies in the field of decentralized technologies. I would also like to thank Gaurav Gnarula, in charge of the CothorityJS and KyberJS modules, who provides me a valuable help to resolve the bugs related to their execution in the NativeScript environment. Finally, I would like to thank the *Blockchain Society*[1] for their patience during the two sessions of real-world testing of CP-MAC.

[1] `https://blockchainsociety.ch`

Contents

1 Introduction

In this report, I will present the work that has been done on the CP-MAC cross-platform mobile application. CP-MAC allows a user to use the framework Cothority, developed at the DEDIS lab, in a functional and user-friendly application. The base application containing all the primitives to use the framework was already done during a previous semester project, thus the focus in this project was to improve the user experience and to extend the application with a new functionality. The chosen functionality is the BeerCoin, a long time running joke at the DEDIS lab which consists of beer tokens that can be distributed in a group using Proof-Of-Personhood. A token can then be used to get a beer and the barman can cryptographically verify if this token is part of the allowed group and unused.

2 Corresponding background

As CP-MAC aims to provide an intuitive and user friendly interface to access the Cothority framework, multiple technologies have been abstracted into the application to give regular users the possibility to use the application without having to know the inner working of the frameworks it uses. During this project, this idea was kept in mind and influenced the different decisions that had to be made.

In this section, I will present the different technologies involved in this project and their role in the application. Note that there are several other technologies on which CP-MAC resides that are not described in this section, as they were not implicated in the realization of the project. Thus, if complementary information is needed to understand other specific parts of the application, please refer to the report covering its first implementation[4]. The last part of this section will then focus on the state of the application as it was before the project and which aspects this project have improved. These aspects will then be detailed during the rest of the report.

2.1 Cothority

The genesis of the CP-MAC application begins with the framework Cothority, developed by the DEDIS laboratory at the EPFL. The Cothority (short name for *collective authority*) framework provides the necessary components to deploy decentralized protocols over a set of servers, individually referred as a *conode*. The framework is based on a list of protocols that resolve the most basic challenges of the framework, such as *ByzCoinX*[2] that implements the consensus protocol, or *dkg*[3] that implements a distributed key generation protocol. However, in order to use these protocols, Cothority provides

[2] `https://github.com/dedis/cothority/tree/master/byzcoinx`

[3] `https://github.com/dedis/cothority/tree/master/ocs/protocol/DKG.md`

services that allows user to communicate with a conode and abstract the protocols. Examples of services are *skipchains*[4], that defines a permissioned blockchain for storing data, or *cosi*[5] that allows a user to get a collective signature of an arbitrary document from the set of conodes.

More generally, a combination of services and protocols defines a Cothority application, which is a high level abstraction to define a useful usage for a set of conodes. The scope is then quite broad, starting from simple command line interface for a specific service to more complex combinations, such as *CISC Identy Skipchain*[6], an extension of the more basic skipchain service that contains, inter alia, a module to handle distributed SSH keys rotation. I will now describe the Cothority application which has been involved in this project, the Proof-Of-Personhood.

2.1.1 Proof-Of-Personhood

The Proof-Of-Personhood results from the work done by Maria Fernanda Borge Chavez at the DEDIS laboratory. She proposes a way to solve a known problem on the Internet, which is the ability for a given service to recognize an user without compromising his anonymity, for example if a user is allowed to get only a specific amount of an arbitrary resource but doesn't want to disclose its identity. To achieve this, every member of a specific group gets a token that can be used to authenticate himself towards a service. This will also generate a tag, specific to the user and the service, which can be used to recognize that user (more information about this process is available at section 6: Linkable Ring Signatures). As the tag will be different among the services, they cannot team up to exclude or identify a specific user. The generation of these tokens is the key topic of her work.

More specifically, two types of actor take part in the generation of the tokens: the organizers and the attendees. Each organizer has a conode, which will be used for the registrations and the cryptographic operations. On the other side, each attendee have their own key pair, which corresponds to a specific identity. This being set up, the following steps can be achieved:

- First, the organizers have to define the conditions of a **PoP-Party**, which is a meeting between all the organizers and attendees. These conditions contain the location, the date, and the list of the organizers' conodes. This is called the **Party Description**. It has to be the exact same for every organizer, as the hash of the description is used to identify the party.

- At the given date and location, every organizer and attendee meet. Each attendee has to pass in front of each organizer to get his public

[4] `https://github.com/dedis/cothority/blob/master/skipchain`
[5] `https://github.com/dedis/cothority/tree/master/cosi`
[6] `https://github.com/dedis/cothority/tree/master/cisc`

key registered (i.e, adding his public key to the set of all the attendees' public key to later prove that he was present at this party). Of course, some precautions have to be taken to prevent an attendee to register multiple times with different keys. For example, each attendee could be marked with permanent pen while leaving the room, which prevents him to go back in and get registered again.

- Once every attendee is registered, the organizers can **Finalize** the party. The list of public keys is added to the end of the party description, and a collective signature of this file among all the organizer conodes is generated. By combining the file and its signature, we get a **Final Statement**, which is the necessary material to verify a signed message. In the other side, the attendee combines the final statement with his key pair, which results in a **PoP-Token**.

- Using Linkable Ring Signature (see 2.2.1) and PoP-Token, a user can sign a message, which can then be verified by the holder of the final statement.

Of course, all of these steps are implemented in the Proof-of-Personhood application of Cothority and can be accessed using an API[7].

2.2 Kyber

The Kyber library[8] is a key element in the Cothority framework. It provides a toolbox for all the cryptographic operations that are needed. Especially, it supports all the algebraic operations on group elements (such as elliptic curve points and scalars), as well as random generators and elements marshaling. It's implemented in Go, but as a lot of front-end applications (such as CP-MAC) require a JavaScript implementation, a partial and developmental port named KyberJS is available. However, it is important to note that some features from Kyber are not available in KyberJS, such as extendable-output functions (see point 2.2.2 for more information) or (Linkable) Ring Signature (see 2.2.1). In this project, a special implementation of the latter has been made as it became necessary for the evolution of the application. Its implementation is described at section 6.

2.2.1 (Linkable) Ring Signature

Ring signature gives the possibility for a user to sign a message on behalf of a group, in the sense that the receiver of such a message will be able to verify that the signer is indeed a member of the group without revealing who specifically signed it. Here, each group member is represented by his

[7] *Application Programming Interface*
[8] `https://github.com/dedis/kyber`

public key. This leads to interesting applications, as the only constraint to create such an anonymity set is to have the public key of each member. For detailed information about the inner working of ring signatures, please refer to the original publication at [5].

Kyber also provides the possibility to create linkable ring signatures, which implementation follows the paper [3]. The additional property that linkable ring signatures asserts is that two signatures from the same signer can be linked. In other words, a link scope is used during the signing and verification process, then, in addition to the signature, a linkage tag is produced. This tag is unique to the user and to this scope. Thus, given two messages signed under the same linked scope, the verifier of the signature can assert (1) whether the signers are members of the group or not and (2) whether the two signers are the same member or two different ones. A tag can also be used to remember if a specific user have already been "seen" under the current link scope. This application will be particularly useful in CP-MAC. More generally, useful applications can be found for linkable ring signature. For example, different services (Wikipedia, E-Voting, and so forth) could get their own view on a group, without them being able to collude to find out link between users, or to exclude specific users.

2.2.2 BLAKE2

During the signature and verification process of linkable ring signatures (as defined by [3]) a cryptographic hash function is required. In Kyber, this task is handled by the BLAKE2 function. BLAKE2 comes with several interesting characteristics : it's faster than most of the current popular hash function (SHA or MD5) and it's at least as secure as SHA-3 (see [2]). BLAKE2 is available in two versions :

- BLAKE2b, which produces digests of size up to 64 bytes and is optimized for 64 bits processors
- BLAKE2s, which produces digests of size up to 32 bytes and is optimized for 32 bits processors

Extendable-output functions

BLAKE2 also offers the possibility to generate extendable-output functions (XOFs) from each variants of BLAKE2, respectively named BLAKE2Xb and BLAKE2Xs. A XOF allows generating digest of arbitrary length (although there is a limit of 256 GiB for BLAKE2Xb and of 128 GiB for BLAKE2Xs), which is especially interesting to create deterministic random bit generators (DRBG). In Kyber, both of BLAKE2 versions and their respective XOF are implemented and BLAKE2Xb is used in the computation of ring signatures. However, these XOFs have not been ported yet to KyberJS.

2.3 CP-MAC

As CP-MAC has already been extensively presented in [4], I will rather do a quick overview of the application and then focus on the state of the application as it was before the beginning of this project. This will then allow me to present the road map of the execution of the project to underline the components of the application that have been improved.

CP-MAC implements a simple interface to use two major applications of the Cothority framework : the Proof-Of-Personhood and the CISC Identity Skipchain. However, these are services that are though to use for a non-experimented user who doesn't have access to an easy-to-use front-end. CP-MAC tries to resolve this problem by combining the mobility offered by a smartphone application with a more intuitive user interface than CLI[9].

NativeScript

Under the hood, CP-MAC is made above the NativeScript framework, which allows developing a native cross-platform mobile application in JavaScript (for the application logic), XML (for UI elements) and CSS (for UI styling). It is a great advantage, as styling an application on NativeScript is essentially the same as styling a website, apart from some exceptions. The vast majority of CSS attributes are available, even some advanced ones, such as animations. Also, on both platform, NativeScript allows calls to native APIs such as *Time* on Android, or *UIAlertView* on iOS. It's then possible to manipulate directly those objects from the JavaScript code.

Also, NativeScript gives the possibility to create Plugins, which are native pieces of code with a common JavaScript interface. For example, one could develop advanced Plugins such as Web Sockets or Fingerprint Authentication that have their specific native implementations on iOS and Android, but that will be manipulated through a single object in JavaScript.

It's however important to note that NativeScript does not directly compile JavaScript to native code. Instead, it uses the power of JavaScript virtual machines to inject global objects representing the native APIs, that are then used to intercept native calls and let NativeScript Runtime redirect them to the native libraries. Apart from that, the application logic is executed in JavaScript by their respective engine, i.e *V8* on Android and *JavaScriptCore* on iOS[10].

3 Planned improvements

The first project dedicated to CP-MAC focused on implementing the building blocks of the application and led to a functional library containing all

[9] *Command Line Interface*
[10] as of June 2018

the necessary parts to continue its development: PoP-Party management (creation, publication, attendees registration and party finalization), user management, and so on. However, as the major part of the work was dedicated to this huge library, the application main weakness resides in his usability. I will then present the main points that I had to work on to improve this weakness.

Interface

As user interface couldn't be improved in the initial implementation of CP-MAC due to lack of time, it has been decided to define new guidelines and implement consistent rules about UI throughout the application. More details can be found at section 4: Interface and user experience.

Cothority v2

The work that has been done on Cothority since the first version of CP-MAC is quite consequent, and several changes to the API appeared. It was then important to make the application consistent. The application now supports Cothorithy v2 standards, such as TLS addresses, hexadecimal keys (instead of previous Base64 keys), and so on. Also, KyberJS and CothorityJS (a Node module that takes care of all the communications between CP-MAC and a conode) are now used in the application and replace the current specific CP-MAC implementations, allowing a more homogenized framework utilization.

Proof-Of-Persoonhood enhancements

Until now, the application allowed the organization of only one party at a time and the support for the attendee part was very limited. One of the idea that popped up was the addition of multiple parties support (for attendees and organizers), with intuitive status tracking (in configuration, finalized, ...).

Attendees should also be able to generate their PoP-Tokens and make use of them (i.e sign messages).

Finally, the way the party configuration is shared between the organizers had to be switched from the current (transitional) way that uses the famous text storing service PasteBin[11] to a more seamless solution. Details about these improvements are available at section 5: PoP Part.

Proof of concept

In order to give CP-MAC a more usable feeling, a simple proof of concept has been designed: it starts from a long-running joke at the DEDIS laboratory called BeerCoin, where a group of person could

[11] `https://pastebin.com`

claim a free beer per day/week/month at the expense of the laboratory. The usage of Proof-Of-Personhood would then be perfectly fitted, as it would ensure that one member can only have one beer per time period, without revealing which member already had his beer. The implementation of this feature is detailed in section 7: BeerCoin Part.

4 Interface and user experience

First, new colors have been chosen to reflect a more sober design :

- is the **primary color**. It's largely used throughout the application and is defined as *rgb(192, 204, 217)*. In the code, the SCSS[12] variable **$accent-light** should be used.
- is the **secondary color**. It's used to give contrast on specific elements and is defined as *rgb(129, 144, 164)*. In the code, the SCSS variable **$accent-dark** should be used.

Also, two principles have been applied to the different screens to try giving a cleaner design :

- As little information as possible should be displayed on the screen at the same time. Typically, everything not useful to the user is considered as surplus and should be dropped. For example, hexadecimal string representing the user keys or conode IDs have been put apart.
- In line with the previous guideline, none of the space given to the elements displayed at the screen should feel too tight. Thus, some padding and margin can be applied to space the elements.

4.1 Lists

As this element is very recurrent in the CP-MAC application (lists of conodes, PoP-Parties, CISC Identity Skipchains, and so on), some general guidelines were defined. First, a global CSS class **basic-list-decorated** is available and can be applied to any NativeScript object of type **RadListView**. Inside, different element types are available (numbers refers to Figure 1 at page 8):

1. Corresponds to the main title of the list entry. It's global CSS class is **list-title**.
2. Corresponds to a more specific detail of the entry.

[12] Stands for *Sassy CSS*. It's a super set of the standard CSS and provides useful features, such as variables or element nesting. SCSS files are then compiled to regular CSS files.

3. Corresponds to an optional status text reflecting the actual state of the underlying object represented by this entry. It's global CSS class is **status-text**.

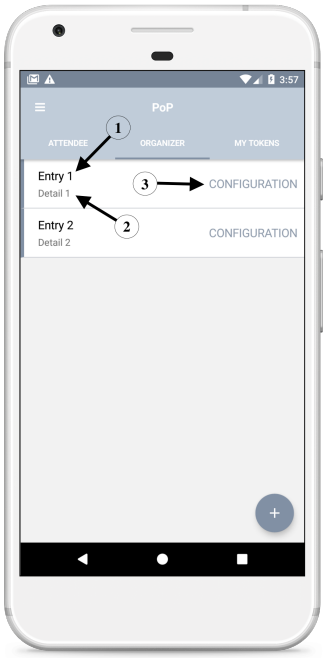


Figure 1: Example of a typical list in CP-MAC

4.2 QR Code Presentation

The modal page that presents a QR code to other users has also been updated. As this page is frequently used (a lot of information is exchanged by this medium, especially for the Proof-of-Personhood part), a special attention has been ported to create a nice-looking page. The result can be seen on Figure 2 at page 9.

5 PoP Part

The Proof-of-Personhood is the part where most of the work has been focused. Some core features changed, as well as auxiliary functionalities that allows user to spare some useless (repeated) configurations. The major changes will be described in this section. Some minor improvements were

Figure 2: Example of the modal QR code presentation page

also done, such as verifying if the conode was already linked before asking for the PIN.

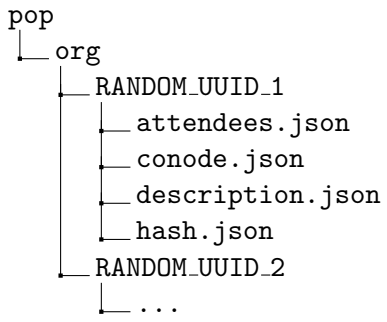
5.1 Multiple parties support

As stated previously, CP-MAC formerly handled only one party at the time. This limitation has been removed, and a user can now create, join or delete parties.

5.1.1 Implementation

The singleton object that was taking care of all the party management operations has been converted to a simple object, where each party is now stored in his own directory. On creation, each party gets attributed a unique identifier with UUIDv4[13] format. The directory in which the party is stored is named after this identifier. The general directory structure now looks as follows :

[13] `https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random)`



The use of this structure allows CP-MAC to just iterate over the entries of the `org` directory to lists the parties, which can be elegantly implemented in NativeScript as it provides interface to execute an action over each entry of a directory using a callback function.

5.1.2 Party statuses

The PoP part of CP-MAC makes use of the status text abstractly described in section 4.1 to show how the party is progressing across time. As these statuses aren't implemented in the Cothority back-end, here is a presentation of the different statuses and how they are deduced from the available data. Most of the statuses are inferred from the response of the `FetchRequest` message, that fetches the final statement from a conode :

`LOADING` is set until a response from the conode is received.

`CONFIGURATION` is set when the returned message is a `"No config found"` error. It informs that the party is only stored locally, thus the conode don't have knowledge of it.

`PUBLISHED` is set when the returned message contains a final statement, but its list of attendees is empty. It informs that the party is stored on the conode, but not yet finalized.

`FINALIZING` is set when the returned message contains a final statement, but its signature is empty. It informs that the instruction of finalizing has been given, but not on every conode.

`FINALIZED` is set when the returned message contains a final statement, with a non-empty list of attendees and a signature. It informs that the instruction of finalizing has been given on every conodes.

`ERROR` is set in the other cases. It informs that either the network is unavailable or an error occurred, thus the status of this party is unknown.

By using NativeScript primitive for time operations, the final statement can be regularly polled to give informative statuses to the user.

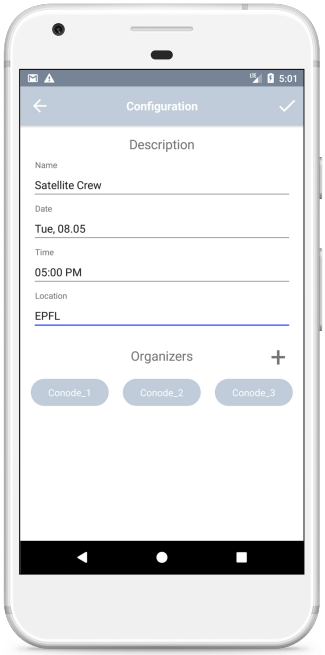


Figure 3: Example of a party configuration

5.2 Party proposals

During the creation of a PoP-Party, its description must be shared through all the organizers. Until now, the chosen temporary solution was to use PasteBin, because the usual way of sharing information in CP-MAC (i.e using QR Code) don't have a sufficient capacity to store a party description. However, this had several drawbacks, such as depending on a third party service or giving public access to the party description.

A new procedure has been designed and works entirely on conodes. The general idea is that whenever an organizer publishes a party, the leader conode is used to propagate the description to the other conodes of the party. This way, the rest of the organizers can poll their conode and retrieve the description directly into CP-MAC. The configuration can then be opened in the standard party configuration page of CP-MAC, as seen on Figure 3 at page 11.

5.2.1 Implementation

To achieve this, one message behavior has been updated in the Cothority back-end, and a new one has been added :

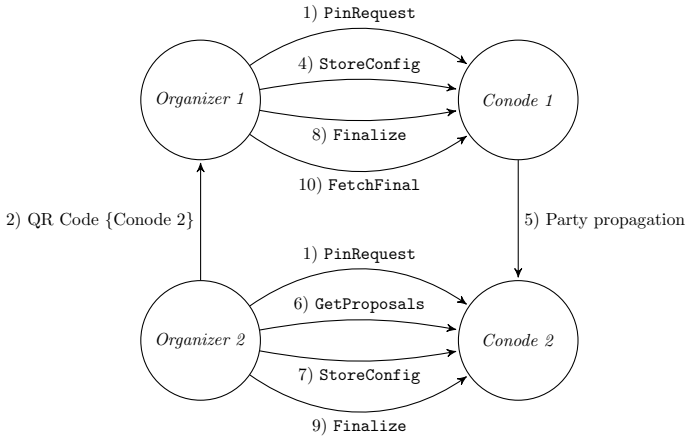


Figure 4: Graph of a party proposition and retrieval

StoreConfig has been updated : besides storing a configuration on the conode, it will check if the conode is the leader one (i.e if it's the first one in the roster) and if so, it propagates the description to the rest of the roster. The conodes that receive the description will store it until it has been confirmed with `StoreConfig` within a maximum time period of one hour.

GetProposals has been added: it retrieves every party proposals available on the conode.

More specifically, here is the typical workflow that might happen for the complete creation of a party involving two organizers (Organizer 1 / Organizer 2) with their respective conode (Conode 1 / Conode 2). The numbers refer to the Figure 4 at page 12, where each link represents either an action or a message to the Cothority back-end.

1. Each organizer authenticate to their respective conode
2. *Organizer 2* presents a QR Code containing his conode information, {Conode 2}
3. *Organizer 1* creates a new party and adds *Organizer 2* to it by scanning the QR Code he is presenting
4. *Organizer 1* can then publish the party to his conode
5. *Conode 1* is the leader (first conode in the roster) so it propagates the party description to *Conode 2*
6. *Organizer 2* can now retrieve the proposals list and notices a new entry

7. If the description of the party suits *Organizer 2*, he can publish it to his conode

8. When the party should end, *Organizer 1* finalizes the party

9. *Organizer 2* also finalizes the party

10. Any of the organizers can now fetch the final statement

5.2.2 Drawbacks and future work

However, some drawbacks still remain. For example, if an organizer decides to change the description of the party, he has to start over and publish a new party. This isn't convenient, as this require more time for the organizer than just editing the fields he wants to modify. Also, the other organizers will see a second entry when retrieving the proposals from their conode, which can be confusing and is more error-prone. Additionally, in the current schema, only one organizer decides of the details of the party, and the others only have the choice to accept (publish the party) or refuse (ignore) the current proposition. Again, a better procedure would allow any organizer to propose changes to the party description and when everybody agrees, they could publish it.

A solution to this problem could reside in the Cothorithy Cisc Identity Skipchains. Indeed, Cisc allows users to share key/value data using a private blockchain, on which any member can propose changes and cryptographically vote to accept or refuse changes. A threshold can also be set to define the minimum number of participant who should accept the changes to effectively add them to the blockchain. This would perfectly suit our requirements: each organizer could propose changes that will be reviewed by the rest and when everybody agrees on a description, they can publish it to their conode.

5.3 Attendee part

CP-MAC now supports a completely independent section for the management of attendee parties: a user can add a party by scanning the QR Code on the organizer's CP-MAC instance and benefits from the same status monitoring the organizers have. To keep things consistent in the code logic, a new base class **Party** has been extracted from the common traits of an attendee's party and an organizer's party. Each part can then have their respective **AttParty** and **OrgParty**, inherited from the base class.

The implementation of the **AttParty** will not be described here, as its implementation follows the **OrgParty** class (multiple parties support, directory structure, status retrieval). The only feature specific to the attendees is the way token are handled.

PoP-Token signature

In contrast to the organizers, attendees make use of the PoP-Tokens, thus, when a party is finalized, the user can choose to generate his PoP-Token. CP-MAC handles the conversion from the final statement to the PoP-Token and makes it available to the user with all the other ones gathered through the previous parties.

This list of PoP-Tokens will now be useful to the users as it is henceforth possible to sign arbitrary data transmitted using a QR Code and present the resulting signature (with the optional tag) via the same medium. Signatures are generated using (linkable) ring signatures, as described in section 6.

6 Linkable Ring Signatures

Linkable ring signatures are the missing link in CP-MAC to make the PoP-Tokens useful. As stated before, they're necessary to sign data, but they are also required for the verification. As this type of digital signatures is not available in KyberJS, it has been necessary to implement it directly in CP-MAC.

6.1 Implementation

The sign and verify algorithms are implemented in **RingSig.js**. It contains also every required auxiliary methods. The execution of the algorithm strictly follows the one implemented in the Kyber Go version and described in [3]. However, as there are major differences between the Go language and JavaScript, some inner mechanisms had to be adapted.

First, as we saw in the BLAKE2 introduction at point 2.2.2, BLAKE2Xb is used by Kyber ring signature algorithm but is not available in KyberJS. Two approaches were considered, each one having their drawbacks :

Implement BLAKE2Xb for CP-MAC
: As there are several modules in JavaScript that implement BLAKE2b, it is possible to create an instance of BLAKE2Xb without coding the complete algorithm, as a BLAKE2X function can be derived from any BLAKE2 instance (be it BLAKE2b or BLAKE2s). The procedure is described in [1]. This would have assured a complete compliance with the Kyber implementation. However, the remaining time available to complete this project wouldn't have allowed implementing and testing it thoroughly, on top of adding the ring signatures feature.

Use another hash function
: The nearest hash function that I could find is an implementation of BLAKE2Xs implemented in the StableLib[14]. As it provides the same

[14] `https://github.com/StableLib/stablelib/tree/master/packages/blake2xs`

capabilities than BLAKE2Xb, it can be used in ring signature as a replacement. However, the drawback of this solution is that it breaks the compatibility with the Go (reference) version of Kyber. Note that this drawback can be highly moderated by parameterizing the hash function in Kyber and KyberJS, as stated at point 6.3. Due to the lack of time, this solution has been chosen.

The second change that had to be operated concerns the way Kyber marshals its data. This happens during the conversion of the cryptographic elements (points, scalars) to standard byte arrays. Kyber uses a DEDIS library called **fixbuf**[15] that allows a fixed length binary encoding of arbitrary Go structures. As there isn't an equivalent library from DEDIS for JavaScript, I implemented a method that reproduces the behavior of **fixbuf** specifically for ring signature structure. Effectively, here is the Go structure representing an unlinkable ring signature :

Listing 1: Unlinkable ring signature structure

```
type uSig struct {
    C0 kyber.Scalar      // generated during the signing process
    S  []kyber.Scalar    // the length of S equals the number
                         // of public keys in the anonymity set
}
```

and the one representing a linkable ring signature :

Listing 2: Linkable ring signature structure

```
type lSig struct {
    C0  kyber.Scalar     // generated during the signing process
    S  []kyber.Scalar    // the length of S equals the number
                         // of public keys in the anonymity set
    Tag kyber.Point      // the tag, unique to the signer under
                         // the given scope
}
```

According to **fixbuf**, those elements will be marshaled and concatenated. This can be reproduced in CP-MAC, as KyberJS (like Kyber) allows marshaling cryptographic elements and those structures only contains points or scalar. It's then sufficient to concatenate the resulting byte array of each element, in the right order.

With these changes in mind, the implementation has been done following these steps : first, a specific development version of Kyber for which the hash function is replaced by BLAKE2Xs has been compiled. Also, Kyber and KyberJS have been configured to use a deterministic random function, this

[15] `https://github.com/dedis/fixbuf`

way the results between each implementation of the ring signature algorithm can be compared. This facilitated the work as it allowed to verify at each step the coherence of the results.

6.2 Unit testing

The tests used to verifiy the **RingSig.js** implementation are the same as in the Go version, available in the **sig_test.go** test suite. Particularly, three scenarios are tested. In each case, signature are tested against the correct and the wrong message, plus the tags are verified (length and validity) when applicable. Here are the different scenarios :

- A trivial unlinkable signing process with a single member anonymity set is tested.
- An unlinkable signing process with a small anonymity set of three members is also tested.
- For linkable signatures, an anonymity set of three members is created and a scope is defined. The tags generated when verifying against the good message and a wrong one are then verified.

6.3 Future work

The most important goal for a future improvement is to make KyberJS fully compatible with Kyber, either by implementing BLAKE2Xb in JavaScript or by parameterizing the hash function. The latter would give a high degree of freedom: for example, it could be possible to create a new suite in Kyber that uses BLAKE2Xs as the hash function. KyberJS would then have to support parameterization of the hash function for a suite, which is currently not the case. However, the former solution have the advantage of being more efficient, as BLAKE2Xb is optimized for 64 bits processor, which begin to be the standard in today smartphones. Of course, by combining the two solutions, we get the best of both world.

7 BeerCoin Part

With the different primitives that are now available on CP-MAC, it is possible to create a real use-case. Here, it has been decided to realize a long-running joke at the DEDIS: the creation of a BeerCoin.

7.1 Description

The idea behind BeerCoin is that a group of people could each benefit from a free beer every month, week, or day at the expense of someone else. It's

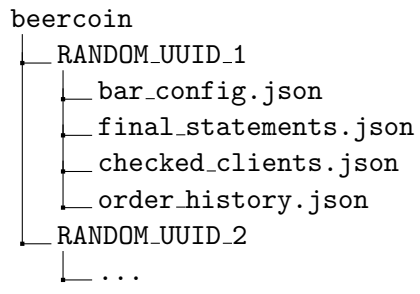
also important to preserve the anonymity of the members of the group. For example, it should not be possible to recognize a user from his signature, as well as deducing information about users between two periods.

Concerning CP-MAC, it should be possible to create a bar from the application, with the possibility to define which group is allowed to get these BeerCoins, and the period before renewal. The bartender could then verify from CP-MAC if a user is allowed to have a beer, and the order history should be kept to allow the BeerCoin supplier to later pay his bill.

7.2 Implementation

To implement the BeerCoin, the PoP-Tokens have been used, because, combined with the linkable ring signature, they fulfill all the above requirements. Here is the way BeerCoin works in CP-MAC: at first, a user has to create a bar. He can select the group of people which will get the BeerCoins. To simplify this process, CP-MAC will present every group of people from the PoP-Parties in which the user was involved. In fact, when a party gets finalized (let it be an attendee party or an organizer party), the final statement is added to a *bank*[16] of final statements. This way, the user can easily choose for which group he wants to pay the beers. He then chooses a time period and a name for his bar, which concludes the configuration of the bar.

Right after the creation, the directory structure for a bar follows this schema :



As we can tell from the `RANDOM_UUID` entries, the structure follows the same name generation than a **Party**[17]. Also, here is the usage of each file :

bar_config.json stores the bar information described above. It also stores the beginning date of the last period. If the difference between the current date and the last reset date is bigger than the defined period for this bar, the list of already seen clients should be reset, and the beginning date updated to reflect the current period.

[16] This *bank* of final statements is in fact the singleton defined in **PoP.js**, which was already implemented.

[17] please refer to point 5.1 for more details.

final_statements.json stores the final statement linked to the party representing the group of people enjoying the BeerCoins. It will be used to verify the signatures of the clients.

checked_clients.json contains the list of the already seen clients. Their tags are saved in this list (see point 7.2.1 for detailed explanations).

order_history.json contains the date of each order that still has to be paid. It can be emptied from the user interface, when all the beers have been paid.

7.2.1 Client verification

Once the bar is set up, the verification of a client is pretty straightforward: he first has to scan a QR Code containing the bar information, composed of a nonce and a linkage scope. The linkage scope must be unique for that period, as it will allow the bar to recognize if a client came twice in the same period. Hence, the linkage scope is generated as follows :

$$scope = bar\_name \| frequency \| year \| month \| day \quad (1)$$
$$frequency \in \{daily, weekly, monthly\}$$

For example, a bar called Satellite that offers a free beer every day for its members would generate the linkage scope *Satellitedaily201868*, assuming the current date is June 8th 2018. The client now signs the nonce with his private key and the linkage scope. The bartender scans a QR Code containing the signature, which is then verified using the final statement public keys as the group of anonymity. If the resulting tag is already in the list of checked clients or if the signature is not valid, the bar refuses the order. Otherwise, the tag is added to the checked clients list, and a new order is logged in the history.

7.3 Drawbacks and future work

One of the drawback of this system is the combination of ring signatures with QR Code. Indeed, as we have seen in the listing 2 (page 15), one of the array size in the signature is proportional to the number of public keys in the anonymity set. However, QR Code maximal capacity is 2954 bytes[18], which could be exceeded depending on the number of member in the group.

But is this limit really constraining ? Let's consider a real-world case: CP-MAC bar implementation use the `edwards25519` curve of KyberJS for the ring signatures. On it, points and scalars are marshaled to 32 bytes long arrays. Again, by referring to listing 2, we can deduce the following inequality :

[18] `http://www.qrcode.com/en/about/version.html`

$$32 + 32 + 32 * n = 32 * n + 64 \leq 2954 \quad (2)$$
$$\implies n \leq 90.3125 \quad (3)$$

where n is the number of member in the anonymity set.

From (3) we get that the maximum size of an anonymity set is 90. In CP-MAC, this wouldn't really be an annoying boundary, however, in a real-world situation, this could cause troubles.

A lot of new features could be thought for future work, such as integrating an in-app payment method, or even by allowing people to exchange BeerCoins as any other decentralized token. It could also be possible to work on the drawback explained above, for example by modifying the protocol and integrating a conode as intermediary. CP-MAC would then just serves as a control board, while all the signing processes and signature exchanges are done directly between the client and the conode.

8 CP-MAC installation

The following section will describe the installation steps to get CP-MAC up and running on your computer. Note that even if NativeScript is compatible with Linux, Windows and macOS, only macOS can run the application on the iOS simulators, due to Apple limitations. However, a new desktop application called NativeScript Sidekick allows running cloud-based builds of iOS applications on Windows and Linux.

1. Install NativeScript

   As NativeScript has a relatively high update frequency and a complete up-to-date guide about its installation[19], the entire description of the steps needed to install NativeScript won't be copied here. However, here are the main steps to follow :

   (a) Install Node.js[20]

   (b) Install the NativeScript command-line interface, which is in fact a simple Node application, using

   ```
   $ npm install -g nativescript
   ```

   (c) Install iOS and Android requirements using the NativeScript automatic dependences installer available on the installation guide page.

   (d) Verify the installation using

[19] https://docs.nativescript.org/start/quick-setup
[20] https://nodejs.org/

```
$ tns doctor
```

It should output *"No issues were detected"*. If it's not the case, the next installation instructions won't work.

(e) Optionally, NativeScript provides plugins for the editor Visual Studio Code[21] or WebStorm[22]. These are not mandatory, but they can help by adding code assistance or templates.

2. Clone CP-MAC repository

   The following command will take care of this step :

```
$ git clone https://github.com/dedis/student_18_xplatform
```

3. Install CP-MAC dependencies

   A Makefile is available and will install the necessary components, as well as applying some patches to ensure that CP-MAC compiles and runs correctly.

```
$ cd student_18_xplatform/cpmac
$ make clean-install
```

4. Run CP-MAC

   The application can now be started with one of the following commands, depending on the desired platform :

```
$ make run-ios
$ make run-android
```

9 Known bugs

In this section I will describe the bugs that are either present in the application or have been fixed but are worth to be known. As some problems are related to JavaScript modules or directly to NativeScript, it's possible that the fix may not be necessary anymore in the future. This also allows understanding the necessity of the Makefile: by now, every solved bugs that concerns inherent issues of using NativeScript are fixed using patches, which are applied during the installation with the Makefile. For future bug fixes, this approach is recommended as it keeps the structure of the application consistent.

[21] `https://www.nativescript.org/nativescript-for-visual-studio-code`
[22] `https://plugins.jetbrains.com/plugin/8588-nativescript`

Already known bugs

The list of bugs stated in the report describing the first implementation of CP-MAC are still relevant. More precisely, the WebSocket problem on iOS has still not been fixed yet. The complete list can be found in [4].

KyberJS dependencies

Sometimes, NPM packages such as KyberJS use built-in Node modules (`crypto`, in our case) that aren't just JavaScript code but make use of primitives that are specific to the platform on which they run (the browser, or Node). For some of them, an equivalent can be found for NativeScript, such as `nativescript-crypto` for the previous example. But the faulty modules that imports platform-specific packages still need to update all their **require**() statements to point to the correct versions. This can be handled by the module `nativescript-nodeify`, which installs a NativeScript hook[23] that will traverse every modules and sub-modules to find incorrect **require**() calls to non-compatible packages and replace them with their NativeScript version.

However, in his current version[24], `nativescript-nodeify` excludes every package containing an Organization scope, which prevents KyberJS from being converted, as KyberJS package is `@dedis/kyber-js`. An issue has been opened on GitHub[25] to find out why such a rule is applied, but no answer has been given yet. Thus, a patch applied by the Makefile has been created to remove this condition. Future versions of `nativescript-nodeify` will maybe fix this issue.

iOS version delay

Unfortunately, due to a bug solved very lately during the project, the iOS version of CP-MAC is not as polished as the Android version. Particularly, some details on the UI still need some work, and some instabilities due to the lack of tests are present. However, some work on this platform is planned and will probably fix these details soon.

The bug that caused this delay was due to the way `UglifyJS` mangled[26] the module CothorityJS. The version of the JavaScriptCore used on iOS was reporting two variables for having the same name, though actual JavaScript specifications on variable scope should allow this case. This was fixed by setting some specifics flags on `UglifyJS` to prevent this situation[27].

[23] Hooks are scripts that are executed at specific moments, such as before the compilation begins.

[24] 0.7.0 as of June 2018.

[25] `https://github.com/EddyVerbruggen/nativescript-nodeify/issues/33`.

[26] Mangling is the process of optimizing JavaScrit code size, for example by renaming variables or functions to single letters.

[27] More details can be read at `https://bugs.webkit.org/show_bug.cgi?id=171041`.

References

[1] Jean-Philippe Aumasson et al. *BLAKE2X.(2016)*. 2016. URL: https://blake2.net/blake2x.pdf.

[2] Jean-Philippe Aumasson et al. *The Hash Function BLAKE*. Springer, 2014.

[3] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. *Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups*. Cryptology ePrint Archive, Report 2004/027. 2004. URL: https://eprint.iacr.org/2004/027.

[4] Vincent Petri and Cedric Maire. *Cross-Platform Mobile Application for the Cothority*. EPFL, 2018. URL: https://github.com/dedis/student_17_mobile/tree/master/report.

[5] Ronald L Rivest, Adi Shamir, and Yael Tauman. "How to leak a secret". In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2001, pp. 552–565.