Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Scalability of Encointer - a Proof-Of-Personhood Cryptocurrency

Master's Thesis

Piero Guicciardi
gpiero@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Tejaswi Nadahalli
Alain Brenzikofer
Prof. Dr. Roger Wattenhofer

January 5, 2022

Acknowledgements

I thank my supervisors Alain Brenzikofer and Tejaswi Nadahalli for supporting me and guiding me through the process of this thesis while always leaving me enough space for my creativity to unfold. Further, I would like to thank my girlfriend, friends and family for giving me the necessary distraction and last but not least, special thanks go to Satoshi Nakamoto for inventing blockchains and thus making this thesis possible in the first place.

Abstract

Encointer is a cryptocurrency using a novel consensus mechanism called Proof-Of-Personhood which is based on people meeting physically at regular intervals in order to attest each other's personhood. After having a digital identity which is provably linked to a real human being, each user receives a universal basic income in one of Encointer's local community currencies. The protocol requires complex logic to be calculated on the blockchain like verifying geolocations or assigning users to meetups in order to guarantee the security of the system. To ensure Encointer's scalability to a large number of users it is crucial that those computations are as efficient as possible which is not yet the case. In this thesis we present solutions to three major scalability problems of Encointer, evaluate them in terms of runtime and security and provide implementations that are ready to be merged into Encointer's production system.

Contents

Chapter 1

Introduction

1.1 Motivation

Since the publication of Bitcoin[1] in 2008 a lot has happened in the blockchain space. Many ecosystems like Ethereum[2] or Polkadot[3] emerged and evolved but also limitations of those technologies came to light. One of them being the consensus mechanisms used at the core of the blockchain systems. Bitcoin's Proof-Of-Work has the issue of giving power only to the owners of specialized hardware and not being very friendly to the environment, while Ethereum's Proof-Of-Stake solves the environment issues but gives all the power to the rich. A new approach to this issue is called Proof-Of-Personhood, which aims at establishing a One-Person-One-Vote paradigm. Encointer[4] is a cryptocurrency that implements Proof-Of-Personhood and issues a universal basic income to all its participants. Like many other blockchain systems, Encointer suffers from scalability problems.

1.2 Goal

The goal of this thesis is to design solutions for three major scalability problems of Encointer as well as implementing them into the existing codebase.

1.3 Outline

In Chapter 2 we introduce the theoretical background of this thesis and Chapter 3 discusses related work. Chapter 4, Chapter 5 and Chapter 6 cover the three scalability problems by introducing the problem, presenting and evaluating our solution and giving directions for future improvements. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background

Encointer[4] is a blockchain ecosystem that allows for the creation of local community cryptocurrencies which provide their users with a universal basic income (UBI). In order to prevent sybil attacks on the issuance of a currency, users have to attend regular key signing meetups (ceremonies) where participants attest the personhood of each other - therefore the term Proof-Of-Personhood. The ceremonies take place every 41 days at high sun and shortly before the ceremonies take place, users are randomly divided into groups and assigned to a meetup location. In order to prevent attackers from attending multiple meetups in one ceremony phase, the meetup locations need to have a minimum distance between each other. Because the meetup time is determined by the position of the sun, distance is measured in solar trip time instead of meters (see Definition 2.1).

Definition 2.1 (Solar Trip Time)**.** The time delta between the time it takes for a human to travel from A to B and the time it takes the sun to travel the same distance relative to the earth. If positive, a human is traveling slower than the sun.

A new community currency needs to be initiated by providing the geographic bounds of the community and the meetup locations where the key signing ceremonies will take place. In addition a setup ceremony with 3 to 12 bootstrappers has to be held. After a community is setup, the protocol is divided into the following phases that repeat every 41 days:

1. **Registration**
   Participants register for a ceremony at least 24 hours ahead of time.
2. **Assignment**
   Users are randomly assigned to meetup locations in groups of 3 to 12 persons.
3. **Meetup**
   Participants meet at their assigned locations and attest the physical attendance of each other using the Encointer mobile app.

4. **Witnessing and Validation**
   All users submit their attestations to the Encointer blockchain, where the attestations are validated. Successful validation grants the user the right to the UBI and gives her reputation for the coming ceremony phases.

5. **Reward**
   A UBI is rewarded to all participants that passed the validation, meaning that there is proof that they physically attended their assigned meetup.

Encointer uses a demurrage[5] mechanism, by which currency is constantly being destroyed in order to incentivize users to spend their money and thus actively take part in the economy. Recently, Encointer was getting more attention as they are aiming to become a Polkadot parachain and a security analysis of the protocol was performed by external researchers[6].

2.1 Polkadot and Substrate

Encointer is built on top of Polkadot[3], which is a heterogeneous multichain protocol allowing for blockchain interoperability and pooled security of multiple blockchains. It consists of a relay chain that is responsible for consensus and of multiple so-called parachains that are secured and connected to each other via the relay chain. Encointer aims at becoming a parachain of Polkadot and uses Substrate[7], a development framework that lets developers create blockchains that will be interoperable with the Polkadot relay chain. We also used Substrate's native benchmarking framework[8] in order to benchmark the algorithm in Chapter 4.

2.2 Rust

Most of the code in this thesis is written in Rust[9], a programming language concerned with writing fast and reliable code and bridging the gap between high-level convenience and low-level flexibility. Rust has a powerful standard library[10] but as Substrate's runtime code is compiled into a WebAssembly[11] binary, which should be as light-weight as possible, one has to develop the blockchain logic without the standard library.

2.3 Geohashing

Geohashing[12] is a technique of mapping a geolocation specified by its latitude and longitude to a hash value of constant length, where one hash value corresponds to a rectangular area on the earth. The more digits the hash value has,

the smaller is that area. As in traditional hashing we call a geohash value and its corresponding area a bucket. At the equator, the area of a geohash bucket is a square and with increasing absolute latitude the area becomes rectangular, as the distance around the world decreases the closer we get to the poles.
For example, the ETH main building is located at coordinates `47.376305, 8.547467`, which map to the hash value `u0qjd`. A visualization of this geohash bucket can be found in Figure 2.1.
Geohashing forms the basis of more complex data structures to store and retrieve geolocations efficiently, like for example GeoTree[13].

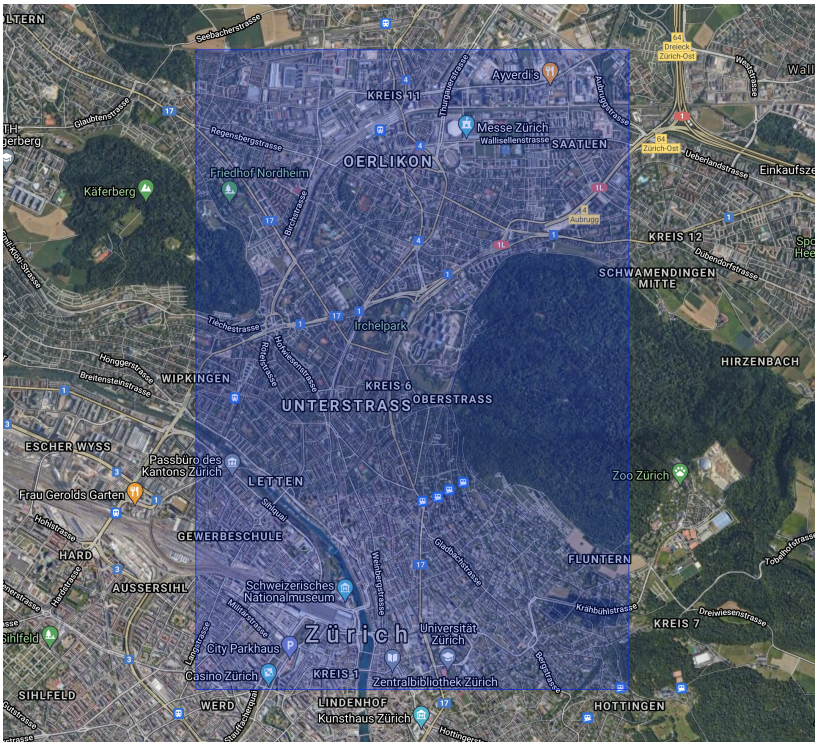


Figure 2.1: Geohash bucket containing the ETH main building, encoded with a hash length of 5 characters

Source: https://www.movable-type.co.uk/scripts/geohash.html

2.4 Modular Arithmetic

To understand Chapter 5, basic knowledge of group theory and modular arithmetic is required. In this section we give a very brief overview of concepts necessary to understand this thesis.

2.4.1 Cyclic Groups

For any n, the integers modulo n form a cyclic group under the addition operation. Any group element that is coprime to n is a generator of this group. When choosing a random prime number p that might be greater than n, we know by the Euclidean algorithm[14] that the remainder $r = p \mod n$ is coprime to n and therefore also a generator of the group. If n itself is a prime number, any non-zero group element is a generator by definition.

2.4.2 Modular Addition and Multiplication

When adding numbers modulo m, the following holds:

$$(a + b) \mod m = ((a \mod m) + (b \mod m)) \mod m \tag{2.1}$$

A similar equation holds for multiplication:

$$(a * b) \mod m = ((a \mod m) * (b \mod m)) \mod m \tag{2.2}$$

Chapter 3

Related Work

In the following, we present a broad area of topics related to this thesis.

3.1 Proof-Of-Personhood

Many approaches exist to bring more democracy to blockchain ecosystems. The main attempt of these protocols is to achieve a shift of paradigm from "One-CPU-One-Vote" (Proof-Of-Work) or "One-Dollar-One-Vote" (Proof-Of-Stake) to "One-Person-One-Vote". Those protocols are summarized under the term Proof-of-Personhood (PoP)[15]. PoP tries to establish a concept called accountable pseudonyms[16], such that users can be anonymous but can be held accountable at the same time. One interesting approach is PoPCoin[17], which relies on physical meetups called pseudonym parties similar to Encointer.

Another broader term found in the literature is Self Sovereign Identity, which refers to technologies that create decentralized identity systems with the aim of improving financial and social inclusion of vulnerable populations[18].

Kleros is a decentralized protocol built on top of Ethereum providing general mechanisms for decentralized decisions[19]. They provide a protocol that aims to solve Proof-Of-Humanity[20], which combines the concepts of web of trust[21], reverse turing tests to distinguish real humans from sybils and decentralized dispute resolution.

Many creative solutions exist to solve PoP. For example one protocol that also carries the name Proof-Of-Humanity, gives users voting power based on the amount of money they donate to non-profit organizations[22].

3.2 UBI Tokens

Building on PoP protocols, a range of cryptocurrencies emerged that give a UBI to their users. Based on the Proof-of-Humanity protocol mentioned in

Section 3.1, Kleros built a contract on the Ethereum blockchain that mints a universal basic income to its users[20].

Other UBI tokens are MYUBI[23], where it is not clear how it solves the problem of sybil attack in a decentralized way, or UBIC[24], which relies on modern passport features, which makes the system centralized.

Even though it is not aiming at providing a UBI to its users, also worth mentioning is Freicoin[25], a cryptocurreny that uses Proof-Of-Work but implements similar demurrage mechanisms as Encointer.

3.3 Delauney Triangulation Approach for Location Validation

There have been prior attempts at Encointer to solve the scalability issue of the location validation algorithm. Originally the idea was to store a global Delauney triangulation[26] along with all the registered locations in order to be able to later retrieve only the closest neighbors of a given location[27]. We investigated this idea at the beginning of this thesis and discontinued it in favor of the geohashing solution because efficiently inserting nodes into and removing nodes from a global Delauney triangulation in non-trivial.

Chapter 4

Verification of Geolocations

In this chapter we describe the first scalability problem of Encounter, present a solution and evaluate it. As mentioned in Section 2.2, all meetup locations need to maintain a minimum distance to each other. This problem is concerned with the validation of locations in terms of this minimum distance.

4.1 The Problem

In the current implementation of Encounter, when a community is bootstrapped, all locations for that community have to be provided and are validated at once using a naive approach that compares each location to every other location. Algorithm 1 shows a pseudocode implementation of this algorithm for the case when a community with just one location is added to the system. As we can see, this algorithm has complexity $O(n)$ for n locations that are already registered. All the locations are stored in a map that maps each community identifier (cid) to a vector of locations. So for each cid there needs to be a database lookup which means that the number of database lookups is linear in the number of registered communities.
Algorithm 1 is a slight simplification of the real algorithm, as it is possible to add communities with more than one location. In this scenario, all the added locations are compared to each other and then Algorithm 1 is applied to all of them, which results in a complexity of $O(m^2 + m * n)$ for m newly added locations and n already registered locations. We make this simplification here as the algorithm will be better comparable to our proposed solution, where locations will be added one by one. The goal will be to reduce to complexity of Algorithm 1 from $O(n)$ to $O(1)$.

Algorithm 1 Naive Algorithm for Location Validation

Storage:
$cids := cid[]$
$locations := map\ cid \rightarrow location[]$

Input: Location to be validated
Output: Boolean indicating if the input location has a distance larger than THRESHOLD to every already registered location

```
1:  function validateLocation(location)
2:      for cid in cids do
3:          locations ← locations[cid]
4:          for l in locations do
5:              d ← solarDistance(l, location)
6:              if d < THRESHOLD then
7:                  return false
8:              end if
9:          end for
10:     end for
11:     return true
12: end function
```

4.2 Improved Algorithm

As mentioned in Section 4.1, there were two main improvements necessary for the location validation algorithm. The first was to allow for incremental community building, which means that locations do not all have to be specified when a community is bootstrapped, but more locations can be added later in separate transactions. Also we implemented a feature that allows locations to be removed from a community, which might be necessary if a location becomes unreachable for some reason. The second improvement concerns the actual asymptotic complexity of the algorithm. Algorithm 2 shows a pseudocode implementation of our improved version.
The central element of the new algorithm is geohashing, where a location is mapped to a rectangular area called bucket. The idea is to narrow down the number of comparisons by only comparing the locations within a certain "radius" of the input location P. The naive approach would be to simply compare P only to locations within the same geohash bucket. This is too simple as locations of the neighboring buckets could also be closer to P than the minimum distance. To improve this situation, we could compare P to all locations inside its geohash bucket and to all locations from neighboring buckets. This approach would be safe as we chose a geohash length of 5, which means that a geohash bucket

has a size of 4.9 km x 4.9 km at the equator, which is large enough to cover all locations within the minimum distance. Note that geohash buckets decrease in width with increasing/decreasing latitude. We refer to Subsection 4.2.1 for detailed explanations why it is always safe to only consider the direct neighbors of the bucket containing P.
Still there is an optimization than can be made: Instead of comparing locations of all neighbor buckets, it suffices to only consider buckets that are closer to P than the minimum distance. As you can see in Figure 4.1, we can compute the direct distance from P to every neighbor bucket and only consider the buckets that have a distance to P smaller than the specified threshold.

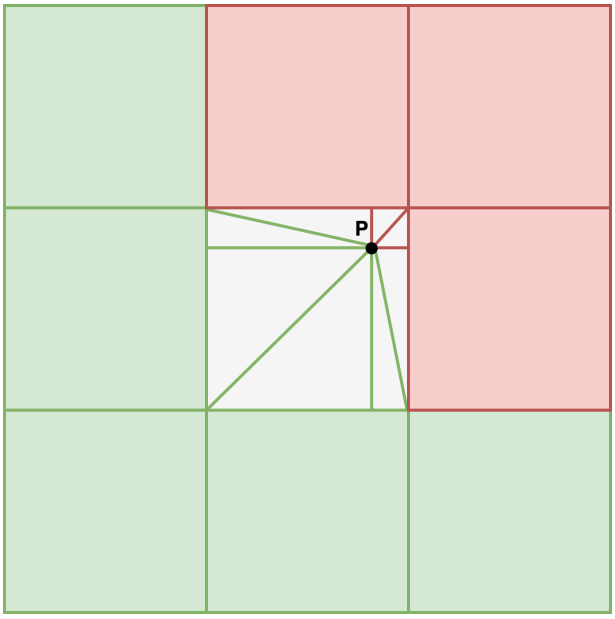


Figure 4.1: Example of neighboring geohash buckets

As the use of geohashes demands for a different data model, we had to adjust the data structures to store the locations. We came up with the following two main data structures:

1. **locations**
   This is the main data structure. It is a double map mapping from community identifier to geohash to location. It simply adds a layer for the geohash concept between cid and location.

2. **cidsByGeohash**
   When only considering the locations data structure from above, one can note that it is not possible to efficiently retrieve all locations for one given geohash bucket without looping over all community identifiers. This is why a second data structure was introduced, which maps from geohash

to community identifier and acts as an index to the locations double map described above.

Algorithm 2 Improved Algorithm for Location Validation

Storage:
$cidsByGeohash := map\ geohash \rightarrow cid$
$locations := map\ cid \rightarrow geohash \rightarrow location[]$

Input: Location to be validated
Output: Boolean indicating if the input location has a distance larger than THRESHOLD to every already registered location

```
1:  function validateLocation(location)
2:      Compute geohash of location
3:      Find position P of location inside geohash bucket
4:      Compute distance of P to edges and corners of geohash bucket
5:      relevantBuckets ← all nearby neighbour buckets
6:      for bucket in relevantBuckets do
7:          cids ← cidsByGeohash[bucket]
8:          for cid in cids do
9:              locations ← locations[cid][bucket]
10:             for l in locations do
11:                 d ← solarDistance(l, location)
12:                 if d < THRESHOLD then
13:                     return false
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return true
19: end function
```

In order to convince the reader that Algorithm 2 has complexity $O(1)$, we quickly go over the 3 loops and describe why each of them iterates only a constant number of times.

1. **Line 6**
   We loop over all the relevant neighbor buckets, plus the bucket in the center. When considering all the neighbors, the maximum number of iterations for this loop is 9.

2. **Line 8**
   This loop iterates over all communities that are contained in a certain geohash bucket. As communities will usually be larger than one geohash

bucket (4.9 km x 4.9 km), we expect this loop to do only one iteration. In certain cases there might be 2 or 3 iterations for geohash buckets that are shared by neighboring communities.

3. **Line 10**
   Here we loop over all locations in one geohash bucket. As the locations need to maintain a minimum distance to each other, there can only be a constant number of locations in one geohash bucket. Assuming a minimum solar trip time of 1s between locations, which corresponds to roughly 850 meters at the equator, one geohash bucket could contain a maximum of $(4900/850)^2 \approx 33$ locations.

4.2.1 Excluded Zones

As the ceremonies take place at high sun, there are some problematic areas that would allow for attacks if not treated accordingly. Those are areas close to the dateline (longitude +/- 180°) and areas close to the poles. In this section we provide calculations that form the basis for excluding certain areas of the world from the system in order to make it safe.

Poles

Two things need to be considered near the poles. Firstly, the sun's relative speed to the earth decreases with increasing/decreasing latitude, which means that it becomes easier for a human to travel from one meetup location to another between the two corresponding meetup times. The seconds point is that geohash buckets decrease in width with increasing/decreasing latitude.

In the following, we compute bounds on latitude that account for the two problems mentioned above. We start by deriving the latitude above which the sun's relative speed is faster than 83 m/s, which is Encointer's assumed maximum speed of an attacker.

The earth rotates once per day, where one rotation corresponds to 360°. From this we get that the sun moves relative to the earth at a pace of

$$\frac{3600 * 24s}{360^\circ} = 240s/^\circ \tag{4.1}$$

This means that it takes the sun 240 seconds to traverse one degree of longitude at latitude x.

The distance in meters of one degree of longitude at latitude x is

$$cos(x^\circ) * 111319 \tag{4.2}$$

(one degree of longitude corresponds to 111319m at the equator)

Assuming a speed of 83 m/s, a human can travel with a pace of

$$\frac{cos(x^\circ) * 111319}{83} \tag{4.3}$$

at latitude x.

Combining 4.1 and 4.3 we get:

$$\frac{cos(x^\circ) * 111319}{83} = 240 \tag{4.4}$$

Solving for x, we get x = 79.6917°. Thus, above a latitude of 79.6917°or below -79.6917°, a human can travel faster than the sun when traveling at constant latitude.

We continue by computing a bound on latitude below which it takes a human more than 1 second of solar trip time (see Definition 2.1) to traverse one geohash bucket.
We make the following assumptions:
Minimum solar time between two locations = 1 s
Maximum human speed = 83 m/s
Geohash length = 5 (Bucket size is 4.9 km * 4.9 km at equator)

Using 4.2 and 4.1, we calculate the sun's relative speed to the earth at latitude x:

$$\frac{cos(x^\circ) * 111319}{240} m/s \tag{4.5}$$

And following from this, we get the time it takes for the sun to traverse one geohash bucket:

$$\frac{4900 * cos(x^\circ)}{\frac{cos(x^\circ) * 111319}{240}} \tag{4.6}$$

Similarly, a human will need

$$\frac{4900 * cos(x^\circ)}{83} s \tag{4.7}$$

to traverse one geohash bucket.

Now, combining 4.7, 4.6 and the assumption that two locations need to apart at least 1s in terms of solar trip time, we get the follwoing equation:

$$\frac{4900 * cos(x^\circ)}{83} - \frac{4900 * cos(x^\circ)}{\frac{cos(x^\circ) * 111319}{240}} = 1 \tag{4.8}$$

Solving for x, we get x = 78.7036°. So below 78.7036°or above -78.7036°it takes more than 1 second solar time to traverse one geohash bucket horizontally. Thus it is safe to only consider the direct neighbors of the the geohash of the location being validated when computing all locations that might be in conflict.

Using the bounds calculated above, we decided to exclude all locations with an absolute latitude of more than 78.7°in order to simplify the algorithm. We can accept this simplification, as the northern most settlement with more than 1000 inhabitants is at 78.2°[28] and the southern most settlement is at -54.93°[29].

Dateline

Similar considerations have to be made for locations close to the dateline. A possible attack scenario at the dateline would be an attacker moving in the opposite direction of the sun and arriving early enough at a second meetup location. Like this, she could attend at two meetups in one ceremony phase. Figure 4.2 shows an illustration of this attack.

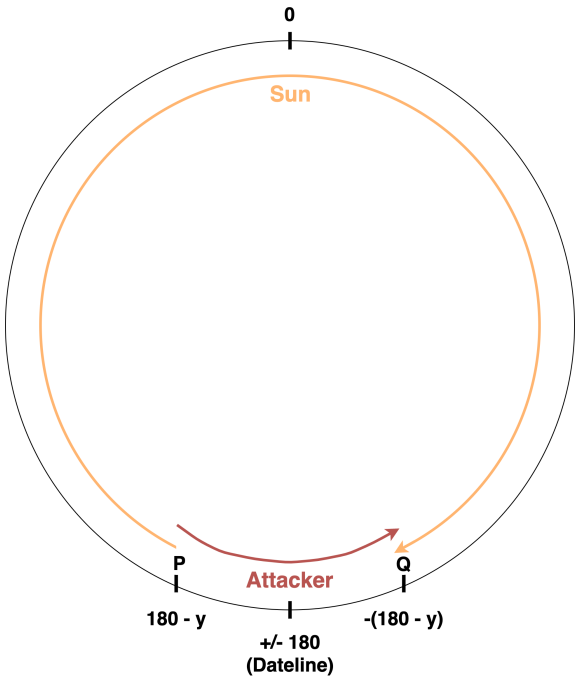


Figure 4.2: Illustration of the attack scenario at the dateline

In the following, we derive a formula to exclude certain areas close to the dateline:

The time difference in seconds between high sun at P and high sun at Q using the sun's relative pace from 4.1 is:

$$360^\circ - 2 * y^\circ * 240s \tag{4.9}$$

The distance in meters from P to Q counterclockwise at latitude x can be calculated using 4.2:

$$2 * y * cos(x^\circ) * 111319 \tag{4.10}$$

So, we get the time in seconds it takes a human with a speed of 83 m/s to travel from P to Q in the opposite direction:

$$\frac{2 * y^\circ * cos(x^\circ) * 111319}{83} \tag{4.11}$$

This leads us to the following equation:

$$\frac{2 * y^\circ * cos(x^\circ) * 111319}{83} = 360^\circ - 2 * y^\circ * 240 \tag{4.12}$$

If we solve for y we get a formula for the number of degrees that need to be excluded at each side of the dateline depending on the latitude x, because for a distance larger than 2*y a human cannot travel from P to Q faster than the sun in the opposite direction. Solving the equation, we get:

$$y = \frac{3585600}{19920 + 111319 * cos(\frac{\pi * x^\circ}{180})} \tag{4.13}$$

A visualization of this equation can be found in Figure 4.3 a). It can be seen that with this solution, one would have to exclude a very big area which is not desirable. Following attempts could be made to mitigate this issue:

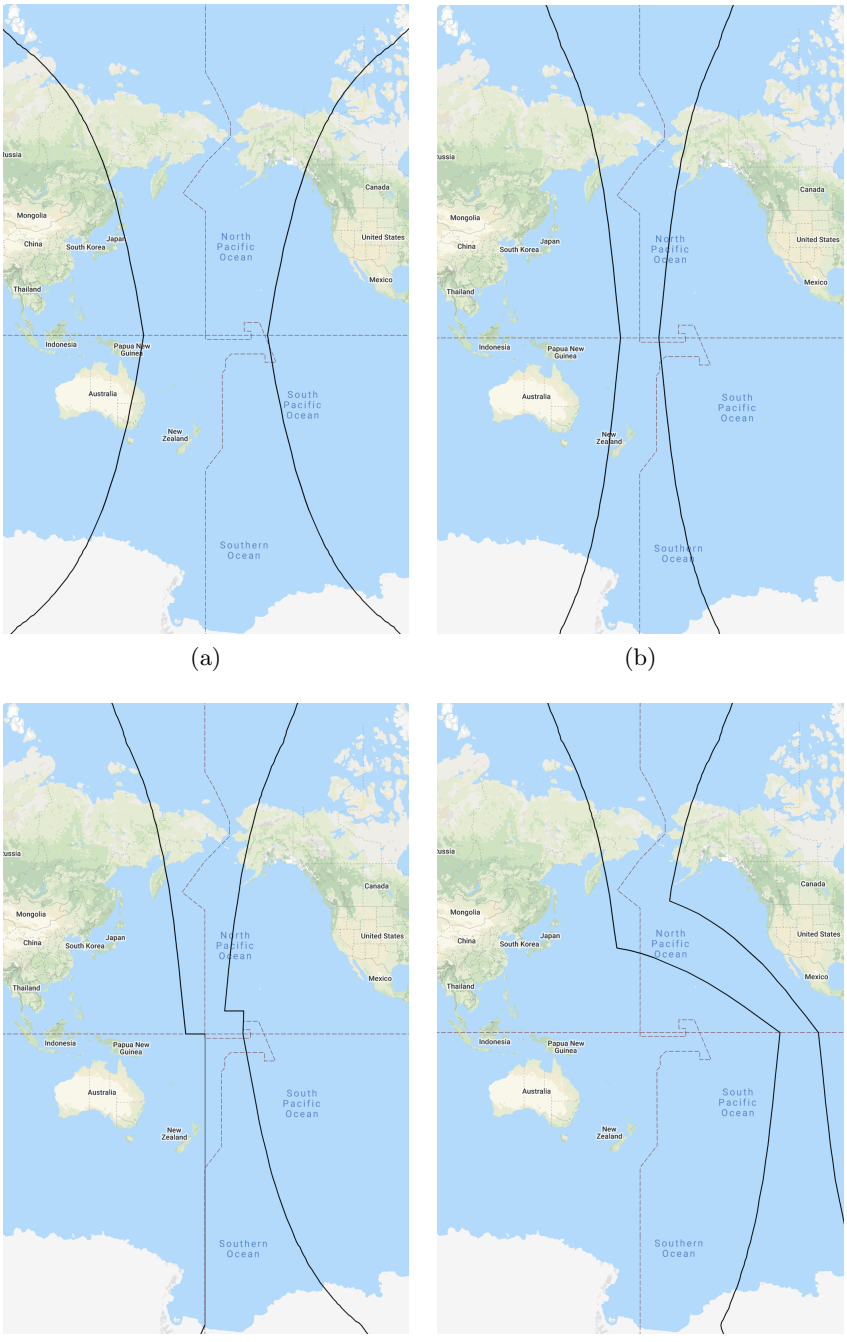
1. **Relax assumptions of maximum human speed**
   For now, Encointer assumes a maximum human speed of 83 m/s or 300 km/h. This is an extreme assumption, because it would require a helicopter or similar to conduct a successful attack. Figure 4.3 b) shows a visualization of the excluded zone assuming a maximum human speed of 23 m/s which corresponds to approximately 82 km/h.

2. **Custom exclusion zones**
   Figure 4.3 c) shows how an excluded zone could look like if the excluded zone would not be spread evenly on both sides of the dateline. This could be beneficial in order to not have to exclude large areas like New Zealand.

3. **Excluded zone not including the dateline**
   In order to maximize the number of human settlements included in the system, one could create a safety zone at a different longitude than the

dateline. This would require the ceremony phase to start at the moment of high sun at the left border of the excluded zone and end at high sun at the right border. This would also lead to ceremonies not all taking place on the same date. Figure 4.3 d) shows an example of this.

In consultation with the Encointer engineers we decided to implement neither of those ideas and just exclude a constant 1000km on each side of the dateline for reasons of simplicity. Like this we accept some theoretical attacks, which are infeasible and thus unlikely. Our above calculations form a basis for possible future changes to the protocol, which would only take a few lines of code to be implemented into the system.



(a) (b)

(c) (d)

Figure 4.3:
(a) Linearly excluded zone assuming human speed of 83 m/s
(b) Linearly excluded zone assuming human speed of 23 m/s
(c) Custom excluded zone assuming human speed of 23 m/s
(d) Custom excluded zone assuming human speed of 23 m/s

Source: https://www.google.com/maps, lines drawn using pyKML library[30]

4.3 Implementation

The implementation of the algorithm was done using the Rust programming language and the Substrate framework. It was built on top of the existing Encointer codebase. For computing geohashes we forked an existing library for Rust[31] and adopted it to use fixed-point instead of floating-point numbers and got rid of all features from the `std` library. Both changes were requirements by Encointer and Substrate, respectively.
All newly implemented features were unit tested.

4.4 Evaluation

In this section we present the evaluation of the performance difference between Algorithm 1 and Algorithm 2. In order to measure the runtime of each algorithm, the Substrate benchmarking framework was used. The experimental setup was as follows: A number of locations are added into a newly created Encointer environment before the experiment and then we measure how long it takes to add one new location to the system. Two different experiments were conducted concerning the pre-registered locations. In the first setup, all locations were registered in the same community and in the second, locations were registered in communities of size 10000. The number of pre-registered locations ranged from 2 to 500000, measurements were taken in steps of 50000 locations. We measured each datapoint four times and took the median as the final result. Test locations were generated using Algorithm 3 which evenly spreads latitudes between -40°and 40°and longitudes between -140°and 140°. Info Box 4.1 shows an overview of the benchmarking environment.

MacBook Pro (15-inch, 2019)
OS: macOS Catalina, Version 10.15.7
CPU: 2.3 GHz 8-Core Intel Core i9
Memory: 16 GB 2400 MHz DDR4

Info Box 4.1: Specification of the benchmarking environment

Algorithm 3 Algorithm to get the ith test location

1: $lat \leftarrow (i/1000) * 0.08 - 40.0$
2: $lon \leftarrow (i\%1000) * 0.28 - 140$
3: **return** (lat,lon)

Below we will discuss the two experiments mentioned above in more detail.

4.4.1 All locations in one community

In the first experiment, we registered a number of locations in one community and measured how long it takes to validate and add one more location to the system. Figure 4.4 a) shows that the initial algorithm has an asymptotic complexity of $O(n)$ while our improved algorithm has constant complexity $O(1)$. Figure 4.4 b) shows only the runtime of the new algorithm. In Figure 4.4 c) we can see that both algorithms have a constant number of database reads. The spike in the line of the new algorithm is random and has to do with the random nature of the locations. In this case the location is most probably close to another geohash bucket and therefore one more database read for that corresponding bucket has to be made. For more details on this, please refer to Figure 4.1. The slight increase in runtime between the first and the second datapoints we attribute to the substrate database and caching mechanisms.

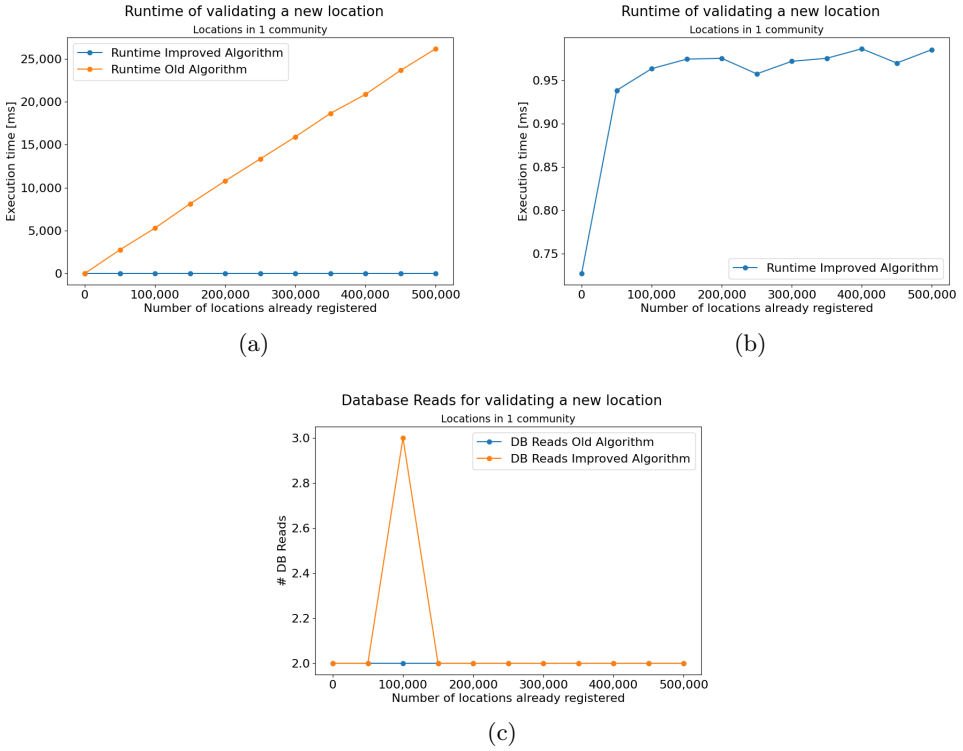


Figure 4.4: Results of the experiment with pre-registered locations in one community

4.4.2 Locations in communities of size 10000

In this section we show the results of the experiment where the pre-registered locations were grouped in communities of size 10000. As in the previous experiment, we can see in Figure 4.5 a) and b) that the improved algorithm has constant complexity. What we can also see in Figure 4.5 c) is that for the old algorithm the number of database reads grows in the number of pre-registered communities, because a database lookup has to be made for each community. Our improved algorithm also mitigates this issue by only considering nearby locations and has a constant number of database lookups. The bump in the orange line is similar to the one in Figure 4.4 c).

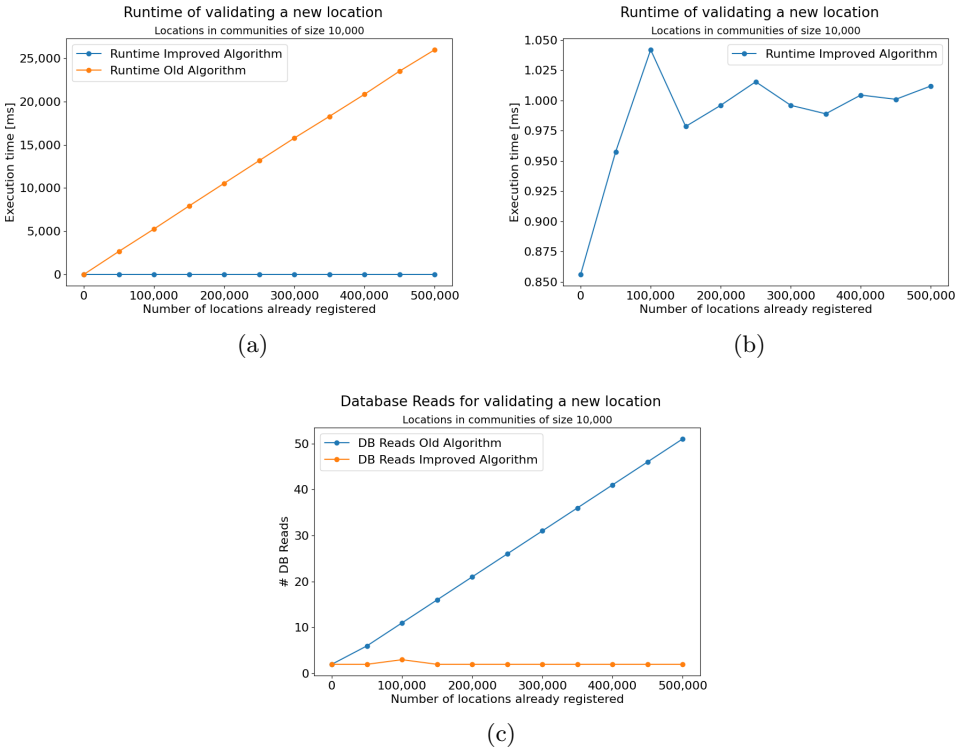


Figure 4.5: Results of the experiment with pre-registered locations in communities of size 10000

4.5 Future Work

The only aspect of this problem that is not fully implemented yet is the issue at the dateline. In Section 4.2.1 we provided several solutions that solve this problem.

Chapter 5

Assigning Participants to Meetup Locations

In this chapter we discuss the scalability problems that Encointer faces in the process of assigning users to meetup locations on the blockchain, present a solution and evaluate it.

5.1 The Problem

After users registered for the meetup phase, they have to be randomly assigned to meetup locations in order to prevent collusion between meetup participants. In addition to the unpredictability of the assignments, there are the following constraints:

1. There are 4 categories of users in Encointer. Each group should be distributed equally among all meetups:
   - **Bootstrappers** are founding members of the community
   - **Reputables** are users that successfully attended a meetup in the past and therefore have reputation
   - **Endorsees** are users that were labeled trustworthy by a bootstrapper
   - **Newbies** have never attended a meetup before
2. There should be a minimum of 3 and a maximum of 12 participants per meetup.
3. No meetups should exist without at least one community bootstrapper or user with reputation.
4. The percentage of users without reputation should be less than 25% for every meetup. We will refer to this percentage as newbie ratio.

Encointer uses a straight-forward approach to calculate random permutations and store them on-chain, the core functionality of which we will describe in the following. The algorithm gets as input a list for each category of users (bootstrappers, reputables, endorsees and newbies) and calculates the permutation as follows:

1. Calculate the number of meetups that can take place based on available locations and number of registered bootstrappers and reputables.
2. Calculate the number of newbies that can participate in order to ensure the newbie ratio of 25%.
3. Shuffle each list of users separately using a random permutation.
4. Concatenate the shuffled lists.
5. Loop over the concatenated list and distribute the users to the locations in a round robin fashion.
6. Store the result in the blockchain storage.

Those steps are repeated for every community. While this approach is designed to fulfill all the constraints described above, the problem is that calculating permutations for every category of users in every registered community is very time consuming and does not scale if there are many communities registered in the system. Also storing all the permutations on-chain is inefficient. In the following section we will describe a new approach to solve the assignment problem, which saves time and storage space at the cost of weakening some of the constraints to a certain extent.

5.2 Proposed Solution

In the following we describe our proposed solution for assigning users to meetup locations. Basic knowledge of modular arithmetic is required to understand this section.

5.2.1 Basic Concepts

The basic idea for our solution is to avoid having to compute and store all the permutations on-chain by making the assignment implicit using some publicly known pre-defined function that takes as input a random seed. In the blockchain transaction, we only have to determine the seed and then every application can calculate the assignment off-chain given the permutation function. While in the ceremony phase we need to be able to calculate for a given user her meetup

location, we also have to be able to find all the users for a given meetup location later in the verification phase. This means that our assignment function has to be invertible.
We designed the assignment function with the help of modular arithmetic and additive cyclic groups. Let's consider a simplified version of the assignment problem, where we have to distribute a number of users N to a number of locations n. The users and locations are identified by their index in $[0, N)$ and $[0, n)$ respectively. We know that a group is cyclic under addition *mod* N and that any group element coprime to N is a generator of this group. So in order to shuffle the users in an unpredictable way, we choose a random prime number $s1$ which is by definition coprime to N (and if $s1 > N$, then $s1 \mod N$ is coprime to N, see Subsection 2.4.1) and therefore a generator of the group and calculate for every user $u \in [0, N)$ a new index

$$u * s1 \mod N \tag{5.1}$$

which will map every $u \in [0, N)$ to another number in $[0, N)$.
Now, in order to distribute the users to the meetup locations we simply calculate

$$(u * s1 \mod N) \mod n \tag{5.2}$$

which will map each user to a location in $[0, n)$.

In the following sections we will discuss problems of this approach in regard with our constraints described in Section 5.1 and do a step-by-step refinement of the formula and the process of defining $s1$ and N.

5.2.2 Problem 1: Index Zero

One problem with Equation 5.2 is that the user with index 0 will always be mapped to location 0, as can be easily verified. To mitigate this problem, we introduce a second prime number $s2$ to act as a random offset. Note that $s2$ does not necessarily need to be prime to fulfill this purpose. This gives us the following formula:

$$(u * s1 + s2 \mod N) \mod n \tag{5.3}$$

The offset $s2$ does not change anything about the fact that every $u \in [0, N)$ is mapped to another number in $[0, N)$, because we are in a cyclic group.
Separate instances of this formula will be used later in order to distribute bootstrappers, reputables, endorsees and newbies equally to the meetup locations.

Now that we have determined our core formula to map users to meetup locations, we can give its counter-part, namely the formula to determine all users given a certain meetup location.
Given a meetup location l, we know from Equation 5.9 that

$$u * s1 + s2 \mod N = n * i + l \tag{5.4}$$

for some i.
Rearranging that formula we get:

$$u = (n * i + l - s2) * s1^{-1} \mod N \tag{5.5}$$

We also know that

$$n * i + l < N \tag{5.6}$$

because we only consider user indices smaller than N. We can rearrange this formula to get:

$$i < \frac{N - l}{n} \tag{5.7}$$

So this leads us to the final formula to determine the set of users U for a given meetup location l:

$$U = (n * i + l - s2) * s1^{-1} \mod N \ \forall i < \frac{N - l}{n} \tag{5.8}$$

5.2.3 Problem 2: Predictability

One requirement for the assignment process is that it is not possible for a user to guess at what location she will have to attend the meetup in order to prevent collusion. When taking a closer look at Equation 5.9 we can see the following problem: Let's consider a setup where there are n meetup locations and N participants and N is divisible by n. The equation now boils down to

$$u * s1 + s2 \mod n \tag{5.9}$$

Note that the $\mod N$ part cancels out because $N \mod n = 0$. We can now see that there will be predictable patterns in the assignment regardless of $s1$ and $s2$. We claim that users with indices $x + i * n$ (users that have indices separated by steps of size n) will end up in the same meetup location $x * s1 + s2 \mod n$ in this scenario. We provide a simple proof using the rules for modular addition and multiplication.

Proof.

$$
\begin{aligned}
&(x + i * n) * s1 + s2 \mod n \\
&= ((x + i * n) * s1) \mod n + s2 \mod n \\
&= ((x + i * n) \mod n * s1 \mod n) \mod n + s2 \mod n \\
&= ((x \mod n + i * n \mod n) \mod n * s1 \mod n) \mod n + s2 \mod n \\
&= ((x \mod n + 0) \mod n * s1 \mod n) \mod n + s2 \mod n \\
&= (x \mod n * s1 \mod n) \mod n + s2 \mod n \\
&= (x * s1) \mod n + s2 \mod n \\
&= x * s1 + s2 \mod n
\end{aligned}
$$

□

As users receive their index based on the order of registration, there is a potential attack where an attacker tries to register various accounts in a way that they always have the same gap n between each other and that the final number of registrations is a number N, which is a multiple of n. Like this the attacker finds all her sybils in the same meetup and can attest the personhood of accounts that do not have a real person behind them.
To mitigate this problem, it suffices to not use the number of registered users as N, but a prime number close to N, because in this way N can never be a multiple of n. Like this, it will not be possible for a user to find out to which meetup location she will be assigned before knowing $s1$ and $s2$. There are now two cases that we need to consider that will have different effects on the outcome of the assignment: Choosing the prime number above N or the prime number below N. We will quickly discuss both cases below.
We denote the new modulus (prime number above N or prime number below N) as M.

Prime number above N When using the prime above N as the modulus, this means that every user $u \in [0, N)$ will be assigned a new index in $[0, M)$, which will then be mapped to a meetup location in $\in [0, n)$, as in the following equation:

$$f(u) = (u * s1 + s2 \mod M) \mod n \tag{5.10}$$

We can now see that there are potential configurations where some meetup locations do not receive any users, even though there would be enough users. Let's consider a simple example: We want to distribute $N = 10$ users to $n = 10$ locations. When choosing M as the prime number above N, we get $M = 13$. We choose $s1 = 19$ and $s2 = 29$. Using Equation 5.10 we get the following assignment:

user u	location f(u)
0	3
1	9
2	2
3	8
4	1
5	7
6	0
7	6
8	2
9	5

Table 5.1: Mapping of users to meetup locations

In Table 5.1 we find that no user is mapped to location 4 but two users are mapped to location 2. The problem can be seen easily when looking closer at users 2 and 8:

$$\begin{aligned} & f(2) \\ &= (2 * 19 + 29 \mod 13) \mod 10 \\ &= 2 \mod 10 \\ &= 2 \\ &= 12 \mod 10 \\ &= (8 * 19 + 29 \mod 13) \mod 10 \\ &= f(8) \end{aligned}$$

The problem here is that the user with index 8 maps to 12, which lies in the gap between $N = n$ and M and can therefore be mapped to an equal location than another user (in this case the user with index 2).
This issue can become problematic for our case, because we want to ensure that every meetup location gets at least one bootstrapper or reputable in order to strengthen the security of the system. If we were to find a configuration as in Table 5.1 when distributing these users over the locations, this could result in locations without any bootstrapper or reputable. This is why we will now consider choosing M as the prime number below N.

Prime number below N When taking M as the prime number below N, we do not have the issues mentioned above. But there is still one adjustment to be made. When considering the example above, we note that taking $M = 7$ would also result in problems, because now $M < n$. We can see that

$$(u * 19 + 29 \mod 7) \mod 10 < 7 \ \forall u \in \mathbb{N} \tag{5.11}$$

which means that locations 8 and 9 would be assigned no users at all. This is why we add another constraint to the system that the number of meetups has to

be smaller or equal than the prime number below Q, where Q is the number of bootstrappers and reputables registered. Like this we can guarantee that every meetup location will be assigned at least one bootstrapper or reputable (note that for newbies and endorsees, we do not have a similar requirement and can accept some locations without users of those categories). In the above example this would mean that we would choose $n = 7$ instead of $n = 10$, which would result in the following assignment:

user u	location f(u)
0	1
1	6
2	4
3	2
4	0
5	5
6	3
7	1
8	6
9	4

Table 5.2: Optimized mapping of users to meetup locations

In Table 5.2, we can now see that choosing M as the prime number below N and the additional constraint, every location in $[0, 7)$ is assigned at least one user.
We can note here that it is no longer necessary for $s1$ to be a prime number, because when we choose the modulus M as a prime number, any non-zero group element will be a generator of our group. So it suffices to choose a random $s1 \in [1, M)$.

5.2.4 Problem 3: Unequal Meetup Sizes

We have now developed our assignment formula and the additional constraint that the first modulus should be the prime number below N. There is one last issue remaining with this setup. Recall that our assignment function maps all N users to a new index in $[0, M)$, where M is the prime number below N and then maps the new index to a location in $[0, n)$ using the modulo function as in Equation 5.10. Now there can be configurations of $s1$ and $s2$, where most of the user indices in $[M, N)$ map to the same location, which causes one location to have over-proportionally many users assigned to it. Consider the following example, where $s1 = 2326$, $s2 = 1099$, $n = 427$, $N = 2761$ and $M = 2753$. The assignment for user indices looks as follows:

user u	location f(u)
0	245
1	245
2	245
3	9
4	9
5	9
6	9
7	9
8	9
9	9
10	200
11	200
...	...
M = 2753	245
2754	245
2755	245
2756	9
2757	9
2758	9
2759	9
2760	9

Table 5.3: Problematic mapping of users to meetup locations when using the prime below N as the modulus

We can clearly see at index $M = 2753$ the pattern repeats, which is to be expected, because M is our modulus. With this configuration, meetup location 9 will be assigned too many participants compared with the other locations and we want to avoid that. To mitigate this we introduce a check when choosing $s1$ and $s2$ that makes sure that no location is assigned more than $\lceil \frac{N-M}{n} \rceil$ users with index $u \in [M, N)$, which will make sure that the users are distributed more equally to the meetup locations. We will refer to this as checking the equal meetup size property. For a more formal reasoning about this constraint and its effects please refer to Subsection 5.4.2.

5.2.5 The Algorithm

We now have all the tools to develop the final algorithm for determining all the necessary random numbers and parameters for the assignment function. As mentioned above, the idea is to use separate instances of the formula to assign boostrappers, reputables, endorsees and newbies such that those groups are dis-

tributed equally to all the meetup locations. Bootstrappers and reputables are distributed together, because they are the trustworthy users of the system and we want to make sure that every meetup gets at least one of them as explained in Subsection 5.2.3.

Algorithm 4 Algorithm for generating parameters for assignment function

Input: Number of boostrappers, reputables, endorsees and newbies
Number of locations

Output: Parameters for the assignment functions:
Number of meetups n
Number of allowed reputables
Number of allowed endorsees
Number of allowed newbies
s1, s2, and M for bootstrappers and reputables (br)
s1, s2, and M for endorsees (end)
s1, s2, and M for newbies (new)

```
1: function generateAssignmentParams
2:     MEETUP_MULTIPLIER = 10
3:     numMeetups = min(numLocations, primeBelow(numBootstrappers + numReputables))
4:     availableSlots = numMeetups * MEETUP_MULTIPLIER
5:     availableSlots -= numBootstrappers
6:     numAllowedReputables = min(numReputables, availableSlots)
7:     availableSlots -= numReputables
8:     numAllowedEndorsees = min(numEndorsees, availableSlots)
9:     availableSlots -= numAllowedReputables
10:    maxAllowedNewbies = (numBootstrappers + numAllowedReputables + numAllowedEndorsees) / 3
11:    numAllowedNewbies = min(numNewbies, availableSlots)
12:    numAllowedNewbies = min(numAllowedNewbies, maxAllowedNewbies)
13:    availableSlots -= numAllowedNewbies
14:    numParticipants = numBootstrappers + numAllowedReputables + numAllowedEndorsees + numAllowedNewbies
15:    n = ceil(numParticipants / MEETUP_MULTIPLIER)
16:
17:    M_br = primeBelow(numBootstrappers + numAllowedReputables)
18:    M_end = primeBelow(numAllowedEndorsees)
19:    M_new = primeBelow(numAllowedNewbies)
20:
21:    for gorup in br, end, new do
22:        while equal meetup size property is not fulfilled do
23:            s1 = randomGroupElement
24:            s2 = randomGroupElement
25:        end while
26:    end for
27:    return all parameters
28: end function
```

Algorithm 4 shows a pseudocode implementation of the parameter generation algorithm for the assignment functions. Lines 2 to 15 calculate how many meetup participants there can be based on the number of available locations, the number of allowed bootstrappers and reputables (recall that every location needs to have at least one bootstrapper or reputable assigned) and the number of newbies that want to participate (recall that there should not be more than 25% newbies per meetup). Note that the parameter MEETUP_MULTIPLIER indicates the target number of participants per meetup. Lines 17 to 19 then find the prime numbers below the number of participants in each category of users as discussed in Subsection 5.2.3 and lines 21 to 25 find random group elements that ensure the equal meetup size property discussed in Subsection 5.2.4. For a detailed analysis of the algorithm and the properties of the generated meetups please refer to Section 5.4.

5.3 Implementation

The implementation of this problem was conducted in two steps. First, we implemented a prototype of the algorithm in Python. Along with that prototype, we developed a benchmarking framework which allowed us to run experiments in order to check the desired properties of the meetup assignments. We also setup a cloud infrastructure in order to run the experiments remotely as those experiments usually ran for multiple hours even though we parallelized independent experiment runs in order to minimize runtime.
After multiple iterations of developing and benchmarking, the algorithm was implemented into Encointer's codebase using the Rust programming language and the Substrate framework.

5.4 Evaluation

In this section we evaluate the runtime of the algorithm given in Subsection 5.2.5, as well as the properties of the meetups generated by our approach. Most analyses presented are based on two empiric experiments conducted:

1. **Experiment A: Small number of participants**
   In this experiment we simulated 1440000 meetup assignments, ie. running 1440000 instances of Algorithm 4, calculated for every meetup location its participants using Equation 5.9 and validated the correctness of the computation using Equation 5.8. As inputs, random configurations were used with $numReputables <= 12$, $numEndorsees <= numBootstrappers * 50$, $numReputables, numNewbies < 10000$ and $numLocations < 50000$. A total of 837273490 meetups were simulated.

2. **Experiment B: Large number of participants**
   As a second experiment we simulated 63888 meetup assignments with $numReputables <= 12$, $numEndorsees <= numBootstrappers * 50$, $10000 <= numReputables <= 200000$, $10000 <= numNewbies <= 100000$ and $10000 <= numLocations < 200000$. A total of 920412879 meetups were simulated.

5.4.1 Predictability

Before we analyze the algorithm runtime, we would like to discuss the unpredictability property of our algorithm. Recall that the previous naive implementation by Encointer calculated a random permutation of all the N users and we use the formula $u * s1 + s2 \mod N$ to get a similar result. When comparing the two approaches, we see that for the naive approach, there are $N!$ different permutations and for our approach there is only a subset thereof with $(N-2)^2$ possible permutations, because every combination of $s1$ and $s2$ gives a different permutation and $s1, s2 \in [1, N-1]$. Although our solution does not guarantee the same true randomness as the previous approach, we argue that it is still impractical enough for a user to guess her meetup location. To validate this claim, we describe and analyze a possible attack scenario in the following. Let's assume the attacker can guess $s1$. She can do so with a chance of $1/N$. Knowing s1 she can assume $s2 = 0$ and register her sybils with indices $i_0...i_s$ in a way that those all map to new indices $j_0...j_s$ that are congruent $\mod n$, which means that they will all map to the same location, if the assumption $s2 = 0$ holds. Even if this assumption does not hold, at least 50% of the sybils will be assigned to the same location, because $s2$ only is an offset. Assume $k_0...k_s$ is a sorted list of the indices $j_0...j_s$. So, if $k_0...k_s$ are are congruent $\mod n$ then $k_0 + s2...k_p + s2$ will be congruent $\mod n$, where k_p is the largest index such that $k_p + s2 < N$. The same holds for $k_{p+1} + s2...k_s + s2$, and in either of the two sets there are at least 50% of the indices.
To sum this up, from a theoretical perspective, an attacker has a $1/N$ chance that she can craft a meetup where she controls at least 50% of the participants. In practice there are many factors that make this attack more infeasible:

1. For the attack to be successful the attacker also has to know n and N. This is possible to a certain extent, because she can register more or less sybils which will lead to a different amount of participants and meetups. On the other hand, the control is limited because the amount of newbies that are allowed in the assignment process is limited.

2. Assuming that there are no endorsees and the amount of meetups is large enough such that the bootstrappers become negligible, the attacker still has to guess $s1$ for the assignment of reputables and $s1$ for the assignment of newbies correctly. Also because it is not sufficient that the attacker's

reputables are together in one meetup and the attacker's newbies are in another, the attacker also has to guess the offsets $s2$ for the assignment of reputables and newbies. This drastically decreases her chances of winning to $1/N^2_{newbies} * 1/N^2_{reputables}$ which is in the order of $1/N^4$. Note that there is another attack where the attacker floods the system with sybils to get a large probability that at least one of the sybils will be assigned to the same meetup where she also controls a majority of the reputables. This attack will be discussed in more detail in Section 5.5.

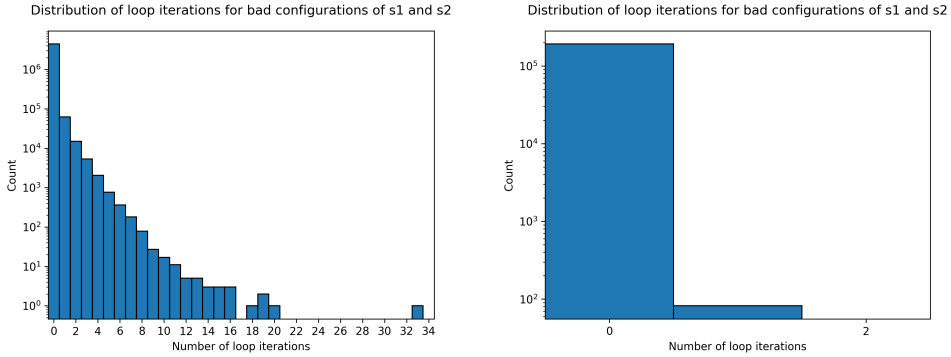
3. The scenario in item 2 only works if the attacker controls enough users that already have reputation. So in the best case scenario for the attacker, she would have to control 3 reputables for every sybil, because the ratio of newbies that are allowed in each meetup has to be below 25%.

4. It is not trivial for an attacker to register her sybils at the indices of her choice, because there are also other participants registering themselves and some other participant could take the attackers desired index.

Even though we have to suffer the loss of some randomness with our approach, we argue that the above points are still enough to guarantee the system's practical security.

5.4.2 Algorithm Runtime

As described in Section 5.1, the implementation by Encointer computed and stored random permutations directly on the blockchain. The optimal runtime for computing random permutations is $O(n)$ for n elements to be permuted [32], so the runtime of their algorithm is $O(N)$ with N being the number of users registered for a meetup ceremony. When looking at Algorithm 4, we can see that there are no loops except for the while-loop in line 22, which is used to determine bad configurations of $s1$ and $s2$ in order to make the equal meetup size property hold. So without this loop Algorithm 4 would clearly have a runtime of $O(1)$. In theory the loop could iterate infinitely, which is undesirable. This is why in practice one should implement a maximum amount of iterations, after which the loop should terminate even if the equal meetup size property is not fulfilled. This would mean that in the case of this premature loop termination, we would have to accept one or two meetups with significantly more participants than the others. In order to find out the number of loop iterations we should allow, we counted the number of loop iterations for all the simulated meetup assignments.



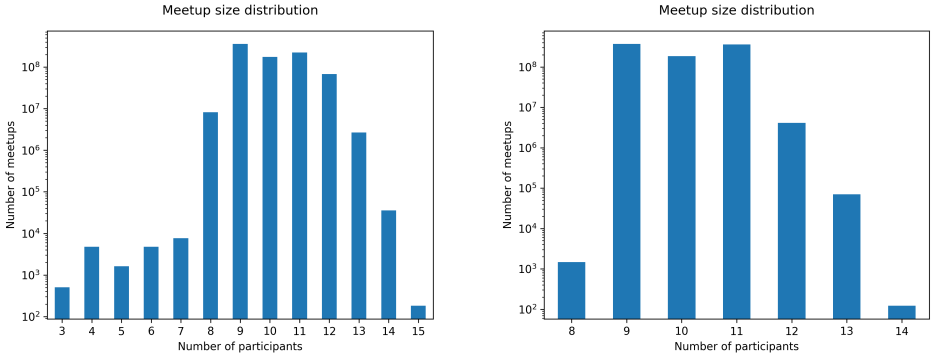
(a) Experiment A (b) Experiment B

Figure 5.1: Number of loop iterations for bad configurations of $s1$ and $s2$

Figure 5.1 shows that for Experiment A, where there were a total of $3 * 1440000 = 4320000$ instances of the loop, the maximum number of loop iterations was 33 and in Experiment B with $3 * 63888 = 191664$ instances, there was a maximum number of 1 loop iteration. Not the logarithmic scale on the y-axis of the plots.
Those experiments suggest that we could choose to cap the amount of loop iterations at for instance 100. We also know that the body of the loop has to check all numbers between a given N and the prime number below it (M). Assuming a maximum community size of $N = 10\,billion$ users, our analysis shows that the maximum gap between any two prime numbers between 0 and 10 billion is 210 (between 20831323 and 20831533). So we can clearly see that with capping the number of loop iterations, we can guarantee that our algorithm has runtime $O(1)$.

5.4.3 Meetup Size

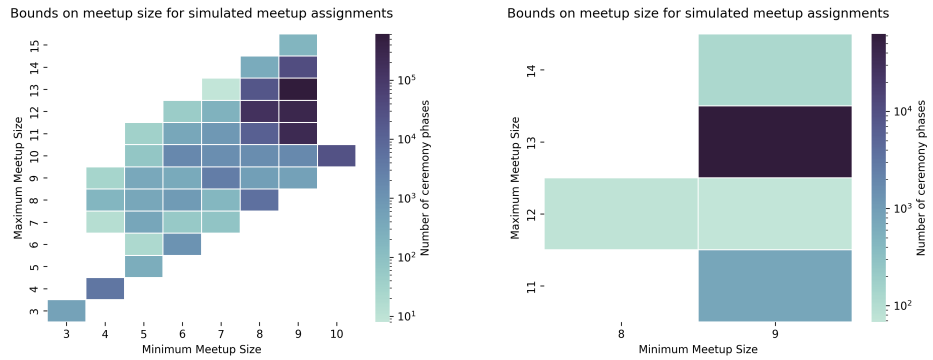
In this section we want to analyze the size of the meetups calculated with our method. Figure 5.2 shows that no meetups had more than 15 participants and that most meetups had a size distributed around 10, which is as expected, because we chose MEETUP_MULTIPLIER = 10.



(a) Experiment A (b) Experiment B

Figure 5.2: Distribution of meetup sizes

Figure 5.3 shows that for most simulated meetup assignments, the meetups had sizes between 9 and 13, which is desirable. The small meetup sizes (eg. meetups with only 3 participants) come from configurations where there are only a small amount of participants registered in the first place and no meetups of larger size were possible. Note that we do not see any small meetup sizes in experiment B. Also note the logarithmic scale of the y-axis in both figures.



(a) Experiment A (b) Experiment B

Figure 5.3: Distribution of bounds of meetupsizes

In addition to Figure 5.2, we would like to provide a mathematical reasoning why there cannot be more than $MEETUP\_MULTIPLIER + 6$ users in one meetup:
We are using 3 instances of Equation 5.10, one for bootstrappers and reputables, one for endorsees and one for newbies in order to distribute users to meetup locations. n is the number of meetup locations for all equations and N is the

prime number below $numBr = numBootstrappers + numReputables$, $numE = numEndorsees$ and $numN = numNewbies$ respectively. Taking the endorsees as an example, we see that there cannot be more than

$$(N \; div \; n + 1) + ((numE - N) \; div \; n + 1) \tag{5.12}$$

endorsees per meetup, where *div* denotes integer division. The first part of the equation comes from all the users with index $u < N$, because those indices are mapped bijectively to other indices $< N$. The second part of the equation comes from all the users with indices in $[N, numE)$ and this holds because of the equal meetup size property discussed in Subsection 5.2.4. Simplifying Equation 5.13, we get

$$numE \; div \; n + 2 \tag{5.13}$$

as an upper bound for endorsees per meetup. Similar bounds hold for bootstrappers/reputables and newbies.

Putting this together, we get an upper bound of users per meetup as follows:

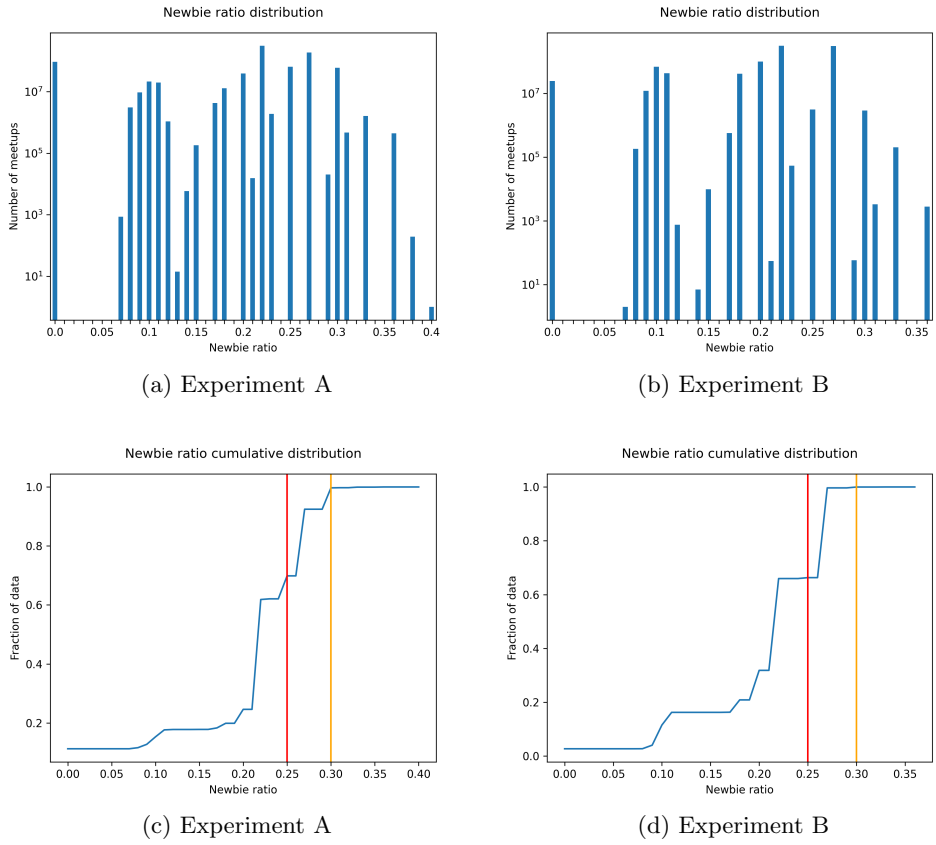
$$\begin{aligned}
&(numBR \; div \; n + 2) + (numE \; div \; n + 2) + (numN \; div \; n + 2) \\
&= (numBR + numE + numN) \; div \; n + 6 \\
&= (numParticipants) \; div \; n + 6 \\
&\leq (n * MEETUP_MULTIPLIER) \; div \; n + 6 \qquad \text{(Algorithm 4, line 15)} \\
&= MEETUP_MULTIPLIER + 6
\end{aligned}$$

Given this upper bound, we can say that we violate the constraint that meetup sizes should be smaller than 12, but we argue that our proposed solution has enough advantages to accept a few meetups with more than 12 participants, as this number was artificially chosen and increasing it does not have any effects on the security of the system.

5.4.4 Newbie Ratio

Figure 5.4 shows the distributions of the newbie ratios for all computed meetups in Experiment A and B. Recall that the newbie ratio is defined as the percentage of newbies in one given meetup. Note the logarithmic scale of the y-axis of the distributions in the first row of the chart. The second row of the chart shows the cumulative distributions of the same data. We can see that the distribution of newbie ratios is not optimal yet, because it should theoretically be below 0.25 in all cases. Although we do not reach a newbie ratio of 0.25 in all cases, we can see by means of the orange lines that we almost always get a newbie ratio below 0.3, namely in 99.7% of the cases in Experiment A and in 99.98% of the cases in Experiment B. In agreement with Encointer, we decided to accept this

solution. With our approach it is hard to guarantee hard bounds, because many parameters in the process are randomized. The main problem is that newbies and other participants are distributed to the meetup locations independently and this can lead to bad configurations where for example the number of reputables in one specific meetup is rounded down and the number of newbies in the same meetup is rounded up. One solution to bring down the newbie ratios would be to tweak line 10 in Algorithm 4 in order to allow fewer newbies overall which would decrease the probability of large newbie ratios at the cost of allowing fewer participants to participate in the meetups. In order to guarantee hard bounds one could potentially use combinatorial optimization in order to determine the amount of newbies allowed.



(a) Experiment A

(b) Experiment B

(c) Experiment A

(d) Experiment B

Figure 5.4: Distribution of newbie ratios

5.4.5 Number of Bootstrappers and Reputables

Because of the implemented measures described in Subsection 5.2.3, all simulated meetups had at least one bootstrapper or reputable as a participant.

5.5 Resilience to Byzantine Nodes

In this section we want to analyze how our algorithm performs in the presence of byzantine nodes in the system. A byzantine node can be either of two types:

1. A user that has already gained reputation and becomes an attacker
2. A sybil node without reputation controlled by an attacker

We will describe an attack scenario, where a group of type 1 byzantine nodes collude and use an unbounded amount of type 2 byzantine nodes to attack the system.
In order to understand the attack, we first describe the assumptions that we make and the rules of the system that are relevant for the attack.

5.5.1 Rules

In order to receive the UBI after attending a meetup, each participant has to fulfill the following rules:

1. A participant is considered a reputable when she had her personhood attested at least once in the past.
2. The reputables in the meetup have a majority vote on the amount of participants present and every user has to agree with this vote in order to get a UBI.
3. Every participant needs to receive at least $(number\ of\ reputables) - 2$ attestations of personhood from other users.

5.5.2 Assumptions

The following assumptions are made:

1. There are no endorsees and we ignore the bootstrappers (there are a maximum of 12, and they have the same voting power as reputables)
2. Each meetup consists of 10 persons with 7 reputables and 3 newbies
3. If the attacker guesses $s1$ correctly, all her nodes will end up in the same meetup (we omit the effect of $s2$)
4. The attacker has a 100% chance to register her nodes in the desired slots (no collisions with legitimate users that happen to register themselves in the same slot)

5. The attacker has each a 50% chance of influencing N and n in her favor by adding more or less nodes in the registration process

6. The attacker controls all newbies in the system

5.5.3 The Attack

We now analyze the effect of an attack, where a group of attackers that already have reputation start creating more byzantine nodes by letting sybil nodes gain reputation. In order to do so, the attackers need to win the majority vote in a meetup. This can be achieved when in one meetup 4 out of 7 reputables are controlled by the attackers. If the attack succeeds, the 3 newbies in this meetup will gain reputation. Assuming there are N reputables, of which a percentage r are malicious, the expected amount of new sybils per ceremony phase is:

$$3 * ((r * N)/4) * (1/N) * 0.5 * 0.5 \tag{5.14}$$

because there are 3 new sybils added if the attack succeeds, there are $((r*N)/4)$ potential meetups where an attack could take place (there need to be 4 malicious reputables in the meetup), the attacker has to guess $s1$ correctly which is in $[1, N]$ and she has each a 50% chance of getting N and n right.

5.5.4 Results

In order to measure the effect of this attack on the entire community, we decided to use inflation as a metric, because each legitimate participant's money loses value when an attacker creates an oversupply of money. Figure 5.5 shows the annual inflation due to the attack over a period of 100 years with the assumption of 1%, 10% of 30% malicious reputables (type 1 byzantine nodes). Each plot shows 5 curves for different community sizes. We analyze the effect of up to 30% malicious reputables, because we want to test the resilience of our assignment algorithm which does not have perfect randomness and a related analysis[6] shows that above 30%, attackers can make profit even if the distribution is perfectly random.

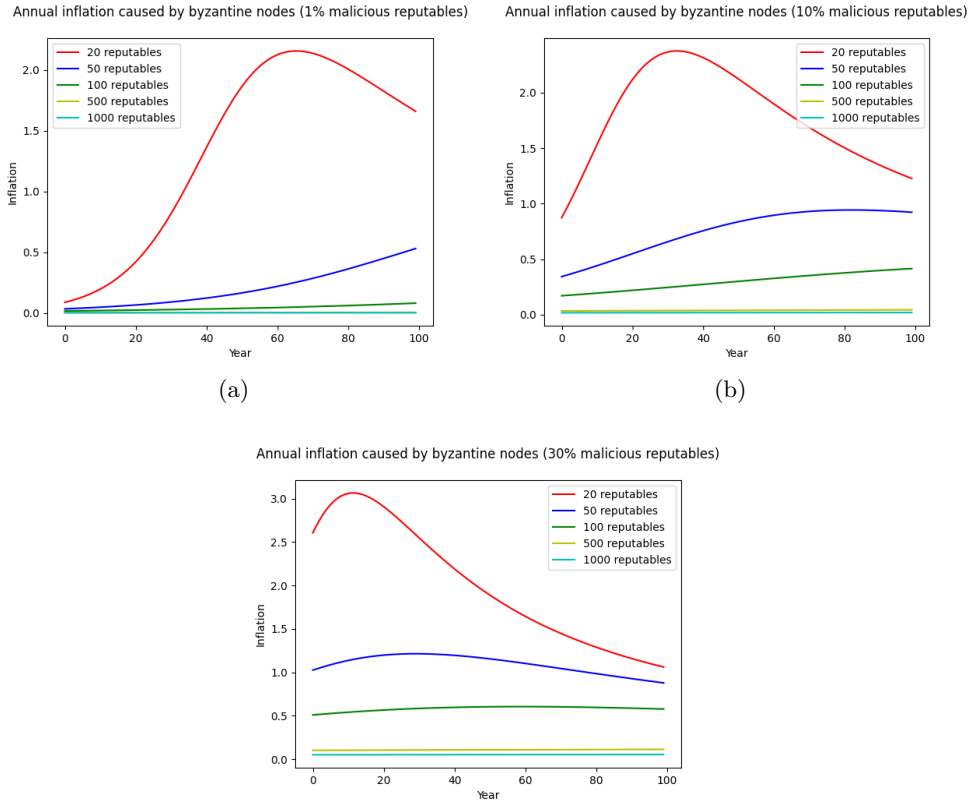


Figure 5.5: Annual inflation caused by byzantine nodes

We assume that an annual inflation of 2% is acceptable and find that the only scenario in which this is exceeded is with very small community sizes. The percentage of malicious reputables only has a secondary influence on the inflation caused. We also have to note that the assumptions made for this analysis are very favorable for the attacker, so we conclude that the system has a good resilience against byzantine nodes if the attacker controls less than 30% of the nodes.

5.6 Future Work

As discussed in Subsection 5.4.1, with our proposed algorithm we do not get full randomness when permuting the participants before assigning them to the meetup locations. Full randomness is not feasible with the approach of creating the permutation from a single seed because for an input size of n elements, in order to enumerate all the $n!$ permutations we would need a seed length of $\log_2(n!)$ bits. We get this formula by solving

$$2^x = n! \tag{5.15}$$

for x.
So already for an input size of 1000, a seed length of approximately 8529 bits would be necessary and for an input size of 10000 a seed length of 118458 bits. Although this does not seem feasible, a trade-off can be made by choosing a seed length of x bits and getting 2^x possible permutations, which is exactly what has been done in our approach. We believe that our approach is sufficient to ensure Encointer's practical security (see Subsection 5.4.1). After observing the algorithm in practice on a testnet with bot-communities, the design choices should be reevaluated and if more randomness is needed, another - potentially larger - subset of permutations can be chosen.
Should a necessity for full randomness at the cost of a lot of blockchain storage space arise during the testing phase, we propose the follwoing procedure to create a fully random permutation from a seed input. It is based on the Fisher-Yates shuffling algorithm[33] which basically works as follows: For an input list l, choose a random element from l and delete it from l. From the remaining list choose again a random element and so on. So we note that we need l random numbers, but not all of the numbers need the same amount of bits. The first random number needs to have a length of $\lceil \log_2(n) \rceil$ bits where n is the length of l, the second random number needs $\lceil \log_2(n-1) \rceil$ bits and so on. So in total we need

$$k = \sum_{i=1}^{n} \lceil \log_2(i) \rceil \tag{5.16}$$

bits of randomness for this approach to work.
After a seed of k random bits is stored on chain, the off-chain applications could simply run the procedure described in Section 5.1, using the Fisher-Yates algorithm for shuffling.
If the blockchain storage becomes a bottleneck with this solution, an alternative approach would be to store a shorter seed on-chain and use a pseudo-random generator like Mersenne Twister[34] in the off-chain applications in order to deterministically create the random permutation.
Although those solutions create better randomness, they push more complexity to another place. As mentioned in Subsection 5.2.1, the permutation should be efficiently invertible in order to find all the participants for a given meetup location. With the solutions above, this is not given. One would have to compute the entire permutation in order to find all users for a given location.

Chapter 6

Issuance of the Currency

6.1 The Problem

The third scalability problem concerns the issuance of the currency. In order to issue the UBI to the participants, the protocol has to check for each participant if she is eligible for the UBI and if so, issue the specified amount. Up until now this was implemented as a simple for-loop that performed the issuance for all users. The problem with this approach is that for a large amount of users this takes longer than the block time and therefore is not feasible. In the following we will present our solution to this problem.

6.2 Proposed Solution

The solution to this problem is straight forward. We changed the implementation such that the issuance of the currency becomes lazy, i.e. each user has to claim the UBI herself in a separate transaction. When a user claims her UBI, all other users that were participating in the same meetup will also receive their UBI, because it is more efficient to load all the personhood attestations for one meetup only once. This makes it necessary to keep state of which meetup participants already claimed their UBI in order to prevent users from claiming the currency twice. The algorithm for claiming the UBI for participant p can be summarized as follows.

1. Find the meetup index m for user p
2. Check if currency was already issued for meetup m
3. For all users of meetup m, check if they are eligible for UBI
4. Issue the currency to all eligible users in meetup m
5. Record into storage that the UBI was issued for meetup m

6.3 Implementation

The proposed solution of this problem was implemented and unit tested in Rust using the Substrate framework.

6.4 Evaluation

It is easy to verify that doing the issuance lazily avoids the bottleneck of looping over all participants. Even though the overall complexity of the algorithm is not reduced ($n * O(1)$ is equal to $O(n)$), this pragmatic approach still solves the scalability problem by removing the bottleneck.

6.5 Future Work

There is one subtle aspect of this new approach that introduces some unfairness, namely the fact that one participant has to pay the transaction fees and all other participants that attended the same meetup will get their UBI without paying any fees. It would be fair to issue a little more currency to the claimer and deduct a fraction of this amount of all other participants in order to compensate for the transaction fees. The problem here is that the transaction fees are paid in the blockchain's native currency while the UBI is issued in the local community currency. So as long as there are no exchange rates available, it is not possible to calculate the amount of compensation necessary. In the future there might be exchanges for established community currencies and oracles making those exchange rates available on-chain. Then it would be possible to implement a scheme as described above.

Chapter 7

Conclusion

In this thesis we solved three major scalability problems of Encointer and implemented the solutions into the existing codebase. For the location validation problem we found an algorithmic solution based on geohashing which reduces the runtime from $O(n)$ to $O(1)$. A very pragmatic solution was chosen for the currency issuance problem. We removed a current bottleneck by changing the issuance of the currency to be conducted in a lazy fashion. The meetup assignment problem was the most challenging and therefore has the most complex solution. We used an approach based on modular arithmetic to generate invertible random permutations based on a random seed input in order to reduce computing time on the blockchain from $O(n)$ to constant time. In multiple iterations over the solution we had to cautiously handle trade-offs between different requirements defined by Encointer. As the entire system consists of many free parameters, it is impossible to formally prove its security. Therefore we conducted an extensive simulation of the system as well as theoretical analyses of different attack scenarios which all indicate that the system is sufficiently secure. Nevertheless it will be important to closely observe the algorithm performing on a testnet and in the early beta stages of the live system in order to evaluate the design choices made in this thesis.
All code developed in this thesis was merged to Encointer's production system which will soon launch its first community currency in Zurich.

Bibliography

[1] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review (2008) 21260

[2] Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014) (2014) 1–32

[3] Wood, G.: Polkadot: Vision for a heterogeneous multi-chain framework. White Paper **21** (2016)

[4] Brenzikofer, A. Encointer whitepaper. https://github.com/encointer/whitepaper/raw/master/encointer_whitepaper.pdf Accessed on 04 Aug 2021.

[5] Gesell, S.: The natural economic order. Owen London (1958)

[6] Hoffmann, L.: Security analysis of proof-of-personhood: Encointer. (2021)

[7] Substrate. https://substrate.dev/docs/en/ Accessed on 04 Aug 2021.

[8] Substrate benchmarking framework. https://substrate.dev/docs/en/knowledgebase/runtime/benchmarking Accessed on 04 Aug 2021.

[9] Klabnik, S., Nichols, C.: The Rust Programming Language (Covers Rust 2018). No Starch Press (2019)

[10] Rust standard library. https://doc.rust-lang.org/std/ Accessed on 17 Aug 2021.

[11] Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with webassembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. (2017) 185–200

[12] Moussalli, R., Srivatsa, M., Asaad, S.: Fast and flexible conversion of geohash codes to and from latitude/longitude coordinates. In: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, IEEE (2015) 179–186

[13] Miller, R., Maguire, P.: Geotree: a data structure for constant time geospatial search enabling a real-time mix-adjusted median property price index. arXiv preprint arXiv:2008.02167 (2020)

[14] Motzkin, T.: The euclidean algorithm. Bulletin of the American Mathematical Society **55**(12) (1949) 1142–1146

[15] Siddarth, D., Ivliev, S., Siri, S., Berman, P.: Who watches the watchmen? a review of subjective approaches for sybil-resistance in proof of personhood protocols. Frontiers in Blockchain **3** (2020) 46

[16] Ford, B., Strauss, J.: An offline foundation for online accountable pseudonyms. In: Proceedings of the 1st workshop on Social network systems. (2008) 31–36

[17] Borge, M., Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Ford, B.: Proof-of-personhood: Redemocratizing permissionless cryptocurrencies. In: 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), IEEE (2017) 23–26

[18] Wang, F., De Filippi, P.: Self-sovereign identity in a globalized world: Credentials-based identity systems as a driver for economic inclusion. Frontiers in Blockchain **2** (2020) 28

[19] Lesae, C., George, W., Ast, F. Kleros yellowpaper. https://kleros.io/yellowpaper.pdf (2021)

[20] Kleros proof of humanity. https://blog.kleros.io/proof-of-humanity-an-explainer/ Accessed on 04 Aug 2021.

[21] Caronni, G.: Walking the web of trust. In: Proceedings IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2000), IEEE (2000) 153–158

[22] Arjomandi-Nezhad, A., Fotuhi-Firuzabad, M., Dorri, A., Dehghanian, P.: Proof of humanity: A tax-aware society-centric consensus algorithm for blockchains. Peer-to-Peer Networking and Applications (2021) 1–13

[23] Myubi whitepaper. https://docs.myubi.io/whitepaper/white-paper Accessed on 13 Aug 2021.

[24] Ubic whitepaper. https://github.com/UBIC-repo/Whitepaper/archive/master.zip Accessed on 13 Aug 2021.

[25] Freicoin. http://freico.in Accessed on 13 Aug 2021.

[26] Lee, D.T., Schachter, B.J.: Two algorithms for constructing a delaunay triangulation. International Journal of Computer & Information Sciences **9**(3) (1980) 219–242

[27] Brenzikofer, A., Langenbacher, C. Delauney triangulaition approach for improving transaction verification algorithm. https://github.com/encointer/pallets/issues/17 Accessed on 04 Aug 2021.

[28] https://en.wikipedia.org/wiki/List_of_northernmost_settlements Accessed on 12 Aug 2021.

[29] https://en.wikipedia.org/wiki/Southernmost_settlements Accessed on 12 Aug 2021.

[30] Erickson, T. Pykml library. https://github.com/pykml/pykml Accessed on 04 Aug 2021.

[31] Geohash for rust. https://github.com/georust/geohash Accessed on 04 Aug 2021.

[32] O'Connor, D.: A historical note on shuffle algorithms. Retrieved Maret **4** (2014) 2018

[33] Eberl, M.: Fisher-yates shuffle. Arch. Formal Proofs **2016** (2016)

[34] Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1) (jan 1998) 3–30