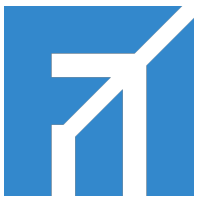
ALEXANDRU IOAN CUZA UNIVERSITY IAŞI

FACULTY OF COMPUTER SCIENCE



BACHELOR THESIS

DC & Mix Networks

submitted by

Vlad Manea

Term: *June, 2011*

Scientific coordinator

Prof. Dr. Ferucio Laurențiu Țiplea

ALEXANDRU IOAN CUZA UNIVERSITY IAŞI

FACULTY OF COMPUTER SCIENCE

DC & Mix Networks

Vlad Manea

Term: *June, 2011*

Scientific coordinator

Prof. Dr. Ferucio Laurenţiu Ţiplea

Declarație privind originalitate și respectarea drepturilor de autor

Prin prezenta declar că Lucrarea de licenţă cu titlul "DC & Mix Networks" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imagini etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Absolvent *Vlad Manea*

Iaşi, ----------

(semnătură în original)

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul "DC & Mix Networks", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea Alexandru Ioan Cuza Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Absolvent *Vlad Manea*

Iaşi, _____

(semnătură în original)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Communication networks are everywhere and incredibly accessible. According to [40], in March 2011 over 30% of the World population has access to this global resource. As Marshall McLuhan stated in the introduction of his *Understanding Media: the Extensions of Man* [41], "Today, after more than a century of electric technology, we have extended our central nervous system in a global embrace, abolishing both space and time as far as our planet is concerned". The use of these communication networks at large may seem that the Internet allows an end user to communicate in partial obscurity. However, the advances of communication infrastructure allow data traffic to be easily gathered, analyzed and modified. This leads to the need for safe systems in which this data is kept secret. This leads to the need of personalized services capable of processing sensitive data while keeping it private and its issuer anonymous.

A communication is private if it includes confidential data. In order to keep this property, it is mandatory for that particular data to not be accessed by any unauthorized entity. Perhaps as important as its privacy, the sensitive data should not be linked to a particular user. We may say that data is untraceable if its source cannot be tracked back from the destination. Some cases where anonymity is paramount are:

- Electronic voting. Votes should not be traced to their voters, even though their values are finally revealed; the initial secrecy of the vote is its privacy property. The electoral authority shall be capable of proving the vote is correct, while keeping any link between a particular ballot and its user secret. This property is the untraceablilty of the vote.

- Medical records. Prevent online user profiling by marketers and researchers by analyzing confidential medical data and personal opinions on sensitive information. According to [42], 87% of the population in the United States had reported characteristics that made them unique based only on their zip code, their gender and their date of birth. Unrestricted access to such data should break anonymity and, with high probability, link patients seen as random to exact citizens.

- Email. Messages are sent across the Internet with information about their sender and receiver, such that anyone is able to observe which hops are traversed from its source to its destination. This allows to identify who is communicating with whom, even though the data in that message is encrypted.

- Governmental agencies. Law enforcement units may provide a tip line anonymous service, in which normal people may report any unlawful activity without being afraid of punishment. Intelligence agencies may monitor malicious organization Websites without having their presence revealed to their webmasters. Anonymity may also be useful for censorship resistance for people who live in regimes where their data are filtered. Providers of freely accessible services on the Internet may stay anonymous in order to not be banned by the oppressive authorities.

- Electronic Shopping. Citizens may not only want their products to be kept secret, but also their location of online shopping to be private. They want to browse the product lists and check out the cart without being profiled by advertizers based on their shopping habits.

An anonymous system over some set of events $E = \{e_1, e_2, ..., e_{n-1}, e_n\}$ has the property that, even though the occurence of an event $e_i \in E$ had happened and it is a publicly known fact, an observer is unable to tell which event had occured. We may say that the set of events E is the anonymity set. This may be of course mapped to event causes, such as citizens who send votes, given that each event has a known cause.

In [43], anonymity is defined as "not being identifiable within a set of subjects, the anonymity set", where the anonymity set is the set of all possible subjects, i.e. in our case, E or the set of causes of E. From an attacker perspective, anonymity of a subject "means that the attacker cannot sufficiently identify the subject within a set of subjects, the anonymity set". Other cryptographic properties related to anonymity are: unlinkability, undetectability and unobservability. Unobservability is defined as:

- "undetectability of the item of interest against all subjects uninvolved in it", and
- "anonymity of the subject(s) involved in the item of interest even against the other subject(s) involved in that item of interest".

In our thesis, we focus on the properties of anonymity and unobservability. DC Networks and Mix Networks are the most widespread mechanisms to provide sender and recipient anonymity [43, p. 14] against strong attackers. By adding dummy traffic to the network, both models provide unobservability.

We contribute to this topic by analyzing the DC network model and collusion of participants on a general model, by surveying the current mix network models that provide sender, recipient and two way anonymity, and by describing the flush algorithms and deployed systems. We propose a simple new algorithm that solves the problem of messages that remain in the mix for an arbitrarily long amount of time, which is the case for many anonymity providing flush techniques. We also provide an implementation of the Parallel Mix Network model [26] by using the Hadoop open source distributed computing framework [36] based on the Map Reduce programming model [35].

The thesis is structured as follows. In chapter 1 we provide the introduction. Chapter 2 describes the original DC problem and its solution, the DC Network, and it also presents a generalization by modelling a graph, by analyzing some particular topologies and by proving the anonymity set holds in a connected graph; the chapter also provides a different solution to the DC problem. Chapter 3 analyzes Mix Networks, by starting from the anatomy of a single mix and by extending it to a network of mixes in which message tracking based on its size and sender, recipient and two way anonymity are discussed. Chapter 4 analyzes flush algorithms for mixes with and without memory, and it also describes our algorithm. Chapter 5 surveys proof techniques for mix networks, whose complexities or proving range from $\mathcal{O}(nk)$ to $\mathcal{O}(k)$ exponentiations, where n is the number of messages and k is the number of mixes. Chapter 6 describes the modus operandi of some practical implementations of mix networks, such as Anonymizer, Onion Routing and Tor. Chapter 7 analyzes this programming model and the Apache Hadoop open source distributed computing framework; it then describes our application model. In chapter 8 we conclude.

Chapter 2

DC Networks

The original problem is introduced by a situation in which three cryptographers are having dinner at their favourite hotel. The waiter informs them that arrangements have been made so that the bill is to be paid either by the U.S. National Security Agency (NSA), or by one of the three anonymously. Any cryptographer respects each other's right to pay in secret for the dinner, but they all wonder if the NSA pay instead.

This chapter is structured as follows: section 2.1 describes the original DC problem and its solution as a communication network; section 2.2 presents an alternative approach for the DC solution; section 2.3 generalizes it by modelling a graph and analyzes some particular graph topologies; section 2.4 provides a proof of anonymity of a set with regard to the collusion of participants; section 2.5 describes a different solution to the DC problem.

2.1 Original problem

The idea of the original protocol in [1] is to have a shared secret between neighbours: each cryptographer flips an unbiased coin and shows it to the cryptographer on his right side. After that, each cryptographer makes a public statement:

- if he did not pay for the dinner, he says if his own coin and the left neighbour's coin fell on the same face or not.
- if he did not pay for the dinner, he says the opposite of the actual event.

If all cryptographers follow the protocol, and there is only one payer (either a cryptographer, or the NSA), a total odd number of differences at the table indicates that a cryptographer payed for the dinner, while an even number of differences indicates that the NSA payed. However, if a cryptographer payed, neither of the other two can tell if he payed. This protocol is known as the Dining Cryptographers (DC) Network.

To prove this, we use a particular boolean algebra $(\{0,1\}, \vee, \wedge, ', 0, 1)$ where 0 is $false$, 1 is $true$, $\vee$ is the disjunction, $\wedge$ is the conjunction, $0'$ is 1, and $1'$ is 0 (more details on this algebra can be found in [2, p. 120]). We convene that 1 means a coin face and 0 stands for the other coin face.

We also use the $\oplus$ operator:

$$x \oplus y = \begin{cases} 1, & x \neq y, \\ 0, & \text{otherwise}, \end{cases}$$

$$\text{for any } x, y \in \{0,1\}^1.$$

We now translate the original problem (where $i \in \{1,2,3\}$) into our chosen algebra.

We denote by K_i the knowledge of the i^{th} participant. When the knowledge of some event is not important, we may omit its value or the element itself.

[1] The $\oplus$ operator is also known as the xor operator.

1. The i^{th} participant flips an unbiased coin c_i and shows it to his right neighbour. Denote $\mathcal{P}[E]$ as the probability of the event E. The protocol states that

$$\mathcal{P}[c_i = 0] = \mathcal{P}[c_i = 1], c_i \in \{0, 1\}. \tag{2.1}$$

   If, for instance, for some secret i, $c_{(i+1)\%3+1} = 0$ and $c_i = 1$, then

$$K_i = \{c_{(i+1)\%3+1} : 0, c_i : 1\}.$$

2. The i^{th} participant publicly states the $\oplus$ sum of his keys.
   Let p_i say whether the i^{th} participant pays or not:

$$p_i = \begin{cases} 1, & i \text{ pays}, \\ 0, & \text{otherwise}. \end{cases}$$

   If, for example, the secret $p_i = 1$, then

$$\begin{aligned} K_i &= \{c_{(i+1)\%3+1}, c_i\} \cup \{p_i : 1\} \\ &= \{c_{(i+1)\%3+1}, c_i, p_i : 1\}. \end{aligned}$$

   Let s_i be the public statement of the i^{th} participant:

$$s_i = c_{(i+1)\%3+1} \oplus c_i \oplus p_i.$$

   For example, if the public $s_1 = 0$, $s_2 = 0$ and $s_3 = 1$, then

$$\begin{aligned} K_i &= \{c_{(i+1)\%3+1}, c_i, p_i\} \cup \{s_1 : 0, s_2 : 0, s_3 : 1\} \\ &= \{c_{(i+1)\%3+1}, c_i, p_i, s_1 : 0, s_2 : 0, s_3 : 1\}. \end{aligned}$$

3. They compute the $\oplus$ sum of all statements.
   If all participants have followed the protocol, and no more than one pay[2], a total even or odd number of differences indicates the payer type: the NSA or one of the participants, respectively. Say t is the modulo 2 sum of declared differences.

   (a) If no participant pays, they will all tell the truth in their s_i's:

$$\begin{aligned} t &= s_1 \oplus s_2 \oplus s_3 \\ &= (c_3 \oplus c_1 \oplus p_1) \oplus (c_1 \oplus c_2 \oplus p_2) \oplus (c_2 \oplus c_3 \oplus p_3) \\ &= c_3 \oplus c_1 \oplus 0 \oplus c_1 \oplus c_2 \oplus 0 \oplus c_2 \oplus c_3 \oplus 0 \\ &= 0. \end{aligned}$$

   (b) If one of them (e.g. the second participant) pays, he will tell a lie in his s_i:

$$\begin{aligned} t &= s_1 \oplus s_2 \oplus s_3 \\ &= (c_3 \oplus c_1 \oplus p_1) \oplus (c_1 \oplus c_2 \oplus p_2) \oplus (c_2 \oplus c_3 \oplus p_3) \\ &= c_3 \oplus c_1 \oplus 0 \oplus c_1 \oplus c_2 \oplus 1 \oplus c_2 \oplus c_3 \oplus 0 \\ &= 1. \end{aligned}$$

   If, for example, $t = 1$, then

$$\begin{aligned} K_i &= \{c_{(i+1)\%3+1}, c_i, p_i, s_1, s_2, s_3\} \cup \{t : 1\} \\ &= \{c_{(i+1)\%3+1}, c_i, p_i, s_1, s_2, s_3, t : 1\}. \end{aligned}$$

[2]If an odd number of them pay, there is an odd number of 1's artificially inserted in the s_i's, and the result will show as if only one of them pays. If the number of payers is even, there will be an even number of 1's in the s_i's, and the result will point the NSA as the payer, leading to a wrong response.

We will prove that if the protocol is respected by all participants, and one of them pays, it is secure (if the NSA pay, there is no sender anonymity problem at all). If a participant pays for the dinner, he already knows he is the payer. This translates to $p_i \in K_i$. Otherwise, he does not pay, he knows that some other participant pays and he wants to find out who exactly. We will prove this is impossible, i.e.

$$p_j \notin K_i, \text{ for any } i, j \in \{1, 2, 3\} \text{ with } i \neq j.$$

Let the participant who questions be the second, the participant who shares the coin face with him be the first and the participant with whom he shares his coin be the third. We prove that in any given situation, two contradictory conclusions regarding p_1 and p_3 emerge at the same time, and there is no way to choose one and drop the other.

We know that $c_3 \notin K_2$, because c_3 is shared between the third and first participants only. The second participant can only assume the actual value of c_3, and each of the two possible values has same odds as the other, as we know (1.1).

Suppose that the first and second participants share the keys $c_1 = c_2 = \chi \in \{0, 1\}$.

1. By assuming that $c_3 = \chi$, the first participant's knowledge is:

$$\begin{aligned} K_2 &= \{c_1 : \chi, c_2 : \chi, p_2 : 0, s_1, s_2 : 0, s_3, t : 1\} \\ &= \{c_1 : \chi, c_2 : \chi, p_2 : 0, s_1, s_2 : 0, s_3, t : 1, p_1 : s_1\} && (p_1 = s_1 \oplus c_3 \oplus c_1) \\ &= \{c_1 : \chi, c_2 : \chi, p_2 : 0, s_1, s_2 : 0, s_3, t : 1, p_1 : s_1, p_3 : s_3\} && (p_3 = s_3 \oplus c_2 \oplus c_3) \\ &= \{p_1 : s_1, p_3 : s_3, ...\}. \end{aligned}$$

2. However, by assuming that $c_3 = \chi'$, the knowledge is inconsistent:

$$\begin{aligned} K_2 &= \{c_1 : \chi, c_2 : \chi, p_2 : 0, s_1, s_2 : 0, s_3, t : 1\} \\ &= \{c_1 : \chi, c_2 : \chi, p_2 : 0, s_1, s_2 : 0, s_3, t : 1, p_1 : s_1'\} && (p_1 = s_1 \oplus c_3 \oplus c_1) \\ &= \{c_1 : \chi, c_2 : \chi, p_2 : 0, s_1, s_2 : 0, s_3, t : 1, p_1 : s_1', p_3 : s_3'\} && (p_3 = s_3 \oplus c_2 \oplus c_3) \\ &= \{p_1 : s_1', p_3 : s_3', ...\}. \end{aligned}$$

Because p_1 has opposite values in equally probable cases, the second cannot say that $p_1 = s_1$ or $p_1 = s_1'$. This applies as well to p_3, s_3 and s_3'. Therefore, $p_1, p_3 \notin K_2$.

Now let's see if the protocol's security holds when the first and second participants share different keys, i.e. $c_1 = \chi$ and $c_2 = \chi'$, where $\chi \in \{0, 1\}$.

1. If the questioning participant supposes that $c_3 = \chi$, then his knowledge will be

$$\begin{aligned} K_2 &= \{c_1 : \chi, c_2 : \chi', p_2 : 0, s_1, s_2 : 1, s_3, t : 1\} \\ &= \{c_1 : \chi, c_2 : \chi', p_2 : 0, s_1, s_2 : 1, s_3, t : 1, p_1 : s_1\} && (p_1 = s_1 \oplus c_3 \oplus c_1) \\ &= \{c_1 : \chi, c_2 : \chi', p_2 : 0, s_1, s_2 : 1, s_3, t : 1, p_1 : s_1, p_3 : s_3'\} && (p_3 = s_3 \oplus c_2 \oplus c_3) \\ &= \{p_1 : s_1, p_3 : s_3', ...\}. \end{aligned}$$

2. If he supposes that $c_3 \neq \chi$, then his knowledge will change:

$$\begin{aligned} K_2 &= \{c_1 : \chi, c_2 : \chi', p_2 : 0, s_1, s_2 : 1, s_3, t : 1\} \\ &= \{c_1 : \chi, c_2 : \chi', p_2 : 0, s_1, s_2 : 1, s_3, t : 1, p_1 : s_1'\} && (p_1 = s_1 \oplus c_3 \oplus c_1) \\ &= \{c_1 : \chi, c_2 : \chi', p_2 : 0, s_1, s_2 : 1, s_3, t : 1, p_1 : s_1', p_3 : s_3\} && (p_3 = s_3 \oplus c_2 \oplus c_3) \\ &= \{p_1 : s_1', p_3 : s_3, ...\}. \end{aligned}$$

Because p_1 has different values in equally probable cases, the second cannot say that $p_1 = s_1$ or $p_1 = s_1'$. This holds true for p_3, s_3 and s_3' too. Therefore, $p_1, p_3 \notin K_2$.

The conclusion is the same, whether the first and second participants flip equal or different coin faces. Therefore, $p_1, p_3 \notin K_2$.

If the second participant finds out the secret key of the third participant (c_3), then he will compute s_3 and then p_3; the protocol will become insecure.

2.2 Another approach

Another version, based on the original 3-cryptographer protocol, is discussed in [3].
We will give an example of how this version works in an abelian group G = $(G, +, -, 0)$ (for details, see [2, p. 246]):

1. The i^{th} participant ($i \in \{1, 2, 3\}$) exchanges a symmetric key with each other.
   The secret keys retained by the two participants are inverse. For instance, the key between the i^{th} and j^{th} participants, will be known as $k_{i,j}$ for the i^{th} participant, and as $k_{j,i}$ for the j^{th} participant, where $k_{i,j} = -k_{j,i}$, for any $i \neq j$.

2. Each participant adds up his keys and publicly declares his sum.
   At most one[3] participant decides to pay for the dinner. If so, he adds an artificial message to the sum, different from 0. If he is not paying, he just adds 0. For instance, if the second pays, his message is $m_2 \neq 0$, and his sum is $s_2 = k_{2,1} + k_{2,3} + m_2$.

3. The three sums are all added up, yielding the result.
   The final result t will reveal the message of the paying participant, if exists:

$$\begin{aligned} t &= s_1 + s_2 + s_3 \\ &= k_{1,2} + k_{1,3} + m_1 - k_{1,2} + k_{2,3} + m_2 - k_{1,3} - k_{2,3} + m_3 \\ &= m_1 + m_2 + m_3. \end{aligned}$$

We will prove that this protocol is as secure as the original version in [1]. If all mesages $m_i = 0$, then the payer is the NSA, and no sender anonymity problem exists. Else, one of the participants is the payer, and $t \neq 0$. In this case, we focus again on one participant that does not pay (his $m_i = 0$), but wants to know which of the other two pays. Say the questioning participant is the second. He knows:

$$\begin{array}{lll} K_2 & = \{k_{2,1}, k_{2,3}, m_2, s_1, s_2, s_3, t\} & \\ & = \{k_{2,1}, k_{2,3}, m_2, s_1, s_2, s_3, t, k_{1,2}, k_{3,2}\} & (k_{ij} = -k_{ji}, \forall i \neq j) \\ & = \{k_{2,1}, k_{2,3}, m_2, s_1, s_2, s_3, t, k_{1,2}, k_{3,2}, (m_1 + k_{1,3})\} & (m_1 + k_{1,3} = s_1 + k_{2,1}) \\ & = \{k_{2,1}, k_{2,3}, m_2, s_1, s_2, s_3, t, k_{1,2}, k_{3,2}, (m_1 + k_{1,3}), (m_3 - k_{1,3})\} & (m_3 - k_{1,3} = s_3 + k_{2,3}) \\ & = \{(m_1 + k_{1,3}), (m_3 - k_{1,3}), ...\}. & \end{array}$$

His most refined knowledge about m_1 and m_3 is that $0 \neq t = m_1 + m_3$, as $m_2 = 0$.
He does not know the secret key k_{13} or its inverse k_{31} either.
His knowledge can be mapped as a system:

$$m_1 = 0 \vee m_3 = 0 \ \} \wedge \left\{ \begin{array}{ll} k_{2,j} & = \kappa_{2,j} \\ s_j & = \sigma_j \\ m_2 & = 0 \end{array} \right\} \wedge \left\{ \begin{array}{ll} s_1 + s_2 + s_3 & \neq 0 \\ k_{f,i} + k_{f,k} + m_f & = s_f. \end{array} \right.$$

There exist at least two different solutions that satisfy the system, and they are contradictory in what regards the origin of the message different to the zero element of G.
The second participant is unable to decide what is the real solution among these:

$$(m_1, m_3, k_{1,3}) \in \{(0, \lambda, \psi), (\lambda, 0, \psi)\}; \lambda \neq 0; \lambda, \psi \in G.$$

Therefore, the protocol remains secure, when analyzed by a non-payer: the unseen secret key k_{13} hides the two messages m_1 and m_3. Of course, a person from outside the dinner table has even less information.

[3]The restriction to have at most one payer still holds true, see [2].

2.3 General model

A generalization consists in increasing the number of games, i.e. more rounds of payment. We will show the procedure for an approach based on the alternative model. For a number r of rounds, the i^{th} participant can have a secret array of r values in G in common with each other participant, so that the j^{th} value of an array is the key shared between the two in the j^{th} round, where $1 \leq j \leq r$. For instance, when $r = 2$ and

$$k_{1,2}[1..3] = \{+\alpha, -\beta, +\gamma\},\ k_{1,3}[1..3] = \{-\mu, -\nu, +\rho\},\ k_{2,3}[1..3] = \{+\tau, -\omega, -\alpha\},$$
$$k_{2,1}[1..3] = \{-\alpha, +\beta, -\gamma\},\ k_{3,1}[1..3] = \{+\mu, +\nu, -\rho\},\ k_{3,2}[1..3] = \{-\tau, +\omega, +\alpha\},$$

then in the second round the keys will be:

$$k_{1,2}[2] = -\beta,\ k_{1,3}[2] = -\nu,\ k_{2,3}[2] = -\omega,$$
$$k_{2,1}[2] = +\beta,\ k_{3,1}[2] = +\nu,\ k_{3,2}[2] = +\omega.$$

Another generalization consists in having any number of participants, greater than three[4].

We can map the dinner table as a graph (V, E), where the i^{th} participant corresponds to the vertex with label i, and each key shared between the i^{th} and j^{th} participant, namely k_{ij} and k_{ji}, corresponds to the same edge $[i, j]$. We will also presume the graph is connected, because each connected component is a dinner table in itself.

We define the anonymity set of a subset of keys as a set of vertices in a connected component of the initial graph from which we remove the edges corresponding to the keys. We explain below some examples:

- Let us consider 8 participants and the set of keys $\{k_{1,2}, k_{1,8}, k_{2,3}, k_{2,6}, k_{2,7}, k_{3,4}, k_{3,6}, k_{5,6}\}$. If we choose the subset of keys $\{k_{2,3}, k_{2,7}, k_{3,6}, k_{5,6}\}$ from the initial set, then in the resulting graph we will select the corresponding edges $[2, 3]$, $[2, 7]$, $[3, 6]$, $[5, 6]$:

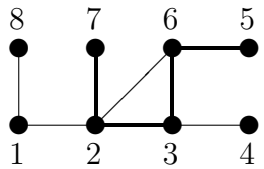


Figure 2.1: Graph in which we choose edges $[2, 3]$, $[2, 7]$, $[3, 6]$, $[5, 6]$.

  The anonymity sets of the keys are the vertices in the connected components of the graph formed by the 8 vertices and the thin edges: $\{1, 2, 6, 8\}, \{3, 4\}, \{5\}, \{7\}$.

- An observer from outside sees the empty set of keys. The graph is connected and remains the same after no edge is removed. Therefore, the only anonymity set seen by no keys is the graph's set of vertices.

- An observer from above the table sees them all. If all possible keys are selected, the graph will lose all its edges. The anonymity sets of all keys are all $\mid V \mid$ sets, such that each vertex forms a set of its own.

- A participant sees himself. If all keys that belong to a participant are known, his corresponding vertex will be in an anonymity set of its own.

- If all keys form a graph such as each pair of vertices are connected by at least two vertex-disjoint paths, a set of keys that correspond to the edges incident in a vertex also see all other $\mid V \mid -1$ vertices in one anonymity set.

[4]When there are only two participants, $t = s_1 + s_2 = m_1 + m_2$ and only an outsider who does not know $k_{1,2}$, nor its inverse $k_{2,1}$ is unable to spot the payer, if exists.

2.4 Collusion of participants

When some of the participants are dishonest and they are cooperating by pooling their keys, a collusion between them happens. A collusion is full if all keys that belong to the subset of participants are pooled.

If all keys that belong to a participant are shared between colluders, his message can be traced simply. However, if two participants maintain a secret key and they do not divulge their messages, all remaining participants in full collusion find nothing about the two messages. As depicted in figure 2.2, where $\nu \neq 0$ and σ_i is the known sum of the other keys that belong to the i^{th} participant, two contradictory and yet equally probable situations may happen.

It can be proven that a subset of colluders can get from one of their keys' anonymity sets just the sum of their messages, but never each message, with the exception of the anonymity sets formed by one vertex only. The proof in [1, s. 1.4] uses random variables and the incidence matrix of the connex component to prove that. We will show a more intuitive picture for this:

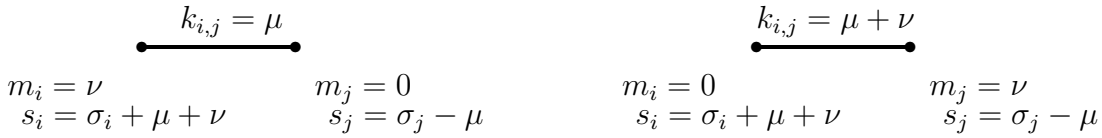


Figure 2.2: Public s_i and s_j stay the same, but secret $k_{i,j}$ makes m_i and m_j ambiguous.

Let us suppose there is one subset of full colluders only, and this subset is interested in an anonymity set that issued the nonzero message and consists of more than one vertex.

We apply a permutation to the vertex labels and we update the edge extremities accordingly. Let vertices 1 to $d-1$ depict the full colluders, vertices d to $f-1$ be the reunion of all other anonymity sets, and vertices f to $\mid V \mid$ represent the anonymity set. Let P be one of the paths from f to k, where $k > f$, such that all inner vertices i along the path satisfy $f < i < k$ and are distinct. Denote by $length(P)$ the number of edges in P.

The knowledges of full colluders are all the same, since every colluder shares all his secret keys with the other colluders, and each colluder issued the message 0. We will prove that for a subset of full colluding participants, $m_f, m_k \notin K_1$, where K_c is the knowledge of a full colluder:

$$K_c = \{m_l : 0, k_{l,v}, k_{v,l} : -k_{l,v}, s_v, t : \nu \neq 0\}, \text{ for all } 1 \leq l < d, 1 \leq v \leq \mid V \mid.$$

Because the anonymity set is connected, path p exists, for any $f < k \leq \mid V \mid$. If inner vertices do not exist, then $[f,k] \in E$, and the situation depicted in figure 2.2 applies. Otherwise, there is at least a vertex i along the path. Let j be the last inner vertex when one follows the path from f to k. Note that sometimes $i = j$. Let also σ_h be the sum of all keys of the h^{th} participant, without those in P.

For both situations in figures 2.2 and 2.3, even though the σ values and even their terms are known, equally probable events, contradictory in what regards m_f, yield to not knowing whether $m_f = 0$ or $m_f = \nu \neq 0$. This applies to m_k, 0 and $\nu \neq 0$ as well:

$$\begin{aligned} \mid G \mid^{-length(p)} = & \; pr[(k_{f,i} = \mu_{f,i}) \wedge (k_{i,next_i} = \mu_{i,next_i}) \wedge ... \\ & \wedge (k_{prev_j,j} = \mu_{prev_j,j}) \wedge (k_{j,k} = \mu_{j,k})] \\ = & \; pr[(k_{f,i} = \mu_{f,i} + \nu) \wedge (k_{i,next_i} = \mu_{i,next_i} + \nu) \wedge ... \\ & \wedge (k_{prev_j,j} = \mu_{prev_j,j} + \nu) \wedge (k_{j,k} = \mu_{j,k} + \nu)]. \end{aligned}$$

The secret messages $m_f \notin K_c$ and $m_k \notin K_c$. This applies for any pair of participants in the anonymity set, which means that each (named the f^{th}) may have message ν with the same probability as each other (named the k^{th}). Therefore, the participants in the set send anonymous messages.

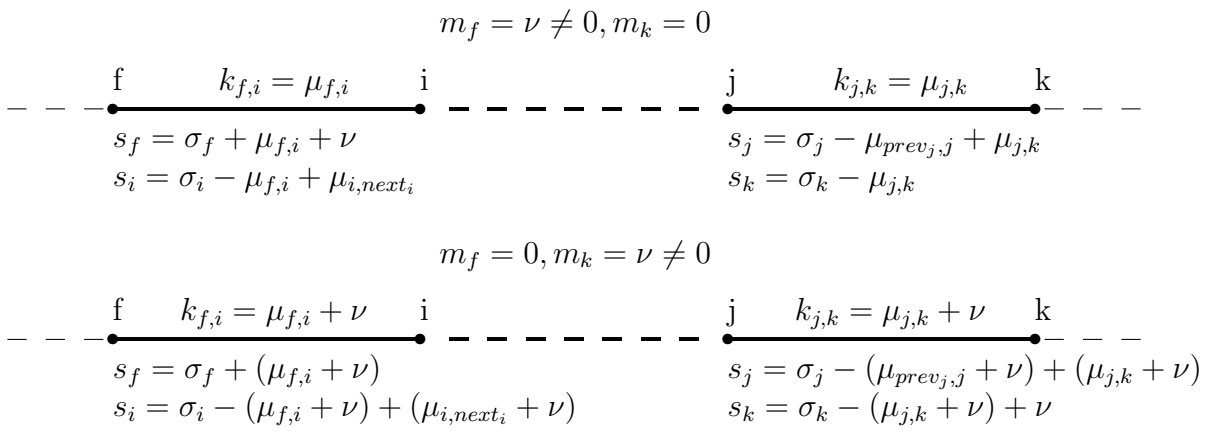


Figure 2.3: Public $s_f = s_k$, but secret $k_{f,i}, ..., k_{j,k}$ make secret m_f and m_k unclear.

2.5 Anonymous Veto Protocol

The Dining Cryptographers Networks use a broadcast channel to send anonymous messages from untraceable senders. They also rely on unconditionally secret channels for pairwise key sharing. The number of these channels gets unpractically large, as they change squarely to the number of participants. Collisions and disruptions are also limitations of the DC Networks, which require complicated solutions in order to be solved.

The Dining Cryptographers networks are not the only solution to the problem. A different perspective is to veto the result, i.e. to vote against the statement that none of the cryptographers has paid. If there exists a veto, one of the cryptographers will have paid. Otherwise, the NSA was the payer.

An efficient veto protocol, the Anonymous Veto Network (AV-Net) is presented in [4]. Their protocol does not need unconditionally secret channels anymore and it is more resistant to collisions and disruptions. The only requirement for the protocol is the existence of an authenticated broadcast channel, which can be created by using digital signatures (for details, see [1, s. 2.4]).

The protocol assumes that G is a finite cyclic group of prime order q and generator g in which the Decision Diffie Hellman (DDH) problem is intractable. We denote by n the number of participants in the protocol who agree on (G, g).

We will say an algorithm is efficient if it is a probabilistic polyomial time algorithm. We also denote by negligible a function which is smaller than $1/n^\alpha$, for all $\alpha > 0$ and a sufficiently large n. Given g^a and g^b, one must not be able to deduce g^{ab}. The DDH assumption states that there is no efficient probabilistic algorithm $\mathcal{A}$ that given any tuple (g, g^a, g^b, g^c) in G^3 outputs 'true' iff $ab = c$. More formally, there does not exist an algorithm A that satisfies, for a fixed $\alpha > 0$, a sufficiently large n and $a, b, c \in_R \mathbb{Z}_q$:

$$| \mathcal{P}[\mathcal{A}(g, g^a, g^b, g^{ab}) = true] - \mathcal{P}[\mathcal{A}(g, g^a, g^b, g^c) = true] | > \frac{1}{n^\alpha}.$$

The protocol consists of two rounds:

1. Each participant P_i broadcasts g^{x_i} and a knowledge proof for x_i, where x_i is a private randomly chosen value ($x_i \in_R \mathbb{Z}_q$). When the round is finished, each participant computes
$$g^{y_i} = \left(\prod_{j=1}^{i-1} g^{x_j} \right) \left(\prod_{j=i+1}^{n} g^{x_j} \right)^{-1}.$$

2. Each participant P_i broadcasts $(g^{y_i})^{c_i}$ and a knowledge proof for c_i.
The value c_i is either x_i, or a random value $x_i \neq r_i \in_R \mathbb{Z}_q$:

$$(g^{y_i})^{c_i} = \begin{cases} (g^{y_i})^{r_i}, & \text{if } P_i \text{ sends a veto,} \\ (g^{y_i})^{x_i}, & \text{if } P_i \text{ otherwise.} \end{cases}$$

After the two rounds are finished, each participant computes the result $R = \prod_{i=1}^{n} (g^{y_i})^{c_i}$:

$$R = \prod_{i=1}^{n} (g^{y_i})^{c_i} = \begin{cases} 1, & \text{if nobody sent a veto,} \\ \lambda \neq 1 & \text{otherwise.} \end{cases}$$

Let us prove that when $c_i = x_i$ for all P_i, then $R = 1$. This is implied by $\sum_{i=1}^{n} x_i y_i = 0$:

$$\sum_{i=1}^{n} x_i y_i = \sum_{i=1}^{n} x_i \left(\sum_{j=1}^{i-1} x_j - \sum_{j=i+1}^{n} x_j \right) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} x_i x_j - \sum_{j=2}^{n} \sum_{i=1}^{j-1} x_j x_i = 0.$$

Each participant P_i must prove he knows the private x_i and c_i without revealing their values. One of the solutions is to use a digital signature, such as the Schnorr signature, where P_i computes:

- a private value $k_i \in_R \mathbb{Z}_q$,
- a public value $e_i = H(g^{x_i}, g^{k_i}, i)$, where H is a public secure hash function,
- a public value $s_i = k_i - x_i e_i$.

Any other P_j can test the knowledge of x_i by testing $e_i \stackrel{?}{=} H(g^{x_i}, g^{s_i} (g^{x_i})^{e_i}, i)$.

We will now prove that this protocol is safe under collusion and collision. In this case, a subset of participants that are colluding compromise their private values x_i in order to trace the role of others in the network. A full collusion against a participant P_i is a collusion where all other $n-1$ participants are colluding. If such a collusion happens, the participant P_i has no point in remaining in the network. We concentrate on partial collusions, that involve less than $n-1$ participants. We will prove that in a partial collusion of $n-2$ participants (which is the worst case scenario), the attackers cannot distinguish between the two public $(g^{y_i})^{x_i}$ and $(g^{y_i})^{r_i}$.

Let us prove that y_i is uniformly distributed over $\mathbb{Z}_q$, from the $n-2$ colluders' point of view. If no more than $n-2$ participants collude, there is a participant P_k with $k \neq i$ who is not colluding. His private value x_k is unknown to the colluders and it is uniformly distributed over $\mathbb{Z}_q$. In the Anonymous Veto networks, y_i is comprised of the private values x_j of all P_j where $j \neq i$. Because y_i has a known term (the sum of colluded values) and an unknown term (x_k) uniformly distributed over $\mathbb{Z}_q$, then y_i is in itself random, due to its uniform distribution over $\mathbb{Z}_q$.

Under the Decision Diffie Hellman assumption, $n-2$ attackers in partial collusion against P_i cannot distinguish between $(g^{y_i})^{x_i}$ and $(g^{y_i})^{r_i}$. The private value x_i is chosen by P_i randomly from $\mathbb{Z}_q$ and the private value y_i is also known to act like a random value. Therefore, even though colluders know the public g^{x_i} and g^{y_i} (by multiplying public values g_{x_l} where $l \neq i$), they cannot distinguish between the value of $(g^{y_i})^{x_i}$ and a random value in G like $(g^{y_i})^{r_i}$.

Another possible attack is the collision of participants. In this case, the protocol remains resistant because by selecting a sufficiently large q (the authors in [4, s. 2.3] suggest a 1024-bit prime q), the probability of having $R = 1$ when more participants send $(g^{y_i})^{c_i}$ with $c_i \neq x_i$ is very small. If collision is seen as a form of disruption, i.e. the participant P_n sends a disrupting value

$$z^{-1} = (g^{y_n})^{c_n} = \left(\prod_{i=1}^{n-1} (g^{y_k})^{c_k} \right)^{-1} \text{ so that } R = zz^{-1} = 1,$$

he will not be able to find the discrete logarithm c_n in $(g^{y_n})^{c_n} = z^{-1}$.

Conclusion

Message transmission with untraceable source is paramount in practice. The Dining Cryptographers approach, published by David Chaum in [1], is an unconditionally or cryptographically secure solution to this problem, depending on whether it is based on one-time-use keys or public keys, respectively. This solution came with its own limitations, which required more general solutions, such as the Anonymous Veto Protocol Networks.

Chapter 3

Mix Networks

Electronic mail systems required the communication of participants whose identities and messages were hidden, while the underlying telecommunication system was unsecure and a universal trusted authority did not exist. The first Mix solution for this problem was published by David Chaum in [6].

This chapter is organized as follows: section 3.1 describes the anatomy of a single mix and extends it to a network of mixes; sections 3.2, 3.3 and 3.4 analyze the subproblems of sender, recipient and two way anonymity respectively, and provide enriched mix network solutions for them; 3.5 provides a further solution for not minimizing the size of messages that pass through a mix network.

3.1 Mix Anatomy

The main idea is to create an intermediary between the sender and recipient of any message, named mix. It must 'mix' the messages so that nobody will be able to determine correspondences between the input and the output of the intermediary, nor to forge them without knowing the necessary parameter values for unsealing them.

We will denote the pair of keys in a public key cryptosystem by k and k^{-1}, where k is the public key and k^{-1} is the private key. The mapping of the cryptosystem function of a message x with the key k will be $y = k(x)$. As for any public key cryptosystem, by mapping the function again with the key k^{-1} to y, we get back the message x in clear. If one finds $k(x_1) = k_(x_2)$, he will be able to tell that $x_1 = x_2$, because the mapping is injective (see [2, p. 269]). This threat can be overcome by attaching a large random string of bits r to x before encrypting it. The seal of x with the key k will be denoted $k(r, x)$. To sign a message y, a public large bit string c is chosen and $x = k^{-1}(c, y)$ is computed. Anyone can verify $c, y \stackrel{?}{=} k(x)$.

In the proposed solution, the mix has a key pair (k, k^{-1}) in order to handle encrypted messages addressed to it. Each participant has an address $j \in \{a, b, ...\}$ and a key pair (k_j, k_j^{-1}), also to handle encrypted messages addressed to him. Suppose participant at the address a wants to send the message m_{ab} to the participant at the address b and participant at the address c wants to send message m_{ca} to the participant at a. The steps each message passes through are:

- Participant a seals m_{ab} with the public key k_b of the recipient from the address b, appends the address b to the result and seals everything with the public key k.
  He then sends $k(r_1, k_b(r_2, m_{ab}))$ to the mix.

- Participant c seals m_{ca} with the public key k_a of the recipient a, appends the address a to the result and seals everything with the public key k of the mix.
  He then sends $k(r_3, k_a(r_4, m_{ca}))$ to the mix.

- The mix receives the input $I_1 = k(r_1, k_b(r_2, m_{ab}), b)$. By decrypting I_1 with its private key k^{-1} and removing the random bits r_1, it obtains $k_b(r_2, m_{ab})$ and b.
- The mix receives the input $I_2 = k(r_3, k_a(r_4, m_{ca}), a)$. By decrypting I_2 with its private key k^{-1} and removing the random bits r_3, it obtains $k_a(r_4, m_{ca})$ and a.
- The mix waited for inputs to gather in order to shuffle them.
  For instance, it may change the order of the inputs when their contents are output.
- The mix sends the output $O_1 = k_a(r_4, m_{ca})$ to the address a.
  The transformation of I_2 into O_1 made by the mix will be denoted by $I_2 \to O_1$.
- The mix sends the output $O_2 = k_b(r_2, m_{ab})$ to the address b.
  The transformation of I_1 into O_2 made by the mix will be denoted by $I_1 \to O_2$.
- The participant at a receives $k_a(r_4, m_{ca})$.
  He decrypts the input with his private key k_a^{-1}, removes r_4 and obtains m_{ca}.
- The participant at b receives $k_b(r_2, m_{ab})$.
  He decrypts the input with his private key k_b^{-1}, removes r_2 and obtains m_{ab}.

The paths of the messages are depicted in the figure 3.1 from below:

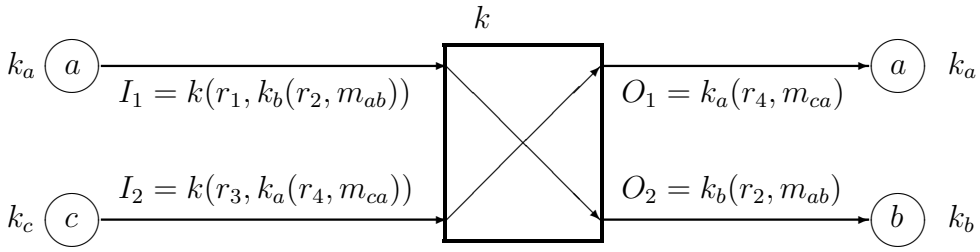


Figure 3.1: The mix receives two messages, shuffles them and sends them further.

The mix hides the correspondences between I_i and O_i by outputting the items of the same size in lexicographically ordered batches. For n randomly valued sealed inputs, the probability of having $I_i \to O_i$ for some $i \in \{1, 2, ..., n\}$ may be as small as n^{-1}.

If items are repeated in the input, the redundant copies will be removed before being output in order to not have a correspondence between equal inputs and equal outputs. In this case, the mix may retain records from previous batches. If the random bit string also includes a time stamp useful for one batch only or the public key k changes, previous records will not be useful any longer. The mix is able to send signed receipts for messages it receives from the participants. If the mix sends to all participants the receipt $z = k^{-1}(c, y = k(r_1, x = k_a(r_0, m), a))$, where c is a constant large bit string, only the sender participant knows r_1, x and a. Any other participant faces with the hard problems of finding k^{-1} or some values whose mapping with k yields y, in order to have $k(z) = c, y$.

In practice, it is safer to use more than one mix in a network of a topology similar to the graph P_n. A message is sent to a cascade of mixes in a similar way to sending it through one mix only. Each mix $i \in \{1, 2, ..., n-1, n\}$ has a key pair (k_i, k_i^{-1}) and an internal, secret permutation π_i.

The permutation of the messages is $\pi = \pi_n \circ \pi_{n-1} \circ ... \circ \pi_2 \circ \pi_1$, where each of the $m!$ permutations of a mix appear with equal probabilities. Note that this still holds true in the original model: a sorting result permutation for equally sized messages stands for the unknown internal permutation, as the result of the encryption of a text cannot be lexicographically linked to the text without decrypting it with the secret key.

Suppose all n mixes but mix n_0 are corrupt, in the way that they provide their internal permutation π_i. Suppose an attacker assigns the identity e to the order of messages from the output of the final mix. If he states that the order of the messages from the input of the initial mix is σ, he must find that $e \stackrel{?}{=} \pi \circ \sigma$, which consists of computing π. Stating that $e = \pi \circ \sigma$ and being correct has the same probability as guessing correctly the value of the unknown π_{n_0}. As all permutations may appear as π_{n_0} with equal probabilities, the chance of being correct is $(m!)^{-1}$:

$$\begin{aligned} e = \pi \circ \sigma \quad & \Leftrightarrow e = (\pi_{n,n_0+1} = \pi_n \circ ... \circ \pi_{n_0+1}) \circ \pi_{n_0} \circ (\pi_{n_0-1,1} = \pi_{n_0-1} \circ ... \circ \pi_1) \circ \sigma \\ & \Leftrightarrow e = \pi_{n,n_0+1} \circ \pi_{n_0} \circ (\tau = \pi_{n_0-1,1} \circ \sigma) \\ & \Leftrightarrow \pi_{n_0} = (\rho = \pi_{n,n_0+1}^{-1} \circ \tau^{-1}) \\ & \Leftrightarrow \pi_{n_0} = \rho. \end{aligned}$$

The knowledge of the sender of a specific message would be as hard to obtain as the knowledge of all messages, since (in the original model) the new position of the output for a known input is also hidden in the unknown π_{n_0}. It is also important to note that $\pi_{n_0}^{-1}$ is also unknown, because this allows a participant to not being linked with the messages that enter the network by analyzing the messages he receives. From now on, we will focus on mix networks with $n > 1$ mixes, on a structure based on [6] and [7].

3.2 Sender anonymity

Suppose each participant i in a set of m participants sends a message m_i in a network with n cascading mixes to be received by another participant whose address is of the form b_i. If a participant sends more than one message, he will be simply seen as two participants. The contents of a message must remain unknown to everyone but its sender and recipient. The sender also wants to remain anonymous to everyone but himself:

- Each sender i seals m_i with k_{b_i} and appends b_i.
  He obtains $M_i^{(n+1)} = k_{b_i}(r_i^{(n+1)}, m_i), b_i$.

- He seals $M_i^{(n+1)}$ with the public key k_n of the n^{th} mix.
  He obtains $M_i^{(n)} = k_n(r_i^{(n)}, M_i^{(n+1)}) = k_n(r_i^{(n)}, k_{b_i}(r_i^{(n+1)}, m_i), b_i)$.

- He seals $M_i^{(n)}$ with the public key k_{n-1} of the $n-1^{th}$ mix.
  He obtains $M_i^{(n-1)} = k_{n-1}(r_i^{(n-1)}, M_i^{(n)}) = k_{n-1}(r_i^{(n-1)}, k_n(r_i^{(n)}, k_{b_i}(r_i^{(n+1)}, m_i), b_i))$.

...

- He seals $M_i^{(3)}$ with the public key k_2 of the 2^{nd} mix.
  He obtains $M_i^{(2)} = k_2(r_i^{(2)}, M_i^{(3)}) = k_2(r_i^{(2)}, ..., k_n(r_i^{(n)}, k_{b_i}(r_i^{(n+1)}, m_i), b_i) \cdots)$.

- He seals $M_i^{(2)}$ with the public key k_1 of the 1^{st} mix and sends $M_i^{(1)}$ to the 1^{st} mix.
  He computed $M_i^{(1)} = k_1(r_i^{(1)}, M_i^{(2)}) = k_1(r_i^{(1)}, ..., k_n(r_i^{(n)}, k_{b_i}(r_i^{(n+1)}, m_i), b_i) \cdots)$.

- The 1^{st} mix receives messages of the form $M_i^{(1)}$ from various senders. By decrypting them with its private key k_1^{-1} and by removing the random bits $r_i^{(1)}$ in each, it obtains messages of the form $M_i^{(2)}$ and sends them mixed to the 2^{nd} mix.

- The 2^{nd} mix receives messages of the form $M_i^{(2)}$ from the 1^{st} mix. By decrypting them with its private key k_2^{-1} and by removing the random bits $r_i^{(2)}$ in each, it obtains messages of the form $M_i^{(3)}$ and sends them mixed to the 3^{rd} mix.

...

- The $n-1^{th}$ mix receives messages of the form $M_i^{(n-1)}$ from the $n-2^{th}$ mix. By decrypting them with its private key k_{n-1}^{-1} and by removing the random bits $r_i^{(n-1)}$ in each, it obtains messages of the form $M_i^{(n)}$ and sends them in mixed order to the n^{th} mix.

- The n^{th} mix receives messages of the form $M_i^{(n)}$ from the $n-1^{th}$ mix. By decrypting them with its private key k_n^{-1} and by removing the random bits $r_i^{(n)}$ in each, it obtains messages of the form $M_i^{(n+1)}, b_i$. It then sends each $M_i^{(n+1)}$ to its b_i.

- The recipient at the address b_i receives some messages of the form $M_i^{(n+1)}$ from the n^{th} mix, decrypts them with his private key $k_{b_i}^{-1}$, removes $r_i^{(n+1)}$ and gets the messages m_i.

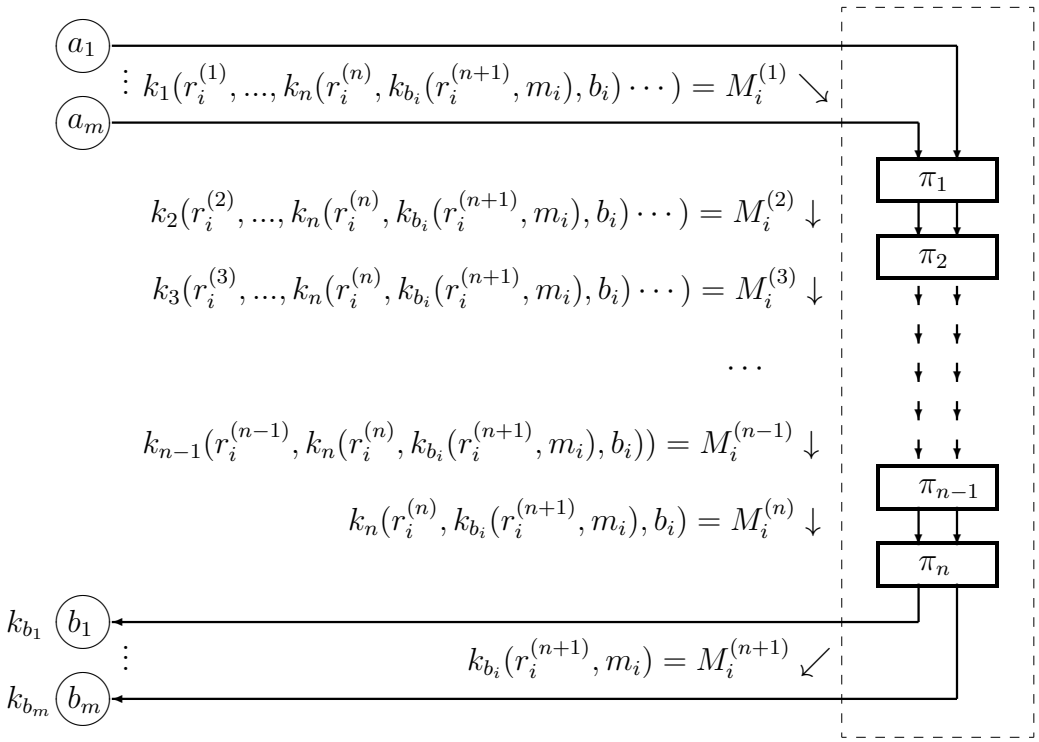


Figure 3.2: A mix network with n mixes where the traces of the m message are lost.

The figure 3.2 shows the path of a message through a network with more than 4 cascading mixes. Each mix applies an unknown internal permutation to the messages before retransmitting them to the previous mix, with the exception of the 1^{st} mix, which sends the messages to the sealed addresses it unsealed with its private key.

When the number of mixes in a real-world network increases, the costs of cascading networks in which all messages travel the same path from 1 to n may increase substantially. Other mix network variant consists of a graph of mixes from which the sender chooses a trustworthy sequence of mixes. We will present a solution based on those presented in [7], where this issue is overcome. All mixes have the same functionality, as opposed to the previous model, where the n^{th} mix had the extra responsibility of sending each message to a variable address specified in the received message. The steps from above still apply, but with some minor changes:

- The sender i computes $M_i^{(j)}$ $(j \in \{1, 2, ..., n-2, n-1\})$ as $k_j(r_i^{(j)}, M_i^{(j+1)}, a_{j+1})$, where a_{j+1} is the address of the $j+1^{th}$ mix. He sends $M_i^{(1)}$ to the first mix as usual.

- A j^{th} mix receives a message of the form $M_i^{(j)}$ from an address of the form a_{j-1}. By decrypting it with its private key k_j^{-1} and by removing the random bits $r_i^{(j)}$, it obtains a message of the form $M_i^{(j+1)}, a_{j+1}$. It then sends $M_i^{(j+1)}$ to the address a_{j+1}. Note that for the n^{th} mix, the address b_i stands for a_{n+1}.

3.3 Recipient anonymity

The previous model allowed a participant to send a message anonymously to another participant whose address is known. The mix networks also allow for a participant a to send a message to a participant b without knowing his real address. This can be accomplished by the future recipient b with the help of a new mix network model:

- The participant at the address b sends an encrypted return address to the mix network as above, such that the participant from the address a will receive $k_n(r_n, b), k_b$. The return address $k_n(r_n, b)$ will be sent back to the address b by the holder of a.

- The participant at the address a encrypts his message m with the key k_b provided by b and encrypts this quantity with the public keys of the n mixes. He sends $M_1 = k_1(r_1, k_2(r_2, ..., k_n(r_n, b) \cdots)), k_b(r_b, m)$ to the 1^{st} mix.

- The 1^{st} mix receives M_1, decrypts the first part with its key k_1^{-1} and further encrypts the second part with r_1. It then sends $M_2 = k_2(r_2, k_3(r_3, ..., k_n(r_n, b) \cdot \cdot\cdot)), r_1(k_b(r_b, m))$ to the 2^{nd} mix.

...

- The $n-1^{th}$ mix receives M_{n-1}, decrypts the first part with its key k_{n-1}^{-1} and further encrypts the second part with r_{n-1}. It then sends $M_n = k_n(r_n, b), r_{n-1}(r_{n-2} \cdots r_1(k_b(r_b, m)) \cdots)$ to the n^{th} mix.

- The n^{th} mix receives M_n, decrypts the first part with its key k_n^{-1} and it sends $M_{n+1} = r_n(r_{n-1} \cdots r_1(k_b(r_b, m)) \cdots)$ to the address b.

- The participant at the address b decrypts M_{n+1} by applying successively the keys r_n^{-1}, r_{n-1}^{-1}, ..., r_2^{-1}, r_1^{-1}, k_b^{-1} to it and then he obtains the plain message m from the participant at the address a.

A variant of this model in which the participant at the address b selects the sequence of mixes is also possible (see [7]). This can be achieved by modifing the protocol from above. For simplicity, we may consider that the address b is denoted as a_{n+1} and that a_j is the address of the j^{th} mix chosen by the participant at b:

- The participant at the address b selects the sequence of mixes and encodes his selection in a parameter denoted s_{n+1}. He then computes, for each mix in the cascade it wants the participant at the address a to respond through, the value $s_j = k_j(r_j, s_{j+1}, a_{j+1})$, for all $j \in \{1, 2, ..., n-1, n\}$. He then builds an encryption of k_b, a_1, s_1 so that it will be transmitted safely through a chosen mix network.

- The participant at the address a receives the message and, after decrypting it with his own private key, he gets k_b, a_1, s_1. He uses k_b to encrypt his message m and sends to the mix at the address a_1 the message $M_1 = k_b(m)$ and s_1.

- The j^{th} mix network receives M_j and s_j. It uses k_j to decrypt s_j and it learns r_j, s_{j+1}, a_{j+1}. It then sends to the mix at a_{j+1} the message $M_{j+1} = r_j(M_j)$ and s_{j+1}.

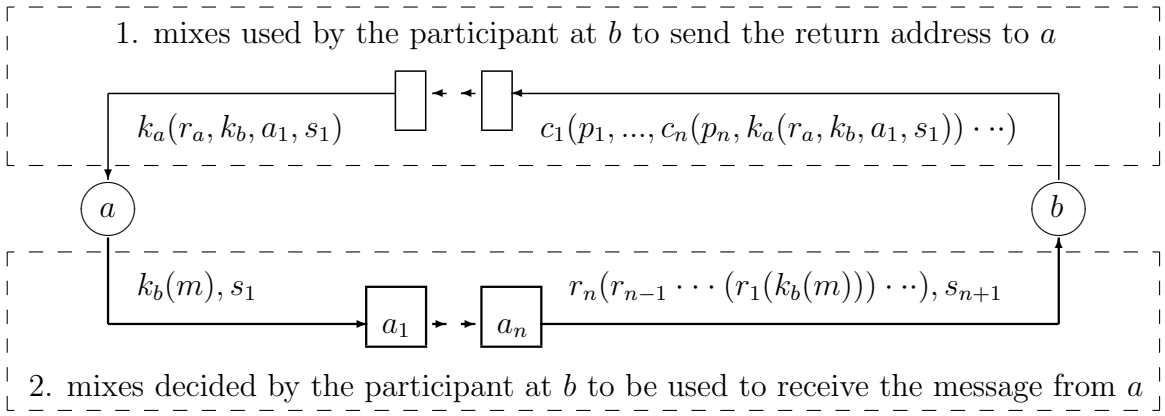


Figure 3.3: after sending a return address to a, b receives m behind a specified mix network

- The participant at the address b receives M_{n+1} and s_{j+1}. It decrypts $M_{n+1} = r_n(r_{n-1} \cdots (r_1(k_b(m))) \cdots)$ by using the other received value s_{j+1} which encoded the sequence of mixes he had chosen in the first place for a to send him a message through. He decrypts s_{j+1} by using the keys $r_n^{-1}, r_{n-1}^{-1}, ..., r_1^{-1}, k_b^{-1}$ and he gets m.

Untraceable return addresses allow an anonymous sender to receive a receipt from the mix network. The receipt attests the correct processing of his message to the output of the last mix. It includes the message and the address of the recipient. The receipt should also be signed by each mix, so that its issuers can be verified by the sender (for instance, after the sender decided the order of mixes for his message to pass through, he can verify that his message was indeed transmitted on that path).

3.4 Two Way Anonymity

The mix networks are able to hide both the sender and the recipient of the message. It is possible to create a model in which participants a and b both send and receive messages anonymously. The idea is to hide for each participant its addresses for sending and receiving messages behind a mix network:

- The participants at the addresses a and b register anonymous return addresses R_a and R_b respectively, by sending them through mix networks to a public news group g. Each address contains a sequence of mixes in the network, so that the messages received by its publisher will be transmitted through them.

- The participant at the address a sends a message to the participant at R_b by choosing a sequence of mixes in the network. The mixes from a to R_b provide sender anonymity for a and the sequence from R_b to b provides recipient anonymity for b.

- The participant at b responds to R_a by choosing a sequence of mixes of his own. His real address is lost in the sequence of mixes between b and R_a, and the address of the recipient a is lost in the mixes between R_a and a.

The figure 3.4 shows the paths of a message from a to b and its response.
For the same reason why redundant messages are deleted from a mix (see [6, p. 3]), a and b should change their initial return addresses R_a and R_b.

3.5 Fixed Sized Blocks

Even though the mix networks are able to hide both the sender and the recipient of the message, the size of the message decreases along its path. Intuitively, if an output message is 'larger' or 'smaller', its output mix will be 'closer' to the sender or the recipient,

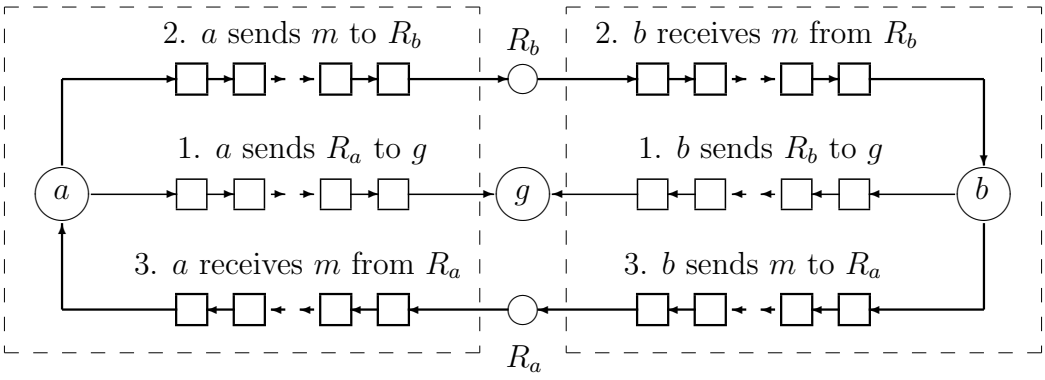


Figure 3.4: a and b lose their traces behind their own sequences of mixes

respectively. In a real-world mix network, trusted chosen mixes can decrypt and discard dummy messages and send meaningful fixed sized messages to the participant.

A mix network protocol, based on (see [6, p. 4]), that allows mix sequence selection by the sender and pads the messages with data in order to not have them shrunk along the path is presented below. Suppose the fixed size of all messages is of l blocks.

We will show the content of a message on rows. The first row shows the info addressed to the receiving mix, the second row shows the info addressed to the next mixes along the path, the third row shows the encrypted splitted message, and an optional fourth row shows the paddings applied in order to preserve the overall length of l blocks.

The participant at the address a_0 plans to send to the participant at a_{n+1} a long message m through a sequence of n mixes with the addresses a_1, a_2, ..., a_n. The path of his message in the mix network is described below:

- The participant at a_0 seals the message m with the public key of a_{n+1}. He then splits the seal into $l-n$ blocks and he obtains $[m_1]$, $[m_2]$, ..., $[m_{l-n-1}]$, $[m_{l-n}]$, where $[\cdot]$ is a block, such that $k_{n+1}(r_{n+1}, m) = [m_1][m_2]...[m_{l-n-1}][m_{l-n}]$. We will also denote a block that does not exist with $[\phi]$, to show on a specific row that blocks of its meaning do not exist:

- The participant at a_0 sends to the 1^{st} mix M_1 of l blocks:

$$\begin{aligned} M_1 =&[k_1(r_1, a_2)] \\ &+[r_1^{-1}(k_2(r_2, a_3))]...[r_1^{-1}(r_2^{-1} \cdots (r_{n-2}^{-1}(r_{n-1}^{-1}(k_n(r_n, a_{n+1})))) \cdots)] \\ &+[r_1^{-1}(r_2^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_1))) \cdots)]...[r_1^{-1}(r_2^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_{l-n}))) \cdots)].+ \quad [\phi]. \end{aligned}$$

- The 1^{st} mix receives M_1, decrypts the first block with its secret key k_1^{-1} and learns r_1 and a_2. He then applies r_1 to the rest of the $l-1$ blocks. He also encrypts a junk block j_1 after the message blocks. He then sends M_2 to the mix at a_2:

$$\begin{aligned} M_2 =&[k_2(r_2, a_3)] \\ &+[r_2^{-1}(k_3(r_3, a_4))]...[r_2^{-1}(r_3^{-1} \cdots (r_{n-2}^{-1}(r_{n-1}^{-1}(k_n(r_n, a_{n+1})))) \cdots)] \\ &+[r_2^{-1}(r_3^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_1))) \cdots)]...[r_2^{-1}(r_3^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_{l-n}))) \cdots)] \\ &+[r_1(j_1)]. \end{aligned}$$

...

- The i^{th} mix receives M_i:

$$\begin{aligned} M_i =&[k_i(r_i, a_{i+1})] \\ &+[r_i^{-1}(k_{i+1}(r_{i+1}, a_{i+2}))]...[r_i^{-1}(r_{i+1}^{-1} \cdots (r_{n-2}^{-1}(r_{n-1}^{-1}(k_n(r_n, a_{n+1})))) \cdots)] \\ &+[r_i^{-1}(r_{i+1}^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_1))) \cdots)]...[r_i^{-1}(r_{i+1}^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_{l-n}))) \cdots)] \\ &+[r_{i-1}(r_{i-2} \cdots (r_2(r_1(j_1))) \cdots)][r_{i-1}(r_{i-2} \cdots (r_3(r_2(j_2))) \cdots)]...[r_{i-1}(j_{i-1})]. \end{aligned}$$

He decrypts the first block with k_i^{-1} and gets r_i and a_{i+1}. He applies r_i to the rest of blocks and he appends its encrypted junk j_i. He then sends M_{i+1} to a_{i+1}:

$$\begin{aligned} M_{i+1} =& [k_{i+1}(r_{i+1}, a_{i+2})] \\ &+[r_{i+1}^{-1}(k_{i+2}(r_{i+2}, a_{i+3}))]...[r_{i+1}^{-1}(r_{i+2}^{-1} \cdots (r_{n-2}^{-1}(r_{n-1}^{-1}(k_n(r_n, a_{n+1})))) \cdots)] \\ &+[r_{i+1}^{-1}(r_{i+2}^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_1))) \cdots)]...[r_{i+1}^{-1}(r_{i+2}^{-1} \cdots (r_{n-1}^{-1}(r_n^{-1}(m_{l-n}))) \cdots)] \\ &+[r_i(r_{i-1} \cdots (r_2(r_1(j_1))) \cdots)][r_i(r_{i-1} \cdots (r_2(j_2)) \cdots)]...[r_i(j_i)]. \end{aligned}$$

...

- The n^{th} mix receives M_n. He decrypts the first block with k_n^{-1} and gets r_n and a_{n+1}. He applies r_n to the rest of blocks and he appends its own encrypted junk j_n. He then sends M_{n+1} to the recipient at the address a_{n+1}:

$$\begin{aligned} M_{n+1} =& [\phi] \\ &+[\phi] \\ &+[m_1]...[m_{l-n}] \\ &+[r_n(r_{n-1} \cdots (r_2(r_1(j_1))) \cdots)][r_n(r_{n-1} \cdots (r_2(j_2)) \cdots)]...[r_n(j_n)]. \end{aligned}$$

- The recipient receives M_{n+1} and gets $[m_1]...[m_{l-n}] = k_{n+1}(r_{n+1}, m)$. By decrypting this quantity with his private key k_{n+1}^{-1}, he obtains the message m in clear.

Conclusion

Some improvements over the original protocol, that ensure sender, receiver or two way anonymity, and also padding to fixed block were described. These techniques are also useful for voting applications, where applicants with untraceable pseudonyms sign their votes so that anyone will be able to count them. In the next chapter we will survey the techniques used by a mix server to handle incoming messages towards the next hop in the network.

Chapter 4

Mix Flush Algorithms

Each mix server receives messages from the network, processes them and sends them further. We may say that all messages that reside in the mix at a moment are a batch. As seen before, the processing of batches of messages mainly consists of two tasks. The first is solved at message level and it consists of gathering metadata, such as the decryption algorithm and address of the next recipient after decryption takes place. The second is at the batch level and it consists of applying a strategy to send some messages from the batch to the next server. The strategy used for this task is known in the literature as the flushing algorithm. Various flushing algorithms were analyzed in [8], together with some successful attacks that speculate possible patterns in their dynamic parameters.

Known attacks rely on the assumption of an active adeversary able to trace and also remove and insert an arbitrary number of messages from the input and output of a mix network in constant time. The mixes also have a limited physical memory to retain messages and a limited bandwidth to receive and send messages at a certain moment. If not stated otherwise, the anonymity is measured from the point of view of a passive attacker, rather than the active intruder in question.

We also propose a different mix flush simple algorithm which overcomes the small but nonzero probability that a message remains in the mix for an undetermined number of rounds (which is present for some of the above presented techniques). Our flush protocol provides a great minimum anonymity, which is comparable to the anonymity provided by the Threshold Pool Mix algorithm, but only good maximum anonymity. We named our algorithm Round Robin Pool Queue mixing.

This chapter consists of the following sections: sections 4.1, 4.2 and 4.3 analyze flush algorithms for mixes with no memory; sections 4.4, 4.5 and 4.6 present flushing procedures for mixes with (pool) memory; sections 4.7, 4.8 and 4.9 survey different approaches to mix flushing; section 4.10 describes our simple algorithm.

4.1 Threshold Flush Mixes

The simplest flushing strategy a mix can apply, which is also discussed in the original paper of Chaum, is to wait for a fixed number of n messages to gather before it decrypts and sends them all to thir next recipients in a random order. The minimum delay of a message is ϵ - the message appears when there are $n-1$ messages in the mix - and the maximum delay is ∞, if there will never be a total of n input messages in the mix so it is able to flush them. The anonymity of a message is $1/n$: the mix always flushes exactly n messages. By assuming r messages enter at a certain moment, the average delay is $(n-1)/2r$. This strategy is subject to an attack in which an intruder gains control of the mix input channel in order to trace the whereabouts of a legitimate message m_i. We suppose the message m_i is the i^{th} message in a total of n messages needed for the mix to flush. The steps of this attack are as follows:

1. Start the attack by retaining the message m_i and further retaining and blocking all input messages from entering the mix server, but send $n-i+1$ dummy messages, until

the mix flushes. These dummy messages must be indistinguishable from legitimate messages from the perspective of any entity but the intruder.

2. A number of $i-1$ legitimate messages were correctly sent to their recipients with this flush, while the rest of dummy (yet simple to recognize as own) messages are taken from the output channel by the intruder.

3. Still block and retain all input, but send the traced message m_i with other $n-1$ dummy messages, so that the mix server is forced to flush again.

4. Analyze all output messages and discard all dummy messages. The remaining message is the message processed from m_i. By letting the message follow its path, i messages are sent to the destination correctly by the mix.

5. Send to the mix all retained messages that came to it.

Note that this attack remains untraced by the sender and recipient of each message, as the intruder retains input messages to let them be sent after the attack is successful. As all other $n-1$ messages are inserted by the intruder, the anonymity of m_i from the point of view of the attacker is 0, and the success rate of this attack is 1. This attack is also known as the $(n-1)$ attack. In this version, the attack needs between $n-1$ and $2(n-1)$ dummy messages to be sent to the mix. However, if the intruder is able to insert $n-1$ messages in between each two messages m_k and m_{k+1}, each message will be traced. This attack can also be mounted on more than one mix server: by knowing all addresses of recipients of the messages, the attack target can simply include the next mix servers in the network.

4.2 Timed Flush Mixes

Another flushing strategy is to wait for a fixed time of t units of time before the mix decrypts and sends all input messages to thir next recipients in a random order. The minimum delay of a message is ϵ - the message appears exactly before the mix flushes, while the maximum delay is only $t-\epsilon$, if it appears immediately after. This is an advantage over threshold flushing. However, there is also a disadvantage in the case of light traffic: smaller to trivial anonymity sets of messages may appear. This affects the minimum anonymity of a message, which may decrease to 0. The maximum anonymity is only bounded by the number of messages the mix is able to retain. The mean anonymity set size is, therefore, rt.

This algorithm is equvalent to the previous. This means that an equivalent attack can be deployed to destroy the anonymity of the message m_i. A difference between this attack and the previous is that there are no necessary dummy messages for the attack to succeed:

1. Retain the message m_i and further retain and block all input messages from entering the mix server, until t units of time have passed, and the mix flushes. Note that no message is flushed by mix server at this moment.

2. Still block and retain all input, but send the traced message m_i to the mix and wait for another t units of time, so that the mix server will flush again.

3. Analyze the single output message and let it follow its path.

4. Send to the mix all retained messages that came to it.

This attack still remains untraced by the sender and recipient of each message. As no other message are inserted by the intruder, the attacker-wise anonymity of m_i is 0, and the success rate of this attack is still 1. This attack is also referred as the *trickle* attack. The case when the intruder is able to delay each message for t units of time, each message will be traced, possibly to the extent of the entire mix network.

4.3 Threshold or/and Timed Flush Mixes

There are also hybrid strategies for mix servers.
One of them is to wait for t units of time or until n messages enter the mix before a full flush. The minimum delay of message forwarding is ϵ, and the maximum is $t - \epsilon$. The advantage over the previous strategies is that the mix is not forced to wait until t units of time to pass before it flushes. If traffic is high, this approach is well suited (the messages will be very quickly forwarded), but low traffic may still allow trivial anonymity sets to appear. This technique is still not resilient to attacks. An even stronger mixed attack can be performed: the intruder is able to select the attack strategy, as follows:

- if the attacker is unwilling to wait for at most $2t$ units of time, it will apply the $n-1$ attack,
- if he does not want to send dummy messages, but it is able to wait for at most t units of time, it will apply the *trickle* attack,
- apply a reunion of the two attacks, but with less effort (the mix is almost at a t moment in time or it has almost $n-1$ messages inside).

Another hybrid strategy is to flush all messages every t units of time, but only if n or more messages have entered. The lower limit of at least n messages yields to the disadvantage of a maximum delay of ∞, but the advantage is that the anonymity of flushed messages is at least $1/n$ and at most the capacity of the mix. There also exists an attack, which is dual to the attack for Threshold or Timed Mixes. The required time is in the interval $[\epsilon, 2t - \epsilon]$ and the amount of necessary dummy messages is between $n-1$ and $2(n-1)$.

4.4 Threshold Pool Mixes

Threshold pool mixes use a probabilistic approach. When $n + N$ messages have entered the mix, it flushes n of them chosen uniformly at random and it keeps the rest of N messages inside. It then waits for other n messages to enter and it repeats the procedure. In this case, we say that n is the threshold and N is the pool, as oposed to previous flushing algorithms, where the threshold is the number of messages needed in the mix in order to flush.

Even though there is a substantial number of input messages, and the mix flushes often, a message m may never leave the mix. This yields to a minimum delay of ϵ and a maximum delay of ∞. The average delay of a message when the rate of incoming messages is r is $(1+\frac{N}{N+n})\frac{n}{r}$ seconds. The minimum anonymity in this case is at least n in the realistic case when all messages have different senders. However, if some messages come indeed from the same sender, their stay in the mix may be longer than one round, which increases their anonymity. Therefore, we may say the anonymity provided by this algorithm is often stronger than the minimum anonymity of threshold mixes. The maximum anonymity, discussed in [10], $-(1-\frac{N}{n})\log(n+N)+\frac{N}{n}\log(N)$ is achieved when a message entered the mix before the first flush and all other messages came from different senders.

The unknown number of rounds until a message m is flushed obliges the attacker to use more time and space resources for message management, until a target message is flushed. However, the attack is exact: if the attacker sees m at the output, the anonymity of m is compromised.

1. The attacker blocks the input of the mix and delays all good messages, including the target message m, from entering the mix. It sends dummy messages to the mix instead. As in the strategies before, these messages must appear as legitimate messages to every entity but the attacker.

2. The flooded mix flushes. If k messages of the illegitimate messages are not flushed, there are $N-k$ legitimate messages left in the mix. The attacker continues to send messages to the input of the mix until all remaining n messages are his. This step is undefined in time, as a legitimate message may stay in the mix forever.

3. The attacker sends the message m to the mix. Now there are n illegitimate messages and one legitimate message, namely m.

4. He floods the mix with bad messages until it flushes. He then checks if m is among them. If not, he follows this step once again. Note that if m stays in the mix and illegitimate messages go by, the time of this step also lasts forever.

As the flushed messages are chosen uniformly at random, the flush of a message m_1 does not depend on the flush of another message m_2. The disadvantage of allowing a message to stay in the mix forever makes the $n-1$ attack uncertain and increases the necessary amount of resources. In [12] an analysis of the probability distribution for the number of rounds required to complete an $(n-1)$ attack is carried out. When round r begins, the mix contains a message pool P_r which consists of all messages that remained in the mix from the previous round. For this particular type of mixes, for any round r, $|P_r| = N$. The batch set of messages that enter the mix in round r is denoted by B_r. Let $S_r = (P_r \cup B_r)$ be the selection set, i.e. the set of messages in the mix in round r from which messages to be flushed are to be selected uniformly at random. The mix selects F_r messages, and the pool for the next round is $P_{r+1} = S_r - F_r$.

For this type of mix, it is necessary that $B_r = n$ for the r^{th} round to continue with the flushing procedure on the messages P_r and B_r. If it is not the case, in the next round $r+1$, simply $P_{r+1} = P_r$ and $B_{r+1} = B_r \cup B_r^0$, where B_r^0 are the incoming messages between rounds r and $r+1$. Suppose that for the first round, P_1 consists of some dummy messages in the set B_0 with $|B_0| = N$. For this type of mix, the flushed set F_r is also of a fixed size n.

The $(n-1)$ attack manipulates the mix for rounds 1, 2, ..., $r-2$, $r-1$, when F_{r-1} messages were flushed. Suppose the attacker wishes to flush the messages left in the mix, namely P_r. The attacker blocks all legitimate incoming messages and submits k batches B_r, B_{r+1}, ..., B_{r+k-2}, B_{r+k-1} of sizes $|F_{r+j}| = |F_{r-1}| = n$, for any $0 \leq j < k$. In this way, the flush condition $B_{r+j} = n$ is satisfied for each round $r+j$, and the mix is forced to flush.

Denote by $\Delta(P_r)$ the random variable which describes the minimum number of rounds that is necessary for flushing the entire pool set of messages P_r, assuming the conditions from above regarding the sizes of P_{r+j}, B_{r+j}. If $\Delta(P_r) = k \geq 1$, then $P_r \cap P_{r+k} = \emptyset$ and $P_r \cap P_{r+j} \neq \emptyset$, i.e. at least one message $m_i \in P_r$ stays in the mix before round $r+k-1$ has started, but not after round $r+k-1$ has ended by flushing F_{r+k-1}, which contains m_i. The aim is to find $\mathcal{P}(\Delta(P_r) > k)$ for any $k \leq 1$. When $\mathcal{P}(\Delta(P_r) > k) < \epsilon$, the success probability of the attack is $1-\epsilon$. The value of $\mathcal{P}(\Delta(P_r) = k)$ for some $k > 1$ can be computed by difference:

$$\mathcal{P}(\Delta(P_r) = k) = \mathcal{P}(\Delta(P_r) > k-1) - \mathcal{P}(\Delta(P_r) > k).$$

In this case of a Threshold Pool Mix, the set sizes are static throughout all rounds: for each round $r \geq 0$, $|B_r| = |F_r| = n$, $|P_r| = N$ and $|S_r| = n + N$. We will compute the probability of flushing P_r in one round:

$$\mathcal{P}(\Delta(P_r) = 1) = \left(\begin{array}{c} n \\ n-N \end{array} \right) \left(\begin{array}{c} n+N \\ n \end{array} \right)^{-1}, \text{ where } \left(\begin{array}{c} n \\ k \end{array} \right) \text{ is } C_n^k = \frac{n!}{k!(n-k)!}.$$

For $k > 1$, the probability $(P)(\Delta(P_r) > k)$ will be computed by decomposing it into events of each message m_i from P_r, where $i \in \{1, 2, ..., N-1, N\}$. Let $A_{i,k}$ be the event that message $m_i \in P_{r+k}$, i.e. m_i was not flushed from the mix in rounds r, $r+1$, ..., $r+k-1$. Based on the observation that at least one message $m \in P_r$ left inside in round k is enough to have $\Delta(P_r) > k$,

$$\mathcal{P}(\Delta(P_r) > k) = \mathcal{P}(\bigcup_{j=1}^{N} A_{i,k}).$$

For determining this value, the inclusion-exclusion principle was used (a description of this principle can be found in [13]). This principle states that for determining the size of the reunion of sets S_1 to S_n, namely $|S_1 \cup S_2 \cup ... \cup S_{n-1} \cup S_n|$, one must compute the value as follows ($j \geq 1$):

- For indexes $1 \leq i_1 < ... < i_{2j-1} \leq n$, add $|\bigcap_{k=1}^{2j-1} S_{i_k}|$,
- For indexes $1 \leq i_1 < ... < i_{2j} \leq n$, subtract $|\bigcap_{k=1}^{2j} S_{i_k}|$,

We obtain

$$S = |\bigcup_{j=1}^{n} S_j| = \sum_{j=1}^{n} |\bigcap_{k=1}^{j} S_{i_k}|(-1)^{j+1}.$$

We will prove that

$$\mathcal{P}(\Delta(P_r) > k) = \left(\begin{array}{c} n+N \\ n \end{array} \right)^{-k} \sum_{j=1}^{N} (-1)^{j+1} \left(\begin{array}{c} N \\ j \end{array} \right) \left(\begin{array}{c} N+n-j \\ n \end{array} \right)^{k}.$$

The probability of having retained in the mix at least the messages m_1, m_2, ..., m_j is

$$\mathcal{P}(A_{1,1}A_{2,1}...A_{j-1,1}Aj, 1) = \left(\begin{array}{c} N+n-j \\ n \end{array} \right) \left(\begin{array}{c} n+N \\ n \end{array} \right)^{-1},$$

which is independent of the indexes 1, 2, ..., $j-1$, j, but only of j. Therefore, for any $1 \leq i_1 < i_2 < ... < i_{j-1} < i_j \leq N$,

$$\mathcal{P}(A_{1,1}A_{2,1}...A_{j-1,1}Aj, 1) = \mathcal{P}(A_{i_1,1}A_{i_2,1}...A_{i_{j-1},1}Ai_j, 1).$$

The flush set for each round is chosen uniformly and independently at random, which yields to the equality

$$\begin{aligned} \mathcal{P}(A_{i_1,k}A_{i_2,k}...A_{i_{j-1},k}Ai_j, k) &= (\mathcal{P}(A_{i_1,1}A_{i_2,1}...A_{i_{j-1},1}Ai_j, 1))^k \\ &= (\mathcal{P}(A_{1,1}A_{2,1}...A_{j-1,1}Aj, 1))^k. \end{aligned}$$

We may now compute $\mathcal{P}(\Delta(P_r) > k)$:

$$\mathcal{P}(\bigcup_{j=1}^{N} A_{i,k}) = \sum_{j=1}^{N}(-1)^{j+1}\left(\sum_{1\le i_1<...<i_j\le N}\mathcal{P}(A_{i_1,k}A_{i_2,k}...A_{i_{j-1},k}A_{i_j,k})\right)$$
$$= \sum_{j=1}^{N}(-1)^{j+1}\left(\sum_{1\le i_1...<i_j\le N}\mathcal{P}(A_{1,k}A_{2,k}...A_{j-1,k}A_{j,k})\right)$$
$$= \sum_{j=1}^{N}(-1)^{j+1}\binom{N}{j}\mathcal{P}(A_{1,k}A_{2,k}...A_{j-1,k}A_{j,k})$$
$$= \sum_{j=1}^{N}(-1)^{j+1}\binom{N}{j}(\mathcal{P}(A_{1,1}A_{2,1}...A_{j-1,1}A_{j,1}))^k$$
$$= \sum_{j=1}^{N}(-1)^{j+1}\binom{N}{j}\binom{N+n-j}{n}^k\binom{n+N}{n}^{-k}$$
$$= \binom{n+N}{n}^{-k}\sum_{j=1}^{N}(-1)^{j+1}\binom{N}{j}\binom{N+n-j}{n}^k.$$

Therefore, $\mathcal{P}(\Delta(P_r) > k) = \binom{n+N}{n}^{-k}\sum_{j=1}^{N}(-1)^{j+1}\binom{N}{j}\binom{N+n-j}{n}^k$.

The inclusion exclusion principle has the property that by adding terms to the computed sum, the value is better and better approximated:

$$\lim_{v\to n}\left|\sum_{j=1}^{v}|\bigcap_{k=1}^{j}S_{i_k}|(-1)^{j+1} - S\right| = 0.$$

If we denote with S_v ($v \in \{1, 2, ..., N-2, N-1\}$) the sum of terms from the first to the v^{th} in in the sum from above, we can say that $\mathcal{P}(\Delta(P_r) > k) \in [S_{v+1}, S_v]$. In particular, $S_2 \le \mathcal{P}(\Delta(P_r) > k) \le S_1$. This way, $\mathcal{P}(\Delta(P_r) > k)$ is between S_2 and S_1. Since

$$S_2 = S_1 - \binom{N}{2}\left(\frac{N(N-1)}{(n+N)(N+n-1)}\right)^k$$
$$= N\left(\frac{N}{N+n}\right)^k - \binom{N}{2}\left(\frac{N(N-1)}{(n+N)(N+n-1)}\right)^k$$
$$= S_1\left(1 - \left(T = \frac{N-1}{2}\left(\frac{N-1}{N+n-1}\right)^k\right)\right),$$

as far as k increases, the term T approaches 0, and $\lim_{k\to\infty}|S_2 - S_1| = 0$. Therefore, for large values of k, S_1 approximates well $\mathcal{P}(\Delta(P_r) > k)$.

The $(n-1)$ attack flushes the current pool P_r and submits the target message m. For a Threshold Pool Mix with parameters N_0 and n_0, we apply the previous result for estimating the number of rounds required to flush m by using the result from above with $N = 1$ (initially there is one message, namely m, in the pool) and $n = n_0 + N_0 - 1$ (the other

$n_0 + N_0$ are injected dummy messages).

The attacker is able to derive the value of n because he has access to all flush sets F_r of size n, but he may only estimate the value of N. According to [12], for any $\delta > 0$, $\mathcal{P}(\Delta(P_r) > k|N) \leq \mathcal{P}(\Delta(P_r) > k|N + \delta)$. In this way, by overestimating N, the attacker can find an upper bound of $\mathcal{P}(\Delta(P_r) > k)$. Another method of estimating N by studying the behavior of the mix when special dummy messages are sent. The probability that a message $m \in S_r$ is in F_r is $p = n/(N + n)$, which can be estimated by sending special dummy messages m_i in rounds i ($i \in \{1, 2, ..., s-1, s\}$, for some s. If χ_i is $true$ iff message m_i is flushed in round i, then the value of $\hat{p}(\chi_1 + \chi_2 + ... + \chi_{s-1} + \chi_s)/s$ better estimates p as s increases. Since the parameters N and n of the mix are fixed, the estimator value of $\hat{p}$ can be refined by a sufficiently large number of rounds s.

4.5 Timed Pool Mixes

Timed pool mixes also are probabilistic. Every t seconds, the mix fires a number of messages such that a pool of N randomly chosen messages remain in the mix. If at most N messages are in the mix before firing, it ceases to fire for that moment. The difference between Threshold and Timed Pool Mixes and these mixes is that the former flushes all messages, while the latter prevents N messages from being flushed. By keeping the notations made in the previous section and [12], the message batch B_r collected in the r^{th} time period is of size n, the mass of messages in round r is $S_r = P_r \cup B_r$, from which the set of messages F_r of size n are flushed and the remaining message set, P_{r+1} of size N is retained in the mix. If the batch does not exist in a round r, P_{r+1} remains the old P_r and there is no flush.

The minimum message delay is ϵ, while the maximum delay is ∞. Note that until more than N messages enter the mix, it does not flush. Because the N messages that remain are chosen randomly, there is also a small probabilty that a message is surpassed by all others and it remains in the mix for an arbitrarily long amount of time. As in the case of Threshold Mixes, if we suppose r messages enter the mix in each round, the anonymity of a message that passes through the mix depends on the mix history. In practice, the last few rounds approximate well the anonymity. The minimum anonymity of a Threshold Pool N Mix is greater than the minimum anonymity of Timed Pool Mixes. This happens because in this case the mix does not wait for $n + N$ messages to gather in the mix.

For this type of mix, an attacker is unable to force the mix to flush at a specific moment in time, but it can add illegitimate messages and prevent legitimate ones from entering the mix. One possible attack may be launched as follows:

1. The attacker blocks the input of the mix and delays all good messages, including the target message m, from entering the mix. Instead, it sends as many dummy messages to the mix as it can. The messages appear as legitimate to every entity but the attacker.

2. In the next moment of time the mix flushes. If N of his dummy messages are not flushed, the attacker knows his attack is successful. If it is not the case, the attacker continues the attack by further sending dummy messages until this condition is met.

This attack is inefficient in the number of sent dummy messages. Their number must be as large as possible, so that the probability of retaining dummy messages will increase. In practice, due to memory and network limitations, there is a limited upper bound for n, which yields to an arbitrarily large probability of success even after the first round, in at most $2t$ seconds.

Let N_{max} be the upper limit of messages retained in the mix. Since N is fixed for any (active) round, we denote by $n_{max} = N_{max} - N$ the maximum batch size. The mix designer is able to choose the pool size N. He selects a maximum batch size n_{max}, based on expected traffic. Suppose an attacker is detected in at most k consecutive intervals of time with the high probability $1-\epsilon$. By knowing the values of n_{max}, k and ϵ, and also that

$\mathcal{P}(\Delta(P_r) > k|N)$ increases as N increases, the mix designer can find the most suitable pool size N so that

$$\mathcal{P}(\Delta(P_r) > k|N) > 1 - \epsilon.$$

If N is small in comparison with n_{max}, then the attacker may flush the legitimate messages out of the mix in less than k rounds and remain unnoticed. The design procedure from above starts with a known n_{max} and computes N so that this is not the case. Suppose the attacker sends n_{max} dummy messages per round. The probability that P_r is entirely flushed in the first round is

$$\mathcal{P}(\Delta(P_r) = 1) = \left(\begin{array}{c} n_{max} \\ N + n_{max} \end{array} \right) \left(\begin{array}{c} N + n_{max} \\ n \end{array} \right)^{-1} = \frac{(n-N+1)(n-N+2)...(n-1)n}{(n+1)(n+2)...(n+N-1)(n+N)}.$$

In [10] it is proven that pool messages ensure a greater amount of the anonymity than incoming messages. However, by increasing the pool size the flush delay of a message increases.

Another attack adds just one dummy message per timed round, for a large amount of rounds. The number n of dummy messages is kept as small as 1, but the waiting time increases. By using $n = 1$ in the formula of $\mathcal{P}(\Delta(P_r) > k)$, we get

$$\mathcal{P}(\Delta(P_r) > k) = \sum_{j=1}^{N} (-1)^{j+1} \left(\begin{array}{c} N \\ j \end{array} \right) \left(1 - \frac{j}{N+1}\right)^k.$$

4.6 Timed Dynamic Pool Mixes

When t more time units pass, the Timed Dynamic Pool determines the pool size $|P_{r+1}|$ as a function of the current number of messages in the mix, namely S_r. At the end of the r^{th} flushing round, the mix checks if $|S_r| > N_{min}$. If this is the case, the mix constructs a random subset of $|S_r|$, namely F_r, of size $|F_r| = min(|S_r| - N_{min}, \lfloor \alpha |S_r| \rfloor)$.
The minimum delay of a message is ϵ and the maximum delay can be ∞. The Dynamic Pool Mix has similar anonymity to the Timed Pool Mix. An increased rate of message entries yields to more frequent flushes and greater chance for a particular message to be flushed. The Timed Dynamic Pool Mix is more resilient to flooding attacks: successful flushing in the first round comes impossible. The number of messages inside the pool mix, namely P_{r-1}, may be large and the pool P_r cannot be reduced below N_{min} in one single round. If a legitimate message is to have a greater anonymity, one might send it when traffic is high.

Even though flushing in the first round is unsuccessful, a $(n-1)$ attack can still be carried out. The attacker selects a round $r-1$ where traffic has been high, in order to expect that $|F_{r-1}| = \lfloor \alpha |S_{r-1}| \rfloor$. He may also inject more messages into the mix, so that this condition takes place with higher probabilities, and then he repeats this step.

If, for a known α, $|F_{r-1}| = \lfloor \alpha |S_{r-1}| \rfloor$ and $|S_{r-1}| = N$, then it can be proven [12] that the failure probability of an $(n-1)$ attack willing to have P_r flushed in at most k rounds, is equal to

$$\mathcal{P}(\Delta(P_r) > k) = \left(\begin{array}{c} N \\ \lfloor \alpha N \rfloor \end{array} \right)^{-k} \sum_{j=1}^{N - \lfloor \alpha N \rfloor} (-1)^{j+1} \left(\begin{array}{c} N - \lfloor \alpha N \rfloor \\ j \end{array} \right) \left(\begin{array}{c} N - j \\ \lfloor \alpha N \rfloor \end{array} \right)^k.$$

We observe that $|F_{r-1}| = \lfloor \alpha N \rfloor$ and $|P_r| = |S_{r-1}| - |F_{r-1}| = N - \lfloor \alpha N \rfloor$. By observing that the behaviour is similar to a Threshold Pool Mix server, the substitution of $n \to |F_{r-1}| = \lfloor \alpha N \rfloor$ and $N \to |P_r| = N - \lfloor \alpha N \rfloor$ in the relation

$$\mathcal{P}(\Delta(P_r) > k) = \left(\begin{array}{c} n+N \\ n \end{array} \right)^{-k} \sum_{j=1}^{N} (-1)^{j+1} \left(\begin{array}{c} N \\ j \end{array} \right) \left(\begin{array}{c} N+n-j \\ n \end{array} \right)^{k},$$

yields the equality.

The attacker is unaware of the internal parameter α of the mix. However, he is able to estimate its value. Suppose the attacker observes that the mix at round r flushes a number $|F_r| = \lfloor \alpha N \rfloor$, where $|S_r| = N$. If the attacker submits $2|F_r|$ messages before round $r+1$, then $|F_{r+1}| = \lfloor \alpha(N - \lfloor \alpha N \rfloor + 2\lfloor \alpha N \rfloor) \rfloor = \lfloor \alpha(N + \lfloor \alpha N \rfloor) \rfloor$. An approximation of α is

$$\alpha^* = \frac{|F_{r+1}|}{|F_r|} - 1 \approx \frac{\alpha(N+\alpha N)}{\alpha N} - 1 = 1 + \alpha - 1 = \alpha.$$

and N can be approximated by $N^* = |F_r|/\alpha^*$. If there are not enough messages in S_r, such that $min(|S_r| - N_{min}, \lfloor \alpha N \rfloor) = \lfloor \alpha N \rfloor$, the attacker is forced to send more dummy messages to the mix until this relation holds true, and then re-mount the attack. If sending messages to the mix is expensive in time, the attacker can wait for k rounds of flushing, after which the estimated value of N^{new} is $N^{old}(1-\alpha)^k$.

4.7 Stop-and-Go Mixes

Stop-and-Go Mixes, proposed in [11], delay individual messages as they pass through the mix network. A client willing to send a message through a series of n mixes, selected with the same probability. Let the sender be 0 and the picked mixes be, in order, 1, 2, ..., n for simplicity.

1. The sender computes for each mix i along the path a time window (TS_i^{min}, TS_i^{max}) and chooses a random delay T_i from an exponential distribution with parameter μ. These parameters are to be encrypted along with the message and they are to be found by the recipient decrypting mix, in the regular way. When the sender defines the time windows, the maximum clock offset between any two machines in the network must be known. Let syn be the maximum clock offset and t_s the local time of the sender. If each mix i has a processing delay of T_i and the directional transmission delay in the network between machines i and j is $d_{i,j}$, the time window will be computed as follows:

$$TS_i^{min} = t_s + \sum_{j=1}^{i-1} T_j + \sum_{j=1}^{i} d_{j-1,j}^{min} - syn$$

$$TS_i^{max} = t_s + \sum_{j=1}^{i-1} T_j + \sum_{j=1}^{i} d_{j-1,j}^{max} + syn$$

2. The decrypting mix i finds TS_i^{min}, TS_i^{max} and T_i. Let t be the arrival time of the package. If $t \notin [TS_i^{min}, TS_i^{max}]$, the message is discarded. If it is not the case, mix i waits for T_i seconds and then sends the message to the next recipient.

The time window has a size of $\Delta T = TS_i^{max} - TS_i^{min} = 2syn + \sum_{j=1}^{i} \Delta d_{j-1,j}$.

This is large enough to allow asynchronous clocks, but it must also be small enough to have, for any mix i, $\Delta T < T_i^a$, where T_i^a is the time needed by an attacker to flush i. A linear decrease of μ yields to an exponentially decreasing success probability of an $(n-1)$ attack. In [11, p. 11], it is shown that the success probability of such an attack for a known value of ΔT is:

$$\mathcal{P}(success) = \sum_{i=0}^{\infty} \frac{(1-e^{-\mu\Delta T})^i (\frac{\lambda}{\mu})^i e^{-\frac{\lambda}{\mu}}}{i!} = exp(\frac{-\lambda e^{-\mu\Delta T}}{\mu}).$$

The anonymity set $\mathcal{U}$ for a message m that appears in the network consists of the recipients of all messages present in the mix i when the m arrives in mix i and the recipients of all messages that will enter mix i before m is flushed away from mix i. Let a_i^m be the arrival time of m at mix i and d_i^m its correspondent departure time. The interval $[a_i^m, d_i^m]$ acts as the time window for all messages with recipients in the anonymity set of m. According to [11], the expected size of the anonymity set of message m is:

$$E(|\mathcal{U}|) = \frac{\lambda}{\mu} + \frac{e^{\frac{\lambda}{\mu}}+1}{2},$$

where λ is the message arrival rate and μ is the mix service rate.

4.8 Binomial Mixes

As the number of rounds a message stays in the mix increases, that message's anonymity increases. In the case of timed pool mixes, the probability of a message that arrived in round r to be flushed in the mix at round $r+k$ is

$$\mathcal{P}(r, r+k) = P(|S_{r+k}|) \prod_{j=r}^{r+k-1} (1 - P(|S_j|),$$

which shows that a message was not flushed in rounds r, $r+1$, ..., $r+k-2$, $r+k-1$, but it was flushed in round $r+k$. As the product in the right is smaller ($P(|S_j|)$ are large), the anonymity decreases (there is a significantly smaller probability to have the message flushed by round $r+k$ than by rounds closer to r).

The Binomial Mix [14] is a generalization of previous mix types. Mixes are seen as functions $P : \mathbb{N} \to [0, 1]$, that for each number of messages that reside in the mix associate a probability for all messages to be sent. The authors found a suitable solution for the function P. They start from the normal distribution

$$f_{\mu,\sigma}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

The parameters for the normal distribution may be chosen by analyzing the mix requirements with regard to the maximum number of users, the maximum delay and the minimum anonymity. Then they use the normal distribution cumulative for each $x = |S_r|$ and some fixed parameters σ and μ as the required function P:

$$P(x) = \int_0^x f_{\mu,\sigma}(t)dt = \frac{1}{\sigma\sqrt{2\pi}} \int_0^x e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt,$$

The advantages of this particular function P are that it grows little wiere there is low traffic (with the cost of a higher delay), but for large amounts of messages, the delay is less than the delay of Timed Dynamic Pool Mixes.

Given the function P as described above, the flusing algorithm consists of iterating through all messages in the mix and, for each message, to flip a coin, with bias $P(n)$, where $n = |S_r|$. On average, the number of flushed messages is $f = nP(n)$. As each message has the probability $P(n)$ to be flushed, the number of flushed messages is taken from a binomial distribution with parameters $(n, P(n))$ and variance $nP(n)(1-P(n))$. This way, the attacker is not able to find n immediately, by only counting f.

If the attacker is able to observe the results of one flush, he has access to f only. According to the binomial distribution, for a given n, the probability of sending f messages can be computed as

$$\mathcal{P}(f|n) = \left(\begin{array}{c} n \\ f \end{array} \right) P(n)^f (1 - P(n))^{n-f},$$

with the meaning that f messages are independently flushed with probability $P(n)^f$ and the rest of $n - f$ messages are independently kept inside the mix with probability $(1 - P(n))^(n - f)$, and there are C_n^f ways of selecting the f messages to be flushed. If the attacker knows f, he may apply the Bayes rule to compute n:

$$\mathcal{P}(n|f) = \frac{\mathcal{P}(f|n)\mathcal{P}(n)}{\mathcal{P}(f)} = \frac{\mathcal{P}(f|n)}{\sum_{i=f}^{N_{max}} \mathcal{P}(i|n)}.$$

The attacker assumes that $n \in \{f, f+1, ..., N_{max}-1, N_{max}\}$. He computes the probability of having a particular n_0 by computing the probability $\mathcal{P}(f|n_0)$ and then use the result to compute $\mathcal{P}(n_0|f)$. He also has to compute the sum from the denominator, namely $\mathcal{P}(i|n)$, for any $i \in \{f, f+1, ..., N_{max}-1, N_{max}\}$.

Assuming all parameters are now known to the attacker, a similar attack as the previous attacks can be mounted at round r, such that the output messages at round $r-1$ (F_{r-1}) were tracked. By having $n = |P_{r-1}|$, the authors in [14] have shown that

$$\mathcal{P}(\Delta(P_r) > k) = 1 - (1 - (1 - P(|S_{r-1}|))^k)^n = \epsilon + \mathcal{O}(\epsilon^2), \text{ for a suitably small } \epsilon.$$

4.9 Red-Green-Black Mixes

The problem posed by $(n-1)$ is twofold. The first problem is to find a way of deciding whether the mix is under attack or not. The second is to provide a strategy to counter or minimize the effects of the attack. In [15] a practical solution, namely Red-Green-Black Mixes, was proposed.

The idea is to send special anonymous messages from a mix to itself (these messages are called red traffic or, alternatively, heartbeat messages). When the mix is attacked, it cannot directly detect which messages are legitimate and which of them are injected by the attacker (these input messages are denoted as black messages), but it can estimate the presence of an attack if red traffic decreases and it is able to send dummy messages (called green traffic) in order to artificially strengthen the anonymity of genuine messages found in the mix. Introducing dummy traffic into the network requires more resources, but the solution clearly states its purpose and tries to minimize the artificial traffic. Red traffic is used to assess the state of the mix only, and green messages are sent to the network just when the mix has identified an $(n-1)$ attack.

The color of each message is unknown to any entity but the mix that issued the message. The difficulty of mounting an $(n-1)$ attack increases, as the attacker is able to trace the color of its own messages as black, but is unable to distinguish red messages from black legitimate messages sent by other entities to the attacked mix. This solution relies on this observation, as the attacker may not have access to the entire network. Therefore, the quantity of received red messages may be a good estimate of the quantity of genuine messages in the mix. For Red-Green-Black Mixes to be deployed, a supplementary assumption is made: the mix is aware of the network it relies on, and it knows the addresses and capabilities of other mixes. This assumption is realistic, as deployed systems are able to maintain lists of registered mixes that are active to receive and send messages. The awareness is required in order to allow the mix to simulate a sender and throw red traffic in the network.

With each flush, the mix sends a fraction of red messages anonymously back to itself. These messages may not be received by the mix that issued them due to two reasons. The first is that the mix is under attack, meaning that the legitimate traffic was blocked and only illegitimate messages are allowed to enter the mix. The attacker is unaware of the

colors of the messages it did not create, and therefore, he is unable to permit red traffic to follow its path and hide the attack in progress. The second reason is that there was not enough time for the messages to travel the network and return back to the mix. Note this may happen primarily when the network starts its activity.

A possible strategy when red message ratio drops is to block the activity of the mix until the red messages are received (this may never happen). If all mixes use this strategy, network level deadlocks may occur, and the $(n-1)$ attack may turn itself into a denial of service attack. The proposed solution in low red traffic situation is to let the mix work as if never happens, but add dummy green messages to the output. Because the mixes are able to establish a protocol to handle green messages, these messages may travel to more hops and be dismissed at the end of their path. As the adversary is unable to trace the green color of these messages, the mix artificially increases the anonymity of the black genuine messages in the mix.

Suppose there are $R + B_f + B_t$ messages received by a mix in one round, where R are red messages, B_f are black flood traffic injected in the input by the attacker, and B_t are genuine black traffic. Under the assumption that traffic volumes are close to constant, the probability of choosing a red message with genuine messages is equal to the probability p of red messages in the mix output. The colour unaware attacker would try to substitute genuine black traffic with dummy traffic, but let as much red traffic pass in order to remain undetected, but he can only choose messages at random.

The numbers of genuine messages that may have been allowed to enter the mix, given R and p, are described by a negative binomial distribution [16]:

$$\mathcal{P}(B_t = x) = \left(\begin{array}{c} R+x-1 \\ R-1 \end{array} \right) p^{R-1}(1-p)^x,$$

with the expected number of genuine black messages given R is $R(1-p)/p$. If $R+B$ are the messages received in the batch, approximately $p(R+B)$ red messages are expected to enter the mix with the batch. In this case, B would be close to $(B+R)(1-p)$. In the case the number of red messages is smaller, the mix must generate green messages to the value

$$\frac{((R+B)p-R)(1-p)}{p} = \frac{(R+B)p(1-p)}{p} - \frac{R(1-p)}{p}.$$

4.10 Round Robin Pool Queue Mixes

We propose an algorithm that overcomes the probability of having a message retained in the mix server for an arbitrary number of rounds. Our algorithm assumes that messages that enter the mix earlier have a slightly greater chance of being flushed faster than messages that enter the mix leter. The algorithm does not add severe computational additional costs over the existing schemes. The idea of our algorithm is to partition the message pool into Q queues and select messages from these queues circularily. Our construction ensures a maximum number of rounds each message remains in the mix and also acceptable minimum and maximum anonymity for the messages. Our model relies on the assumption of the existence of an ideal random oracle.

The mix server is initially populated with N dummy messages that are spread to the Q queues uniformly at random. That is, each message m_i has the probability of being pushed into the queue Q_j equal to $1/Q$, for any i and j. The dummy messages may look like any message from outside the mix server; they may be sent anonymously to the issuer mix itself or to another mix in the network, and later discarded, as based on a particular protocol. The mix server then creates a structure based on a simple circular doubly linked list where each node contains as information the index j of a queue Q_j such that $Q_j \neq \emptyset$. A pointer f to the first node in the list may also be retained as the first node.

At each round, the mix server receives M messages as input messages and spreads them to the Q queues uniformly at random, in a similar manner to the initial step. The mix

server now contains $N + M$ messages in Q queues, which yields to $(N + M)/Q$ messages per queue on average and a majority of queues whose sizes have little difference to this size. Note that some queues may have been empty before the spread of input messages to the queues, and therefore each of these queues may be inserted as information in a new node in the circular doubly linked list. A simple procedure is to insert each new node as the previous node of the node pointed by f and update f as the pointer to the new node.

In the same round, after the messages are received, the mix network starts from the current node of the list (in the first round, this node is the first node of the list), reads its information, i.e. the queue index j, and selects exactly $min(|Q_j|, M)$ messages from this queue. The algorithm proceeds to the next node in the list in circular manner. If there are less than N messages selected, the algorithm continues by reading the information in the current node and by collecting $min(|Q_{next(j)}, M|)$ messages from the queue $Q_{next(j)}$ and by moving to the next nonempty queue, that has index $next(j)$. The process continues until M messages were collected. In the process, some of the queues may be removed from the circular doubly linked list, with possible updates on f. Note that in the round, the algorithm proceeds to the following queue at least once.

The round is ended by applying a random permutation on the M output messages and by sending them further as flush output. It is simple to see that the mix remains with N messages among the initial $N + M$ messages that were received in that round. This way, there will be a void circular doubly linked list of nodes only before the algorithm starts.

We abstract the way the doubly linked list is handled. For details on how to perform operations on linked lists, see [23, p. 204]. We will only retain these procedures or functions on the structure based on the circular doubly linked list that we proposed:

- $add(q)$ is a procedure that adds a new node to the circular doubly linked list. The node information is the queue index $q \in \{0, 1, ..., Q-2, Q-1\}$. This procedure may modify f.
- $remove(q)$ is a procedure that removes the node from the circular doubly linked list that has as information the queue index $q \in \{0, 1, ..., Q-2, Q-1\}$. This procedure may also modify f.
- $next(q)$ is a function that returns the queue index retained as information in the node that is next to the node with information $q \in \{0, 1, ..., Q-2, Q-1\}$ in the list. It is guaranteed that $Q_{next(q)} \neq \emptyset$.

We also abstract the handling of queues. See also [23, p. 200] for details on queues. Our algorithm only needs the following procedures and functions on a queue Q, which are quite similar to the basic operations on queues. We do not consider returned messages that are not retained as errors (see $clear(Q)$ for details):

- $push(m, Q)$ is a procedure that adds message m at the tail of queue Q.
- $pop(Q)$ is a function that both returns the message G located at the head of queue Q and removes the message from the head of queue Q.
- $clear(Q)$ is a procedure that calls $pop(Q)$ (without retaining any message returned by $pop(Q)$), until $Q = \emptyset$.

In order to maintain a M fixed size array of messages to be flushed after being popped from the queues, we abstract simple procedures on arrays:

- $set(m, p, A)$ is a procedure that adds message m to the array A at the position p.
- $get(p, A)$ is a function that returns the content of the array at position $p \in \{0, 1, ..., N-2, N-1\}$.

- $clear(A)$ is a procedure that empties the array by setting $A = \emptyset$ and by having all its positions free. The first free position is 0.
- $randomize(A)$ is a procedure that permutes the array A according to a uniformly random permutation π.

We will also need auxiliary procedures, functions and streams for our algorithm.

- $dummyMessage()$ is a function that returns a dummy message, that is indistinguishable to messages sent by users from the exterior, but that may follow specific protocols in the mix network that handle dummy messages and discard them if necessary.
- $randomNumber(min, max)$ is a function that returns a random integer number from the set $\{min, min+1, ..., max-1, max\}$ where each element of this set has the same probability of being returned as the other. Note that this function may be built on top of an ideal random oracle.
- $[input]$ is a stream that returns the next input message when invoked, with blocking.
- $[output]$ is a stream that receives a message when invoked, with blocking.

We denote our index variables used in the algorithm as follows:

- q is used as a queue index. It is granted that $q \in \{0, 1, ..., Q-2, Q-1\}$.
- n and m are used as message indexes. $n \in \{0, 1, ..., N-1, N\}$ and $m \in \{0, 1, ..., M-1, M\}$ are algorithm invariants.

The pseudocode for our algorithm is presented in figure 4.1.

We analyze the time complexity of the preprocessing (steps 1 through 8). Steps 1 to 3 are performed in $\mathcal{O}(Q)$. There exists a simple algorithm that makes step 2 in $O(1)$ by retaining static queues (note that a queue never has more than $N + M$ messages at any round). Step 3 of $\mathcal{O}(1)$ is repeated Q times. Step 4 executes N times. Steps 5 and 6 are made in constant time, for constant time $randomNumber$ and $dummyMessage$ functions. The addition at step 7 is also $\mathcal{O}(1)$. The *push* operation is constant in time as well. The preprocessing yields to an $\mathcal{O}(N + Q)$ time complexity.

Let us now analyze a round. Steps 10 and 11 are in $\mathcal{O}(M)$. Note that this is not a severe problem, since reading in steps 12 to 14 and writing in steps 30 to 32 are also of an $\mathcal{O}(M)$ time complexity. Steps 15 to 19 are similar in nature with steps 4 to 8, thus yielding to an $\mathcal{O}(M)$ cost in time. Steps 20 to 28 are also $\mathcal{O}(M)$, since in each loop in the inner while at step 22 the variable m is incremented and both **while** structures have the condition $m < M$. The lines 23 to 25 and 26 to 28 are in constant time as well. Step 29 is performed in $\mathcal{O}(M)$ time, given there exist algorithms for performing a random permutation of size M in $\mathcal{O}(M)$, such as in [27]. This way, the round is no more than $\mathcal{O}(M)$. The step of *randomize* may be regarded as the step where not only the mix shuffles the output, but also it reencrypts and provides proofs of correct reencryption.

If the step 29 is not performed, the outputs will be subject to a simple pattern: the $[output]$ would be filled with messages that have the structure of a concatenation of of sorted concatenations of messages. Each queue may have messages from different generations, and this property is more proeminent as N/M increases. We will analyze the minimum and maximum anonymity of a message that passes such a mix server. We measure the anonymity of a message m as the number of messages m' that exist the mix network between the entrance and exit of m. If more messages come with m or exit with m, they are counted for the anonymity set size. Our scheme provides from the minimum anonymity of M in $\mathcal{O}(1)$ rounds to the maximum anonymity of $MQ+N$ in $\mathcal{O}(MQ)$ rounds.

```
   // clear queues and list
 1 for q ← 0 to Q − 1 do
 2 |  clear(Q_q)
 3 |  remove(q)

   // add N dummy messages
 4 for n ← 0 to N − 1 do
 5 |  d ← dummyMessage()
 6 |  q ← randomNumber(0, Q − 1)
 7 |  if Q_q = ∅ then add(q)
 8 |  push(d, Q_q)

   // start the rounds of the protocol
 9 while true do
   |  // clear input and output arrays
10 |  clear(I)
11 |  clear(O)
   |  // read M input messages from [input] to I
12 |  for m ← 0 to M − 1 do
13 |  |  i ← [input]
14 |  |  set(i, m, I)
   |  // spread the M input messages from I to queues Q_*
15 |  for m ← 0 to M − 1 do
16 |  |  i ← get(m, I)
17 |  |  r ← randomNumber(0, Q − 1)
18 |  |  if Q_r = ∅ then add(r)
19 |  |  push(i, Q_r)
   |  // select M messages from queues Q_* to O
20 |  m ← 0
21 |  while m < M do
   |  |  // take at most M messages from the current queue Q_q
22 |  |  while m < M and Q_q ≠ ∅ do
23 |  |  |  o ← pop(Q_q)
24 |  |  |  set(o, m, O)
25 |  |  |  m ← m + 1
26 |  |  p ← q
27 |  |  q ← next(q)
28 |  |  if Q_p = ∅ then remove(p)
   |  // randomize array O before output
29 |  randomize(O)
   |  // write M messages from O to [output]
30 |  for m ← 0 to M − 1 do
31 |  |  o ← get(m, O)
32 |  |  [output] ← o
```

Figure 4.1: Round Robin Pool Queue Flushing

It is simple to see that the anonymity has the lower bound of $M + N$. Suppose the N messages that reside in the mix are in queues Q_2, Q_3, ..., Q_{Q-1}, Q_Q, that all M messages enter in the queue Q_1 and the round finds $q = 1$. All M messages that entered in Q_1 are flushed, thus having an anonymity of M. We add to the anonymity the existing N messages. In this case, the time spent by the message in the mix server is $\mathcal{O}(\infty)$ rounds.

The maximum anonymity is upper bounded by $MQ + N$. Suppose that $N \geq M$ and N messages that reside in the mix are spread in the queues such that in queue Q_0 there is one message, in queue Q_{Q-1} there are $M - 1$ messages and the remaining residing $N - M$ messages occupy the queues Q_1 up to the first M slots, then Q_2 up to the first M slots, ..., until N messages occupy the queues. Suppose that in this round the target message m enters the queue Q_{Q-1} and the other $M - 1$ messages enter the queue Q_0. By letting $q = 0$ and by having at each subsequent round all messages enter the queue that is to be processed at that particular round, after Q rounds the message m exits the mix network. There are are MQ messages that entered the queue which are added to the N existing messages. The delay is $\mathcal{O}(MQ)$ rounds.

Conclusion

The traffic analysis problem, which is to keep confidential who communicates with whom and when, can be solved by using public key cryptography. The underlying telecommunication system permits anyone to add, remove or modify messages and also know the origin and destination of each message. We brought a simple solution to the problem of arbitrarily long delays by queueing messages.
The table 4.1, based on [9, p. 15] summarizes the average delay and the anonymity of the presented flush algorithms. The value n stands for the number of messages, the number k stands for the threshold, t is for the time amount of the round, and r is the rate of arrival of messages. By $\mathcal{A}$ we denote anonymity.

algorithm	avg. delay	min. $\mathcal{A}$	max. $\mathcal{A}$
4.1 Threshold [9]	$\frac{n-1}{2r}$	n	n
4.2 Timed [9]	$\frac{t}{2}$	0	mix capacity
4.3 Threshold or Timed [9]		0	n
4.3 Threshold and Timed [9]		n	mix capacity
4.4 Threshold Constant Pool [9]	$\frac{n}{r}\left(1 + \frac{k}{n+k}\right)$	n	$\frac{k}{n} logf - \left(1 - \frac{k}{n}\right) log(n + k)$
4.5 Timed Constant Pool [9]		1	$<$ # senders
4.6 Timed Dynamic Pool [9]		1	$<$ # senders
4.7 Stop & Go [11]	Σt_i	0	$\frac{\lambda}{\mu} + \frac{e^{\frac{\lambda}{\mu}}+1}{2}$
4.8 Binomial [14]	$\frac{t}{P(n)}$	$nP(n)$	mix capacity
4.9 Red Green Black [15]	t	# black	# black
4.10 Round Robin Pool Queue		n	nQ

Table 4.1: Anonymity and delay per mix flush algorithm on n messages

The next chapter will describe the techniques mix servers use in order to prove that the permutation π they applied to messages was indeed correct.

Chapter 5

Mix Proof Techniques

Some mix networks not only permute the messages they receive, but also they reencrypt them in order to be impossible for a curious observer to trace the origins of a message. The difficulty of such mix networks lies in proving the correct mixing and reencryption. The techniques grade from the most inefficient to the most efficient in the number of exponentiation required to reencrypt the inputs, verify correct mixing and decrypt the outputs.

This chapter has 5 sections: sections 5.1, 5.2 and 5.3 survey proof techniques that require $\mathcal{O}(nk)$ exponentiations to prove their work, while sections 5.4 and 5.5 analyze protocols that need only $\mathcal{O}(k)$ exponentiations, for n messages and k servers. The observations are based on [8].

5.1 Cut and Choose Zero Knowledge

Mix networks do not work if one of the mixes stops. The authors of [17] show a fault tolerant multivalued scheme, in which at most half the mixes can stop or deviate from the protocol. We present their scheme based on the Discrete Log problem which uses the ElGamal public key cryptosystem with its reencryption properties and the Shamir (k, n)-threshold secret sharing scheme.

Suppose there are l senders, namely P_1, P_2, ..., P_{l-1}, P_l, such that each P_i sends the message m_i. Assume also that there are n mixes, denoted by M_1, M_2, ..., M_{n-1}, M_n. Let p and q be primes such that $q|p-1$. Let $g \in \mathbb{Z}_p$ be a q^{th} root of the unity. The protocol is carried out as follows:

1. Each mix M_j chooses a secret key $x_j \in \mathbb{Z}_q^*$ and publishes $y_j \equiv g^{x_j} \mod p$ as his public key. He then distributes its secret key x_j to by using the (k, n) secret sharing scheme.

2. Each sender P_i proves by an interactive zero knowledge proof that he knows m_i, encrypts it and publishes the result

$$(G_i, M_i) = (g^r \mod p,\ m_i \left(\prod_{j=1}^{n} y_j^r \right) \mod p).$$

3. Each mix M_j reencrypts and shuffles all (G_i, M_i) sequentially. After n reencryptions, mix M_n has published

$$(\hat{G}_i, \hat{M}_i) = (g^{\left(r + \sum_{j=1}^{n} r_{ij} \right)} \mod p,\ m_i \prod_{j=1}^{n} y_j^{\left(r + \sum_{j=1}^{n} r_{ij} \right)} \mod p).$$

4. For each $(\hat{G}_i, \hat{M}_i)$, each mix M_j computes and publishes $D_j = \hat{G}_i^{x_j} \bmod p$. He then proves he knows the value of D_j by an interactive zero knowledge proof.

5. The plaintext m_i is obtained by

$$m_i \equiv \hat{M}_i \prod_{j=1}^{n} D_j^{-1} \equiv m_i \prod_{j=1}^{n} g^{x_j \left(r + \sum_{j=1}^{n} r_{ij} \right)} \prod_{j=1}^{n} g^{-x_j \left(r + \sum_{j=1}^{n} r_{ij} \right)} \bmod p.$$

If there some D_j values were not published by some mixes M_j, these values can be computed by accessing the secret values x_j by k of the remaining active mixes in a (k, n) threshold manner. A version of this procedure is also presented in detail in [18, p. 3].

5.2 Pairwise Permutations

There are applications where the batch size is small. The authors of Millimix [21] have proposed it as a threshold mix network scheme that is very efficient for input batches of small size. Even though the time complexity of their algorithm is $\mathcal{O}(nlogn)$, as opposed to other solutions with the cost $\mathcal{O}(n)$, they state that the small constant makes their solution highly practical for smaller values of n, and also publicly verifiable.

Their scheme also relies on the properties of ElGamal cryptosystem. The input of their mix network is a sequence of l ElGamal ciphertexts $E = \{E_1, E_2, ..., E_{l-1}, E_l\}$ and the output is another sequence of l ElGamal ciphertexts $E' = \{E'_1, E'_2, ..., E'_{l-1}, E'_l\}$, such that there exists a secret and uniformly at random permutation and reencryption σ under which the plaintext of every E'_i is equal to the plaintext of $E_{\sigma(i)}$.

Millimix is based on a comparison network architecture, which consists of wires and comparitors (details on this can be found in [22] and [23]). A comparitor is a device that takes as input an ordered pair of values $< x, y >$ and outputs $f(< x, y >)$, which is either $< x, y >$, or $< y, x >$. A simple comparitor that is used in sorting networks is

$$f(< x, y >) = \begin{cases} < x, y >, & x < y \\ < y, x >, & x \geq y \end{cases}$$

The proposed network consists of n input wires and n output wires. The flow in the network does not permit for a comparitor to be used twice, and input values are permuted in parallel by the network comparitors and turn themselves into output values after a known number of comparisons. The authors denote by F^* the set of potentially different reordering functions f for the network comparitors and see F^* as a parameter of the network. The network topology is that of a permutation network, as described in [21], which has $nlogn - n + 1$ comparitors. The permutation network Π takes inputs $x_1, x_2, ..., x_{l-1}, x_l$ and an easily computable set of reordering and random reencryption functions F^*_σ, where σ is a permutation among the $l!$ possible permutations. The output of the network is a set of values $x_{\sigma(1)}, x_{\sigma(2)}, ..., x_{\sigma(l-1)}, x_{\sigma(l)}$. These values are computed by using the semantically secure ElGamal reencryption property.

The protocol makes use of a bulletin board to which ElGamal ciphertexts are submitted initially. Then each of the n mixes selects as input all ciphertexts in the bulletin board, operates on them and posts as output the resulting ElGamal ciphertexts to the bulletin board. At each step, each server proves he worked correctly with the ciphertexts, by showing each comparitor in Π worked correctly. In order to not show the value of $f(< x, y >)$ in a comparitor, the mix proves that just one of the two reencryptions performed at a comparitor step was performed as expected.

Each mix m_i chooses a permutation σ_i uniformly at random and executes the actions of Π under the set of functions $F^*_{\sigma_i}$. If all mixes followed this protocol correctly, then the final

output ciphertext E' is the result of applying ElGamal reencryptions and a permutation σ to the input ciphertexts E, where $\sigma = \sigma_n \circ \sigma_{n-1} \circ \cdots \circ \sigma_2 \circ \sigma_1$. By having at least one honest mix server m_i in the network, with regard to the secrecy of σ_i and the private reencryption parameters, its unknown permutation σ_i keeps the final permutation σ secret. Each mix server proves its correct operations in Π at the level of its comparitors, by showing that the pair of input plaintexts is the same as the pair of output plaintexts. The described protocol relies on two operations: the proof of plaintext equivalence (named by its authors $\mathcal{PEP}$) and the disjunctive proof of plaintext equivalence (named by its authors $\mathcal{DISPEP}$).

Let for some plaintext m be the ciphertext $(\alpha, \beta) = (my^{\gamma_1}, g^{\gamma_1})$ and the reencryption ciphertext $(\alpha', \beta') = (my^{\gamma_2}, g^{\gamma_2})$, for some γ_1 and $\gamma_2 \in \mathcal{G}_\Pi$. The plaintext equivalence proof ($\mathcal{PEP}$) has the aim of convincing a verifier that the player reencrypted (α, β) correctly by knowing and yet not revealing the value of $\gamma = \gamma_1 - \gamma_2$. This proof is constructed by using the homomorphism property of the ElGamal cryptosystem. If (α, β) and (α', β') are ciphertexts for the same plaintext m, then

$$(\alpha/\alpha', \beta/\beta') = ((my^{\gamma_1})/(my^{\gamma_2}), g^{\gamma_1}/g^{\gamma_2}) = (y^{\gamma}, g^{\gamma})$$

is an encryption of the plaintext 1. By letting $Y = (\alpha/\alpha')^z(\beta/\beta') = y^{\gamma z}g^{\gamma}$ and $G = y^z g$, the values can be proven by the Schnorr identification algorithm with regard to the reencryption secret parameter γ.

In order to follow the disjunctive plaintext equivalence proof ($\mathcal{DISPEP}$), the mixes use the Disjunctive Schnorr identification protocol. This is a variant of the Schnorr identification scheme in which the mix does the proof of knowledge of the private key that associated with one of two public keys (Y_1, G_1) and (Y_2, G_2), rather than with a single public key (Y, G). The verifier is unable to determine the public key for which the prover has knowledge of the secret exponent. Suppose the user knows only one of the two secret logarithms x_1 and x_2. Let it be $x_1 = log_{G_1} Y_1$, which is associated to the public pair (Y_1, G_1). The disjunctive Schnorr scheme works as follows:

1. The prover chooses randomly e_1 and s_2 and a challenge c_2. He computes $w_1 = G_1^{e_1}$ and $w_2 = G_2^{-s_2} Y_2^{c_2}$ and sends these two values to the verifier.
2. The verifier chooses a random challenge c and sends it to the prover.
3. The prover computes $c_1 = c \oplus c_2$ and $s_1 = c_1 x_1 - e_1$ and sends s_i, c_i to the verifier.
4. The verifier checks that $Y_i^{c_i} = G_i^{s_i} w_i$, for $i \in \{1, 2\}$.

Indeed, if all computations were peformed correctly, both equalities $Y_i^{c_i} = G_i^{s_i} w_i$ hold:

1. $G_1^{s_1} w_1 = G_1^{s_1} G_1^{e_1} = G_1^{c_1 x_1 - e_1} G_1^{e_1} = G_1^{c_1 x_1 - e_1 + e_1} = G_1^{c_1 x_1} = (G_1^{x_1})^{c_1} = Y_1^{c_1}$
2. $G_2^{s_2} w_2 = G_2^{s_2} G_2 - s_2 Y_2^{c_2} = G_2^{s_2 - s_2} Y_2^{c_2} = Y_2^{c_2}$.

The prover may not know the value of $x_2 = log_{G_2} Y_2$, but, similarly to the Schnorr scheme, without knowledge of x_1 either, he could not carry out the protocol. The disjunctive plaintext equivalence proof allows a mix to demonstrate that an ElGamal ciphertext (α, β) is a reencryption of one of the two different ElGamal ciphertexts (α_1, β_1) and (α_2, β_2). The $\mathcal{DISPEP}$ protocol is employed on the two pairs of values $(Y_1, G_1) = (\alpha/\alpha_1, \beta/\beta_1)$ and $(Y_2, G_2) = (\alpha/\alpha_2, \beta/\beta_2)$.

The execution of a comparitor starts with receiving as input two ElGamal ciphertexts (α_1, β_1) and (α_2, β_2) with their respective plaintexts m_1 and m_2. After the execution of one of the functions in F_σ^*, the comparitor outputs two ElGamal reencryptions (α_1', β_1') and (α_2', β_2') with corresponding plaintexts m_1' and m_2'. The mix must prove that either $(m_1, m_2) = (m_1', m_2')$, or $(m_1, m_2) = (m_2', m_1')$. He proves two equalities:

- $m_1 = m_1' \vee m_1 = m_1'$ by using the $\mathcal{DISPEP}$,

- $m_1 m_2 = m_1' m_2'$ by using the homomorphic property of the ElGamal cryptosystem and by applying $\mathcal{PEP}$ on the product ciphertexts.

A similar approach is described in [24] under the algorithm name *MiP-2*. The article also analyzes an alternative algorithm with the name *MiP-1* that uses permutation networks and runs in time $\mathcal{O}(tNlogN)$, where t is the number of dishonest servers.

MiP-1 has one phase only, as opposed to *MiP-2*, which has two. The phase consists of a randomized decryption where servers perform decryption and random permutation at the same time. This protocol is built on top of a series of $t+1$ permutation networks with a total of $d = (2logn - 1)(t+1)$ comparitor columns. Each server is assigned to some column comparitors within a permutation network. The server in charge with the column j has a public encryption key $y_j = g^{x_j}$ and a secret decryption key $x_j \in G_q$. The latter is distributed via a (k, n) threshold secret sharing scheme, in order to be used if the mix encounters problems. Denote by x the sum $x_1 + x_2 + ... + x_{d-1} + x_d$ and by y the product $y_1 y_2 \cdots y_{d-1} y_d$ and let $\hat{y}_j = y_j y_{j+1} \cdots y_{d-1} y_d$. The public encryption key is $y = y_1 y_2 \cdots y_{d-1} y_d$. The authors have also chosen three functions: an encryption function E and a decryption function D and a reencryption function R, with the properties that $D_{x_j}(E_{\hat{y}_j}(m, s)) = E_{\hat{y}_{j+1}}(m, s)$ and also $R_y(E_y(m, s), r) = E_y(m, s + r)$, for any message m and random factors s and r. For the edge case, $D_{x_d}(E_{y_d}(m, s)) = m$. The $+$ and $\cdot$ may in particular be seen as addition and multiplication in a group. Fixed comparitors in the permutation network also apply D_{x_j} only.

The messages m sent by users to the network are in the form of ElGamal encryptions with the public key y:

$$\begin{aligned} message_k &= (m_k y^{s_k}, g^{s_k}) \\ &= (m_k (y_1 y_2 \cdots y_{d-1} y_d)^{s_k}, g^{s_k}) \\ &= (m_k (g^{x_1} g^{x_2} \cdots g^{x_{d-1}} g^{x_d})^{s_k}, g^{s_k}) \\ &= (m_k g^{(x_1 + x_2 + ... + x_{d-1} + x_d) s_k}, g^{s_k}) \\ &= (M_k, G_k). \end{aligned}$$

By letting $I_0 = E_{\hat{y}_j}(m_0, s_0)$ and $I_1 = E_{\hat{y}_j}(m_1, s_1)$ be the inputs of a switching comparitor in the column j, the switching comparitor chooses r_0 and r_1 uniformly at random and computes output values O_b and $O_{\bar{b}}$, for $b \in_R \{0, 1\}$, in the following manner:

$$\begin{aligned} O_b &= R_{\hat{y}_{j+1}}(D_{x_j}(I_0), r_0) = R_{\hat{y}_{j+1}}(E_{\hat{y}_{j+1}}(m_0, s_0), r_0) = E_{\hat{y}_{j+1}}(m_0, s_0 + r_0), \\ O_{\bar{b}} &= R_{\hat{y}_{j+1}}(D_{x_j}(I_1), r_1) = R_{\hat{y}_{j+1}}(E_{\hat{y}_{j+1}}(m_1, s_1), r_1) = E_{\hat{y}_{j+1}}(m_1, s_1 + r_1). \end{aligned}$$

For instance, when a message $I_0 = (M[0]_k^1, G[0]_k^1)$ is input to a switching comparitor in the first column ($j = 1$), the output O_b is computed as follows:

$$\begin{aligned} O_b &= R_{\hat{y}_2}(D_{x_1}(I_0), r_{0,1}) \\ &= R_{\hat{y}_2}(D_{x_1}(M_k, G_k), r_{0,1}) \\ &= R_{\hat{y}_2}(D_{x_1}(m_k g^{(x_1 + x_2 + ... + x_{d-1} + x_d) s_k}, g^{s_k}), r_{0,1}) \\ &= R_{\hat{y}_2}(m_k g^{(x_1 + x_2 + ... + x_{d-1} + x_d) s_k} (g^{s_k})^{-x_1}, g^{s_k}), r_{0,1}) \\ &= R_{\hat{y}_2}(m_k g^{(x_1 + x_2 + ... + x_{d-1} + x_d - x_1) s_k}, g^{s_k}), r_{0,1}) \\ &= R_{\hat{y}_2}(m_k g^{(x_2 + x_3 + ... + x_{d-1} + x_d) s_k}, g^{s_k}), r_{0,1}) \\ &= (m_k g^{(x_2 + x_3 + ... + x_{d-1} + x_d) s_k} \hat{y}_2^{r_0}, g^{s_k} g^{r_{0,1}}) \\ &= (m_k \hat{y}_2^{s_k} \hat{y}_2^{r_{0,1}}, g^{s_k} g^{r_{0,1}}) \\ &= (m_k \hat{y}_2^{s_k + r_{0,1}}, g^{s_k + r_{0,1}}) \\ &= E_{\hat{y}_2}(m_k, s_k + r_{0,1}) \\ &= (\hat{M}[b]_k^1, \hat{G}[b]_k^1). \end{aligned}$$

The output of a switching gate in the column $j+1$ is a message encrypted with the key $\hat{y}_{j+1}$. In the last column, the output only gates apply D_{x_d}. Each gate then proves in zero knowledge (without revealing x_j, r_0, r_1 or b) that it applied D_{x_j} and $R_{\hat{y}_{j+1}}$ correctly (see [24, p. 268] for details).

By applying the procedure to the last column, we obtain m as

$$D_{x_d}(R_{\hat{y}_d}(D_{x_{d-1}} \cdots R_{\hat{y}_2}(D_{x_1}(I_b), r_b) \cdots)) = m_b$$

Concretely, the message m from above reaches the last gate as

$$\begin{aligned} I[b'] &= (M[b]_k^d, G[b]_k^d) \\ &= (m_k g^{x_d(s_k + r_{b_1,1} + r_{b_2,2} + \ldots + r_{b_{d-2},d-2} + r_{b_{d-1},d-1})}, g^{s_k + r_{0,1} + r_{\cdot,2} + \ldots + r_{\cdot,d-2} + r_{\cdot,d-1}}), \end{aligned}$$

where $b_j \in \{0,1\}$ is according to the selection of $b \in \{0,1\}$ made at that particular step. By dividing $M[b]_k^d$ to $(G[b]_k^d)^{x_d}$, the plaintext message m is finally decrypted.

5.3 Randomized Partial Checking

In [25] a different technique is proposed under the name of *Randomized Partial Checking*. In order to provide strong evidence of its correct application, the mix does not provide a proof of complete correctness, but rather it reveals a randomly selected subset of its input messages and their corresponding output messages. This technique suits both original decryption mix networks, as described in [1], and reencryption networks.

Suppose there is a sequence of l ciphertexts I_1, I_2, ..., I_{l-1}, I_l corresponding to input messages m_1, m_2, ..., m_{l-1}, m_l and a sequence of output messages O_1, O_2, ..., O_{l-1}, O_l. As expected, when the mix network that consists of n mix servers S_j performs its operations correctly, the output sequence $< O_i >$ is a permutation of the input sequence $< I_i >$. The proposed solution is described with regard to public and secret parameters (e.g. cryptographic keys) of the network, named pk and sk, respectively, and their plaintext aware encryption (E) and decryption (D) algorithms. The private keys may be secretly shared between more servers or authorities, possibly in a (k,n) threshold manner, in order to prevent malfunction of the entire network due to the failure of a minority of mixes.

The authors of [25] make use of a public bulletin board where digitally signed messages are written in append mode. All input and output to the mix servers will be posted on the bulletin board. Denote the initial ciphertext of message m_i with $C_{i,0} = E_{pk}(m_i)$ and the input set as $< C_{1,0}, C_{2,0}, ..., C_{l-1,0}, C_{l,0} >$. Each server S_j transforms each message $C_{i,j-1}$ by using a cryptographic transformation function f_j. The transformation may consist of decryption/reencryption (by using sk) and shuffle of the input (by using a secret random permutation π_j of $\{1, 2, ..., l-1, l\}$, so that

$$C_{i,j} = f_j(C_{\pi_j(i),j-1}).$$

For reencryption mixes, the final decryption operation may have the form $O_i = D_{sk}(C_{i,t})$ and it may be operated by a quorum of mixes that have access to the secret parameters sk. For decryption mixes, this operation will be included as a subroutine of f: each mix S_j decrypts a ciphertext $C_{\pi_j(i),j-1}$ by using its private parameters sk_j.

In order to be verified, each server S_j commits to its private permutation π_j, in one of the two following ways: either it uses the sequence $< \pi_j(1), \pi_j(2), ..., \pi_j(l-1), \pi_j(l) >$ by committing $\Gamma_j^{in} =< \gamma_{w_{i,j}}(\pi_j(i)) >$ $(i \in \{1, 2, ..., l-1, l\})$, or it uses the sequence $< \pi_j^{-1}(1), \pi_j^{-1}(2), ..., \pi_j^{-1}(l-1), \pi_j^{-1}(l) >$ by committing $\Gamma_j^{out} =< \gamma_{w_{i,j}}(\pi_j^{-1}(i)) >$ $(i \in \{1, 2, ..., l-1, l\})$, depending on the role of the mix in the network. The value of $\gamma_w(x)$ may be computed by selecting a random string s and a hash function (e.g. SHA-1) h:

$$\gamma_w(x) = h(s||x),$$

where $||$ is the concat operator. Each server will reveal a fraction $p > 0$ of its correspondences between an input and an output message. The subset that is to be revealed will be selected by other servers or by using a random oracle. In order for server S_j to reveal information for input $C_{k,j-1}$ mapping to $C_{i,j}$, the server reveals the triple $(k, i, R_{i,j,k}$, where $R_{i,j,k}$ is the information necessary to validate that $C_{i,j} = f_j(C_{\pi_j(i),j-1})$. For original mix networks, $R_{i,j,k}$ may be the random encryption padding; for reencryption networks - a proof of knowledge of the secret parameters. The mix S_j also decommits its commitment to $\pi_j(k)$ (if it provided a commitment of the form Γ_j^{in}) or to $\pi_j^{-1}(i)$ (if it committed $\Gamma_j^{-1}(out)$).

Servers must prove correct operations for a subset of the input to output mappings. This subset may be chosen by having every server S_j publish a uniformly at random value r_j and retain the random seed r as $\oplus_{j=1}^{n} r_j$, where $\oplus$ may be the xor operator. Servers then compute a random value $q = h(r||\mathcal{B})$, where $\mathcal{B}$ is the full content of the bulletin board after all servers published their data and h is a hash function. For each server S_j, a seed $q_j = h(q||j)$ may be computed for the predicates $P_{in}(q_j, k)$ and $P_{out}(q_j, i)$:

- If $P_{in}(q_j, k)$ is *true*, then S_j is obliged to publish the correspondance of input $C_{k,j-1}$ and output $C_{\pi_j^{-1}(k),j}$.
- If $P_{out}(q_j, i)$ is *true*, then S_j is obliged to publish the input correspondance of output $C_{i,j}$ and output $C_{\pi_j(i),j-1}$.

For some versions, the predicates P_{in} and P_{out} may be *true* with a probability p, including $p = 0$. The authors have presented two mix network models.

The first scheme works as follows: each server S_j commits Γ_j^{out} on mappings from outputs to inputs. When the input to output relations are to be revealed, $P_{in} = false$ and $P_{out} = true$ with probability p. In this way, only some messages will remain untraceable after passing the first mix server in the network. However, as the messages are passing the network, every message will have its origin hidden with high probability. The authors state that for $p = 1/2$, if

$$n \leq log_2\left(\frac{n}{\epsilon}\right),$$

then the probability of a final output that can be linked to its corresponding input is less than ϵ.

The second scheme is to perform pairwise dependent checks. Each two adjacent mix servers are seen as a pair. Supposing that n is even, the first two mix servers are 1 and 2, and the last two mix servers are $n - 1$ and n. Each odd indexed server S_j commits to Γ_j^{in} on the mapping from input messages to output messages and each even indexed server S_{j+1} commits to Γ_j^{out} on the mapping from output messages to input messages. The process of revealing input to output relations is described by the predicates

- $P_{in}(q_j, \pi_j(i)) = false$,
- $P_{out}(q_j, i) = true$ with probability $p = 1/2$,
- $P_{in}(q_{j+1}, i) = true$ iff $P_{out}(q_j, i) = false$,
- $P_{in}(q_{j+1}, \pi_{j+1}^{-1}(i)) = false$.

For less than $n/2$ corrupt mix servers, it is simple to observe that there exists a pair (S_j, S_{j+1}) in which both odd and even servers are honest. The authors prove in [25, p. 11] that an adversary who alters k votes remains undetected with probability of at most 2^{-k}.

In figure 5.1, by having $P_{in}(q_{j+1}, i) = true$ iff $P_{out}(q_j, i) = false$, no message path is completely published. The figure shows only the public segments after the protocol has ended. The segments between mixes are subject to public view. Those inside the mix are made public by the mix to show it performed its tasks correctly. For simplicity, we considered 4 origins of messages, namely α, β, γ and δ. During the traversal of each message through the pair (S_j, S_{j+1}), according to the path segments that have been published, we put in the circles its origin, or ?, if its trace is already lost. Note that for $p = 1/2$, between mixes S_j and S_{j+1}, each message for which its origin is lost is hidden among $l/2$ messages with unknown origins.

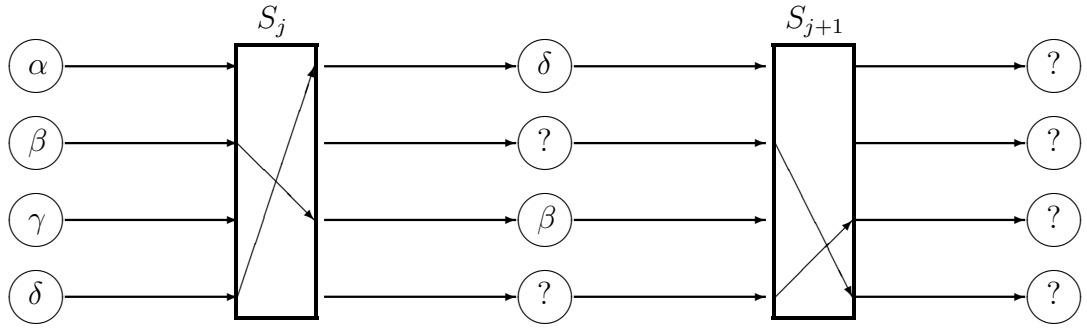


Figure 5.1: In the end of its traversal, each message loses its origin.

5.4 Optimistic Mixing

The Optimistic Mixing technique [18] resembles the ElGamal reencryption technique, but brings with itself a highly efficient method for proving that the mixing was performed correctly and, if the proof fails and cheating by a server is detected, a method for falling back on a less efficient mix. The optimistic procedure relies on the existence of a public bulletin board, which is resistant to tampering and denial of service attacks.

Each input message is a ciphertext of a plaintext which includes a cryptographic checksum. In order to verify the correctness of a mix server in operating the messages, the mix must prove that the product of the plaintexts corresponding to the input messages equals the product of the plaintexts corresponding to the output messages. If a mix server did not mix the messages correctly, it had to introduce in the output at least a ciphertext with a corresponding incorrect checksum in the plaintext. After output decryption, invalid checksums are traced. If this is the case, either the sender has miscalculated the checksum, or a mix server was cheating. User i sends to the mix an ElGamal encryption of a plaintext m_i and an ElGamal encryption of a cryptographic hash function applied to m, namely $h(m)$:

$$< E_y(m_i), E_y(h(m_i)) >=< (g^r, m_i y^r), (g^{r'}, h(m_i) y^{r'}) >.$$

The anonymity of users whose messages have been corrupted cannot be restored, even if a second round of mixing is performed. Suppose the first mix server is corrupt and replaces the input of user 1 by $< E_y(m_1, r_1)E_y(m_2, r_2), E_y(h(m_1), r_1')E_y(h(m_2), r_2') >$ and the input of user 2 by $< 1, 1 >$. The solution for this problem is to encrypt the input messages twice, so that when the verification takes place, the output messages are decrypted only once, and their contents remain encrypted until all operations are proven to be correctly performed. If this is not the case, the output ciphertexts may become the input to a less efficient mixing algorithm which shuffles them safety before their plaintexts are revealed. A cheating server in the first mixing round may only find relations between an encrypted output and its twice encrypted corresponding input, which does not imply an extra knowledge of the final corresponding plaintext. User i must start from his message $(G_i, M_i) = (g^{\hat{r}_i}, m_i y^{\hat{r}_i'})$ and send to the network the tuple:

$$< E_y(G_i, r_i), E_y(M_i, r_i', E_y(h(G_i, M_i), r_i") >.$$

The first step of the optimistic procedure is the setup. The mix servers generate the ElGamal parameters (p, q, g, x, y). The private key x is shared between mixes in a (k, n) threshold manner. The l users submit their input messages m_i to the mix net:

1. The user encrypts the input m_i and sends to the mix $E_y(m_i) = (G_i, M_i) = (g^{f_i}, m_i y^{f_i})$.

2. The user computes $H_i = h(E_y(m_i))$. In practice, h may be a public cryptographic hash function.

3. The user submits to the mix network the tuple $< E_y(G_i), E_y(M_i), E_y(H_i) >$. The mix servers perform an elliminatory check that every element of the tuple is in the group G_q and that the input is not duplicate. If the input is a duplicate, the message is discarded. If the elements are not in the group, the user is disqualified and the tuple is deleted.

4. The user proves he knows the values of G_i, M_i, H_i in order to not allow other users to post his input. The proof of knowledge is also elliminatory and linked to a unique mix session identifier, so that cross session security is ensured. Some users who send tuples such that $H_i \neq h(E_y(m_i))$ do not force the mix network to follow the less efficient mix protocol. This is followed only when dishonest mixes are found.

The second step of the optimistic procedure is to randomize messages in the same manner as in ElGamal reencryption. Each mix server reencrypts and randomizes all input tuples and proves that it performed these tasks correctly, by showing equalities of message products:

1. Each mix fetches from the bulletin board the shuffled reencryptions of initial tuples $< E_y(G_i), E_y(M_i), E_y(H_i) >$, performed by previous mixes. These tuples have the form $< (g^{r_i}, a_i y^{r_i}), (g^{s_i}, b_i y^{s_i}), (g^{t_i}, c_i y^{t_i}) >$, for any $i \in \{1, 2, ..., l-1, l\}$.

2. The mix randomizes the tuples according to a secret random permutation, by letting the three components of each tuple remain in their order.

3. The mix reencrypts each component of each tuple independently and obtains the output results of the form $< (g^{r_i'}, a_i' y^{r_i'}), (g^{s_i'}, b_i' y^{s_i'}), (g^{t_i'}, c_i' y^{t_i'}) >=< E_y(G_i'), E_y(M_i'), E_y(H_i') >$, for any initial $i \in \{1, 2, ..., l-1, l\}$.

4. The mix proves that $\Pi a_i = \Pi a_i'$, $\Pi b_i = \Pi b_i'$, and $\Pi c_i = \Pi c_i'$. In order to prove this, the mix proves, e.g. for $\Pi a_i = \Pi a_i'$, that $log_g(G_i'/G_i) = log_y(M_i'/M_i)$ by using the Chaum Pedersen protocol. See [18, p. 5] and [20, p. 3] for details.

The third and last step is the decryption phase.

1. A quorum of at least k mixes have access to the secret key x and obtain G_i, M_i and H_i by decrypting the tuples. All tuples where $h((G_i, M_i)) = H_i$ are valid tuples.

2. Invalid tuples are investigated, in order to see if the messages were sent incorrectly by users or cheating mixes are to be found. The mix servers must show the path of each invalid tuple through their permutations. If all invalid tuples were just sent incorrectly by their users, a quorum of mixes decrypts the ciphertexts $E_y(m_i) = (G_i, M_i)$ for all valid tuples and the final output is defined as the set of valid plaintexts.

3. If one mix server is unable to prove the path of an invalid handled message, that particular mix is disqualified, and the quorum of mixes sends the (G_i, M_i) pair to a less efficient mix network. After mixing in this new network has been performed, the messages are decrypted and the original plaintexts m_i are revealed.

A mix network is publicly verifiable, according to the definition of public verifiability stated in [18, p. 12], if there exists a polynomially bounded verifier that takes as input the transcript of the mixing posted on the bulletin board, outputs *valid* if the set of valid outputs is a permuted decryption of all valid inputs, and otherwise outputs *invalid* with overwhelming probability.

The proposed mix net is publicly verifiable if there exists a group G in which the discrete logarithm problem is intractable. The proof of this fact is done by contradiction. The authors assume that one or more mix servers cheat during the mix protocol, and yet produce a transcript that makes an outsider believe the mixing was performed correctly. These cheating mix servers, however, are able to compute discrete logarithms in G, which is a contradiction.

5.5 Proof of Subproduct

In [19], the inventors of Optimistic Mixing propose a solution based on the reencryption properties of the ElGamal cryptosystem which has a constant number of exponentiations with regard to the number l of handled messages. The proof of almost entirely correct mixing is based on the following scenario: a computationally bounded prover convinces a verifier that for two sets of n elements, namely $M = \{m_1, m_2, ..., m_{l-1}, m_l\}$ and $M' = \{m'_1, m'_2, ..., m'_{l-1}, m'_l\}$, there exists a permutation ψ of l elements, such that for all $1 \leq i \leq l$, $m'_i = m_{\psi(i)}$. The prover must not reveal information about the sets M and M', and he must show as little information about the permutation ψ as possible.

The first step of the protocol is the setup. The quorum of decryption mix servers generate the ElGamal parameters (p, g, x, y) in a group of order p with the generator g. The private key x, such that $y = g^x$ is shared among decryption mixes by using a (k, n) secret sharing threshold scheme. The servers publish the parameters (q, g, y). These parameters may stay the same until mix servers enter or leave the network. The mix servers also agree on a security parameter α, which is a small integer (higher values for α yield to a stronger guarantee of correct mixing, but decrease the level of privacy).

The second step of the protocol consists of the input message submission by network users. Each user submits his message m_i as the ElGamal ciphertext $(g_i^r, m_i y^{r_i})$ to the bulletin board and proves the knowledge of m_i.

The third step in the protocol is the shuffle and reencryption of messages by each mix sequentially. Each mix, starting from M_1, reads the messages from the bulletin board, performs the operations described below, and writes the result messages back to the bulletin board:

1. The mix M_j reads l ElGamal ciphertexts of the form $C_i = (g^{r_i}, m_i y^{r_i})$ from the bulletin board.

2. The mix M_j randomizes and reencrypts the ciphertexts, yielding l messages of the form $C'_i = (g^{r'_i}, m'_i y^{r'_i})$ that are sent to the bulletin board. The mix must retain and keep secret its permutation ψ_j and the reencryption factors r'_i in order to respond to any further verification.

The fourth step is the verification. During this step, no mix is allowed to abort in order to not be considered a cheater and remain in the network. All mixes generate a random string r. Each mix M_j proposes a random string r_j, from which the final random string r will be computed by applying the $\oplus$ (xor) operator. Each mix M_j then proves it performed all operations correctly by following the next steps:

1. The two values in C'_i are checked to be in the group $\mathbb{G}$, and then, by using the Chaum Pedersen protocol [20], the mix server M_j proves that $\Pi m_i = \Pi m'_i$, by taking into account all $1 \leq i \leq l$.

2. By using the randomly generated string r, all mixes generate α subsets S_1, S_2, ..., $S_{\alpha-1}$, S_α, where each set S_i is a subset of $\{1, 2, ..., l-1, l\}$. These sets are generated independently and every index $1 \leq k \leq l$ is included in S_i independently at random with a probability of $1/2$. The index k is included in S_i for the mix M_j iff the least significant bit of $h(r||B||i||j||k)$ is 1, where h is a hash function and its argument is formed by concatenation ($||$). The mix M_j receives the α generated sets in order to prove it performed the mixing operations correctly.

3. The mix M_j must produce α subsets S'_1, S'_2, $S'_{\alpha-1}$,S'_α of $\{1, 2, ..., l-1, l\}$, such that for each $1 \leq i \leq \alpha$, $|S_i| = |S'_i|$ and $\Pi m_k = \Pi m'_k$, for each $k \in S_i$. This proof may also be performed by using the protocol in [20].

4. If the quorum of decryption mixes confirm that mix M_j was cheating, the mix is removed from the network and the remaining honest mixes restart the protocol, by using the initial input messages stored by the bulletin board. If no dishonest mix was found, the quorum mixes decrypt and prove the decryption of all messages m_i.

The authors of this proof strategy studied in [19] the properties of their mix network. The cost of proving that the mixing is almost entirely correct needs $2\alpha(2n-1)$ exponentiations per mix and the cost of decryption is $l(4n+2)$. Cheating will be detected with probability $1-(5/8)^\alpha$ on the assumption that the discrete log problem in $\mathbb{G}$ is intractable.

Conclusion

The table 5.1, based on [8, p. 3] summarizes the cost per server of handling n messages with k servers, and also shows cryptographic properties that are traded off with performance. The analyzed procedures are reencryption ($\mathcal{R}$), proof ($\mathcal{P}$) and decryption ($\mathcal{D}$). The value α is a constant security parameter involved in the protocol where it appears.

scheme	$\mathcal{R}$	$\mathcal{P}$	$\mathcal{D}$	trade off
5.1 Cut and Choose [17]	$2n$	$642nk \in \mathcal{O}(nk)$	$n(2+4k)$	
5.2 Pairwise Permutations [21]	$2n$	$7nlogn(2k-1) \in \mathcal{O}(nk)$	$n(2+4k)$	
5.3 Random Partial Check [25]	$2n$	$n/2(2k-1) \in \mathcal{O}(nk)$	$n(2+4k)$	private
5.4 Optimistic Mixing [18]	$6n$	$6+12k \in \mathcal{O}(k)$	$n(5+10k)$	
5.5 Proof of Subproduct [19]	$2n$	$2\alpha(2k-1) \in \mathcal{O}(k)$	$n(2+4k)$	correct

Table 5.1: Cost per server of mixing n items with k mix servers

In the next chapter we will briefly describe some deployed systems based on mix networks.

Chapter 6

Mix Deployed Systems

The basic building block for anonymity systems of high latency is the mix. However, low latency systems, where speed of message delivery is a more important factor in establishing their performance, are based on proxies. While mixes batch, maintain pools, reencrypt and shuffle messages, proxies simply forward incoming messages without any reordering and therefore they deliver the messages way faster than mixes. By closely following the examples provided in [8], this chapter surveys some models that provide an acceptable balance between strong anonymity and great performance.

Each section of this chapter briefly describes the modus operandi of some practical implementations of these systems: sections 6.1 and 6.2 present the anonymizer and crowds systems, respectively; section 6.3 describes the Onion Routing procedure; sections 6.4, 6.5 and 6.6 show some systems based on approaches similar to the Onion Routing, among which some are deployed at a large scale.

6.1 Anonymizer

Anonymizer Inc is a company that provides proxy services to customers who pay for a subscription. Based on the service description on their website [28], anonymizer protects the privacy and personal information, such as credit card numbers, financial transactions et. al. from abuse by redirecting all Web traffic of the client through its secure servers. This yields to making the IP adress of the client private, and therefore his identity protected. One of their products, which is designed for home users, creates an encrypted tunnel between the client computer and the anonymizer servers. They also provide email services that enable the client to create disposable email addresses which can be configured such that every message received on the disposable email address is sent by anonymizer to the real email of the customer.

6.2 Crowds

Crowds [29] was designed for Web browsing anonymously. Users in the system are represented by processes which are assigned to a crowd by an administrative blender process. The blender also informs new processes of other existing processes from within the crowd. When the browser of a user issues a Web request message m, his process selects a random path through the network by first picking a process (possibly itself) from the crowd and then forwarding the request to that process. That process then flips a coin biased with a forwarding probability $p_f > 1/2$. If forwarding is decided, then another process is selected and the procedure continues. If this is not the case, the message is transmitted to the recipient. Each process may retain the process it received the message m from, in order to direct a possible reply to the request sender. Connections between $friend$ processes are maintained by secret key sharing.

When a process receives a message request m, it is unable to know if the process it received the message from is the original sender or it is another hop. The designers of crowds proved [29, p. 11] that if, for c dishonest processes in a total of n processes,

$$n \geq (c+1)\frac{p_f}{p_f - \frac{1}{2}}$$

then no honest predecessor is more likely to be the sender than a simple hop in the network. It is simple to observe the trade-off that as p_f increases, the number of processes n must increase as well.

6.3 Onion Routing

The low latency anonymous communication model called Onion Routing is described in [30]. Onion routing does not relay traffic for other clients, but it has a set of n servers R_1, R_2, ..., R_{n-1}, R_n that relay traffic for clients in the network. Each onion router has a public and private key pair.

To establish an anonymous connection over a network of onion routers, the sender must construct a circuit through the network by selecting an ordered sequence of servers, in a similar way to the mix network. If in the case of mix networks, asymmetric operations performed by the sender were suited for email purposes, for Web traffic, where Web pages may be loaded in less than one second, this approach would have dramatic impact on performance. In order to minimize this problem, users in the network use public key cryptography to establish the encrypted circuit in conjunction with faster symmetric cryptography for the actual message transfer.

The sender generates two random symmetric secret keys: a forward key kf_i and a backward key kb_i, for each router R_i on the path he selected. He also chooses forward an backward cryptographic functions f_i and f_i^{-1}, that correspond to the symmetric keys previously generated. The pair $< kf_i, f_i >$ is used to encrypt data as it traverses the path from the sender to the receiver, and the pair $< kb_i, f_i^{-1} >$ is applied to encrypt data on the way back, from the responding receiver to the sender. If the sender chose the path $P =< P_1, P_2, ..., P_{m-1}, P_m >$, the initiator computes the following values

- $e_m = E_m(t_m, f_m, kf_m, f_m^{-1}, kb_m, \emptyset)$,
- $e_{m-1} = E_{m-1}(t_{m-1}, f_{m-1}, kf_{m-1}, f_{m-1}^{-1}, kb_{m-1}, R_m, e_m)$,

...

- $e_2 = E_2(t_2, f_2, kf_2, f_2^{-1}, kb_2, R_3, e_3)$,
- $e_1 = E_1(t_1, f_1, kf_1, f_1^{-1}, kb_1, R_2, e_2)$,

where t_i is the expiration time of the onion, and $\emptyset$ is used to indicate that P_m is the last onion router for the message. The sender sends e_1 to router R_1, which is able to remove the first layer of encryption and access t_1, f_1, kf_1, f_1^{-1}, kb_1, R_2 and e_2. It pads the remaining encrypted content e_2 with a random bitstring for constant length of onions on their way and sends the result to R_2. Each server repeats the procedure until the onion reaches the end of its path.

After the circuit is constructed, the sender uses the sent forward symmetric keys to encrypt the (Web request) message by using in this order kf_m, kf_{m-1}, ..., kf_2 and then kf_1, and he sends the encrypted quantity to R_1. Each router R_i is able to use f_i and kf_i to decrypt the received message and forward it to R_{i+1} or to the recipient, if $i = m$.

When the response is sent through the reverse path, each server R_i encrypts the content with the backward symmetric key kb_i and the backward function f_i^{-1}. After receiving the response message, the initial sender removes all layers of encryption by using the secret keys he knows.

6.4 Tarzan

Tarzan [31] is an anonymity system in which senders construct circuits through the network by generating symmetric keys and using public keys of the servers in the circuit to encrypt them, in a similar way to the Onion Routing protocol. Each new participant in the network finds other servers by selecting a random neighbor server (suppose each new node in the network is given a set of initial servers) and by finding all its known servers. The requester can then randomly select another server in the set he learnt and repeat the procedure.

In order to obscure traffic patterns and prevent global passive adversaries, Tarzan adds cover traffic to the network. When a new node joins the network, it selects k *mimic* nodes, based on a hash function applied on the IP of the node and the moment in time. Mimic selections are therefore universally verifiable. Mimic selection is symmetric, so that messages will generate traffic in both ways. Mimic messages is indistinguishable to observers from application messages. The rates at which a node sends and at which a node receives messages are similar.

When a node is to send a message, it randomly selects the first hop from the set of mimics it knows. The second hop is selected by the first hop from its own set of mimics, and the process continues. Application messages use a layered encryption similar to Onion Routing. As the sender also exchanges mimic traffic with other nodes, an observer cannot identify it as a source of a circuit and follow the message. Note that this applies to any hop in the network.

6.5 MorphMix

MorphMix [32] is another peer to peer low latency anonymity system. Its procedures, however, are more complex than the case of Onion Routing. The circuit initiator A selects a node B from the set of known nodes. A then requests from B a subset of the nodes it knows, in order for A to select the next hop of the circuit. Let C be the next hop chosen from A in this manner. Let W be another node A knows, which will *witness* the procedure.

Node A creates a symmetric key with C by sending to B half of a Diffie Hellman handshake result, encrypted with the public key of W. B sends the cipher text to W, who forwards the result to C. C generates its own half of the Diffie Hellman handshake result and it sends it to A via B. Both nodes A and C are able to share their symmetric key, and they are the only nodes which known both Diffie Hellman results. The initiator of the circuit repeats this procedure for each hop in the circuit of its messages.

The scheme was proven unsecure to an attack when a malicious node in the role of B only selects witness and next nodes it colludes with. Even though the mecanism proposed in [32], of detecting collusion by analyzing the tracks where colusing nodes appear together with higher probability, the approach can be defeated by offering sets of colluding nodes that intersect with little probability.

6.6 Tor

Tor [33] is a large scale deployed anonymity system, which at the moment consists of more than 1000 servers worldwide. According to their website [34], *Tor is free software and an open network that helps you defend against a form of network surveillance that threatens personal freedom and privacy, confidential business activities and relationships, and state security known as traffic analysis*.

Some servers, called *directory* servers, aggregate and distribute signed data about known routers. This information can be replicated by other routers in order to reduce load on the main servers. The initiator of a circuit negotiates short lasting session keys with each router in the path by using the Diffie Hellman protocol. When a key is established with the first hop, the initiator tunnels the message through the encrypted connection and

establishes an ephemeral session key with the second hop, and the procedure continues. The session keys are discarded after the usage of a circuit.

Tor also allows users to provide services, such as Web servers, that accessible to every Tor client, but without having revealed the location of that particular server. Hidden servers have connections to nodes in the network that are known as introduction points. The clients connect to these points and specify a rendezvous point. Both client and hidden server connect to that rendezvous node and communicate through an encrypted tunnel.

Conclusion

This chapter described models of practical implementations of mix networks. Some of these models are large scale and serve millions of clients. In the next chapter we will describe a parallel approach of mix networks and provide an application model for this paradigm built on top of an open source distributed computing framework.

Chapter 7

Case Study

The Parallel Mix Network model, described in [26], brings a substantial reduction in overall mixing time of messages. We will closely follow the structure of the whole article to describe the model. At any moment in a synchronous mix network, one server is processing the messages and all others are idle. A very significant part of computational resources are thus wasted. As opposed to the synchronous mix networks, which for l inputs and n mix servers provide an output in $\mathcal{O}(nl)$, by parallelizing the operations, their model produces an output in at most $2n$ units of time.

This parallel model can be integrated with the Map Reduce programming model and implementations. We briefly analyze this programming model and the Apache Hadoop open source distributed computing framework and we then describe our application model based on all these. We may leave our implementation agnostic of the underlying cryptographic protocols, which is in concordance with the Parallel Mix Network properties.

7.1 Parallel Mix Networks

The technique meets the privacy requirements of a synchronous mix network, but with time expenses dropped by a factor of $\mathcal{O}(n)$. When $n > 2$ and the required communication latency is small, the parallel mix network may recommend itself for otherwise time consuming applications such as electronic voting and exit polls.

The model offers security guarantees that are comparable to those provided by synchronous networks. The model can be applied to many reencryption mix networks, such as those described in [19] and [21]. Throghout these mix models, the main operations of mix nets remain unchanged. The mix network receives as input l ciphertexts, reencrypts/decrypts these ciphertexts and outputs l messages in a randomly permuted order of the input messages. This model is built on top of reencryption mix networks, rather than decryption nets, and it does not focus on the final decryption step that reencryption networks follow.

The most simple approach towards parallelizing the operations of a mix network is to divide the l inputs into b batches of size l/b and send these smaller batches to the synchronous mix network as fast as possible, in order to keep the servers full. The time in which all messages are processed by the mix network this way is $\mathcal{O}(n+b)lb^{-1}$. For instance, if $b = \mathcal{O}(\sqrt{n})$, the time in which all messages are processed by the mix network is $\mathcal{O}(l\sqrt{n})$. The drawback of this approach is that it loses privacy: each imput message is mixed with other l/b messages rather than all l input messages.

In both types of mix networks (synchronous/sequential and parallel), each server implements its own mixing operations on the sets of input ciphertexts. For instance, each server applies a secret, random permutation. Robust mix networks also include cryptographic procedures (such as [18], [19], [20]) to prove they performed the operations correctly. The proposed model does not take into account these proof techniques, as its parallel construction is *agnostic* of the underlying cryptographic proof algorithms, but rather it provides a way to perform many local operations on small data, as opposed to the large and slow

operations on the full set of l messages. The model relies on some ideal assumptions:

- Ideal correctness: the authors assume that a perfect public verifiability is possible, as each server that misbehaved is detected. They full verifiability is on the fact that a server has executed a valid permutation, rather than a truly random permutation.
- No communication latency: they assume that data is transmitted in constant time between two servers. This assumption allows them to focus on the computation time, which in practice is substantially higher than the cost of communication, e.g. when the input size is large.
- Linear local operational costs: it is assumed that the proof and reencryption operations for the local mix operations are linear in the number of input messages. The full processing of any single message is treated as a *step*.
- Bulletin board: the model uses a publicly accessible memory with universal read and append rights. If an adversary controls the majority of servers, they assume the bulletin board is safely maintained by a trusted entity.
- Ideal randomness source: a publicly verifiable and ideal source of random is assumed. In practice, this may be achieved by a distributed protocol among all involved servers.

The authors define their construction essentially from a combinatorial perspective, and their basic measure of efficiency is the total time required to run the mix network on an input of l messages, when all servers perform their local operations and communication procedures correctly.

The mix network construction is structured in terms of a sequence of synchronous rounds. A round is a period of time during which mixes execute local operations without communicating. After a round ends, servers may exchange batches of message ciphertexts. Let r be the number of rounds in the construction. At the beginning of each round of mixing, the mix network operates on an ordered set of l ciphertexts. The authors have modelled a vector of l slots, so that initially each input i ciphertext occupies the slot i. The operation of a mix network is defined by a schedule that assigns each mix to a subset of slots in a round. The mix permutes and reencrypts the ciphertexts from the slots it was assigned, in order to hide the permutation.

The article [26] uses the following notations:

- the symbol $k \in \{1, 2, ..., r-1, r\}$ is used for indexing rounds,
- the symbol $j \in \{1, 2, ..., n-1, n\}$ is used for indexing mixes,
- the symbol $i \in \{1, 2, ..., l-1, l\}$ is used for indexing ciphertexts,
- the symbol $s \in \{1, 2, ..., l-1, l\}$ is used for indexing slots.

In the beginning, the ciphertext i occupies the slot s, where $s = i$. In the future, there may or may not be the case that $s = i$.

Based on the assumption that mixes perform their tasks correctly, the *path* of an input message through the mix network is the ordered list of slots $< s_1, s_2, ..., s_{r+1} >$ occupied by that message in the course of mixing. Before the g^{th} round ($g \in \{1, 2, ..., r-1, r\}$), the ciphertext occupies the slot s_g. The final slot occupied by the message is indexed as s_{r+1}. For two identical plaintext messages, the path may be seen as the permutations performed by the mixes.

This way, the Parallel Mix Network is defined by the assignment of slots to mixes before each round. Let $S_k(j) \subseteq \{1, 2, ..., l-1, l\}$ be the set of slots assigned to mix j in round k. A slot is never assigned to more than one server in the same round ($S_k(j_1) \cap S_k(j_2) = \emptyset$, for any two mix servers $j_1 \neq j_2$). However, the set of sets $S_k = \{S_k(j)\}_{j=1}^{n}$ may not represent a partition of $\{1, 2, ..., l-1, l\}$, as some slots may not be assigned to any mix in round k. In a Parallel Mix Network described as

$$S = \bigcup_{k=1}^{r} \left(\bigcup_{j=1}^{n} S_k(j) \right)$$

the computational cost for the server j is

$$\sum_{k=1}^{r} |S_k(j)|.$$

This way, the total cost of mixing is

$$\sum_{k=1}^{r} max_j(|S_k(j)|).$$

Synchronous networks may also be described by using these notations: the number of rounds is the same as the number of servers in the mix network, i.e. $r = n$. For each $1 \leq j, k \leq n$, server j is assigned to the whole vector of slots, but only at round $k = j$:

$$S_k(j) = \begin{cases} \{1, 2, ..., l-1, l\}, & k = j, \\ \emptyset, & \text{otherwise.} \end{cases}$$

The cost for a single server is $\mathcal{O}(l)$, and the total mixing time is $\mathcal{O}(nl)$.

The authors state that two sets $S = \{S_k(j)\}$ and $T = \{T_k(j)\}$, for $j = 1, 2, ..., n-1, n$ and $k = 1, 2, ..., r-1, r$ that define two Parallel Mix Networks on l messages with r rounds are isomorphic if there exists a permutation π of the l slots such that $S_k(j) = \pi(T_k(j))$, for all k and j. Therefore, the mix network isomorphism spans to a reordering of the slot indexes.

In this model, by having assumed ideal correcness of mixes, there is no need for considering an active adversary. By having assumed public verifiability, the adversary $\mathcal{A}$ may control up to $n-1$ mixes. Let $\mathcal{A}(M)$ be the set of mix servers controlled by the adversary. The mixes in this set perform their local operations correctly, but the adversary has access to all permutations performed by them. In the case when $\mathcal{A}(M) = \emptyset$, the adversary is *external* and he has access to all inputs and outputs of the mix servers.

As a minimum requirement in order for privacy to be meaningful, suppose the adversary is able to find the paths of at most $l-2$ messages. More precisely, the adversary selects $n-2$ messages and he learns, for each such message, which slot the message is taken from in the first round and which slot the message reaches after all mixing operations are done. Let $\mathcal{A}(I)$ denote the subset of slots initially occupied by messages that are controlled by the adversary and $\mathcal{A}(O)$ the subset of slots in which these messages end up after all mixing procedures are completed.

The authors define the anonymity formally by considering the probability $\mathcal{P}_{\mathcal{A}}(k, s_0, s_1)$ that the path of a message until round k starts with s_0 and ends with s_1 or, alternatively, the probability that the ciphertext occupying slot s_1 after the end of round k was located in the slot s_0 before the mix procedures began. An input message in a slot s can be traced to its source immediately:

$$\mathcal{P}_{\mathcal{A}}(0, s, s') = \begin{cases} 1, & s = s', \\ 0, & \text{otherwise.} \end{cases}$$

The anonymity at slot s after round k in what matters adversary $\mathcal{A}$ is defined as

$$anon_k(s) = (max_{s_0}(\mathcal{P}_{\mathcal{A}}(k, s_0, s)))^{-1}$$

and the anonymity of the mix network for round k with respect to $\mathcal{A}$ as

$$anon_k = min_s(anon_k(s)).$$

For instance, if $anon_k = n$, then the adversary learns no information about the relation between inputs and messages in mixing round k; if $anon_k = 1$, there exists one message whose path is known to the adversary. For the last round, by assuming that $\mathcal{A}$ knows some mappings of input messages to output messages, i.e. $anon_r(s) = 1$ for all $s \in \mathcal{A}(O)$, the authors refine the anonymity at round k as

$$anon_r = min_{s \notin \mathcal{A}(O)}(anon_r(s)).$$

By assuming that mix j is not under control by $\mathcal{A}$, in round k it processes $S_k(j)$. For all slots $s_1 \in S_k(j)$,

$$\mathcal{P}_{\mathcal{A}}(k, s_0, s_1) = \frac{1}{|S_k(j)|} \sum_{s \in S_k(j)} \mathcal{P}_{\mathcal{A}}(k-1, s_0, s).$$

Indeed, a message could have occupied in its path slots from s_0 to $s \in S_k(j)$ with the probability $\mathcal{P}_{\mathcal{A}}(k, s_0, s)$. The probability decreases by a factor of $|S_k(j)|$ due to the secret permutation which is unknown to the adversary. If mix j is, however, controlled by $\mathcal{A}$, the probability remains the same as for the previous round. For all $s_1 \in S_k(j)$ we will have

$$\mathcal{P}_{\mathcal{A}}(k, s_0, \pi_{j,k}(s_1)) = \mathcal{P}_{\mathcal{A}}(k-1, s_0, s_1),$$

where $\pi_{j,k}(s_1)$ is the slot in which the ciphertext from slot s_1 will be put after round k performed by mix j on $S_k(j)$.

To simplify the protocol model, it is assumed that $n^2|l$, i.e. the number of messages is a multiple of the squared number of mixes. This condition can be accomplished by adding dummy input messages. The mixing round k is an equiround if

$$|S_k(j)| = l/n, \text{ for all } j \in \{1, 2, ..., n-1, n\}.$$

The model relies on two basic operations:

1. Rotation: a rotation means that the subset of slots that ar assigned to the mix server j in round k will be assigned to the server $j+1$ in round $k+1$. Formally, mixing round $k+1$ is a rotation of mixing round k if

$$S_{k+1}(j) = S_k((n + j - 2) \bmod n + 1), \text{ for all } j \in \{1, 2, ..., n-1, n\}.$$

2. Distribution: the slots of every server are uniformly assigned across all servers in the network. By assuming that $n^2|l$ and that mixing round k is an equiround, mixing round $k+1$ is a distribution of mixing round k if

$$|S_{k+1}(j) \cap S_k(j')| = l/n^2, \text{ for all } j, j' \in \{1, 2, ..., n-1, n\}.$$

   This implies that $k+1$ is an equiround as well, because $|S_{k+1}(j)| = nl/n^2 = l/n$.

For the presented model, the adversary $\mathcal{A}$ is able to control a number of mix servers and know the paths of at most $l-2$ messages. Let the the number of mix servers that are controlled by $\mathcal{A}$ be n', where $n' < n$. For $j \in \{1, 2, ..., n-1, n\}$, also let $B_S(j) = S_1(j)$ and $C_S(j) = S_{n'+2}(j)$. The protocol follows four steps:

1. The first round is an equiround. According to the assumed ideal source of randomness, the slots are divided at random into buckets $B(1)$, $B(2)$, ..., $B(n)$ such that bucket $B(j)$ is assigned to mix j. The adversary cannot control the partition of slots into buckets. Each server mixes the assigned bucket.

2. The rounds $1, 2, ..., n'-1, n'$ (a total of n' rounds) are each a rotation of the previous round: $S_{k+1}(j) = S_k((n + j - 2) \bmod n + 1)$, for all $k \in \{1, 2, ..., n' - 1, n'\}$ and $j \in \{1, 2, ..., n-1, n\}$. In this step, each server passes its buckets to the next server in order (the last server passes the buckets to the first). By the end of this step, every bucket will be processed by at least one honest mix, because the total number of mixing rounds is $n' + 1$, and at least n' servers are corrupt. Therefore, the messages that are found in the slots of every bucket are shuffled in that particular bucket from the point of view of $\mathcal{A}$. This way, a specific message can be traced in an input set of size at least l/n, and $anon_k \geq l/n$.

3. The round $n' + 2$ is a distribution of round $n' + 1$. In the round $n' + 1$, the original buckets are redistributed into a new set of buckets $C_S(1)$, $C_S(2)$, ..., $C_S(n-1), C_S(n)$. After this step, every bucket $C_S(i)$ contains slots from each original set of buckets $\{B_S(1), B_S(2), ..., B_S(n-1), B_S(n)\}$ in equal quantities. The level of anonymity of a message does not change, since this step can be viewed by the adversary $\mathcal{A}$.

4. The rounds $n' + 2, n' + 3, ..., 2n', 2n' + 1$ (a total of n' rounds) are each a rotation of the previous round, i.e. $S_{k+1}(j) = S_k((n + j - 2) \bmod n + 1)$, for all $k \in \{n' + 2, n' + 3, ..., 2n', 2n' + 1\}$ and $j \in \{1, 2, ..., n - 1, n\}$. This last step consists of the same procedures as in the second step, but they are applied to the buckets $\{C_S(1), C_S(2), ..., C_S(n-1), C_S(n)\}$. Each bucket will be processed by at least one honest server, since the number of mixes is $n' + 1 > n$. The distribution operation in the third step yields to a complete mixing of the messages. In particular, each bucket $C_S(i)$ includes an equal number of slots from each (original) bucket in $\{B_S(1), B_S(2), ..., B_S(n-1), B_S(n)\}$, $\mathcal{A}$ is unable to trace a given output message to its original input bucket $B_S(j)$. Since each original bucket $B_S(j)$ was also mixed by at least one honest server, $\mathcal{A}$ will not trace an output to a corresponding input, with a probability of success higher than the probability of success of a random guess.

The figure 7.1 shows a parallel mix network with $n = 2$ servers and $n' = 1$. The origin slots of messages, denoted by α and β, are shown only where the order is known, and before mixing takes place. The horizontal lines through mixes are quantitative guidelines only. In fact, the messages in slots are shuffled before they are sent to the next mix server, so the order does not keep. Therefore, before the last mix operation, the original slots are already unknown. This explains that when no mix is corrupt, no rotation operations are necessary. Also note that, by isomorphism, this representation is valid for more than one Parallel Mix Network.

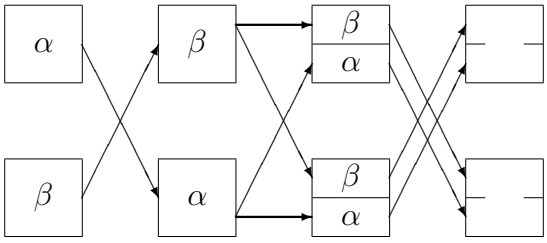


Figure 7.1: This model ensures the same anonymity as synchronous mix nets

The authors of [26] show that the operations define a unique mix network model up to isomorphism. By letting $\{S_k(j)\}$ and $\{T_k(j)\}$ be two sequential mix networks with the properties above, $\sigma_{j_0,j_1} = B_S(j_0) \cap C_S(j_1)$ and $\tau_{j_0,j_1} = B_T(j_0) \cap C_T(j_1)$, the sets $\{\sigma\}$ and $\{\tau\}$ are partitions of $\{1, 2, ..., l - 1, l\}$ and each of them has the size l/n^2, according to the third step. There exists a permutation π such that messages in σ_{j_0,j_1} are mapped to messages in τ_{j_0,j_1}. This permutation defines an isomorphism between $\{S_k(j)\}$ and $\{T_k(j)\}$, since the first, second and last step in the protocol change the order of messages, but not the contents of these sets.

The adversary $\mathcal{A}$ knows the slots $\mathcal{A}(I)$ and $\mathcal{A}(O)$, that contain the slots of $|\mathcal{A}(I)| = |\mathcal{A}(O)|$ messages before and after the protocol ends. For each $j \in \{1, 2, ..., n-1, n\}$, let $\alpha(j) = |\mathcal{A}(I) \cap B_S(j)|$ be the number of slots that are in the input bucket $B_S(j)$ and contain a message controlled by the adversary, and $\gamma(j) = |\mathcal{A}(O) \cap C_S(j)|$ be the number of slots that are in the output bucket $C_S(j)$ and contain a message controlled by $\mathcal{A}$. By defining $\delta(j_0, j_1)$ as the number of slots that belong to both buckets $B_S(j_0)$ and $C_S(j_1)$ and that contained a message controlled by the adversary at the end of the step,

$$\mathcal{P}_{\mathcal{A}}(r, s_0, s_1) = \frac{\frac{l}{n^2} - \delta(j_0, j_1)}{(\frac{l}{n} - \alpha(j_0))(\frac{l}{n} - \gamma(j_1))}, \text{ for any } s_0 \in B_S(j_0) \text{ and } s_1 \in C_S(j_1).$$

In [26], the authors prove this result as follows. Let $\mathcal{P}_0$ be the probability that, from the point of view of $\mathcal{A}$, the input message that was put in slot s_0 before the start of the protocol landed in the output bucket $C_S(j_1)$ after the distribution round. Let also $\mathcal{P}_1$ be probability that, under the assumption of $\mathcal{A}$, a message that landed in $C_S(j_1)$ after the distribution round ends up in the slot s_1 after the protocol is finished. It is easy to see that $\mathcal{P}_{\mathcal{A}}(r, s_0, s_1) = \mathcal{P}_0\mathcal{P}_1$, for any fixed s_0 and s_1.

In order to compute $\mathcal{P}_0$, note that each output bucket contains a mixture of elements from each input bucket, in equal proportions. By considering $B_S(j_0)$, $\mathcal{A}$ knows the output bucket $\alpha(j_0)$ that was assigned for each message in $B_S(j_0)$. The $l/n - \alpha(j_0)$ messages that remain are assigned uniformly at random to buckets of the type $C_S(\cdot)$. However, $\mathcal{A}$ already knows that $\delta(j_0, j_1)$ messages were moved from $B_S(j_0)$ to $C_S(j_1)$. Therefore, the probability is computed regardless of the messages whose paths are known to $\mathcal{A}$:

$$\mathcal{P}_0 = \frac{\frac{l}{n^2} - \delta(j_0, j_1)}{\frac{l}{n} - \alpha(j_0)}.$$

The n' rotations after the distribution step have the property that each message passes at least one honest server. Therefore, all elements that are not in a slot that is part of $\mathcal{A}(O)$ are evenly distributed. Thus,

$$\mathcal{P}_1 = \frac{1}{\frac{l}{n} - \gamma(j_1)},$$

and this yields the value of $\mathcal{P}_{\mathcal{A}}(r, s_0, s_1)$.

As a consequence to the following statement, by assuming that the adversary $\mathcal{A}$ does not know any input to output mapping, i.e. $\mathcal{A}(I) = \mathcal{A}(O) = \emptyset$, the Parallel Mix Network has the anonymity l with regard to $\mathcal{A}$. Indeed,

$$\mathcal{P}_{\mathcal{A}}(r, s_0, s_1) = \frac{\frac{l}{n^2} - \delta(j_0, j_1)}{(\frac{l}{n} - \alpha(j_0))(\frac{l}{n} - \gamma(j_1))} = \frac{\frac{l}{n^2} - 0}{(\frac{l}{n} - 0)(\frac{l}{n} - 0)} = \frac{1}{l}.$$

Even though the values of $\alpha(j_0)$, $\gamma(j_1)$ and $\delta(j_0, j_1)$ are equal to their mean values for all j_0 and j_1, the parallel mix network keeps the anonymity for the messages with unknown paths. By choosing $expected(\alpha(j_0)) = expected(\gamma(j_1)) = |\mathcal{A}(I)/n|$ and $expected(\delta(j_0, j_1)) = |\mathcal{A}(I)|/n^2$,

$$\mathcal{P}_{\mathcal{A}}(r, s_0, s_1) = \frac{\frac{l}{n^2} - \delta(j_0, j_1)}{(\frac{l}{n} - \alpha(j_0))(\frac{l}{n} - \gamma(j_1))} = \frac{\frac{l}{n^2} - \frac{|\mathcal{A}(I)|}{n^2}}{(\frac{l}{n} - \frac{|\mathcal{A}(I)|}{n})(\frac{l}{n} - \frac{|\mathcal{A}(I)|}{n})} = \frac{1}{l - \mathcal{A}(I)}.$$

The authors of [26] also prove that if an adversary $\mathcal{A}$ controls $n' = n - 1$ mix servers, a Parallel Mix Network with anonymity $anon_r = l$ with regard to $\mathcal{A}$ has a total mix time in number of steps at least $2l$. Their protocol achives the optimal total time of mixing if $n' = n - 1$.

7.2 Map Reduce Programming Model

The Map Reduce functional programming model and associated implementation are useful for real World processing and generating large data sets. We will briefly describe the technique by closely following its original source [35]. Users specify two functions:

- the *map* function processes a key value pair to generate a set of intermediate key value pairs,
- the *reduce* function merges all intermediate values associated with the same intermediate key.

The computation takes as input a set of key value pairs and produces an output of key value pairs. The *map* function takes a single input pair and generates a set of intermediary key value pairs. The map reduce implementation groups all intermediate values that have the same key and passes them to the reduce function. The *reduce* function accepts as input an intermediate key k and the set of values with that key and it outputs a result for those values in order to form a smaller set of values. The intermediate values are supplied to the reduce function via in iterator. This allows handling large amounts of values that may not fit in the limited memory of the machine.

The most simple practical problem for which this model provides an elegant distributed solution is to count the number of occurences of each word in a large collection of documents. The *map* function for this example would receive the file name as the key and the contents as the value, and it would emit as intermediary pairs each word as the key, among with the value 1, to state that there is an occurence of that word. The pseudocode for the *map* function may be seen in 7.2.

Input: document name as key
Input: document contents as value
Output: word and 1 as intermediate pair

```
1 for word ∈ value do
2 ⌊ [output] ← (word, 1)
```

Figure 7.2: Original example *map* function

The *reduce* function for this example would receive a word as a key and an iterator in which there are as many values as the number of appearances of the word which is the key in all documents processed by the *map* function. This function may conveniently return the sum of those values. The pseudocode for the *reduce* function can be consulted at 7.3.

Input: word as key
Input: iterator of numbers as values
Output: word and 1 as intermediate pair

```
1 result ← 0
2 for count ∈ values do
3 ⌊ result ← result + count
4 [output] ← (key, result)
```

Figure 7.3: Original example *reduce* function

Note that the data types of the key value pairs in the input may differ. The user of the map reduce system expresses the computation as the two functions, *map* and *reduce* with particular, but not fixed types. However, the generic types associated to the functions are:

- $map(k_1, v_1) \rightarrow list(k_2, v_2)$,
- $reduce(k_2, list(v_2)) \rightarrow list(v_2)$.'

Therefore, the type restrictions are that the output types from *map* is of the same types as the input for the *reduce*. The map reduce original model was also extended with some useful functionalities:

- Partition: the user specifies the number of reduce tasks per output r. Data is partitioned across these tasks by using a partition function on the intermediary key k. The default partitioning function is $h(k)\ mod\ r$, which results to fairly balanced partitions.
- Order: it is guaranteed that within a given partition, the intermediate key value pairs are processed in increasing order based on their keys. This allows to generate sorted outputs easily.
- Combination: the *combiner* function is executed on each machine that performs a map function. The combiner realizes a partial merge of the results of the map function. For instance, in the detailed example from above, note that we added *count* and not 1. If a combiner would have been employed, its function would have been exactly the function that describes the reduce operation.

Some other simple examples where map reduce is suited are:

1. Distributed grep: the map function emits a line if it matches a given pattern; the reduce function is the identity function.
2. Count of URL access frequency: the map function reads logs of Web page requests and outputs $(URL, 1)$; the reduce function adds up the values received from the mapping. This example is similar to the detailed example from above.
3. Inverted index: the map function parses documents and emits sequences of $(word, document\ identifier)$. The reduce function then sorts the document identifiers and outputs a list of $(word, list(document\ identifier))$. This can be of course enriched to retain positions of the words as well.
4. Sorting: the map and reduce functions are simply the identity functions, and the keys are simply the sorted items. Note that before the reduce phase, all map output is sorted by key.

7.3 Hadoop Map Reduce Framework

Apache Hadoop is, according to the official site [36], an open source software library for reliable and scalable distributed computing. The library provides the functionality of a framework that allows distributed processing of large data sets across clusters of computers by using a simple programming model. The design of Hadoop allows it to scale from single servers to thousands of machines, while handling failures at the application layer on top of a prone to error cluster. One of the components of this project is the software framework called Hadoop MapReduce, which is useful for writing applications that process quickly large sets of data.

In the spirit of the Map Reduce programming model [35], a map reduce job usually splits the input set of data into independent parts, which are processed by map tasks in parallel. Hadoop sorts the outputs of the mappers which are fed as input to the reducers. The framework consists of a single master, called *job tracker*, and one slave per cluster node, called *task tracker*. The master is in charge with scheduling, monitoring and reassigning failed jobs to the slaves, while the slaves execute the tasks as they were ordered.

Applications specify the locations of input and output and supply suitable *map* and *reduce* functions by implementing suitable interfaces or abstract classes. Along with other job parameters, the configuration of the job is done. The Hadoop job client then submits the job configuration to the *job tracker*. As expected, the framework operates on $(key, value)$ pairs, conceivably of different types, under the following format:

$$[in] \rightarrow (k_1, v_1) \rightarrow map \rightarrow (k_2, v_2) \rightarrow combine \rightarrow (k_2, v_2) \rightarrow reduce \rightarrow (k_3, v_3) \rightarrow [out].$$

7.4 Application Model

In order to apply the principles of Map Reduce 7.2 by using Hadoop 7.3 for the scheme presented in 7.1, more than one map reduce operation must be performed. The activity of the mix network may be simulated by a sequence of map reduce operations on ever changing versions of the input file. For simplicity, our application considers all input in a large text file with messages from senders.

By taking this sequential workflow into account, we designed a *moderator* class that executes the map reduce jobs, which are parallel in nature, in a sequential manner. The results of a map reduce job are taken by the next map reduce job in order to analyze further the input. Take for instance the computation of the minimum size of a bucket, as described in 7.1.

At first, messages from *input* are associated to a uniformly at random bucket in the set $0, 1, ..., n-2, n-1$ via a map reduce procedure. The results of this are the messages with their numbers in *associated*. This file is then taken as input by a map reduce job that for each bucket counts the number of messages that are in this bucket and store n output pairs $< bucket, count >$ in the relatively smaller file *counted*. This file is then taken by another map reduce job that finds the minimum of *count* values in the computed set sizes from above and outputs to *minimized* the minimum number of messages in a bucket.

Note that this simplistic approach yields to inexact results. The set of input messages must obey the properties that $l = kn^2$ and that for each bucket j, $B_S(j) = nk$. This yields to having to compute a minimum equal to the largest multiple of n and which does not exceed the minimum size of a bucket, rather than the minimum described at first. This may lead to postponing remaining messages for the next round, by applying a few more specific map reduce jobs that filter the input messages down to exactly n buckets with precisely kn messages in each bucket. This protocol change does not decrease the anonymity of postponed messages, since they will be selected to pass through the network and lose their traces in a future round.

In order to demonstrate that our algorithm in 4.10 is functional, we may employ it in our mix network as the flushing algorithm, with N initial dummy messages per mix server. This technique may delay a selected message for up to $\mathcal{O}(Qknn')$ rounds, because each mix server allows a maximum delay of $\mathcal{O}(Q(kn))$ rounds, and the number of mix servers a message passes is $\mathcal{O}(n')$. To minimize the delay, we may implement this algorithm with a pool of $N = 0$; this limitation will not interfere in any way with its correct functioning.

As our algorithm is only a possibility of choice for the flushing algorithm, we left the application model agnostic of the internal cryptographic protocol that is to be used and thought the mix server in itself as a programme able to employ any of the discussed flush algorithms. This property was also stated by the authors of the Parallel Mix Networks in [26, p. 2].

Our demonstration application is split into modules, which consist of packages, and inner classes for each package. In figure 7.4 the UML class and package diagrams can be seen (details on UML can be found in [44]). The *Moderator* class uses subclasses of the abstract class *EntryPoint* in order to perform the map reduce jobs in a sequential manner. For instance, the most complex such subclass is *Message Randomizer Entry Point*. In the *run* method of this class, according to the value of the *redistribute*, the job invoked is configured to have as the reduce class either *Message Randomizer Redistribute Reducer* (which selects k messages for each of the n servers and sets their associated values accordingly), or *Message Randomizer Proceed Reducer* (which sets all messages of value v with the associated value $(v + 1)$ *mod* n). The invoked job is also configured to have *Message Randomizer Mapper* as the map class. We may say that this map reduce task has the name *Message Randomizer*.

We chose to implement our application by using the Java framework (see [37] and [38] for details on Object Oriented Programming in Java). Let $input \rightarrow name \rightarrow output$ be a

map reduce task with the name *name*, that takes as input a large folder *input* and yields as output to a large folder *output*. Figure 7.4 depicts the sequence of procedures in the randomize map reduce module.
Further we describe the sequence of map reduce operations to be performed in order to process the messages in run r:

1. $input \rightarrow Probable\ Mix\ Assigner \rightarrow assigned$ is defined only by the map method $Probable\ Mix\ Assigner\ Mapper :: map$, in which each input pair of the form $< \cdot, word >$ is mapped to an output pair of the form $< bucket, word >$, where $bucket \in \{0, 1, ..., n-2, n-1\}$.

2. $assigned \rightarrow Assigned\ Mix\ Counter \rightarrow counted$ is defined by both map and reduce methods. The map method $Assigned\ Mix\ Counter\ Mapper :: map$ takes as input a pair of the form $< bucket, word >$ and outputs a pair of the form $< bucket, 1 >$. The reduce method $Assigned\ Mix\ Counter\ Reducer :: reduce$ takes as input a pair of the form

$$< bucket, list(1_0, 1_1, ..., 1_{v-1}, 1_v) >$$

and outputs a pair of the form $< bucket, count >$, where *count* is the size of the list, namely the number of messages in the bucket v. Note that this job may be optimized by also specifying the reduce method as the combiner.

3. $counted \rightarrow General\ Minimum\ Computer \rightarrow minimized$ is also defined by both map and reduce methods. The map method $General\ Minimum\ Computer\ Mapper :: map$ takes as input a pair of the form $< bucket, count >$ and outputs a pair of the form $< true, count >$. The reduce method $General\ Minimum\ Computer\ Reducer :: reduce$ takes as input a pair of the form

$$< true, list(count_0, count_1, ..., count_{n-2}, count_{n-1}) >$$

and outputs a pair of the form $< true, min >$, where *min* is the minimum as computed above. See that by specifying *true* for all inputs to the map procedure, one reducer will receive all n inputs and process them accordingly, thus yielding to a single pair of the form $< true, min >$.

4. $assigned \rightarrow Message\ Selector \rightarrow selected$ is defined as follows: the map method $Message\ Selector\ Mapper :: map$ takes as input a pair of the form $< bucket, word >$ and outputs the same pair; the reduce method $Message\ Selector\ Reducer :: reduce$ takes as input a pair of the form

$$< bucket, list(word_0, word_1, ..., word_{d-2}, word_{d-1}) >$$

and outputs *min* pairs as $< bucket, word_i >$ and $d - min$ pairs as $< \infty, word_j >$. The pairs that contain a finite number in the *bucket* field will be the final input for our network. The other messages will be sent back to the initial input and processed in a future round. It is easy to see that now each bucket consists of kn messages, as planned.

5. $selected \rightarrow Message\ Filter\ [criteria = true] \rightarrow round_0$ is defined by a map only: $Message\ Filter\ Pass\ Mapper :: map$ takes as input a pair of the form $< bucket, word >$ and outputs the pair if $bucket \neq \infty$. Now $round_0$ is the final input for the mix network.

6. $selected \rightarrow Message\ Filter\ [criteria = false] \rightarrow next$ is defined by a map as well: $Message\ Filter\ Fail\ Mapper :: map$ takes as input a pair of the form $< bucket, word >$ and outputs the pair if, on the contrary, $bucket = \infty$. Now *next* contains the messages that were not selected and that will be used as input along with messages in the next round.

7. $round_k \rightarrow Message\ Randomizer\ [redistribute = false] \rightarrow round_{k+1}$ is defined by the map method $Message\ Randomizer\ Mapper :: map$ that takes as input a pair of the form $< bucket, word >$ and outputs the same pair, and the reduce method $Message\ Randomizer\ Proceed\ Reducer :: reduce$ that takes as input a pair of the form

$$< bucket, list(word_0, word_1, ..., word_{d-2}, word_{d-1}) >$$

and outputs each $< (bucket + 1) \bmod n, word_i >$, but in an order given by a random permutation. This is the step where cryptographic procedures may be employed to select messages uniformly at random. This step follows for $k \in \{0, 1, ..., n' - 3, n' - 2\}$.

8. $round_{n'-1} \rightarrow Message\ Randomizer\ [redistribute = true] \rightarrow round_{n'}$ is defined by the map method $Message\ Randomizer\ Mapper :: map$ that takes as input a pair of the form $< bucket, word >$ and outputs the same pair, and the reduce method $Message\ Randomizer\ Redistribute\ Reducer :: reduce$ that takes as input a pair of the form

$$< bucket, list(word_0, word_1, ..., word_{d-2}, word_{d-1}) >$$

and outputs uniformly at random $< bucket', word_i >$, such that each bucket is given kn messages.

9. $round_k \rightarrow Message\ Randomizer\ [redistribute = false] \rightarrow round_{k+1}$ is then run for the last $k \in \{n', n' + 1, ..., 2n' - 2, 2n' - 1\}$ steps.

The general responsibility assignment software patterns [39] we used are:

- Information expert. The *Moderator* class knows the number of mixes and number of messages. All entry points extend the *Entry Point* abstract class. This class knows the *Moderator* and points to it for being asked about these values by the subclasses. The *Entry Point* abstract class also acts as an information expert for its subclasses.
- Creator and Controller. Entry points are created by the *Moderator*. The *Moderator* is the controller for the entry points, while each *Entry Point* is an indirect controller for the map and reduce classes located in its package. The control is indirect, as these classes are handled by the Hadoop via the configuration provided by the *Entry Point*.
- Low Coupling and High Cohesion. Each class depends on its superclass (i.e. *Message Randomizer Entry Point* depends on *Entry Point*) or on its controller (i.e. *Message Randomizer Mapper* asks *Message Randomizer Entry Point* for data about the number of mixes, which is provided through the coupling between the parent of *Message Randomizer Entry Point*, namely *Entry Point*, and the *Moderator*). Each class also has a number as little as possible of methods. The Hadoop framework encourages this by specifying interfaces for map and reduce classes to implement.

A sample code of our demonstration application can be found in appendix A.

Conclusion

This chapter describes a practical application that brings the Map Reduce programming model and the Parallel Mix Network anonymity model together, via the Hadoop open source distributed computing framework. Our aim for implementing mix networks in this framework, was scalability. In the future, refined versions of our implementation may also be deployed on thousands of machines.

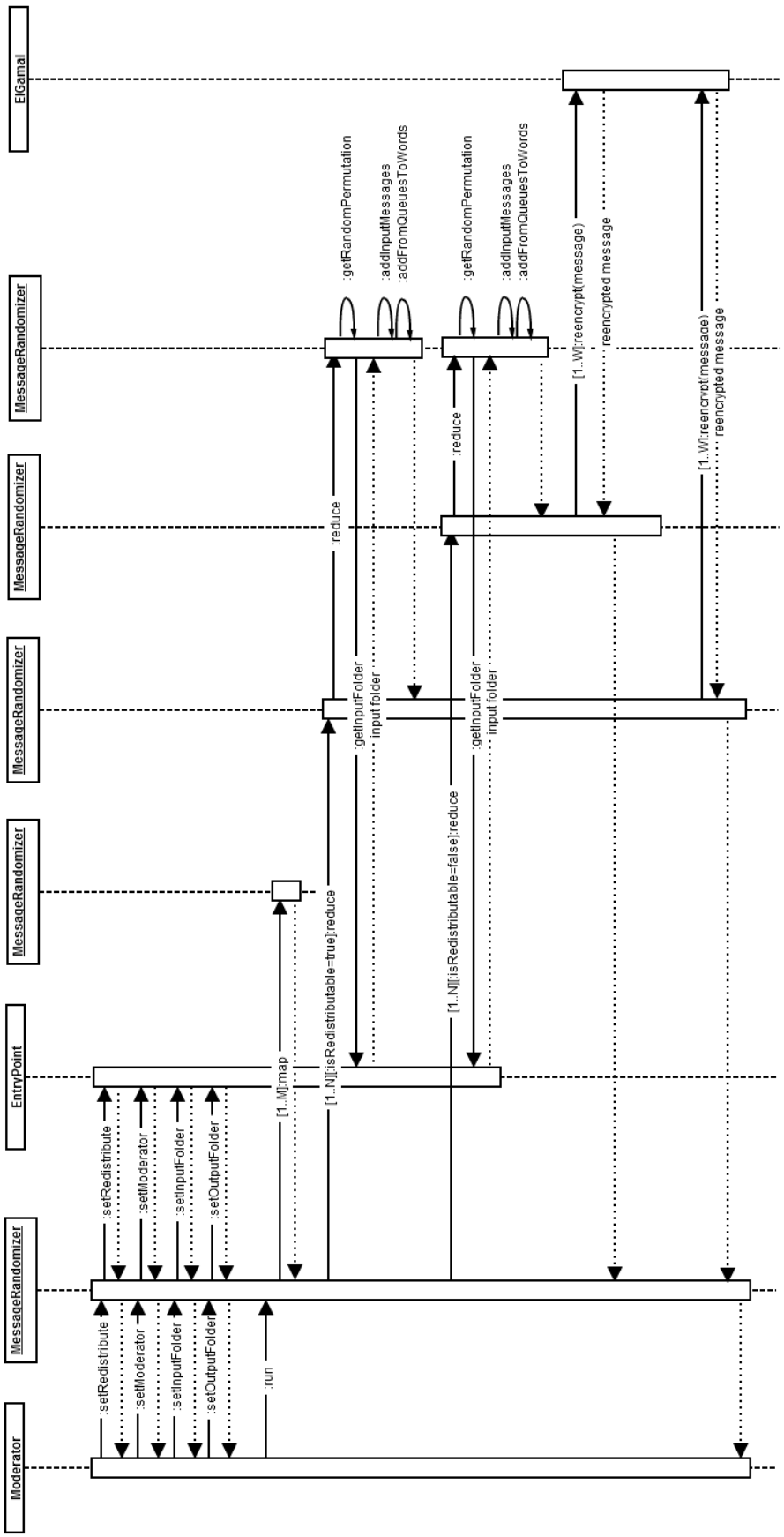


Figure 7.4: Sequence Diagram of the randomizing map reduce module

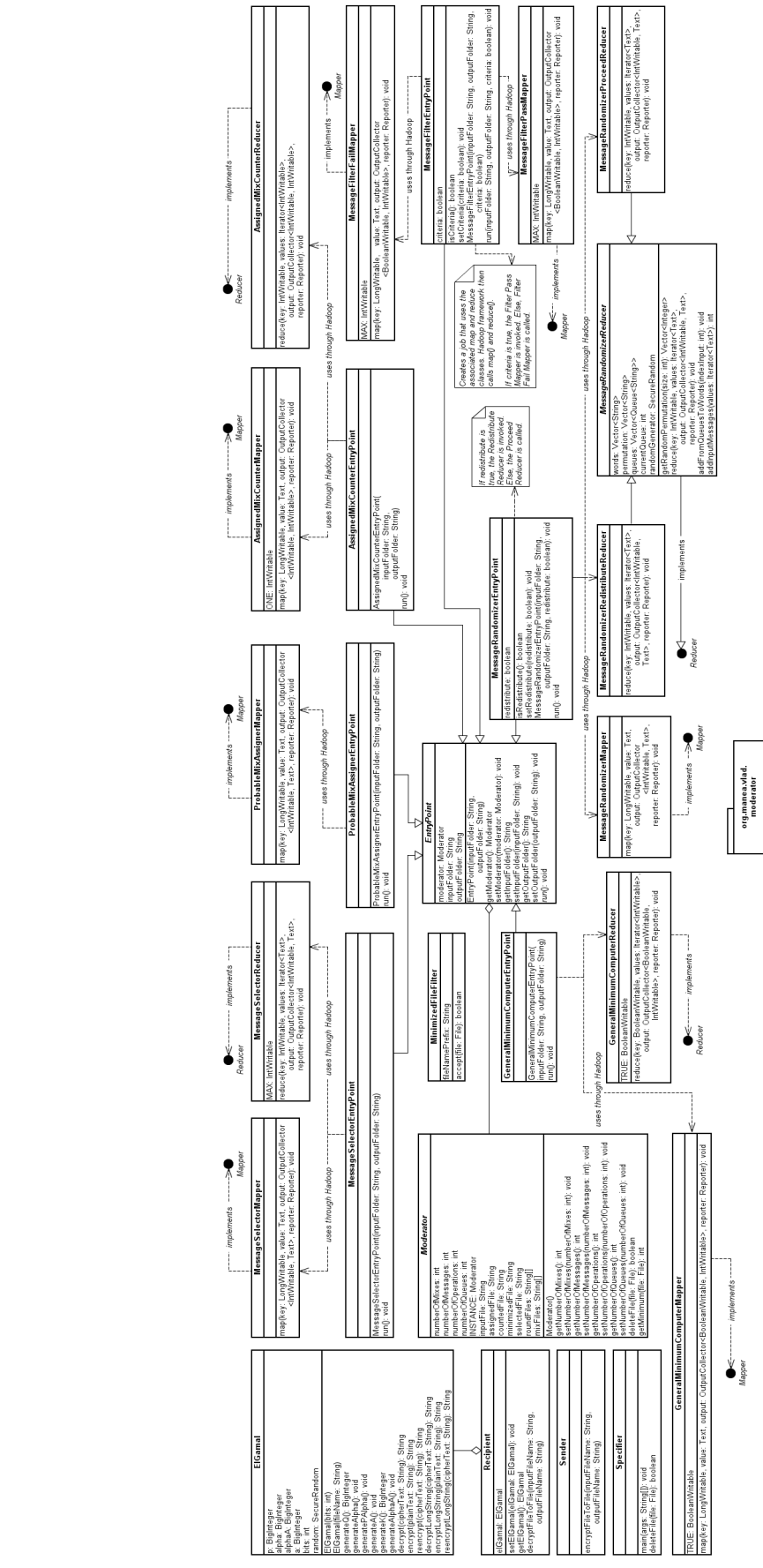


Figure 7.5: Classes and Packages Diagram of the application

Chapter 8

Conclusions

The digital society supposes that most activities will be performed on the Internet. We keep all our business, banking, news and personal data on unknown machines in the wild. Somewhere between, threats to our privacy are increasing, as data storage is cheap. Advertizers mine information to better target their offers and create new products. The advances of communication allow traffic exchange for large sums of money that come from information buying companies. The sensitivity level of our data is large requires not only its privacy, but also its anonymity and unobservability.

We learnt that there is always a trade-off between anonymity, cryptographic properties, correctness and speed, as described in the techniques we surveyed. Some models provide proxy speed, but their anonymity level is lower, while others keep messages for an undetermined amount of time, yielding to an arbitrarily large anonymity for them. The academics and the companies have advanced in anonymous communication by presenting ever progressing models, algorithms, techniques, frameworks and systems for the public to use.

We contributed to the topic by analyzing DC Networks and Mix Networks. We proposed a simple algorithm for the problem of arbitrarily long delays of messages in the mix, which is known to exist in current flush techniques. We also modelled and implemented an application of the Parallel Mix Network model [26] by using the Hadoop open source distributed computing framework [36] based on the Map Reduce programming model [35]. Our implementation provides a basis for future possibly open source development of real life distributed anonymity systems.

Many fields benefit from the anonymity of mix networks and other cryptographic constructions and architectures. However, for anonymous communication, the mix network is still the most suitable solution, when both security and scalability are crucial. Even though various techniques have been proposed, the general detection and prevention of collusions are still an open problem. An inherent problem is also the implementation of such anonymity services at a large scale. As part of one's life has moved on the Internet, both institutions and people are in need of scalable and fast anonymous communication systems that cross oppressive regimes and authority misconduct.

Appendix A

Application code

This appendix contains the implementation of a demonstation application that uses Hadoop to permute given text messages. The implementation includes the ElGamal cryptosystem and also our flush algorithm with queues. Note that the implemented state of the algorithm can be further optimized by adding the circular double linked lists, as described in 7.4. All code from below is writtern in the Java programming language. Each section represents a package and each subsection represents a class in that particular package.

A.1 org.manea.vlad.moderator

A.1.1 Specifier

```
package org.manea.vlad.moderator;

import java.io.File;

import javax.xml.parsers.*;
import org.w3c.dom.*;

/**
 * it specifies all parameters for the moderator and mixes
 *
 * @author administrator
 */
public class Specifier {

    // constant string names
    private static final String mixesTagName = "mixes";
    private static final String queuesTagName = "queues";
    private static final String operationsTagName = "operations";
    private static final String inputTagName = "input";
    private static final String selectedTagName = "selected";
    private static final String nextTagName = "next";
    private static final String assignedTagName = "assigned";
    private static final String countedTagName = "counted";
    private static final String minimizedTagName = "minimized";
    private static final String roundTagName = "round";
    private static final String mixTagName = "mix";
    private static final String initialTagName = "initial";
    private static final String finalTagName = "final";
    private static final String elgamalTagName = "elgamal";
    private static final String inputFileName = "/input";

    // delete file warning message
    public static final String deleteFileWarningMessage = "WARN: could not delete file: ";

    /**
     * deletes possibly non empty file
     *
     * @param file
     *            the file
     * @return true if and only if file was deleted
     */
    public static boolean deleteFile(File file) {

        // check if the file is a directory
        if (file.isDirectory()) {

            // get files located in the folder
            String[] children = file.list();

            // iterate all children
            for (int index = 0; index < children.length; index++) {

                // delete the file recursively and return false on failure
                if (!deleteFile(new File(file, children[index])))
                    return false;
```

```
        }
    }

    // return the deletion of the file which is now surely empty
    return file.delete();

}

/**
 * main method
 *
 * @param args
 *            arguments
 * @throws Exception
 */
public static void main(String[] args) throws Exception {

    // test if argument strings are correct
    if (args == null)
        throw new NullPointerException();

    // test if argument string contains the file name
    if (args.length < 1)
        throw new Exception();

    // create file
    File XMLfile = new File(args[0]);

    // create document builder factory
    DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory
            .newInstance();

    // create document builder
    DocumentBuilder documentBuilder = documentBuilderFactory
            .newDocumentBuilder();

    // create document
    Document document = documentBuilder.parse(XMLfile);

    // normalize document root element
    document.getDocumentElement().normalize();

    // set number of mixes
    Moderator.INSTANCE.setNumberOfMixes(Integer.parseInt(document
            .getDocumentElement().getElementsByTagName(mixesTagName)
            .item(0).getChildNodes().item(0).getNodeValue()));

    // set number of queues
    Moderator.INSTANCE.setNumberOfQueues(Integer.parseInt(document
            .getDocumentElement().getElementsByTagName(queuesTagName)
            .item(0).getChildNodes().item(0).getNodeValue()));

    // set number of operations
    Moderator.INSTANCE.setNumberOfOperations(Integer.parseInt(document
            .getDocumentElement().getElementsByTagName(operationsTagName)
            .item(0).getChildNodes().item(0).getNodeValue()));

    // set input file
    Moderator.INSTANCE.setInputFile(document.getDocumentElement()
            .getElementsByTagName(inputTagName).item(0).getChildNodes()
            .item(0).getNodeValue());

    // set assigned file
    Moderator.INSTANCE.setAssignedFile(document.getDocumentElement()
            .getElementsByTagName(assignedTagName).item(0).getChildNodes()
            .item(0).getNodeValue());

    // set counted file
    Moderator.INSTANCE.setCountedFile(document.getDocumentElement()
            .getElementsByTagName(countedTagName).item(0).getChildNodes()
            .item(0).getNodeValue());

    // set counted file
    Moderator.INSTANCE.setNextFile(document.getDocumentElement()
            .getElementsByTagName(nextTagName).item(0).getChildNodes()
            .item(0).getNodeValue());

    // set minimized file
    Moderator.INSTANCE.setMinimizedFile(document.getDocumentElement()
            .getElementsByTagName(minimizedTagName).item(0).getChildNodes()
            .item(0).getNodeValue());

    // set selected file
    Moderator.INSTANCE.setSelectedFile(document.getDocumentElement()
            .getElementsByTagName(selectedTagName).item(0).getChildNodes()
            .item(0).getNodeValue());

    // get nodes for round files
    NodeList roundFileNodes = document.getDocumentElement()
            .getElementsByTagName(roundTagName);

    // create round file names
    String[] roundFileNodeNames = new String[roundFileNodes.getLength()];
```

```
        // iterate all round file nodes
        for (int index = 0; index < roundFileNodes.getLength(); ++index)
            roundFileNodeNames[index] = roundFileNodes.item(index)
                    .getChildNodes().item(0).getNodeValue();

        // set round files if correct
        Moderator.INSTANCE.setRoundFiles(roundFileNodeNames,
                1 + 2 * Moderator.INSTANCE.getNumberOfOperations());

        // get nodes for mix files
        NodeList mixFileNodes = document.getDocumentElement()
                .getElementsByTagName(mixTagName);

        // create mix file names
        String[] mixFileNodeNames = new String[mixFileNodes.getLength()];

        // iterate all mix file nodes
        for (int index = 0; index < mixFileNodes.getLength(); ++index)
            mixFileNodeNames[index] = mixFileNodes.item(index).getChildNodes()
                    .item(0).getNodeValue();

        // set round files if correct
        Moderator.INSTANCE.setMixFiles(mixFileNodeNames,
                Moderator.INSTANCE.getNumberOfMixes());

        // set receiver algorithm
        Recipient.setElGamal(document.getDocumentElement()
                .getElementsByTagName(elgamalTagName).item(0).getChildNodes()
                .item(0).getNodeValue());

        // encrypt data
        Sender.encryptFileToFile(document.getDocumentElement()
                .getElementsByTagName(initialTagName).item(0).getChildNodes()
                .item(0).getNodeValue(), document.getDocumentElement()
                .getElementsByTagName(inputTagName).item(0).getChildNodes()
                .item(0).getNodeValue()
                + inputFileName);

        // run moderator
        Moderator.INSTANCE.run();

        // decipher data
        Recipient.decryptFileToFile(
                roundFileNodeNames[roundFileNodeNames.length - 1], document
                        .getDocumentElement()
                        .getElementsByTagName(finalTagName).item(0)
                        .getChildNodes().item(0).getNodeValue());

    }

}
```

A.1.2 Moderator

```
package org.manea.vlad.moderator;

import java.io.*;
import java.util.StringTokenizer;

import org.manea.vlad.assignedmixcounter.AssignedMixCounterEntryPoint;
import org.manea.vlad.generalminimumcomputer.GeneralMinimumComputerEntryPoint;
import org.manea.vlad.messagefilter.MessageFilterEntryPoint;
import org.manea.vlad.messagerandomizer.MessageRandomizerEntryPoint;
import org.manea.vlad.messageselector.MessageSelectorEntryPoint;
import org.manea.vlad.probablemixassigner.ProbableMixAssignerEntryPoint;

/**
 * is the moderator class
 *
 * @author administrator
 */
public class Moderator {

    // create unique instance
    public final static Moderator INSTANCE = new Moderator();

    private final String warningZeroMessages = "WARN: Minimum could not be found. Maybe you have no messages!";

    // number of mixes
    private int numberOfMixes = 1;

    // number of operations
    private int numberOfOperations = 1;

    // number of queues
    private int numberOfQueues = 1;

    // number of memories
    private int numberOfMemory = 1;

    // number of messages
    private int numberOfMessages = Integer.MAX_VALUE;

    // input file location
    private String inputFile;

    // assigned file location
```

```
private String assignedFile;

// counted file location
private String countedFile;

// minimized file location
private String minimizedFile;

// selected file location
private String selectedFile;

// next file location
private String nextFile;

// round files locations
private String[] roundFiles;

// mix files locations
private String[] mixFiles;

/**
 * moderator constructor
 */
private Moderator() {

    // implemented for singleton purpose

}

/**
 * gets the input file
 *
 * @return the inputFile
 */
public synchronized String getInputFile() {

    // get input file
    return inputFile;

}

/**
 * sets the input file
 *
 * @param inputFile
 *            the input file
 *
 * @throws Exception
 */
public synchronized void setInputFile(String inputFile) throws Exception {

    // test for incorrect input file
    if (inputFile == null)
        throw new NullPointerException();

    // set input file
    this.inputFile = inputFile;

}

/**
 * gets the assigned file
 *
 * @return the assignedFile
 */
public synchronized String getAssignedFile() {

    // get assigned file
    return assignedFile;

}

/**
 * sets the assigned file
 *
 * @param assignedFile
 *            the assigned file
 *
 * @throws Exception
 */
public synchronized void setAssignedFile(String assignedFile)
        throws Exception {

    // test for incorrect assigned file
    if (assignedFile == null)
        throw new NullPointerException();

    // set assigned file
    this.assignedFile = assignedFile;

}

/**
 * gets the next file
 *
 * @return the nextFile
 */
public synchronized String getNextFile() {

    // get next file
    return nextFile;
```

```
}

/**
 * sets the next file
 *
 * @param nextFile
 *            the next file
 *
 * @throws Exception
 */
public synchronized void setNextFile(String nextFile) throws Exception {

    // test for incorrect next file
    if (nextFile == null)
        throw new NullPointerException();

    // set next file
    this.nextFile = nextFile;

}

/**
 * gets the minimized file
 *
 * @return the minimizedFile
 */
public synchronized String getMinimizedFile() {

    // get minimized file
    return minimizedFile;

}

/**
 * sets the minimized file
 *
 * @param minimizedFile
 *            the minimized file
 *
 * @throws Exception
 */
public synchronized void setMinimizedFile(String minimizedFile)
        throws Exception {

    // test for incorrect minimized file
    if (minimizedFile == null)
        throw new NullPointerException();

    // set selected file
    this.minimizedFile = minimizedFile;

}

/**
 * gets the selected file
 *
 * @return the selectedFile
 */
public synchronized String getSelectedFile() {

    // get selected file
    return selectedFile;

}

/**
 * sets the selected file
 *
 * @param selectedFile
 *            the selected file
 *
 * @throws Exception
 */
public synchronized void setSelectedFile(String selectedFile)
        throws Exception {

    // test for incorrect selected file
    if (selectedFile == null)
        throw new NullPointerException();

    // set selected file
    this.selectedFile = selectedFile;

}

/**
 * gets the counted file
 *
 * @return the countedFile
 */
public synchronized String getCountedFile() {

    // get counted file
    return countedFile;

}

/**
 * sets the counted file
 *
 * @param countedFile
```

```
     *            the counted file
     *
     * @throws Exception
     */
    public synchronized void setCountedFile(String countedFile)
            throws Exception {

        // test for incorrect counted file
        if (countedFile == null)
            throw new NullPointerException();

        // set counted file
        this.countedFile = countedFile;

    }

    /**
     * gets the round file
     *
     * @param round
     *            number of round
     *
     * @return the round file
     */
    public synchronized String getRoundFile(int round) {

        // test for round in bounds
        if (round < 0 || round >= roundFiles.length)
            throw new ArrayIndexOutOfBoundsException();

        // get round file
        return roundFiles[round];

    }

    /**
     * sets the round files
     *
     * @param roundFiles
     *            the round files
     * @param size
     *            the required size
     *
     * @throws Exception
     */
    public synchronized void setRoundFiles(String[] roundFiles, int size)
            throws Exception {

        // test for incorrect round files
        if (roundFiles == null)
            throw new NullPointerException();

        // test for incorrect number of round files
        if (roundFiles.length != size)
            throw new Exception();

        // set this round files
        this.roundFiles = new String[size];

        // set array by copying
        System.arraycopy(roundFiles, 0, this.roundFiles, 0, size);

    }

    /**
     * gets the mix file
     *
     * @param mix
     *            index of mix
     *
     * @return the mix file
     */
    public synchronized String getMixFile(int mix) {

        // test for mix in bounds
        if (mix < 0 || mix >= roundFiles.length)
            throw new ArrayIndexOutOfBoundsException();

        // get mix file
        return mixFiles[mix];

    }

    /**
     * sets the mix files
     *
     * @param mixFiles
     *            the mix files
     * @param size
     *            the required size
     *
     * @throws Exception
     */
    public synchronized void setMixFiles(String[] mixFiles, int size)
            throws Exception {

        // test for incorrect mix files
        if (mixFiles == null)
            throw new NullPointerException();

        // test for incorrect number of mix files
```

```
    if (mixFiles.length != size)
        throw new Exception();

    // set this mix files
    this.mixFiles = new String[size];

    // set array by copying
    System.arraycopy(mixFiles, 0, this.mixFiles, 0, size);

}

/**
 * gets the number of queues
 *
 * @return the numberOfQueues
 */
public synchronized int getNumberOfQueues() {

    // get number of queues
    return numberOfQueues;

}

/**
 * sets the number of queues as a non-negative value
 *
 * @param numberOfQueues
 *            the numberOfQueues to set
 * @throws Exception
 */
public synchronized void setNumberOfQueues(int numberOfQueues)
        throws Exception {

    // test for incorrect number of queues
    if (numberOfQueues <= 0)
        throw new Exception();

    // set number of queues
    this.numberOfQueues = numberOfQueues;

}

/**
 * gets the number of mixes
 *
 * @return the numberOfMixes
 */
public synchronized int getNumberOfMixes() {

    // get number of mixes
    return numberOfMixes;

}

/**
 * sets the number of mixes as a non-negative value
 *
 * @param numberOfMixes
 *            the numberOfMixes to set
 * @throws Exception
 */
public synchronized void setNumberOfMixes(int numberOfMixes)
        throws Exception {

    // test for incorrect number of mixes
    if (numberOfMixes <= 0)
        throw new Exception();

    // set number of mixes
    this.numberOfMixes = numberOfMixes;

}

/**
 * gets the number of memory
 *
 * @return the numberOfMemory
 */
public synchronized int getNumberOfMemory() {

    // get number of memory
    return numberOfMemory;

}

/**
 * sets the number of memory as a non-negative value
 *
 * @param numberOfMemory
 *            the numberOfMemory to set
 * @throws Exception
 */
public synchronized void setNumberOfMemory(int numberOfMemory)
        throws Exception {

    // test for incorrect number of memory
    if (numberOfMemory <= 0)
        throw new Exception();

    // set number of memory
```

```
    this.numberOfMemory = numberOfMemory;
}

/**
 * gets the number of operations
 *
 * @return the numberOfOperations
 */
public synchronized int getNumberOfOperations() {

    // get number of operations
    return numberOfOperations;

}

/**
 * sets the number of operations as a non-negative value
 *
 * @param numberOfOperations
 *            the numberOfOperations to set
 * @throws Exception
 */
public synchronized void setNumberOfOperations(int numberOfOperations)
        throws Exception {

    // test for incorrect number of operations
    if (numberOfOperations <= 0)
        throw new Exception();

    // set number of operations
    this.numberOfOperations = numberOfOperations;

}

/**
 * gets the number of messages
 *
 * @return the numberOfMessages
 */
public synchronized int getNumberOfMessages() {

    // get number of messages
    return numberOfMessages;

}

/**
 * sets the number of messages as a non-negative value<br />
 * the number of messages is the number of messages each mix must receive<br />
 * the number of messages must be a multiple of the number of mixes
 *
 * @param numberOfMessages
 *            the numberOfMessages to set
 * @throws Exception
 */
public synchronized void setNumberOfMessages(int numberOfMessages)
        throws Exception {

    // test for incorrect number of messages
    if (numberOfMessages < 0)
        throw new Exception();

    // test for not multiple of n
    if (numberOfMessages % getNumberOfMixes() != 0)
        throw new Exception();

    // set number of messages
    this.numberOfMessages = numberOfMessages;

}

/**
 * gets minimum
 *
 * @param file
 *            the file to be read
 * @return the minimum number
 * @throws IOException
 */
private int getMinimum(File file) throws IOException {

    // test if file is not directory
    if (!file.isDirectory())
        throw new IOException();

    // obtain children files
    File[] children = file.listFiles(new MinimizedFileFilter());

    // check if children has files
    if (children.length < 1)
        throw new IOException();

    // set new file
    file = children[0];

    // create a buffer reader
    BufferedReader bufferReader = new BufferedReader(new FileReader(
            file.getAbsolutePath()));
```

```
        // create string
        String line = bufferReader.readLine();

        // try to read line
        if (line == null) {

            // write warning
            System.out.println(warningZeroMessages);

            // there is no line
            return 0;

        }

        // create string token creator
        StringTokenizer stringTokenizer = new StringTokenizer(line);

        // check if the line has more tokens
        if (!stringTokenizer.hasMoreTokens())
            throw new IOException();

        // lose the first token
        stringTokenizer.nextToken();

        // check if the line has more tokens
        if (!stringTokenizer.hasMoreTokens())
            throw new IOException();

        // return the integer value of the next token
        return Integer.parseInt(stringTokenizer.nextToken());

    }

    /**
     * run method
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create probable mix assigner
        ProbableMixAssignerEntryPoint probableMixAssignerEntryPoint = new ProbableMixAssignerEntryPoint(
                getInputFile(), getAssignedFile());

        // try to delete the assigned folder
        if (!Specifier.deleteFile(new File(getAssignedFile())))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getAssignedFile());

        // run probable mix assigner
        probableMixAssignerEntryPoint.run();

        // create assigned mix counter
        AssignedMixCounterEntryPoint assignedMixCounterEntryPoint = new AssignedMixCounterEntryPoint(
                getAssignedFile(), getCountedFile());

        // try to delete the counted folder
        if (!Specifier.deleteFile(new File(getCountedFile())))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getCountedFile());

        // run assigned mix counter
        assignedMixCounterEntryPoint.run();

        // create assigned mix counter
        GeneralMinimumComputerEntryPoint generalMinimumComputerEntryPoint = new GeneralMinimumComputerEntryPoint(
                getCountedFile(), getMinimizedFile());

        // try to delete the minimized folder
        if (!Specifier.deleteFile(new File(getMinimizedFile())))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getMinimizedFile());

        // run assigned mix minimum computer
        generalMinimumComputerEntryPoint.run();

        // read number of messages
        int numberOfMessages = getMinimum(new File(getMinimizedFile()));

        // test for positive minimum
        if (numberOfMessages <= 0)
            return;

        // set number of messages
        Moderator.INSTANCE.setNumberOfMessages(numberOfMessages);

        /*
         * write the number of messages
         * System.out.println("The number of accepted messages per server is " +
         * getNumberOfMessages());
         */

        // create message selector
        MessageSelectorEntryPoint messageSelectorEntryPoint = new MessageSelectorEntryPoint(
                getAssignedFile(), getSelectedFile());
```

```
        // try to delete the selected folder
        if (!Specifier.deleteFile(new File(getSelectedFile())))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getSelectedFile());

        // run assigned mix selector
        messageSelectorEntryPoint.run();

        // create message filter[true]
        MessageFilterEntryPoint messageFilterEntryPointTrue = new MessageFilterEntryPoint(
                getSelectedFile(), getRoundFile(0), true);

        // try to delete the round folder
        if (!Specifier.deleteFile(new File(roundFiles[0])))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getRoundFile(0));

        // run assigned mix filter
        messageFilterEntryPointTrue.run();

        // create message filter[false]
        MessageFilterEntryPoint messageFilterEntryPointFalse = new MessageFilterEntryPoint(
                getSelectedFile(), getNextFile(), false);

        // try to delete the next folder
        if (!Specifier.deleteFile(new File(getNextFile())))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getNextFile());

        // run assigned mix filter
        messageFilterEntryPointFalse.run();

        // set number of operations
        Moderator.INSTANCE.setNumberOfOperations(numberOfOperations);

        // set the index
        int index;

        // make less than the operations non-redistribute randomize operations
        for (index = 1; index <= Moderator.INSTANCE.getNumberOfOperations() - 1; ++index) {

            // create message randomizer[false]
            MessageRandomizerEntryPoint messageRandomizerEntryPointI = new MessageRandomizerEntryPoint(
                    getRoundFile(index - 1), getRoundFile(index), false);

            // try to delete the output folder
            if (!Specifier.deleteFile(new File(getRoundFile(index))))
                System.out.println(Specifier.deleteFileWarningMessage
                        + getRoundFile(index));

            // run assigned mix randomizer
            messageRandomizerEntryPointI.run();

        }

        // create message randomizer[false]
        MessageRandomizerEntryPoint messageRandomizerEntryPoint = new MessageRandomizerEntryPoint(
                getRoundFile(index - 1), getRoundFile(index), true);

        // try to delete the output folder
        if (!Specifier.deleteFile(new File(getRoundFile(index))))
            System.out.println(Specifier.deleteFileWarningMessage
                    + getRoundFile(index));

        // run assigned mix randomizer
        messageRandomizerEntryPoint.run();

        // make the operations non-redistribute randomize operations
        for (++index; index <= 2 * Moderator.INSTANCE.getNumberOfOperations(); ++index) {

            // create message randomizer[false]
            MessageRandomizerEntryPoint messageRandomizerEntryPointI = new MessageRandomizerEntryPoint(
                    getRoundFile(index - 1), getRoundFile(index), false);

            // try to delete the output folder
            if (!Specifier.deleteFile(new File(getRoundFile(index))))
                System.out.println(Specifier.deleteFileWarningMessage
                        + getRoundFile(index));

            // run assigned mix randomizer
            messageRandomizerEntryPointI.run();

        }

    }
}
```

A.1.3 MinimizedFileFilter

```
package org.manea.vlad.moderator;

import java.io.File;
```

```
import java.io.FileFilter;

/**
 * is a file filter for the minimized file
 *
 * @author administrator
 *
 */
public class MinimizedFileFilter implements FileFilter {

    // prefix name
    public final String fileNamePrefix = "part-";

    /**
     * accepts the file
     *
     * @param file
     *            the file
     */
    public boolean accept(File file) {

        // return true if and only if the file has the prefix
        return file.getName().startsWith(fileNamePrefix);

    }

}
```

A.1.4 ElGamal

```
package org.manea.vlad.moderator;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.Random;
import java.util.StringTokenizer;

/**
 * Class ElGamal
 *
 * @author Vlad Manea
 */
public class ElGamal {

    // ElGamal parameters
    protected BigInteger p, alpha, alphaA;
    private BigInteger a;
    private int bits;
    protected Random random = new SecureRandom();

    /**
     * generateQ generates prime q
     *
     * @return q
     */
    private BigInteger generateQ() {
        // generate q as a prime
        BigInteger q = BigInteger.probablePrime(bits - 1, random);
        // while 2*q+1 is not a prime in itself
        while (!q.multiply(BigInteger.valueOf(2)).add(BigInteger.valueOf(1))
                .isProbablePrime(10)) {
            // generate q again
            q = BigInteger.probablePrime(bits - 1, random);
            System.out.println("q");
        }
        // return a prime q
        return q;
    }

    /**
     * generates alpha
     */
    private void generateAlpha() {
        // generate new random element
        alpha = new BigInteger(p.bitLength() - 1, random);
        // while alpha is not in Zp*
        while (p.gcd(alpha).compareTo(BigInteger.valueOf(1)) > 0
                || alpha.modPow(BigInteger.valueOf(2), p).compareTo(
                        BigInteger.valueOf(1)) == 0
                || alpha.modPow(
                        p.subtract(
                                BigInteger.valueOf(1).divide(
                                        BigInteger.valueOf(2))).mod(
                                p.subtract(BigInteger.valueOf(1))), p)
                        .compareTo(BigInteger.valueOf(1)) == 0) {
            // generate new random element
            alpha = new BigInteger(p.bitLength(), random);
            System.out.println("alpha");
        }
    }

    /**
```

```
 * generatePAlpha generates parameters p and alpha
 */
private void generatePAlpha() {
    // generate a random prime q = ans[1] of 1023 bits
    BigInteger q = generateQ();
    // now you know p = 2*q+1 = 2*ans[1]+1 where p-1 = 2^1*q^1
    p = q.multiply(BigInteger.valueOf(2)).add(BigInteger.valueOf(1));
    // choose a random element in Zp*
    generateAlpha();
}

/**
 * generateA generates parameter a
 */
private void generateA() {
    // generate a random element
    a = generateK();
}

/**
 * generateK generates parameter k
 *
 * @return k
 */
private BigInteger generateK() {
    // generate a random element
    BigInteger k = new BigInteger(p.bitLength(), random);
    // while it is not in [1, p-2]
    while (BigInteger.ONE.compareTo(k) > 0
            || p.subtract(BigInteger.valueOf(2)).compareTo(k) < 0) {
        // generate new random element
        k = new BigInteger(p.bitLength(), random);
    }
    // return it
    return k;
}

/**
 * generateAlphaA generates alpha^a mod p
 */
private void generateAlphaA() {
    alphaA = alpha.modPow(a.mod(p.subtract(BigInteger.valueOf(1))), p);
}

/**
 * deciphers encrypted text
 *
 * @param cipherText
 *            the cipher text
 * @return deciphered text
 */
String decrypt(String cipherText) throws Exception {
    StringTokenizer stringTokenizer = new StringTokenizer(cipherText, "!");
    // test for existence
    if (!stringTokenizer.hasMoreTokens())
        throw new Exception();
    // set gamma
    BigInteger gamma = new BigInteger(stringTokenizer.nextToken());
    // test for existence
    if (!stringTokenizer.hasMoreTokens())
        throw new Exception();
    // set delta
    BigInteger delta = new BigInteger(stringTokenizer.nextToken());
    // return deciphered text
    return delta
            .multiply(
                    gamma.modPow(p.subtract(BigInteger.ONE).subtract(a), p))
            .mod(p).toString();
}

/**
 * encrypt encrypts plain text
 *
 * @param plainText
 *            the plain text
 * @return
 * @throws Exception
 */
public String encrypt(String plainText) throws Exception {
    BigInteger m = new BigInteger(plainText);
    // test plain text validity
    if (m.compareTo(BigInteger.ZERO) < 0 || m.compareTo(p) >= 0) {
        // throw invalid exception
        throw new Exception("message is not in the interval [0, p-1]");
    }
    // generate k as a random in [1, p-2]
    BigInteger k = generateK();
    // set gamma
    BigInteger gamma = alpha.modPow(k, p);
    // set delta
    BigInteger delta = m.multiply(alphaA.modPow(k, p)).mod(p);
    // return encrypted text
```

```
    return gamma.toString() + "!" + delta.toString();
}

/**
 * re-encrypt re-encrypts cipher text
 *
 * @param cipherText
 *            the cipher text
 * @return
 * @throws Exception
 */
public String reencrypt(String cipherText) throws Exception {
    StringTokenizer stringTokenizer = new StringTokenizer(cipherText, "!");
    // test for existence
    if (!stringTokenizer.hasMoreTokens()) {
        System.out.println("first problem for string " + cipherText);
        throw new Exception();
    }
    // set gamma
    BigInteger gamma = new BigInteger(stringTokenizer.nextToken());
    // test for existence
    if (!stringTokenizer.hasMoreTokens()) {
        System.out.println("second problem for string " + cipherText);
        throw new Exception();
    }
    // set delta
    BigInteger delta = new BigInteger(stringTokenizer.nextToken());
    // generate k as a random in [1, p-2]
    BigInteger k = generateK();
    // set gamma as gamma alpha^k
    gamma = gamma.multiply(alpha.modPow(k, p)).mod(p);
    // set delta as delta alphaA^k
    delta = delta.multiply(alphaA.modPow(k, p)).mod(p);
    // return encrypted text
    return gamma.toString() + "!" + delta.toString();
}

/**
 * constructor
 *
 * @param bits
 *            number of bits
 * @throws Exception
 */
ElGamal(int bits) throws Exception {
    // are bits enough?
    if (bits <= 0) {
        // throw invalid exception
        throw new Exception("Number of bits is too small");
    }
    // set number of bits
    this.bits = bits;
    // generate p and alpha
    generatePAlpha();
    // generate a
    generateA();
    // compute alpha^a
    generateAlphaA();
    // System.out.println(p.toString());
    // System.out.println(alpha.toString());
    // System.out.println(a.toString());
}

ElGamal(String fileName) throws IOException {

    // create buffered reader
    BufferedReader bufferedReader = new BufferedReader(new FileReader(
            fileName));

    // create line
    String line;

    // test line
    if ((line = bufferedReader.readLine()) == null)
        throw new IOException();

    // get p
    p = new BigInteger(line);

    // test line
    if ((line = bufferedReader.readLine()) == null)
        throw new IOException();

    // get alpha
    alpha = new BigInteger(line);

    // test line
    if ((line = bufferedReader.readLine()) == null)
        throw new IOException();

    // get a
    a = new BigInteger(line);

    // generate alphaA
```

```
    generateAlphaA();

    // close reader
    bufferedReader.close();

}

/**
 * encrypts a long plain text
 *
 * @param plainText
 *            plain text
 * @return
 */
public String encryptLongString(String plainText) {

    // set final content
    String content = "";

    // set initial simplePlain
    BigInteger simplePlain = BigInteger.ZERO;

    // iterate all bytes
    for (int index = 0; index < plainText.length(); ++index) {

        // reset simplePlain
        if (index % 56 == 0)
            simplePlain = BigInteger.ZERO;

        // add value to simple plain
        simplePlain = simplePlain.or(new BigInteger(new Integer(
                (int) plainText.charAt(index)).toString())
                .shiftLeft(index % 56 * 16));

        // is it time to pack?
        if (index % 56 == 55 || index == plainText.length() - 1) {

            // add size to simple plain
            simplePlain = simplePlain.or(new BigInteger(new Integer(
                    index % 56).toString()).shiftLeft(896));

            // add random bits to simple plain
            simplePlain = simplePlain.or((new BigInteger(120, random))
                    .shiftLeft(902));

            try {

                // set content with this new number
                content += "#"
                        + new String(encrypt(simplePlain.toString()));

            } catch (Exception exception) {

            }

        }

    }

    // return content
    return content.substring(1);

}

/**
 * decrypts long ciphertext
 *
 * @param cipherText
 *            cipher text
 * @return
 * @throws Exception
 */
public String decryptLongString(String cipherText) throws Exception {

    // reset content
    String content = "";

    // set complicated text
    BigInteger complicatedText;

    // set number of characters
    int numberOfCharacters;

    // add # token extractor
    StringTokenizer hashTokenizer = new StringTokenizer(cipherText, "#");

    // iterate all tokens
    while (hashTokenizer.hasMoreTokens()) {

        // decipher token
        complicatedText = new BigInteger(decrypt(hashTokenizer.nextToken()));

        // get number of characters
        numberOfCharacters = Integer.parseInt(complicatedText
                .shiftRight(896)
                .and(new BigInteger(new Integer("63").toString()))
                .toString());

        // iterate for number of characters
        for (int jndex = 0; jndex <= numberOfCharacters; ++jndex) {
```

```
                // set character
                char character = (char) Integer.parseInt(complicatedText
                        .shiftRight(jndex * 16)
                        .and(new BigInteger(new Integer("65535").toString()))
                        .toString());

                // add byte
                content = content + character;

            }

        }

        // return content
        return content;

    }

    /**
     * re-encrypts long ciphertext
     *
     * @param cipherText
     *            cipher text
     * @return
     * @throws Exception
     */
    public String reencryptLongString(String cipherText) throws Exception {

        // reset content
        String content = "";

        // add # token extractor
        StringTokenizer hashTokenizer = new StringTokenizer(cipherText, "#");

        // iterate all tokens
        while (hashTokenizer.hasMoreTokens()) {

            // set content
            content = content + "#" + reencrypt(hashTokenizer.nextToken());

        }

        // return content
        return content.substring(1);

    }

}
```

A.1.5 Sender

```
package org.manea.vlad.moderator;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Vector;

public abstract class Sender {

    /**
     * ciphers a file
     *
     * @param inputFileName
     *            the input file name
     * @param outputFileName
     *            the output file name
     * @throws FileNotFoundException
     * @throws IOException
     * @throws Exception
     */
    public static void encryptFileToFile(String inputFileName,
            String outputFileName) {

        // create file input and output handlers
        BufferedReader bufferedReader = null;
        PrintWriter printWriter = null;

        // create vector of words
        Vector<String> words = new Vector<String>();

        try {

            // create buffered reader
            bufferedReader = new BufferedReader(new FileReader(inputFileName));

            // create line
            String line;

            // read all lines
            while ((line = bufferedReader.readLine()) != null) {

                // add content to words
                words.add("0 " + Recipient.getElGamal().encryptLongString(line));

            }
```

```
        } catch (Exception exception) {

            // write warning
            System.out.println("WARN: could not open initial file.");

        } finally {

            try {

                // close reader
                if (bufferedReader != null)
                    bufferedReader.close();

            } catch (IOException exception) {

                // do nothing with exception

            }

        }

        try {

            // create print writer
            printWriter = new PrintWriter(new FileWriter(outputFileName, false));

            // iterate words
            for (int index = 0; index < words.size(); ++index) {

                // put print writer
                printWriter.println(words.get(index));

            }

            // print writer flush
            printWriter.flush();

        } catch (Exception exception) {

            // write warning
            System.out.println("WARN: could not open input file.");

        } finally {

            // close reader
            if (printWriter != null)
                printWriter.close();

        }

    }

}
```

A.1.6 Recipient

```
package org.manea.vlad.moderator;

import java.io.*;
import java.util.StringTokenizer;

/**
 * receives messages
 *
 * @author administrator
 */
public abstract class Recipient {

    // create elGamal instance
    private static ElGamal elGamal;

    /**
     * registers elGamal
     *
     * @param fileName
     *            file name
     * @throws IOException
     * @throws Exception
     */
    public static void setElGamal(String fileName) throws IOException,
            Exception {

        // test for null string
        if (fileName == null)
            throw new Exception();

        // set algorithm
        elGamal = new ElGamal(fileName);

    }

    /**
     * gets ElGamal
     *
     * @return elGamal system
     */
    public static ElGamal getElGamal() {

        // get elGamal system
```

```
        return elGamal;
    }

    /**
     * deciphers a file
     *
     * @param inputFileName
     *            the input file name
     * @param outputFileName
     *            the output file name
     * @throws FileNotFoundException
     * @throws IOException
     * @throws Exception
     */
    public static void decryptFileToFile(String inputFileName,
            String outputFileName) throws FileNotFoundException, IOException,
            Exception {

        // create input file
        File inputFile = new File(inputFileName);

        // check if input file is not directory
        if (!inputFile.isDirectory())
            throw new IOException();

        // create print writer
        PrintWriter printWriter = new PrintWriter(new FileWriter(
                outputFileName, false));

        // get all children of the file
        File[] children = new File(inputFileName)
                .listFiles(new MinimizedFileFilter());

        // iterate all children
        for (int index = 0; index < children.length; ++index) {

            // create buffered reader
            BufferedReader bufferedReader = new BufferedReader(new FileReader(
                    children[index].getAbsolutePath()));

            // create line string and content
            String line;

            // read all lines
            while ((line = bufferedReader.readLine()) != null) {

                // add space token extractor
                StringTokenizer spaceTokenizer = new StringTokenizer(line);

                // check for initial number existence
                if (!spaceTokenizer.hasMoreTokens())
                    throw new Exception();

                // lose the first number
                spaceTokenizer.nextToken();

                // check for second number existence
                if (!spaceTokenizer.hasMoreTokens())
                    throw new Exception();

                // add content to answer
                printWriter.println(elGamal.decryptLongString(spaceTokenizer
                        .nextToken()));

                // print writer flush
                printWriter.flush();

            }

            // close buffered reader
            bufferedReader.close();

        }

        // close print writer
        printWriter.close();

    }
}
```

A.2 org.manea.vlad.entrypoint

A.2.1 EntryPoint

```
package org.manea.vlad.entrypoint;

import org.manea.vlad.moderator.Moderator;

/**
 * is entry point superclass for entry points which know their moderators
 *
 * @author administrator
 */
public abstract class EntryPoint {
```

```
// moderator
private static Moderator moderator = Moderator.INSTANCE;

// inputFolder
private String inputFolder;

// outputFolder
private String outputFolder;

/**
 * gets the moderator
 *
 * @return the moderator
 */
public static synchronized final Moderator getModerator() {

    // return the moderator
    return moderator;

}

/**
 * gets the input folder
 *
 * @return the inputFolder
 */
protected synchronized final String getInputFolder() {

    // return the input folder
    return inputFolder;

}

/**
 * sets the input folder
 *
 * @param inputFolder
 *            the inputFolder to set
 */
private synchronized final void setInputFolder(String inputFolder) {

    // check to see if input folder exists
    if (inputFolder == null)
        throw new NullPointerException();

    // set the input folder
    this.inputFolder = inputFolder;

}

/**
 * gets the output folder
 *
 * @return the outputFolder
 */
protected synchronized final String getOutputFolder() {

    // return the output folder
    return outputFolder;

}

/**
 * sets the output folder
 *
 * @param outputFolder
 *            the outputFolder to set
 */
private synchronized final void setOutputFolder(String outputFolder) {

    // check to see if output folder exists
    if (outputFolder == null)
        throw new NullPointerException();

    // set the output folder
    this.outputFolder = outputFolder;

}

/**
 * configures and runs a job
 *
 * @throws Exception
 */
public abstract void run() throws Exception;

/**
 * constructs the entry point
 *
 * @param inputFolder
 *            the input folder
 * @param outputFolder
 *            the output folder
 */
public EntryPoint(String inputFolder, String outputFolder) {

    // set all fields
    setInputFolder(inputFolder);
    setOutputFolder(outputFolder);
```

```
    }
}
```

A.3 org.manea.vlad.probablemixassigner

A.3.1 ProbableMixAssignerEntryPoint

```
package org.manea.vlad.probablemixassigner;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.fs.Path;

import org.manea.vlad.entrypoint.EntryPoint;

/**
 * is the configure class for probable mix assignment to messages
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
public class ProbableMixAssignerEntryPoint extends EntryPoint {

    /**
     * constructs the entry point for message filter
     *
     * @param inputFolder
     *            the input folder
     * @param outputFolder
     *            the output folder
     * @throws Exception
     */
    public ProbableMixAssignerEntryPoint(String inputFolder, String outputFolder) {

        // call super
        super(inputFolder, outputFolder);

    }

    /**
     * provides an entry point for probable mix assignment to messages
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create a job configuration
        JobConf job = new JobConf(ProbableMixAssignerEntryPoint.class);

        // set job name
        job.setJobName(ProbableMixAssignerEntryPoint.class.getName());

        // set output key and value classes
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(LongWritable.class);

        // set map class
        job.setMapperClass(ProbableMixAssignerMapper.class);

        // set input and output paths for files
        FileInputFormat.setInputPaths(job, new Path(getInputFolder()));
        FileOutputFormat.setOutputPath(job, new Path(getOutputFolder()));

        // set no reducers and argument number of mixes
        job.setNumReduceTasks(0);

        // run job
        JobClient.runJob(job);

    }
}
```

A.3.2 ProbableMixAssignerMapper

```
package org.manea.vlad.probablemixassigner;

import java.io.IOException;
import java.util.Random;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the probable mix assigner map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class ProbableMixAssignerMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, IntWritable, Text> {
```

```
    /**
     * maps (word) to (mix, word) where mix is a random in {1, 2, ..., n-1, n}
     *
     * @param key
     *            number of file
     * @param value
     *            contents of file
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void map(LongWritable key, Text value,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // get line
        String line = value.toString();

        // create a random generator
        Random randomGenerator = new Random();

        // create string token generator
        StringTokenizer tokenizer = new StringTokenizer(line);

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // lose the mix token
        tokenizer.nextToken();

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the word token
        Text word = new Text(tokenizer.nextToken());

        // output (mix, word) where mix in {0, 1, ..., n-2, n-1}
        output.collect(
                new IntWritable(randomGenerator
                        .nextInt(ProbableMixAssignerEntryPoint.getModerator()
                                .getNumberOfMixes())), word);

    }
}
```

A.4 org.manea.vlad.assignedmixcounter

A.4.1 AssignedMixCounterEntryPoint

```
package org.manea.vlad.assignedmixcounter;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.fs.Path;

import org.manea.vlad.entrypoint.EntryPoint;

/**
 * is the configure class for assigned mix count of messages
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
public class AssignedMixCounterEntryPoint extends EntryPoint {

    /**
     * constructs the entry point for assigned mix counter
     *
     * @param inputFolder
     *            the input folder
     * @param outputFolder
     *            the output folder
     * @throws Exception
     */
    public AssignedMixCounterEntryPoint(String inputFolder, String outputFolder) {

        // call super
        super(inputFolder, outputFolder);

    }

    /**
     * runs for assigned mix counter
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create a job configuration
```

```
        JobConf job = new JobConf(AssignedMixCounterEntryPoint.class);

        // set job name
        job.setJobName(AssignedMixCounterEntryPoint.class.getName());

        // set output key and value classes
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(IntWritable.class);

        // set map, combine and reduce classes
        job.setMapperClass(AssignedMixCounterMapper.class);
        job.setCombinerClass(AssignedMixCounterReducer.class);
        job.setReducerClass(AssignedMixCounterReducer.class);

        // set input and output paths for files
        FileInputFormat.setInputPaths(job, new Path(getInputFolder()));
        FileOutputFormat.setOutputPath(job, new Path(getOutputFolder()));

        // run job
        JobClient.runJob(job);

    }
}
```

A.4.2 AssignedMixCounterMapper

```
package org.manea.vlad.assignedmixcounter;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the assigned mix count map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class AssignedMixCounterMapper extends MapReduceBase implements
Mapper<LongWritable, Text, IntWritable, IntWritable> {

// constant one
private static final IntWritable ONE = new IntWritable(1);

/**
 * maps (mix, word) to (mix, 1)
 *
 * @param key
 *            number of file
 * @param value
 *            contents of file
 * @param output
 *            output to write to
 * @param reporter
 *            reporter of actions
 * @throws IOException
 */
@Override
public void map(LongWritable key, Text value,
OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
throws IOException {

// get line
String line = value.toString();

// create string token generator
StringTokenizer tokenizer = new StringTokenizer(line);

// test for message token existence
if (!tokenizer.hasMoreTokens())
throw new IOException();

// get the mix token
IntWritable mix = new IntWritable(Integer.parseInt(tokenizer
.nextToken()));

// output (mix, 1)
output.collect(mix, ONE);

}

}
```

A.4.3 AssignedMixCounterReducer

```
package org.manea.vlad.assignedmixcounter;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.mapred.*;
import org.apache.hadoop.io.*;
```

```
/**
 * is the assigned mix count reduce class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class AssignedMixCounterReducer extends MapReduceBase implements
        Reducer<IntWritable, IntWritable, IntWritable, IntWritable> {

    /**
     * reduces (mix, 1) to (mix, count(mix))
     *
     * @param key
     *            number of mix
     * @param value
     *            counter of mixes
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void reduce(IntWritable key, Iterator<IntWritable> values,
            OutputCollector<IntWritable, IntWritable> output, Reporter reporter)
            throws IOException {

        // initialize sum value
        int counter = 0;

        // iterate all values in iterator
        while (values.hasNext()) {

            // add count to sum
            counter += values.next().get();

        }

        // output (mix, count(mix))
        output.collect(key, new IntWritable(counter));

    }

}
```

A.5 org.manea.vlad.generalminimumcomputer

A.5.1 GeneralMinimumComputerEntryPoint

```
package org.manea.vlad.generalminimumcomputer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.fs.Path;

import org.manea.vlad.entrypoint.EntryPoint;

/**
 * is the configure class for general minimum compute of mix counts
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
public class GeneralMinimumComputerEntryPoint extends EntryPoint {

    /**
     * constructs the entry point for general minimum computer
     *
     * @param inputFolder
     *            the input folder
     * @param outputFolder
     *            the output folder
     * @throws Exception
     */
    public GeneralMinimumComputerEntryPoint(String inputFolder,
            String outputFolder) {

        // call super
        super(inputFolder, outputFolder);

    }

    /**
     * runs for general minimum computer
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create a job configuration
        JobConf job = new JobConf(GeneralMinimumComputerEntryPoint.class);

        // set job name
        job.setJobName(GeneralMinimumComputerEntryPoint.class.getName());
```

```
        // set output key and value classes
        job.setOutputKeyClass(BooleanWritable.class);
        job.setOutputValueClass(IntWritable.class);

        // set map and reduce classes
        job.setMapperClass(GeneralMinimumComputerMapper.class);
        job.setReducerClass(GeneralMinimumComputerReducer.class);

        // set input and output paths for files
        FileInputFormat.setInputPaths(job, new Path(getInputFolder()));
        FileOutputFormat.setOutputPath(job, new Path(getOutputFolder()));

        // run job
        JobClient.runJob(job);

    }

}
```

A.5.2 GeneralMinimumComputerMapper

```
package org.manea.vlad.generalminimumcomputer;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the general minimum compute map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class GeneralMinimumComputerMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, BooleanWritable, IntWritable> {

    // constant true
    static final BooleanWritable TRUE = new BooleanWritable(true);

    /**
     * maps (mix, count) to (TRUE, count)
     *
     * @param key
     *            number of file
     * @param value
     *            contents of file
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void map(LongWritable key, Text value,
            OutputCollector<BooleanWritable, IntWritable> output,
            Reporter reporter) throws IOException {

        // get line
        String line = value.toString();

        // create string token generator
        StringTokenizer tokenizer = new StringTokenizer(line);

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // lose the message token
        tokenizer.nextToken();

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the count token
        IntWritable count = new IntWritable(Integer.parseInt(tokenizer
                .nextToken()));

        // output (TRUE, count)
        output.collect(TRUE, count);

    }

}
```

A.5.3 GeneralMinimumComputerReducer

```
package org.manea.vlad.generalminimumcomputer;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.mapred.*;
```

```
import org.apache.hadoop.io.*;

/**
 * is the general minimum compute reduce class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class GeneralMinimumComputerReducer extends MapReduceBase implements
        Reducer<BooleanWritable, IntWritable, BooleanWritable, IntWritable> {

    // constant true
    static final BooleanWritable TRUE = new BooleanWritable(true);

    /**
     * reduces (TRUE, count) to (TRUE, minimum)
     *
     * @param key
     *            TRUE
     * @param value
     *            count of messages assigned to mix
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void reduce(BooleanWritable key, Iterator<IntWritable> values,
            OutputCollector<BooleanWritable, IntWritable> output,
            Reporter reporter) throws IOException {

        // initialize current count and minimum value
        int count, min = Integer.MAX_VALUE;

        // iterate all values in iterator
        while (values.hasNext()) {

            // get count and update minimum
            count = values.next().get();
            min = (min > count ? count : min);

        }

        // update minimum as multiple of n
        min = Math.min(min, min
                / GeneralMinimumComputerEntryPoint.getModerator()
                        .getNumberOfMixes()
                * GeneralMinimumComputerEntryPoint.getModerator()
                        .getNumberOfMixes());

        // output (TRUE, count(mix))
        output.collect(TRUE, new IntWritable(min));

    }

}
```

A.6 org.manea.vlad.messageselector

A.6.1 MessageSelectorEntryPoint

```
package org.manea.vlad.messageselector;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.fs.Path;

import org.manea.vlad.entrypoint.EntryPoint;

/**
 * is the configure class for message select
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
public class MessageSelectorEntryPoint extends EntryPoint {

    /**
     * constructs the entry point for message filter
     *
     * @param inputFolder
     *            the input folder
     * @param outputFolder
     *            the output folder
     * @throws Exception
     */
    public MessageSelectorEntryPoint(String inputFolder, String outputFolder) {

        // call super
        super(inputFolder, outputFolder);

    }

    /**
     * runs for message selector
```

```
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create a job configuration
        JobConf job = new JobConf(MessageSelectorEntryPoint.class);

        // set job name
        job.setJobName(MessageSelectorEntryPoint.class.getName());

        // set output key and value classes
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(Text.class);

        // set map and reduce classes
        job.setMapperClass(MessageSelectorMapper.class);
        job.setReducerClass(MessageSelectorReducer.class);

        // set input and output paths for files
        FileInputFormat.setInputPaths(job, new Path(getInputFolder()));
        FileOutputFormat.setOutputPath(job, new Path(getOutputFolder()));

        // run job
        JobClient.runJob(job);

    }
}
```

A.6.2 MessageSelectorMapper

```
package org.manea.vlad.messageselector;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the message selector map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class MessageSelectorMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, IntWritable, Text> {

    /**
     * maps (mix, word) to (mix, word)
     *
     * @param key
     *            number of file
     * @param value
     *            contents of file
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void map(LongWritable key, Text value,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // get line
        String line = value.toString();

        // create string token generator
        StringTokenizer tokenizer = new StringTokenizer(line);

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the mix token
        IntWritable mix = new IntWritable(Integer.parseInt(tokenizer
                .nextToken()));

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the word token
        Text word = new Text(tokenizer.nextToken());

        // output (mix, word)
        output.collect(mix, word);

    }
}
```

A.6.3 MessageSelectorReducer

```
package org.manea.vlad.messageselector;

import java.io.IOException;
import java.util.Iterator;
import java.util.Random;
import java.util.Vector;

import org.apache.hadoop.mapred.*;
import org.apache.hadoop.io.*;

/**
 * is the message select reduce class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class MessageSelectorReducer extends MapReduceBase implements
        Reducer<IntWritable, Text, IntWritable, Text> {

    // constant max
    static final IntWritable MAX = new IntWritable(Integer.MAX_VALUE);

    /**
     * reduces (mix, word) to (mix, word)<br />
     * only a number of selected messages
     *
     * @param key
     *            number of mix
     * @param value
     *            word for the mix
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void reduce(IntWritable key, Iterator<Text> values,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // initialize an array of messages
        Vector<String> words = new Vector<String>();

        // initialize an array of booleans
        Vector<Boolean> selected = new Vector<Boolean>();

        // create a random number generator
        Random randomGenerator = new Random();

        // iterate all values in iterator
        while (values.hasNext()) {

            // add value to words and add true to selected
            words.add(values.next().toString());
            selected.add(true);

        }

        // set index and limit
        int index, limit = words.size()
                - MessageSelectorEntryPoint.getModerator()
                        .getNumberOfMessages();

        // select up to limit messages to be discarded
        for (int step = 0; step < limit; ++step) {

            do {

                // select at random an index
                index = randomGenerator.nextInt(words.size());

            } while (!selected.elementAt(index));

            // discard message at index
            selected.set(index, false);

        }

        // iterate all messages
        for (index = 0; index < words.size(); ++index) {

            // output (mix, word) if word is not discarded and else (MAX, word)
            if (selected.elementAt(index)) {
                output.collect(key, new Text(words.elementAt(index)));
            } else {
                output.collect(MAX, new Text(words.elementAt(index)));
            }

        }

    }
}
```

A.7 org.manea.vlad.messagefilter

A.7.1 MessageFilterEntryPoint

```
package org.manea.vlad.messagefilter;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.fs.Path;

import org.manea.vlad.entrypoint.EntryPoint;

/**
 * is the configure class for message filter of messages
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
public class MessageFilterEntryPoint extends EntryPoint {

    // filter criteria
    private boolean criteria = false;

    /**
     * gets the criteria
     *
     * @return the criteria
     */
    boolean isCriteria() {

        // get criteria
        return criteria;

    }

    /**
     * sets the criteria
     *
     * @param criteria
     *            the criteria to set
     */
    private void setCriteria(boolean criteria) {

        // set number of messages
        this.criteria = criteria;

    }

    /**
     * constructs the entry point for message filter
     *
     * @param inputFolder
     *            the input folder
     * @param outputFolder
     *            the output folder
     * @param criteria
     *            the criteria
     * @throws Exception
     */
    public MessageFilterEntryPoint(String inputFolder, String outputFolder,
            boolean criteria) {

        // call super
        super(inputFolder, outputFolder);

        // set criteria
        setCriteria(criteria);

    }

    /**
     * runs for message filter
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create a job configuration
        JobConf job = new JobConf(MessageFilterEntryPoint.class);

        // set job name
        job.setJobName(MessageFilterEntryPoint.class.getName());

        // set output key and value classes
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(Text.class);

        // set map class according to criteria
        if (isCriteria())
            job.setMapperClass(MessageFilterPassMapper.class);
        else
            job.setMapperClass(MessageFilterFailMapper.class);

        // set input and output paths for files
        FileInputFormat.setInputPaths(job, new Path(getInputFolder()));
        FileOutputFormat.setOutputPath(job, new Path(getOutputFolder()));

        // set no reducers
        job.setNumReduceTasks(0);
```

```
        // run job
        JobClient.runJob(job);

    }
}
```

A.7.2 MessageFilterPassMapper

```
package org.manea.vlad.messagefilter;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the current input message assign map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class MessageFilterPassMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, IntWritable, Text> {

    // constant max
    static final IntWritable MAX = new IntWritable(Integer.MAX_VALUE);

    /**
     * maps (mix, word) to (mix, word) if mix is not MAX<br />
     * discards other (mix, word) pairs
     *
     * @param key
     *            number of file
     * @param value
     *            contents of file
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void map(LongWritable key, Text value,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // get line
        String line = value.toString();

        // create string token generator
        StringTokenizer tokenizer = new StringTokenizer(line);

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the mix token
        int mix = Integer.parseInt(tokenizer.nextToken());

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the word token
        Text word = new Text(tokenizer.nextToken());

        // output (word, mix)
        if (mix != Integer.MAX_VALUE)
            output.collect(new IntWritable(mix), word);

    }
}
```

A.7.3 MessageFilterFailMapper

```
package org.manea.vlad.messagefilter;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the current input message assign map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class MessageFilterFailMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, IntWritable, Text> {
```

```
    // constant max
    static final IntWritable MAX = new IntWritable(Integer.MAX_VALUE);

    /**
     * maps (mix, word) to (mix, word) if mix is not MAX<br />
     * discards other (mix, word) pairs
     *
     * @param key
     *            number of file
     * @param value
     *            contents of file
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void map(LongWritable key, Text value,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // get line
        String line = value.toString();

        // create string token generator
        StringTokenizer tokenizer = new StringTokenizer(line);

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the mix token
        int mix = Integer.parseInt(tokenizer.nextToken());

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the word token
        Text word = new Text(tokenizer.nextToken());

        // output (word, mix)
        if (mix == Integer.MAX_VALUE)
            output.collect(MAX, word);

    }

}
```

A.8 org.manea.vlad.messagerandomizer

A.8.1 MessageRandomizerEntryPoint

```
package org.manea.vlad.messagerandomizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.fs.Path;

import org.manea.vlad.entrypoint.EntryPoint;

/**
 * is the configure class for message randomize
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
public class MessageRandomizerEntryPoint extends EntryPoint {

    // redistribute of messages
    private boolean redistribute = false;

    /**
     * gets the redistribute
     *
     * @return the redistribute
     */
    boolean isRedistribute() {

        // get redistribute
        return redistribute;

    }

    /**
     * sets the redistribute<br />
     * if redistribute is true, the messages are sent evenly to all n mixes<br />
     * if redistribute is false, the messages are all sent to the next mix<br />
     *
     * @param numberOfMessages
     *            the numberOfMessages to set
     */
    private void setRedistribute(boolean redistribute) {

        // set redistribute
```

```
        this.redistribute = redistribute;
    }

    /**
     * constructs the entry point for message randomizer
     *
     * @param inputFolder
     *            the input folder
     * @param outputFolder
     *            the output folder
     * @param redistribute
     *            the redistribute
     * @throws Exception
     */
    public MessageRandomizerEntryPoint(String inputFolder, String outputFolder,
            boolean redistribute) {

        // call super
        super(inputFolder, outputFolder);

        // set redistribute
        setRedistribute(redistribute);

    }

    /**
     * runs for message randomizer
     *
     * @throws Exception
     */
    public void run() throws Exception {

        // create a job configuration
        JobConf job = new JobConf(MessageRandomizerEntryPoint.class);

        // set job name
        job.setJobName(MessageRandomizerEntryPoint.class.getName());

        // set output key and value classes
        job.setOutputKeyClass(IntWritable.class);
        job.setOutputValueClass(Text.class);

        // set map class
        job.setMapperClass(MessageRandomizerMapper.class);

        // set reduce class according to redistribute
        if (isRedistribute())
            job.setReducerClass(MessageRandomizerRedistributeReducer.class);
        else
            job.setReducerClass(MessageRandomizerProceedReducer.class);

        // set input and output paths for files
        FileInputFormat.setInputPaths(job, new Path(getInputFolder()));
        FileOutputFormat.setOutputPath(job, new Path(getOutputFolder()));

        // run job
        JobClient.runJob(job);

    }
}
```

A.8.2 MessageRandomizerMapper

```
package org.manea.vlad.messagerandomizer;

import java.io.IOException;
import java.util.StringTokenizer;

import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;

/**
 * is the message randomize map class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
class MessageRandomizerMapper extends MapReduceBase implements
        Mapper<LongWritable, Text, IntWritable, Text> {

    /**
     * maps (mix, word) to (mix, word)
     *
     * @param key
     *            number of file
     * @param value
     *            contents of file
     * @param output
     *            output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void map(LongWritable key, Text value,
```

```
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // get line
        String line = value.toString();

        // create string token generator
        StringTokenizer tokenizer = new StringTokenizer(line);

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the mix token
        IntWritable mix = new IntWritable(Integer.parseInt(tokenizer
                .nextToken()));

        // test for message token existence
        if (!tokenizer.hasMoreTokens())
            throw new IOException();

        // get the word token
        Text word = new Text(tokenizer.nextToken());

        // output (mix, word)
        output.collect(mix, word);

    }

}
```

A.8.3 MessageRandomizerReducer

```
package org.manea.vlad.messagerandomizer;

import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.security.SecureRandom;
import java.util.Collections;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.Random;
import java.util.Vector;
import java.util.Queue;

import org.apache.hadoop.mapred.*;
import org.apache.hadoop.io.*;
import org.manea.vlad.moderator.Moderator;

/**
 * is the message randomize reduce class
 *
 * @author administrator
 */
@SuppressWarnings("deprecation")
abstract class MessageRandomizerReducer extends MapReduceBase implements
        Reducer<IntWritable, Text, IntWritable, Text> {

    // initialize an array of messages
    protected Vector<String> words = new Vector<String>();

    // initialize an array of booleans
    protected Vector<Integer> permutation;

    // queues
    private Vector<Queue<String>> queues = new Vector<Queue<String>>();

    // set current queue
    private int currentQueue = 0;

    // create buffered reader
    private BufferedReader bufferedReader = null;

    // create print writer
    private PrintWriter printWriter = null;

    // create a random number generator
    Random randomGenerator = new SecureRandom();

    /**
     * returns a random permutation of given size
     *
     * @param size
     *            the size of permutation
     * @return a vector with given permutation
     */
    protected Vector<Integer> getRandomPermutation(int size) {

        // create an array of size
        Vector<Integer> permutation = new Vector<Integer>();

        // iterate through the array and set identity permutation
        for (int index = 0; index < size; ++index)
            permutation.add(index);
```

```
    // iterate through the array and set random swap
    for (int index = 0; index < size; ++index)
        Collections.swap(permutation, index,
                index + randomGenerator.nextInt(size - index));

    // return the constructed array
    return permutation;

}

/**
 * reduces (mix, word) to (mix, word), but in other order<br />
 * if redistribute is true, messages are evenly redistributed to mixes<br />
 * if redistribute is false, messages are all redistributed to the next mix
 *
 * @param key
 *            number of mix
 * @param value
 *            word for the mix
 * @param output
 *            -i output to write to
 * @param reporter
 *            reporter of actions
 * @throws IOException
 */
@Override
public void reduce(IntWritable key, Iterator<Text> values,
        OutputCollector<IntWritable, Text> output, Reporter reporter)
        throws IOException {

    // remove words
    words.clear();

    // remove queues
    queues.clear();

    // iterate queues
    for (int index = 0; index < Moderator.INSTANCE.getNumberOfQueues(); ++index)
        queues.add(new LinkedList<String>());

    try {

        // read from mix file
        // readFromMixFile(key.get());

    } finally {

        // close buffer reader
        if (bufferedReader != null)
            bufferedReader.close();

    }

    // add dummy messages
    /*
     * addDummyMessages(Moderator.INSTANCE.getNumberOfMemory() -
     * readMessages, key.get());
     */

    // add input messages
    int indexInput = addInputMessages(values);

    // check for multiple of n messages
    if (indexInput
            % MessageRandomizerEntryPoint.getModerator().getNumberOfMixes() != 0)
        throw new IOException();

    // add to words from queues
    addFromQueuesToWords(indexInput);

    // write messages to file
    writeToMixFile(key.get());

    // set random permutation
    permutation = getRandomPermutation(indexInput);

}

/**
 * writes back to the file the remaining messages
 *
 * @param key
 *            the key
 * @throws IOException
 */
private void writeToMixFile(int key) throws IOException {

    // create print writer
    printWriter = new PrintWriter(new FileWriter(
            Moderator.INSTANCE.getMixFile(key), false));

    // print the first number
    printWriter.println(currentQueue);

    // print all messages in queues
    for (int index = 0; index < Moderator.INSTANCE.getNumberOfQueues(); ++index) {
```

```
            // iterate queue and write message
            while (!queues.get(index).isEmpty())
                printWriter.println(index + " " + queues.get(index).poll());

        }

    }

    /**
     * adds messages to words from queues
     *
     * @param indexInput
     *            the number of words extracted from the queues
     */
    private void addFromQueuesToWords(int indexInput) {

        // add until index input
        for (int index = 0; index < indexInput;) {

            // iterate the current queue
            for (; !queues.get(currentQueue).isEmpty() && index < indexInput; ++index) {

                // take a message from the queue
                words.add(queues.get(currentQueue).poll());

            }

            // go to the next queue
            currentQueue = (currentQueue + 1)
                    % Moderator.INSTANCE.getNumberOfQueues();

        }

    }

    /**
     * adds input messages
     *
     * @param values
     *            the iterator with messages
     * @return the number of read messages
     * @throws IOException
     */
    private int addInputMessages(Iterator<Text> values) throws IOException {

        // number of input messages
        int indexInput = 0;

        // iterate all values in iterator and add them to words
        while (values.hasNext()) {

            // add input message to queue
            queues.get(
                    randomGenerator.nextInt(Moderator.INSTANCE
                            .getNumberOfQueues()))
                    .add(values.next().toString());

            // new input message
            ++indexInput;

        }

        // return index input
        return indexInput;

    }

}
```

A.8.4 MessageRandomizerProceedReducer

```
package org.manea.vlad.messagerandomizer;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.mapred.*;
import org.apache.hadoop.io.*;
import org.manea.vlad.moderator.Recipient;

/**
 * is the message randomize proceed reduce class
 *
 * @author administrator
 */
class MessageRandomizerProceedReducer extends MessageRandomizerReducer {

    /**
     * reduces (mix, word) to (mix, word), but in other order<br />
     * if redistribute is true, messages are evenly redistributed to mixes<br />
     * if redistribute is false, messages are all redistributed to the next mix
     *
     * @param key
     *            number of mix
     * @param value
     *            word for the mix
     * @param output
```

```
     *            -i output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void reduce(IntWritable key, Iterator<Text> values,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // call parent
        super.reduce(key, values, output, reporter);

        // get next mix
        IntWritable nextKey = new IntWritable((key.get() + 1)
                % MessageRandomizerEntryPoint.getModerator().getNumberOfMixes());

        try {

            // iterate messages and output them permuted and re-encrypted
            for (int index = 0; index < words.size(); ++index) {

                output.collect(nextKey, new Text(Recipient.getElGamal()
                        .reencryptLongString(words.get(permutation.get(index)))));

            }

        } catch (Exception exception) {

            // throw further exception
            exception.printStackTrace();
            throw new IOException();

        }

    }

}
```

A.8.5 MessageRandomizerRedistributeReducer

```
package org.manea.vlad.messagerandomizer;

import java.io.IOException;
import java.util.Iterator;

import org.apache.hadoop.mapred.*;
import org.apache.hadoop.io.*;
import org.manea.vlad.moderator.Recipient;

/**
 * is the message randomize redistribute reduce class
 *
 * @author administrator
 */
class MessageRandomizerRedistributeReducer extends MessageRandomizerReducer {

    /**
     * reduces (mix, word) to (mix, word), but in other order<br />
     * if redistribute is true, messages are evenly redistributed to mixes<br />
     * if redistribute is false, messages are all redistributed to the next mix
     *
     * @param key
     *            number of mix
     * @param value
     *            word for the mix
     * @param output
     *            -i output to write to
     * @param reporter
     *            reporter of actions
     * @throws IOException
     */
    @Override
    public void reduce(IntWritable key, Iterator<Text> values,
            OutputCollector<IntWritable, Text> output, Reporter reporter)
            throws IOException {

        // call parent
        super.reduce(key, values, output, reporter);

        // get the number of messages per mix
        int messagesPerMix = words.size()
                / MessageRandomizerEntryPoint.getModerator().getNumberOfMixes();

        // iterate through all messages
        for (int index = 0; index < words.size(); ++index) {

            // get the mix to be given this message
            int mix = permutation.elementAt(index) / messagesPerMix;

            try {

                // output (mix, word)
                output.collect(new IntWritable(mix), new Text(Recipient
                        .getElGamal().reencryptLongString(words.elementAt(index))));
```

```
            } catch (Exception exception) {
                // throw further exception
                throw new IOException();
            }
        }
    }
}
```

Bibliography

[1] Chaum, David
The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability
Journal of Cryptology, volume 1, issue 1, 1988

[2] Ţiplea, Ferucio Laurenţiu
Algebraic Foundations of Computer Science
Polirom, 2006

[3] Scholz, Immanuel
Dining Cryptographers - The Protocol
Chaos Communication Congress, Berlin, 2007

[4] Hao, Feng; Zieliński Piotr
A 2-Round Anonymous Veto Protocol
International Workshop on Security Protocols, Cambridge, 2006

[5] Boneh, Dan
The Decision Diffie Hellman Problem
Algorithmic Number Theory Symposium,
Lecture Notes in Computer Science, volume 1423, Springer, 1998

[6] Chaum, David
Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms
Communications of the ACM, volume 24, issue 2, 1981

[7] Fischer-Hübner, Simone
Mix Nets: Unlinkability of Sender and Recipient
Privacy-Enhancing Technologies, PhD course, 2001

[8] Edman, Matthew; Yener, Bülent
On Anonymity in an Electronic Society: A Survey of Anonymous Communication Systems
ACM Computing Surveys, 2009

[9] Serjantov, Andrei; Dingledine, Roger; Syverson, Paul
From a trickle to a flood: Active attacks on several mix types
Information Hiding Workshop
Lecture Notes in Computer Science, volume 2578, Springer, 2002

[10] Serjantov, Andrei; Danezis, George
Towards an information theoretic metric for anonymity
Privacy Enhancing Technologies,
Lecture Notes in Computer Science, volume 2482, Springer, 2003

[11] Kesdogan, Dogan; Egner, Jan; Brüschkes, Roland
Stop-and-Go-MIXes Providing Probabilistic Anonymity in an Open System
Information Hiding,
Lecture Notes in Computer Science, volume 1525, Springer, 1998

[12] O'Connor, Luke
On blending attacks for mixes with memory
Information Hiding,
Lecture Notes in Computer Science, volume 3727, Springer, 2005

[13] Chen, Beifang
Combinatorial Analysis
Math Course taught at Hong Kong University of Science and Technology, 2011

[14] Díaz, Claudia; Serjantov, Andrei
Generalising Mixes
Privacy Enhancing Technologies Symposium,
Lecture Notes in Computer Science, volume 2760, Springer, 2003

[15] Danezis, George; Sassaman, Len
Heartbeat Traffic to Counter ($n-1$) Attacks - Red-Green-Black Mixes
Workshop on Privacy in the Electronic Society, 2003

[16] Weisstein, Eric W.
Negative Binomial Distribution
From MathWorld - A Wolfram Web Resource
$http://mathworld.wolfram.com/NegativeBinomialDistribution.html$

[17] Ogata, Wakaha; Kuroshawa, Kaoru; Sako, Kazue; Takatani, Kazunori
Fault Tolerant Anonymous Channel
Information and Communications Security,
Lecture Notes in Computer Science, volume 1334, Springer, 1997

[18] Golle, Philippe; Zhong, Sheng; Boneh, Dan; Jakobsson, Markus; Juels, Ari
Optimistic Mixing for Exit-Polls
International Conference on the Theory and Application of Cryptology and Information Security
Lecture Notes in Computer Science, volume 2591, Springer, 2002

[19] Boneh, Dan; Golle, Philippe
Almost Entirely Correct Mixing With Applications to Voting
ACM Conference on Computer and Communications Security 2002

[20] Canetti, Ran; Rivest, Ronald
Verifiable Mix-Net Voting
Special Topics in Cryptography Course, MIT, 2004
$http://courses.csail.mit.edu/6.897/spring04/L19.pdf$

[21] Jakobsson, Markus; Juels, Ari
Millimix: Mixing in Small Batches
DIMACS Technical Report 99-33, 1999

[22] Waksman, Abraham
A Permutation Network
Journal of the ACM, volume 15, issue 1, 1968

[23] Cormen, Thomas; Leiserson, Charles; Rivest, Ronald
Introduction to Algorithms Second Edition
MIT Press, 2002

[24] Abe, Masayuki
Mix-Networks on Permutation Networks
Asiacrypt 1999

[25] Jakobsson, Markus; Juels, Ari; Rivest, Ronald
Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking
USENIX Security Symposium, 2002

[26] Golle, Philippe; Juels, Ari
Parallel Mixing
ACM Conference on Computer and Communications Security, 2004

[27] Latif, Usman
Random Permutation Generation
$http://www.techuser.net/randpermgen.html$

[28] Anonymizer, Inc.
Anonymizer
$http://www.anonymizer.com$

[29] Reiter, Michael; Rubin, Aviel
Crowds: Anonymity for Web Transactions
ACM Transactions on Information and System Security, volume 1, issue 1, 1998

[30] Goldschlag, David; Reed, Michael; Syverson, Paul
Hiding Routing Information
Information Hiding First International Workshop
Lecture Notes in Computer Science, volume 1174, Springer, 1996

[31] Freedman, Michael; Morris, Robert
Tarzan: A Peer-to-Peer Anonymizing Network Layer
ACM Conference on Computer and Communication Security, 2002

[32] Rennhard, Marc; Plattner, Bernhard
Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection
Workshop on Privacy in the Electronic Society, 2002

[33] Dingledine, Roger; Mathewson, Nick; Syverson, Paul
Tor: The Second-Generation Onion Router
USENIX Security Symposium, 2004

[34] The Tor Project, Inc.
Tor
$https://www.torproject.org; https://www.torproject.org/about/overview.html.en$

[35] Dean, Jeffrey; Ghemawat, Sanjay
MapReduce: Simplified Data Processing on Large Clusters
Symposium on Operating System Design and Implementation, 2004
$http://labs.google.com/papers/mapreduce.html$

[36] Apache Software Foundation
Apache Hadoop
$http://hadoop.apache.org; http://hadoop.apache.org/common/docs/current/$

[37] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad
The Java Language Specification Third Edition
Addison Wesley, 2005

[38] Eck, David
Introduction to Programming Using Java
Department of Math and Computer Science, Hobart and William Colleges, 2011
$http://math.hws.edu/javanotes/$

[39] Larman, Craig
Applying UML and Patterns Second Edition
Prentice Hall, 2002
$http://www.utdallas.edu/\ chung/CS6354/applying-uml-and-patterns.pdf$

[40] Miniwatts Marketing Group
Internet Growth Statistics
$http://www.internetworldstats.com/emarketing.htm; http://www.internetworldstats.com/stats.htm$

[41] McLuhan, Marshall
Understanding Media: the Extensions of Man
McGraw Hill, 1964

[42] Sweeney, Latanya
k-**Anonymity: a Model for Protecting Privacy**
International Journal of Uncertainty, Fuzziness and Knowledge-based systems, volume 10, issue 5, 2002

[43] Pfitzmann, Andreas; Hansen, Marit
Anonymity, Unlinkability, Undetectability, Unobservability, Pseudonymity, and Identity Management
A Consolidated Proposal for Terminology
$http://dud.inf.tu-dresden.de/Anon_Terminology.shtml$

[44] Fowler, Martin
UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language
Addison Wesley Professional, 2004