P. J. Safarik University

Faculty of Science

SECURE MULTI-PARTY COMPUTATION OF A RANDOM PERMUTATION

DIPLOMA THESIS

Field of Study:	**Computer Science**
Institute:	**Institute of Computer Science**
Tutor:	**RNDr. Jozef Jirásek PhD.**

Košice 2012

Bc. Ján Jerguš

Thanks

I would like to thank RNDr. Jozef Jirásek PhD. for his valuable efforts in supervising this project.

Abstrakt

Popíšeme kryptografický protokol, ktorý umožňuje ľubovoľnej skupine φ ľudí spoločne (bez pomoci dôveryhodnej tretej strany) vygenerovať náhodnú permutáciu π prvkov a potom postupne odhaľovať prvky tejto permutácie len určeným účastníkom. Na dešifrovanie prvku vygenerovanej permutácie musí účastník protokolu získať súhlas aspoň ψ účastníkov ($2 \leq \psi \leq \varphi$).

Typickým využitím takéhoto protokolu je miešanie kariet pri tzv. *mentálnom pokri*, teda v protokoloch, ktoré umožňujú skupine ľudí hrať poker prostredníctvom počítačovej siete bez pomoci dôveryhodného servera.

Súčasťou práce je aj implementácia nášho protokolu ako knižnica v jazyku Java, ktorá je ľahko použiteľná v ľubovoľnej sieťovej aplikácii.

Abstract

We describe a cryptographic protocol that allows a group of φ people to cooperatively (without the help of a trusted third party) generate a random permutation of π elements, and then gradually reveal elements of the generated permutation only to specific participants. To decrypt an element of the generated permutation, a participant must gain cooperation of at least ψ participants ($2 \leq \psi \leq \varphi$).

A typical use of such protocol is to shuffle cards in the so-called *mental poker* protocols, which allow a group of people to play poker over a computer network without a trusted server.

We also give an implementation of our protocol as a Java library, which can be easily used by any network application.

Contents

Notation

$:=$	'is defined as' or 'is set to' (see also Definition 2.8)
$\mathbb{N}$	the set of all natural numbers $\{0, 1, 2, \ldots\}$
$\mathbb{Z}$	the set of all integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
$\mathbb{Z}^+$	the set of all positive integers $\{1, 2, \ldots\}$
$\mathbb{Z}_n$	$:= \{0, 1, 2, \ldots, n-1\}$ (Definition 2.1)
$\mathbb{Z}_n^*$	$:= \{z \in \mathbb{Z}_n : \gcd(z, n) = 1\}$ (Definition 2.1)
$(a_1, \ldots, a_n)$	the n-tuple (ordered list) of $a_1, \ldots, a_n$
$\lvert A \rvert$	the number of elements in a (finite) set A
$A \setminus B$	set difference, ie. $\{a \in A : a \notin B\}$
$A \times B$	Cartesian product, ie. $\{(a, b) : a \in A \wedge b \in B\}$
$f \circ g$	the composition of functions f and g, ie. $(f \circ g)(x) = g(f(x))$
$\lfloor a \rfloor$	the largest integer b such that $b \leq a$
$a \mid b$	a divides b, ie. $b = ka$ for some $k \in \mathbb{Z}$
$\gcd(a, b)$	the greatest common divisor of a and b
$\mathrm{lcm}(a, b)$	the least common multiple of a and b
$\phi(n)$	Euler's totient function (Definition 2.2)
$a = b \pmod{n}$	a is congruent to b modulo n (Definition 2.7)
$a^{-1} \pmod{n}$	the inverse of a modulo n (Definition 2.10)
$\frac{a}{b} \pmod{n}$	$:= ab^{-1}$ (Definition 2.10)
$a^{-b} \pmod{n}$	$:= (a^b)^{-1} = (a^{-1})^b$ (Definition 2.10)

φ	the number of participants of the protocol
ψ	the minimum number of participants necessary to uncover an element of the permutation
π	the size of the generated permutation
n	typically the modulus of a congruence (which may fulfil the role of a participant's public key)

Introduction

In many situations, people who do not know each other well, and therefore have no reason to trust each other, need to cooperate to achieve some common goal. Some of them might even have conflicting goals of their own (such as a buyer and a seller who cooperate on completing a trade, or a group of players who want to enjoy a game of poker while each of them also wants to win). To this end, they use various techniques which guarantee that none of them can gain an unfair advantage – for example, flipping a coin to provide true randomness or using a deck of playing card to achieve secrecy and provability. Unfortunately, such techniques often require physical presence of the participants, or some of their security properties are void. For example, if Alice flips a coin, then Bob, who is not physically present, cannot verify that Alice didn't lie about the result – or whether she flipped a coin at all. For years, researchers, mainly in the field of cryptography, have tried to devise techniques which would guarantee the same security properties as these physical techniques, but without the need for physical presence of the participants (only their ability to communicate with each other).

The most straightforward way to achieve this goal is to use a *trusted third party* – a special participant of the protocol who is trusted by all other participants. This idea is not original to cryptography – it is often found in real life as well (a notary public, a card dealer in a casino, etc.). However, it is not always practical or even possible to establish such a trusted third party. As a result, cryptographers have been working to find alternatives to the aforementioned techniques that would retain the various security properties even when no participant of the protocol can be trusted. A protocol which allows a group of participants without a trusted third party to cooperatively achieve some goal, but allows each of them to keep various partial information secret, is usually called a *secure multi-party computation*.

In this thesis, we describe a secure multi-party computation protocol for generating a random permutation of a given size. After the permutation is generated, elements of this permutation can be gradually revealed to different participants of the protocol.

Such protocol can be put to use in many situations – for instance, we could use it to shuffle a deck of cards (a permutation of 52 elements), or to randomly assign different tasks to a group of people (a permutation of size equal to the number of tasks).

A more detailed description of the protocol is given in the next chapter. We also list various assumptions and requirements that ensure that the properties of the protocol are as close as possible to the properties of the aforementioned physical techniques.

Previous research

Various problems that could be classified as secure multi-party computation have been studied for many years. Perhaps the best known example is „coin flipping by telephone“, first described in [1]. Later, the notion of general-purpose secure multi-party computation (ie. designing a protocol for secure multi-party computation of any computable function) has been studied. A good overview can be found in [2], which gives [3] as the original source.

Although, as far as we know, the specific problem of generating a random permutation and then disclosing its elements to different participants using secure multi-party computation, as stated in our thesis, has not been studied before, similar protocols have often been used in the context of other larger protocols (e.g. voting schemes or mental card game protocols). Indeed, all mental card game protocols need to shuffle the deck of cards at one point or another, and therefore necessarily contain a special case of our protocol as a subroutine.

The problem of mental card games was first stated in [4], which describes the requirements for playing *mental poker* and then discusses the feasibility of different solutions. First mental poker implementation that satisfied all security requirements, including confidentiality of players' strategies, was described in [5] and later generalized to accommodate different card games in [6]. Our work is a modified and generalized version of the card-shuffling protocol from [7] and [8]. Again, a good overview of the different mental poker protocols designed to date can be found in [9].

Of notable interest is also the problem of *verifiable secret shuffle* [10], which covers the permutation part of our protocol. However, it does not concern itself with the process of revealing the permuted elements to specific participants.

Structure of this thesis

In Chapter 1, we start by formalizing the goals and security requirements of our protocol. In Chapter 2, we then describe the necessary mathematical background and cryptographic primitives used in the protocol. The first part of our work is concluded by a complete, step-by-step description of the protocol run in Chapter 3.

As a second part of this work, we have implemented the permutation protocol as a Java library which can be used from any network application. Overview of our implementation is given in Chapter 4. In addition, we provide a simple example application based on our library.

Chapter 1

Goals

Our goal is to design a cryptographic protocol which would allow any group of people to cooperatively generate a random permutation of a given size, and then disclose individual elements of this permutation to selected participants. The elements need not be disclosed all at once and some might even stay undisclosed – the order of the participants and number of elements disclosed to each of them may be completely arbitrary.

In this chapter, we formally define the requirements of the protocol and give some reasonable assumptions which we will be allowed to make. These are designed so that the properties of the protocol are as close as possible to the analogous physical techniques, such as card shuffling or drawing names from a hat.

1.1 Protocol requirements

Let φ denote the number of participants. Before the protocol run, the participants decide on two parameters:

- π – the number of permutation elements (e.g. 52 if this protocol is used to shuffle a standard deck of cards, or φ if it is used to generate a permutation of the participants)
- ψ – the number of participants required to uncover an element of the permutation (for maximum security, ψ could be set equal to φ – however, in many card games, such as poker, it is common for participants to drop out of the game before it ends, and setting $\psi < \varphi$ would allow some of these participants to disconnect from the game server even though the protocol run is not finished)

We may assume that $\varphi \geq 2$, $\pi \geq 2$ and $2 \leq \psi \leq \varphi$, since otherwise the protocol is trivial.

The protocol run then consists of two phases:

1. In the first phase, an encrypted permutation of the set $\{1, \ldots, \pi\}$ is generated. Each participant must be allowed to provide input for this step, and the protocol must guarantee that the generated permutation is random as long as at least one participant's input is random. After this step is completed, no participant may be able to gain any information about the generated permutation. However, each participant must still be able to verify that the result is indeed an encryption of a permutation of $\{1, \ldots, \pi\}$ (ie. it does not contain any non-unique or invalid elements).

2. The second phase comprises several (no more than π) steps. In each step, the participants decide on a single participant P and provide this participant with enough information so that P can decrypt a single element of the generated permutation. The participant P must be able to verify that this is indeed a valid permutation element (ie. that it belongs to the set $\{1, \ldots, \pi\}$ and that the same element has not been uncovered by a different participant in one of the previous steps). Additionally, no other participants may gain any information about the permutation that they did not have before.

   Unlike the first phase, each step of the second phase only requires inputs from ψ of the φ participants (each step can be completed by a different set of ψ participants). No set of $\psi - 1$ or less participant must be able to gain any information about the permutation.

At an arbitrary time after the protocol completion, a participant P might want to prove to other participants that a specific permutation element was uncovered to P in a particular step. The information gained by the participants during the protocol run must be sufficient to achieve this.

1.2 Assumptions

Our protocol will operate under the following assumptions.

Assumption 1.1. *All communication between participants is public.*

Although in practice, the communication will most likely be implemented as a set of point-to-point communication channels between each pair of participants, we will not assume that these communication channels are secure. Under Assumption 1.1, a malicious participant may gain access to the communication channel between any two different participants. As a consequence, we will need to encrypt any messages that have the potential to break the protocol if they become public.

We may need to base the security of the encryption scheme on the assumption that a particular mathematical problem is intractable. All such assumptions will be listed in sections relevant to the specific cryptographic schemes.

Assumption 1.2. *There is no group of ψ or more dishonest participants cooperating with each other.*

Note that we do not guarantee the honesty of any particular participant. In fact, there can be as many as $k - 1$ dishonest participants cooperating with each other trying to break the protocol.

If ψ or more participants were secretly cooperating with each other, then (by the aforementioned requirements) they could just uncover all the permutation elements and thus render the whole cryptographic protocol meaningless. However, Assumption 1.2 can be made stronger by setting $\psi := \varphi$. In this case, we simply assume that at least one participant is honest, which is a natural and trivial assumption.

As a consequence of the lack of any assumptions about honesty of the specific participants, we must provide *verifiability* in each step of our protocol. Whenever there is an opportunity for some participant to cheat, this participant must provide a proof that they completed the step in question correctly. We may need to use zero-knowledge proofs in the protocol steps where secret information could be leaked otherwise, but we will also use simpler (but knowledge-leaking) proofs in other steps.

Chapter 2

Building blocks

2.1 Mathematical background

All of our cryptographic functions work on specific subsets of integers. In this section, we first provide some basic definitions and then proceed to show various important properties of these subsets.

Definition 2.1. Let $n \in \mathbb{Z}^+$. Then

$$\begin{aligned} \mathbb{Z}_n &:= \{0, 1, 2, \ldots, n-1\} \\ \mathbb{Z}_n^* &:= \{z \in \mathbb{Z}_n : \gcd(z, n) = 1\} \end{aligned}$$

It is easy to see that $|\mathbb{Z}_n| = n$. The size of $\mathbb{Z}_n^*$ is harder to determine, and is traditionally represented by the so-called *Euler's totient function.*

Definition 2.2. Let $n \in \mathbb{Z}^+$. *Euler's totient function* $\phi(n)$ is defined as the number of positive integers less than n that are relatively prime to n, ie.

$$\phi(n) := |\mathbb{Z}_n^*|$$

We will now derive values of $\phi(n)$ for some important special cases.

Lemma 2.3. *Let p be a prime number. Then*

$$\begin{aligned} \phi(p) &= p - 1 \\ \phi(p^2) &= p(p-1) \end{aligned}$$

Proof. The only integer from $\mathbb{Z}_p$ not relatively prime to p is 0. Therefore, $\phi(p) = |\mathbb{Z}_p| - 1 = p - 1$.

p^2 is relatively prime to all of $\mathbb{Z}_p$ except the multiples of p. There are exactly $\left\lfloor \frac{p^2}{p} \right\rfloor = p$ such multiples in $\mathbb{Z}_p$, and therefore $\phi(p^2) = |\mathbb{Z}_{p^2}| - p = p^2 - p = p(p-1)$. □

Lemma 2.4. *Let $n = pq$ be a product of two distinct prime numbers. Then*

$$\phi(n) = (p-1)(q-1)$$

Proof. Clearly, the only prime divisors of n are p and q. Any integer not relatively prime to n is therefore either a multiple of p or a multiple of q. Let P be the set of multiples of p from $\mathbb{Z}_n$ and Q be the set of multiples of q from $\mathbb{Z}_n$. We can now write $\mathbb{Z}_n^*$ as $\mathbb{Z}_n \setminus (P \cup Q)$, and since $P \cup Q \subseteq \mathbb{Z}_n$, it follows that $|\mathbb{Z}_n^*| = |\mathbb{Z}_n| - |P \cup Q|$.

$|P| = \left\lfloor \frac{n}{p} \right\rfloor = \left\lfloor \frac{pq}{p} \right\rfloor = q$ and similarly $|Q| = p$. Note that $P \cap Q = \{0\}$, since the second smallest natural number divisible by both p and q is $pq = n \notin \mathbb{Z}_n$. By the inclusion-exclusion principle, it follows that $|P \cup Q| = |P| + |Q| - |P \cap Q| = q + p - 1$.

Putting it all together:

$$\begin{aligned}
\phi(n) &= |\mathbb{Z}_n^*| \\
&= |\mathbb{Z}_n| - |P \cup Q| \\
&= n - (q + p - 1) \\
&= pq - q - p + 1 \\
&= (p-1)(q-1)
\end{aligned}$$

□

Lemma 2.5. *Let $n = pq$ be a product of two distinct prime numbers. Then*

$$\phi(n^2) = n\phi(n)$$

Proof. Again, p and q are the only prime divisors of n^2. Therefore, $|\mathbb{Z}_{n^2}^*| = |\mathbb{Z}_{n^2}| - |P \cup Q|$, where P is the set of multiples of p from $\mathbb{Z}_{n^2}$ and Q is the set of multiples of q from $\mathbb{Z}_{n^2}$.

$|P| = \left\lfloor \frac{n^2}{p} \right\rfloor = \left\lfloor \frac{p^2q^2}{p} \right\rfloor = pq^2$ and similarly $|Q| = p^2q$. $P \cap Q$ contains all multiples of $\mathrm{lcm}(p, q) = pq$ smaller than n^2, and therefore $|P \cap Q| = \left\lfloor \frac{n^2}{pq} \right\rfloor = \left\lfloor \frac{p^2q^2}{pq} \right\rfloor = pq$. By the inclusion-exclusion principle, it follows that $|P \cup Q| = |P| + |Q| - |P \cap Q| = pq^2 + p^2q - pq$.

$$\begin{aligned}
\phi(n^2) &= |\mathbb{Z}_{n^2}^*| \\
&= |\mathbb{Z}_{n^2}| - |P \cup Q| \\
&= p^2q^2 - (pq^2 + p^2q - pq) \\
&= pq(1 - q - p + pq) \\
&= pq(p-1)(q-1) \\
&= n\phi(n)
\end{aligned}$$

□

2.1.1 Congruences modulo $n = pq$ and n^2

Although some operations can be proven to work on $\mathbb{Z}_n$ or $\mathbb{Z}_n^*$ for any $n \in \mathbb{Z}^+$, we will usually require n to be of a specific form. The following definition gives an useful restriction that we will commonly use.

Definition 2.6. Let $n \in \mathbb{Z}^+$. We say that n is *admissible*, if n is a product of two distinct odd prime numbers p, q such that

$$p \nmid (q-1) \qquad \text{and} \qquad q \nmid (p-1)$$

Note that for all admissible $n = pq$, the numbers p, q, $p-1$ and $q-1$ are pairwise relatively prime. As a consequence, n is also relatively prime to $\phi(n)$ (see Lemma 2.4).

We now define the *congruence* relation and some of its basic properties.

Definition 2.7. Let $n \in \mathbb{Z}^+$, $a, b \in \mathbb{Z}$. We say that *a is congruent to b modulo n* if

$$n \mid (a-b)$$

We write

$$a = b \pmod{n}$$

Definition 2.8. Let $n \in \mathbb{Z}^+$. We say that x is a *solution of the congruence*

$$f(x) = 0 \pmod{n} \tag{$*$}$$

if $(*)$ holds and $x \in \mathbb{Z}_n$.

We will also sometimes use the notation

$$x := a \pmod{n} \qquad \text{for some } a \in \mathbb{Z}, n \in \mathbb{Z}^+$$

to denote that the variable x should be set to the value of the solution of the congruence

$$x = a \pmod{n}$$

In such cases, we will always make certain that this congruence has a unique solution.

Theorem 2.9. *Let $n \in \mathbb{Z}^+$, $a \in \mathbb{Z}$. If $\gcd(a, n) = 1$, then the congruence*

$$ax = 1 \pmod{n}$$

has exactly one solution. If $\gcd(a, n) \neq 1$, the congruence has no solutions.

Proof. Proven as Theorem 2.9 in [11]. □

Definition 2.10. Let $n \in \mathbb{Z}^+$, $a \in \mathbb{Z}_n^*$. By $a^{-1} \pmod{n}$, we will denote the unique solution of the congruence $ax = 1 \pmod{n}$. We call this number *the inverse of a modulo n*. We will also use the following notation:

$$\begin{aligned} \frac{a}{b} &:= ab^{-1} \pmod{n} \\ a^{-b} &:= (a^b)^{-1} = (a^{-1})^b \pmod{n} \end{aligned}$$

From basic number theory [11], we know that Definition 2.10 preserves the distributive properties of exponentiation, ie. for any $a, b \in \mathbb{Z}_n^*$, $c, d \in \mathbb{Z}$ and $n \in \mathbb{Z}^+$, it holds that

$$\begin{aligned} a^c b^c &= (ab)^c \pmod{n} \\ a^c a^d &= a^{c+d} \pmod{n} \\ (a^c)^d &= a^{cd} \pmod{n} \end{aligned}$$

It is also known that the set $\mathbb{Z}_n^*$ is closed under multiplication modulo n and, as a consequence, also closed under modular exponentiation.

Let us now prove the validity of some simple congruence relations.

Lemma 2.11. *Let $n, m \in \mathbb{Z}^+$, $a, b \in \mathbb{Z}$. Then if*

$$a = b \pmod{n} \qquad \textit{and} \qquad a = b \pmod{m}$$

it must also hold that

$$a = b \pmod{\operatorname{lcm}(n, m)}$$

Proof. From Definition 2.7:

$$n \mid (a - b) \qquad\qquad m \mid (a - b)$$

which immediately gives that $\operatorname{lcm}(n, m) \mid (a - b)$. □

Lemma 2.12. *Let $n, m \in \mathbb{Z}^+$, $a, b \in \mathbb{Z}$ and let $m \mid n$. Then*

$$a = b \pmod{n} \qquad \textit{implies} \qquad a = b \pmod{m}$$

Proof. From Definition 2.7:

$$\begin{aligned} & a = b \pmod{n} \\ \Rightarrow\ & n \mid (a - b) \\ \Rightarrow\ & m \mid (a - b) && (\text{since } m \mid n) \\ \Rightarrow\ & a = b \pmod{m} \end{aligned}$$

□

Corollary 2.13. *Let $n \in \mathbb{Z}^+$, $a \in \mathbb{Z}_n^*$ and let*

$$\bar{a} := a^{-1} \pmod{n^2}$$

Then

$$\bar{a} = a^{-1} \pmod{n}$$

Proof. First, note that by Theorem 2.9 both $a^{-1} \pmod{n}$ and $a^{-1} \pmod{n^2}$ are well-defined and unique, since $\gcd(a, n) = 1$ implies that $\gcd(a, n^2) = 1$. Now

$$\begin{aligned}
\bar{a} &= a^{-1} \pmod{n^2} \\
a\bar{a} &= 1 \pmod{n^2} \\
a\bar{a} &= 1 \pmod{n} && \text{(by Lemma 2.12)} \\
\bar{a} &= a^{-1} \pmod{n}
\end{aligned}$$

Note that we cannot directly apply Lemma 2.12 to the original congruence, since $a^{-1} \pmod{n}$ and $a^{-1} \pmod{n^2}$ might be two different numbers. □

Lemma 2.14. *Let $n \in \mathbb{Z}^+$, $a, b \in \mathbb{Z}$. Then*

$$a = b \pmod{n}$$

implies that

$$a^n = b^n \pmod{n^2}$$

Proof. Since $a = b \pmod{n}$, we can write a as $b + kn$ for some $k \in \mathbb{Z}$. Then, by the binomial theorem:

$$\begin{aligned}
a^n &= (b + kn)^n \\
&= \binom{n}{0} b^n + \binom{n}{1} b^{n-1} kn + \binom{n}{2} b^{n-2} (kn)^2 + \ldots + \binom{n}{n} (kn)^n \\
&= \binom{n}{0} b^n + \binom{n}{1} b^{n-1} kn \\
&\quad \text{(since all the remaining terms are multiples of } n^2\text{)} \\
&= 1b^n + nb^{n-1}kn \\
&= b^n + b^{n-1}kn^2 \\
&= b^n \pmod{n^2}
\end{aligned}$$

□

To show that the last two relations hold, we will require the following generalisation of Fermat's theorem.

Theorem 2.15 (Euler's theorem)**.** *Let $n \in \mathbb{Z}^+$, $a \in \mathbb{Z}_n^*$. Then*

$$a^{\phi(n)} = 1 \pmod{n}$$

Proof. Proven as Theorem 2.8 in [11]. □

Lemma 2.16. *Let $n = pq$ be admissible and let*

$$\lambda := \operatorname{lcm}(p-1, q-1)$$

Then for all $a \in \mathbb{Z}_n^*$ it holds that

$$a^{n\lambda} = 1 \pmod{n^2}$$

Proof. Since $(p-1) \mid \lambda$, we can write $\lambda = k(p-1)$ for some $k \in \mathbb{Z}$. Then

$$\begin{aligned} a^{n\lambda} &= a^{pqk(p-1)} \\ &= (a^{p(p-1)})^{qk} \\ &= (a^{\phi(p^2)})^{qk} \\ &= 1^{qk} && \text{(by Theorem 2.15)} \\ &= 1 \pmod{p^2} \end{aligned}$$

Similarly, $a^{n\lambda} = 1 \pmod{q^2}$. Since $\operatorname{lcm}(p^2, q^2) = p^2q^2 = n^2$, by Lemma 2.11 it follows that

$$a^{n\lambda} = 1 \pmod{n^2}$$

□

Lemma 2.17. *Let $n = pq$ be admissible and let*

$$\bar{n} := n^{-1} \pmod{\phi(n)}$$

Then for all $a \in \mathbb{Z}_n^*$ it holds that

$$a^{n\bar{n}} = a \pmod{n}$$

Proof. First, note that $\bar{n}$ is well-defined, since the admissibility of n guarantees that $\gcd(n, \phi(n)) = 1$.

Since $n\bar{n} = 1 \pmod{\phi(n)}$, it follows that $\phi(n) \mid (n\bar{n} - 1)$ and therefore we can write

$$n\bar{n} - 1 = k\phi(n) \qquad \text{for some } k \in \mathbb{Z}$$
$$n\bar{n} = k\phi(n) + 1$$

Then

$$\begin{aligned} a^{n\bar{n}} &= aa^{n\bar{n}-1} \\ &= aa^{k\phi(n)+1-1} \\ &= a(a^{\phi(n)})^k \\ &= a1^k && \text{(by Theorem 2.15)} \\ &= a \pmod{n} \end{aligned}$$

□

2.1.2 Powers of $1 + n \pmod{n^2}$

Our encryption scheme, described in section 2.2, will be based on powers of $1 + n$ modulo n^2. In this section, we explore the basic form of these powers and, most importantly, show how this exponentiation operation can be reversed (ie. how to compute discrete logarithms with respect to the base $1 + n$ modulo n^2).

Lemma 2.18. *Let $n \in \mathbb{Z}^+$, $a \in \mathbb{N}$. Then*

$$(1 + n)^a = 1 + an \pmod{n^2}$$

Proof. By induction over a:

- For $a = 0$:
$$(1 + n)^0 = 1 = 1 + 0n \pmod{n^2}$$
- Let $(1 + n)^a = 1 + an \pmod{n^2}$ for some a. For $a + 1$, it then follows:
$$\begin{aligned} (1 + n)^{a+1} &= (1 + n)^a(1 + n) \\ &= (1 + an)(1 + n) \\ &= 1 + n + an + an^2 \\ &= 1 + n + an + 0 \\ &= 1 + (a + 1)n \pmod{n^2} \end{aligned}$$

□

This regular form of the powers of $1+n$ modulo n^2 allows us to compute discrete logarithms to the base $1+n$ modulo n^2 easily.

Definition 2.19. Let $n \in \mathbb{Z}^+$. We define the function

$$\mathrm{L}_n : \{1 + kn : k \in \mathbb{Z}_n\} \to \mathbb{Z}_n$$

such that

$$(1+n)^{\mathrm{L}_n(a)} = a \pmod{n^2} \qquad \text{for all } a \in \{1 + kn : k \in \mathbb{Z}_n\}$$

Note that this function is well-defined, since for each $a \in \mathbb{Z}_n$ the result of $(1+n)^a = 1 + an \pmod{n^2}$ is different.

Corollary 2.20. *Let $n \in \mathbb{Z}^+$. Then for all $a \in \{1 + kn : k \in \mathbb{Z}_n\}$ it holds that*

$$\mathrm{L}_n(a) = \frac{a-1}{n}$$

Proof. The formula follows directly from Lemma 2.18. Note that this is not a modular division, but a regular division over $\mathbb{Z}$. The form of a guarantees that $n \mid (a-1)$. □

Lemma 2.21. *Let $n \in \mathbb{Z}^+$, $a, b \in \mathbb{N}$. Then*

$$(1+n)^a = (1+n)^b \pmod{n^2}$$

if and only if

$$a = b \pmod{n}$$

Proof. The following relations are equivalent:

$$\begin{aligned}
(1+n)^a &= (1+n)^b \pmod{n^2} \\
1 + an &= 1 + bn \pmod{n^2} && \text{(by Lemma 2.18)} \\
an &= bn \pmod{n^2} \\
n^2 &\mid (an - bn) \\
n &\mid (a-b) \\
a &= b \pmod{n}
\end{aligned}$$

□

2.1.3 Injectiveness of $x^n \pmod{n^2}$

Raising elements of $\mathbb{Z}_n^*$ to the n-th power modulo n^2 is another operation that we will commonly use in our encryption scheme. In this section, we show that for an admissible n, this operation is also uniquely reversible.

We start by studying the number of solutions of the congruence $x^n = 1 \pmod{n^2}$ for an admissible n. It will then follow that the function $x^n \pmod{n^2}$ is an injection from $\mathbb{Z}_n^*$ to $\mathbb{Z}_{n^2}^*$ (Lemma 2.27).

First, we reference two known number-theoretical results.

Theorem 2.22. *Let $n, m \in \mathbb{Z}^+$, $\gcd(n, m) = 1$ and let $p(x)$ be a polynomial with integer coefficients. Let N denote the number of solutions of the congruence*

$$p(x) = 0 \pmod{n}$$

and let M denote the number of solutions of the congruence

$$p(x) = 0 \pmod{m}$$

Then the congruence

$$p(x) = 0 \pmod{nm}$$

has exactly NM solutions.

Proof. This is a corollary to the Chinese Reminder Theorem. It is proven as Theorem 2.20 in [11]. □

Theorem 2.23. *Let p be an odd prime number and $a, b \in \mathbb{Z}^+$. Then the congruence*

$$x^a = 1 \pmod{p^b}$$

has exactly $\gcd(a, \phi(p^b))$ solutions.

Proof. This is a special case of Corollary 2.42 in [11]. □

Lemma 2.24. *Let $n = pq$ be admissible. Then the congruence $x^n = 1 \pmod{n^2}$ has exactly n solutions.*

Proof. By Theorem 2.23, the congruence $x^n = 1 \pmod{p^2}$ has $\gcd(n, \phi(p^2))$ solutions. From Lemma 2.3, we get

$$\begin{aligned} \gcd(n, \phi(p^2)) &= \gcd(p^2q^2, p(p-1)) \\ &= p \qquad \text{(since } q \text{ and } p-1 \text{ are relatively prime)} \end{aligned}$$

Similarly, the congruence $x^n = 1 \pmod{q^2}$ has q solutions. By Theorem 2.22, the congruence $x^n = 1 \pmod{p^2q^2}$ then has $pq = n$ solutions. □

Lemma 2.25. *Let $n = pq$ be admissible. Then the solutions of the congruence $x^n = 1 \pmod{n^2}$ are $\{1 + kn : k \in \mathbb{Z}_n\}$.*

Proof. By Lemma 2.18, it holds for all $k \in \mathbb{Z}_n \subseteq \mathbb{N}$ that

$$(1 + kn)^n = 1 + nkn = 1 \pmod{n^2}$$

By Lemma 2.24, there can be no more than these n solutions. □

Corollary 2.26. *Let $n = pq$ be admissible. Then the congruence $x^n = 1 \pmod{n^2}$ has exactly one solution in $\mathbb{Z}_n$: the number 1.*

Proof. All other solutions given by Lemma 2.25 are clearly larger than n. □

Lemma 2.27. *Let $n = pq$ be admissible. Then the function $f : \mathbb{Z}_n^* \to \mathbb{Z}_{n^2}^*$ defined as*

$$f(x) := x^n \pmod{n^2}$$

is injective.

Proof. Let $x_1, x_2 \in \mathbb{Z}_n^*$ such that $x_1^n = x_2^n \pmod{n^2}$. Then by Lemma 2.12 it also holds that

$$\begin{aligned}
x_1^n &= x_2^n \pmod{n} \\
\frac{x_1^n}{x_2^n} &= 1 \pmod{n} \\
\left(\frac{x_1}{x_2}\right)^n &= 1 \pmod{n} \\
\frac{x_1}{x_2} &= 1 \pmod{n} && \text{(by Corollary 2.26)} \\
x_1 &= x_2 \pmod{n}
\end{aligned}$$

And since $x_1, x_2 \in \mathbb{Z}_n^*$, it must hold that $x_1 = x_2$. □

2.2 Encryption scheme

Since (by Assumption 1.1) all communication between participants is publicly readable, we need to employ an encryption scheme to guarantee that no data is read by

anyone other than the intended recipient. However, to be able to successfully complete our protocol, we will require this encryption scheme to satisfy some additional requirements.

In this section, we first describe these requirements in more detail, and then proceed to describe a specific cryptosystem that satisfies them.

2.2.1 Requirements

- We require a *public-key* cryptosystem: At the beginning of the protocol run, each participant will generate a pair of keys – a private key which they will keep secret, and a public key which will be released to all other participants. The public key will then allow anyone to generate encrypted messages that will only be readable by the owner of the respective private key.

- The encryption function must be *probablistic*, ie. it must be a function of two arguments: the secret message, and an additional large random value. This guarantees that the set of possible ciphertexts is much larger than the set of all possible plaintexts, and therefore the probability of encrypting the same message to the same ciphertext twice is negligible. If the encryption function were deterministic (the same plaintext would always encrypt to the same ciphertext), any ciphertext could be decrypted by simply encrypting all possible values and comparing the result to the original ciphertext. In our case, such an attack would be devastating, since in most cases, the set of possible plaintexts is small (on the order of the permutation size).

- The function must satisfy the following *homomorphic property*:

$$E(x + y) = E(x) \star E(y)$$

  where $\star$ denotes any operation that can be easily computed (such as modular multiplication or exponentiation). This will allow us to perform the addition operation on arbitrary ciphertexts without decrypting them. Our protocol will use this homomorphic property extensively, both for 'masking' ciphertexts (so that no one but the intended recipient is able to tell whether two ciphertexts encrypt the same value), and for participants to prove their honesty without revealing their secrets (see section 2.5).

2.2.2 Paillier cryptosystem

Several well-known public-key cryptosystems (RSA, ElGamal) have a homomorphic property. However, the homomorphic property of both RSA and ElGamal is multiplicative and not additive as specified by our requirements (furthermore, RSA is not probablistic without additional modifications). We have therefore decided to reach for a cryptosystem based on a different mathematical problem.

In [12], Goldwasser and Micali propose a probablistic cryptosystem based on the *quadratic residuosity problem*. This cryptosystem satisfies a homomorphic property

$$E(x)E(y) = E(z) \qquad \text{where } z := x + y \pmod 2$$

Although this makes it additively homomorphic as required, the small modulus 2 makes it insufficient for our purposes. The Goldwasser–Micali cryptosystem is also known to be inefficient, since it only allows for encryption of a single bit at a time.

However, this cryptosystem spawned several other schemes based on the higher-degree *composite residuosity problem*, which overcome the inefficiency problems, as well as satisfy an additive homomorphic property with higher modulus. For a good overview of some of these cryptosystems, see [13] or [14]. There have also been some specialized schemes proposed for the various protocols, such as in [7] which our protocol is based on.

We have, however, decided not to use such a specialized encryption scheme, but instead base our work on a well-known and tested cryptosystem. We have therefore chosen the scheme proposed by Pascal Paillier in [15], which is efficient and has a very regular mathematical structure. It has been an object of extensive research since its publication (see e.g. [16], [14]) and has not yet fallen to any cryptanalytic attacks.

In the following sections, we describe a simplified version of this cryptosystem which satisfies all of our requirements. The composite residuosity problem, which the scheme's security is based on, is then formally defined and discussed in section 2.2.6.

2.2.3 Encryption function

In [15], Paillier defines the encryption function as

$$\varepsilon_g(x, y) := g^x y^n \pmod{n^2}$$

where the values of n and g fulfil the role of the public key, and the factorization of n is kept secret as the private key.

Paillier then proceeds to show that this function is a bijection from $\mathbb{Z}_n \times \mathbb{Z}_n^*$ to $\mathbb{Z}_{n^2}^*$, as long as g and n satisfy certain conditions. As suggested in [16], we will set $g := 1 + n$ instead of using a variable g (the public key will therefore consist of the single number n). This simplifies the cryptosystem considerably and also allows for a more efficient implementation. As can be seen by Theorem 2.29, this choice of g is valid, as long as n is of a particular form.

Definition 2.28. Let $n \in \mathbb{Z}^+$. We define the *Paillier encryption function*

$$\mathrm{E}_n : \mathbb{Z}_n \times \mathbb{Z}_n^* \to \mathbb{Z}_{n^2}^*$$

as

$$\mathrm{E}_n(x, y) := (1 + n)^x y^n \pmod{n^2}$$

Note that by Lemma 2.18

$$\mathrm{E}_n(x, y) := (1 + xn) y^n \pmod{n^2}$$

The properties of this function now depend solely on the choice of the public key n. For our purposes, it is above all required that the encryption function is invertible, ie. for any ciphertext c, there must be a unique message x such that $\mathrm{E}_n(x, y) = c$ for some y. This property is covered by the following theorem.

Theorem 2.29. *Let n be admissible (according to Definition 2.6). Then E_n is a bijection.*

Proof. By Lemma 2.5, the number of elements $|\mathbb{Z}_n \times \mathbb{Z}_n^*| = n\phi(n)$ is equal to the number of elements of $\mathbb{Z}_{n^2}^*$. It is therefore sufficient to prove that E_n is an injection.

Let $(x_1, y_1), (x_2, y_2) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ such that $\mathrm{E}_n(x_1, y_1) = \mathrm{E}_n(x_2, y_2)$ and let $\lambda := \mathrm{lcm}(p - 1, q - 1)$.

$$\begin{aligned}
(1+n)^{x_1} y_1^n &= (1+n)^{x_2} y_2^n \pmod{n^2} \\
(1+n)^{x_1\lambda} y_1^{n\lambda} &= (1+n)^{x_2\lambda} y_2^{n\lambda} \pmod{n^2} \\
(1+n)^{x_1\lambda} &= (1+n)^{x_2\lambda} \pmod{n^2} && \text{(by Lemma 2.16)} \\
(1+n)^{(x_1-x_2)\lambda} &= 1 \pmod{n^2} \\
1 + (x_1 - x_2)\lambda n &= 1 \pmod{n^2} && \text{(by Lemma 2.18)} \\
(x_1 - x_2)\lambda n &= 0 \pmod{n^2} \\
n^2 &\mid (x_1 - x_2)\lambda n
\end{aligned}$$

$$
\begin{aligned}
& n \mid (x_1 - x_2)\lambda \\
& n \mid (x_1 - x_2) && \text{(since } \gcd(\lambda, n) = 1\text{)} \\
& x_1 = x_2 \pmod{n}
\end{aligned}
$$

and since $x_1, x_2 \in \mathbb{Z}_n$, this implies that $x_1 = x_2$. Substituting $x_1 = x_2 = x$ in the original equation, we get

$$
\begin{aligned}
(1+n)^x y_1^n &= (1+n)^x y_2^n \pmod{n^2} \\
y_1^n &= y_2^n \pmod{n^2} \\
y_1 &= y_2 \pmod{n} && \text{(by Lemma 2.27)}
\end{aligned}
$$

□

We have shown that choosing an admissible n as their public key provides each participant with a valid encryption function.

However, we cannot rely on the honesty of the participants. As described in section 2.5.2, a dishonest participant might in some cases gain an advantage by intentionally choosing a public key that would allow for non-unique decryption of some messages. The following theorem will give us a simple way to detect such attempts.

Theorem 2.30. *Let $n \in \mathbb{Z}^+$ such that E_n is not an injective function. Then for all $(x, y) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ there exists $(x', y') \in \mathbb{Z}_n \times \mathbb{Z}_n^*$, $(x', y') \neq (x, y)$ such that $\mathrm{E}_n(x', y') = \mathrm{E}_n(x, y)$.*

Proof. Since E_n is not an injective function, there exists $(x_1, y_1), (x_2, y_2) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ such that $(x_1, y_1) \neq (x_2, y_2)$ and $\mathrm{E}_n(x_1, y_1) = \mathrm{E}_n(x_2, y_2)$. Let us take one such x_1, y_1, x_2, y_2. Without loss of generality, we will assume that $x_1 \geq x_2$.

$$
\begin{aligned}
\mathrm{E}_n(x_1, y_1) &= \mathrm{E}_n(x_2, y_2) \\
(1+n)^{x_1} y_1^n &= (1+n)^{x_2} y_2^n \pmod{n^2} \\
(1+n)^{x_1 - x_2} \left(\frac{y_1}{y_2}\right)^n &= 1 \pmod{n^2} \qquad (1)
\end{aligned}
$$

Note that $y_2 \in \mathbb{Z}_n^*$, which implies that $\gcd(y_2, n) = 1$ and therefore $\gcd(y_2, n^2) = 1$, so y_2^{-1} is well-defined.

For the given (x, y), let us now set

$$
\begin{aligned}
& x'' := x + x_1 - x_2 \\
& y'' := y\frac{y_1}{y_2} = y y_1 \bar{\bar{y}}_2 \pmod{n^2} && \text{where } \bar{\bar{y}}_2 := y_2^{-1} \pmod{n^2}
\end{aligned}
$$

and

$$\begin{aligned} x' &:= x'' \pmod{n} \\ y' &:= y\frac{y_1}{y_2} = yy_1\bar{y}_2 \pmod{n} \qquad \text{where } \bar{y}_2 := y_2^{-1} \pmod{n} \end{aligned}$$

Then by Lemma 2.21

$$(1+n)^{x'} = (1+n)^{x''} \pmod{n^2} \tag{2}$$

Furthermore, by Corollary 2.13

$$\bar{\bar{y}}_2 = \bar{y}_2 \pmod{n}$$

and therefore by Lemma 2.14

$$\bar{\bar{y}}_2^n = \bar{y}_2^n \pmod{n^2}$$

Then

$$\begin{aligned} \left(y'\right)^n &= (yy_1\bar{y}_2)^n \\ &= y^n y_1^n \bar{y}_2^n \\ &= y^n y_1^n \bar{\bar{y}}_2^n \\ &= \left(y''\right)^n \pmod{n^2} \end{aligned} \tag{3}$$

We can now show that $\mathrm{E}_n(x', y') = \mathrm{E}_n(x, y)$:

$$\begin{aligned} \mathrm{E}_n(x', y') &= (1+n)^{x'}\left(y'\right)^n \\ &= (1+n)^{x''}\left(y''\right)^n && \text{(from (2) and (3))} \\ &= (1+n)^{x+x_1-x_2}\left(y\frac{y_1}{y_2}\right)^n \\ &= (1+n)^x y^n (1+n)^{x_1-x_2}\left(\frac{y_1}{y_2}\right)^n \\ &= \mathrm{E}_n(x, y)1 && \text{(from (1))} \\ &= \mathrm{E}_n(x, y) \pmod{n^2} \end{aligned}$$

Note that if $x_1 \neq x_2$, then $x_1 - x_2 \in \{1, 2, \ldots, n-1\}$ and therefore $x' \neq x$. Similarly, if $y_1 \neq y_2$, then $\frac{y_1}{y_2} \neq 1 \pmod{n}$ and therefore $y' \neq y$. Since either $x_1 \neq x_2$ or $y_1 \neq y_2$, this implies that $(x', y') \neq (x, y)$. □

2.2.4 Decryption function

Definition 2.31. Let n be admissible. We define a pair of *Paillier decryption functions*

$$\mathrm{D}_n : \mathbb{Z}_{n^2}^* \to \mathbb{Z}_n$$
$$\mathrm{D}'_n : \mathbb{Z}_{n^2}^* \to \mathbb{Z}_n^*$$

as follows:

$$\mathrm{D}_n(c) = x$$
$$\text{and } \mathrm{D}'_n(c) = y$$

if there exists $(x, y) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ such that $\mathrm{E}_n(x, y) = c$.

By Theorem 2.29, the admissibility of n guarantees that both decryption functions are well-defined. However, Theorem 2.29 alone does not lead to any efficient way of computing $\mathrm{D}_n(c)$ and $\mathrm{D}'_n(c)$.

The following theorem gives a modified version of Paillier's original formula for D_n [15], which is suitable for our simplified version of the scheme.

Theorem 2.32. *Let $n = pq$ be admissible. Then for all $c \in \mathbb{Z}_{n^2}^*$, $\mathrm{D}_n(c)$ can be computed as follows:*

$$\begin{aligned} \lambda &:= \operatorname{lcm}(p-1, q-1) \\ \hat{c} &:= c^\lambda \pmod{n^2} \\ \mathrm{D}_n(c) &:= \frac{\mathrm{L}_n(\hat{c})}{\lambda} \pmod{n} && \textit{(see Definition 2.19)} \end{aligned}$$

Proof. Since n is admissible, there exists a unique $(x, y) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ such that $\mathrm{E}_n(x, y) = c$. Then

$$\begin{aligned} (1+n)^x y^n &= c \pmod{n^2} \\ (1+n)^{x\lambda} y^{n\lambda} &= \hat{c} \pmod{n^2} \\ (1+n)^{x\lambda} &= \hat{c} \pmod{n^2} && \text{(by Lemma 2.16)} \\ (1+n)^{x\lambda} &= (1+n)^{\mathrm{L}_n(\hat{c})} \pmod{n^2} && \text{(by Definition 2.19)} \\ x\lambda &= \mathrm{L}_n(\hat{c}) \pmod{n} && \text{(by Lemma 2.21)} \\ x &= \frac{\mathrm{L}_n(\hat{c})}{\lambda} \pmod{n} \\ \mathrm{D}_n(c) &= \frac{\mathrm{L}_n(\hat{c})}{\lambda} \pmod{n} \end{aligned}$$

And since $\mathrm{D}_n(c) \in \mathbb{Z}_n$, it must indeed be equal to the (unique) solution of this congruence. Note that $\gcd(n, \lambda) = 1$ is guaranteed by the admissibility of n. □

Although the second argument to the encryption function will never carry any useful data, we will sometimes need to decrypt it to provide verifiability in our protocol.

Theorem 2.33. *Let n be admissible. Then for all $c \in \mathbb{Z}_{n^2}^*$, $\mathrm{D}'_n(c)$ can be computed as follows:*

$$\bar{n} := n^{-1} \pmod{\phi(n)}$$
$$\mathrm{D}'_n(c) := c^{\bar{n}} \pmod{n}$$

Proof. Since n is admissible, there exists a unique $(x, y) \in \mathbb{Z}_n \times \mathbb{Z}_n^*$ such that $\mathrm{E}_n(x, y) = c$. Then

$$\begin{aligned} (1+n)^x y^n &= c \pmod{n^2} \\ (1+n)^x y^n &= c \pmod{n} && \text{(by Lemma 2.12)} \end{aligned}$$

Note that $1 + n = 1 \pmod{n}$:

$$\begin{aligned} 1^x y^n &= c \pmod{n} \\ y^n &= c \pmod{n} \\ y^{n\bar{n}} &= c^{\bar{n}} \pmod{n} \\ y &= c^{\bar{n}} \pmod{n} && \text{(by Lemma 2.17)} \\ \mathrm{D}'_n(c) &= c^{\bar{n}} \pmod{n} \end{aligned}$$

And since $\mathrm{D}'_n(c) \in \mathbb{Z}_n^*$, it must indeed be equal to the solution of this congruence. □

2.2.5 Homomorphic property

It is easy to see that Paillier cryptosystem satisfies the required homomorphic property: Multiplication of ciphertexts results in an encrypted sum of the respective plaintexts. Additionally, as can be seen by the following theorem, the encryption function satisfies a similar homomorphic property for the second argument.

Theorem 2.34. *Let n be admissible, $x_1, x_2 \in \mathbb{Z}_n$ and $y_1, y_2 \in \mathbb{Z}_n^*$. Let*

$$x := x_1 + x_2 \pmod{n}$$
$$y := y_1 y_2 \pmod{n}$$

Then

$$\mathrm{E}_n(x_1, y_1)\,\mathrm{E}_n(x_2, y_2) = \mathrm{E}_n(x, y) \pmod{n^2}$$

Proof.

$$\begin{aligned}
\mathrm{E}_n(x_1, y_1)\,\mathrm{E}_n(x_2, y_2) &= (1+n)^{x_1} y_1^n (1+n)^{x_2} y_2^n \\
&= (1+n)^{x_1+x_2} (y_1 y_2)^n \\
&= (1+n)^{x} (y_1 y_2)^n && \text{(by Lemma 2.21)} \\
&= (1+n)^{x} y^n && \text{(by Lemma 2.14)} \\
&= \mathrm{E}_n(x, y) \pmod{n^2}
\end{aligned}$$

□

2.2.6 Security

As discussed before, the security of Paillier cryptosystem depends on the intractability of the *composite residuosity problem*. We prove this result formally in this section.

Definition 2.35. Let $n \in \mathbb{Z}^+$. We say that $c \in \mathbb{Z}_{n^2}$ is an *n-th residue modulo n^2*, if there exists $y \in \mathbb{Z}_{n^2}$ such that

$$c = y^n \pmod{n^2}$$

Otherwise, we say that c is an *n-th non-residue modulo n^2*.

Definition 2.36. We define the *composite residuosity problem* as follows:

Given an admissible n and $c \in \mathbb{Z}^*_{n^2}$, determine whether c is an n-th residue modulo n^2.

The security of Paillier cryptosystem relies on the following assumption.

Assumption 2.37. *The composite residuosity problem is intractable (there exists no polynomial-time algorithm that solves this problem).*

We will show that the cryptosystem is secure as long as this assumption holds, by reducing the composite residuosity problem to the problem of computing D_n in polynomial time. We will need the following lemma.

Lemma 2.38. *Let n be admissible and $c \in \mathbb{Z}^*_{n^2}$. Then c is an n-th residue modulo n^2 if and only if*

$$\mathrm{D}_n(c) = 0$$

Proof. ($\Rightarrow$) Since c is an n-th residue modulo n^2, there exists $y \in \mathbb{Z}_{n^2}^*$ such that

$$y^n = c \pmod{n^2}$$

Let us set $y' := y \pmod{n}$. Then by Lemma 2.14

$$\begin{aligned} \left(y'\right)^n = y^n = c \pmod{n^2} \\ \mathrm{E}_n(0, y') = (1+n)^0 \left(y'\right)^n = c \pmod{n^2} \end{aligned}$$

and therefore $\mathrm{D}_n(c) = 0$.

($\Leftarrow$) Since $\mathrm{D}_n(c) = 0$, there exists $y \in \mathbb{Z}_n^* \subseteq \mathbb{Z}_{n^2}^*$ such that

$$\begin{aligned} \mathrm{E}_n(0, y) = c \pmod{n^2} \\ (1+n)^0 y^n = c \pmod{n^2} \\ y^n = c \pmod{n^2} \end{aligned}$$

and therefore c is an n-th residue modulo n^2. □

Theorem 2.39. *Let n be admissible. Then the problem of computing D_n is intractable.*

Proof. For the purpose of contradiction, let us assume that the problem is tractable, ie. we have a polynomial-time algorithm to compute D_n. Then, given $c \in \mathbb{Z}_{n^2}^*$, we can use the following polynomial-time algorithm to determine if c is an n-th residue modulo n^2:

1. Use the polynomial-time algorithm for D_n to compute $\mathrm{D}_n(c)$.
2. If the algorithm returned $\mathrm{D}_n(c) = 0$, return *'residue'*. Otherwise return *'non-residue'*.

The correctness of this algorithm follows from Lemma 2.38. Since by Assumption 2.37 there is no polynomial-time algorithm to solve the composite residuosity problem, this is a contradiction. □

Note that the decryption function can be efficiently computed if the attacker can factor the public key n or if they can compute discrete logarithms modulo n^2 efficiently. The problem of computing D_n therefore cannot be harder then any of these problems. However, no equivalence has been proven yet. Some additional reductions and further discussion is given by Paillier in [15].

2.3 Using the encryption scheme

2.3.1 Notation

Although we have used E_n, D_n and D'_n with n denoting the value of the specific public key throughout this section, this will not be practical in later sections when we are not discussing the internals of the cryptosystem. In further sections, we will therefore use the notation $\mathrm{E}_i, \mathrm{D}_i, \mathrm{D}'_i$ to denote encryption and decryption using the i-th participant's public key.

That is, if there are φ participants with public keys $n_1, \ldots, n_\varphi$, we will use

$$\mathrm{E}_{n_i}, \mathrm{D}_{n_i}, \mathrm{D}'_{n_i}$$

and

$$\mathrm{E}_i, \mathrm{D}_i, \mathrm{D}'_i$$

interchangeably. We will make sure that the context makes it clear which notation is used.

Since the second argument to the function E_i never carries any useful data (it only provides the encrypted function with the required probablistic property), we will also sometimes use the notation

$$\mathrm{E}_i(x) \quad \text{or} \quad \mathrm{E}_{n_i}(x)$$

to denote that the second argument is irrelevant at this specific point (ie. it should be chosen randomly from all applicable values).

2.3.2 Encrypting large values

By Definition 2.28, it is only possible to encrypt messages from $\mathbb{Z}_n$ with a given public key n. Sometimes, however, we will need to send a message that is not guaranteed to belong in this set.

In such cases, we will use the following procedure to encrypt the (possibly) large value x:

ENCRYPT-LARGE(n, x)
  $c \leftarrow 0$
  repeat $c \leftarrow n^2 c + \mathrm{E}_n(x \bmod n)$
    $x \leftarrow x \operatorname{div} n$
  until $x = 0$
  return c

This function splits x into its base-n digits (each of which therefore belongs to $\mathbb{Z}_n$), encrypts each digit separately and the resulting encrypted values (which belong to $\mathbb{Z}_{n^2}$) are combined into a single number c by taking each encrypted value as one base-n^2 digit of c.

Decryption function is then analoguous:

DECRYPT-LARGE(n, c)

$x \leftarrow 0$
repeat $x \leftarrow nx + \mathrm{D}_n(c \bmod n^2)$
$c \leftarrow c \operatorname{div} n^2$
until $c = 0$
return x

In our protocol, we should never need to send messages larger than approximately $2n$, and thus no message should ever be split into more than two 'digits'.

2.4 Threshold scheme

The requirement that each permutation element must be decryptable by any ψ of the φ participants naturally calls for a *threshold secret sharing scheme*. One such threshold scheme, which is well-known and good for our purposes, is described by Shamir in [17].

In Shamir's scheme, a secret value x is split into φ values $x_1, \ldots, x_\varphi$ such that any ψ of these values are both sufficient and necessary to compute the original value of x. The secret shares $x_1, \ldots, x_\varphi$ are created as follows:

1. A prime number p is chosen such that $p > \varphi$ and $p > x$.
2. $\psi - 1$ random numbers $p_1, \ldots, p_{\psi-1} \in \mathbb{Z}_p$ are generated.
3. The φ shares are computed as follows:

$$x_i := x + p_1 i + p_2 i^2 + \ldots + p_{\psi-1} i^{\psi-1} \pmod{p}$$

From number theory [11], we know the following result.

Lemma 2.40. *Let p be a prime number. Then $(\mathbb{Z}_p, +, \cdot)$, where both binary operations are modulo p, is a field.*

Let P be a polynomial of degree $\psi - 1$ with coefficients $(x, p_1, \ldots, p_{\psi-1})$ over this field. It is easy to see that each share x_i is simply the value of P evaluated at the point i. Additionally, evaluating this polynomial at 0 gives us the value of x:

$$P(0) = x + p_1 0 + p_2 0^2 + \ldots p_{\psi-1} 0^{\psi-1} = x$$

Every polynomial of degree $\psi - 1$ is uniquely determined by any ψ of its points, which implies that this scheme satisfies the required threshold property.

Furthermore, if less than ψ shares are known, then for each possible $x \in \mathbb{Z}_p$ there exists at least one polynomial Q of degree $\psi - 1$ such that $Q(i) = x_i$ for all known shares x_i and $Q(0) = x$, and an attacker has no way of knowing which of these polynomials is the secret polynomial P and therefore has absolutely no information about which of the possible $x \in \mathbb{Z}_p$ is the correct secret value.

2.4.1 Generating the shares

It is usually assumed that a trusted third party prepares the shares $x_1, \ldots, x_\varphi$ and then secretly distributes them to the respective participants. Neither of these assumptions is applicable to our protocol, as we do not have access to a trusted third party nor a secret communication channel.

We will solve this problem by generating the shares cooperatively by all the participants:

1. First, a trivial set of coefficients $(x, 0, \ldots, 0)$ is used to generate the φ secret shares. This results in φ equal shares:

   $$(x, \ldots, x)$$

2. The participants now take turns to modify the set of shares in the following way: Let

   $$(x_1, \ldots, x_\varphi)$$

   be the original set of shares. The participant generates $\psi - 1$ random numbers $r_1, \ldots, r_{\psi-1} \in \mathbb{Z}_p$ and computes for all $i \in \{1, \ldots, \varphi\}$:

   $$y_i = 0 + r_1 i + r_2 i^2 + \ldots + r_{\psi-1} i^{\psi-1} \pmod{p}$$

   The new secret shares are then computed as follows:

   $$(x_1 + y_1, \ldots, x_\varphi + y_\varphi)$$

It is easy to see that $(0, r_1, \ldots, r_{\psi-1})$ form the coefficients of a polynomial R of degree $\psi - 1$ with $R(0) = 0$ and $R(i) = y_i$ for all $i \in \{1, \ldots, \varphi\}$. If P denotes the polynomial defined by the original shares, then the new set of shares defines a polynomial Q with degree $\psi - 1$ such that $Q(0) = P(0) + R(0) = x + 0 = x$ and therefore the new shares encode the same secret value.

2.4.2 Keeping the shares secret

The proposed scheme does not satisfy Assumption 1.1, since if all communication is public, then no amount of modification of the shares prevents a participant from simply accessing all the shares and computing the secret value.

To prevent a participant from reading other participants' secret shares, each share must be encrypted by the respective participant's public key. Instead of

$$(x_1, \ldots, x_\varphi)$$

we must therefore work with the tuple

$$(\mathrm{E}_1(x_1), \ldots, \mathrm{E}_\varphi(x_\varphi))$$

It might seem that encrypting the values would prevent us from performing the computations described in the previous section. However, this is where the homomorphic property of Paillier cryptosystem comes into play. Instead of adding the computed value y_i to the original value of the share x_i, we will encrypt y_i using the same public key and then simply multiply the encrypted values. The resulting set of shares can thus be computed as follows:

$$(\mathrm{E}_1(x_1)\,\mathrm{E}_1(y_1), \ldots, \mathrm{E}_\varphi(x_\varphi)\,\mathrm{E}_\varphi(y_\varphi))$$

Using the homomorphic property of Paillier cryptosystem, it follows that

$$\mathrm{D}_i(\mathrm{E}_i(x_i)\,\mathrm{E}_i(y_i)) = x_i + y_i \pmod{i\text{-th participant's public key}}$$

which satisfies our requirements as long as $x_i + y_i$ does not exceed the value of any of the participants' public keys.

2.4.3 Recovering the secret value

Given ψ secret shares

$$x_{i_1}, \ldots, x_{i_\psi} \qquad (i_j \in \{1, \ldots, \varphi\} \text{ for all } j \in \{1, \ldots, \psi\} \text{ and no two } i_j \text{ are equal})$$

we need to find the unique polynomial of degree $\psi - 1$ that passes through all the points

$$(i_1, x_{i_1}), \ldots, (i_\psi, x_{i_\psi})$$

Although it is possible to compute the original coefficients of the polynomial used to generate the shares, this would be unnecessarily complicated for our purposes. To recover the secret value, we just need to evaluate the resulting polynomial at the point 0, and therefore we will only compute a simpler form of the polynomial called a *Lagrange interpolating polynomial* [18]. The resulting Lagrange polynomial can then easily be evaluated at any given point, including 0.

Lagrange interpolating polynomial

Let F be a field, $k \in \mathbb{Z}^+$ and let $(x_1, y_1), \ldots, (x_k, y_k) \in F^2$.

Definition 2.41. We define the polynomial P_i for all $i \in \{1, \ldots, k\}$ as

$$\mathrm{P}_i(\xi) = y_i \prod_{\substack{j=1 \\ j \neq i}}^{k} \frac{\xi - x_j}{x_i - x_j}$$

Lemma 2.42. *For all $i \in \{1, \ldots, k\}$, it holds that*

$$\mathrm{P}_i(x_i) = y_i$$

Proof. For $\xi = x_i$, all of the fractions are equal to

$$\frac{x_i - x_j}{x_i - x_j} = 1$$

and therefore the whole product is equal to 1.

$$\mathrm{P}_i(x_i) = y_i 1 = y_i$$

□

Lemma 2.43. *For all $i, j \in \{1, \ldots, k\}$, $i \neq j$, it holds that*

$$\mathrm{P}_i(x_j) = 0$$

Proof. In each case, the product now includes the fraction

$$\frac{x_j - x_j}{x_i - x_j} = 0$$

and therefore the whole product is equal to 0.

$$\mathrm{P}_i(x_j) = y_i 0 = 0$$

□

Lemma 2.44. P_i *is of degree* $k-1$.

Proof. Follows immediately from the fact that P_i is a product of a constant term and $k-1$ linear terms. □

Definition 2.45. We can now define the *Lagrange interpolating polynomial* as

$$\mathrm{P}(\xi) = \sum_{i=1}^{k} \mathrm{P}_i(\xi)$$

Lemma 2.46. P *is a polynomial of degree* $k-1$ *and for all* $i \in \{1, \ldots, k\}$ *it holds that*

$$\mathrm{P}(x_i) = y_i$$

Proof. As a sum of a finite number of polynomials of degree $k-1$, the degree of P cannot exceed $k-1$.

For all $i \in \{1, \ldots, k\}$, it holds that

$$\begin{aligned}
\mathrm{P}(x_i) &= \mathrm{P}_1(x_i) + \ldots + \mathrm{P}_{i-1}(x_i) + \mathrm{P}_i(x_i) + \mathrm{P}_{i+1}(x_i) + \ldots + \mathrm{P}_k(x_i) \\
&= \mathrm{P}_1(x_i) + \ldots + \mathrm{P}_{i-1}(x_i) + y_i + \mathrm{P}_{i+1}(x_i) + \ldots + \mathrm{P}_k(x_i) && \text{(by Lemma 2.42)} \\
&= 0 + \ldots + 0 + y_i + 0 + \ldots + 0 && \text{(by Lemma 2.43)} \\
&= y_i
\end{aligned}$$

□

2.5 Cheating prevention

By the requirements listed in Chapter 1, we cannot rely on the honesty of any participant. If there is a protocol step where a participant P may gain an advantage by not following the protocol correctly, this step must always be followed by a step where P proves to all other participants that they did not take advantage of this opportunity to cheat them.

In this section, we discuss all such exploitable protocol steps and, for each of these steps, provide a method for the participants to prove their honesty.

Since in some cases the specific protocol step may involve information that needs to stay secret, some of these proofs will need to be constructed in such a way that the verifying participant does not gain access to the secret information.

The original technique used to construct such proofs required all participants to reveal this secret information *after the end* of the protocol run (for example, after a

game of poker, all players would have to reveal all cards that they received during the game). This is often undesirable, so most of the current protocols use *zero-knowledge proofs* for this purpose instead. These allow a participant to prove to another participant that an encrypted value has some property, without revealing any information about the value. For a good overview of zero-knowledge proofs, see e.g. [19].

2.5.1 Cheating when generating the permutation

During the first phase of our protocol, the participants take turns to modify the encrypted values of the permutation elements' secret shares, as described in section 2.4.2.

A dishonest participant might try to cheat in this step by modifying the elements in an invalid way: If P represents the original polynomial with $P(0) = x$, then adding a polynomial R with $R(0) \neq 0$ to P would result in a modified value of the secret permutation element x. Although the cheating participant would be unable to predict what the resulting value would be, this would still very likely result in an invalid permutation (with a duplicate element or an element not belonging to the set $\{1, \ldots, \pi\}$).

We therefore require all participants to prove that the polynomial R they added to a specific set of secret shares satisfies the condition that

$$R(0) = 0$$

Obviously, they cannot disclose the coefficients of the polynomial or the resulting values $R(1), \ldots, R(\varphi)$ that were added to the shares, so this is an example of a protocol step where zero-knowledge proof is necessary.

However, simply modifying all permutation elements by adding a random polynomial to their secret shares is not enough. As will be described later (in section 3.2), the participants must also secretly shuffle the permutation elements to hide their order from the remaining participants. The complete statement that needs to be proven is thus as follows.

Proof statement

Let

$$M = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,\varphi} \\ c_{2,1} & c_{2,2} & \dots & c_{2,\varphi} \\ \vdots & \vdots & \ddots & \vdots \\ c_{\pi,1} & c_{\pi,2} & \dots & c_{\pi,\varphi} \end{pmatrix}$$

be a matrix where each row contains the encrypted secret shares of a permutation element, and let M' be another $\pi \times \varphi$ matrix generated by a participant P. P now needs to prove that M' is a valid permutation of the rows of M with a polynomial R with $R(0) = 0$ added to each row – ie. that the matrix M' is of the form

$$M' = \begin{pmatrix} c_{f(1),1}\,\mathrm{E}_1(R_1(1), y_{1,1}) & c_{f(1),2}\,\mathrm{E}_2(R_1(2), y_{1,2}) & \dots & c_{f(1),\varphi}\,\mathrm{E}_\varphi(R_1(\varphi), y_{1,\varphi}) \\ c_{f(2),1}\,\mathrm{E}_1(R_2(1), y_{2,1}) & c_{f(2),2}\,\mathrm{E}_2(R_2(2), y_{2,2}) & \dots & c_{f(2),\varphi}\,\mathrm{E}_\varphi(R_2(\varphi), y_{2,\varphi}) \\ \vdots & \vdots & \ddots & \vdots \\ c_{f(\pi),1}\,\mathrm{E}_1(R_\pi(1), y_{\pi,1}) & c_{f(\pi),2}\,\mathrm{E}_2(R_\pi(2), y_{\pi,2}) & \dots & c_{f(\pi),\varphi}\,\mathrm{E}_\varphi(R_\pi(\varphi), y_{\pi,\varphi}) \end{pmatrix}$$

where

1. f is a permutation of $\{1, \dots, \pi\}$
2. for all $i \in \{1, \dots, \pi\}$, R_i is a polynomial such that $R_i(0) = 0$

A zero-knowledge interactive proof

Let P be the participant who proves their honesty and V the verifying participant. They repeat the following steps T times:

1. P generates
   - a random permutation g of $\{1, \dots, \pi\}$
   - π random polynomials $Q_1, \dots, Q_\pi$
   - $\pi\varphi$ random values $z_{1,1}, \dots, z_{\pi,\varphi}$
2. P sends the following matrix M'' to V:

$$\begin{pmatrix} c_{g(f(1)),1}\,\mathrm{E}_1(R_{g(1)}(1) + Q_1(1), y_{g(1),1} z_{1,1}) & \dots & c_{g(f(1)),\varphi}\,\mathrm{E}_\varphi(R_{g(1)}(\varphi) + Q_1(\varphi), y_{g(1),\varphi} z_{1,\varphi}) \\ c_{g(f(2)),1}\,\mathrm{E}_1(R_{g(2)}(1) + Q_2(1), y_{g(2),1} z_{2,1}) & \dots & c_{g(f(2)),\varphi}\,\mathrm{E}_\varphi(R_{g(2)}(\varphi) + Q_2(\varphi), y_{g(2),\varphi} z_{2,\varphi}) \\ \vdots & \ddots & \vdots \\ c_{g(f(\pi)),1}\,\mathrm{E}_1(R_{g(\pi)}(1) + Q_\pi(1), y_{\pi,1} z_{\pi,1}) & \dots & c_{g(f(\pi)),\varphi}\,\mathrm{E}_\varphi(R_{g(\pi)}(\varphi) + Q_\pi(\varphi), y_{g(\pi),\varphi} z_{\pi,\varphi}) \end{pmatrix}$$

3. V chooses one of the following:

   (a) P reveals

   - the permutation g
   - values of $z_{0,1}, \ldots, z_{\pi,\varphi}$
   - the values of $Q_i(1), \ldots, Q_i(n)$ for all $i \in \{1, \ldots, \pi\}$

   which V then verifies by comparing the matrices M' and M''

   This is possible due to the homomorphic property of Paillier cryptosystem:

$$\mathrm{E}_j(R_{g(i)}(j) + Q_i(j), y_{g(i),j} z_{i,j}) = \mathrm{E}_j(R_{g(i)}(j), y_{g(i),j}) \, \mathrm{E}_j(Q_i(j), z_{i,j})$$

   (b) P reveals

   - the combined permutation $f \circ g$
   - values of the products $y_{0,1} z_{0,1}, \ldots, y_{\pi,\varphi} z_{\pi,\varphi}$
   - the values of $R_{g(i)}(1) + Q_i(1), \ldots, R_{g(i)}(n) + Q_i(n)$ for all $i \in \{1, \ldots, \pi\}$

   which V then verifies by comparing the matrices M and M''

   In either case, V should verify that the revealed permutation is a correct permutation of $\{1, \ldots, \pi\}$ and that all revealed polynomials evaluate to 0 at the point 0.

Note that P can cheat in this scheme (and get away with an invalid matrix M') in two different ways:

- If P correctly guesses that V will choose option (a) in step 3, P can prepare the matrix M'' such that the relationship between M' and M'' is correct, and this gives V no information about the relationship between M and M'.
- If P correctly guesses that V will choose option (b), P can prepare M'' such that the relationship between M and M'' is correct despite that the matrix M' is invalid.

However, P needs to commit to the matrix M'' before V's choice is revealed, and therefore has only a $\frac{1}{2}$ chance of cheating without detection. Repeating these steps T times reduces the probability of cheating to an arbitrarily low number $\frac{1}{2^T}$.

Also note that P never reveals the secret polynomials R_i to V: In option (a), they are not used at all, and in option (b), they are 'masked' by the random polynomials Q_i.

Similarly, the secret permutation f is either not used in the proof at all, or masked by a second random permutation g. The proof therefore provides V with no knowledge about the relationship between matrices M and M', except that M' is (very probably) of the correct form with respect to M.

Non-interactive proof

There is a universal way to replace a zero-knowledge interactive proof by a non-interactive proof [20]. We simply 'replace' the verifier V by a cryptographic hash function – instead of V choosing one of the options in step 3, P performs the following steps:

1. P generates T matrices $M''_1, \ldots, M''_T$
2. P uses all T generated matrices as an input to the cryptographic hash function
3. P takes the first T bits of the resulting hash value, and for all $i \in \{1, \ldots, T\}$ proceeds as follows:
   (a) if the i-th bit is 0, P proceeds as if V has chosen the option (a) for the matrix M''_i in step 3
   (b) if the i-th bit is 1, P proceeds as if V has chosen the option (b) for the matrix M''_i
4. P releases all of the following values:
   - all the matrices $M''_1, \ldots, M''_T$
   - for each of the matrices, all the values that would have been revealed to V in the interactive proof

For P to be able to cheat in this scheme, they would have to be able to predict the output of the hash function. P might also try to cheat by repeating step 1 until they get a specific hash value that allows P to cheat – however, the expected number of tries is 2^T, which is infeasible except for very small T.

2.5.2 Cheating when uncovering an element

When uncovering a permutation element, a participant P_i receives secret shares from other participants. To prevent these participants from cheating P_i, they need to prove

that what they sent is indeed the value of $\mathrm{D}_j(c)$, where c is a particular encrypted secret share (note that the value of c, generated during the first phase of the protocol, is known to all participants).

Let P_j be the participant who proves that c is an encryption of x to P_i. To achieve this, P_j simply sends the value of $y = \mathrm{D}'_j(c)$ to P_i, who can then easily verify that

$$c = \mathrm{E}_i(x, y)$$

(in case x or y needs to stay secret from the remaining participants, P_j would encrypt them with P_i's public key before sending).

This can be accepted as a proof of correctness, since by Theorem 2.29, there can be no other x' or y' such that $c = \mathrm{E}_i(x', y')$. However, Theorem 2.29 depends on the condition that P_j's public key is admissible – and, as we cannot rely on the honesty of P_j to choose an admissible n, we must not take this condition for granted.

Although it is possible to prove that a specific public key is admissible without revealing its prime factors [21], this is a very complicated and inefficient process. Fortunately, Theorem 2.30 provides us with an easy way to prove a weaker, but still sufficient condition.

Proving the uniqueness of decryption with a specific public key

Before the protocol starts, we will require all participants to prove that for each ciphertext c encrypted with their public key n, there is no more than one possible decryption (x, y) such that

$$\mathrm{E}_n(x, y) = c$$

Theorem 2.30 gives us a very important result: If there exists any ciphertext with more than one possible decryption, then *all* valid ciphertexts have at least two possible decryptions. This means that proving that one ciphertext has a unique decryption is equivalent to proving that all ciphertexts have unique decryptions.

A participant P can use the following steps to persuade another participant V that n provides unique decryptions, with probability at least $\frac{1}{2}$:

1. V generates two random numbers $x \in \mathbb{Z}_n$, $y \in \mathbb{Z}_n^*$ and sends

$$c = \mathrm{E}_n(x, y)$$

   to P

2. P decrypts the received value and sends

$$\begin{aligned} x' &= \mathrm{D}_n(c) \\ \text{and} \quad y' &= \mathrm{D}'_n(c) \end{aligned}$$

   to V

3. V verifies that

$$\begin{aligned} x' &= x \\ \text{and} \quad y' &= y \end{aligned}$$

If the public key n were such that the decryption is not unique, then P would have no way of knowing which of the possible decryptions (x', y') is equal to the values (x, y) that V generated. Since by Theorem 2.30 there are at least two possibilities (in many cases even more), the probability of P correctly guessing is no larger than $\frac{1}{2}$.

The probability of P cheating can be decreased to an arbitrarily low number $\frac{1}{2^T}$ by repeating these steps T times.

Note that it is absolutely necessary to perform these steps before the protocol has started (ie. before there are any secret data encrypted with the public key n). Otherwise, a dishonest verifier V could send these secret values as c in step 1 and have them inadvertedly decrypted by P. Due to the homomorphic property of the cryptosystem, this could not be prevented even if P verified that c is not equal to any of the encrypted values that need to stay secret.

Chapter 3

The protocol

In this chapter, we describe a complete run of the protocol, step by step. We will use the following notation, as described in Chapter 1:

- φ – the number of participants of the protocol (we will use an index $i \in \{1, \ldots, \varphi\}$ to denote a specific participant)
- ψ – the minimum number of participants necessary to uncover an element of the permutation
- π – the size of the permutation (we will be generating a permutation of the set $\{1, \ldots, \pi\}$)

Additionally, the participants should decide on the following security parameters before the protocol starts:

- S – bit length of the prime numbers used to generate Paillier cryptosystem public keys

  This should be chosen such that factorization of a product of two S-bit numbers is infeasible.

- T – the number of iterations in a single proof

  All of our proofs are constructed such that the probability of an attempt to cheat to stay undetected in any one iteration is at most $\frac{1}{2}$. The probability of an incorrect T-iteration proof is therefore at most $\frac{1}{2^T}$. T should be chosen large enough so that this probability is negligible.

For optimal security, we recommend $S \geq 1024$ and $T \geq 64$.

3.1 Initialization

3.1.1 Key generation

All participants should generate two prime numbers p_i, q_i of bit length S and store them securely as their private key. They should then compute the product $n_i = p_i q_i$, which becomes their public key. Each participant should distribute their public key to all other participants.

Note that the same bit length of p_i and q_i guarantees that

$$p_i < 2q_i \qquad \text{and} \qquad q_i < 2p_i$$

which in turn guarantees that their product is admissible (Definition 2.6).

3.1.2 Proving key correctness

Each participant must now prove correctness of their public key, as described in section 2.5.2. The following exchange must thus be performed between each pair of participants.

Without loss of generality, let us assume that the participant P_1 is proving correctness of their public key n_1 to participant P_2.

1. P_2 generates T random values $x_1, \ldots, x_T \in \mathbb{Z}_{n_1}$ and another T random values $y_1, \ldots, y_T \in \mathbb{Z}^*_{n_1}$ and sends

   $$\mathrm{E}_1(x_1, y_1), \ldots, \mathrm{E}_1(x_T, y_T)$$

   to P_1.

2. P_1 decrypts all received values and sends the decrypted values

   $$x'_1, \ldots, x'_T \quad \text{and} \quad y'_1, \ldots, y'_T$$

   back to P_2.

3. P_2 verifies that

   $$x'_1 = x_1 \quad \ldots \quad x'_T = x_T$$
   $$y'_1 = y_1 \quad \ldots \quad y'_T = y_T$$

3.1.3 Modulus of the field for Shamir's scheme

To satisfy all the requirements, we shall set the modulus p to the smallest prime number such that

$$p > \max\{\varphi, \pi\}$$

We will now use polynomials over the field $(\mathbb{Z}_p, +, \cdot)$ for all shared secret values (for details about Shamir's scheme, see section 2.4).

3.2 Generating the permutation

As each permutation element is described by a set of φ secret shares (see section 2.4), the whole permutation can be represented by a $\pi \times \varphi$ matrix where each row represents a single permutation element.

As described in section 2.4.1, we start with each element represented by a set of trivial secret shares. The initial permutation is thus represented by the following matrix:

$$M_0 = \begin{pmatrix} \mathrm{E}_1(1,1) & \mathrm{E}_2(1,1) & \dots & \mathrm{E}_\varphi(1,1) \\ \mathrm{E}_1(2,1) & \mathrm{E}_2(2,1) & \dots & \mathrm{E}_\varphi(2,1) \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{E}_1(\pi,1) & \mathrm{E}_2(\pi,1) & \dots & \mathrm{E}_\varphi(\pi,1) \end{pmatrix}$$

All participants P_i ($i \in \{1, \dots, \varphi\}$) then perform the following steps:

1. randomly permute the rows of M_{i-1} to get M'_{i-1}
2. for each row of M'_{i-1}:
   (a) generate $\psi - 1$ random numbers from $\mathbb{Z}_p$ – the coefficients of a random polynomial R over the field $(\mathbb{Z}_p, +, \cdot)$ such that $R(0) = 0$
   (b) for all $j \in \{1, \dots, \varphi\}$, multiply the j-th column by the encrypted value $\mathrm{E}_j(R(j))$ (due to the homomorphic property of Paillier cryptosystem, this results in addition of $R(j)$ to the original value)

   More details can be found in sections 2.4.1 and 2.4.2.
3. publish the resulting matrix as M_i
4. publish the non-interactive zero-knowledge proof from section 2.5.1 to prove correctness of M_i to all other participants

The final encrypted permutation is represented by the matrix M_φ, which all participants receive from the participant P_φ.

3.3 Uncovering a permutation element

This step can be repeated up to π times.

Since the permutation is random, the order of uncovering the elements is irrelevant. We will thus uncover the elements from the final permutation matrix M_φ row by row.

In each step, the participants decide on a single participant P_i ($i \in \{1, \ldots, \varphi\}$) who shall receive the next permutation element.

Let

$$\mathrm{E}_1(x_1, y_1), \ldots, \mathrm{E}_\varphi(x_\varphi, y_\varphi)$$

be the secret shares of the current permutation element to be revealed.

For all $j \in \{1, \ldots, \varphi\}$, $j \neq i$, the participant P_j now performs the following exchange with P_i:

1. P_j decrypts their secret share and sends both x_j and y_j to the receiving participant P_i:
$$\mathrm{E}_i(x_j), \mathrm{E}_i(y_j)$$
2. P_i decrypts the received values (let x' and y' denote the decrypted values) and verifies that
$$\mathrm{E}_j(x', y') = \mathrm{E}_j(x_j, y_j) \text{ from } M_\varphi$$

After P_i has received and verified shares from at least $\psi - 1$ other participants, P_i can compute the value of the permutation element x:

1. let x_i be P_i's secret share, and let $j_1, \ldots, j_{\psi-1} \in \{1, \ldots, \varphi\}$ be the indices of the participants from which P_i has received and verified their secret shares $x_{j_1}, \ldots, x_{j_{\psi-1}}$
2. compute the Lagrange interpolating polynomial P from the set of points
$$(i, x_i), (j_1, x_{j_1}), \ldots, (j_{\psi-1}, x_{j_{\psi-1}})$$
3. the value of the permutation element is now
$$x = \mathrm{P}(0)$$

3.3.1 Proving correctness to other participants

At an arbitrary time during the protocol run or after the protocol has finished, P_i might want to prove to other participants that x is the value they received in a particular step.

To do this, P_i simply reveals at least ψ shares x_j and y_j from the specific row of M_n. All other participants can then simply verify that $\mathrm{E}_j(x_j, y_j)$ is equal to the value in their copy of M_φ.

If P_i is only revealing the received element to a specific subset of the remaining participants, P_i simply encrypts the values of x_j and y_j by their respective public keys.

Note that this proof does not require any interaction and all participants can easily verify it, as long as they still have access to the matrix M_φ and all public keys generated during the run of the protocol.

Chapter 4

Implementation

4.1 Framework

To allow us to implement such a complicated protocol without running into technicalities at every step, we have designed a simple framework which allows us to abstract from various technical details (such as network communication) and write the protocol implementation in a way similar to the abstract mathematical notation that is commonly used in cryptography literature.

For instance, if a cryptographic protocol contained the following step:

$$A \to B : n, m$$

The implementation for participant A would contain this simple Java statement:

```
b.send("n", "m");
```

Here, `b` is an object of the class `RemoteParticipant`, which handles all communication with the participant B.

On B's machine, there is a similar `RemoteParticipant` object representing participant A. This object listens on the communication channel between A and B in the background, and when it receives the values of n and m, it automatically stores them. B can then access the stored values by simply calling

```
a.get("n")
a.get("m")
```

Any call to the `get` method *blocks* until the requested value becomes available. Therefore, it is not important whether A calls `b.send("n")` before or after B calls

`a.get("n")` – our framework handles *synchronization* of the participants automatically.

Often (especially in protocols with many participants), it is advantageous to run multiple parallel computation threads even on a single machine. For instance, while one of A's threads is blocked because it requires a value not yet received from B, another thread might continue performing computations not depending on this value.

To synchronize these local threads, our framework provides two methods: `set` and `get` (this is in fact the same method as `RemoteParticipant.get`, but in a different object). Any thread trying to `get` a value which has not yet been `set` is blocked until it becomes available.

A short overview of the framework design and its features follows. More complete documentation is provided with the source code on the attached CD.

4.1.1 Framework classes

Abstract class `DataBackedObject`

This abstract class provides the described `get` and `set` functionality to its subclasses.

In addition, it provides a special `kill` method. When an object is killed, all threads waiting on this object's `get` calls are unblocked and these `get` calls throw an exception.

This is useful to prevent deadlocks when it is clear that the values will never become available – for instance, because the network connection to a participant has been lost.

The class `DataBackedObject` has two subclasses.

Abstract class `Protocol` (extends `DataBackedObject`)

This class should be extended by all protocol implementations. The extending classes must provide a `run` method which starts the protocol. Additionally, they may provide other methods which perform the specific protocol steps. For example, our permutation protocol implementation contains an `uncover` method which uncovers the next permutation element.

The `Protocol` class provides various useful features to its subclasses:

- When a new `Protocol` object is created, a `RemoteParticipant` object is automatically created for each participant.

- Since this class extends `DataBackedObject`, it provides the `get` and `set` methods which can be used to synchronize threads performing various steps of the protocol.

  The `kill` method can also be useful when the protocol run has failed for some reason (e.g. a disruption by participants trying to cheat) and needs to be ended before all threads have finished their computations.

- The `runWithEach(ParticipantAction action)` method starts a new thread for each remote participant and runs the specified action in this thread. This is usually much faster then running the action for each participant in turn, since while one thread is waiting to receive a value from the remote participant, other threads can continue performing computations and communicating with the other participants.

  The `ParticipantAction` is an interface with a single method:

  ```
  public void run(RemoteParticipant p);
  ```

  By default, when any of the launched threads fail (throw an uncaught exception), the `kill` method of the protocol is called immediately to stop the protocol.

- `broadcast(key)` is useful for sending a value to all other participants in paralell, ie. it provides an abbreviated syntax for the following statement:

  ```
  runWithEach(new ParticipantAction() {
      public void run(RemoteParticipant p) {
          p.send(key);
      }
  });
  ```

The constructor of `Protocol` (and therefore of all its subclasses as well) requires that an `InputStream` and an `OutputStream` object is provided for each remote participant. The initialization of these objects is left to the programmer of the particular application, however, we provide some helper classes to make this easier (see section 4.1.2).

Class `RemoteParticipant` (extends `DataBackedObject`)

An array of `RemoteParticipant` objects is automatically initialized by the `Protocol` class. Subclasses of `Protocol` can either access these objects directly using the `participants` array, or they can use the `runWithEach` method described earlier.

Each `RemoteParticipant` object handles all operations related to the specific participant:

- When a `RemoteParticipant` object is initialized, a background thread is started automatically which reads all data received from the participant and stores it using the inherited `set` method.
- The inherited `get` method can be used to access any value received from the remote participant.
- When connection to the participant is lost, the inherited `kill` method is called to immediately stop all threads depending on data from this participant.

  This usually results in the thread throwing an uncaught exception and killing the whole protocol, but it does not have to be so. For instance, when uncovering a permutation element in our protocol, we catch this exception and only kill the whole protocol if less than ψ participants are left.
- The `send(key)` method can be used to send values from *local* (`Protocol`) data to the participant. This method calls `protocol.get(key)`, and is therefore blocked if the respective value is not available yet.

4.1.2 Helper classes

The described framework classes do not handle network connections in any way. When creating a new `Protocol` instance, a set of `InputStream` and `OutputStream` objects must already be prepared by the application creating the `Protocol` object. To simplify this process for most common cases, we provide some simple helper classes.

Input/output stream multiplexing

Each protocol implemented using our framework needs to have its own communication channel open with each participant. Therefore, it is often necessary to have more than one communication channel with the same participant (for example, one for protocol data and one for application data).

To prevent application developers from having to open multiple connections to the same host, we provide two simple classes: `OutputStreamMultiplexer` takes one `OutputStream` objects and splits it into up to 256 new `OutputStream`s.

On the other end, an `InputStreamDemultiplexer` must be used to split the resulting combined output stream into its parts.

Example usage:

```
OutputStream os = ...;
OutputStreamMultiplexer osm = new OutputStreamMultiplexer(os):
osm.getOutputStream(0).write(a);
osm.getOutputStream(1).write(b);

InputStream is = ...;
InputStreamDemultiplexer ism = new InputStreamDemultiplexer(is);
System.out.println(ism.getInputStream(0).read());
    // outputs the value of a
System.out.println(ism.getInputStream(1).read());
    // outputs the value of b
```

Class `ConnectionManager`

This class can be used to initialize connections with all remote participants easily.

As an input, it takes an array of network addresses, the ID of the local participant (the application developer must make sure that the ID is unique) and optionally the number of input/output streams that are to be provided for each participant.

The `ConnectionManager` then opens a connection to each of the given addresses and provides an array of `InputStream` and `OutputStream` objects ready to be used in the `Protocol` constructor. If the number of streams requested was more than one, each stream is split into more streams using `OutputStreamMultiplexer` and `InputStreamDemultiplexer`.

4.2 Protocol implementation

Using the described framework, we have implemented the complete permutation protocol from this thesis. The resulting Java archive, `cryperm.jar`, can be attached to any Java application and then used freely from within this application. This section describes the basic usage of the library.

The protocol implementation is represented by the class `Cryperm` (which, as required, extends the abstract framework class `Protocol`).

To securely generate a random permutation from within a network application, simple initialize an object of this class (a new object can be initialized for each permutation):

```
Cryperm perm1 =
    new Cryperm(
        localId,
        inputStreams,
        outputStreams,
        requiredParticipantCount,
        permutationSize
    );
```

where

- `localId` is the index of the local participant (the application must guarantee that on each machine, the `localId` is unique and that no participant is missing)
- `inputStreams`, `outputStreams` – arrays of streams connected to the other participants (the length of both arrays must be equal to the number of participants)

  `ConnectionManager` can be used to initialize connections between each pair of participants and prepare the `inputStreams` and `outputStreams` arrays.
- `requiredParticipantCount` – the value of ψ (as used in this document)
- `permutationSize` – the value of π (as used in this document)

After the `Cryperm` object is created, the protocol is immediately started (this spawns several threads, as described in the previous section).

The permutation elements can then be uncovered by calling

```
int element = perm1.uncover(recipientId);
```

or

```
int element = perm1.uncover(recipientId, elementId);
```

This performs the steps necessary for the participant `recipientId` to successfully uncover the next permutation element. All participants must make sure to call this method with the same arguments. If `recipientId` is the local participant, then the method blocks until the result is ready and returns the uncovered value.

See section 4.3.1 for an example of how all the parts of the library are combined.

4.2.1 Parallelism

Our implementation of the protocol takes full advantage of the framework's parallelism capabilities. For instance, the correctness of a participant's public key is proven to all other participants in parallel. Similarly, after the permutation is generated, all requests to uncover an element are handled paralelly – and thus, if one participant is too slow or even drops out, it does not slow down the other participants that are waiting to have their elements uncovered.

As soon as the first participant receives all public keys, they can generate the initial permutation matrix and start the permutation process (this can be done in parallel to the public key correctness proofs). Furthermore, if the first participant shuffles the elements (rows of the matrix) first and then starts adding random polynomials to each row, they can send each row to the second participant as soon as the values of this row are calculated (ie. there is no reason to wait until the whole matrix is ready).

Unfortunately, if the second participant were to proceed in the same way (shuffle the rows first and then add the polynomials and send them out), they would first have to wait until the whole matrix is received, which would cause all parallelism to be lost.

Therefore, we have decided to break the symmetry by having all odd-numbered participants $(1, 3, \ldots)$ shuffle the rows before adding the polynomials and sending them out, and all even-numbered participants $(2, 4, \ldots)$ add the polynomials to the rows as they are received, and only after all the rows are received and modified, shuffle the rows and send them out to the next participant (see Figure 4.1).

This speeds up the permutation process by approximately a factor of 2. Although the asymptotic time complexity of the protocol stays the same, it is of considerable importance in practice, especially for the common special case $\varphi = 2$, where, as a result, the protocol is almost completely parallel.

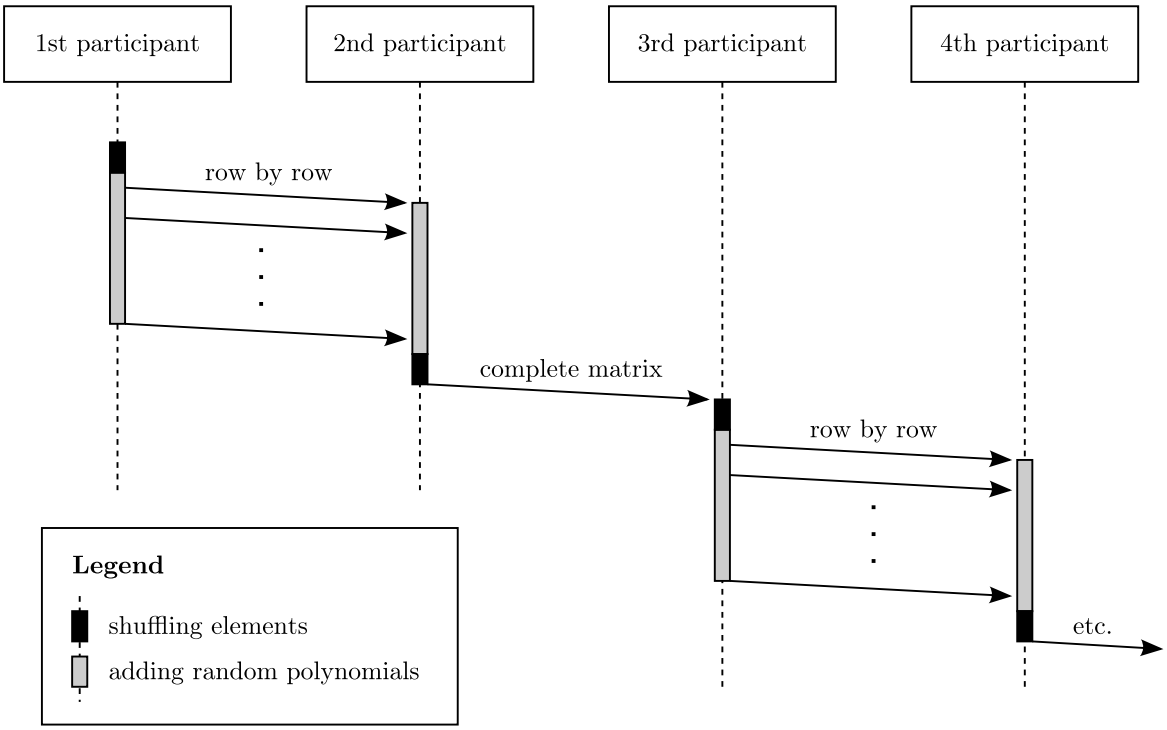

Figure 4.1: Scheme of parallelism in the permutation generating step

4.3 Example application

We have used our library to implement a simple console (text-only) poker application. This application first deals five cards to each player and then allows the players to replace some of the received cards.

The following is a sample output of one run of the application:

```
Card 1:  4♢
Card 2:  K♣
Card 3:  Q♢
Card 4:  2♢
Card 5:  10♡

Replace cards:  1 4
Replacing 2 cards...

Card 1:  2♡
Card 2:  K♣
Card 3:  Q♢
```

```
Card 4:  5♣
Card 5:  10♡
```

Although we decided to leave out the implementation of game mechanics that are not directly related to card dealing (such as betting or keeping track of the players' money), all of these could easily be handled by the players communicating in a separate channel, e.g. an external chat room. Therefore, our application provides a complete and working (albeit not exactly user friendly) solution to players who want to play poker securely over a computer network.

4.3.1 Implementation

Our application needs a configuration file which contains the number of players (φ), number of players necessary to uncover a card (ψ) and network addresses of all the players (this configuration file must be the same for each player). The game must then be started for each player (on the machine with the respective address) as follows:

```
./crypoker <path to configuration file> <player ID>
```

After parsing the configuration file, the application uses `ConnectionManager` to establish connections to all other players:

```
ConnectionManager cm = new ConnectionManager(localId, addresses, 2);
```

Note that we have requestes two streams to be created for each player – one for the permutation protocol and one for the poker application. The connection manager provides us with arrays of streams necessary for starting the permutation protocol:

```
Cryperm cryperm =
    new Cryperm(
        localId,
        cm.getPartialInputStreams(1),
        cm.getPartialOutputStreams(1),
        requiredPlayerCount,
        52
    );
```

First, five cards are uncovered to each player:

```
int[] cards = new int[5];

for (int i = 0; i < playerCount; ++i) {
    for (int j = 0; j < 5; ++j) {
        int card = cryperm.uncover(i);
        if (i == localId) {
            cards[j] = card;
        }
    }
}
```

The second set of streams created by `ConnectionManager` is then used to communicate the number of cards to exchange to the remaining players. Additional calls to `cryperm.uncover()` are then used to exchange the correct number of cards for each player.

Conclusion

We have successfully designed and implemented a cryptographic protocol according to the stated requirements.

As a proof of concept, we have also implemented a simple poker application and tested it using various parameters.

As expected (by the large number of operations on very large numbers, especially in the zero-knowledge proofs), the application is not always fast enough to be used in practice. However, relaxing the security parameters to smaller (but still reasonable) values can make the complete protocol run fast enough to successfully play a game of poker.

For example, with $S = 256$ and $T = 16$ (see Chapter 3 for explanation of the parameters), the card-shuffling process was completed in approximately one minute on our testing machines. Although these parameters are much smaller than the recommended values, an attacker would still need several hours to break the protocol even with considerable resources at their disposal. Such parameters are therefore perfectly sufficient for playing a game of poker which should not take more than a few minutes.

Further speed gains could be achieved by optimizing both the protocol implementation and our framework (e.g. the network communication model). However, we believe that our implementation has successfully shown that the proposed protocol is not only of theoretical interest, but can be implemented and used for practical purposes as well.

On-line resources

For up-to-date information about the status of the project, please check the project website at `http://s.ics.upjs.sk/~jjergus/dp/`.

Resumé

Našim cieľom bolo popísať a implementovať kryptografický protokol, ktorý ľubovoľnej skupine φ ľudí umožní spoločne, bez pomoci dôveryhodnej tretej strany, vygenerovať náhodnú permutáciu π prvkov. Prvky permutácie potom môžu byť postupne odhaľované jednotlivým účastníkom protokolu, pričom počet odhalených prkov ani priradenie jednotlivých prvkov účastníkom nie je určené.

Typickým využitím takéhoto protokolu je napríklad miešanie kariet pri tzv. *mentálnom pokri*, teda v protokoloch, ktoré umožňujú skupine ľudí hrať poker prostredníctvom počítačovej siete bez pomoci dôveryhodného servera.

Protokol

Prvá fáza protokolu (vytvorenie zašifrovanej permutácie) vyžaduje vstupy od všetkých účastníkov, aby za predpokladu, že aspoň jeden účastník je čestný, bola zaručená náhodnosť výslednej permutácie. Na dešifrovanie prvku vygenerovanej permutácie však už stačia len vstupy od ψ účastníkov ($2 \leq \psi \leq \varphi$), kde ψ je parameter, na ktorom sa účastníci pred spustením protokolu dohodnú. V špeciálnom prípade môže byť zvolené $\psi = \varphi$, čím sa dosiahne maximálna bezpečnosť.

Pri návrhu protokolu sme vychádzali z predpokladu, že všetka komunikácia medzi účastníkmi je verejná, a teda všetky informácie, ktoré musia zostať pred niektorými účastníkmi utajené, musia byť *zašifrované*. Na šifrovanie sme použili *Paillierov kryptosystém* [15], ktorý má okrem dostatočnej bezpečnosti (ze predpokladu, že tzv. *problém vyšších rezíduí zložených čísel* nie je efektívne riešiteľný) má aj ďalšie matematické vlastnosti nevyhnutné pre náš protokol.

Aby sme splnili požiadavku, že na odkrytie prvku permutácie stačí spolupráca ψ z φ účastníkov protokolu, použili sme *Shamirovu prahovú schému na zdieľanie tajomstva* [17]. Každý prvok permutácie je tak rozdelený na φ zašifrovaných častí, pričom ľubovoľných ψ z nich stačí na dešifrovanie príslušného prvku.

Počas generovania zašifrovanej permutácie musí každý z účastníkov dokázať, že postupoval podľa protokolu. Aby nebola porušená bezpečnosť protokolu, používame na tento účel *bezznalostný dôkaz*.

Implementácia

Na uľahčenie implementácie protokolu sme si najprv vytvorili *framework*, ktorý nám umožnil abstrahovať od technických detailov ako sú sieťová komunikácia či synchronizácia jednotlivých účastníkov.

S pomocou tohto frameworku sme potom implementovali kompletný navrhnutý protokol ako knižnicu jazyka Java, ktorá je jednoducho použiteľná z ľubovoľnej sieťovej aplikácie.

Ako ukážku sme implementovali jednoduchú textovú verziu hry poker.

Hoci naša implementácia s doporučenými bezpečnostnými parametrami môže byť príliš pomalá na praktické používanie, znížením týchto parametrov na nižšiu (no stále rozumnú) úroveň sa dá rýchlosť dostatočne zvýšiť. Podarilo sa nám teda ukázať, že navrhnutý protokol nie je len teoretickým výsledkom, ale je uplatniteľný aj v praxi.

On-line zdroje

Na stránke `http://s.ics.upjs.sk/~jjergus/dp/` sú zverejňované aktuálne informácie o projekte.

Bibliography

1 BLUM, Manuel. Coin flipping by telephone a protocol for solving impossible problems. *SIGACT News*. 1983, vol. 15, pp. 23–27. ISSN 0163-5700.

2 HIRT, Martin. *Multi-Party Computation: Efficient Protocols, General Adversaries, and Voting*. 2001. Reprint as vol. 3 of *ETH Series in Information Security and Cryptography*, ISBN 3-89649-747-2, Hartung-Gorre Verlag, Konstanz, 2001.

3 YAO, Andrew C. Protocols for secure computations. In. *Proceedings of the 23rd Annual IEEE Symposium on Foundations of Computer Science, FOCS'82*. 1982, pp. 160–164.

4 SHAMIR, Adi; RIVEST, Ronald L.; ADLEMAN, Leonard M. Mental Poker. *The Mathematical Gardner*. 1981, pp. 37–43.

5 CRÉPEAU, Claude. A Zero-Knowledge Poker Protocol that Achieves Confidentiality of the Players' Strategy or How to Achieve an Electronic Poker Face. In ODLYZKO, Andrew M. (ed.). *Advances in Cryptology - CRYPTO '86: Proceedings*. 1986, pp. 239–247. Lecture Notes in Computer Science. ISBN 3-540-18047-8.

6 SCHINDELHAUER, Christian. *A Toolbox for Mental Card Games*. 1998.

7 KUROSAWA, Kaoru; KATAYAMA, Yutaka; OGATA, Wakaha. Reshufflable and Laziness Tolerant Mental Card Game Protocol. *IEICE Transactions Fundamentals*. 1997, vol. E80-A, no. 1, pp. 72–78. ISSN 0916-8508.

8 KUROSAWA, Kaoru; KATAYAMA, Yutaka; OGATA, Wakaha; TSUJII, Shigeo. General public key residue cryptosystems and mental poker protocols. In. *Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology*. Aarhus, Denmark : Springer-Verlag New York, Inc., 1991, pp. 374–388. EUROCRYPT '90. ISBN 0-387-53587-X.

9 CASTELLÀ-ROCA, Jordi. *Contributions to Mental Poker*. 2005. ISBN 84-689-6233-3.

10 NEFF, C. Andrew. A verifiable secret shuffle and its application to e-voting. In. *Proceedings of the 8th ACM conference on Computer and Communications Security*. Philadelphia, PA, USA : ACM, 2001, pp. 116–125. CCS '01. ISBN 1-58113-385-5.

11 NIVEN, Ivan; ZUCKERMAN, Herbert S.; MONTGOMERY, Hugh L. *An introduction to the theory of numbers*. New York, NY, USA : John Wiley & Sons, Inc., 1991. ISBN 0-471-62546-9.

12 GOLDWASSER, Shafi; MICALI, Silvio. Probabilistic encryption & how to play mental poker keeping secret all partial information. In. *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. San Francisco, California, United States : ACM, 1982, pp. 365–377. STOC '82. ISBN 0-89791-070-2.

13 GJØSTEEN, Kristian. Homomorphic public-key systems based on subgroup membership problems. *IACR Cryptology ePrint Archive*. 2003, vol. 2003, pp. 131.

14 AKINWANDE, Mufutau. Advances in Homomorphic Cryptosystems. *J. UCS*. 2009, vol. 15, no. 3, pp. 506–522.

15 PAILLIER, Pascal. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In STERN, Jacques (ed.). *EUROCRYPT*. 1999, pp. 223–238. Lecture Notes in Computer Science. ISBN 3-540-65889-0.

16 JURIK, Mads J. *Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols*. 2003. PhD thesis. xii+117 pp.

17 SHAMIR, Adi. How to share a secret. *Commun. ACM*. 1979, vol. 22, pp. 612–613. ISSN 0001-0782.

18 ARCHER, Branden; WEISSTEIN, Eric W. *Lagrange Interpolating Polynomial*. From *MathWorld* – A Wolfram Web Resource. Available from Internet: ⟨`http://mathworld.wolfram.com/LagrangeInterpolatingPolynomial.html`⟩.

19 GOLDREICH, Oded. *Zero-Knowledge twenty years after its invention*. 2002.

20 SCHNEIER, Bruce. *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*. New York, NY, USA : John Wiley & Sons, Inc., 1995. ISBN 0-471-11709-9.

21 CAMENISCH, Jan; MICHELS, Markus. Proving in zero-knowledge that a number is the product of two safe primes. In. *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*. Prague, Czech Republic : Springer-Verlag, 1999, pp. 107–122. EUROCRYPT'99. ISBN 3-540-65889-0.

Attachments

A. CD with source code of the Java library and example application.