Institute of Information Security

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Secure Distributed Paillier Key Generation with Application to the Ordinos E-Voting System

Felix Truger

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Ralf Küsters
Supervisor:	Julian Liedtke, M.Sc.
Commenced:	November 5, 2019
Completed:	June 30, 2020

Abstract

Ordinos is a novel verifiable tally-hiding e-voting system. At its heart, a homomorphic encryption scheme and secure multi-party computation (MPC) are used to tally votes and securely determine the voting result, without necessarily revealing the full tally (e.g., the number of votes per candidate). The proof of concept implementation of Ordinos is based on a threshold variant of the Paillier encryption scheme and two MPC protocols for the comparison of encrypted numbers (greater-than and equality). Due to the threshold construction, the decryption key is shared among a set of trustees. The MPC protocols for comparison require precomputed encrypted randomness of certain shape. Formerly, a trusted party was employed to generate the key shares and randomness and distribute them to the trustees. In this thesis, the trusted party was replaced by MPC protocols that allow to generate the key shares and randomness among the trustees. The protocols provide security against malicious parties in the honest-majority setting. The key generation follows a proposal by Nishide and Sakurai (2010) that is based on verifiable secret sharings and zero-knowledge proofs for committed values. We introduce a few adaptations to reduce its runtime using mostly standard techniques. The generation of randomness is based on the Paillier encryption scheme as an arithmetic black box and standard zero-knowledge proofs for Paillier encrypted values. The protocols were implemented and their performance was evaluated in a local network. Most notably, the implemented key generation protocol for threshold Paillier showed an expected average runtime around 95 minutes for generating 2048-bit keys among 3 trustees with a threshold of 2. Since existing implementations provide security only in the semi-honest setting, this is the first time that an approach with security against malicious parties was implemented and evaluated. Overall, the distributed generation of both key shares and randomness takes considerably more time compared to the use of a trusted party, but avoids security risks and trust problems that occur with trusted parties.

Zusammenfassung

Ordinos ist ein neuartiges verifizierbares E-Voting System mit der sogenannten Tally-Hiding-Eigenschaft. Im Kern werden ein homomorphes Verschlüsselungsschema und sichere Multi-Party Computations (MPC) verwendet, um Stimmen zu zählen und das Ergebnis einer Abstimmung zu bestimmen, ohne notwendigerweise das vollständige Auszählungsergebnis (z. B. die Anzahl der Stimmen je Kandidat) preiszugeben. Die Proof of Concept Implementierung von Ordinos basiert auf einer Threshold-Variante des Paillier-Verschlüsselungsschemas und zwei MPC-Protokollen für den Vergleich von verschlüsselten Zahlen (größer-als und Gleichheit). Aufgrund der Threshold-Variante wird der private Schlüssel unter mehreren Trustees verteilt aufbewahrt (engl.: sharing). Die MPC-Protokolle für die Vergleiche erfordern vorberechnete verschlüsselte Zufallszahlen bestimmter Form. Bisher wurde eine Trusted Third Party eingesetzt, um die Schlüssel-Shares und Zufallszahlen zu generieren und an die Trustees zu verteilen. In dieser Arbeit wurde die Trusted Third Party durch MPC-Protokolle ersetzt mit denen die Schlüssel-Shares und Zufallszahlen unter den Trustees generiert werden. Die Protokolle bieten Sicherheit gegen aktive Angreifer im Honest-Majority Szenario. Die Schlüsselgenerierung basiert auf einem Paper von Nishide and Sakurai (2010), wobei hierzu verifizierbare Secret-Sharings und Zero-Knowledge Beweise für Commitments eingesetzt werden. Um die Laufzeit zu reduzieren, werden einige Anpassungen vorgestellt, die sich überwiegend an Standardtechniken orientieren. Die Generierung der Zufallszahlen basiert auf dem

Paillier-Verschlüsselungsschema als arithmetische Black Box und Zero-Knowledge Beweisen für Paillier-Chiffren. Die Protokolle wurden implementiert und ihre Performance in einem lokalen Netzwerk evaluiert. Hervorzuheben ist, dass die implementierte Schlüsselgenerierung eine zu erwartende durchschnittliche Laufzeit von etwa 95 Minuten zwischen 3 Trustees mit einen Threshold von 2 und der Schlüssellänge 2048 Bit benötigt. Bestehende Implementierungen bieten Sicherheit gegen passive Angreifer. Somit setzt diese Arbeit zum ersten Mal ein Ansatz mit Sicherheit gegen aktive Angreifer um und evaluiert diesen. Insgesamt benötigt die verteilte Generierung der Schlüssel-Shares und Zufallszahlen zwar erheblich mehr Zeit verglichen zum Einsatz einer Trusted Third Party, vermeidet aber Sicherheitsrisiken und Vertrauensprobleme, die mit solchen auftreten.

Contents

List of Figures

List of Tables

1 Introduction

Democracy is a fundamental concept of modern human life, that relates not only to political elections and the political system of states or parties, but also to (everyday) decisions in enterprises, associations or simply groups of people. Be it a decision about an upcoming investment, appointment or the election of a new executive board – in most cases a quick voting will make it easy. Evidently, there are many modes and varieties of voting. For example, the decision on a simple yes-no question by majority vote compared to the decision on the allocation of seats in a parliament by proportional representation. Also, different ways to realize a voting are known, such as acclamation (e.g., by hand signal or voice vote) and voting by casting a ballot (e.g., a sheet of paper with the name of the preferred choice or candidate). Each of which have their advantages and disadvantages depending on the desired properties of the election or personal preferences of the voter.

For the majority of votings the properties freedom of choice, secrecy and transparency are essential. I.e., the voter shall be able to make their choice without being influenced or observed by others and the system should keep the choice of each individual voter private while allowing for transparency, such that everyone can easily verify, that the voting was fair with respect to certain regulations. For example, it should be ensured, that only eligible voters submit their ballot at most once and their vote is tallied correctly (unless it does not conform to the formal regulations). Traditionally in many official national elections, ballot papers, voting booths, urns, and public counting of votes are measures that ensure these properties. However, this procedure is often very costly, slow, and prone to errors of manual work. Even worse, despite all measures, the transparency may still be insufficient. For example, in elections that include several voting sites, one individual cannot observe the tallying process on all sites. Obviously, one would need to trust others, to monitor the tallying there. Similarly, trust is required for the correct totaling after the tallying process on each site has finished. Thus, a voting system that allows for more transparency is desirable.

Starting already in the 19th century (mechanical) voting machines were proposed to improve the process of voting with regards to the mentioned properties, mostly concentrating on secrecy and freedom of choice [30]. More recently, with the wide spread of electronic devices, the use of computer systems for voting has been proposed. One can think about many kinds of electronic voting (e-voting) aids or systems. For example, computers can be used to scan and tally paper ballots [see, e.g., 53], a standalone e-voting machine could sequentially collect all votes via an electronic interface and tally them for a voting of smaller scale, or a voting software distributed on several computer systems and connected via the internet can be employed [e.g., 1, 33]. The advantage of such systems is that, depending on the construction, some of their properties may be proven formally.

For the purpose of this thesis we focus on the latter kind of e-voting systems and more specifically on such systems that make use of cryptographic means to (provably) ensure certain desired properties of voting systems. Obviously, it is especially essential for an e-voting system to provide security and privacy features in order preserve the aforementioned properties. Otherwise, a wide range of election fraud could not be prevented: The system should ensure that only votes of eligible voters that conform to the regulations of the voting (e.g., at most one ballot per voter) are considered for the calculation of the result, so that an adversary cannot maliciously submit (additional) votes to the system. It should be able to check the authenticity and integrity of ballots, so that an adversary cannot fake or change the votes of others. It should inherently keep the votes of all participants secret, so that no adversary can derive the contents of a ballot of other voters and/or stop the transfer of selected ballots. Furthermore, the system should be transparent, so that every voter can *verify* that their vote was actually considered for the calculation of the final outcome and the calculation was conducted correctly.

Generally, there is a set of usually desired properties for e-voting systems, that includes vote privacy and verifiability. A recently proposed e-voting system, named Ordinos, adds a new property to this set [33]: **Tally-hiding** means, that the e-voting system is able to reveal only the voting result. For example, only the ranking of the candidates or choices could be revealed, while keeping the full tally, possibly including the number of votes per candidate, secret. Similarly, only a part of the ranking, such as a predefined number of the first or last ranks could be revealed or, even simpler, only the winner(s) of the voting. A tally-hiding e-voting can bring several advantages compared to other e-voting systems, that regularly reveal the full tally, usually including the number of votes accumulated by each candidate. It can improve vote privacy in elections with only a few voters, as clearly the privacy level of such votings is very low. Furthermore, in elections with several rounds, e.g., such that include runoff elections, a tally-hiding system can help to reduce bias of the voters that might come with too much information about intermediate results. Also, the embarrassment of loosing candidates might be lowered or avoided.

In a nutshell, the Ordinos protocol is run by a voting authority, who organizes the election and schedules its phases, a number of voters, who cast their votes, and a number of trustees, who tally the votes and jointly calculate the result function. After the election, any party can use the publicly available information to check and verify the run of the protocol. The tally-hiding property is achieved via appropriate cryptographic measures, that include homomorphic encryption and secure multi-party computation (MPC). The instantiation of Ordinos presented by Küsters et al. relies on a threshold variant of the Paillier encryption scheme, because it has a more efficient decryption algorithm compared to exponential ElGamal. Simply speaking, each trustee possesses a public/private key share pair. From the public keys of these pairs the voters can deduce the overall public key and use it to encrypt their votes. After the voting was closed, the trustees tally the encrypted votes and determine the encrypted accumulated votes per candidate. This is possible due to the additive homomorphicity of Paillier's encryption scheme. In the next step, they engage in MPC to compare the encrypted numbers in order to calculate a ranking of the candidates and possibly the desired subset for the tally-hiding result. Finally, the tally-hiding result is decrypted by distributed decryption among the trustees. The threshold setup for the encryption scheme ensures, that a certain number of trustees is required to cooperate in order to decrypt ciphertexts. During the protocol none of the participants learn the content of single votes or the total number of votes per candidate (unless that was the desired goal of the voting).

In the current implementation the trustees' public/private key share pairs are generated by a trusted third party. Generally, trusted third parties yield some problems: In an environment of many distrustful parties it is hard to establish a single party that accumulates the trust of all participants. Furthermore, the central generation and administration of keys result in a single point of attack, that can lead to leakage of information. An attacker who is able to get access to the trusted third party, e.g., through flaws in the implementation, could be able to learn the private keys of multiple trustees. The goal of this thesis is to investigate the possibility of a secure distributed generation of the key share pairs among the trustees and implement such a solution in order to omit the trusted third party from the protocol. This includes distributedly generating encrypted randomness of certain shape, which is used for the MPC protocols and currently additionally provided by the trusted third party.

The thesis is structured as follows:

- **Chapter 2 – Fundamentals** introduces the Paillier encryption scheme and threshold encryption as well as the building blocks for the secure key generation protocol.
- **Chapter 3 – Related Work** contains a review of existing work on secure distributed Paillier key generation and related topics.
- **Chapter 4 – Secure Distributed Paillier Key Generation** describes a construction for secure distributed Paillier key generation.
- **Chapter 5 – Implementation** presents the main aspects of the implementation of the construction proposed in the previous chapter, some adaptations of the protocol and an evaluation.
- **Chapter 6 – Generation of Randomness for MPC Protocols** deals with the task of jointly and securely generating randomness of certain shape for MPC protocols used in the tallying phase of Ordinos.
- **Chapter 7 – Conclusion** summarizes and concludes the thesis with an outlook on possible future work.

2 Fundamentals

This chapter introduces some definitions and fundamentals for the secure key generation protocol. More precisely, it gives a brief introduction into public-key encryption, Paillier and threshold encryption before providing cryptographic primitives for the distributed key generation protocol.

2.1 Public-Key Encryption Schemes

Public-key cryptosystems (also known as asymmetric cryptosystems) became subject to research in the mid ’70s [see, e.g., 32, 46]. RSA is one of the best-known and most widely adopted of such cryptosystems. While most public-key cryptosystems are fit for multiple purposes, we focus on their usage as encryption schemes.

The main difference compared to traditional symmetric encryption is that the key in public-key schemes consists of two parts. Compared to one key in traditional symmetric encryption that is known to both communication partners and used for encryption and decryption of messages on the respective side of a channel, in public-key encryption a key-pair consisting of a public key and a private key is used. The public key is known to the sender of a message, usually even available to the public, while the private key remains a secret known only to the receiver. Messages encrypted using the public key can be decrypted using the corresponding private key.

More formally, a public-key encryption scheme is a triple of algorithms $\mathcal{S} = (KeyGen, Enc, Dec)$, where:

- $KeyGen$ is a probabilistic polynomial time algorithm, that outputs - at least with a high probability - a public/private key pair $(k_{\text{publ}}, k_{\text{priv}}) \in K$ (where K is the range of $KeyGen$).
- Enc is a probabilistic polynomial time encryption algorithm that takes a public key k_{publ} and plaintext m as in put and outputs a ciphertext c.
- Dec is a deterministic polynomial time decryption algorithm that takes a private key k_{priv} and ciphertext c as input and outputs a plaintext m. On failure it outputs $\perp$.

such that: $\forall m \in \{0,1\}^*, \forall (k_{\text{publ}}, k_{\text{priv}}) \in K : D_{k_{\text{priv}}}(E_{k_{\text{publ}}}(m)) = m$ (perfect correctness) holds.

Security of most public-key encryption schemes is based on a so-called “trapdoor one-way function”. Basically, such an encryption function is efficiently computable, while its inverse (i.e., decryption) function is not efficiently computable unless a certain secret, the corresponding private key, is known. The well-known example RSA is based on the factorization problem and the RSA assumption: In order to compute the decryption function, one needs to know the private key, which can be derived merely from the prime factorization of a composite parameter, the RSA modulus. In general, computing the prime factorization of composite numbers is believed to be computationally hard to solve. Thus, an attacker would not be able to retrieve the prime factors in order to calculate the private key with reasonable effort. The RSA assumption states, that it is also hard in general to compute x from $x^e \mod n$ knowing only the RSA modulus n and the public exponent e.

2.2 The Paillier Encryption Scheme

The Paillier encryption scheme is a public-key encryption scheme, that was proposed by Paillier in 1999 [43]. It was soon adapted by several authors and generalized by Damgård and Jurik [16], who also proposed one of the first threshold variants of the scheme. The following sections introduce some mathematical foundations as prerequisites before giving a definition of Paillier's scheme and briefly explaining its homomorphicity property, which is relevant for its application in Ordinos. The descriptions are adapted from [31, 43] and [18].

2.2.1 Prerequisites

We denote by $\mathbb{Z}_n$ the group under addition modulo n of elements in range $\{0, ..., n-1\}$ and by $\mathbb{Z}_n^*$ the multiplicative group of integers in $\mathbb{Z}_n$ that are relatively prime to n and thus invertible modulo n. I.e.,

$$\mathbb{Z}_n^* := \{i | i \in \mathbb{Z}_n \land gcd(i, n) = 1\}$$

It is well known, that Euler's totient function ϕ calculates the number of elements in $\mathbb{Z}_n$ that are relatively prime to n and thus the order of such groups: $|\mathbb{Z}_n^*| = \phi(n)$.

Let p, q be distinct large primes of the same binary length l, i.e. $2^l < q < p$ (w.l.o.g.) $< 2^{l+1}$, and $n := p \cdot q$. We know that $\phi(n) = (p-1) \cdot (q-1)$. Then $gcd(n, \phi(n)) = 1$: Clearly, since $n = p \cdot q$ and $p > p-1 > q > q-1$ we have $gcd(p, \phi(n)) = 1$ and $gcd(q, q-1) = 1$. It is only left to show $gcd(q, p-1) = 1$. Suppose $gcd(q, p-1) \neq 1$, then $gcd(q, p-1) = q$. Subsequently, $\exists k \geq 2 : p - 1 = k \cdot q$, because $(p-1) \neq q$. However, that would require $p = (k \cdot q) + 1$ for $k \geq 2$, which contradicts the choice of p, q of the same binary length.

2.2.2 The Group $\mathbb{Z}_{n^2}^*$

An Isomorphism $f : \mathbb{Z}_n \times \mathbb{Z}_n^* \to \mathbb{Z}_{n^2}^*$

Paillier found in [43], that the function

$$f : \mathbb{Z}_n \times \mathbb{Z}_n^* \to \mathbb{Z}_{n^2}^*; (x, y) \mapsto g^x \cdot y^n \mod n^2$$

for $g \in \mathbb{Z}_{n^2}^*$ is bijective, if the order[1] of $g \mod n^2$ is a nonzero multiple of n. As suggested by Paillier himself, it is sufficient to set $g := n + 1$, which we will do from now on. It is easy to see, that

$$(n+1)^a \mod n^2$$

[1]Recall, that the order of an element g modulo n is the smallest positive integer i, such that $g^i \mod n = 1$

$$= \sum_{i=0}^{a} \binom{a}{i} \cdot n^i \mod n^2$$

$$= \binom{a}{0} \cdot n^0 + \binom{a}{1} \cdot n^1 + ... + \binom{a}{a} \cdot n^a \mod n^2$$

$$= 1 + a \cdot n \mod n^2$$

and thus the order of $(1+n) \mod n^2$ is n (because $a := n$ is obviously the smallest positive integer leading to $1 + n \cdot n \mod n^2 = 1$).

Using Euler's totient function we can deduce, that

$$|(\mathbb{Z}_n \times \mathbb{Z}_n^*)| = n \cdot \phi(n) = (p \cdot q) \cdot (p-1) \cdot (q-1) = (p \cdot (p-1)) \cdot (q \cdot (q-1)) = \phi(n^2) = |\mathbb{Z}_{n^2}^*|$$

So bijectivity of f would follow from its injectivity, since both sets are of the same finite size. Injectivity is shown for $a, b \in \mathbb{Z}_n$ and $c, d \in \mathbb{Z}_n^*$, such that $f(a,c) = f(b,d)$:

$$(n+1)^a \cdot c^n \mod n^2 = (1+n)^b \cdot d^n \mod n^2$$

Since both $(n+1)^b$ and d^n are invertible $\mod n^2$:

$$\frac{(n+1)^a}{(n+1)^b} \cdot \frac{c^n}{d^n} \mod n^2 = 1 \mod n^2$$

$$(n+1)^{a-b} \cdot \left(\frac{c}{d}\right)^n \mod n^2 = 1 \mod n^2$$

We can raise both side to the power of $\phi(n)$, while $1^{\phi(n)} = 1$ remains unchanged:

$$(n+1)^{(a-b)\cdot\phi(n)} \cdot \left(\frac{c}{d}\right)^{n\cdot\phi(n)} \mod n^2 = 1 \mod n^2$$

Since $n \cdot \phi(n)$ is the order of $\mathbb{Z}_{n^2}^*$ and raising any element to the order of its finite group results in the neutral element:

$$(n+1)^{(a-b)\cdot\phi(n)} \mod n^2 = 1 \mod n^2$$

Now, we can use the fact that $(n+1)$ has the order $n \mod n^2$, i.e.

$$(n+1)^{(a-b)\cdot\phi(n)} \mod n^2 = 1 \mod n^2 \implies (a-b) \cdot \phi(n) \mod n = 0$$

We know, that $gcd(n, \phi(n)) = 1$, so n is not divisible by $\phi(n)$ while $a, b \in \mathbb{Z}_n$. Thus

$$(a-b) \cdot \phi(n) \mod n = 0 \implies n | ((a-b) \cdot \phi(n)) \implies a = b$$

Using this result and plugging it into the formula, we derived above, we obtain:

$$(n+1)^0 \cdot \frac{c^n}{d^n} \mod n^2 = 1 \mod n^2$$

$$c^n = d^n \mod n^2$$

Certainly, in that case both sides should also be equal modulo n. Since we also have that $gcd(n, \phi(n)) = 1$ and $|\mathbb{Z}_n^*| = \phi(n)$, the exponent n is invertible $\mod n$.

$$c^n = d^n \mod n^2 \implies c^n = d^n \mod n$$

Thus, raising both sides to the power of the inverse of n, we receive:

$$(c^n)^{n^{-1}} = (d^n)^{n^{-1}} \mod n$$

$$c^{n \cdot n^{-1}} = d^{n \cdot n^{-1}} \mod n \implies c = d \mod n$$

Since $c, d \in \mathbb{Z}_n^*$, we have $c = d$ and can conclude that f is injective, and therefore bijective for $g = n + 1$.

We can go further and show that f is even an isomorphism by showing additionally

$$\forall (a, b), (c, d) \in \mathbb{Z}_n \times \mathbb{Z}_n^* : f(a, b) \cdot f(c, d) = f(a + c \mod n, b \cdot d \mod n)$$

Clearly,

$$f(a, b) \cdot f(c, d) = ((n + 1)^a \cdot b^n) \cdot ((n + 1)^c \cdot d^n) \mod n^2 = (n + 1)^{a+c} \cdot (b \cdot d)^n \mod n^2$$

Since $(n + 1)$ has the order n modulo n^2, we can just calculate modulo n in the exponent:

$$= (n + 1)^{a+c \mod n} \cdot (b \cdot d)^n \mod n^2$$

Now we have $(a + b) \in \mathbb{Z}_n$ and it remains to show, that also $(b \cdot d) \in \mathbb{Z}_n^*$. For that, we observe that $b, d \in \mathbb{Z}_n^*$, thus both are not divisible by n and $\exists r \in \mathbb{N}\backslash\{0\}, q \in \mathbb{Z} : b \cdot d = r + q \cdot n$. It follows, that

$$(b \cdot d)^n \mod n^2 = (r + q \cdot n)^n \mod n^2 = \sum_{i=0}^{n} \binom{n}{i} r^{n-i} \cdot (q \cdot n)^i \mod n^2$$

As before, we can eliminate the addends that are multiples of n^2.

$$= r^n + n \cdot r^{n-1} \cdot (q \cdot n) \mod n^2$$

Which we can further reduce to

$$= r^n$$

Obviously, $r = b \cdot d \mod n$, and thus $(b \cdot d)^n \mod n^2 = r^n \mod n^2 = (b \cdot d \mod n)^n \mod n^2$. Finally, we conclude

$$f(a, b) \cdot f(c, d) = (n + 1)^{a+c \mod n} \cdot (b \cdot d \mod n)^n \mod n^2 = f(a + c \mod n, b \cdot d \mod n)$$

2.2.3 Definition of Paillier's Encryption Scheme

Paillier's encryption scheme is defined as $\mathcal{S}_{\text{Paillier}} = (KeyGen, Enc, Dec)$ with the algorithms:

- *KeyGen*: Randomly selects two large primes $p \neq q$ of the same binary length. Returns public/private key pair

$$(n, \lambda) \text{ using } n := p \cdot q \text{ and } \lambda := (p - 1) \cdot (q - 1)$$

- *Enc*: Let $m \in \mathbb{Z}_n$ be the plaintext to encrypt, $k_{\text{publ}} := n$ be the public key for encryption. Choose random $r \in \mathbb{Z}_n^*$. Return ciphertext

$$c := (n + 1)^m \cdot r^n \mod n^2$$

- *Dec*: Let $c \in \mathbb{Z}^*_{n^2}$ be the ciphertext to decrypt. Calculate

  $$\hat{c} := \frac{(c^\lambda \mod n^2) - 1}{n}$$

  Return plaintext

  $$m := \hat{c} \cdot \lambda^{-1} \mod n$$

Perfect correctness is given, as $Dec_{k_{\text{priv}}}(Enc_{k_{\text{publ}}}(m)) = m$:

$$\hat{c} = \frac{(c^\lambda \mod n^2) - 1}{n} = \frac{((n+1)^m \cdot r^n)^\lambda \mod n^2 - 1}{n}$$

Note, that both exponents m and n are to be understood modulo n, because $c \in \mathbb{Z}^*_{n^2} \equiv (m, r) \in \mathbb{Z}_n \times \mathbb{Z}^*_n$. Since $\lambda = \phi(n)$, and $\phi(n) = |\mathbb{Z}^*_n|$:

$$= \frac{\left((n+1)^{m \cdot \phi(n) \mod n} \cdot r^{n \cdot \phi(n) \mod n}\right) \mod n^2 - 1}{n}$$

$$= \frac{\left((n+1)^{m \cdot \phi(n) \mod n}\right) \mod n^2 - 1}{n}$$

As was shown before, $(n+1)^a \mod n^2 = 1 + a \cdot n \mod n^2$, thus:

$$= \frac{1 + (m \cdot \phi(n) \mod n) \cdot n \mod n^2 - 1}{n}$$

$$= \frac{(m \cdot \phi(n) \mod n) \cdot n \mod n^2}{n}$$

Clearly, $(m \cdot \phi(n) \mod n) \cdot n < n^2$:

$$= \frac{(m \cdot \phi(n) \mod n) \cdot n}{n}$$

$$= (m \cdot \phi(n) \mod n)$$

Finally, we calculate the plaintext by plugging in $\hat{c}$:

$$\hat{c} \cdot \lambda^{-1} \mod n = \hat{c} \cdot \phi(n)^{-1} \mod n = (m \cdot \phi(n) \cdot \phi(n)^{-1} \mod n) = m$$

2.2.4 Homomorphicity

Since f as introduced is an isomorphism $\mathbb{Z}_n \times \mathbb{Z}^*_n \leftrightarrow \mathbb{Z}^*_{n^2}$, $\mathcal{S}_{\text{Paillier}}$ has some interesting homomorphic properties. Recall, that $f(a, b) \cdot f(c, d) = f(a + b \mod n, c \cdot d \mod n)$. It follows for the two plaintexts $m_1, m_2 \in \mathbb{Z}_n$ using two integers $r_1, r_2 \in \mathbb{Z}^*_n$ randomly selected during the encryption of m_1 and m_2, respectively:

$$Enc_n((m_1)) \cdot Enc_n((m_2)) = f_{g=(n+1)}(m_1, r_1) \cdot f_{g=(n+1)}(m_2, r_2) =$$

$$f_{g=(n+1)}(m_1 + m_2 \mod n, r_1 \cdot r_2 \mod n) = Enc_n((m_1 + m_2 \mod n))$$

Where the randomness for the encryption of the sum of m_1 and m_2 in $\mathbb{Z}_n$ is $r_1 \cdot r_2 \mod n$. I.e., the product of two (or more) ciphertexts yields the ciphertext of the sum of the corresponding plaintexts modulo n. Thus, with a sufficiently large n encrypted sums of plaintexts can be derived directly from the corresponding ciphertexts without any decryption. This homomorphicity property enables us to retrieve the ciphertext of tallied votes in e-voting systems like Ordinos without decrypting the votes.

Similarly, this property can be used to multiply a plaintext by a scalar k without decrypting it by raising the ciphertext to the power of k:

$$(Enc_n(m_1))^k = f_{g=(n+1)}(m_1, r_1)^k = \underbrace{f_{g=(n+1)}(m_1, r_1) \cdot ... \cdot f_{g=(n+1)}(m_1, r_1)}_{k\text{-times}}$$

$$= f_{g=(n+1)}(\underbrace{m_1 + ... + m_1}_{k\text{-times}} \mod n, \underbrace{r_1 \cdot ... \cdot r_1}_{k\text{-times}} \mod n) = Enc_n(k \cdot m_1 \mod n)$$

2.3 Threshold Public-Key Encryption

The basic idea behind threshold encryption is to distribute the private key among a set of trustees such that at least t of them are required to participate in order to decrypt a ciphertext efficiently and successfully. A subset of less than t trustees should not be able to retrieve useful information, while the overall security should stay strong. [see, e.g., 16, 29, 33, 49] For the purpose of this thesis we use the following definition of a threshold encryption scheme, which is basically equivalent to the one used by Küsters et al. [see 33, appendix A].

Let n_{trustees} be the overall number of trustees and t a threshold. An (n_{trustees}, t)-threshold public-key encryption scheme is a tuple of 5 efficient algorithms $S_{threshold} = (KeyShareGen, PublicKeyGen, Enc, DecShare, Dec)$ where

- $KeyShareGen$ is a probabilistic algorithm run by each trustee (with index $i \in \{1, ..., n_{\text{trustees}}\}$) individually and takes n_{trustees} as an input to generate a key share pair $(k_{\text{publ\_share},i}, k_{\text{priv\_share},i})$ consisting of the public-key share $k_{\text{publ\_share},i}$ and the private-key share $k_{\text{priv\_share},i}$.
- $PublicKeyGen$ takes the public-key shares $k_{\text{publ\_share},i}$ of all trustees as input and determines an overall public key k_{publ}, if the key shares are valid.
- Enc works just like the regular probabilistic encryption algorithm of a public-key encryption scheme, i.e., takes a plaintext m and public key k_{publ} as input and outputs a ciphertext c.
- $DecShare$ is a probabilistic algorithm run by each trustee i individually and takes a ciphertext and the respective private-key share $k_{\text{priv\_share},i}$ of the trustee as an input to compute a decryption share $dec_{\text{share},i}$.
- Dec takes a set of decryption shares for a ciphertext as input and deterministically outputs the decrypted message m, if at least t valid shares were provided, fails otherwise (outputs $\perp$ in that case).

Clearly, perfect correctness needs to hold accordingly. I.e., a plaintext m decrypted by Enc under a public key generated by the use of $KeyShareGen$ and $PublicKeyGen$ to the ciphertext c has to be decrypted to the original plaintext m by the use of the corresponding private key shares in an execution of $DecShare$ and Dec on c.

An example of how these algorithms can be constructed for the Paillier encryption scheme is an essential part of this thesis.

2.4 Building blocks

2.4.1 Secret Sharing

Threshold encryption schemes are often based on *secret sharing* mechanisms, which allow for sharing a secret (e.g., a private key) among several parties. One wide spread mechanism is known as *Shamir's secret sharing*. An adaptation is secret sharing over the integers.

Shamir's Secret Sharing over Prime Fields

The idea behind Shamir's secret sharing is quite comprehensible [49]: Every polynomial p of degree k can be reconstructed efficiently from $k + 1$ data points on the graph. For example, a quadratic polynomial $p(x) = a \cdot x^2 + b \cdot x + c$ can be reconstructed from three data points. Especially for polynomials over (finite) prime fields sufficiently efficient interpolation is available.

This can be used to share a secret s among n parties as follows:

- Select a prime field, i.e., the set of integers modulo a prime number q, $\mathbb{Z}_q$, such that $s < |\mathbb{Z}_q| > n$.
- Choose a random $(n - 1)$-degree polynomial $p := \sum_{i=0}^{n-1} a_i \cdot x^i$ over $\mathbb{Z}_q$ and set $a_0 := s$, i.e., select $n - 1$ random coefficients a_i for $i \in \{1, ..., n - 1\}$. Note, that $p(0) = s$.
- For each party $i \in \{1, ..., n\}$ compute and send to the party one data point on p: $(i, p(i) \mod q)$, i.e., this data point is the i^{th} party's share of the secret.

Clearly, the secret can be reconstructed by calculating $p(0)$ when all parties cooperate in interpolation of p from their data points.

A slight adaptation allows for secret sharing with a threshold t: We can select a random $(t-1)$-degree polynomial instead of an $(n - 1)$-degree polynomial, but still generate n shares. Thus, any subset of (at least) t of the n parties is able to reconstruct the shared secret.

Secret Sharing Over Integers

A variant that allows secrets sharing over integers is described as follows [41, 45]: Assume, that $[-I, I]$ is the interval of a secret s, that is to be shared among n parties with threshold t. Then Δs is shared by the polynomial $p := \Delta s + \sum_{i=1}^{t-1} a_i \cdot x^i$. Note that Δs where $\Delta := n!$ is shared instead of s. The coefficients a_i are randomly chosen from $[-K\Delta^2 I, K\Delta^2 I]$ where $\frac{1}{K}$ is negligible. As in the original secret sharing scheme, data points on p are distributed to the parties and s can be reconstructed by interpolation. Statistical security of this variant is proven in [45].

2.4.2 Lagrange Interpolation

While both variants of threshold secret sharing might be feasible for some applications, they are not yet sufficient for a threshold public-key encryption scheme as described above. Obviously, when sharing a private decryption key via Shamir's secret sharing, arbitrary ciphertexts may be decrypted once the key was recovered. However, this is not always desirable, because the cooperating parties might want to decrypt only specific messages, while making sure, that others are kept secret. Especially for the homomorphic Paillier encryption scheme as used in Ordinos, only certain results should be decrypted, while single votes, that are encrypted under the same public key should remain undisclosed.

This can be achieved using Lagrange interpolation. Lagrange interpolation has been known since the late 18th century. Proofs can be found in the literature and are omitted here.

Knowing $t + 1$ distinct data points (x_i, y_i) for $i \in \{0, ..., t\}$ on a t-degree polynomial $p(x)$, p can be interpolated as follows:

$$p(x) = \sum_{i=0}^{t} y_i \cdot \lambda_i(x)$$

where

$$\lambda_i(x) = \prod_{j=0, j\neq i}^{t} \frac{x - x_j}{x_i - x_j}$$

Note that computations are carried out modulo q when sharing a secret over a prime field as in Shamir's secret sharing (section 2.4.1). Each party can compute an additive share of the secret $p(0)$. Clearly, $\lambda_i(0)$ can be pre-computed easily. Thus, $c^{p(0)}$ can be computed as $\prod_{i=0}^{t} c^{y_i \cdot \lambda_i(0)}$. This relation can be used to raise ciphertexts to a shared secret value and is interesting for threshold encryption schemes.

2.4.3 Commitment Schemes

Another building block of our protocol are commitment schemes. Simply speaking, commitments can be used to bind a party to the choice of a certain value without revealing the value itself. In this section we give a definition and introduce the Pedersen commitment scheme as an example, which is another building block of our protocol.

Definition

More formally, a commitment scheme consists of two algorithms:

- $GenCom(1^\kappa)$ takes a security parameter κ and generates parameters p for an instantiation of the commitment scheme.
- $Com_p(m)$ takes a message m and generates a commitment c for m using the parameters p.

Furthermore, a commitment c can be opened (and verified) by revealing m and possible randomness used by $Com_p()$. Thus, a verifier could check, that c is indeed a commitment for m.

Security

For the security of a commitment scheme, we usually require, that commitments are *computationally hiding* and *computationally binding*. We give an intuition of these requirements and refer the reader to literature for their formal definition.

- By computationally hiding we mean that an adversary should not be able to get any information about m from a commitment c generated by $Com_p(m)$ with reasonable effort.
- By computationally binding we mean that an adversary should not be able to open $c = Com_p(m)$ to an $m' \neq m$ with reasonable effort. I.e. the party that generated the commitment should be bound to open it to m only.

Pedersen commitments

For the Pedersen commitment scheme [44], $GenCom(1^\kappa)$ and $Com(m)$ are defined as follows [see, e.g., 20, 21]:

- $GenCom(1^\kappa)$: Generate a prime q of sufficient length determined by κ and find a prime p, such that $q|(p-1)$. Find two generators g, h of a subgroup $\mathcal{G}$ of $\mathbb{Z}_p^*$ such that the order of g and h is q. I.e. $g^q \mod p = 1 \wedge h^q \mod p = 1$, $\mathcal{G} = \langle g \rangle = \langle h \rangle$ and $|\mathcal{G}| = q$. The discrete logarithm $\log_g h$ must be unknown to committers. Note that such generators can be found easily, since any element in $\mathcal{G}$ except for 1 generates the group.
- $Com(m)$: Select r randomly from $\mathbb{Z}_q$. Return $c := g^m \cdot h^r \mod p$. ($c$ can be opened by revealing m and r.)

Intuitively, this scheme is computationally hiding: For any h there is $\log_g h \in \mathbb{Z}_q$. Subsequently, $c \equiv g^m \cdot g^{r \cdot \log_g h} \equiv g^{m + r \cdot \log_g h \mod q} \mod p$. Thus, with the knowledge of $\log_g h$, c could be opened to any m' by finding $r' \in \mathbb{Z}_q$ such that $m' + r' \cdot \log_g h \mod q = m + r \cdot \log_g h \mod q$. So no information about m can be deduced from c.[2] On the other hand, while $\log_g h$ is unknown to the committer, the scheme is computationally binding under the assumption that $\log_g h$ cannot be computed efficiently. It is generally believed that discrete logarithms in $\mathbb{Z}_{p'}$ where p' is prime are

[2]This makes the Pedersen commitment scheme what is called *perfectly* hiding, as even an unlimited adversary could not gain information about m from a commitment.

hard to compute. Then, an attacker could only guess a suitable r' and the likelihood of guessing correctly is negligible depending on $|\mathbb{Z}_q|$, thus the choice of q. Typically, we would use the binary length of q as security parameter κ for $GenCom()$.

2.4.4 Verifiable Secret Sharing

Using Shamir's secret sharing and Pedersen commitments, we are able to construct verifiable secret sharing (VSS) over a prime field as follows [41, 44]:

Suppose, a sender S wants to generate a secret sharing for multiple receivers R_j. S generates a secret sharing polynomial $p(x)$ as in section 2.4.1 and a random companion polynomial $p'(x)$ with random coefficients $b_i \in \mathbb{Z}_q$. I.e. $p' := \sum_{i=0}^{t-1} b_i \cdot x^i$. S then generates and publishes commitments C_i for each coefficient of $p(x)$ using the corresponding coefficient of p' as randomness: $C_i = \mathrm{Com}_{b_i}(a_i) = g^{a_i} \cdot h^{b_i} \mod p$.

Secret shares are generated as in section 2.4.1 and distributed to the receivers individually. Additionally, R_j receives the corresponding data point on p', i.e., $(j, p'(j) \mod q)$. Then R_j can verify the correctness of its share as follows:

- Compute from the public commitments: $c' = \prod_{i=0}^{t-1} (C_i)^{j^i} \mod p$
- Compute from the received shares: $c = Com_{p'(j)}(p(j))$
- If $c = c'$ the share is considered correct, otherwise the share is considered invalid.

We review the verification briefly:

$$\begin{aligned}
\prod_{i=0}^{t-1} (C_i)^{j^i} \mod p &= (C_0)^{j^0} \cdot \ldots \cdot (C_{t-1})^{j^{t-1}} \mod p \\
&= (g^{a_0} \cdot h^{b_0})^{j^0} \cdot \ldots \cdot (g^{a_{t-1}} \cdot h^{b_{t-1}})^{j^{t-1}} \mod p \\
&= (g^{a_0 \cdot j^0} \cdot h^{b_0 \cdot j^0}) \cdot \ldots \cdot (g^{a_{t-1} \cdot j^{t-1}} \cdot h^{b_{t-1} \cdot j^{t-1}}) \mod p \\
&= g^{a_0 + \ldots + a_{t-1} \cdot j^{t-1}} \cdot h^{b_0 + \ldots + b_{t-1} \cdot j^{t-1}} \mod p \\
&= g^{p(j)} \cdot h^{p'(j)} \mod p = Com_{p'(j)}(p(j))
\end{aligned}$$

A verifiable secret sharing over the integers can be constructed similarly [41, 45]. First, Δs is shared over the integers as in section 2.4.1. This results in the polynomial $p := \Delta s + \sum_{i=1}^{n-1} a_i \cdot x^i$. Additionally a random companion polynomial $p' := \sum_{i=0}^{n-1}$ is generated and commitments are published just as for VVS over prime fields. Additionally, proofs for $s \in [-I, I]$ and $a_i \in [-K\Delta^2 I, K\Delta^2 I]$ $(i \in \{1, ..., n-1\})$ are published. These can be based on the zero-knowledge range proof protocol presented in the following section. The receivers verify their shares as described above and additionally verify the provided range proofs and check the range of their share $p(j) \leq (\Delta I + tn^t K\Delta^2 I)$.

2.4.5 Zero-Knowledge Proofs

Zero-knowledge proofs are protocols that allow one party, called prover P, to prove interactively to another party, called verifier V, that a statement is true without revealing any further information. For our construction, we will need proofs for certain statements about committed values. While one party should be able to prove these statements to the others, we do not want the other parties to learn anything else about the committed value. In this section, we briefly define zero-knowledge proofs and give some protocols to prove statements about committed values.

Definition

Formally, P and V are two connected (interactive) Turing machines and we call (P, V) an interactive proof system. P wants to prove to V that $x \in \mathcal{L}$ for some language $\mathcal{L}$. At the end of the protocol the verifier decides whether he was convinced or not, by outputting "accept" or "reject". A zero-knowledge proof is a protocol between P and V with the following properties [see, e.g., 20]:

- Completeness: If the statement is true, an honest P will convince an honest V with a high probability. E.g., $\forall x \in \mathcal{L} : Pr[(P, V)(x) = \text{"accept"}] \geq \frac{2}{3}$.
- Soundness: If the statement is false, no P (whether honest or not) can convince V of the validity of the statement, except with a low probability. I.e., $\forall x \notin \mathcal{L} : Pr[(P, V)(x) = \text{"accept"}] \leq \frac{1}{3}$.[3]
- Zero-knowledge: If the statement is true, no V (honest or not) learns anything but the fact that it is true. Usually, this is shown by constructing a simulator that given some x can generate valid transcripts of a run of the protocol between an honest P and any V (honest or not) without knowing P's secret(s). Additionally, the transcripts generated by the simulator should be indistinguishable from transcripts of actual instances of the protocol. Intuitively, this means, that no knowledge about P's secret(s) can be gained from the transcripts, as they can be simulated without any knowledge.

Remark: A weaker definition includes what is known as "honest verifier zero-knowledgeness" instead of (perfect) zero-knowledgeness. This essentially means that a protocol is zero-knowledge if the verifier is an honest one. I.e., there is an honest verifier simulator that can generate valid transcripts of the communication between P and an honest V, which are indistinguishable from actual transcripts. However, this is not a problem, as we will use a method to create non-interactive protocols from zero-knowledge protocols that guarantee honesty of the verifier. Therefore, we will not always distinguish between zero-knowledgeness and honest verifier zero-knowledgeness.

Proof of Knowledge

One common application of zero-knowledge proofs is a proof of knowledge. Assume, for example, that P wants to prove knowledge of the discrete logarithm $\log_g C \in \mathbb{Z}_q$ of a number (commitment) $C \in \mathbb{Z}_p$ (parameters chosen analogous to section 2.4.3). In order to show that a zero-knowledge protocol is a proof of knowledge, we need to have additional properties [see, e.g., 3]:

[3]Note that we can reduce the soundness bound arbitrarily by repetition of proofs.

- Non-triviality: If P and V are both honest, then $\forall x \in \mathcal{L} : Pr[(P, V)(x) = \text{“accept”}] = 1$.
- Validity: There exists a knowledge extractor K, that can query P as an oracle to generate any of P’s messages during a communication with V and extract the secret(s) of P from these messages efficiently. Intuitively, that would prove, that P does indeed have the claimed knowledge.

Proof of Knowledge of Discrete Logarithm

Assume that P wants to prove knowledge of the discrete logarithm $\log_g C \in \mathbb{Z}_q$ of a number (commitment) $C \in \mathbb{Z}_p$ (parameters chosen analogous to section 2.4.3). I.e., we have $\mathcal{L} = \{C \in \mathbb{Z}_p | C = g^a \mod p \wedge a \in \mathbb{Z}_q\}$ here. P and V can run the following protocol to convince V, that P knows $a = \log_g C$ [8, 27, 47]:

The protocol is depicted in Figure 2.1. P selects a random element $r_0 \in \mathbb{Z}_q$ and sends a commitment for r_0, $C_0 = g^{r_0} \mod p$, to V. V generates a random challenge $c \in \mathbb{Z}_{2^l}$ and sends it to P, where l is a security parameter for the protocol. P replies by sending $r = r_0 - ac$, where $a = \log_g C$. Finally, V checks that $g^r \cdot C^c \mod p = C_0$. If the latter holds true, V is convinced, that P knows $\log_g C$.

This protocol is known as Schnorr’s identification protocol from [47].

Prover		**Verifier**
$r_0 \in_R \mathbb{Z}_q$	$\xrightarrow{C_0=g^{r_0} \mod p}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}_{2^l}$
$r = r_0 - ac$	$\xrightarrow{r}$	$g^r \cdot C^c \mod p \overset{?}{=} C_0$

Figure 2.1: $\pi^{\kappa}_{\text{dlog}}$: Zero-knowledge proof of knowledge of discrete logarithm

We briefly discuss, why a successful run, of $\pi^{\kappa}_{\text{dlog}}$ convinces V that P knows $\log_g C$ and review the requirements for zero-knowledge proofs of knowledge.

- Completeness (and non-triviality): If P knows $a = \log_g C$, then P will always be able to convince V, because $g^r \cdot C^c \equiv g^{r_0-ac} \cdot g^{ac} \equiv g^{r_0} \equiv C_0 \mod p$. Thus, an honest V would always accept such a proof. So we have $\forall x \in \mathcal{L} : Pr[(P, V)(x) = \text{“accept”}] = 1$.
- Soundness: If P does not know $\log_g C$, then the probability that he can still convince V is very low. The best chance is that P guesses c in advance and commits to $(C_0 :=)g^{r_0} \cdot C^c \mod p$ in the first step. Then P could reply by sending $r = r_0$ and V would verify that $g^r \cdot C^c \equiv C_0 \mod p$. The chance of guessing the correct challenge c is 2^{-l}, thus negligible in the security parameter l.
- (Honest verifier) zero-knowledge: Similarly, a simulator, that constructs valid proofs without knowledge of $\log_g C$ can be constructed. The simulator has the advantage to obtain the challenge before generating the commitment.

- Validity: It remains to discuss, why the above protocol proves knowledge. Assume, we can access P as an oracle to generate its messages during communication with some V. We can build a knowledge extractor that completes two runs of $\pi^{\kappa}_{\text{dlog}}$ with challenges c_1 and c_2 while P uses the same randomness in both runs (i.e., r_0 is the same for both runs). We denote the respective responses of P by r_1 and r_2. Then we can calculate $r_1 - r_2 = r_0 - ac_1 - (r_0 - ac_2) = (c_2 - c_1)a$. Subsequently, we can extract $a = \frac{r_1 - r_2}{c_2 - c_1}$ efficiently.

Remark: In our protocol, we use this proof of knowledge in order to prove, that a party P has committed to zero in the Pedersen commitment scheme. I.e., P committed to $C = g^0 \cdot h^r$ for a randomly chosen r. In order to prove, that C is indeed a commitment for 0, P will prove knowledge of $\log_h C$. Clearly, $C = 1 \cdot h^r$, and since $\log_g h$ is unknown to P, he cannot commit to $g^a \cdot h^r$ for some $a \in \mathbb{Z}_q \backslash \{0\}$ and prove knowledge of $\log_h C$. Thus, $\pi^{\kappa}_{\text{dlog}}$ for $\log_h C$ convinces the other parties, that C is a commitment for zero.

Zero-Knowledge Range Proof

The following zero-knowledge proof for Pedersen commitments from [42] can be used in order to prove that a committed value is in a certain range [see also 51]. Assume that P is committed to $C = g^a \cdot h^r \mod p$ and wants to prove that $a \in [A, B]$. Let $e = B - A$.

The protocol is depicted in Figure 2.2. P chooses random $t_1 \in [0, e]$, sets $t_2 = t_1 - e$ and commits to these. V sends a random bit b. Depending on b, P either opens commitments for t_1 and t_2, so that V can check correct choice of values or P sends $a + t_i$ and $r + r_i$ for an i of his choice, so that V can verify the range.

Prover		**Verifier**
$t_1 \in_R [0, e]$		
$t_2 = t_1 - e$		
$r_1, r_2 \in_R \mathbb{Z}_q$		
$T_1 = g^{t_1} \cdot h^{r_1}$		
$T_2 = g^{t_2} \cdot h^{r_2}$	$\xrightarrow{T_1, T_2 \text{ (unordered)}}$	
	$\xleftarrow{b}$	$b \in_R \{0, 1\}$
if $b = 0$: $r = (t_1, t_2, r_1, r_2)$		
else: $r = (a + t_i, r + r_i)$, s.t. $s + t_i \in [A, B]$	$\xrightarrow{r}$	if $b = 0$: $t_1 \overset{?}{\in} [0, e] \wedge t_2 \overset{?}{=} t_1 - e$
		$T_1 \overset{?}{=} g^{t_1} \cdot h^{r_1} \wedge T_2 \overset{?}{=} g^{t_2} \cdot h^{r_2}$
		else: $g^{a+t_i} \cdot h^{r+r_i} \overset{?}{\in} \{T_1 \cdot C, T_2 \cdot C\}$
		$(a + t_i) \in [A, B]$

Figure 2.2: π_{range}: Zero-knowledge range proof for committed value

Calculations and checks are done $\mod p$ as in the Pedersen commitment scheme. The proof is repeated l times to get an error probability of $\frac{1}{2^l}$. Note that π_{range} is not tight, but it ensures that $a \in [A - e, B + e]$. I.e., a malicious party can generate valid proofs, even if a is out of the desired range within certain bounds. More precisely, $A - e = A - (B - A) = A - B + A = 2A - B$ and $B + e = B + (B - A) = 2B - A$. Thus, the expansion rate of π_{range} is $\frac{(2B-A)-(2A-B)}{B-A} = \frac{3B-3A}{B-A} = 3$.

Remark: In our protocol, we will use this proof to ensure that no wrap-around $\mod q$ occurs for chosen values. Thus, q needs to be chosen large enough, so that $(2A - B), (2B - A) \in \mathbb{Z}_q$.

Zero-Knowledge Proof of Multiplicative Relation

Suppose, P is committed to a and b by $C_1 = g^a \cdot h^{r_1} \mod p$ and $C_2 = g^b \cdot h^{r_2} \mod p$. Now, P wants to create a new commitment $C_{\text{mult}} = (C_1)^b \cdot h^{r_3} \mod p$ for some $r_3 \in \mathbb{Z}_q$ and prove that C_{mult} is a commitment for the product of previously committed values a and b.

A protocol for this proof that can be derived from the multiplication protocol in [10] is depicted in Figure 2.3. P commits to a random r using randomness r_a in the underlying Pedersen commitment scheme: $C_a = g^r \cdot h^{r_a}$. Additionally, P commits to r using the group elements C_1 and h as bases: $C_b = C_1^r \cdot h^{r_b}$. V generates and sends a challenge c chosen from $\mathbb{Z}_q$. P replies by sending $z = r + bc$, $z_a = r_a + bc$, and $z_b = r_b + br_3$ (mod q respectively). Finally, V checks whether $g^z \cdot h_a^z = g^{r+cb} \cdot h^{r_a+cr_2} = g^r \cdot h^{r_a} \cdot g^{cb} \cdot h^{cr_2} = g^r \cdot h^{r_a} \cdot (g^b \cdot h^{r_2})^c = g^r \cdot h^{r_a} \cdot (C_2)^c$ and $(C_1)^z \cdot h^{z_b} = (C_1)^{r+cb} \cdot h^{r_b+cr_3} = (C_1)^r \cdot (C_1)^{cb} \cdot h^{r_b} \cdot h^{cr_3} = (C_1)^r \cdot g^{abc} \cdot h^{bcr_1} \cdot h^{r_b} \cdot h^{cr_3}$ $= C_b \cdot (g^{ab} \cdot h^{br_1} \cdot h^{r_3})^c = C_b \cdot ((C_1)^b \cdot h^{r_3})^c = C_b \cdot (C_{\text{mult}})^c$ hold true.

Prover		**Verifier**
$r, r_a, r_b \in_R \mathbb{Z}_q$		
$C_a = g^r \cdot h^{r_a}$		
$C_b = {C_1}^r \cdot h^{r_b}$	$\xrightarrow{C_a, C_b}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}_q$
$z = r + cb \mod q$		
$z_a = r_a + cr_2 \mod q$		
$z_b = r_b + cr_3 \mod q$	$\xrightarrow{z, z_a, z_b}$	$g^z \cdot h^{z_a} \stackrel{?}{=} C_a \cdot (C_2)^c$
		$(C_1)^z \cdot h^{z_b} \stackrel{?}{=} C_b \cdot (C_{\text{mult}})^c$

Figure 2.3: π_{mrel}: Zero-knowledge proof of multiplicative relation

Calculations and checks are done mod p as in the Pedersen commitment scheme. This proof convinces V, that C_{mult} is of the form $(C_1)^b \cdot h^{r_3} = g^{a \cdot b} \cdot h^{r_1 b + r_3} \mod p$ and thus proves multiplicative relation between C_1 and C_2. The soundness error of π_{mrel} is $\frac{1}{q}$ and can be decreased by repetition.

Zero-Knowledge Proof for Verification Keys

Nishide and Sakurai provided a proof for generating verification keys in appendix B of [41]. The goal of this proof is to generate verification keys in order to prove correctness in the decryption process. Assume, that P is committed to x by $C = g^x \cdot h^y \mod p$ (a Pedersen commitment as used before). Now, P and V know the public element v of a group of squares mod N^2. I.e., $v \in \{a^2 \mod N^2 | a \in \mathbb{Z}_{N^2}\}$. P wants commit to $C' = v^x$ and prove that both C and C' are commitments for the same x, i.e., $\log_g C/h^y = \log_v C'$. For our purpose we need an additional integrated range proof. This is because q, the order of $\mathcal{G}$ in our instance of the Pedersen commitment scheme, is public. Thus, we need to prevent P from fraudulently using v^{x+iq} for some i instead of C'. Actually, we will use an upper bound T for x, that depends on the context of the proof and the embedded range proof works for $|x| < T$.

A protocol for this proof is depicted in Figure 2.4 [see 41, appendix B].

Prover		**Verifier**
$r \in_R [0, 2^l T]$		
$r' \in_R \mathbb{Z}_q$		
$R_1 = g^r \cdot h^{r'} \mod p$		
$R_2 = v^r \mod N^2$	$\xrightarrow{R_1, R_2}$	
	$\xleftarrow{c}$	$c \in_R \{0, 1\}$
$z = r + cx$		
$z' = r' + cx \mod p$	$\xrightarrow{r}$	$g^z \cdot h^{z'} \overset{?}{=} R_1 \cdot C^c \mod p$
		$v^z \overset{?}{=} R_2 \cdot (C')^c \mod N^2$
		$z \overset{?}{\in} [0, 2^l T]$

Figure 2.4: π_{VKeq}: Zero-knowledge proof for verification keys

This proof is executed h times in parallel, i.e., P commits to h random values at once, V generates h random challenge bits and receives the respective responses from P. It convinces V that $x \in [-2^l T, 2^l T]$ additionally to the equality of the aforementioned logarithms. The completeness bound is $1 - \frac{1}{2^l}$ and the soundness bound is $\frac{1}{2^h}$. In order to ensure that no wrap-around mod q occurs, we need $q > 2^{l+1} T$.

Zero-Knowledge Proof for Partial Decryption Share

We need another zero-knowledge proof in order to check decryption shares when decrypting Paillier ciphertexts in our threshold encryption scheme. The idea is that every party P_i raises the ciphertext c to its secret key share $k_{\text{priv\_share},i}$ as hinted in section 2.4.2 in order to compute $c^{k_{\text{priv}}}$ where k_{priv} is the overall secret key. For this, we want P_i to prove that the exponentiation was conducted correctly.

We assume that P is committed to $C = v^{\Delta x}$, computes $C' = c^{2\Delta x} \mod N^2$ and needs to prove $\log_{c^{4\Delta}}(C')^2 = \log_{v^{\Delta}} C$. Here, $c \in \mathbb{Z}_{N^2}$ is the ciphertext and Δ is a constant of our protocol (more precisely $\Delta = n_{\text{trustees}}!$). As in π_{VKeq}, we have $|x| < T$.

A protocol of the proof provided in appendix C of [41] is depicted in Figure 2.5. l and t are the security parameters of the protocol and need to be chosen such that $\frac{1}{2^l}$ and $\frac{1}{2^t}$ are negligible.

Prover		**Verifier**
$r \in_R [0, 2^{t+l}T]$		
$R_1 = v^{\Delta r} \mod N^2$		
$R_2 = c^{4\Delta r} \mod N^2$	$\xrightarrow{R_1, R_2}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}_{2^t}$
$z = r - cx$	$\xrightarrow{z}$	$v^{\Delta z} \overset{?}{=} R_1 \cdot C^c \mod N^2$
		$c^{4\Delta z} \overset{?}{=} R_2 \cdot (C')^{2c} \mod N^2$

Figure 2.5: π_{partDec}: Zero-knowledge proof for partial decryption shares

Zero-Knowledge Proof of Equality of Commitments

For an adaptation of Nishide and Sakurai's protocol, we need to change the instance of the Pedersen commitment scheme during key generation. Since we want to ensure that the parties continue using the same values, we need to have a zero-knowledge proof for equality of committed values in two instances of the Pedersen commitment scheme.

Assume that P is committed to $C = g_1^x \cdot h_1^{r_1} \mod p_1$ and generated a commitment $C' = g_2^x \cdot h_2^{r_2} \mod p_2$. g_1 is a generator for the subgroup $\mathcal{G}_1$ of order q_1 in $\mathbb{Z}_{p_1}^*$ and g_2 generates $\mathcal{G}_2$ in $\mathbb{Z}_{p_2}^*$ analogously. I.e., $\langle g_1 \rangle = \mathcal{G}_1$; $\langle g_2 \rangle = \mathcal{G}_2$; $|\mathcal{G}_1| = q_1$; $|\mathcal{G}_2| = q_2$. h_1 and h_2 are chosen as required for Pedersen commitments, i.e., $\log_{g_1} h_1$ and $\log_{g_2} h_2$ remain unknown to P and V.

A proof that two commitments hide the same secret can be found in [6, 37]. However, their proof is meant for g_1, g_2, h_1, and h_2 that generate the same subgroup. I.e., they assume $p_1 = p_2$ and $q_1 = q_2$.

We use a very similar proof, but as we have $q_2 \gg q_1$ (w.l.o.g.), we need an additional range proof in order to prevent a wrap-around mod q_1 in C'. The protocol for the proof of equality of committed values is depicted in Figure 2.6.[4] As before, l and t are security parameters and $\frac{1}{2^l}$ and $\frac{1}{2^t}$ should be negligible.

$z \in \mathbb{Z}_{2^{l+t}q_1}$ holds with overwhelming probability which leads to a high completeness bound. Clearly, the soundness bound is 2^{-t} with a similar argumentation as in section 2.4.5. Since the range for r is much larger than that for cx, z statistically hides x. π_{comEq} convinces V that C' has the form

[4]Thanks to Takashi Nishide for sharing a memo about his idea how this proof can be realized. The memo is included in Appendix A.

Prover		**Verifier**
$r \in_R \mathbb{Z}_{2^{l+t}q_1}$		
$r_a \in_R \mathbb{Z}_{q_1}$		
$r_b \in_R \mathbb{Z}_{q_2}$		
$R_a = g_1^r \cdot h_1^{r_a} \mod p_1$		
$R_b = g_2^r \cdot h_2^{r_b} \mod p_2$	$\xrightarrow{R_a, R_b}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}_{2^t}$
$z = r + cx$		
$z_a = r_a + cr_1 \mod q_1$		
$z_b = r_b + cr_2 \mod q_2$	$\xrightarrow{z, z_a, z_b}$	$g_1^z \cdot h_1^{z_a} \overset{?}{=} R_a \cdot C^c$
		$g_2^z \cdot h_2^{z_b} \overset{?}{=} R_b \cdot (C')^c$
		$z \overset{?}{\in} \mathbb{Z}_{2^{l+t}q_1}$

Figure 2.6: π_{comEq}: Zero-knowledge of equality of committed values

$g_2^{x+iq_1} \cdot h_2^{r_2}$ for some i. We note again, that in order to prove that C' actually commits to x (and not to $x + iq_1$ for $i \neq 0$), P needs to provide a range proof for C', that convinces V that C' is a commitment for some value below q_1. Depending on the context, π_{range} with expansion rate 3 may suffice.

2.4.6 Random Oracle Model

Random oracles are used in theory to generate random hash values for messages. A random oracle can be imagined as a server, that receives messages and replies by sending random values for these messages. If the message has never been queried, the random value is generated freshly and stored for later requests. Otherwise, if the message was queried before, the previously stored response is sent again.

Thus, random oracles model the ideal hash function, that generates truly random and distinct hash values for each input. It is common to use cryptographic hash functions as implementations of random oracles. However, constructions proven secure in the random oracle model are not always secure, when implemented using hash functions [see, e.g., 9].

For our protocol, a random oracle is assumed and implemented by a hash function in order to generate common random values among the parties. We note, that the usage of the hash function could be replaced by a conjoint computation of randomness by the parties, e.g., by computing the sum of random values generated by each party. [see also remarks in 41]

2.4.7 Non-Interactive Zero-Knowledge Proofs

From the proofs presented in section 2.4.5 we notice that each one has an interactive 3-move form consisting of

1. P sending commitments a to V

2. V sending a challenge c to P
3. P sending a reply z based on the value of c

In literature this is known as the Σ-form [see, e.g., 14, 28]. A technique to transform such protocols into non-interactive protocols was first used by Fiat and Shamir in [22] and is known as Fiat-Shamir transform or Fiat-Shamir heuristic.

Fiat-Shamir transform is based on the random oracle model. Assume that P and V have access to a random oracle that generates random values of the length required for the challenge in the respective proof. We denote the function to retrieve a value for message m from the random oracle as $h(m)$.

The idea is to omit the 2^{nd} step of the protocol by generating c as $c := h(a)$. Thus, P can generate V's challenge by himself without previous communication. However, the challenge is not arbitrary; the random oracle merely replaces an honest verifier that chooses a random challenge. Subsequently, the protocol collapses into one step of communication, where P sends (a, c, z). V conducts its checks as in the original protocol and additionally checks correctness of the received c by comparison with $h(a)$.

The transform seems to give P more power, since P can generate commitments a and challenges c without participation of V. So, P might try to generate values until he is able to respond correctly in order to construct a valid proof even though the statement might not hold. However, as long as the domain for c (thus the range of $h()$) is very large, this is not an efficient strategy. [see 14]

In order to avoid an overhead of communication in our protocol, we use Fiat-Shamir transform to make the zero-knowledge proofs non-interactive. This also enables us to use honest verifier zero-knowledge protocols and gain security in the random oracle model, since the randomness of the challenge (and thus honesty of the verifier) can be assumed. For that purpose, we use a cryptographic hash function that is believed to be secure with respect to generating unpredictable random output. Note that if we have a hash function H with domain $\mathbb{Z}_a$ and we want to generate smaller hashes of length $b < a$, we can use $h(m) := H(m) \mod b$ under the assumption that H generates uniformly distributed random values and $b \mid a$. In case $b \nmid a$, but $a \gg b$ such that $\frac{a \mod b}{a}$ is negligible, this works as well without significantly influencing the uniform distribution.[5]

[5]For example, when using a hash function that generates 512-bit output to generate numbers in $\mathbb{Z}_{100}$, the numbers $[0, ..., 95]$ are slightly more likely to occur since $2^{512} - 1 \mod 100 = 95$. However, the additional probability for their occurrence is $\frac{96}{2^{512}}$, which is negligibly small.

3 Related Work

Secure distributed generation of key shares for a threshold Paillier encryption scheme is closely related to distributed key generation for threshold RSA. Both schemes are based on so called *RSA moduli*, that are products of two large and distinct primes. However, unlike threshold variants of RSA, a threshold variant of Paillier's encryption scheme is a bit more complicated due to the nature of the decryption algorithm. The original decryption algorithm requires not only to raise the ciphertext to the power of the private key (as for RSA), but also dividing it by the private key in an additional step. Thus, special care needs to be taken while constructing shares of the private key. Threshold variants of Paillier's encryption schemes were proposed by several authors [e.g., 17, 23] soon after its emergence [43] in 1999.

A secure distributed key generation for threshold Paillier in particular has been addressed by Nishide and Sakurai [41] and Hazay et al. [29] in 2010 and 2011, respectively. To the best of our knowledge these are the only two complete proposals for that purpose. Nishide and Sakurai's protocol is based on verifiable secret sharing (VSS) with Pedersen commitments. Using zero-knowledge proofs for committed values, the parties prove in every step of the protocol that they behaved correctly. In Hazay et al.'s protocol the parties use a shared key for exponential ElGamal for joint computations and engage in two-party protocols based on Paillier for each pair of parties. Zero-knowledge proofs are used to ensure that encrypted values are chosen and used correctly in each step. The main difference of both proposals is that Nishide and Sakurai's protocol is designed to be secure in the malicious setting with an honest majority, while Hazay et al. present a solution secure against a dishonest majority[1]. Furthermore, Nishide and Sakurai's protocol is limited to generating key shares for an (n, t)-threshold scheme with the restriction $t - 1 < \frac{n}{2}$, which is due to a sub-protocol for secure multiplication (n be the number of parties and t the threshold).

Implementations of adaptations of both protocols for the semi-honest setting were presented and evaluated by Nicolosi [39] for [29] and Veugen et al. [52] for [41]. Both implementations came with respective improvements of parts of the protocols. We compare these implementations in Table 3.1. The table presents parameters and average runtime of the protocols as presented in [39], [29], and [52]. Numbers are given for runs that generate a 2048-bit modulus N. The runs of [52] cover the whole key setup whereas those of [39] merely focus on generating RSA moduli. However, the numbers are still comparable, since RSA modulus generation is the most time-consuming part of the protocols. For both protocols (network) communication was not considered. Note that [29] was implemented for two-party computation, which simplifies some steps of the protocol. Especially, computing the RSA modulus is significantly different from the general multi-party case. [52] was executed among three parties for the evaluation. Due to construction of [41] for the general multi-party case, the actual RSA modulus is slightly larger than the desired 2048 bits. In

[1]More precisely, [29] is even secure against corruption of all but one of the parties.

Proposal	Hazay et al. [29]	Nishide and Sakurai [41]
Setting (Proposal)	Dishonest majority	Honest majority
Setting (Implementation)	Semi-honest, two-party (see Nicolosi [39] and [29])	Semi-honest, multi-party (see Veugen et al. [52])
Parties (in tests)	2	3
Hardware	Intel Core i5 dual core 2.3 GHz (256KB L2 cache per core, 3 MB L3 cache) 8GB RAM	2.4 GHz single-core CPU 8GB RAM
Threads	unknown	1
Runs	10	1097
Average runtime 2048-bit composite	900s	66s
Limitations/ Comparability	Network communication omitted Only RSA modulus generation	Network communication omitted Modulus size is actually $\lvert N_b \rvert > 2(1024 + \log_2(3)) > 2048$

Table 3.1: Comparison of existing implementations

[41] we have $k(= 1024)$ as a parameter, that determines the binary length of additive shares of the primes composing N. Evidently, the observed runtimes for the protocol of Nishide and Sakurai were significantly lower than those for Hazay et al.'s protocol.

Implementations for threshold RSA key generation with security in the semi-honest setting delivered similar results for 2048-bit moduli: Malkin et al. [35] measured an average 1088 seconds in a three party computation in 1999. Nguyen [38] measured an average of 283 seconds in a three party computation in 2005. Frederiksen et al. [24] recently reported an impressive estimated runtime of 42 seconds[2] for their two-party protocol with security in the *malicious* setting. However, it remains unknown how to extend the latter for the general multi-party case.

For the purpose of this thesis we focus on [41] for several reasons: Since we have to expect significant additional runtime in an implementation for the malicious setting, their protocol is more promising to generate a key sharing with an acceptable runtime. Even though there is the limitation $t - 1 < \frac{n}{2}$ for the threshold t depending on the number of parties n, it is still suitable for many cases. For an odd n, we can have a t consisting of a majority of the participants. E.g., the protocol is perfectly usable for the case $(n, t) = (3, 2)$, which is very interesting for demonstration purposes as well as for possible real world applications. Regarding the security of [29] against an adversary that corrupts all but one party, we note that it depends on the concrete setup and chosen threshold, whether that kind of (costly) security is actually meaningful. Even though we may have security against up to $n_c \leq n - 1$ corrupted parties for the key generation, the threshold might be lower than n_c, which would enable an adversary to gain knowledge of the overall private key anyway. This applies similarly to [41]. However, it seems reasonable for our case to trade off some (possibly ineffective) security for more efficiency.

[2]35 seconds measured when using 8 threads + 7 seconds estimated for computations in garbled circuits that were not implemented.

4 Secure Distributed Paillier Key Generation

This chapter introduces Nishide and Sakurai's protocol for distributed Paillier key generation [41]. For this, we first review the steps of local key generation and then transform them to an MPC protocol for the semi-honest setting. Finally, we derive the protocol for the honest-majority setting.

4.1 Local Key Generation

Key generation for the general Paillier encryption scheme (without a threshold) was briefly mentioned for the $KeyGen$ algorithm in section 2.2.3. The setup of the public key is essentially the same for a threshold Paillier cryptosystem. However, constructing private key shares is not as straightforward.

We examine the steps of $KeyShareGen$, which is essentially consistent with the description in section 2.3, except that it is run by one single trusted party T. The input for $KeyShareGen$ is the binary length k of the desired public key as a security parameter, the number n of parties who share the private key and the desired threshold t. As in [41] we follow the approach of [23], except that we apply Nishide and Sakurai's assumption for avoiding safe primes [see also 52].[1]

1. T selects two primes p and q of binary length $\frac{k}{2}$. This can be done efficiently by choosing random numbers of the desired length until they pass a prime test. A polynomial time prime test known as AKS test was presented in [2]. A probabilistic prime test with a very low probability of accepting a non-prime number may suffice. Due to the prime number theorem, the number of primes that are lower or equal to x is roughly $\pi(x) \approx \frac{x}{\ln x}$. This signifies, that approximately every $(\ln x)^{\text{th}}$ number lower than x is prime. For $k = 2048$ this means, that the probability that a random number is prime is around $\frac{1}{710}$. Thus, our probabilistic algorithm will be able to find 2 primes on average after ≈ 1420 trials. When excluding even numbers from the beginning, only half as many trials are necessary. The public key is composed as $N := pq$.
2. Select a random element $\beta \in_R \mathbb{Z}_N^*$. Note that $\phi(N) = (p-1)(q-1)$. A secret sharing of $\beta\phi(N)$ over $\mathbb{Z}_{\phi(N^2)}$ with threshold t is created as in section 2.4.1. We denote the resulting polynomial as $f(x) := \beta\phi(N) + \sum_{i=1}^{t-1} a_i \cdot x^i$. ($\beta\phi(N)$ is the private key blinded by β.) The resulting private key shares for n parties P_i are $k_{\text{priv\_share},i} = f(i)$ for $i \in \{1, ..., n\}$.
3. $\theta = \beta\phi(N) \mod N$ is computed and appended to the public key.

[1] A prime p is called safe prime, if there exists a prime p' such that $p = 2p' + 1$. The original protocol of [23] is based on safe primes. However, Nishide and Sakurai showed how to relax that requirement [see 41] based on [19].

The public key consisting of N and θ is published. The private key share $k_{\text{priv\_share},i}$ is distributed to the respective party P_i. The encryption algorithm is identical to the one for the regular Paillier scheme (i.e., using only N, see section 2.2.3).

A ciphertext c is decrypted as follows:

1. t parties need to participate. We denote the set of their indices as S. $|S| = t$. Each party P_i publishes $c_i = c^{k_{\text{priv\_share},i}} \mod N^2$ for $i \in S$ as their decryption share.
2. $c^{\beta\phi(n)} \mod N^2$ is reconstructed from the decryption shares using Lagrange interpolation as in section 2.4.2. I.e., $c^{\beta\phi(n)} \mod N^2 = \prod_{i \in S} c_i^{\lambda_i(0)} \mod N^2$.
3. Finally, m can be reconstructed by using the public value θ: $m = \frac{c^{\beta\phi(N)} \mod N^2 - 1}{N} \cdot \theta^{-1} \mod N$, where θ^{-1} is the multiplicative inverse of θ in $\mathbb{Z}_N$

Note that due to the blinding of $\phi(N)$ by multiplication with random β, θ does not leak $\phi(N)$. The factor β of the private key becomes a factor of the intermediate plaintext during decryption and is eliminated by multiplication with θ^{-1}.

4.2 Distributed Key Generation with Semi-Honest Security

4.2.1 Overview

As a first step, we describe a distributed protocol to set up a key sharing. Each party P_i has to follow these steps in order to generate key shares [see 41, 52]:

1. The parties agree on a random large prime P'.
2. P_i generates two random numbers p_i and q_i of binary length $\frac{k}{2} - 1$. Then a secret sharing with threshold t for each number is generated in $\mathbb{Z}_{P'}$ as in section 2.4.1. We denote the resulting $(t-1)$-degree polynomials as $f_i(x)$ for p_i and $g_i(x)$ for q_i.
3. P_i generates an additional sharing of 0 over $\mathbb{Z}_{P'}$ with threshold $2t - 1$ resulting in the random polynomial $h_i(x)$ of degree $2(t-1)$ where $h_i(0) = 0$. The random polynomial h_i is necessary for randomization in the multiplication protocol of [4] (BGW protocol). Ben-Or et al. noted that the resulting polynomial without h_i in the following multiplication is not completely random. E.g., it is definitely not irreducible. This is fixed by adding the random polynomial h_i to the protocol. P_i sends the shares $\langle f_i(j), g_i(j), h_i(j)\rangle$ to P_j for each $j \in \{1, \dots, n\} \setminus \{i\}$.
4. From the received shares and $\langle f_i(i), g_i(i), h_i(i)\rangle$, P_i computes $\sum_{j=1}^{n} f_j(i)$, $\sum_{j=1}^{n} g_j(i)$, and $\sum_{j=1}^{n} h_j(i)$, i.e., sharings of $p = \sum_{i=1}^{n} p_i$ and q analogously as well as an overall random sharing of 0. Note that $f_\Sigma(x) = \sum_{j=1}^{n} f_j(x)$ is essentially a $(t-1)$-degree polynomial with $f_\Sigma(0) = \sum_{j=1}^{n} p_i$. From these sharings the parties can compute shares of a candidate for N (denoted as N_{cand}): $N_i = \sum_{j=1}^{n} f(j) \cdot \sum_{j=1}^{n} g(j) + \sum_{j=1}^{n} h_j(i)$. N_i is now a data point on a $(2 \cdot (t-1))$-degree polynomial that is shared among the parties and $N_{\text{cand}} = pq$ is the shared secret of that polynomial. These are the essential steps of the BGW protocol for multiplication. Each party publishes its share N_i.
5. From $2t - 1$ of these shares P_i can compute N_{cand} using Lagrange interpolation as described in section 2.4.2.

6. After N_{cand} was computed, the parties test for biprimality of N_{cand}. More details for these tests are given in section 4.2.2. If N_{cand} turns out not to be the product of two primes, then the protocol is started anew in order to generate a new candidate for N.
7. Once a candidate passed biprimality tests, i.e., a suitable N is found, the parties have an additive sharing of N's prime factors. The parties inherently have a sharing of $\phi(N)$ since $\phi(N) = (p-1)(q-1) = pq - p - q + 1 = N - p - q + 1$. Hence, the share of P_i is $f_\phi(i) = N - \sum_{j=1}^{n} f_j(i) - \sum_{j=1}^{n} g_j(i) + 1 \mod P'$.
8. Each P_i chooses a random $\beta_i \in \mathbb{Z}_N$ and $R_i \in \mathbb{Z}_{KN}$. Where K is chosen as required for secret sharing over integers (see section 2.4.1). β_i and R_i are additive shares of β and R. Similar to the computation of N, the parties compute $\theta = \Delta\phi(N)\beta + N\Delta R + 0 \mod P'$, where 0 is shared by a random polynomial. Since R is shared over the integers Δ appears as a factor in front of R as in section 2.4.1.
9. From the public value θ, P_i computes its share of $-\Delta\phi(N)\beta$ as $f_{\text{priv\_key}}(i) = N\Delta R_i - \theta$. Note that the private key is shared over the integers, while θ is calculated mod P'. $\theta \mod N$ is appended to the public key.

In order to correctly compute θ and N in $\mathbb{Z}_{P'}$ the prime P' needs to be chosen large enough. I.e., $P' > N \land P' > \theta$. The maximum values of N and θ are denoted as $N_{\max}$ and $\theta_{\max}$. Due to the choice of p_i and q_i we have $N_{\max} = p_{\max}^2 < (n \cdot 2^{\frac{k}{2}})^2$. $\theta_{\max}$ directly depends on $N_{\max}$ and is bounded by $2n\Delta(1+K)N_{\max}^2$. In order to prevent a wrap-around mod P', $P' > 2\theta_{\max}$ should hold. [41]

A ciphertext c is decrypted as follows (similarly to section 4.1):

1. Again, participation of t parties is required and every party raises c to its private key share.
2. $\hat{c} = c^{-\Delta\phi(N)\beta}$ is computed using Lagrange interpolation.
3. $m = \frac{c^{-\Delta\beta\phi(N)} \mod N^2 - 1}{N} \cdot (-\theta)^{-1} \mod N$

4.2.2 Biprimality Test

Biprimality tests play an important role in the protocol. The parties need to ensure, that N consists of exactly 2 large prime factors in order to have security based on the factorization problem and a working Paillier encryption scheme. Regular primality tests for the factors of N do not qualify, since none of the parties can learn p or q. A distributed biprimality test was presented by Boneh and Franklin [5]. While a biprime N is definitely accepted by their test, the test rejects non-biprime N with a probability $> \frac{1}{2}$. Thus, the test needs to be repeated in order to achieve a good confidence that N is in deed the product of 2 primes. The repetitions lower the efficiency of the test. Therefore, some simpler tests, that check for small prime factors may sort out numerous candidates N_{cand} that are not biprime. This section covers Boneh and Franklin's distributed biprimality test as well as two kinds of small prime test.

Distributed Biprimality Test

The biprimality test in [5] requires $p \equiv q \equiv 3 \mod 4$. This can be achieved by having the parties select their shares p_i such that $p_i \mod 4 = 0$ except P_1 chooses $p_1 \mod 4 = 3$. Analogously $q \mod 4 = 3$ is ensured.

The test is based on the Jacobi symbol. Note that Jacobi symbols can be computed efficiently [see, e.g., 36]. The parties run the following protocol [see also 41, 52]:

1. The parties agree on a random $g \in_R \mathbb{Z}_N$ such that the Jacobi symbol of g is $(\frac{g}{N}) = 1$.
2. P_1 computes and publishes $Q_1 = g^{\frac{N-p_1-q_1+1}{4}} \mod N$, all other parties publish $Q_i = g^{\frac{p_i+q_i}{4}} \mod N$.
3. If $Q_1 \equiv \pm \prod_{i \neq 1} Q_i \mod N$ does not hold, N is considered not to be composed of 2 primes.

This test is repeated with several different g, until the probability of N non-biprime is sufficiently low. More precisely, the probability that a non-biprime N passes this test after κ repetitions is at most $\frac{1}{2^\kappa}$.

Observe that $\frac{Q_1}{\prod_{i \neq 1} Q_i} = g^{\frac{N-p-q+1}{4}} = g^{\frac{\phi(N)}{4}}$. Furthermore, $g^{\frac{\phi(N)}{4}} = g^{\frac{p-1}{2} \cdot \frac{q-1}{2}}$. Now suppose that p and q are prime, then we have due to Euler's criterion: $g^{\frac{p-1}{2}} = (\frac{g}{p}) \mod p = \pm 1$ (analogously for q). Since $p \equiv q \equiv 3 \mod 4$ we have that $\frac{p-1}{2}$ is odd (same for q). Thus, $(g^{\frac{p-1}{2}})^{\frac{q-1}{2}} = (\frac{g}{p})$ (analogously for q). So both Jacobi symbols are equal and $(\frac{g}{N}) = (\frac{g}{p}) \cdot (\frac{g}{q}) = 1$. Subsequently, $g^{\frac{\phi(N)}{4}} \mod N = \pm 1$ and $Q_1 \equiv \pm \prod_{i \neq 1} Q_i \mod N$.

Boneh and Franklin showed that the probability of a non-biprime N to pass the test is upper bounded by $\frac{1}{2}$ [5].

Distributed Trial Division

In order to reduce the number of relatively costly biprimality tests, Nishide and Sakurai suggested to conduct trial divisions for p and q prior to the computation of N in order to sort out unusable numbers. Essentially, they test in a distributed manner whether p or q is a multiple of a small prime factor $p' \geq 3$. A semi-honestly secure version of their protocol was derived by Veugen et al. [see 52, appendix]:

1. Each party P_i chooses a random $r_i \in_R \mathbb{Z}_{p'}$, so they share $r = \sum_{i=1}^{n} r_i$.
2. P_i generates a secret sharing mod p' with threshold t of q_i and r_i and distribute the shares to the respective parties.
3. P_i uses the received shares and its own shares in order to compute its share of $r \cdot q \mod p'$. The resulting sharing is based on a $(2(t-1))$-degree polynomial.
4. The parties publish $2t - 1$ of their shares in order to retrieve $r \cdot q \mod p'$. If $r \cdot q \mod p'$ is zero, then either r or q is a multiple of p'.

The test is repeated several times. If the result is 0 in every run, the probability that p' divides q is high depending on the number of repetitions. The test is run for a number of small primes lower than some threshold B. Working with a random multiple $r \cdot q \mod p'$ is necessary to prevent leaking $q \mod p'$. Unlike the other tests in this section, this one should be executed right after the parties chose p_i and q_i. If a prime $p' \in [3, B]$ divides p or q, then the whole protocol is restarted.

Local Trial Division[2]

Another way to check primality is trying find factors of N after its computation. Intuitively, since N is meant to be the product of two large primes, it should be hard find divisors of N. If a divisor is found, then either p and q can be derived efficiently, which breaks security, or – much more likely – N is not the product of two large primes. The advantage is that local trial divisions can be conducted without any interaction once N was computed.

Hazay et al. and Veugen et al. suggest to use local trial divisions in combination with [see 29] or instead of [see 52] distributed trial divisions. The idea is simply that each party checks $N \mod p' \neq 0$ locally for all primes lower than or equal to some threshold B. As before, once a divisor of N was found, N is discarded and the protocol is restarted. In [29] the authors decided to use distributed trial divisions only for very small prime numbers lower than B_1 and continue with local trial divisions for primes in range $[B_1, B_2]$. On the other hand the authors of [52] found that for their protocol it is more efficient to sort out unsuitable candidates using only local trial divisions.

4.2.3 Efficiency

Regarding efficiency of the presented protocol, the most important question is: How often do the parties need to choose p_i and q_i in order to generate a suitable N? Apart from that, all steps of the protocol can clearly be implemented with sufficient efficiency. Since we have a probabilistic protocol, the question can be expressed as: How many trials does it need on average to find additive sharings of primes that compose a suitable N? As the parties cannot check for primality of p and q individually, we need to consider the probability that 2 numbers that were randomly chosen in one single shot are both prime.

Based on the prime number theorem Hazay et al. calculated that a biprime 2048-bit modulus will be generated after an average of $\left(\frac{\ln 2^{1024}}{2}\right)^2 \approx 126\,000$ trials [29]. This is because we require $p \equiv q \equiv 3 \mod 4$, which sorts out even candidates from the very beginning. In deed Veugen et al. [52] measured an average of $\approx 132\,000$ iterations for generating a biprime N in their evaluation, which supports the estimations.

Furthermore, we follow the analysis of [5] based on [7] regarding the number of biprimality tests. According to this analysis the probability that an n-bit prime candidate p is actually a prime after passing trial divisions up to a threshold B is

$$Pr[p \text{ prime} | p \text{ passed trial div up to B}] \approx 2.57 \cdot \frac{\ln B}{n} \left(1 + o\left(\frac{1}{n}\right)\right)$$

[2]Note that local trial division was not mentioned in [41], but used by both Hazay et al. and Veugen et al. in their implementations.

For 1024-bit primes and $B = 20\,000$ this is $\approx \frac{1}{40}$. I.e., an approximate $40^2 = 1\,600$ biprimality tests need to be conducted until a suitable N is found. Veugen et al. measured a similar number in their experiments (1 692 failed biprimality tests on average).

4.2.4 Security

Security of the above protocol is ensured as long as all parties adhere to the protocol. I.e., the protocol is secure in the semi-honest setting. During the generation of N security is based on the security of Shamir's secret sharing. For the biprimality test, security is based on the discrete logarithm assumption. Nishide and Sakurai show that their construction of θ does not leak information about $\phi(N)$. Essentially, this is the case because θ is known mod P' and mod N. In the mod P' case a random multiple of N hides the value of $\Delta\phi(N)\beta$. On the other hand $\theta \mod N = \Delta\phi(N)\beta \mod N$ does not leak information about $\phi(N)$ either. For the detailed security proof refer to [41][3].

4.3 Robust Distributed Key Generation

This section introduces the protocol for distributed key generation as proposed by Nishide and Sakurai [41]. This protocol is secure against malicious adversaries who may control up to $t < \frac{n}{2}$ parties and arbitrarily deviate from the protocol. The basic mechanism is the same as in section 4.2. The idea is to ensure that every party adheres to the protocol and malicious behavior is detected by the honest parties, thus making the protocol robust against such behavior. For this, we employ (among other measures) the zero-knowledge proofs presented in section 2.4.5.

Again, let n be the number of parties, t be the desired threshold and k the desired bit length of N. We denote the prime length $\frac{k}{2}$ as τ.

4.3.1 Instantiation of the Pedersen Commitment Scheme

1. The parties agree on a suitable large prime P'. (Requirements for P' are developed throughout the presentation of the overall protocol. See also section 4.2.)
2. They compute the lowest prime P that fulfills the requirements for the Pedersen commitment scheme (see section 2.4.3), i.e., $P'|(P-1)$. This can be done by iterating over multiples of P' until the lowest i is found such that $i \cdot P' + 1$ is prime.
3. Generators g and h are chosen such that $\log_g h$ remains unknown. It is necessary that the parties agree on the same generators. g can be chosen by letting P_1 generate random candidates until $g^{P'} \mod P = 1$ and distribute g to the other parties, who check that g is a valid generator. h is another generator of the same group. However, since $\log_g h$ shall remain unknown it needs to be chosen differently to prevent the parties from cheating. A suitable protocol for choosing an anonymous shared secret is presented in [44]. Note that despite

[3]The full version of their paper with a detailed security proof is completed, but yet to be published. Thanks to Takashi Nishide for sharing it beforehand.

the attack against the protocol presented in [25], it is still suitable for generating h such that $\log_g h$ is unknown to all parties [see remarks in section 4 of 25]. The sub-protocol works as follows:

a) P_i chooses a random secret $s_i \in \mathbb{Z}_{P'}$.

b) P_i publishes $c_i = g^{s_i} \mod P$.

c) From the published commitments $h = \prod_{i=1}^{n} c_i \mod P$ is computed. Note that $\prod_{i=1}^{n} c_i \equiv \prod_{i=1}^{n} g^{s_i} \equiv g^{\sum_{i=1}^{n} s_i} \mod P$. The parties have an additive sharing of $\log_g h$.

4.3.2 Robust Modulus Generation

In order to securely generate the RSA modulus N, the parties engage in the following sub-protocol:

1. P_i chooses a random p'_i in $[2^{\tau-3}, 2^{\tau-2} - 1]$ and $r'_i \in \mathbb{Z}_{P'}$. P_i publicly commits to p'_i by broadcasting $c_i = g^{p'_i} \cdot h^{r'_i} \mod P$ and appends a zero-knowledge range proof using π_{range} in order to prove that c_i is a commitment for p'_i in the mentioned range. A commitment and range proof for q'_i is generated and published analogously.

2. Generate a verifiable secret sharing (see section 2.4.4) with threshold t for $p_i := 4 \cdot p'_i$ and $q_i := 4 \cdot q'_i$. Thus, $p_i \equiv q_i \equiv 0 \mod 4$ and $p_i, q_i \in [2^{\tau-1}, 2^{\tau} - 1]$ is ensured. P_1 sets $p_1 := 4 \cdot p'_1 + 3$ and $q_1 := 4 \cdot q'_1 + 3$ instead to ensure $p_1 \equiv q_1 \equiv 3 \mod 4$. We denote the resulting polynomial for sharing p_i as $f_i(x) = \sum_{j=0}^{t-1} a_{ij} \cdot x^j \mod P'$ and the random companion polynomial $f'_i(x) = \sum_{j=0}^{t-1} a'_{ij} \cdot x^j \mod P'$. Thus, $f_i(0) = a_{i0} = p_i$. $f'_i(0) = a'_{i0}$ needs to be set to $4 \cdot r'_i$. Analogously, verifiable secret sharing of q_i is done using $g_i(x) = \sum_{j=0}^{t-1} b_{ij} \cdot x^j \mod P'$ and $g'_i(x) = \sum_{j=0}^{t-1} b'_{ij} \cdot x^j \mod P'$. Consequently, all parties have published commitments $C_{ij}^{p} = g^{a_{ij}} \cdot h^{a'_{ij}} \mod P$ and $C_{ij}^{q} = g^{b_{ij}} \cdot h^{b'_{ij}} \mod P$.

   P_i distributes the shares $\langle f_i(k), f'_i(k) \rangle$ and $\langle g_i(k), g'_i(k) \rangle$ to each of the other parties P_k privately and receives $\langle f_k(i), f'_k(i) \rangle$ and $\langle g_k(i), g'_k(i) \rangle$ for $k \in \{1, ..., n\} \backslash \{i\}$.

3. P_i verifies the shares that it received from the other parties as in section 2.4.4. I.e., it checks $g^{f_k(i)} \cdot h^{f'_k(i)} \mod P = \prod_{j=0}^{t-1} \left(C_{kj}^{p}\right)^{i^j} \mod P$ and $c_k^4 = C_{k0}^{p}$. For P_1's shares $c_k^4 \cdot g^3 = C_{k0}^{p}$ needs to hold. If any of these checks fail P_i complains publicly about P_k and if a majority of the parties complains against P_k the malicious party is disqualified.

4. P_i generates an additional verifiable secret sharing with threshold $2t - 1$ of the value 0. Therefore, a random polynomial $h_i(x) = 0 + \sum_{j=1}^{2(t-1)} d_{ij} \cdot x^j \mod P'$ and companion polynomial $h'_i(x) = \sum_{j=0}^{2(t-1)} d'_{ij} \cdot x^j \mod P'$ is generated. Analogously to f_i and g_i, commitments C_{ij}^{h} for h_i's coefficients are published and the shares $\langle h_i(k), h'_i(k) \rangle$ are distributed to the other parties in private. In order to prove $h_i(0) = 0$, P_i provides a zero-knowledge proof of knowledge of discrete logarithm of C_{i0}^{h} to the base h using $\pi_{\text{dlog}}^{\kappa}$, which is verified by each receiving party.

5. P_i adds up the shares that it received and generated by itself, respectively, in order to compute a share of the $(2(t-1))$-degree polynomial sharing N. The ith share for a sharing of N is $N_i = \left(\sum_{j=1}^{n} f_j(i)\right) \cdot \left(\sum_{j=1}^{n} g_j(i)\right) + \sum_{j=1}^{n} h_j(i) \mod P'$. P_i computes the commitments $C_{i\Sigma}^{p} = g^{\sum_{j=1}^{n} f_j(i)} \cdot h^{\sum_{j=1}^{n} f'_j(i)} \mod P$ and $C_{i\Sigma}^{q} = g^{\sum_{j=1}^{n} g_j(i)} \cdot h^{\sum_{j=1}^{n} g'_j(i)} \mod P$. Additionally, P_i

publishes a commitment $C_i^{pq} = \left(C_{i\Sigma}^{p}\right)^{\sum_{j=1}^{n} g_j(i)} \cdot h^{r_3} \mod P$ (for a freshly chosen randomness r_3) together with a zero-knowledge proof for the multiplicative relation of $C_{i\Sigma}^{p}$ and $C_{i\Sigma}^{q}$ using π_{mrel}.

6. The parties verify the validity of each N_i by checking that $g^{N_i} \cdot h^{N_i'} \mod P = C_i^{pq} \cdot \prod_{k=1}^{n} \prod_{j=0}^{2(t-1)} \left(C_{kj}^{h}\right)^{i^j} \mod P$. Additionally, they check the proof of multiplicative relation provided by the respective party in order to verify, that C_i^{pq} commits to $\sum_{j=1}^{n} f_j(i) \cdot \sum_{j=1}^{n} g_j(i)$. Note that N_i's companion N_i' needs to be computed and provided by P_i as $N_i' = \left(\sum_{j=1}^{n} f_j'(i)\right) \cdot \left(\sum_{j=1}^{n} g_j(i)\right) + r_3 + \sum_{j=1}^{n} h_j'(i) \mod P'$. Essentially, the parties make use of the fact that they can compute any of the commitments for the shares from the public commitments as in section 2.4.4. Thus, they can also compute the commitments for the sums of any party's shares, which are used for verification of the provided proof.

7. From $2t - 1$ data points the parties can compute N using Lagrange interpolation (see section 2.4.2): Note that $f(x) = \sum_{i=1}^{n} f_i(x)$ is a sharing of $p = \sum_{i=1}^{n} p_i$ and $g(x)$ is a sharing of q analogously. Similarly, $h(x)$ is a sharing of 0. Thus, N_i are shares of the polynomial $\alpha(x) = f(x) \cdot g(x) + h(x) \mod P'$. At least $2t - 1$ of the parties need to publish their shares N_i and N_i' in order to compute the resulting $\alpha(0) = pq = N$.

Clearly, so far P' is required to be large enough to allow for reconstruction of N. Thus, $P' > (n \cdot 2^{\tau})^2 > N$. Because of the expansion rate of the range proof π_{range}, it is ensured that $p_i \leq 2 \cdot (2^{\tau} - 1) - 2^{\tau-1} = 2^{\tau+1} - 2 - 2^{\tau-1}$. So $p \leq (\lceil \frac{n}{2} \rceil - 1) \cdot (2^{\tau+1} - 2 - 2^{\tau-1}) + (\lfloor \frac{n}{2} \rfloor + 1) \cdot (2^{\tau} - 1)$ since the majority of the parties is honest. This can be upper bounded (roughly) by $p < n \cdot 2^{\tau+1}$. Thus, we should choose $P' > (n \cdot 2^{\tau+1})^2$.

4.3.3 Robust Biprimality Test

In the next step, we need to check whether $N = pq$ is the product of exactly two primes. For this we can use a robust version of the biprimality test described in section 4.2. However, since the biprimality test is quite costly in terms of computation time and communication, we can use trial divisions to sort out unsuitable Ns. As soon as a divisor of N (or p or q in the distributed trial divisions) is found, the protocol is restarted in order to generate a new N. Since local trial division based on the public value N does not pose any security risk, we only need robust variants of the distributed trial division protocol and distributed biprimality test.

Robust Distributed Biprimality Test

In order to make the distributed biprimality test secure, we need to ensure that every party adheres to the protocol described in section 4.2.2. Recall, that the parties raise an element g' with Jacobi symbol $\left(\frac{g'}{N}\right) = 1$ to the power of $\frac{p_i+q_i}{4}$ or $\frac{N-p_1-q_1+1}{4}$, respectively. From the committed values in the modulus generation protocol every party can compute a commitment of $p_i + q_i$ as $C_{i0}^{p} \cdot C_{i0}^{q} \mod P$. A commitment for $N - p_1 - q_1 + 1$ can be derived by computing $g^{N+1}/(C_{1,0}^{p} \cdot C_{1,0}^{q}) \mod P$. Note that for all values $p_i + q_i \equiv N - p_1 - q_1 + 1 \equiv 0 \mod 4$, i.e., they are divisible by 4. Thus, we can view $g^{p_i+q_i}$ as $\left(g^4\right)^{\frac{p_i+q_i}{4}}$ and the respective commitments as commitments using the bases g^4 and h. For the secure protocol P_i is required to additionally provide a proof that $\log_{g^4}\left(C^p_{i0} \cdot C^q_{i0} / h^{a'_{i0}+b'_{i0}}\right) = \log_{g'} Q_i$ where $Q_i = g'^{\frac{p_i+q_i}{4}}$. P_1 needs to prove that $\log_{g^4}\left(g^{N+1} / \left(C^p_{1,0} \cdot C^q_{1,0} / h^{a'_{1,0}+b'_{1,0}}\right)\right) = \log_{g'} Q_1$. The proof is constructed analogously to π_{VKeq} (see Figure 2.4) by setting $v := g'$ and calculating mod N instead of N^2. It simultaneously ensures that no wrap-around mod P' occurs due to the integrated range proof. The parties receiving Q_i and the corresponding proof will check consistency of the values with previous commitments and verify the proof in order to detect cheating parties. Apart from that, the protocol remains unchanged.

For the integrated range proof of π_{VKeq} $P' > 2^{l+1}T$ is required where l is a security parameter of the protocol for the proof and T is a upper bound for the committed values $p_i + q_i$ and $N - p_1 - q_1 + 1$. This is clearly dominated by the latter. Following the considerations in section 4.3.2, we can set $T = N_{\max} = (n \cdot 2^{\tau+1})^2$. Subsequently, the requirement $P' > 2^{l+1} \cdot (n \cdot 2^{\tau+1})^2$ overrides the previous lower bound for P'.

Robust Distributed Trial Division

For a robust variant of the protocol described in section 4.2.2 computation mod the small prime p' is not possible, because the parties have a sharing of p only over $\mathbb{Z}_{P'}$. Therefore, the parties "simulate" the computations mod p' in $\mathbb{Z}_{P'}$. Nishide and Sakurai formally prove in appendix A of [41] that the following protocol does not leak information about p if K is chosen such that $\frac{1}{K}$ is negligible.

1. P_i randomly selects $R_{a,i} \in \mathbb{Z}_{p'}$ and $R_{b,i} \in \mathbb{Z}_{K \cdot p_{\max}}$. Thus, the parties have an additive sharing of $R_a = \sum_{i=1}^n R_{a,i}$ and $R_b = \sum_{i=1}^n R_{b,i}$. The parties publish commitments for $R_{a,i}$ and $R_{b,i}$ along with zero-knowledge range proofs based on π_{range} to prove that both were chosen correctly.

2. Analogously to the computation of N, the parties securely compute $\gamma = p \cdot R_a + p' \cdot R_b \mod P'$. P_i checks the generated range proofs and consistency of the computations by the other parties including the necessary proofs of multiplicative relations based on π_{mrel}.

3. If $\gamma \mod p' \neq 0$, then p' definitely does not divide p and p is accepted. Otherwise, the protocol is repeated up to ω times with new R_a, R_b and if $\gamma \mod p' = 0$ in every run, then p' is a divisor of p with a very high probability.

The protocol is executed for a number of small primes p' lower or equal to some threshold B. If a divisor of p is found, then p is discarded and the whole protocol is restarted.

4.3.4 Robust Sharing of the Secret Key

Once a sharing of p and q is found such that $N = pq$ is biprime, i.e., N passed the distributed biprimality test, the parties continue to compute a sharing of the secret key. Note that the steps described so far are the most crucial ones regarding runtime of the overall protocol, since they need to be repeated many times until a suitable N is found (see also the considerations about the number of necessary trials in section 4.2.3).

Recall, that from the computation of N each party can deduce its share of $\phi(N)$ because $\phi(N) = N - p - q + 1$. Thus, the ith party's share of $\phi(N)$ is $N - \sum_{j=1}^{n} f_j(i) - \sum_{j=1}^{n} g_j(i) + 1 \mod P'$, which we denote as $f_\phi(i)$ where $f_\phi(0) = \phi(N)$. Furthermore, P_i can derive a companion for its share $f'_\phi(i) = N - \sum_{j=1}^{n} f'_j(i) - \sum_{j=1}^{n} g'_j(i) + 1 \mod P'$. The commitments $C_i^\phi = g^{f_\phi(i)} \cdot h^{f'_\phi(i)} \mod P$ for these shares can be computed from the commitments published during the computation of N. Using these values, the parties run the following sub-protocol in order to securely generate a sharing of a multiple of $\phi(N)$ based on the idea presented in section 4.2:

1. P_i chooses a random $\beta_i \in \mathbb{Z}_N$ and $R_i \in \mathbb{Z}_{KN}$ where $\frac{1}{K}$ is negligible. Thus, the parties share $\beta = \sum_{i=1}^{n} \beta_i$ and $R = \sum_{i=1}^{n} R_i$. For the following computations each of them generates a verifiable secret sharing of β_i and a verifiable secret sharing *over the integers* of R_i. This includes commitments for β_i, R_i and ΔR_i. Note that Δ appears as a factor for R_i because of the sharing over the integers (see section 2.4.1). With the verifiable secret sharing over the integers P_i already provides a zero-knowledge range proof for the range of R_i. P_i adds a zero-knowledge range proof for the correct range of β_i. Both proofs are based on π_{range}. The shares of $f_{\beta_i}(0) = \beta_i$ and $f_{R_i}(0) = R_i$ are distributed to the respective parties privately.

2. The parties compute $\theta = \Delta\phi(N)\beta + N\Delta R \mod P'$ analogously to the protocol for computing N while checking consistency with the previously used values and verifying the provided range proofs. This also includes proofs of multiplicative relations based on π_{mrel} and randomization of the resulting polynomial using $h(x)$ such that $h(0) = 0$ with the respective zero-knowledge proofs based on $\pi_{\text{dlog}}^{\kappa}$.

3. The parties can now deduce their shares of $-\Delta\phi(N)\beta$ over the integers from the sharing of θ generated in the previous step. More precisely, the sharing of ΔR and θ can be used to compute a sharing over the integers of $N(\Delta R) - \theta = -\Delta\phi(N)\beta$. We denote the resulting polynomial as $f_{\text{sk}}(x)$, where $f_{\text{sk}}(0) = -\Delta\phi(N)\beta$. Assuming that the sharing of ΔR generated in the previous steps is based on the polynomial $f_R(x)$ and $f_R(i) = \sum_{j=1}^{n} f_{R_j}(i)$ the share of P_i is $f_{\text{sk}}(i) = N f_R(i) - \theta$. Similarly to the steps so far, public companions N' and θ' as well as the (private) companions $f'_R(i)$ and $f'_{\text{sk}}(i)$ are computed and provided by the parties. As in section 4.2, $\theta \mod N$ is added to the public key.

Regarding the lower bound for P' we now need to consider that θ should be reconstructable. Thus, $P' > \theta$ is required. According to [41], $\theta_{\max} = 2n\Delta(1+K)N_{\max}^2$. We assume $N_{\max} = (n \cdot 2^{\tau+1})^2$ as before. Note that this is dominated by $N_{\max}^2$ for reasonable parameters K and τ and is larger than the previous bound $2^{l+1} \cdot N_{\max}$ from the biprimality tests for practical l. For the commitments of $f_{\text{sk}}(i)$, $P' > 2|f_{\text{sk}}(i)|$ is required. $|f_{\text{sk}}(i)|$ is bounded by $T := 2N_{\max}n(\Delta K N_{\max} + (t-1)n^{t-1}K\Delta^2 K N_{\max}) + \theta_{\max}$ [41]. Thus, $P' > 2T$ is required.

Now that the parties have a sharing of the secret key over the integers, they are able to conduct distributed decryption. As described before, each of them raises the ciphertext c to the power of their private key share in order to (essentially) reconstruct $c^{-\Delta\phi(N)\beta}$. However, it is not ensured yet, that each party raises c to their private key share. A malicious party could torpedo the decryption process in this step. In order to have verifiable decryption the parties need to construct a verification key for their respective private key share as follows:

1. The parties agree on a random element in the group of squares in $\mathbb{Z}_{N^2}^*$ denoted as $v \in_R Q_{N^2}$. Nishide and Sakurai suggest choosing v by using a hash function H as a random oracle, i.e., $v := H(N)^2 \mod N^2$.

2. From the commitments generated and published in the previous steps the commitments $g^{f_{\text{sk}}(i)} \cdot h^{f'_{\text{sk}}(i)} \mod P$ can be deduced by all participants. P_i additionally publishes a commitment $VK_i := v^{\Delta f_{\text{sk}}(i)} \mod N^2$ and a proof based on π_{VKeq}, that $\log_{v^\Delta} = f_{\text{sk}}(i)$, thus that both commit to the same value. VK_i is the public verification key of P_i.

The use of π_{VKeq} requires $P' > 2^{l+1}T$ (T as above) where l is a security parameter of the proof (see section 2.4.5). Therefore, $P' > 2^{l+1}T$ is the greatest and as such the effective lower bound for the overall key generation protocol.

4.3.5 Verifiable Shared Decryption

We now assume that some plaintext m was encrypted using the regular encryption algorithm for Paillier's encryption scheme (see section 2.2.3) with the public key N (θ is not relevant for encryption). The resulting ciphertext is $c := (N+1)^m \cdot r^N \mod N^2$ where r is selected randomly from $\mathbb{Z}_N^*$. The parties decrypt c in a shared decryption as follows:

Compute Decryption Share

The following sub-protocol is executed by every party P_i that is willing to participate in the decryption. Note that only t parties need to participate in order to compute the plaintext due to the threshold property of the presented encryption scheme.

1. P_i computes and publishes the value $c_i := c^{2\Delta f_{\text{sk}}(i)} \mod N^2$.
2. P_i generates and publishes a zero-knowledge proof that $\log_{v^\Delta} VK_i = \log_{c^{4\Delta}}(c_i)^2$. This proof is based on π_{partDec}.

Formally, these steps represent the functionality of a secure variant of the algorithm *DecShare* in section 2.3.

Combine Decryption Shares

The share combination can be executed by anyone, as it relies only on public or publicly computable information. Therefore, the steps are described as a general algorithm *Dec* (referring to section 2.3).

1. The algorithm takes a set of decryption shares for c as input, denoted as $\{c_i | i \in \{1, ..., n\}\}$. For these shares, verify the corresponding proof provided by P_i. If a party did not provide a share or proof, then simply assume that the share is invalid. The algorithm proceeds only with the valid shares. The set of indices of the parties that provided a valid share is denoted as $S = \{i | P_i\text{'s share } c_i \text{ is valid}\}$.
2. If $|S| < t$ output $\perp$. I.e., abort the decryption since the number of shares is insufficient. Otherwise, remove indices from S until $|S| = t$ to have the correct number of shares for decryption. Then combine the shares by computing $\hat{c} := \prod_{i \in S} c_i^{2\Delta\lambda_i(0)} \mod N^2$ where $\lambda_i(x)$ is defined as for Lagrange interpolation (see section 2.4.2).
3. Reconstruct the plaintext as $\frac{\hat{c}-1}{N} \cdot (-4\Delta^2\theta)^{-1} \mod N$ where $()^{-1}$ denotes the inverse mod N.

Correctness

We briefly review the correctness of the decryption process. Clearly, the following holds due to Lagrange interpolation and the construction of c_i:

$$\hat{c} = c^{2\Delta \cdot \sum_{i \in S}(2\Delta f_{\text{sk}}(i)\lambda_i(0))} \equiv c^{4\Delta^2 \cdot (-\Delta\phi(N))} \mod N^2$$

Plugging in c, we have

$$c^{4\Delta^2 \cdot (-\Delta\phi(N))} = ((N+1)^m \cdot r^N)^{-4\Delta^3\beta\phi(N)} \mod N^2$$

Since $N \cdot \phi(N)$ is the order of $\mathbb{Z}_{N^2}$, the power of r is eliminated. Following the equality $(n+1)^a \mod n^2 = 1 + a \cdot n \mod n^2$ from section 2.2.3, we now have:

$$\hat{c} = 1 + N \cdot m \cdot (-4\Delta^3\beta\phi(N))$$

Due to step 3 of the algorithm above, we receive the result:

$$\frac{m \cdot (-4\Delta^3\beta\phi(N))}{-4\Delta^2\theta} \mod N$$

And since $\theta = \Delta\phi(N)\beta \mod N$, we receive

$$\frac{m \cdot (\Delta\beta\phi(N))}{\theta} \mod N = m$$

5 Implementation

We implemented the distributed key generation protocol of Nishide and Sakurai with security in the honest-majority setting as a protocol for the trustees in the existing implementation of Ordinos. For some early testing we also implemented an adaptation of the protocol for the semi-honest setting as given in section 4.2 (and thus similar to [52]). For simplicity, the former is occasionally referred to as the robust protocol version, since it is robust against malicious behavior, whereas the latter is called semi-honest protocol version. The programming language is Python. We use the modules `gmpy2` and its submodule `mpz` for computations with large integers as well as the submodule `jacobi` thereof for computing Jacobi symbols, `secrets` for generating random numbers and `sympy` for primality tests as well as generating lists of primes. `SHA512` from `cryptography.hazmat.primitives.hashes` is used to generate hash values. The trustees are connected via full-duplex TCP byte streams using sockets. Data is converted into bytes by using the package `pickle`. We note, however, that this is not a secure way of communication and serves for demonstration purpose only. Secure channels and serialization need to be used for security in real-world applications. All zero-knowledge proofs are implemented as non-interactive proofs based on the hash function.

In order to enhance the performance we adapted the protocol in different ways. This chapter covers the major adaptations and evaluation of our implementation.

5.1 Adaptations

5.1.1 Omitting Distributed Trial Division

Following the observations of [52], we omit the distributed trial divisions from the protocol in favor of local trial divisions conducted after the secure computation of N. The main reason is, that the robust distributed trial division protocol (see section 4.3.3) is essentially as complex as the computation of N. Furthermore, distributed trial division is a probabilistic protocol, that needs to be repeated several times to receive reliable results. On the other hand, local trial divisions are obviously very efficient. They require only one iteration of secure multiplication, namely the one for computing N, and other than that add no additional communication. The honest parties will derive the same judgment for a candidate N after trial division up to some threshold B.

5.1.2 Bulk Generation of Candidates

After the first tests of a semi-honest protocol among the trustees, we noticed that key generation with large N (assume $|N_b| \geq 2048$) took more time than Veugen et al.'s semi-honest implementation. The apparent reason was that our parties actually communicated with each other in contrast to [52]. Obviously, the protocol had a communication overhead since the parties kept generating new additive shares of p and q followed by the communication necessary to compute N. As mentioned earlier, approximately 126 000 iterations are necessary on average to generate a sharing of a biprime N. For each iteration, the semi-honest protocol requires exchanging information between each pair of parties once in order to distribute shares of p and q. Additionally, each party needs to publish its share of N afterwards, i.e., one additional broadcast for each party is necessary.

The communication overhead could be lowered by generating bulks of candidates and communicate per bulk instead of communication for each single choice of candidates. For this, we introduce a parameter b as “bulk size”. The parties generate b additive shares of p and q at once. They then send the respective b shares to each of the other parties, compute their shares of N and publish them in one broadcast. Thus, only one mutual communication between each pair of parties and only one broadcast per party and bulk is necessary.

After the computation of b candidates for N, the parties conduct local trial division to sieve out the candidates that are multiples of small primes up to B. Each time a candidate passes trial division a distributed biprimality test follows. If one of them turns out to be biprime, the parties immediately stop testing the other candidates and continue the rest of the protocol with the additive shares of the biprime candidate. If none of the candidates turns out biprime, they restart the protocol with a new bulk.

A conceptual result of this adaptation is an increase of the minimum runtime of the protocol. This is because without bulk generation the parties could find a suitable N in only one iteration and with bulk generation they always compute at least b candidates before finding that one of them is biprime. A good number for b certainly depends on the circumstances like the hardware in use and network quality, but also on the other parameters of the protocol and requirements regarding the runtime of the protocol.

5.1.3 Switching Pedersen Commitment Schemes

As we have seen throughout section 4.3 there are different requirements for P': More precisely, the lower bounds of P' are:

1. $P' > N_{\max}$ for the computation of N.
2. $P' > 2^{l_1+1} \cdot N_{\max}$ for the biprimality test due to π_{VKeq}.
3. $P' > 2T$, where $T := 2N_{\max}n(\Delta K N_{\max} + (t-1)n^{t-1}K\Delta^2 K N_{\max}) + \theta_{\max}$ for the secure sharing of the secret key.
4. $P' > 2^{l_2+1} \cdot T$ for the construction of verification keys using π_{VKeq}.

Note that the former two are dominated by $N_{\max}$ while the latter two are dominated by $K^2 \cdot N_{\max}^2$, if we assume l_1, l_2 fixed somewhere between 40 and 80. Further, the first two bounds are the ones relevant for generating the public key, while the other two are necessary only for computing the secret key shares and verification keys. For a one-to-one implementation of the protocol it is necessary to use the largest of these bounds for the whole protocol. This is because the computations rely on the sharings generated and used in the previous steps, respectively.

The idea for this adaptation is to use the 2nd bound for generating a biprime N and then switch to using the 4th bound for generating secret key shares and verification keys. This would enable us to use a much lower P' while searching for a suitable N, which is the most heavily executed part of the protocol. A smaller P' reduces the size of shares, proofs and communication. Thus, switching P' would lead to more efficiency during generation of the modulus. To have some numbers we estimate for reasonable parameters generating a 2048-bit biprime modulus among 3 parties requires $P' \approx 2^{2100}$ (referring to the 2nd lower bound). However, securely sharing the secret key and generating the verification keys in the next steps requires $P' \approx 2^{4200}$ (referring to the 4th lower bound). Note that these are very rough estimations, but they show the magnitude of the different requirements for P'.

We adapt the protocol as follows:

1. The parties run the modulus generation including the biprimality test using the 2nd lower bound, thus $P'_1 > 2^{l_1+1} \cdot N_{\max}$, where l_1 is the security parameter for π_{VKeq}. The parameters P_1, g_1, h_1 are chosen as before. Afterwards, the parties have additive shares of p and q and know N. Furthermore, the parties know commitments of each others shares, e.g., $C^p_{i0} = g_1^{p_i} \cdot h_1^{r_{1,i}} \mod P_1$, and the random companion N'.

2. The parties choose a new P'_2 larger than the 4th lower bound, thus $P'_2 > 2^{l_2+1} \cdot T$ where l_2 is the security parameter of the instance of π_{VKeq} used for the verification keys. New parameters P_2, g_2, h_2 are chosen analogously for P'_2, thus choosing a new instance of the Pedersen commitment scheme. The parties publish new commitments for their additive shares, such as $C'^p_{i0} = g_2^{p_i} \cdot h_2^{r_{2,i}} \mod P_2$. P_i uses π_{comEq} to prove that C^p_{i0} and C'^p_{i0} commit to the same value. As we have learned in section 2.4.5, this requires an additional range proof to prevent a wrap around mod P'_1. In other words, P_i needs to prove additionally that C'^p_{i0} commits to a value below P'_1.

   Due to the protocol for modulus generation, C'^p_{i0} should actually commit to a value in range $[2^{\tau-1}, 2^\tau - 1]$. We denote the lower bound as A and the upper bound as B as in section 2.4.5. Because of the expansion rate of π_{range} (see section 2.4.5) we can use this proof to effectively prove that a committed value is in range $[2A - B, 2B - A]$. For our choice of A, B this is $[1, 2^{\tau+1} - 2^{\tau-1} - 2]$. Recall, that P'_1 was chosen to be larger than $2^{l_1+1} \cdot N_{\max}$, where $N_{\max} = (n \cdot 2^{\tau+1})^2$. This is clearly larger than $2^{\tau+1} - 2^{\tau-1} - 2$. Thus, this range proof suffices to prove that the committed value in C'^p_{i0} is lower than P'_1. Subsequently π_{range} can be used in combination with π_{comEq} to prove that C'^p_{i0} and C^p_{i0} commit to the same additive share p_i or q_i, respectively. The advantage of using the loose range proof π_{range} here is that we do not need to implement another (potentially less efficient) range proof as suggested in Appendix A. Note that this combination of the two zero-knowledge proofs may not suffice in other settings.

3. Analogously to the original protocol, the parties can compute a sharing of $\phi(N)$ over $\mathbb{Z}_{P'_2}$. Since N was already computed they do not need to execute the multiplication protocol again. Instead, each P_i generates a verifiable secret sharing of p_i and q_i as before and distributes the shares to the other parties. Then P_i sums up the received shares of p and q, analogously to section 4.3 as $\sum_{j=1}^n f_j(i)$ and $\sum_{j=1}^n g_j(i)$ and computes its share of $\phi(N)$ as $f_\phi(i) = N - \sum_{j=1}^n f_j(i) - \sum_{j=1}^n g_j(i) + 1 \mod P'_2$. The companion $f'_\phi(i)$ is derived similarly using N' and the companions of the received shares. As shown before, each party can derive commitments for the other parties' shares of $\phi(N)$ and use them to verify consistency with previously used values.

4. The secure sharing of $-\Delta\phi(N)\beta$ and the public value θ is computed just as in the original protocol using the “new” sharing of $\phi(N)$ over $\mathbb{Z}_{P'_2}$. Thus, the parties can continue with the rest of the protocol without further changes.

A similar adaptation for the semi-honest version of the protocol is possible, yet much easier since no proofs are necessary. In the semi-honest protocol, the parties can just compute a new sharing of $\phi(N)$ using their additive shares of p, q during computation of the secret key shares and thus switch to a larger P'.

5.1.4 Postponement of Proofs During Modulus Generation

During implementation the sheer number of proofs turned out as a bottleneck. Note again that in an average run of the protocol most of the time is spent generating new additive shares, computing N and testing its biprimality. As mentioned before, generating a 2048-bit modulus requires an average of approximately 126 000 trials. In the original protocol the parties need to prove for each choice of p_i, q_i that their values are in the specified range, $h_i(0) = 0$ and that they multiplied correctly. This totals to an average of 252 000 range proofs based on π_{range} per party and half as many proofs based on $\pi^{\kappa}_{\text{dlog}}$ and π_{mrel} each until a suitable modulus was generated.

The first observation is that it should suffice if the parties postpone all proves until after a biprime N was found. Postponing proofs was also proposed in section 6.3. of [29]. This somewhat accommodates the fact that most of the additive shares and their proofs will be discarded during a run of the protocol anyway. However, we cannot simply omit all proofs during modulus generation. The problem is, that an adversary could use values that definitely lead to a non-biprime N in every run of the protocol. Consider a malicious party that chooses $p'_i \in Z_{P'}$ such that $(p_i :=)4 \cdot p'_i \mod P' \equiv 1 \mod 4$ ($i > 1$ in this example). If all other parties still adhere to the protocol, this would lead to $p \equiv 0 \mod 4$ and thus a non-prime p. As long as the malicious party continues choosing p_i in that way, the protocol will never end and the honest parties will not be able to detect the cheating party because the postponed proofs are never reached. Other ways of deliberately producing non-prime p, q or a non-biprime N are thinkable. Note that the parties may detect some deviation from the protocol by checking $N \overset{?}{\equiv} 1 \mod 4$. However, they are not immediately able to blame a specific party for deviating from the protocol if this relation does not hold.

In order to mitigate this problem, the protocol may randomly require the parties to provide some proofs during modulus generation. Even more efficient, the parties can simply open all shares and values used after N failed trial divisions or the biprimality test. Thus, the honest parties may check whether all values are consistent with earlier commitments and chosen or computed according to the protocol. The selection of which shares to open can be based on the adaptation for bulk generation of candidates.

Our adaptation is summarized as follows:

1. The parties do *not generally* provide proofs during modulus generation.
2. If all candidates generated in a bulk fail trial divisions or biprimality test, they open a certain number of their shares for these candidates as described in the following. Let b be the number of candidates generated in a bulk and d the number of candidates to open. A sharing is opened by publishing p_i, q_i, and the coefficients for the polynomials f_i, g_i, h_i, and the respective companions as well as the randomness used to compute the companion N'_i. P_j

verifies honesty of P_i by checking, that p_i, q_i, and the coefficients are consistent with the previously published commitments. P_j further verifies the range of p_i and q_i and verifies $h_i(0) = 0$. The shares that were previously received from P_i are checked by evaluation of the respective data points on the respective polynomials. Finally, P_j executes the multiplication steps of P_i and verifies that N_i was computed correctly. We consider the candidates within a bulk indexed and denote as S the set of the indices of candidates which shall be opened, where $|S| = d$.

In order to randomly generate S, each of the parties publicly commits to an arbitrary random value, afterwards the parties open their random value and sum them up $(\bmod\ P')$. Meanwhile, the parties verify that each of the others in deed opened the value that they committed to. The resulting jointly generated randomness is denoted as r and concatenation is denoted by $||$. S consists of the first d distinct indices generated as $H(r||j) \mod b$ where j is a counter starting at 0.

Thus, all parties know which shares to open. If any of the values from the other parties is missing or found to be inconsistent, P_i complains publicly about the respective other party and the cheating party is disqualified if a majority complained against it.

3. In case of finding a biprime N, the parties need to provide all proofs for their additive shares of p, q, for $h_i(0) = 0$ and for correct multiplication. The other parties check consistency of all values and validity of the proofs as in the original protocol. This is a logical consequence of postponing proofs and guarantees that the final N was constructed correctly in the sense of the original protocol.

From the security point of view the adapted protocol is at least as secure as the original protocol since the correctness of the final N is proven and verified identically to the original protocol. From the opened values from the honest parties an adversary does not learn any information, since all of these were random values that have been discarded for any further usage. Regarding the range of the parties' shares, the opening of shares instead of using π_{range}, even tightens the room for deviation by malicious parties. With π_{range} comes an expansion rate of 3 which allows for some deviation in the range of shares, however the range of opened values can be verified directly and precisely. Thus, honest parties are enabled to detect misbehavior to some extent also in this regard.

To demonstrate the effectivity of this adaptation, we compute the probability of catching a cheating party. For simplicity, we fix d to a percentage share of b and set b to 100. Say, d is set to 1% of d, i.e., the parties open only 1 of the candidates of a bulk during modulus generation. We assume, that a cheating party tries to cheat at least once in a bulk, otherwise it would risk that modulus generation is finished with honestly chosen values. Clearly, the probability that a cheating party is detected after processing the first bulk is $\geq 1\%$. The complementary probability is $\leq 99\%$. Thus, a continuously cheating party is detected after processing the k^{th} bulk with probability $\geq 1 - (0.99)^k$. During generation of a 2048-bit N, the parties need approximately $k = 126\,000 \div b$ bulks on average. The probability that a cheating party is caught after the average number of runs is thus: $Pr[P_i \text{ is caught cheating after } k \text{ bulks} | P_i \text{ cheats continuously}] \geq 1 - (0.99)^{1260} \approx 99.9996\%$. For a bulk size of 1000 we get a probability $\geq 71.8\%$ when checking $d = 10$ (1% of b) and a probability $\geq 99.9998\%$ for $d = 100$ (10% of b). Therefore, we can adjust the parameter d (and/or b) to ensure that continuously cheating parties are caught after the average runtime of the protocol with a very high probability. Note, however, that the probability to catch such a party becomes nearly 100% if we allow the protocol to run longer. Thus, a party that deliberately prevents the honest parties from finding a suitable N will be caught eventually in any run of the adapted protocol.

Note that if we simultaneously apply the adaptation for switching the Pedersen commitment scheme as described in the previous section, we do not need to actively postpone range proofs, since a range proof for the additive shares is already included in this adaptation. Thus, we may just omit these proofs and it is ensured, that range proofs for the shares that compose the final N are conducted in a suitable way during the switch to another instance of the Pedersen commitment scheme.

5.1.5 Postponement of Proofs in Biprimality Tests

The proofs provided during biprimality tests can be reduced similarly. First the parties conduct the biprimality tests without providing any proofs. Recall, that a biprimality test consists of several iterations and fails as soon as one of these iterations confirms non-biprimality of N. Thus, in case of a failed biprimality test, only the honesty of the parties in the failing iteration needs to be ensured. This can be done by letting all parties open their shares p_i, q_i and verifying that they have raised g to the correct power, including verification of consistency of p_i and q_i with previous commitments. In case of a successful biprimality test the parties need to provide their proofs based on π_{VKeq} for every single iteration. They verify the consistency and validity of the proofs provided by the other parties as in the original protocol. This ensures, that all iterations of a successful biprimality test were in deed conducted correctly.

5.2 Evaluation

In this section we evaluate the runtime of the protocol and the influence of the adaptations and several parameters on the average runtime. The protocol is run among $n = 3$ parties with threshold $t = 2$.

5.2.1 Parameters

Security Parameters

We fix the following security parameters of the protocol:

- Range proofs based on π_{range} are repeated $l = 40$ times to get an error probability of $\frac{1}{2^{40}}$.
- The security parameter for $\pi^{\kappa}_{\text{dlog}}$ is fixed to $l = 40$ as well to reduce the prover's chance of cheating.
- We use the parameters $l = 40$ and $h = 40$ for π_{VKeq} to have a completeness bound of $1 - \frac{1}{2^{40}}$ and a soundness bound of $\frac{1}{2^{40}}$.
- Biprimality tests are repeated 100 times with different gs, so that the generated N is biprime with probability $\geq 1 - \frac{1}{2^{100}}$.

- $K = 2^{40}$ is set for secret sharing over the integers.
- $K = 2^{40}$ is set for the computation of a sharing of the secret key (range for R_i).
- For π_{partDec} the security parameters are set to $l = t = 40$ as well.

Hardware & Network

All tests were run on machines with Intel Core i5-7500T CPUs (4 cores, 2.7 GHz base frequency) and a total memory of 16 GB. The machines have access to a 1 Gbit/s local network. Due to scarcity of resources, two parties run on the same host and communicate with another party running on a different host with identical hardware in the same local network. The round-trip time is around 0.1 to 0.3 ms. The trustees run in one thread each, thus using only one core of the CPU per party.

5.2.2 Performance

Bulk Generation in Semi-Honest Protocol

For the evaluation of our bulk generation approach we ran the semi-honest version of the protocol 100 times each while setting the parameter b to powers of 10 (Figure 5.1) and multiples of 1 000 (Figure 5.2). We fixed the parameter for trial divisions to $B = e^{11} \approx 59\,874$ using experimental results. A way of determining a suitable B is elaborated on another occasion throughout the evaluation. Just as in [52], 2048-bit moduli were generated. The respective figures depict the average, minimum, and maximum runtime of the protocol until completion of the whole key setup measured for each choice of b. Time for setting up the connections between the parties is not included in the measured runtime. Note that both graphs show the results on a logarithmic scale for the vertical axis. The case $b = 1$ resembles a run of the protocol without bulk generation of candidates.

Figure 5.1 shows that running key setup without bulk generation (i.e., $b = 1$) takes significantly longer on average than any run of the protocol with bulk generation and a reasonable choice of b. This is due to the immense communication overhead when communicating for each set of candidates. On the other hand, when increasing b too much the average runtime will increase as well, because the parties generate a lot of candidates ahead and check for their suitability only afterwards. We observed the best average runtime (35 seconds) at $b = 1\,000$.

For the minimum runtime we observe relatively high values for both, high and very low choices of b. However, in theory we should observe the lowest minimum runtime at $b = 1$ when running the protocol more often, since we could be lucky to find a suitable sharing on the first shot. For our experiments it took 1 970 shots and 47 seconds for the fastest run with $b = 1$. The increase of minimum runtime for very high b is a logical consequence of the bulk generation approach similar to the increase of average runtime.

From these results we deduced that the best average runtime could be found somewhere near the lower multiples of 1 000. Therefore, the test was repeated for b between 2 000 and 9 000 (again 100 runs for each choice of b). The results are depicted in Figure 5.2 and supplemented by the results for $b = 1\,000$ and $b = 10\,000$.

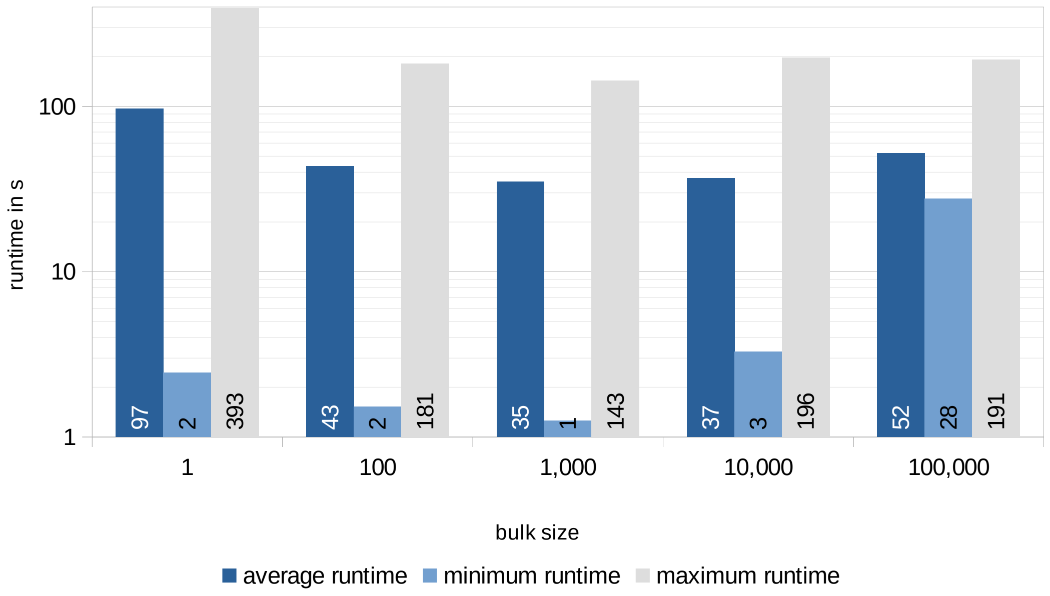

Figure 5.1: Average, minimum, and maximum runtime of semi-honest protocol version for 2048-bit moduli and bulk sizes $b = 1$ to 100 000

The average runtime in these experiments was between 32 and 40 seconds for each choice of b indicating only insignificant fluctuation. The results are complemented by the observed average number of failed trial divisions and biprimality tests depicted in Table 5.1. Although we measured the lowest average runtime for $b = 2\,000$ and the highest average runtime for $b = 6\,000$ these results alone do not qualify for much conclusion. After all, key generation is a random process and thus prone to fluctuation. More precisely, we observed between 120 000 and 130 000 probes per bulksize on average, but only 112 678 for $b = 2\,000$ and exceptional 138 542 for $b = 6\,000$. This difference in the number of processed probes likely accounts for most of the difference in observed average runtimes. For our setup it shall suffice to know that any b between 1 000 and 10 000 may lead to an acceptable runtime around 35 seconds.

b	failed trial divisions	failed biprimality tests	b	failed trial divisions	failed biprimality tests
1	130,720	1,374	6,000	138,542	1,448
100	124,670	1,311	7,000	115,170	1,210
1,000	126,694	1,333	8,000	124,906	1,310
2,000	112,678	1,188	9,000	131,572	1,381
3,000	124,603	1,311	10,000	125,609	1,314
4,000	123,720	1,305	100,000	107,403	1,123
5,000	122,039	1,284	**total average**	**123,717**	**1,299**

Table 5.1: Observed average number of failed trial divisions and biprimality tests per bulk size (semi-honest protocol version, 2048-bit moduli)

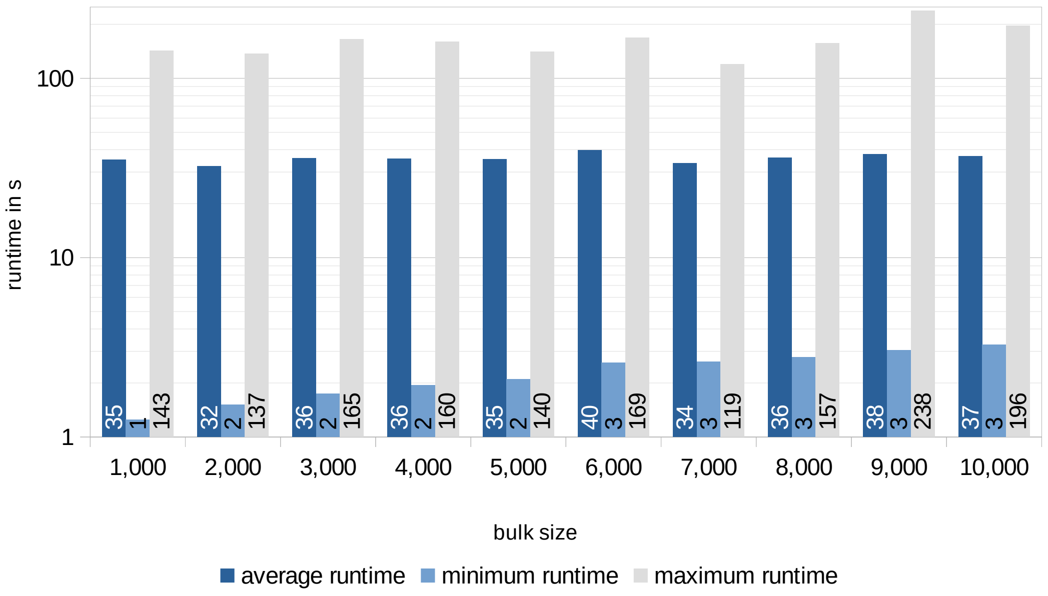

Figure 5.2: Average, minimum, and maximum runtime of semi-honest protocol version for 2048-bit moduli and bulk sizes $b = 1\,000$ to $10\,000$

Bulk Generation in the Robust Protocol Version

We repeated the experiments for the robust version of the protocol using all of the described adaptations except for switching the instance of the Pedersen commitment scheme. For the postponed proofs during modulus generation the parameter d was fixed to 100% of b. I.e., if a suitable modulus was not found within a bulk, all shares are opened and checked by the parties. Again, we fixed B to 59 874 since it turned out as one of the best choices in experiments. For starters, we decided to run the protocol to generate keys for 512-bit moduli, since runtime for larger moduli is significantly longer and does not allow for reasonable evaluation. Following the considerations in section 4.2.3, the expected average number of probes for 512-bit moduli is $\approx 7\,874$ and the expected number of biprimality tests for $B = 59\,874$ is ≈ 82. We ran the protocol for b chosen as powers of 10. As before, the average, minimum and maximum runtime for 100 runs for each choice of b were measured. The results are depicted in Figure 5.3.

They show that bulk generating candidates has less impact on average runtime than observed for the semi-honest protocol version and that average runtime is much longer in general. The increase of minimum runtime depending on b is still clearly visible, but with higher values than before. Both effects are due to additional runtime for generating companion values and commitments for verifiable secret sharings as well as proofs and a number of checks and verifications and thus less dominance of communication itself. However, the impact of communication and thus of bulk generation on runtime may be larger when running the protocol in a less optimal network. Different from before, it is not as clear which values for b are good choices regarding average runtime of the protocol. In our experiments $b = 10$ delivered the shortest average runtime. But this is mainly due to fewer probes processed in these runs on average (see Table 5.2 for complementary numbers).

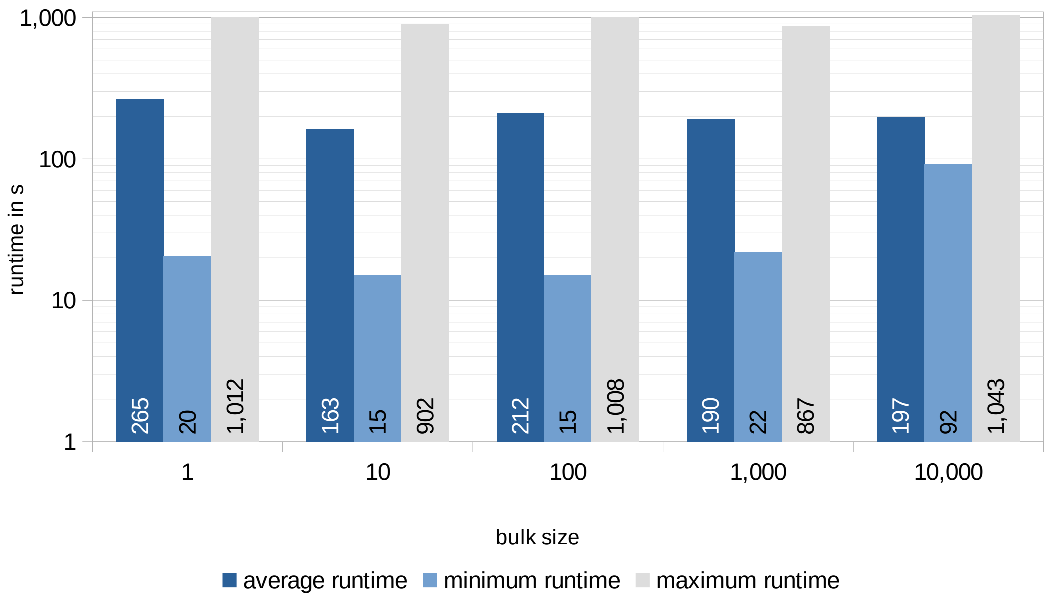

Figure 5.3: Average, minimum, and maximum runtime of robust protocol version for 512-bit moduli and bulk sizes $b = 1$ to $10\,000$

Switching Pedersen Commitment Scheme

The previous experiments for generating 512-bit moduli were repeated while additionally enabling the adaptation for switching the instance of the Pedersen commitment scheme after finding a suitable N. Other than that nothing was changed. The results for $b = 1$ to $b = 10\,000$ are depicted in Figure 5.4 and complemented by the average number of trial divisions and failed biprimality tests depicted in Table 5.2. Obviously, the average, minimum and maximum runtime for each choice of b turned out significantly lower than in the previous experiments. This is due to the usage of a much lower P' while generating N, which results in much smaller and faster proofs, commitments, and consistency checks. Evidently, the cost for switching the instance of the Pedersen commitment scheme once after generating N, which includes usage of both π_{comEq} and π_{range} as well as generating a new sharing of $\phi(N)$, is outweighed by the benefits of using a smaller P' while generating candidates for N. We measured that only 0.74 seconds were spent for switching the Pedersen commitment scheme on average.

Securely Generating 2048-bit Moduli

Finally, secure key generation with the aim of 2048-bit moduli using the robust protocol is evaluated. We used all of the presented adaptations to receive the lowest possible average runtime. First the choice of some parameters needs to be elaborated:

- b: The previous results indicated, that the range of acceptable b is relatively large and the effect of b on the runtime of the protocol is much lower in the malicious setting than it is in the semi-honest setting. The influence of communication on runtime can be expected to

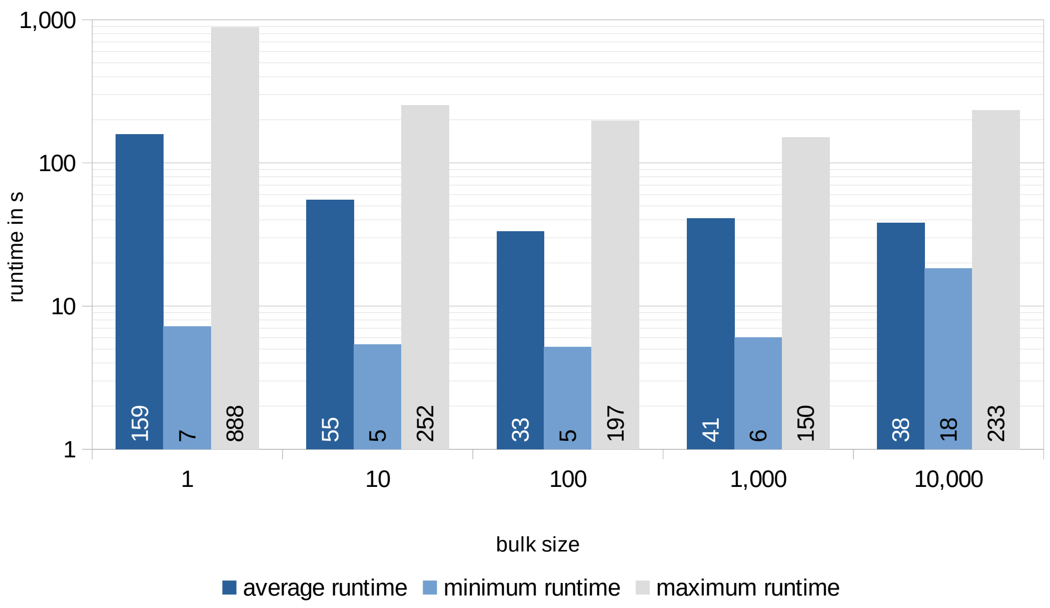

Figure 5.4: Average, minimum, and maximum runtime of robust protocol version for 512-bit moduli and bulk sizes $b = 1$ to 10 000 with switching of commitment scheme

b	switch PCS	failed trial divisions	failed biprimality tests
1	False	7,160	74
10	False	5,891	61
100	False	8,343	86
1,000	False	7,610	79
10,000	False	8,218	84
1	True	7,259	75
10	True	8,498	91
100	True	6,887	72
1,000	True	9,056	93
10,000	True	8,149	84
total average	**-**	**7,707**	**80**

Table 5.2: Observed average number of failed trial divisions and biprimality tests per bulk size (robust protocol version, 512-bit moduli) with and without switching the Pedersen commitment scheme (PCS)

be very low when generating 2048-bit moduli using the robust version of the protocol, even with low b. Therefore, b is simply fixed to 100. Thus, b is relatively low, and we may get a relatively low minimum runtime as well. Also, $b = 100$ is suitable for setting d to a low percentage share of b. Clearly, d is at least 1 and thus a lower b of, say, 10 would require to open at least 10% of the values used in a bulk. When choosing b in another setup, especially with slow or high latency network connection, other values may turn out more suitable.

- d: Following our considerations in section 5.1.4, we set $d = 1$, thus achieving a probability of $\geq 99.9996\%$ during an average run for catching a party that continuously uses malicious values of any kind for modulus generation.
- B: Clearly, the robust version of distributed biprimality tests consumes more runtime than its equivalent for the semi-honest setting. This is due to generating and checking proofs based on π_{VKeq} and opening shares for checks as described in section 5.1.5, respectively. Also communication itself during biprimality tests has an impact on the runtime. On the other hand, our local trial divisions consume the same runtime in both the semi-honest and malicious setting. In order to determine a suitable B for our protocol for the malicious setting, we measured the runtime consumed for trial divisions and biprimality tests. The raw results of these measurements are depicted in Table 5.4. The results were extrapolated for expected average runs with 126 000 trial divisions (TD) and the respective number of expected biprimality tests (BPT) for B (see section 4.2.3). Also the time consumed for generating the prime list for trial divisions (using the function `sympy.sieve.primerange`) was included. The expected average runtimes for trial divisions and biprimality tests as well as the total expected runtime spent on failed primality tests are depicted in Table 5.3. As a side note we learned, that a full biprimality test, thus including all proofs, consumes approximately 2 minutes. The experimental results show, that we can expect least runtime (around 25 seconds) spent on primality tests when setting $B = e^{13} \approx 442\,413$.

	Expected	# of primes	Measured		Time gen	Expected runtime for		
B	# of BPT	in range [2, B]	TD/s	BPT/s	prime list	TD	f. BPT	**total**
e^{9}	1,960	1,019	203017.02	51.21	0.00	0.62	38.28	38.90
e^{10}	1,588	2,466	134572.81	52.58	0.00	0.94	30.20	31.14
e^{11}	1,312	6,048	74157.05	48.34	0.01	1.70	27.14	28.85
e^{12}	1,102	14,912	41048.91	49.41	0.02	3.07	22.30	25.40
e^{13}	939	37,128	22327.18	48.74	0.07	5.64	19.27	**24.98**
e^{14}	810	93,117	10995.19	50.87	0.19	11.46	15.92	27.57
e^{15}	706	234,855	5403.77	55.00	0.52	23.32	12.84	36.67
e^{16}	620	595,341	2587.79	56.67	1.42	48.69	10.94	61.05
e^{17}	549	1,516,233	1163.72	54.88	3.87	108.27	10.00	122.15
e^{18}	490	3,877,186	481.72	52.10	10.64	261.56	9.41	281.61
e^{19}	440	9,950,346	215.20	38.83	29.31	585.50	11.33	626.14
e^{20}	397	25,614,562	115.64	34.79	80.33	1089.62	11.41	1181.36

Table 5.3: Experimental results for influence of B on runtime

The results of 100 runs of the protocol with these settings are depicted as histograms in Figure 5.5, Figure 5.6, and Figure 5.7. Figure 5.5 depicts the total runtime of the protocol. The average runtime was 6 099.61 seconds ($\approx$ 102 minutes) with a very high standard deviation of 6 040.88 seconds. Almost half of the runs (49) finished within 4 000 seconds. Figure 5.6 shows the observed number of failed trial divisions in the runs. The average number was 139 551, thus notably larger than the theoretical value. Most runs (61), however, finished with less than 120 000 failed trial divisions. Some, on the other hand, needed considerably more probes to find a suitable modulus. The largest number of failed trial divisions in a single outlier was 944 191, which significantly affects both the observed deviation in *all* presented values and the resulting averages. Figure 5.7 shows the observed

		observed		measured runtime for			
B	$B \approx$	# of TD	# of f. BPT	gen prime list	TD	failed BPT	final BPT
e^9	8,103	15,152	239	0.00	0.07	4.67	120.02
e^{10}	22,026	81,667	1,033	0.00	0.61	19.65	120.57
e^{11}	59,874	19,878	225	0.01	0.27	4.65	119.95
e^{12}	162,755	123,353	1,097	0.02	3.01	22.20	120.22
e^{13}	442,413	75,810	544	0.07	3.40	11.16	119.93
e^{14}	1,202,604	56,008	351	0.19	5.09	6.90	117.76
e^{15}	3,269,017	51,161	285	0.52	9.47	5.18	120.04
e^{16}	8,886,111	62,514	295	1.42	24.16	5.21	119.98
e^{17}	24,154,953	273,497	1,176	3.87	235.02	21.43	119.93
e^{18}	65,659,969	127,484	529	10.64	264.64	10.15	119.98
e^{19}	178,482,301	41,686	156	29.31	193.71	4.02	121.41
e^{20}	485,165,195	11,733	28	80.33	101.46	0.80	120.50

Table 5.4: Raw results of runs with different choices of B

number of failed biprimality tests. On average 1 048 failed biprimality tests preceded the respective final (successful) biprimality test. This is, again, a bunch more than the expected 939 biprimality tests.

As mentioned, the results are biased by an outlier. Most certainly, the impact of the outlier would be negligibly small, if significantly more runs were measured. However, running more tests is quite costly in terms of time. Therefore, we recomputed the average values under exclusion of the outlier, in order to get more realistic values for the average duration of the protocol. As an alternative, the values can be extrapolated to a run with the theoretically expected number of trial divisions and biprimality tests. Leaving out the outlier, the average runtime was 5 755.77 seconds ($\approx$ 96 minutes), the average number of failed trial divisions was 131 424 and the average number of failed biprimality tests was 989. When extrapolating the runtime spent on modulus generation, trial divisions, failed biprimality tests, and opening of shares to an expected 126 000 probes including 939 biprimality tests while assuming the time consumed by other parts of the protocol to be constant[1] the overall runtime is 5 644.49 seconds ($\approx$ 94 minutes). Both numbers are quite close and allow us to expect the average duration of the protocol in the given setup close to 95 minutes.

Additionally, the runtime spent on different parts of the overall protocol was measured. The average values are depicted in Figure 5.8. It indicates that most of the time (more than 5 700 seconds) was spent on generating new moduli. Only around 149 seconds were spent on primality tests, 7 of which on trial divisions, 22 on failed biprimality tests and 120 on the final biprimality test. Other than in the semi-honest protocol version, where primality tests make up a large portion of the overall runtime, they do not dominate in the robust version of the protocol. 221 seconds on average were spent on other parts of the protocol: 3 on preparations (instantiation of the Pedersen commitment schemes), 148 on opening and verifying (1% of) the shares for candidates that failed primality tests. Less than 1 second was consumed for postponed proofs, once a suitable N was

[1]Numbers for different parts of the protocol are presented in following paragraph and Figure 5.8.

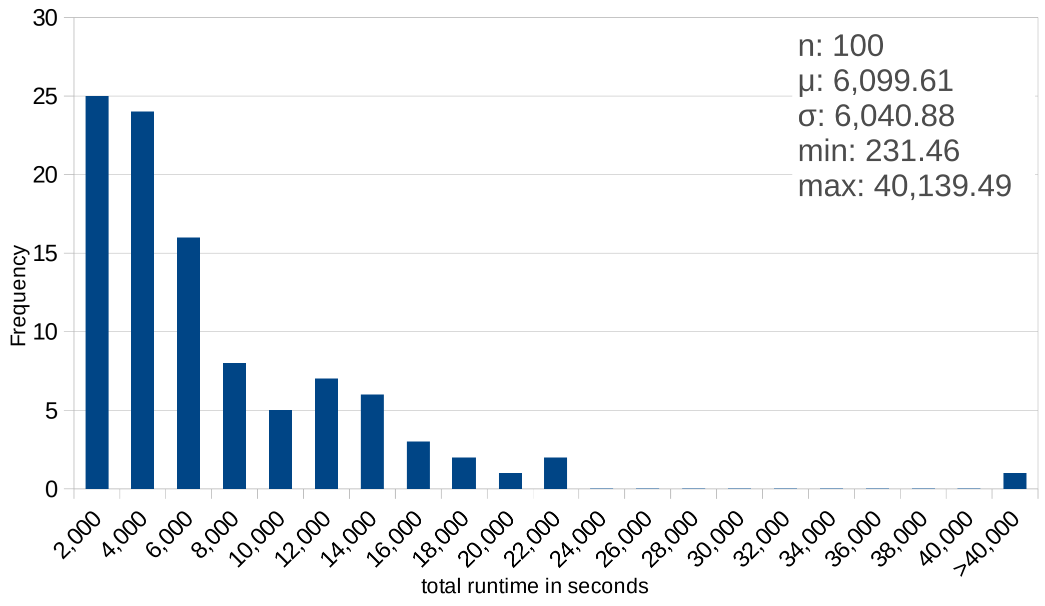

Figure 5.5: Measured runtime of robust protocol version for 2048-bit moduli

generated. 40 seconds were spent on switching the Pedersen commitment scheme (including range proofs). Computation of the secret key shares took 21 seconds on average and 9 seconds were spent generating verification keys.

5.3 Comparison of Results for Semi-Honest Protocol

Veugen et al. [52] measured an average runtime of 66 seconds for generating a key sharing with 2048-bit moduli in 1 097 runs among 3 parties with a threshold of 2. They fixed B to 20 000 using experimental results. However, they did not implement communication among the parties, but rather simulated the parties' roles in one single thread on a 2.4 GHz single-core CPU. They also omitted randomization when using the BGW protocol [4]. Additionally, they simplified the computation of θ and were able to omit the secret sharing of R over the integers and the larger lower bound for P'. Our implementation sticks to secret sharing over integers as presented by Nishide and Sakurai, but we were also able to use a smaller lower bound for P' during modulus generation as mentioned in section 5.1.4. The overall effect of the different computation of the secret key shares on runtime is, however, rather marginal: We measured that computing the sharing of the secret key takes only around 0.5 seconds on average in our setup. Clearly, the average runtime in our experiments was lower for most choices of b, which is mostly due to the fact that effectively 3 threads on more powerful quad-core CPUs were used.

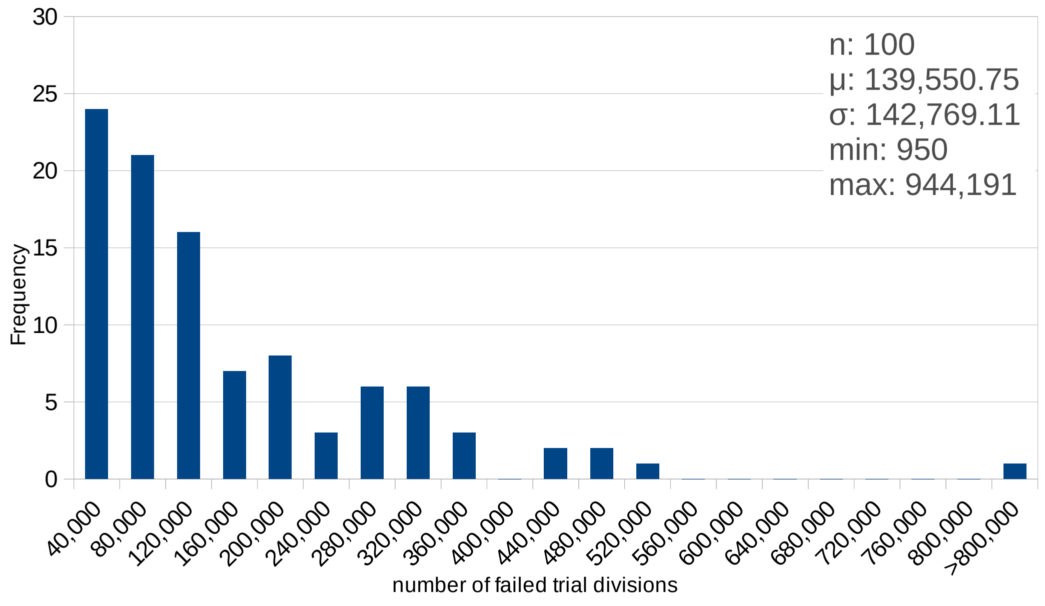

Figure 5.6: Observed number of failed trial divisions in runs of robust protocol version for 2048-bit moduli

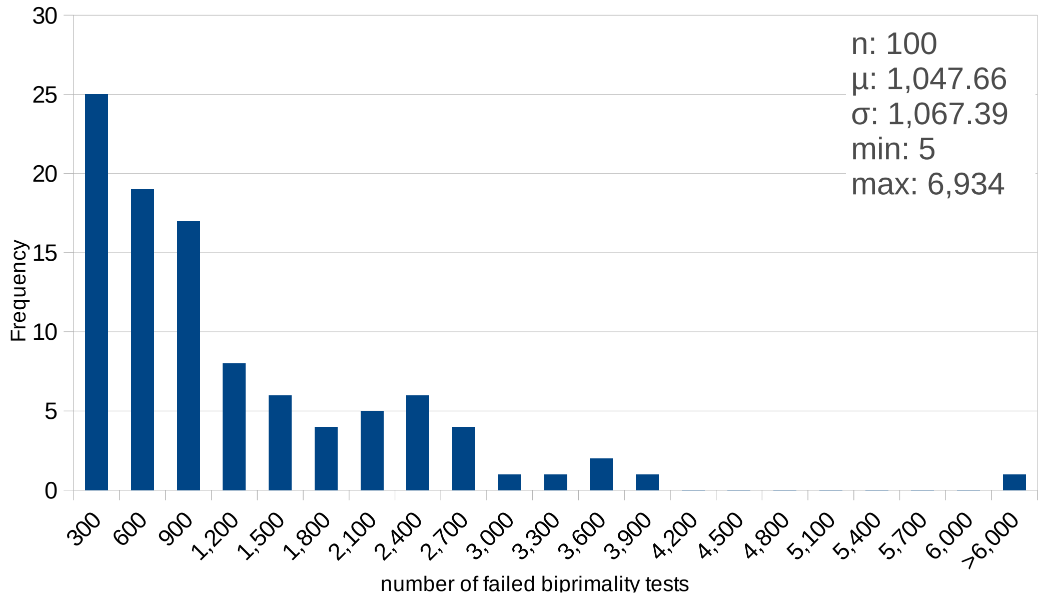

Figure 5.7: Observed number of failed biprimality tests in runs of robust protocol version for 2048-bit moduli

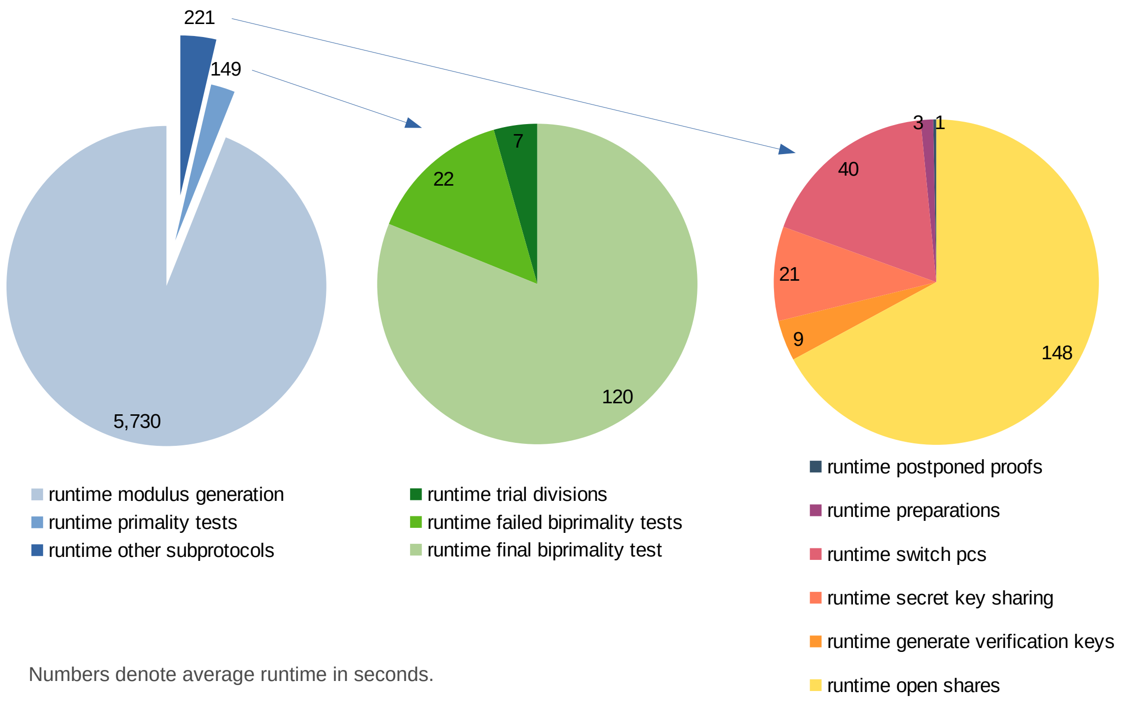

Figure 5.8: Distribution of average runtime on different sub-protocols

6 Generation of Randomness for MPC Protocols

The next chapter is focused on generating randomness of certain shape for MPC protocols used in Ordinos. As mentioned in the introduction, the goal is to abolish of the trusted party, who provides this randomness along with the key shares. More precisely, Ordinos' tallying protocol makes use of a "greater-than" ($P^{\text{gt}}_{\text{MPC}}$) and an "equals" protocol ($P^{\text{eq}}_{\text{MPC}}$). Küsters et al.'s instantiation of Ordinos uses the protocols by Lipmaa and Toft [34] to instantiate $P^{\text{gt}}_{\text{MPC}}$ and $P^{\text{eq}}_{\text{MPC}}$ because of their superior online complexity. $P^{\text{eq}}_{\text{MPC}}$ takes two encrypted numbers x, y as inputs and computes an encrypted bit b that determines whether $x == y$. $P^{\text{gt}}_{\text{MPC}}$ similarly determines whether $x \geq y$. The latter actually computes a relation that is often called "greater-equals". However, protocols for the actual relation "greater-than" as well as for "lower-than" and "lower-equals" can be deduced from $P^{\text{eq}}_{\text{MPC}}$ and $P^{\text{gt}}_{\text{MPC}}$ quite easily.[1]

The two protocols are used for securely computing a ranking of the candidates from their tallied number of votes and/or revealing only the desired part of the election result. The former is achieved by generating a comparison matrix. For n_{options} electable options in a voting, an $n_{\text{options}} \times n_{\text{options}}$ encrypted comparison matrix for pairwise $\geq$ comparisons is created. Clearly, the diagonal values will all decrypt to 1 since $\forall x : x \geq x$ holds. Furthermore, $x \geq y$ can be computed from $y \geq x$ once the result of $x == y$ was determined. Therefore, only $\frac{(n_{\text{options}}-1)\cdot n_{\text{options}}}{2}$ comparisons based on $P^{\text{gt}}_{\text{MPC}}$ are necessary, and the other values can be deduced from the result of $\frac{(n_{\text{options}}-1)\cdot n_{\text{options}}}{2}$ runs of (the cheaper protocol) $P^{\text{eq}}_{\text{MPC}}$. The matrix allows to compute the number of pairwise wins of each candidate. E.g., for an option that received the fewest votes, the only pairwise win is the one compared to itself; similarly parties that received the most votes have n_{options} pairwise wins. Depending on the concrete result function for the voting, additional tests may be required. If only the options on the first place shall be disclosed, then additional runs of $P^{\text{eq}}_{\text{MPC}}$ are necessary to find which parties' pairwise wins equal n_{options}. Many useful result functions are conceivable, some of which are discussed in [33].

Both protocols consist of an "offline" and "online" phase, respectively. The offline phase resembles a preprocessing (i.e., setup) phase, while the online phase refers to the actual usage of the protocols to compare encrypted numbers. Due to the construction of the protocols, randomness of certain shapes needs to be prepared during the offline phases, respectively. In order to remove the trusted party from the instantiation, this randomness needs to be computed by the trustees in a secure manner. In the course of this chapter, we introduce the mechanisms of $P^{\text{gt}}_{\text{MPC}}$ and $P^{\text{eq}}_{\text{MPC}}$ before focusing on generation of the necessary randomness, its implementation and an evaluation.

[1]For consistency we stick to the naming used by the other authors.

6.1 Comparing Encrypted Numbers

Lipmaa and Toft [34] proposed their protocols for arithmetic black boxes (ABB). For the purpose at hand, the ABB is instantiated by the Paillier encryption scheme. ABBs essentially provide functionality for secure arithmetic operations on secret values. Recall the homomorphicity of Paillier's encryption scheme (section 2.2.3), that allows for addition of two encrypted values and (as a result) for multiplication of encrypted values with scalars. Thus, two encrypted plaintexts can be added without revealing anything but the ciphertext of their encrypted sum and (in our setup) the parties sharing the secret key could decrypt the sum if desired. Secure multiplication of two secret values is less straightforward and requires interaction of the parties. In this section we assume that a multiplication protocol for Paillier exists, but introduce it only later in section 6.2.

Following the notation of [34], secrets stored in the ABB (thus, encrypted values) are written in brackets $[\![ ]\!]$ and arithmetic operations are written infix while actually they may refer to a whole protocol that realizes the operation. Due to the use of the Paillier encryption scheme, all operations are carried out mod N.

6.1.1 Equality of Encrypted Numbers

Trivially, equality $x == y$ can be reformulated as $x - y == 0$. $[\![x - y]\!]$ can be computed from $[\![x]\!]$, $[\![y]\!]$ using the additive homomorphicity of Paillier's encryption scheme. Thus, the equality tests (denoted as $P^{\text{eq}}_{\text{MPC}}$) actually focus on testing whether a ciphertext encrypts 0. The equality test for Küsters et al.'s instantiation of Ordinos is based on Lipmaa and Toft's test from Hamming distance, more precisely, the variant for bounded inputs [34, section 3.1]. The test works as follows:

1. Let $l = \lceil \log_2(N) \rceil$ and n be the number of parties.
2. Assuming that the parties share the encryption of an unknown (random) r: Mask $[\![x - y]\!]$ by adding r, i.e., compute $[\![m]\!] = [\![x - y + r]\!]$ and reveal (decrypt) m. Now, if $x == y$ holds, then $m == r$ holds.
3. Further assuming, that the parties have encryptions of r's bits, $[\![r_i]\!]$, such that $r = \sum_{i=0}^{l-1} r_i$ and $r_i \in \{0, 1\}$, they can compute the Hamming distance $[\![H]\!]$ of m and r as $[\![H]\!] = \sum_{i=0}^{l-1} [\![r_i]\!] \oplus m_i$. The xor-operation ($\oplus$) can be computed as $[\![r_i]\!] \oplus m_i = m_i + [\![r_i]\!] - 2 \cdot m_i \cdot [\![r_i]\!]$.
4. Clearly, $H == 0$ would hold if $r == m$. However, H cannot be revealed since it could disclose information about the relationship of m and r. The parties only need to find out whether H is 0. This is done using the polynomial $P_l(x) = \sum_{i=0}^{l} \alpha_i \cdot x^i$, that maps 1 to 1 and $x \in \{2, \ldots, l + 1\}$ to zero.[2] The goal is to evaluate $P_l(H + 1)$. Normally, evaluating the polynomial would require a number of multiplications in the online phase, which is avoided by the use of another random value $[\![R]\!]$, its powers $[\![R^0]\!], \ldots, [\![R^l]\!]$ and its inverse $[\![R^{-1}]\!]$ (i.e., $R \in \mathbb{Z}_N^*$). Now, one multiplication is carried out during the online phase and $m_H = [\![R^{-1}]\!] \cdot [\![H + 1]\!]$ is revealed. Then the parties can locally compute the powers of $H + 1$

[2]The polynomial can be constructed using general interpolation techniques. An efficient way for computing the coefficients is provided in Appendix B.

from m_H: $m_H^i \cdot [\![R^i]\!] = (R^{-1} \cdot (H+1))^i \cdot [\![R^i]\!] = [\![R^{-i} \cdot R^i \cdot (H+1)^i]\!] = [\![(H+1)^i]\!]$. Finally, $[\![P_l(H+1)]\!]$, which yields the desired result, can be computed locally by scalar multiplication with the coefficients of P_l and summation.

Security of the protocol is intuitively given by the use of Paillier as an ABB and the fact that both revealed values m and m_H are masked by random values r and R^{-1}, respectively and thus also random values.

For inputs bounded by 2^l, where $2^{l+k+\log_2 n} \ll N$, n is the number of parties and k is a statistical security parameter[3], $P_{\text{MPC}}^{\text{eq}}$ (denoted as $P_{\text{MPC}}^{\text{eq},l}$) can be adapted to reduce the work:

- Each party provides a random k-bit value as $[\![r^{(j)}]\!]$.
- The parties randomly generate only the least significant l bits $[\![r_i]\!]$ of r.
- r is computed as $[\![r]\!] = \sum_{j=1}^{n} [\![r^{(j)}]\!] \cdot 2^l + \sum_{i=0}^{l-1} 2^i \cdot [\![r_i]\!]$.

In this case, r statistically masks $x - y$ since $m \mod 2^l$ is random as before and the carry-bit of the addition $x - y + r$ is masked by $r^{(j)}$ of any honest party P_j, which makes $\lfloor m/2^l \rfloor$ indistinguishable from a random number.

The offline phase of the protocol consists of generating $[\![r]\!], [\![r_0]\!], \ldots, [\![r_{l-1}]\!], [\![R]\!], [\![R^{-1}]\!], \ldots, [\![R^l]\!]$ and the coefficients $\alpha_0, \ldots, \alpha_l$ of $P_l(x)$.

6.1.2 Greater-Than Comparison of Encrypted Numbers

The greater-than test employed in Küsters et al.'s instantiation of Ordinos is based on section 4 of [34] (denoted as $P_{\text{MPC}}^{\text{gt}}$), which is an improved version of the test presented in [50].

$P_{\text{MPC}}^{\text{gt}}$ requires multiple instances of $P_{\text{MPC}}^{\text{eq},l}$ for different l. Figuratively, though not literally, it recursively compares the most significant $\frac{l}{2}$ bits of two l-bit values and in case they are equal continues to compare the least significant $\frac{l}{2}$ bits. The test is carried out as follows:

1. Let x, y be l-bit numbers and assume that l is a power of 2 and $2^{l+k+\log_2 n} \ll N$[4], where n is the number of parties and k is a statistical security parameter.
2. If $l == 1$ return $[\![b]\!] = 1 - [\![y]\!] + [\![x]\!] \cdot [\![y]\!]$. This will later also break the recursion. Trivially, b reflects $x \geq y$.
3. Otherwise, the protocol aims to transform the comparison of the l-bit values into comparisons of $\frac{l}{2}$-bit values: First, the parties compute $[\![z]\!] = 2^l + [\![x]\!] - [\![y]\!]$ and $[\![m]\!] = [\![z]\!] + [\![R^{(l)}]\!]$, where $R^{(l)}$ is a precomputed random $(l+k)$-bit value. m is revealed to the parties. Clearly, only if $x \geq y$, then $z \geq 2^l$ and subsequently $m \geq 2^l + R^{(l)}$. Put differently, the $(l+1)^{\text{th}}$ bit of z determines whether $x \geq y$ holds. It can be retrieved by computing $\lfloor z/2^l \rfloor$, which could be decrypted from $2^{-l}([\![z]\!] - [\![z \mod 2^l]\!])$. Thus, the parties need to compute $[\![z \mod 2^l]\!]$, for which holds $z \mod 2^l \equiv m - R^{(l)} \mod 2^l$. In the next steps the parties will be able to

[3] which is the case in virtually any voting, since the number of votes is usually much lower than a secure modulus N

[4] This can always be achieved by viewing x, y as numbers with larger bit length. Usually 2^l will be much smaller than N, otherwise N can normally be chosen arbitrarily large to achieve that.

determine $[\![m \mod 2^l - R^{(l)} \mod 2^l]\!]$. However, due to the reduction mod 2^l *before* the subtraction, 2^l needs to be added in case the difference is negative. Thus, the parties need to compare $R^{(l)} \mod 2^l$ and $m \mod 2^l$.

4. Assume, that $[\![R^{(l)}]\!]$ was precomputed such that the parties also hold $[\![r_\perp^{(l)}]\!]$ and $[\![r_\top^{(l)}]\!]$ where $r_\perp^{(l)}$ resembles the $\frac{l}{2}$ least significant bits of $R^{(l)}$ and $r_\top^{(l)}$ the following $\frac{l}{2}$ bits of $R^{(l)}$. I.e., $2^{l/2} \cdot r_\top^{(l)} + r_\perp^{(l)}$ are the l least significant bits of $R^{(l)}$. The parties similarly determine $m_\perp = m \mod 2^{l/2}$ and $m_\top = \left\lfloor \frac{m}{2^{l/2}} \right\rfloor \mod 2^{l/2}$. In an intermediate step, they compute $[\![b]\!]$ where $b = m_\perp \leq r_\perp^{(l)}$ using $P_{\text{MPC}}^{\text{eq}, \frac{l}{2}}$.

5. Now the parties want to continue comparing either $m_\top, r_\top^{(l)}$ or $m_\perp, r_\perp^{(l)}$ depending on b. However, they should not learn which half they continue with, because otherwise they could gain knowledge about where exactly x and y differ. Therefore, they securely compute $[\![\tilde{m}]\!] = [\![b]\!] \cdot (m_\perp - m_\top) + m_\top$ and $[\![\tilde{r}]\!] = [\![b]\!] \cdot ([\![r_{,\perp}^{(l)}]\!] - [\![r_\top^{(l)}]\!]) + [\![r_\top^{(l)}]\!]$. Then they recursively run $P_{\text{MPC}}^{\text{gt}}$ for the $\frac{l}{2}$-bit values $[\![\tilde{m}]\!], [\![\tilde{r}]\!]$ and receive $[\![f]\!] = 1 - [\![\tilde{m} \geq \tilde{r}]\!]$. In the following computation the bit f determines whether 2^l needs to be added as explained in step 3. (f is 1 if the difference is negative and 0 otherwise.)

6. The parties finally compute $[\![z \mod 2^l]\!] = ((m \mod 2^l) - (2^{l/2}[\![r_\top^{(l)}]\!] + [\![r_\perp^{(l)}]\!]) + 2^l \cdot [\![f]\!]$ and the result $[\![b]\!] = [\![x \geq y]\!] = 2^{-l}([\![z]\!] - [\![z \mod 2^l]\!]))$.

The offline phase consists of the offline phases for $P_{\text{MPC}}^{\text{eq}, \frac{l'}{2}}$ and generating $[\![R^{(l')}]\!], [\![r_\top^{(l')}]\!]$, and $[\![r_\perp^{(l')}]\!]$ for $l' \in \{2^1, 2^2, ..., l\}$.

6.2 Paillier as an ABB

Due to the homomorphicity of Paillier's encryption scheme (see section 2.2.3), some operations of the ABB can be implemented very easily, however others require additional work. The following explains implementation of all black box operations seen in $P_{\text{MPC}}^{\text{eq}}$and $P_{\text{MPC}}^{\text{gt}}$. $Enc_{(r_a)}(x)$ denotes the ciphertext of x encrypted in some instance of Paillier's encryption scheme using randomness r_a. Note that plaintexts are viewed mod N.

- Open addition: $[\![x]\!] + y = y + [\![x]\!] = Enc_{(r_a)}(x) \cdot Enc_{(0)}(y) \mod N^2$

  A public number can be added to a number stored in the ABB by encrypting the public number without using randomness and then multiplying the ciphertexts. Omitting the randomness when encrypting y ensures that all parties deduce the same result.

- Secure addition: $[\![x]\!] + [\![y]\!] = [\![y]\!] + [\![x]\!] = Enc_{(r_a)}(x) \cdot Enc_{(r_b)}(y) \mod N^2$

  Two numbers stored in the ABB can be added by multiplying the corresponding ciphertexts.

- Open subtraction: $[\![x]\!] - y = -y + [\![x]\!] = Enc_{(r_a)}(x) \cdot Enc_{(0)}(-y) \mod N^2$

- Secure subtraction: $[\![x]\!] - [\![y]\!] = -[\![y]\!] + [\![x]\!] = Enc_{(r_a)}(x) \cdot Enc_{(r_b)}(y)^{-1} \mod N^2$ (see multiplication)

- Open multiplication: $[\![x]\!] \cdot y = y \cdot [\![x]\!] = Enc_{(r_a)}(x)^y \mod N^2$

  A number stored in the ABB can be multiplied by a public number by raising the ciphertext to the respective power.

- Secure multiplication: Multiplication of two numbers stored in the ABB needs some more effort. The idea of the protocol $Mul(]\!]x]\!], [\![y]\!])$ is due to [11]:

  1. Each party P_i selects a random value $r_i \in \mathbb{Z}_N$ and publishes $[\![r_i]\!]$ and $[\![r_i \cdot y]\!]$.
  2. The parties compute $[\![x + \sum_{i=0}^{n} r_i]\!]$.
  3. The parties then decrypt and disclose $x + \sum_{i=0}^{n} r_i$ before computing $[\![y]\!] \cdot (x + \sum_{i=0}^{n} r_i) = [\![x \cdot y + y \cdot \sum_{i=0}^{n} r_i]\!]$.
  4. Finally the parties compute and return the result $[\![x \cdot y]\!] = [\![x \cdot y + y \cdot \sum_{i=0}^{n} r_i]\!] - [\![\sum_{i=0}^{n} r_i \cdot y]\!]$.

  Security against semi-honest parties is given since only one value is revealed, which is additively hidden by an unknown random value. Making the protocol secure against adversaries that deviate from the protocol requires that the parties provide proof that $[\![r_i]\!]$ is a correct encryption (i.e., that they know the corresponding plaintext and randomness) and proof of multiplicative relation of $[\![r_i \cdot y]\!]$, $[\![r_i]\!]$, and $[\![y]\!]$. Therefore, the remainder of this section is dedicated to construction of appropriate zero-knowledge proofs.

- Exponentiation: Raising a number stored in the ABB to a known power, say computing $[\![x]\!]^a$, can be based on $Mul()$ by implementing it as a series of $n-1$ multiplications. $[\![x^2]\!] = Mul([\![x]\!], [\![x]\!]), [\![x^3]\!] = Mul([\![x^2]\!], [\![x]\!]), \ldots, [\![x^{a-1}]\!] = Mul([\![x^{a-2}]\!], [\![x]\!]), [\![x]\!]^a = [\![x^a]\!] = Mul([\![x^{a-1}]\!], [\![x]\!])$. A faster variant can be implemented analogously to fast exponentiation. However, since $P_{\text{MPC}}^{\text{eq}}$ needs the intermediate results as well, a chain of multiplications suffices.

Zero-Knowledge Proof of Knowledge of Paillier Plaintext

A zero-knowledge proof of knowledge of plaintext for a known ciphertext is due to [11]. Assume, that the prover knows x, r_a such that $C = Enc_{(r)}(x) = (N+1)^x \cdot r_a^N \mod N^2$, where N is a public Paillier key, and shall prove knowledge of the plaintext of C. The protocol for this proof is depicted in Figure 6.1.

Prover		**Verifier**
$y \in_R \mathbb{Z}_N, r_b \in \mathbb{Z}_N^*$	$\xrightarrow{B=(N+1)^y \cdot r_b^N \mod N^2}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}$
$r = y + c \cdot x \mod N$ $t = \frac{y+c\cdot x-r}{N}$ $r' = r_b \cdot r_a^c \cdot (N+1)^t \mod N^2$	$\xrightarrow{r,r'}$	$(N+1)^r \cdot (r')^N \overset{?}{\equiv} BC^c \mod N^2$

Figure 6.1: $\pi_{\text{plaintext}}^{\kappa}$: Zero-knowledge proof of knowledge of Paillier plaintext

Zero-Knowledge Proof of Multiplicative Relation of Paillier Ciphertexts

A zero-knowledge proof for correctness of multiplication is also found in [11]. Note that there are other proofs available, that require, however, knowledge of both plaintext factors of the product and are thus not suitable for the task at hand [see, e.g., 17]. For the protocol of this proof, assume that the parties know the ciphertext $A = (N+1)^x \cdot r_a^N \mod N^2$, the prover has published a ciphertext $B = (N+1)^y \cdot r_b^N \mod N^2$ and is required to prove that some ciphertext C encrypts the product xy. Note that a simple way to construct C is setting $C = A^y \cdot r_c^N \mod N^2 (\equiv (N+1)^{xy} \cdot (r_a^y \cdot r_c)^N \mod N^2)$. The protocol for this proof is depicted in Figure 6.2.

Prover		**Verifier**
$z \in_R \mathbb{Z}_N,\ r_d, r_e \in \mathbb{Z}_N^*$	$\xrightarrow{D=A^z \cdot r_d^N \mod N^2,\ E=(N+1)^z \cdot r_e^N \mod N^2}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}$
$r = z + c \cdot y \mod N$		
$t = \frac{z+c\cdot y-r}{N}$		
$r' = r_e \cdot r_b^c \cdot (N+1)^t \mod N^2$		
$r'' = r_d \cdot A^t \cdot r_c^c \mod N^2$	$\xrightarrow{r, r'}$	$(N+1)^r \cdot (r')^N \overset{?}{\equiv} EB^c \mod N^2$
		$A^r \cdot (r'')^N \overset{?}{\equiv} DC^c \mod N^2$

Figure 6.2: $\pi_{\text{PaillierMult}}$: Zero-knowledge proof of correct multiplication of Paillier plaintexts

Note that $\pi_{\text{PaillierMult}}$ includes $\pi^{\kappa}_{\text{plaintext}}$ for B, which ensures that B is actually a valid Paillier ciphertext (and the Prover knows the corresponding plaintext). The implicit sub-protocol is characterized by E, c, and r' with the corresponding check. Thus, the presented multiplication protocol requires parties only to provide a proof for correct multiplication. As before, Fiat-Shamir transform (see section 2.4.7) can be used to make the proofs non-interactive.

6.3 Securely Generating Randomness

Generating the desired randomness in the offline phases of both protocols requires some elementary functionality, that is introduced in this section. Lipmaa and Toft [34] suggest that this functionality could be implemented as described in [15]. The authors of [15] describe a method to securely compute the bit-decomposition of a shared secret $[\![x]\!]$, i.e., to determine $[\![x_i]\!]$ for $i \in \{0, \ldots, \lceil \log_2(x) \rceil\}$, such that x_i are the bits composing x. This allows for bit-wise computations on shared secrets. However, their method does not appear appropriate for our purpose: The protocol presented in [15] is rather complex and elaborate, consisting of several sub-protocols. It is constructed for operating on secrets modulo a prime. We are unaware of whether it can be adopted without changes for secrets modulo composites as RSA moduli used in Paillier's encryption scheme. Furthermore, we have two advantages that allow for a simpler construction: x is not fixed, but rather needs to be constructed by the parties and the randomness is generated in the offline phase of the protocols. The latter indicates that best possible time efficiency is not necessarily of the essence and a practicable, simpler solution may suffice. Therefore, we focus on a construction for generating bits $[\![x_i]\!]$ and determining $[\![x]\!]$ only afterwards. That way, random l-bit values can be generated rather easily. Additionally, we need a method for generating $[\![R]\!]$ such that $R \in_R \mathbb{Z}_N^*$ and the inverse $[\![R^{-1}]\!]$.

6.3.1 Composing Random l-bit Values

Jointly generate a bit

Clearly, a single bit $[\![b]\!]$ can be generated by having the Parties P_j for $j \in \{1, ..., n\}$ each provide a bit $[\![b_j]\!]$ and computing $\bigoplus_{j=1}^{n} [\![b_j]\!]$. There are several (at least 3) ways to implement this [48]. One way to compute $\oplus$ was already used in $P_{\text{MPC}}^{\text{eq}}$ (with a revealed b there): $[\![a]\!] \oplus [\![b]\!] = [\![a]\!] + [\![b]\!] - 2 \cdot [\![a]\!] \cdot [\![b]\!]$. For the secure multiplication we may either use the multiplication protocol $Mult()$ or let $n-1$ parties multiply intermediate results by their secret and publish a proof based on $\pi_{\text{PaillierMult}}$. For simplicity and in anticipation of a relatively low number of trustees, the latter method shall suffice for our purpose.

In order to have security against malicious parties, a method of proving that $[\![b_j]\!]$ actually encrypts $b_j \in \{0, 1\}$ is necessary. Each party will then additionally to $[\![b_j]\!]$ provide proof, that b_j is a bit. [18] provides an (honest-verifier) zero-knowledge proof that convinces the verifier that some ciphertext encrypts exactly one out of two plaintexts. This proof is based on an (honest-verifier) zero-knowledge proof that some ciphertext encrypts zero and a technique from [13]. Essentially that leaves the prover to show that either $[\![b_j]\!]$ or $[\![b_j]\!]/(N+1)$ encrypts 0, which convinces the verifier, that $b_j \in \{0, 1\}$. Due to the construction of Paillier's encryption scheme, proving that a ciphertext encrypts zero is the same as proving that it is an n^{th} power, which is done by proving knowledge of the n^{th} root. Protocols for both proofs $\pi_{n^{\text{th}}\text{Root}}^{\kappa}$ and $\pi_{\text{PaillierBit}}^{\kappa}$ are depicted in Figure 6.3 and Figure 6.4, respectively, assuming that the prover shall prove a ciphertext $C = (N+1)^{b_j} \cdot (r_a)^N \bmod N^2$, that is known to both parties, encrypts $b_j \in \{0, 1\}$ ($b_j = 0$ for $\pi_{n^{\text{th}}Root}^{\kappa}$). It is assumed that the challenge c is bounded by 2^t, which is smaller than both factors of N. This is necessary for constructing a knowledge extractor for the proof of knowledge.

$\pi_{n^{\text{th}}\text{Root}}^{\kappa}$ is straight forward, as it is similar to previous proofs (such as $\pi_{\text{dlog}}^{\kappa}$ and $\pi_{\text{plaintext}}^{\kappa}$). It can trivially be transformed into a proof that a ciphertext C' encrypts a specific plaintext x' by proving that $C'/(N+1)^{x'}$ encrypts zero. This fact is utilized by $\pi_{\text{PaillierBit}}^{\kappa}$ for proving that C encrypts one out of two known plaintexts. Clearly, the prover has good chances to generate a valid reply for only one of the two ciphertexts. The "trick" is that the prover generates one transcript of $\pi_{n^{\text{th}}\text{Root}}^{\kappa}$ all by himself for the ciphertext that he cannot prove to encrypt zero. For that he uses the honest verifier simulator (see section 2.4.5) for $\pi_{n^{\text{th}}\text{Root}}^{\kappa}$, which enables him to generate an accepted transcript even though the ciphertext does not encrypt zero. The other ciphertext is proven to encrypt zero in interaction with the verifier as usual and finally the verifier has two transcripts to verify but does not know which was generated by the simulator. Thus, both transcripts are valid and one was created in a run of $\pi_{n^{\text{th}}\text{Root}}^{\kappa}$, which proves that C encrypts either 0 or 1. Note that the same proof can be used for one out of two arbitrary plaintexts, which is interesting, e.g., for proving correctness of ballots in e-voting [18].

Prover		**Verifier**
$r_b \in \mathbb{Z}_N^*$	$\xrightarrow{B=(N+1)^0\cdot r_b^N \mod N^2}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}_{2^t}$
$r = r_b \cdot r_a^c \mod N$	$\xrightarrow{r}$	$(N+1)^0 \cdot (r)^N \overset{?}{\equiv} BC^c \mod N^2$
		$gcd(N,C) \overset{?}{=} gcd(N,B) \overset{?}{=} gcd(N,r) \overset{?}{=} 1$

Figure 6.3: $\pi^{\kappa}_{n^{\text{th}}\text{Root}}$: Zero-knowledge proof of knowledge of n^{th} root

Prover		**Verifier**
$C_0 = C, C_1 = C/(N+1)$		$C_0 = C, C_1 = C/(N+1)$
$r' \in \mathbb{Z}_N^*$		
$B_{b_j} = (N+1)^0 \cdot r'^N \mod N^2$		
$(B_{1-b_j}, c_{1-b_j}, r_{1-b_j}) :=$		
Simulate $\pi^{\kappa}_{n^{\text{th}}\text{Root}}$ on C_{1-b_j}	$\xrightarrow{B_0, B_1}$	
	$\xleftarrow{c}$	$c \in_R \mathbb{Z}_{2^t}$
$c_{b_j} = c - c_{1-b_j} \mod 2^t$		
$r_{b_j} = r' \cdot r_a^{c_{b_j}} \mod N$	$\xrightarrow{c_0, c_1, r_0, r_1}$	$(N+1)^0 \cdot (r_0)^N \overset{?}{\equiv} B_0 C_0^{c_0} \mod N^2$
		$(N+1)^0 \cdot (r_1)^N \overset{?}{\equiv} B_1 C_1^{c_1} \mod N^2$
		$c = c_0 + c_1 \mod 2^t$
		$gcd(N,C_0) \overset{?}{=} gcd(N,C_1) \overset{?}{=} 1$
		$gcd(N,B_0) \overset{?}{=} gcd(N,B_1) \overset{?}{=} 1$
		$gcd(N,r_0) \overset{?}{=} gcd(N,r_1) \overset{?}{=} 1$

Figure 6.4: $\pi^{\kappa}_{\text{PaillierBit}}$: Zero-knowledge proof of knowledge of 1-out-of-2 n^{th} roots

Compose l bits

Now that we have means to generate encrypted random bits, the goal is to combine them to l-bit values. The method is straightforward:

1. The parties jointly generate l random bits $[\![b_0]\!], \ldots, [\![b_{l-1}]\!]$ and prove correct behavior in each step by the proofs introduced in the previous section. They verify each of the provided proofs and if a proof is found to be invalid by a majority of the parties, the cheating party is excluded from the protocol.[5]
2. Everyone can then compute the l-bit value r as $r := \sum_{i=0}^{l-1} 2^i \cdot [\![b_i]\!]$.

[5]Thus creating security against a dishonest minority. With a few adaptations the construction may gain stronger security.

Very similarly, single parties can provide an l-bit value $[\![r_i]\!]$ and prove its length by providing every encrypted bit $[\![b_0]\!], ..., [\![b_{l-1}]\!]$, such that $r_i := \sum_{i=0}^{l-1} 2^i \cdot [\![b_i]\!]$ along with the proofs that each b_i is indeed a bit. In that case the other parties verify the provided proofs based on $\pi^{\kappa}_{\text{PaillierBit}}$ and are able to detect cheating by P_i.

6.3.2 Generating Randomness in $\mathbb{Z}_N$

In theory the same methods could be used to generate randomness up to $2^{\lfloor \log_2 N \rfloor}$. However, that would induce a large overhead for generating every single bit and the respective proofs. The encryption of a random number $R \in \mathbb{Z}_N$ can obviously be generated more efficiently by letting each party P_i encrypt a random $R_i \in \mathbb{Z}_N$ and prove knowledge of plaintext based on $\pi^{\kappa}_{\text{plaintext}}$. Then $[\![R]\!] = \sum_{i=1}^{n} [\![R_i]\!]$ encrypts a random number jointly chosen from $\mathbb{Z}_N$. Clearly $R = \sum_{i=0}^{n} R_i \mod N$ is random as long as at least one party chooses R_i randomly. A similar method for secrets modulo a prime is mentioned in [40].

6.3.3 Inverting a Random Secret

Additionally, a method for generating secrets along with their inverse mod N is necessary. This obviously requires that the secret is in $\mathbb{Z}_N^*$. The probability that a random number $R \in \mathbb{Z}_N$ lies in $\mathbb{Z}_N^*$ is quite high for an RSA modulus N: $\frac{\phi(N)}{N} = \frac{(p-1)\cdot(q-1)}{pq} = \frac{pq-p-q+1}{pq} = 1 - \frac{1}{q} - \frac{1}{p} + \frac{1}{pq}$ is very close to 1 for large primes p, q. Thus, the protocol can be built on the method for generating random numbers in $\mathbb{Z}_N$. A random number and its inverse are generated as follows [see, e.g., 12]:

1. Generate two random secrets $[\![R]\!], [\![R']\!]$ using the method above.
2. Securely compute $[\![R \cdot R']\!]$ using the $Mult()$ protocol and reveal $R \cdot R' \mod N$.
3. If $R \cdot R' \mod N = 1$, then return R and its inverse R'. Otherwise check that $gcd(R \cdot R', N) = 1$. This is the case if $R, R' \in \mathbb{Z}_N^*$ and occurs with a high probability for an RSA modulus N. In the unlikely case $gcd(R \cdot R', N) \neq 1$: Restart the protocol from scratch.
4. Then compute $(R \cdot R')^{-1} \mod N$[6] and return $[\![R]\!], (R \cdot R')^{-1} \cdot [\![R']\!]$. In that case the inverse of R is encrypted by $(R \cdot R')^{-1} \cdot [\![R']\!] = [\![R^{-1} \cdot R'^{-1} \cdot R']\!] = [\![R^{-1}]\!]$.

6.4 Implementation & Evaluation

Based on the presented techniques and proofs, we implemented preprocessing for $P^{\text{eq}}_{\text{MPC}}$ and $P^{\text{gt}}_{\text{MPC}}$ as protocols for the trustees in the existing Ordinos implementation. Just as for the key generation protocol, we use the modules `gmpy2`, `secrets`, and hashes based on `SHA512`. The network connection is also realized analogously and all zero-knowledge proofs are implemented as non-interactive proofs based on the hash function. The protocol was somewhat optimized regarding communication in terms that upon joint generation of random bits, the parties communicate (broadcast) only once instead of communicating for each single bit. I.e., they generate and broadcast two-dimensional

[6]The inverse modulo N can be computed efficiently using the well known extended Euclidean algorithm.

arrays of encrypted bits and the corresponding proofs where one dimension reflects the bit length l and the other reflects the number of random l-bit elements to be generated. Similarly, the parties provide all their encrypted bits for their individual k-bit contributions and the corresponding proofs at once.

For the evaluation, the statistical security parameter k is fixed to 40. As before, the setup consists of three trustees operating in a local network with a decryption threshold of 2. In order to obtain results for a realistic voting the key length is fixed to 2048 bits. As noted in the chapter introduction, $\frac{(n_{\text{options}}-1)\cdot n_{\text{options}}}{2}$ runs of each $P_{\text{MPC}}^{\text{eq}}$ and $P_{\text{MPC}}^{\text{gt}}$ are necessary to generate the ranking matrix. Additional runs of the protocols may be necessary to compute the result function based on the ranking matrix. For the evaluation, we assume a result function that reveals the options on the first k positions and requires n_{options} additional runs of $P_{\text{MPC}}^{\text{gt}}$.[7] This resembles the most costly tally-hiding result function presented in [33], that does not reveal a ranking. Obviously, the runtime of preprocessing for both protocols depends on the bit length of the RSA modulus, the number of options n_{options} as well as the number of trustees (most notably) due to the secure multiplication protocol and verifiable decryptions. Also the bit length of the compared numbers, which is determined by the maximum number of votes per option, influences the runtime of both protocols, since more random bits need to be generated by the parties and more recursive preprocessing is necessary for larger bit lengths in $P_{\text{MPC}}^{\text{eq}}$ and $P_{\text{MPC}}^{\text{gt}}$, respectively. Although there is a probabilistic component during the generation of random invertible elements, we can assume it negligible, since the probability that a random element is not invertible is extremely low. The evaluation includes the computation of the coefficients for the polynomial used in $P_{\text{MPC}}^{\text{eq}}$, which is constant for each l and does not depend on n_{options}. We re-used the existing implementation from the trusted party, which follows the idea presented in Appendix B. The runtime consumed for the local computation of the coefficients by each party is negligible (sub-second in total).

Runtime of the whole preprocessing for a voting, thus preprocessing for $\frac{(n_{\text{options}}-1)\cdot n_{\text{options}}}{2} + n_{\text{options}}$ runs of $P_{\text{MPC}}^{\text{gt}}$ and $\frac{(n_{\text{options}}-1)\cdot n_{\text{options}}}{2}$ runs of $P_{\text{MPC}}^{\text{eq}}$ with different numbers of options (n_{options}) and bit lengths (l) is depicted in Figure 6.5. For $n_{\text{options}} \leq 10$ preprocessing took less than 2 hours for every choice of bit length l. Even for the preparation of a very large-scaled voting, that allows for more than 4 billion ($2^{32} - 1$) votes per option and 40 electable options, a runtime of just less than 22 hours was measured. Note that the lines for 32 and 16 bits complement (to some extent) the results of Küsters et al. [33], who evaluated runtime of the tallying phase for 5 electable options when working with 32 bit integers as well as for the same choices of n_{options} as presented here with 16 bit integers.

Additionally to runtime, we measured the size of the prepared values when stored in memory. For the largest-scaled voting approximately 116 MiB of memory were consumed in total. For votings with less than 10 options less than 8 MiB of memory were consumed for storing all generated values. Not surprisingly, the growth behavior is very similar to that observed for runtime of the protocols, since both depend on the same parameters in a very similar way. Clearly, this memory consumption is independent of the implementation of the protocols and interesting for the decision whether prepared randomness should be preloaded to or kept in memory to obtain a (slightly) improved performance in the online phase.

[7] The options of the first k positions are revealed by using $P_{\text{MPC}}^{\text{gt}}$ to compare the number of pairwise wins of each option to $[\![n_{\text{options}} - k + 1]\!]$ and decrypting the resulting bit. See [33]

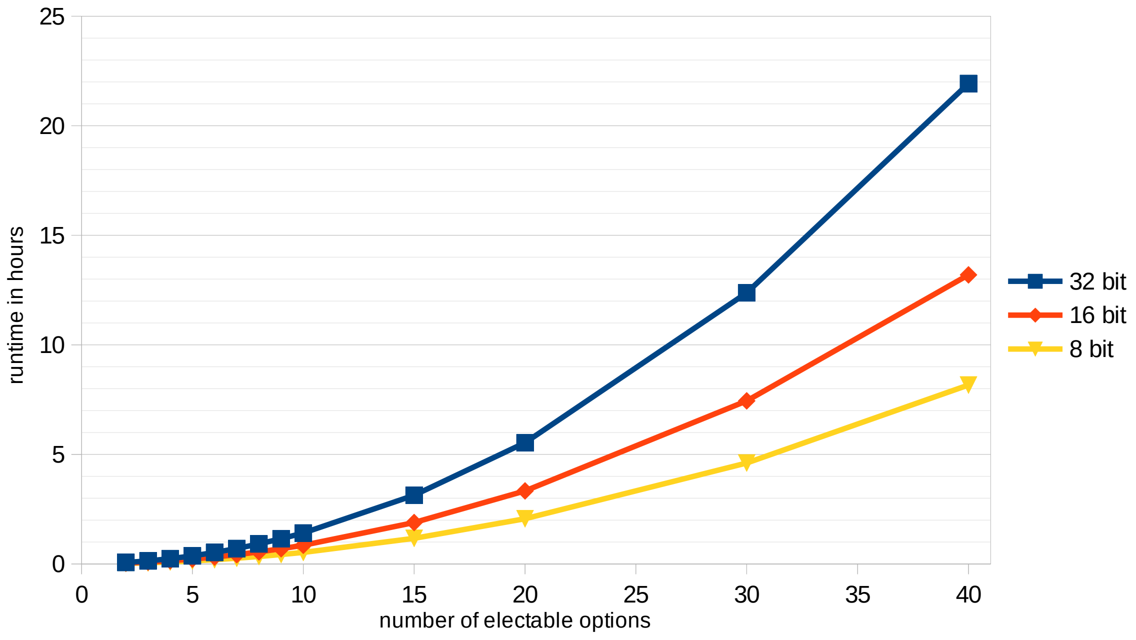

Figure 6.5: Runtime of preprocessing for $n_{\text{options}} = 2$ to 40 and $l \in \{8, 16, 32\}$ based on a 2048 bit modulus

7 Conclusion

A key generation protocol for threshold Paillier and protocols for the preparation of randomness with security in the honest-majority setting were implemented successfully. This allows for removal of the trusted party from the proof of concept implementation of Ordinos, thus avoiding potential security risks and trust problems induced by its usage.

We evaluated the implementations for key generation and preparation of randomness regarding runtime among 3 trustees with a threshold of 2. Using everyday hardware and one thread per trustee in a local network, the expected average runtime of the secure key generation for a key length of 2048 bit is around 95 minutes. The preparation of randomness depends on the parameters of the voting, especially on the number of electable options and voters. For these preparations we measured a runtime of up to 22 hours, when preparing a large-scaled voting. Considerably less time was consumed for preparation of votings with lower vote count or less electable options. Overall, the runtime for both key generation and preparations seems practical for a large variety of votings.

In an intermediate step we also implemented and evaluated a variant of the key generation protocol that provides security in the semi-honest setting and compared it to the most closely related existing implementation. For this implementation an improved runtime was achieved due to the use of multiple threads (i.e., one per party) and an adaptation that reduces communication overhead.

Future work could focus on an implementation of distributed key generation with security in the dishonest-majority setting as well as elimination of the restriction on the threshold in the present protocol. Then the protocol for key generation could be selected dynamically depending on the actual parameters of a voting in order to achieve the fastest possible key generation. As for generating randomness it may be interesting to implement more time efficient protocols if large-scaled e-votings at shorter notice were in demand. Alternatively or additionally all protocols could be implemented multi-threaded to reduce runtime of both key generation and preparations.

Bibliography

[1] B. Adida. “Helios: Web-Based Open-Audit Voting”. In: *USENIX security symposium*. Vol. 17. 2008, pp. 335–348 (cit. on p. 11).

[2] M. Agrawal, N. Kayal, N. Saxena. “PRIMES Is in P”. In: *Annals of Mathematics* (2004), pp. 781–793 (cit. on p. 35).

[3] M. Bellare, O. Goldreich. “On Defining Proofs of Knowledge”. In: Mar. 1999. DOI: 10.1007/3-540-48071-4_28 (cit. on p. 25).

[4] M. Ben-Or, S. Goldwasser, A. Wigderson. “Completeness Theorems for Non-cryptographic Fault-tolerant Distributed Computation (extended Abstract)”. In: Jan. 1988, pp. 1–10. DOI: 10.1145/62212.62213 (cit. on pp. 36, 60).

[5] D. Boneh, M. Franklin. “Efficient Generation of Shared RSA Keys”. In: *Advances in Cryptology — CRYPTO ’97*. Ed. by B. S. Kaliski. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 425–439. ISBN: 978-3-540-69528-8 (cit. on pp. 37–39).

[6] F. Boudot. “Efficient Proofs That a Committed Number Lies in an Interval”. In: *Advances in Cryptology — EUROCRYPT 2000*. Ed. by B. Preneel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 431–444. ISBN: 978-3-540-45539-4 (cit. on p. 30).

[7] N. de Bruijn. “On the Number of Uncancelled Elements in the Sieve of Eratosthenes”. In: *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen: Series A: Mathematical Sciences* 53.5-6 (1950), pp. 803–812 (cit. on p. 39).

[8] J. Camenisch, M. Michels. “Proving in Zero-Knowledge That a Number Is the Product of Two Safe Primes”. In: *Advances in Cryptology — EUROCRYPT ’99*. Ed. by J. Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 107–122. ISBN: 978-3-540-48910-8 (cit. on p. 26).

[9] R. Canetti, O. Goldreich, S. Halevi. “The Random Oracle Methodology, Revisited”. In: *Journal of the ACM* 51.4 (July 2004), pp. 557–594. ISSN: 0004-5411. DOI: 10.1145/1008731.1008734 (cit. on p. 31).

[10] R. Cramer, I. Damgård. “Zero-Knowledge Proofs for Finite Field Arithmetic, or: Can Zero-Knowledge Be for Free?” In: *Advances in Cryptology — CRYPTO ’98*. Ed. by H. Krawczyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 424–441. ISBN: 978-3-540-68462-6 (cit. on p. 28).

[11] R. Cramer, I. B. Damgård, J. B. Nielsen. *Multiparty Computation from Threshold Homomorphic Encryption*. RS-00-14. BRICS, June 2000. URL: https://www.brics.dk/RS/00/14/BRICS-RS-00-14.pdf (cit. on pp. 67, 68).

[12] R. Cramer, I. Damgård, J. B. Nielsen. “Multiparty Computation from Threshold Homomorphic Encryption”. In: *Advances in Cryptology — EUROCRYPT 2001*. Ed. by B. Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 280–300. ISBN: 978-3-540-44987-4 (cit. on p. 71).

[13] R. Cramer, I. Damgård, B. Schoenmakers. “Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols”. In: *Advances in Cryptology — CRYPTO ’94*. Ed. by Y. G. Desmedt. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 174–187. ISBN: 978-3-540-48658-9 (cit. on p. 69).

[14] I. Damgard. *On σ-Protocols* (cit. on p. 32).

[15] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, T. Toft. “Unconditionally Secure Constant-Rounds Multi-Party Computation for Equality, Comparison, Bits and Exponentiation”. In: *Theory of Cryptography*. Ed. by S. Halevi, T. Rabin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 285–304. ISBN: 978-3-540-32732-5 (cit. on p. 68).

[16] I. Damgård, M. Jurik. “A Generalisation, a Simplification and Some Applications of Paillier’s Probabilistic Public-key System”. In: *Lecture Notes in Computer Science* 7 (Dec. 2000). DOI: 10.7146/brics.v7i45.20212 (cit. on pp. 16, 20).

[17] I. Damgård, M. Jurik. “A Generalisation, a Simplification and Some Applications of Paillier’s Probabilistic Public-key System”. In: *Public Key Cryptography*. Ed. by K. Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 119–136. ISBN: 978-3-540-44586-9 (cit. on pp. 33, 68).

[18] I. Damgård, M. Jurik, J. B. Nielsen. “A Generalization of Paillier’s Public-Key System with Applications to Electronic Voting”. In: *International Journal of Information Security* 9.6 (Dec. 2010), pp. 371–385. ISSN: 1615-5270. DOI: 10.1007/s10207-010-0119-9 (cit. on pp. 16, 69).

[19] I. Damgård, M. Koprowski. “Practical Threshold RSA Signatures without a Trusted Dealer”. In: *Advances in Cryptology — EUROCRYPT 2001*. Ed. by B. Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 152–165. ISBN: 978-3-540-44987-4 (cit. on p. 35).

[20] H. Delfs, H. Knebl. *Introduction to Cryptography: Principles and Applications*. Berlin, Heidelberg, 2015. DOI: 10.1007/978-3-662-47974-2 (cit. on pp. 23, 25).

[21] D. Demirel, J. Lancrenon. *How to Securely Prolong the Computational Bindingness of Pedersen Commitments*. Cryptology ePrint Archive, Report 2015/584. https://eprint.iacr.org/2015/584. 2015 (cit. on p. 23).

[22] A. Fiat, A. Shamir. “How to Prove Yourself: Practical Solutions to Identification and Signature Problems”. In: *Advances in Cryptology — CRYPTO’ 86*. Ed. by A. M. Odlyzko. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987, pp. 186–194. ISBN: 978-3-540-47721-1 (cit. on p. 32).

[23] P.-A. Fouque, G. Poupard, J. Stern. “Sharing Decryption in the Context of Voting or Lotteries”. In: *Financial Cryptography*. Ed. by Y. Frankel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 90–104. ISBN: 978-3-540-45472-4 (cit. on pp. 33, 35).

[24] T. K. Frederiksen, Y. Lindell, V. Osheter, B. Pinkas. “Fast Distributed Rsa Key Generation For semi-honest and Malicious Adversaries”. In: *Advances in Cryptology — CRYPTO 2018*. Ed. by H. Shacham, A. Boldyreva. Cham: Springer International Publishing, 2018, pp. 331–361. ISBN: 978-3-319-96881-0 (cit. on p. 34).

[25] R. Gennaro, S. Jarecki, H. Krawczyk, T. Rabin. “Secure Distributed Key Generation for Discrete-Log Based Cryptosystems”. In: *Journal of Cryptology* 20.1 (2007), pp. 51–83 (cit. on p. 41).

[26] R. L. Graham, D. E. Knuth, O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. 2nd. USA: Addison-Wesley Longman Publishing Co., Inc., 1994. ISBN: 0201558025 (cit. on p. 83).

[27] F. Hao. *Schnorr Non-Interactive Zero-Knowledge Proof*. RFC 8235. Sept. 2017. DOI: 10.17487/RFC8235 (cit. on p. 26).

[28] C. Hazay, Y. Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions*. Springer Science & Business Media, 2010 (cit. on p. 32).

[29] C. Hazay, G. L. Mikkelsen, T. Rabin, T. Toft, A. A. Nicolosi. "Efficient RSA Key Generation and Threshold Paillier in the Two-party Setting". In: *Journal of Cryptology* 32.2 (Apr. 2019), pp. 265–323. ISSN: 1432-1378. DOI: 10.1007/s00145-017-9275-7 (cit. on pp. 20, 33, 34, 39, 50).

[30] D. W. Jones. "Early Requirements for Mechanical Voting Systems". In: *2009 First International Workshop on Requirements Engineering for e-Voting Systems*. IEEE. 2009, pp. 1–8 (cit. on p. 11).

[31] J. Katz, Y. Lindell. *Introduction to Modern Cryptography*. 2. ed. Chapman & Hall/CRC cryptography and network security. CRC Press, 2015. ISBN: 978-1-4665-7026-9 (cit. on p. 16).

[32] L. M. Kohnfelder. *Towards a Practical Public-Key Cryptosystem.* 1978 (cit. on p. 15).

[33] R. Küsters, J. Liedtke, J. Müller, D. Rausch, A. Vogt. *Ordinos: A Verifiable Tally-Hiding Remote E-Voting System*. Tech. rep. 2019. URL: https://publ.sec.uni-stuttgart.de/kuestersliedtkemuellerrauschvogt-ordinos-tr-2019.pdf (cit. on pp. 11, 12, 20, 63–65, 72).

[34] H. Lipmaa, T. Toft. "Secure Equality and Greater-Than Tests with Sublinear Online Complexity". In: *Automata, Languages, and Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 645–656. ISBN: 978-3-642-39212-2 (cit. on pp. 63–65, 68).

[35] M. Malkin, T. D. Wu, D. Boneh. "Experimenting with Shared Generation of RSA Keys". In: *NDSS*. 1999 (cit. on p. 34).

[36] S. M. Meyer, J. P. Sorenson. "Efficient Algorithms for Computing the Jacobi Symbol". In: *International Algorithmic Number Theory Symposium*. Springer. 1996, pp. 225–239 (cit. on p. 38).

[37] E. Morais, T. Koens, C. van Wijk, A. Koren. "A Survey on Zero Knowledge Range Proofs and Applications". In: *CoRR* abs/1907.06381 (2019). arXiv: 1907.06381 (cit. on p. 30).

[38] H. Nguyen. "RSA Threshold Cryptography". In: *Department of Computer Science, University of Bristol* (2005) (cit. on p. 34).

[39] A. A. Nicolosi. *Efficient RSA Key Generation Protocol in a Two-party Setting and Its Application into the Secure Multiparty Computation Environment*. 2011 (cit. on pp. 33, 34).

[40] T. Nishide, K. Ohta. "Multiparty Computation for Interval, Equality, and Comparison without Bit-Decomposition Protocol". In: *Public Key Cryptography – PKC 2007*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 343–360. ISBN: 978-3-540-71677-8 (cit. on p. 71).

[41] T. Nishide, K. Sakurai. "Distributed Paillier Cryptosystem without Trusted Dealer". In: *International Workshop on Information Security Applications*. Springer. 2010, pp. 44–60 (cit. on pp. 3, 22, 24, 29–31, 33–40, 43, 44, 47, 60).

[42] T. Okamoto. “An Efficient Divisible Electronic Cash Scheme”. In: *Advances in Cryptology — CRYPT0’ 95*. Ed. by D. Coppersmith. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 438–451. ISBN: 978-3-540-44750-4 (cit. on p. 27).

[43] P. Paillier. “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In: *Advances in Cryptology — EUROCRYPT ’99*. Ed. by J. Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238. ISBN: 978-3-540-48910-8 (cit. on pp. 16, 18, 33).

[44] T. P. Pedersen. “Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing”. In: *Annual International Cryptology Conference*. Springer. 1991, pp. 129–140 (cit. on pp. 23, 24, 40).

[45] T. Rabin. “A Simplified Approach to Threshold and Proactive RSA”. In: *Advances in Cryptology — CRYPTO ’98*. Ed. by H. Krawczyk. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 89–104. ISBN: 978-3-540-68462-6 (cit. on pp. 22, 24).

[46] R. L. Rivest, A. Shamir, L. Adleman. “A Method for Obtaining Digital Signatures and Public-key Cryptosystems”. In: *Communications of the ACM* 21.2 (Feb. 1978), pp. 120–126. ISSN: 0001-0782. DOI: 10.1145/359340.359342 (cit. on p. 15).

[47] C. P. Schnorr. “Efficient Identification and Signatures for Smart Cards”. In: *Advances in Cryptology — CRYPTO ’89 Proceedings*. New York, NY: Springer New York, 1990, pp. 239–252. ISBN: 978-0-387-34805-6 (cit. on p. 26).

[48] B. Schoenmakers, P. Tuyls. “Efficient Binary Conversion for Paillier Encrypted Values”. In: *Advances in Cryptology — EUROCRYPT 2006*. Ed. by S. Vaudenay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 522–537. ISBN: 978-3-540-34547-3 (cit. on p. 69).

[49] A. Shamir. “How to Share a Secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613 (cit. on pp. 20, 21).

[50] T. Toft. “Sub-Linear, Secure Comparison with Two Non-Colluding Parties”. In: *Public Key Cryptography – PKC 2011*. Ed. by D. Catalano, N. Fazio, R. Gennaro, A. Nicolosi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 174–191. ISBN: 978-3-642-19379-8 (cit. on p. 65).

[51] T. Van Le, K. Q. Nguyen, V. Varadharajan. “How to Prove That a Committed Number Is Prime”. In: *Advances in Cryptology — ASIACRYPT ’99*. Ed. by K.-Y. Lam, E. Okamoto, C. Xing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 208–218. ISBN: 978-3-540-48000-6 (cit. on p. 27).

[52] T. Veugen, T. Attema, G. Spini. *An Implementation of the Paillier Crypto System with Threshold Decryption without a Trusted Dealer*. Cryptology ePrint Archive, Report 2019/1136. https://eprint.iacr.org/2019/1136. 2019 (cit. on pp. 33–36, 38–40, 47, 53, 60).

[53] K. Wang, E. Rescorla, H. Shacham, S. Belongie. *Openscan: A Fully Transparent Optical Scan Voting System*. Aug. 2010 (cit. on p. 11).

All links were last followed on June 19, 2020.

A ZK-Proof of Equality of Commitments

memo

Takashi Nishide

February 24, 2020

1 memo

- In the paper, I assumed that the largest P' is used throughout the protocol. But the following strategy may be used to be more efficient.
- What you want to do is (in my understanding):
  - We assume $P' \ll P''$.
  - There are 2 commitmets (where the secret x satisfies $x < P'$)

    $$C_1 = g_1^x h_1^{x'}, \;\; C_2 = g_2^x h_2^{x''} \;\; \text{where the order of } g_1, h_1 \text{ is } P', \text{ and the order of } g_2, h_2 \text{ is } P''$$

    and the ZKP (zero-knowledge proof) that C_1 and C_2 have the same x is necessary.

    - I think in your situation, first you have C_1, and later generates C_2 with the corresponding ZKP.
- With the following method, I think you can do what you need to do.

$$\begin{array}{ccc}
\boxed{\text{Prover}} & & \boxed{\text{Verifier}} \\
\text{choose random } r_1'' \in \mathbb{Z}_{P'},\ r_2'' \in \mathbb{Z}_{P''} & \xrightarrow{g_1^{r_1'} h_1^{r_1''},\ g_2^{r_1'} h_2^{r_2''}} & \\
r_1' \in \mathbb{Z}_{2^\ell \cdot 2^t \cdot P'} & & \\
 & \xleftarrow{c} & \text{choose random } c \in \mathbb{Z}_{2^t} \\
z_1' \leftarrow r_1' + c \cdot x \text{ (over the integers)} & & \\
z_1'' \leftarrow r_1'' + c \cdot x' \bmod P' & & \\
z_2'' \leftarrow r_2'' + c \cdot x'' \bmod P'' & \xrightarrow{z_1',\ z_1'',\ z_2''} & g_1^{z_1'} h_1^{z_1''} \overset{?}{=} g_1^{r_1'} h_1^{r_1''} (C_1)^c, \\
 & & g_2^{z_1'} h_2^{z_2''} \overset{?}{=} g_2^{r_1'} h_2^{r_2''} (C_2)^c \\
 & & 0 \overset{?}{<} z_1' \overset{?}{<} 2^\ell \cdot 2^t \cdot P'
\end{array}$$

Remark.

- The above ZKP will guarantee that x in C_2 will be like $\{x \text{ in } C_1 + \text{multiple of } P'\}$, so C_2 may be of the form $g_2^{x+P'} h_2^{x''}$ rather than $g_2^x h_2^{x''}$.
  - To avoid this, Prover also needs to give the range ZKP which proves that x in $C_2 = g_2^x h_2^{x''}$ satisfies $0 \le x < P'$ (or $0 \le x < 2^{\ell'}$ may suffice where ℓ' is the bit length of P' or max possible x).
  - There are several ways to give the range proof. One simple (but not so efficient) way can be found in Section 2.
- ℓ is called statistical security parameter, and typically $40 \le \ell \le 80$ in practice.
  - z_1' will not leak the information about x (i.e., statistically hide x) because the space $\mathbb{Z}_{2^\ell \cdot 2^t \cdot P'}$ for r_1' is much larger than the max possible value $2^t \cdot P'$ of cx.
    - Let the bit length of P' be ℓ', and we will also be able to use $\mathbb{Z}_{2^\ell \cdot 2^t \cdot 2^{\ell'}}$ instead of $\mathbb{Z}_{2^\ell \cdot 2^t \cdot P'}$

  - The condition $0 \overset{?}{<} z'_1 \overset{?}{<} 2^\ell \cdot 2^t \cdot P'$ will be satisfied with overwhelming probability because cx is much smaller than r'_1 with overwhelming probability
    - In other words, the probability distribution of z'_1 is statistically indistinguishable from that of r'_1.
- t should be chosen such that $\frac{1}{2^t}$ is negligibly small (i.e., it should be difficult to guess random c in advance), and typically $40 \le t \le 80$ in practice.

2 Simple Range Proof

- Suppose there is a commitment $C = g^x h^r$ and we want to prove that $0 \le x < 2^t$ without revealing x in zero-knowledge.
- For simplicity, I explain with a concrete example by assuming that $x = 4$ and $t = 3$.
  1. First obtain the binary representation of x, and here $4 = (b_2, b_1, b_0) = (1, 0, 0)_2$.
  2. For each $b_i \in \{0, 1\}$, compute commitments $C_i = g^{b_i} h^{r_i}$.
  3. For each C_i, give the ZKP that $b_i = 0$ or $b_i = 1$ (see Section 3).
  4. Compute
  $$C' \leftarrow \frac{C}{C_2^{2^2} \cdot C_1^{2^1} \cdot C_0^{2^0}} = \frac{g^x h^r}{(g^{b_2} h^{r_2})^{2^2} \cdot (g^{b_1} h^{r_1})^{2^1} \cdot (g^{b_0} h^{r_0})^{2^0}} = \frac{h^r}{(h^{r_2})^{2^2} \cdot (h^{r_1})^{2^1} \cdot (h^{r_0})^{2^0}}$$
  and Prover proves in zero-knowledge that it knows $r' = r - 2^2 r_2 - 2^1 r_1 - 2^0 r_0$ where $C' = h^{r'}$ with base h.

3 OR Proof

Prover proves that b_i in $C_i = g^{b_i} h^{r_i}$ is 0 or 1 in zero-knowledge.

- The basic idea: if b_i is 0 or 1, Prover knows the exponent of one of $C_{\text{1st}} = C_i$ and $C_{\text{2nd}} = \frac{C_i}{g}$ with base h.
- Suppose now $b_i = 1$ (i.e., $C_{\text{1st}} = g h^{r_i}$ and $C_{\text{2nd}} = h^{r_i}$) and the ZKP is as follows.

$$\begin{array}{ccc}
\boxed{\text{Prover}} & & \boxed{\text{Verifier}} \\
\text{choose random } z_{\text{1st}}, c_{\text{1st}}, r_2 \in \mathbb{Z}_{P'} & \xrightarrow{\frac{h^{z_{\text{1st}}}}{(C_{\text{1st}})^{c_{\text{1st}}}},\ h^{r_2}} & \\
 & \xleftarrow{c} & \text{choose random } c \in \mathbb{Z}_{P'} \\
\text{Prover sets } c_{\text{2nd}} \text{ such that } c = c_{\text{1st}} + c_{\text{2nd}} \bmod P' & & \\
z_{\text{2nd}} \leftarrow r_2 + c_{\text{2nd}} \cdot r_i \bmod P' & \xrightarrow{z_{\text{1st}},\, c_{\text{1st}},\, z_{\text{2nd}},\, c_{\text{2nd}}} & h^{z_{\text{1st}}} \overset{?}{=} \frac{h^{z_{\text{1st}}}}{(C_{\text{1st}})^{c_{\text{1st}}}} (C_{\text{1st}})^{c_{\text{1st}}}, \\
 & & h^{z_{\text{2nd}}} \overset{?}{=} h^{r_2} (C_{\text{2nd}})^{c_{\text{2nd}}} \\
 & & c \overset{?}{=} c_{\text{1st}} + c_{\text{2nd}} \bmod P'
\end{array}$$

- For $C_{\text{2nd}} = h^{r_i}$, Prover can give the correct ZKP, but for $C_{\text{1st}} = g h^{r_i}$, Prover cannot because C_{1st} is not of the form h°.
  - So Prover does cheating by choosing z_{1st}, c_{1st} in advance, and for any c_{2nd} which is determined later, Prover can give the correct ZKP by using r_i.
  - This is the famous technique called sometimes OR proof.

References

2

B Computing the Coefficients of $P_l(x)$

The coefficients α_i of $P_l(x) = \sum_{i=0}^{l} \alpha_i \cdot x^i$, that maps 1 to 1 and $x \in \{2, ..., l+1\}$ to 0 can be computed in various ways using general interpolation techniques. Here we present one based on a recurrence relation similar to Stirling numbers of the first kind. $P_l(x)$ is clearly characterized by its l zeros. More precisely, it is a multiple such that 1 maps to 1 of $P_l'(x) = (x-2)\cdot(x-3)\cdots(x-l-1)$ with l factors. The factor c for $c \cdot P_l'(x) = P_l(x)$ is the inverse of $P_l'(1) = (-1)\cdot(-2)\cdots(-l) = (-1)^l \cdot (l)!$ (inverse modulo N when working in the finite field $\mathbb{Z}_N$). It remains to determine the coefficients of $P_l'(x)$. $P_l'(x)$ is part of the falling factorial, which is defined as $x^{\underline{n}} = x\cdot(x-1)\cdot(x-2)\cdots(x-n+1)$ with n factors. Obviously, $P_l'(x) = \frac{x^{\underline{l+2}}}{x^{\underline{2}}}$. Signed Stirling numbers of the first kind (denoted as $(-1)^{n-k} \cdot \begin{bmatrix} n \\ k \end{bmatrix}$) are the coefficients of the expansion of the falling factorial: $x^{\underline{n}} = \sum_{k=0}^{n}(-1)^{n-k} \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^k$. I.e., $\begin{bmatrix} n \\ k \end{bmatrix}$ is the absolute value of the k^{th} coefficient of the expansion of $x^{\underline{n}}$. The Stirling numbers of the first kind (unsigned) are known to be computed by the following recurrence relation [see, e.g., 26]:

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = n\begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix}$$

with the initial conditions $\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1$ ($x^{\underline{0}}$ is the empty product) and $\begin{bmatrix} n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ n \end{bmatrix} = 0$ for $n > 0$ (the 0^{th} coefficient is always 0 and all other coefficients of $x^{\underline{0}}$ are 0).

The absolute values of the coefficients of $P_l'(x)$ can be computed analogously. From here on $\begin{bmatrix} n \\ k \end{bmatrix}$ denotes our modified Stirling numbers that are absolute values of the coefficients of $P_n'(x)$. We need $\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 1$, $\begin{bmatrix} 0 \\ n \end{bmatrix} = 0$ for $n > 0$ as well, since $P_0'(x)$ is the empty product, thus the 0^{th} coefficient is 1 while all others are 0. Clearly, $\begin{bmatrix} n \\ -1 \end{bmatrix} = 0$, because the -1^{th} coefficient of the polynomial will always be 0. This shifts the coefficients by one position, such that x disappears as a factor. In the recurrence, we need to replace the factor n by $n+2$:

$$\begin{bmatrix} n+1 \\ k \end{bmatrix} = (n+2)\begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix}$$

which, figuratively, sets the "start" of the recurrence to the factor $(x-2)$ of the falling factorial. With the modified recurrence relation Table B.1 is generated. Note that the coefficients still need to be signed correctly.

The proof for $P_n'(x) = \sum_{k=0}^{n}(-1)^{n-k} \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^k$ by induction works as follows: Obviously, $P_0'(x) = 1 \cdot x^0$ is correct. Given $P_n'(x) = \sum_{k=0}^{n}(-1)^{n-k} \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^k$, we have

$$P_{n+1}'(x) = \sum_{k=0}^{n+1}(-1)^{n+1-k} \cdot \begin{bmatrix} n+1 \\ k \end{bmatrix} \cdot x^k$$

By plugging in the recurrence we get:

$$= \sum_{k=0}^{n+1}(-1)\cdot(-1)^{n-k} \cdot \left((n+2)\cdot\begin{bmatrix} n \\ k \end{bmatrix} + \begin{bmatrix} n \\ k-1 \end{bmatrix}\right) \cdot x^k$$

n \ k	-1	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0	0
1	0	2	1	0	0	0	0	0
2	0	6	5	1	0	0	0	0
3	0	24	26	9	1	0	0	0
4	0	120	154	71	14	1	0	0
5	0	720	1044	580	155	20	1	0
6	0	5040	8028	5104	1665	295	27	1

Table B.1: Recurrence table for the (unsigned) coefficients of $P'_n(x)$ with $n < 7$

By expansion and taking out the factor -1:

$$= (-1) \cdot \sum_{k=0}^{n+1} (-1)^{n-k} \cdot (n+2) \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^k + (-1)^{n-k} \cdot \begin{bmatrix} n \\ k-1 \end{bmatrix} \cdot x^k$$

Splitting up the sums and taking out the factor $n + 2$ from the first one:

$$= (-1) \cdot \left((n+2) \cdot \sum_{k=0}^{n+1} (-1)^{n-k} \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^k + \sum_{k=0}^{n+1} (-1)^{n-k} \cdot \begin{bmatrix} n \\ k-1 \end{bmatrix} \cdot x^k \right)$$

Writing the last summand of the first sum separately we can plug in P'_n:

$$= (-1) \cdot \left((n+2) \cdot P'_n(x) + (-1)^{n-n+1} \cdot \begin{bmatrix} n \\ n+1 \end{bmatrix} \cdot x^{n+1} + \sum_{k=0}^{n+1} (-1)^{n-k} \cdot \begin{bmatrix} n \\ k-1 \end{bmatrix} \cdot x^k \right)$$

Since $\left[\begin{smallmatrix} n \\ n+1 \end{smallmatrix}\right] = 0$ (the $(n+1)^{\text{th}}$ coefficient of P'_n):

$$= (-1) \cdot \left((n+2) \cdot P'_n(x) + \sum_{k=0}^{n+1} (-1)^{n-k} \cdot \begin{bmatrix} n \\ k-1 \end{bmatrix} \cdot x^k \right)$$

Writing the first summand of the remaining sum separately (which turns 0 due to the initial conditions) and then shifting indices, we receive the desired result:

$$\begin{aligned}
&= (-1) \cdot \left((n+2) \cdot P'_n(x) + 0 + \sum_{k=1}^{n+1} (-1)^{n-k} \cdot \begin{bmatrix} n \\ k-1 \end{bmatrix} \cdot x^k \right) \\
&= -(n+2) \cdot P'_n(x) - \sum_{k=0}^{n} (-1)^{n-k+1} \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^{k+1} \\
&= -(n+2) \cdot P'_n(x) - (-x) \sum_{k=0}^{n} (-1)^{n-k} \cdot \begin{bmatrix} n \\ k \end{bmatrix} \cdot x^k \\
&= -(n+2) \cdot P'_n(x) + xP'_n(x)) \\
&= (-(n+2) + x) \cdot P'_n(x) \\
&= (x - n - 2) \cdot P'_n(x) \\
&= (x - (n+1) - 1) \cdot P'_n(x)
\end{aligned}$$

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature