Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Klasszikus kommunikációs lépések megvalósítása CV-QKD rendszerben

TDK dolgozat

Készítette:

Márton Botond László

Konzulens:

dr. Bacsárdi László
dr. Kis Zsolt

2022

Tartalomjegyzék

Kivonat

A mai világban a biztonságos kommunikáció és adatközlés az egyik legfontosabb feladat, amit meg kell oldani. A legelterjedtebb megoldás erre a célra a publikus kulcsú kriptográfia, aminek a segítségével például a titkosított internetes kommunikáció jelentőse része zajlik. Ezen megoldások biztonságosságát nem bizonyították, hanem arra a feltevésre alapulnak, hogy a feltörésükhöz szükséges számítások hatékonyan nem oldhatóak meg egy klasszikus számítógéppel (pl. az RSA esetében nagy számok prímhatványokra való bontása). Az elmúlt évtizedekben megjelentek olyan kvantumos algoritmusok, melyek segítségével ezek a feladatok hatékonyan megoldhatóak egy kvantumszámítógépen (pl. a Shor-algoritmus). Bár ezen eszközök használta még távol van, de már most érdemes olyan megoldások után kutatni, melyek használhatóak akkor is, ha a támadó birtokában van egy kvantumszámítógép.

Egy ilyen megoldás a történelemben már régen is ismert szimmetrikus kulcsú kriptográfia, melynek egyik fajtája, a One-Time Pad séma bizonyítottan biztonságos tud lenni, ha a megfelelő minőségű és hosszúságú kulccsal rendelkezik a két fél. Ennek a használatához viszont szükséges, hogy a két oldal megkapja a kommunikáció előtt a közös, titkos kulcsot, ami eddig nehéz feladatnak bizonyult a hatékony és a biztonságos megvalósítás problémája miatt. A kvantumos kulcsszétosztás (quantum key distribution) segítségével a két fél biztonságosan létre tud hozni és megosztani egymással egy közös kulcsot úgy, hogy biztosak lehetnek benne, hogy csak ők ismerik azt a lépések végrehajtása után.

A dolgozatban a Műegyetemen épülő, Magyarországon elsőként létrehozott folytonos változójú kvantumos kulcsszétosztó rendszeren (CV-QKD) végzett munkámat szeretném bemutatni. Feladatom a kvantumos kommunikáció utáni klasszikus csatornán folytatott protokoll implementálása volt, melynek segítségével a két fél képes a kvantumos adatokból létrehozni egy közös titkos kulcsot, amit később fel tudnak használni titkosításra. A kulcs használatára példaként egy szöveges üzenetek és képek titkosított küldésére használható programot is készítettem. A rendszert teszteltük több ízben egy magyar telekommunikációs szolgáltató hálózatán is két különböző távolságban, mely tesztek eredményét is bemutatom.

Abstract

In today's world, secure communication and data transfer is one of the most important tasks. The most widely used solution is public key cryptography, which is used for a significant part of encrypted internet communications. The security of these solutions has not been proven mathematically, but they are based on the assumption that the computations required to crack them cannot be efficiently performed by a classical computer (e.g., in case of RSA, finding the prime factors of a large number). In the last decades, quantum algorithms have emerged that allow these tasks to be solved efficiently on a quantum computer (e.g., Shor's algorithm). Although the use of these machines is still far away, it is already worthwhile to look for solutions that can be used even if the attacker has a quantum computer in his possession.

One such solution is the well-known symmetric key cryptography, one type of which, the One-Time Pad scheme, has been shown to be secure if the two parties have keys of the right quality and length. However, its use requires that the two sides obtain a shared secret key before communication, which has so far proved difficult due to problems of efficient and secure implementation. With quantum key distribution, the two parties can securely create and share a common key with eachother and they can be sure that only they know this key, after performing the required steps.

In this paper, I will present my work on the first ever Hungarian Continuous Variable Quantum Key Distribution (CV-QKD) system built at the Budapest University of Technology and Economics. My task was to implement the protocol on the classical channel after the quantum communications has taken place, which allows both parties to generate a shared secret key from the quantum data, which can be used for encryption later. I also developed a program that can be used to send encrypted text messages and images. The system was tested several times on the network of a Hungarian telecom operator at two different distances, and the results of these tests are also presented.

1. fejezet

Bevezetés

Nap mint nap kommunikálunk titkosítva az interneten keresztül még ha ezt nem is vesszük észre. Bármikor, amikor egy `https`-sel kezdődő oldalra látogatunk publikus kulcsú titkosítást használunk: például ellenőrizzük, hogy arra az oldalra navigáltunk-e, amelyikre szerettünk volna, vagy elérjük, hogy a forgalom a webszerver és a böngészőnk között ne legyen olvasható egy harmadik fél számára. Elvárjuk, hogy ezek a megoldások könnyen és a felhasználó számára láthatatlanul működjenek, de azt is, hogy ne lehessen őket feltörni. Ennek ellenére a publikus kulcsú megoldásoknál nincs kimondott bizonyítás arra, hogy a titkosítás feltöréséhez szükséges műveletek nem oldhatóak meg hatékonyan egy klasszikus számítógépen, csak feltételezések, mint például az egyik legelterjedtebb ilyen titkosítás az RSA [1] esetében, ahol a biztonság alapját adó probléma az, hogy a nagy számokat nehéz prímtényezőkre bontani.

A kvantuminformatika megjelenésével viszont megalkottak olyan algoritmusokat, melyek képesek megoldani ezeket a nehéz problémákat és mindezt hatékonyan viszik végbe. Az említett RSA probléma esetében ez a Peter Shor által elkészített algoritmus [2]. Olyan kvantumszámítógépek megjelenése, melyek képesek a ma használt hosszúságú RSA kulcsokat feltörni még a közeljövőben sem várható, de érdemes felkészülni olyan megoldásokkal melyek egy kvantumszámítógép jelenlétében is biztonságosan használhatóak. Ilyen megoldások már léteznek: az egyik ilyen a szimmetrikus kulcs titkosítások közé tartozó One-Time Pad, amiről Shannon belátta, hogy biztonságos abban az esetben, ha a kulcs véletlenszerűen választott és elég hosszú. Ahhoz, hogy ezek a feltételek teljesüljenek egy olyan rendszert kell használni, mellyel a két fél képes gyorsan nagy méretű és véletlen kulcsokat megosztani egymással, méghozzá úgy, hogy azt egy harmadik fél nem tudja megszerezni. Ezt valósítja meg a kvantumos kulcsszétosztás (QKD - Quantum Key Distribution).

A QKD segítségével a kvantummechanika törvényeit és egy klasszikus autentikált csatornát kihasználva a két kommunikáló fél képes létrehozni egy közös titkos

kulcsot úgy, hogy detektálni tudják egy harmadik, rosszindulatú lehallgató jelenlétét. A QKD protokollok első verziói egyesével használtak fel fotonokat az információ átvitelére. Ezeket diszkrét megoldásoknak nevezzük és az első ilyen protokoll a Bennet és Brassard által megalkotott BB84 volt [3]. Voltak olyan megoldások, melyek kihasználták a kvantumfizikában egyedüli összefonódást is pl. az E91 protokoll [4]. A diszkrét megoldások elterjedése után az úgynevezett folytonos változójú protokollok is megjelentek [5] [6]. Ezek a megoldások a diszkrét társaikkal szemben nem egy fotonba kódolják a diszkrét információt (pl. a BB84 estében a polarizáció), hanem fotoncsomagokba (pl. azok fázisába vagy amplitúdójába) és olyan detektorokat tudnak használni, melyek nem igényelnek pl. extra hideg környezetet vagy minél jobb fénytől való elszigetelést. Ezen tulajdonságok miatt a folytonos változójú QKD megoldások (CV-QKD - *Continous Variable Quantum Key Distribution*) azzal a lehetőséggel kecsegtetnek, hogy könnyebben be lehet őket építeni meglévő optikai rendszerekben így segítve az elterjedésüket. A folytonos változójú rendszereknek is sok fajtája létezik, vannak melyek összenyomott állapotokat használnak [7], de vannak olyanok is, amik koherens állapotokkal [8] dolgoznak. A CV-QKD megoldások gyors fejlődése során egyre nagyobb távokat tudtak áthidalni az elkészült eszközökkel elérve a 80-100 km-es távolságot is [9] [10] a közelmúltban.

A Budapesti Műszaki Egyetemen is foglalkoznak kvantumos kulcsszétosztással, mely tevékenység során rendszer épült pl. a BB84 protokoll megvalósítására is. A diszkrét protokollok mellett a folytonos változójú megoldásokkal kapcsolatban is történik kutatás, melynek keretében elkészült Magyarország első CV-QKD rendszere, amelynek az elkészítésében én is részt vehettem. Feladatom az optikai szálon folyó kvantumos kommunikáció lezajlása után történő klasszikus üzenetváltás elkészítése volt, melynek végén a kvantumos mérési eredményekből egy titkos és mindkét félnél megegyező kulcs jön létre, amit fel tudnak használni titkosításra. Ahhoz, hogy egy ilyen felhasználási megoldást bemutassunk, készítettem egy chat-alkalmazást, melynek segítségével titkosítottan tud a két fél üzeneteket és képeket váltani egymással felhasználva a rendszer által létrehozott kulcsot. A CV-QKD eszközöket több alkalommal is teszteltük egy magyar telekommunikációs vállalat optikai rendszerén és mindkét esetben sikerült kulcsot létrehozni.

1.1. A dolgozat felépítése

A dolgozat első részében szeretném bemutatni a különböző QKD megoldásokat. A 2. fejezet foglalkozik a diszkrét megoldásokkal, ahol a BB84 protokoll kerül bemutatása. Ezután a 3. fejezetben először a folytonos változójú protokollokhoz szükséges

információt vezetem be, majd kitérek a koherens és az összenyomott állapotokra a 3.1.1. és 3.1.2. fejezetekben.

Ezt követően a Budapesti Műszaki Egyetemen épülő rendszer felépítést és az optikai kommunikáció működését mutatom be a 3.2.1. fejezetben, majd a klasszikus csatornán történő lépéseket ismertetem a 3.2.2. fejezetben, kitérve arra, hogyan lehet megbecsülni egy rendszer által elérhető kulcsgenerálási sebességet (3.2.2.1. fejezet).

A dolgozat második részében a saját munkámat mutatom be. Először a 4.1. fejezetben azokat a megoldásokat veszem végig, melyeket tanulmányoztam a munkám során, de végül elvetettem a használatukat. Ezután pedig áttérek a választott és később implementált megoldás bemutatására, kezdve az általam használt hibajavító kód bevezetésével a 4.2. fejezetben, végül pedig a 4.3. fejezetben a végrehajtandó lépések ismertetésével. Ezt követően a 4.4. fejezet mutatja be az elkészült programot. A dolgozat végén pedig a tesztelések eredményét foglalom össze és értékelem. Ezek a tesztek Magyarországon az első sikeres demonstrációi voltak egy optikai hálózaton történő kvantumos kulcsszétosztásnak.

2. fejezet

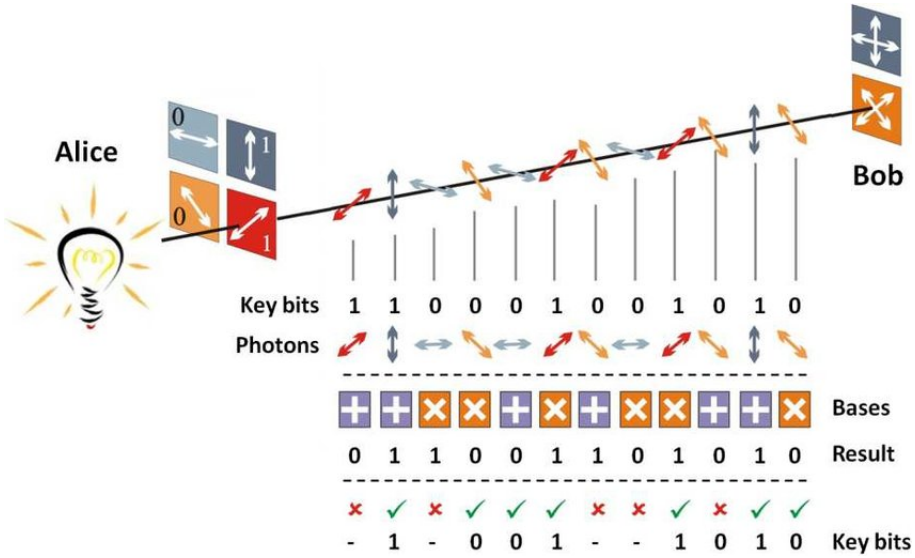
Diszkrét kvantumkommunikáció

A kvantuminformatika mellett új területként természetesen a kvantumos alapon történő kommunikáció is megjelent. Ezen a téren viszont sokkal több, már a való világban is működő megoldásról lehet beszélni, melyeket széles körben használnak a banki szektorban működő cégek vagy egyes kormányzatok. Ennek a területnek alapvetően két ága különböztethető meg. Az egyik a tisztán kvantumos kommunikáció, amikor a két fél csak a kvantumos csatornán oszt meg egymással információt. A másik csoportba azok a megoldások tartoznak, melyek a kvantumos csatorna mellett klasszikus csatornán is kapcsolatban állnak és mindkét csatornát használják a kommunikáció során. Ebben a fejezetben a diszkrét kvantumkommunikációhoz tartozó QKD megoldások egyikét fogom bemutatni.

Fontos elsőként arra rávilágítani, hogy mit is jelent az, hogy diszkrét kvantumos kommunikáció. Diszkrét esetben egy kvantumrendszer véges dimenziójú állapotterének vektoraiba kódoljuk az információt. Ilyen például a foton polarizációs állapota vagy egy egyfotonos hullámcsomag megérkezésének időpontja. Folytonos esetben viszont a fizikai rendszer olyan mérhető tulajdonságába helyezzük azt, mely folytonos értéket vesz fel egy tartományon. Ilyen lehet egy sokfotonos hullámcsomag fázisa, mely a $[0, 2\pi]$ tartományon vesz fel értékeket, vagy amplitúdója (intenzitása) például koherens, vagy az összenyomott állapotú fény esetében. Diszkrét esetben egyfoton detektorokat használunk, melyek nevük alapján arra képesek, hogy egy foton érkezését jelezzék, így kimenetükön, megfelelő egyszerűsítéssel élve, azt olvashatjuk le, hogy érkezett-e foton, vagy pedig nem. A folytonos esettel 3. fejezet foglalkozik.

2.1. BB84 protokoll

A legelső algoritmus a QKD területén a Brennett és Brassard által 1984-ben bemutatott algoritmus volt, mely nevét a készítőiről és a publikáció évszámáról kapta, így lett BB84 [3]. Az algoritmus használata közben a két fél, akik létre szeretné-



2.1. ábra. A BB84 lépések összefoglalása. Forrás: [11]

nek hozni egy közös kulcsot, kvantumos és egy autentikált klasszikus csatornán is képesek egymással kommunikálni. Az algoritmus működése során az egyik fél véletlenszerűen választ lehetséges polarizációs bázisok (horizontális – vertikális illetve jobbra- – balra-diagonális) közül egy sorozatot, melyekhez előzetesen biteket rendelt a két fél. A választott sorozat alapján polarizált állapotú fotonokat küld át a másik félnek. A másik fél ezen fotonok polarizációját nem tudja, így ő is véletlenszerűen fog választani egy bázissorozatot, amelyben megméri őket. A két fél ezután a nyilvános autentikált csatornán megbeszéli, hogy mi volt az a két bázissorozat, amit választottak és ha a két sorozat egy-egy eleme megegyezik, akkor ideális átvitel és detekció esetén biztosak lehetnek benne, hogy a mérés eredményeként kapott bit a fogadó oldalán és a sorsolt bit a küldő oldalán megegyezik. Ezeket a biteket fogják felhasználni a kulcs létrehozására. Mivel véletlenszerűen választ bázist a fogadó, ezért a kialakult bitsorozat kb. fele lesz használható kulcsként. Az algoritmus nagy előnye, hogy ha egy rosszindulatú támadó elkapja az egyik küldött fotont, akkor a No Cloning Theorem értelmében azt nem tudja lemásolni és tovább küldeni, ezért ő is csak azt tudja tenni, amit a fogadó oldali fél. Mivel véletlenszerűen választ ő is, ezért az esetek felében rosszul fog mérni és ezért rossz polarizációjú fotont fog továbbküldeni, így a valódi fogadónál az esetek negyedében eltérő lesz az eredmény. A megnövekedett hibaarány alapján a kommunikáló felek tehát képesek észlelni egy támadót a kvantumos csatornán. A protokoll lépései a 2.1. ábrán láthatóak összefoglalva. Ezt az algoritmust már ma is használjak vezetékes, illetve szabad levegős változatban.

3. fejezet

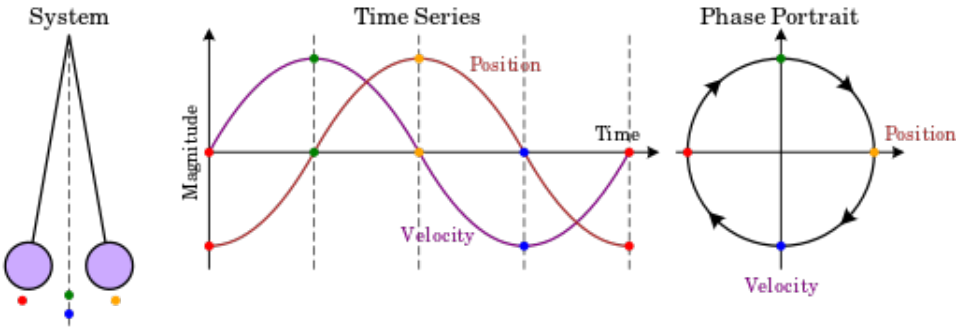
Folytonos változójú kvantumos kulcsszétosztás

Míg a diszkrét esetben az egyfoton detektor szolgáltatja a kimenetet, addig a folytonos esetben a homodin (vagy egyes esetekben heterodin) detektor. Mind a homodin, mind a heterodin detektor a beérkező fény kvadratúráit méri meg, melyet a fázistérben értelmezünk. A kvadratúra operátorokra úgy tekinthetünk, mint a fázistér x és p tengelyével párhuzamos operátorokra, melyek folytonos értéket vesznek fel a fény kvantumállapotát leíró állapottérben.

3.1. A folytonos változójú kvantumos kommunikáció alapjai

Mielőtt bemutatnám a megvalósítás alatt álló protokollt, fontos, hogy megismerkedjünk egy új jelölésrendszerrel, melyet a kvantumoptikában sokat használnak és segítségével jobban lehet bemutatni a fény állapotát. Ez a jelölésrendszer az optikai fázistér. Ez a megközelítés a klasszikus mechanikában használt fázisteret gondolta tovább. Ha egy inga pozíciójának (x) és a momentumának (p) segítségével létrehozunk egy koordináta-rendszert, akkor képesek vagyunk az inga összes lehetséges állapotát felírni ebben a rendszerben. Ezt az analógiát mutatja a 3.1. ábra. A kép bal oldalán látható az inga és annak négy kitüntetett állapota színes pontokkal van jelölve, míg a jobb oldalon ezen négy állapot és a többi lehetséges állapot a fázistérben, miután az inga elkezdett mozogni.

Az optikai fázistér is hasonló megközelítéssel készíthető el. A fény egy elektromágneses hullám, melyet az egymásra merőleges elektromos illetve mágneses térerősségvektorok segítségével lehet jellemezni. A klasszikus fényhullám minden térbeli pontban egy vektor, mely a fázistérben forog, avagy oszcillál, mint egy harmonikus



3.1. ábra. Az inga felírás fázistérben. Forrás: Wikipedia

oszcillátor. Ezt ki lehet terjeszteni kvantumos fázistérre oly módon, hogy a klasszikus X és Y változók helyett kvantum operátorokat vezetünk be, melyeket kvadratúra operátoroknak is neveznek. Az optikai fázistérben ezen két kvadratúra segítségével tudunk megadni minden lehetséges állapotot. A két operátort pedig a következőképpen tudjuk kifejezni

$$\hat{q} = \frac{\hat{a} + \hat{a}^\dagger}{2} \text{ és } \hat{p} = \frac{\hat{a} - \hat{a}^\dagger}{2i},$$

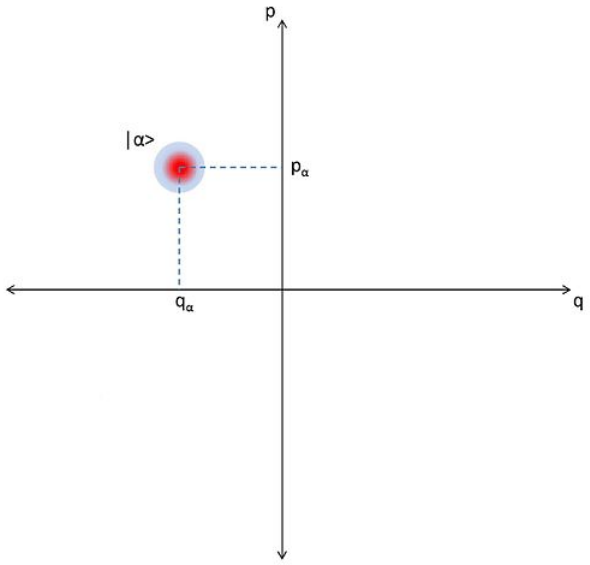
a harmonikus oszcillátor $\hat{a}$ eltüntető és $\hat{a}^\dagger$ keltő operátorok (angolul *annihilation* és *creation*) segítségével. Az eltüntető és keltő operátorokkal lehet a fotonszám állapotban lévő fotonok számát csökkenteni vagy növelni eggyel. Ezen kijelentéshez fontos, hogy megismerjük a fotonszám-állapotokat, vagy más néven számállapotokat, melyeket $|n\rangle$-nel jelölünk, ahol $n \in N$ és nevüket onnan kapták, hogy a szabad elektromágneses mező energia operátorának sajátállapotai, melyek a fotonok. (például a $|n = 0\rangle$ a vákuum állapot). Ezen állapotok egymásra ortogonálisak és teljes rendszert adnak a Hilbert-térben, azaz bármely állapot kifejezhető a szuperpozíciójukként. Az, hogy az eltüntető és keltő operátorok a fotonok számát csökkenteni és növelni tudják az jelenti, hogy segítségükkel ezen számállapotok között tudunk lépni a következő szabályok szerint:

$$\hat{a}^\dagger |n\rangle = \sqrt{n+1}\,|n+1\rangle \text{ és } \hat{a}\,|n\rangle = \sqrt{n}\,|n\rangle$$

Ezen operátorokra ezen kívül még igaz, hogy $[\hat{a}, \hat{a}^\dagger] = 1$, ahol a $[\hat{A}, \hat{B}] = \hat{A}\hat{B} - \hat{B}\hat{A}$ a kommutátor. Az eltüntető és keltő operátorokra az imént felsorolt tulajdonságokból következik, hogy

$$[\hat{q}, \hat{p}] = \frac{i}{2},$$

azaz a két operátor egymással nem felcserélhető. Ebből következik, hogy egyidejűleg pontosan nem mérhetőek, ezért a mért értékük szórása kielégíti a Heisenberg-féle



3.2. ábra. Egy állapot jelölése az optikai fázistérben Forrás: Wikipedia

határozatlansági relációt tetszőleges ψ állapotban:

$$(\triangle\hat{q})_\psi(\triangle\hat{p})_\psi \geq \frac{1}{4}$$

Egy tetszőleges $|\alpha\rangle$ koherens állapot felrajzolását a 3.2. ábra mutatja. Fontos megjegyezni, hogy itt már nem egy konkrét pontról beszélünk a koordinátarendszerben, hanem egy kör alakú foltról, mivel a $|\alpha\rangle$ koherens állapot nem sajátállapota sem az $\hat{x}$ sem a $\hat{p}$ operátoroknak, ezért a kvadratúra mérés mindig bizonytalan értéket ad.

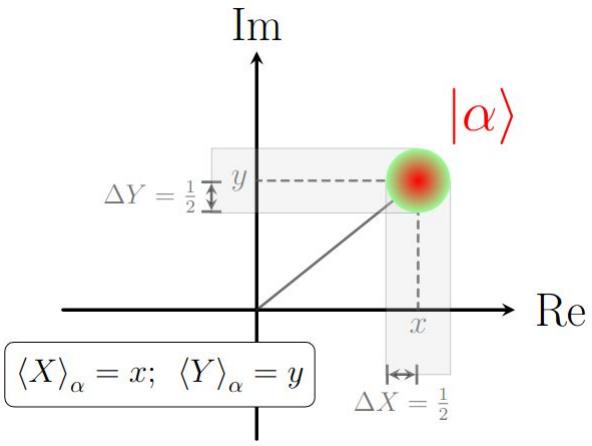
A folytonos változójú kvantumos kulcsszétosztásnak (és magának a folytonos változójú kvantumkommunikációnak) az az alapötlete, hogy az átküldendő információt a két kvadratúrába kódoljuk. A továbbiakban bemutatom két gyakran használt QKD protokoll alapötletét.

3.1.1. Koherens állapot

A koherens (angolul *coherent*) állapotok speciálisak, megadni őket pedig a következő képlettel lehet:

$$|\alpha\rangle = e^{\frac{-|\alpha^2|}{2}} \sum_n \frac{\alpha^n}{\sqrt{n!}} |n\rangle$$

Itt $|n\rangle$ a korábban bevezetett számállapot. Az ilyen $|\alpha\rangle$ állapotok egyik legfontosabb tulajdonsága, hogy ha α-t egy komplex számként képzeljük el $\alpha = x + iy$ alakban,



3.3. ábra. Egy koherens állapot jelölése, mint komplex szám. Megjegyzés: X és Y itt megfelel a két kvadratúrának Forrás: Benedict Mihály: KVANTUMELEKTRODINAMIKA ÉS KVANTUMOPTIKA, SZTE TTIK, 2015.

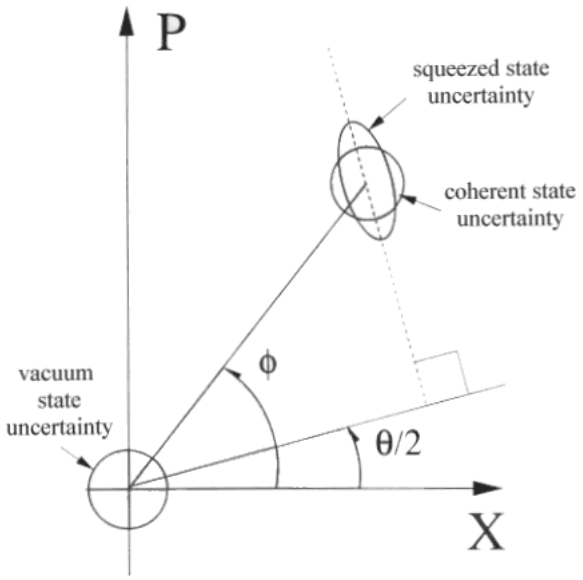
akkor a kvadratúrák várható értékei x és y lesznek, viszont α-tól függetlenül ezen állapotokban a szórásaik $\frac{1}{2}$-ed értéket vesznek fel, mindkettő esetében. Azaz a Heisenberg egyenlőtlenséget ezek az állapotok minimalizálják és egyenlőséggé alakítják. A koherens állapotokat így az optikai fázistérben úgy tudjuk jelölni, hogy a megfelelő pont körül egy $\frac{1}{2}$ sugarú kört húzunk. Egy ilyen állapot felírása a 3.3. ábrán látható.

Korábban megjegyeztem, hogy a $|\alpha\rangle$ koherens állapot nem sajátállapota sem az $\hat{x}$ sem a $\hat{p}$ operátoroknak, emiatt a kvadratúra mérés Gauss-eloszlású. Ezt kihasználva az egyik fél el tud küldeni koherens állapotokat és amikor a vevő megméri az általa választott kvadratúrát, akkor mivel az állapot koherens, tudni lehet a szórás mértékét (a szórás az átvitel során torzulhat).

3.1.2. Összenyomott állapotok

A úgynevezett kvadratúra összenyomott (angolul *squeezed*) állapotok fontos tulajdonsága, hogy a kvadratúrához tartozó szórást módosítjuk. Például az $\hat{x}$ operátor szórása csökken, ugyanakkor a $\hat{p}$ szórása növekszik, miközben a két szórás szorzata továbbra is 1/4. Így létre lehet hozni olyan állapotokat, melyben a koherens állapotokhoz képest a vetületek eloszlásai módosulnak. Egy ilyen példát mutat a 3.4. ábra.

Az összenyomott állapotoknak az az előnye, hogy mivel az egyik kvadratúra szórása nagyon kicsi is lehet, ezért sokkal pontosabban lehet mérni azt. Emiatt, ha



3.4. ábra. Egy összenyomott állapot jelölése. Megjegyzés: X és Y itt megfelel a két kvadratúrának Forrás: Uncertainty Regions for the Vacuum, Coherent, and Squeezed States

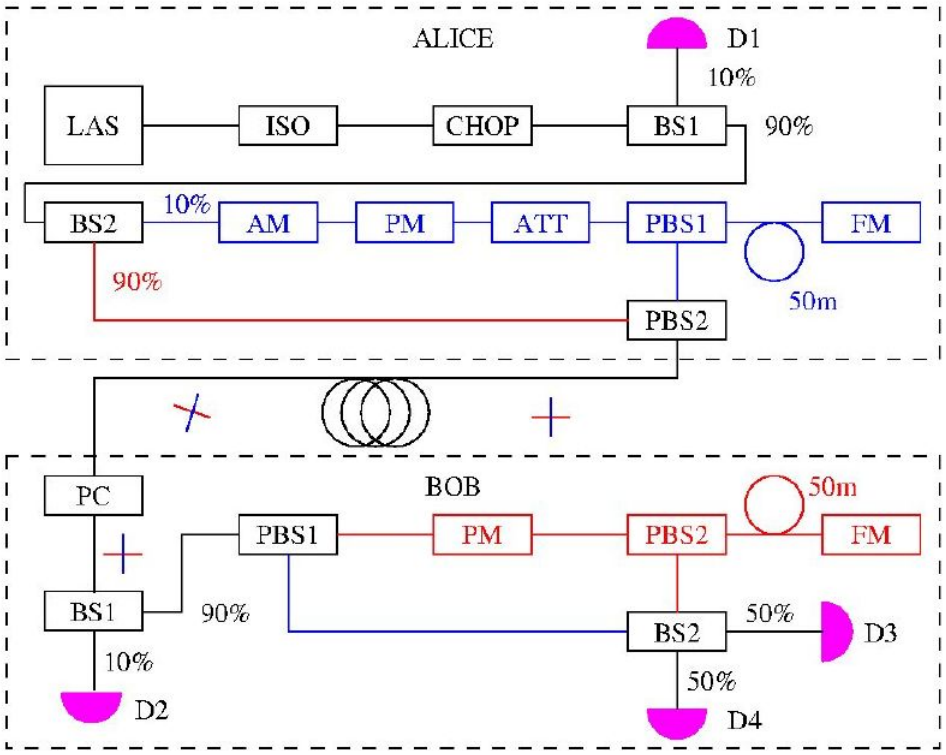
az adott detektort a kvadratúra szórásán kívül más nem befolyásolja, akkor a mérés pontosabb lesz.

3.2. Az épülő rendszer

Ebben a fejezetben röviden bemutatom azt a CV-QKD rendszert, ami pillanatnyilag a laborban épül.

3.2.1. A rendszer felépítése

A rendszer működésének leírása [12] alapján készült, a rendszer felépítése pedig a 3.5. ábrán látható. A protokollban a két kommunikáló fél Alice és Bob. A protokoll úgy kezdődik, hogy Alice előállít egy $z = x + iy$ komplex számot, melynek valós és képzetes részét véletlen módon hozza létre. Ezek alkotják azt az információt, melyet koherens fényimpulzusok kvadratúráiba fog kódolni. A lézerből jövő folytonos fényt egy chopper (CHOP) segítségével impulzusokra osztjuk, melyek két nyalábosztón haladnak át (BS1 és BS2). A BS2 jelű nyalábosztó 10%-90%-os, a bemenetén kapott impulzusokat hasznos jelre és referencia jelre osztja szét. A referencia jel Bob oldalán lesz fontos, ennek a segítségével tudja majd az általa választott kvadratúrát megmérni. A hasznos jel áthalad ezután egy amplitúdó- és fázismodulátoron, amelyek előállítják azt a jelet, melynek kvadratúrái z valós és képzetes részével arányosak. Ezután egy késleltető vonal segítségével időben elkülönítjük a hasznos és a



3.5. ábra. A rendszer felépítése. Forrás: Működési terv [12]

referencia jelet és közös szálon továbbítjuk ezeket Bob-hoz. Bob homidin detektálást végez. Az ő oldalán BS2-vel jelölt nyalábosztó segítségével a bejövő impulzust és a hasznos jelet szuperponálja. Az így megjelenő kimeneti intenzitásokat fogja mérni a D3 és D4 jelű detektorral. A detektorok áramából képzett különbségi áramot egy transzimpedancia erősítővel feszültséggé tudja alakítani, mely arányos lesz a két intenzitás különbségével. A Bob oldalán lévő fázismodulátor segítségével be lehet állítani, hogy melyik kvadratúrát szeretné mérni Bob, így megkapja az adott kvadratúra értéket, melynek értéke annak szórása miatt eltérhet az Alice által küldöttől. Ilyen impulzusokból Alice annyit küld, amennyiből előállíthatnak egy megfelelő méretű kulcsot. Az elküldendő impulzusok számára hatással van az esetleges zaj és egy rosszindulatú támadó lehetséges jelenléte is.

3.2.2. A protokoll működésének lépései

A protokoll egyes lépéseinek leírása és a használt képletek [13] alapján készültek. Ahogy láthattuk korábban a megvalósítás alatt álló protokoll koherens állapotokat használ és hasonlóan a BB84-hez itt is az első fázis abból áll, hogy Alice elküldi Bobnak az által elkészített kvantumos információt, amely tartalmazza az általa sorsolt klasszikus információt. A 3.2.1. fejezet végén láttuk, hogy miután Bob végrehajtja a

mérést, csak az egyik kvadratúrához tartozó információ lesz a birtokában és az sem fog teljesen megegyezni Alice-éval. A protokoll kvantumos része itt véget ért.

A második fázisban már csak klasszikus kommunikáció megy végbe, ami egy publikus, de autentikált csatornát használ. Az első lépés, hogy Bob átküldi Alicenak a klasszikus csatornán, hogy mely kvadratúrákat mérte meg az első fázis során. Ez a lépés azért fontos, mert bár Alice két kvadratúrában is kódolta az információt, de Bob csak az egyiket tudta megmérni a homodin detekcióval, így a másikat Alice-nak el kell dobnia. Ezután a lépés után mindkét félnek ugyanazon kvadratúrákhoz lesz csak értéke. A következő lépés, hogy a két fél felfedi a náluk lévő két adatsor egy részét (természetesen ez egymáshoz tartozó elküldött és fogadott értékeket jelent). Ezen adatok segítségével elvégzik a paraméterbecslést, melynek során becslést adnak az átviteli veszteségre és az úgynevezett *excess noise*-ra, mely egy plusz tag, ami hozzáadódik a kvadratúrák szórásához. Ezen becslések alapján el tudják dönteni, hogy mennyi a közöttük lévő információ I_{AB} és hogy mennyi a rosszindulatú támadó, Eve által birtokolt információ χ. Ha a közöttük lévő információ kisebb, mint az Eve által birtokolt, akkor abbahagyják a protokoll futását, hiszen Eve-nek több információja van a kulcsról. A paraméterbecsléssel a 3.2.2.1. fejezet foglalkozik bővebben.

Ha $I_{AB} > \chi$, akkor a következő lépés az úgynevezett *information reconciliation*[14]. Ebben a pillanatban Alice-nak és Bob-nak bár ugyanazon kvadratúrához tartozó értékei vannak már csak, de ezek nem egyeznek meg a küldés során fellépő csatornaveszteség, a kvadratúrák szórása, a mérési közbeni zaj és Eve miatt. Ennek a lépésnek az a célja, hogy kialakuljon mindkét oldalon egy, a másikéval megegyező adatsorozat. Így ez a lépés felfogható egyfajta hibajavításként is. Az *information reconciliation* lehet egyirányú, amikor csak az egyik fél küld információt és lehet kétirányú is. Általában az egyirányút szokták használni, mely két további fajtára bontható attól függően, hogy ki javítja ki az adatsorozatát. Ha Alice értékei alapján Bob javítja ki a saját adatát, akkor a *direct reconciliation*-ről beszélünk, míg ha Alice javítja ki a nála lévő adatsorozatot Bob adatai alapján, akkor *reverse reconciliation*-ről lehet beszélni. A *direct* esetben, ha a teljes veszteség 3 dB, akkor Eve több információval rendelkezhet, így nem lehet egy közös titkos kulcsot létrehozni [15]. A *reverse* változat esetében ez a 3 dB-s határ legyőzhető, mivel itt Bob küldi el a méréseinek eredményét Alice-nak és mivel Alice mindig több információval rendelkezik Bob mérésének eredményéről, mint Eve, ezért a I_{AB} is nagyobb lesz mint χ. Az *information reconciliation* során használt algoritmusokkal a 3.2.2.2. fejezet foglalkozik röviden.

Az *information reconciliation* utáni következő lépés, hogy meggyőződjenek arról a felek, hogy a hibajavítás tényleg sikeres volt és mindkét félnél ugyanaz az adatsorozat van, ezt a lépést *confirmation*-nek hívják. Ezután történik meg a *pri-*

vacy amplification [16]. Ekkor Alice-nál és Bob-nál már azonos adatsorozatok vannak (nagy valószínűséggel), de Eve még mindig birtokolhat valamennyi információt ezekről. Hogy ezt az információját megszüntessék a *privacy amplificaiton* segítségével a két fél előállít a közös adatsorozatból egy titkot, amit közös kulcsként fognak felhasználni. Így megszületett a szimmetrikus kulcs, amit később tudnak használni a felek. Ezzel a két lépéssel a 3.2.2.3. fejezet foglalkozik röviden.

Fontos megjegyezni, hogy mint ahogy láthattuk a protokoll második felében, fontos, hogy a felek publikus, de autentikált csatornán beszéljenek, hiszen csak így lehetnek biztosak abban, hogy tényleg egymással kommunikálnak és nem a támadó ékelte be magát közéjük. Ehhez szükséges, hogy az ottani kommunikációt autentikálják egy már meglévő kulccsal, valamilyen módon. Emiatt elmondható, hogy a QKD protokoll nem hoz létre a semmiből egy közös titkos kulcsot, mivel működéséhez elengedhetetlen egy már létező kulcs.

3.2.2.1. Paraméterbecslés

A porotokoll egyik legfontosabb tulajdonsága az elérhető kulcsráta K, azaz, hogy milyen sebességgel lehet kulcsot előállítani az alkalmazásával. A kulcsrátát a következő módon lehet megadni:

$$K = f_{symbol} * r.$$

Az f_{symbol} jelöli, hogy milyen gyorsan érkeznek a szimbólumok (szimbólum/ms), r pedig a titok hányad, azaz, hogy mennyi kulcsbit van egy átvitt szimbólumban (bit/szimbólum).

A kulcsrátára *reverse reconciliation* esetében a következő alsó becslés adható

$$r \geq (1 - \mathrm{FER})(1 - \nu)(\beta I_{AB} - \chi_{EB}),$$

ahol FER $\in [0, 1]$ a kerethiba ráta, $\beta \in [0, 1]$ az *information reconciliation* hatékonysága, $\nu \in [0, 1]$ azon szimbólumok hányada, amelyet paraméterbecslés alatt fel kellett használni, χ_{EB} pedig a Holevo korlát (vagy információ) [17], ami egy felső becslést ad arra, hogy mennyi információja lehet Eve-nek Bob adatáról. I_{AB} és χ_{EB} kivételével a többi paraméter ismert, ezen két paramétert kell becsülni a paraméterbecslése során. Eve estében feltételezhetjük, hogy eléri a Holevo korlátot, így ez az alsó korlát egy egyenlőséggé módosul. Ekkor χ_{EB} a Holevo-információt adja.

A protokoll első lépésében láttuk, hogy Alice a két kvadratúra, $\hat{q}$ és $\hat{p}$ értékét véletlenül sorsolja. Legyen ez a két érték két azonos eloszlású, de független normál eloszlásból választva Q és P, melyekre igaz, hogy

$$Q \sim P \sim N(0, \tilde{V}_{mod})$$

.

A *signal-to-noise-ration* vagy másnéven SNR felírható a következőképpen:

$$\mathrm{SNR} = \frac{T * V_{mod}}{1 + \xi},$$

ahol $V_{mod} = 4\tilde{V}_{mod}$ a kvadratúrákhoz tartozó *operátorok* varianciája, T a csatorna vesztesége és ξ a korábbi *excess noise*. Az SNR segítségével pedig felírható az Alice és Bob közötti információ a következőképpen:

$$I_{AB} = \frac{1}{2}\log_2(1 + SNR) = \frac{1}{2}\log_2(1 + \frac{T * V_{mod}}{1 + \xi}).$$

T, ξ és χ_{EB} becsléséhez szükség van újabb jelölések bevezetéshez.

Egy N módosú Gauss állapotot (Gauss állapotnak nevezünk egy olyan állapotot, melynek fázistérben felírt eloszlásfüggvénye Gauss-eloszlást követ) meg lehet adni egy $2N$ dimenziós elmozdulásvektorral $\widehat{\mathbf{x}}$ és $2N \times 2N$ dimenziós kovariancia mátrixszal. Az elmozdulásvektort meg lehet adni a következőképpen:

$$\widehat{\mathbf{x}} = (\hat{q}_1, \hat{p}_1, \hat{q}_2, \hat{p}_2, ..., \hat{q}_N, \hat{q}_N)$$

Az elmozdulásvektor minden tagjára teljesül, hogy

$$[\hat{x}^j, \hat{x}^k] = 2i\Omega^{jk},$$

ahol Ω egy $2N \times 2N$ dimenziós mátrix, mely kifejezhető a következő módon:

$$\Omega = \bigoplus_{l=1}^{N} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.$$

A kovariancia mátrix elemeit pedig felírhatjuk úgy, hogy:

$$\sum^{ij} = \mathrm{E}(\hat{x}^j, \hat{x}^k).$$

Emiatt a főátló mentén az elemek:

$$\sum^{jj} = V(\hat{x}^j)$$

A kovariancia mátrix arra világít rá, hogy milyen korreláció van az egyes módusok operátorai között. Milyen hatással van ez egyikre, egy másik változása.

Egy Gauss állapot Neumann entrópiája leírható az állapot szimpleptikus sajátértékeinek segítségével és a következőképpen adható meg:

$$S = \sum_i g(\nu_i).$$

Itt ν_i az állapot $\sum$ kovariancia mátrixának szimpleptikus sajátértékei és $g(\nu_i)$, pedig a következőképpen írható fel:

$$g = (\frac{\nu+1}{2}) \log_2(\frac{\nu+1}{2}) - (\frac{\nu-1}{2}) \log_2(\frac{\nu-1}{2})$$

A χ_{EB} Holevo-információ kiszámítható a következőképpen:

$$\chi_{EB} = S_{AB} - S_{A|B},$$

ahol AB Alice és Bob "közös" állapota Bob mérése előtt, míg $A|B$ a "közös" állapota Bob mérése után.

Miután Alice elkészítette véletlenül sorsolt kvadratúrákkal rendelkező koherens állapotot és elküldte Bob-nak egy kvantumos csatornán, melynek vesztesége T és a zaj ξ, ezután ezt Bob megmérte, akkor a közös kovariancia mátrix a következő:

$$\sum = \begin{pmatrix} V_{mod}\mathbb{I}_2 & \sqrt{T}V_{mod}\mathbb{I}_2 \\ \sqrt{T}V_{mod}\mathbb{I}_2 & TV_{mod} + 1 + \mathbb{I}_2 \end{pmatrix}$$

Ebből a mátrixból megkapható a [13]-ban leírtak alapján mind χ_{EB}, T és ξ

3.2.2.2. Information reconciliation

Ahogy említettem korábban az *information reconciliation* során egyfajta hibajavítást hajtunk végre. A folyamat során a két fél információt küld egymásnak (a protokoll típusától függ, hogy ki kinek), amely segítségével úgy javíthatják a náluk lévő adatokat, hogy az nagy valószínűséggel megegyezzen a két oldalon. Természetesen ezt az információt Eve is el tudja olvasni (mivel a klasszikus csatorna csak autentikált, de nem titkosított), ezért mindezt úgy kell megtenniük, hogy a lehető legkevesebb információ szivárogjon ki a nyers kulcsról.

A különböző QKD protokollokhoz sokfajta *information reconciliation* megoldást dolgoztak ki. A diszkrét esetben a legtöbbet használt algoritmus a Cascade [14] vagy a Winnow [18]. Ezek a megoldások bitsorozatokat javítanak ki egymáshoz képest. A folytonos esetben például az egyik sokat használt algoritmus a *sliced error correction* [19], ami a folytonos változókat először diszkretizálja és utána javítja őket. Egyes *information reconciliation* megoldásokat több dimenzióban is lehet végezni.

Ekkor az értékeket nem egyesével javítjuk, hanem nagyobb csoportokban. Ezzel a megközelítéssel gyorsítani lehet a feldolgozást. Egy ilyen kiterjesztésről lehet olvasni a [15]-ben vagy [20]-ban. A Cascade és a *sliced error correction* működését a 4.1.1. és a 4.1.2. fejezet mutatja be részletesen.

3.2.2.3. Confirmation és Privacy amplification

Ahogy említettem, a *confirmation* lépés során a két fél el szeretné dönteni, hogy tényleg azonos sorozat van-e náluk. Ehhez egy univerzális hash függvényt használhatnak. Véletlenszerűen kiválasztanak egy univerzális hash függvényt és elkészítik a náluk lévő adatsorozat lenyomatát. Ezután összevetik, hogy mindkét oldalon ugyanaz az érték található-e. Ha igen, akkor folytathatják a protokollt, ha nem akkor megállnak.

A *privacy amplification* lépésben a két fél, azért, hogy Eve-nek a maradék információját is megszüntessék a kulccsal kapcsolatban egy *seed*-elt véletlenség extraktort használnak, amivel előállítják a közös kulcsot. Itt is általában egy univerzális hash függvényt használhatnak.

4. fejezet

A klasszikus kommunikáció implementálása

4.1. A lehetséges information reconciliation protokollok

Az irodalomkutatás során megkerestem azokat az *information reconciliation* megoldásokat, amelyeket széles körben használnak a QKD rendszerekben, képesek magas kulcsrátát elérni és hatékonyak a hibák javításában. A használandó módszer kiválasztása során természetesen arra törekedtem, hogy egy *reverse reconciliation* protokollt találjak, mivel ahogy korábban említettem ezzel a változattal lehet viszonylag alacsony SNR mellett is végrehajtani a kulcs javítását. Ez azt is jelenti, hogy a jövőben esetlegesen nagyobb távolságokon is használhatóak lehetnek az eszközök. Természetesen ez nem csak a *information reconciliation*-t végző protokollon, hanem az eszközök fizikai tulajdonságain is múlik.

Ebben a fejezetben bemutatok két protokollt, amit előszeretettel használnak meglévő QKD rendszerekben. A működésük ismertetése után kitérek a használhatóságukra általában és a mi rendszerünk keretében. A fejezet végén pedig kifejtem, hogy miért esett a választásom a legvégül implementált protokollra.

4.1.1. A Cascade protokoll

Brassard és munkatársai a [14] cikkben mutatták be a Cascade protokollt, de céljuk nem egy *information reconciliation* eljárás létrehozása volt a QKD protokollok számára, hanem azzal a problémával foglalkoztak, hogy két fél (Alice és Bob) szeretné egy publikus csatornán megbeszélni, hogy mindkét félnél ugyanaz a kulcs van-e, miután Alice elküldte azt Bobnak egy bináris zajos csatornán (BSC). Mindezt természetesen úgy szeretnék megtenni, hogy egy harmadik fél számára a lehető

legkevesebb információt fedik fel erről a kulcsról. Láthatjuk, hogy ez az elrendezés nagyban hasonlít arra az állapotra, ami előáll egy QKD protokoll során a kvantumos kommunikáció után. Abban viszont eltér, hogy milyen közegben történt a kulcs átvitele és hogy milyen paraméterekkel rendelkezik a csatorna. A Cascade-t felhasználták pl. a BB84-es protokollban is, hiszen a szerzői részben ugyanazok voltak. A protokoll működésének leírását [14] és [21] alapján készítettem el.

Ahogy említettem a protokoll ott illeszkedik be a kvantumos kulcsmegosztásba, ahol a kvantumos csatornán történő kommunikáció lezajlott. Egy ilyen kulcsmegosztásban ezen a ponton Alice-nál és Bob-nál elméletben ugyanaz az adatsorozat elérhető, de a gyakorlatban a csatorna tulajdonságai, a mérési berendezés és akár egy rosszindulatú támadó hatása miatt a Bob-nál mért értékek eltérnek az eredetiektől. A Cascade feltételezi, hogy az adatsorozat (maga a nyers kulcs) egy bináris sorozat és a működéséhez szüksége van még a kvantumos bithiba-valószínűségre is (QBER - *quantum bit error rate*). A bithiba mondja meg, hogy mekkora a valószínűsége, hogy a kommunikáció során egy bit átbillent (egyről nullára vagy nulláról egyre). Ennek az értékét a felek meg tudják becsülni az adatsorozat egy részének felhasználásával. A Cascade lefutása után a két oldalon nagy valószínűséggel ugyanaz az adatsorozat lesz megtalálható és megadja a kiszivárgott információ mennyiségét is. Fontos megjegyezni, hogy a két oldalon tényleg csak nagy valószínűséggel lesz egyenlő a két adatsorozat, a Cascade nem vállalja, hogy biztosan kijavít minden hibát. Természetesen a Cascade futása után kialakult kulcsokban a bithibák valószínűsége sokkal kisebb lesz. Ahhoz, hogy a két fél megbizonyosodjon arról, hogy a Cascade futása sikeres volt a lépések leírásánál is említett *confirmation*-t kell lejátszaniuk.

A Cascade működése során Alice és Bob iterációkat hajt végre, amelyek során Bob kérdéseket tesz fel partnerének és a válaszok alapján próbálja meg kijavítani az eredeti kulcs nála lévő zajos változatát. Ebből látható, hogy ez a protokoll a *forward reconciliation* megoldások közé tartozik. Négy iterációt hajtanak végre a felek és mindegyik, kivéve az elsőt úgy kezdődik, hogy Bob megkeveri a nála lévő biteket. Ez a keverés nem kell, hogy titkos legyen vagy hogy használjon valódi vagy álvéletlenséget, de bizonyos megkötéseknek meg kell felelnie. Az egyik, hogy Bob tudja, hogy melyik iterációnál milyen keverést használt, a másik pedig, hogy az egyik iterációban se kerüljön egy bit ugyanabba a pozícióba.

A következő lépésben Bob felosztja a már összekevert bitsorozatot egyenlő részekre. Ha esetleg a kulcs hossza nem többszöröse a blokk méretének, akkor az utolsóba kevesebb bit kerül. A blokkok mérete függ az aktuális iteráció sorszámától és a korábban megbecsült QBER értékétől is. A blokkméretek az iteráció során a következőképpen változnak:

$$k_{1,Q} = 0.73/Q$$
$$k_{2,Q} = 2 * k_{1,Q}$$
$$k_{3,Q} = 2 * k_{2,Q}$$
$$k_{4,Q} = 2 * k_{3,Q}$$

A fentiekben $k_{i,Q}$ az i. iterációban használt blokkméret, míg Q a becsült QBER érték. Ahogy látható a képzési szabályokból, a blokkok mérete az iterációkkal együtt csökken. Az első iterációban használt blokkméretre a következő logika adható [21]: Ha a mérete $1/Q$ lenne, akkor QBER definíciója alapján egy blokkba átlagban egy hibás bit kerülne. Ha $0.73/Q$ használunk, akkor egy blokkban átlagban egynél kevesebb hibás bit lesz megtalálható. Viszont nem használhatunk túl kicsi blokkokat sem, mert később Alice meghatározza ezek paritását és egy kisméretű blokk paritása sok információt szivárogtat ki a támadónak.

Miután Bob felosztotta egységes méretű blokkokra a nála lévő bináris sorozatot, minden blokknak kiszámolja a paritását. Ezután Bob megkéri Alice-t, hogy küldje el neki az ezen blokkok nála található paritás értékét. Bob fel tudja használni a saját maga által kiszámolt és a megkapott, helyes paritásértéket arra, hogy megtudja az adott blokkban páros vagy páratlan számú bithiba található-e. Ez azért lehetséges, mivel ha egy adott blokkban páratlan számú bit lesz átbillentve, akkor a paritás is páratlan alkalommal változik, így 0-ból 1 illetve 1-ből 0 lesz. Hasonlóan a páros számú átbillenésre, a paritás ekkor nem változik. Így ha Bob azt tapasztalja az egyik blokknál, hogy pl. az általa kiszámolt paritás 1, míg az Alice által kapott érték ehhez a blokkhoz 0, akkor biztos lehet benne, hogy páratlan számú hiba van a blokkjában. A paritások összevetése után Bob csak a hibák számának páros vagy páratlan mivoltát ismeri, azt, hogy melyik bitek billentek át nem. Természetesen a paritások elküldése szivárogtat információt a kulcsról: Ha egy blokk mérete k, akkor 2^k lehetséges értéke lehet. Viszont ha tudjuk a paritást, akkor csak 2^{k-1} lehetőséget kell végigpróbálni.

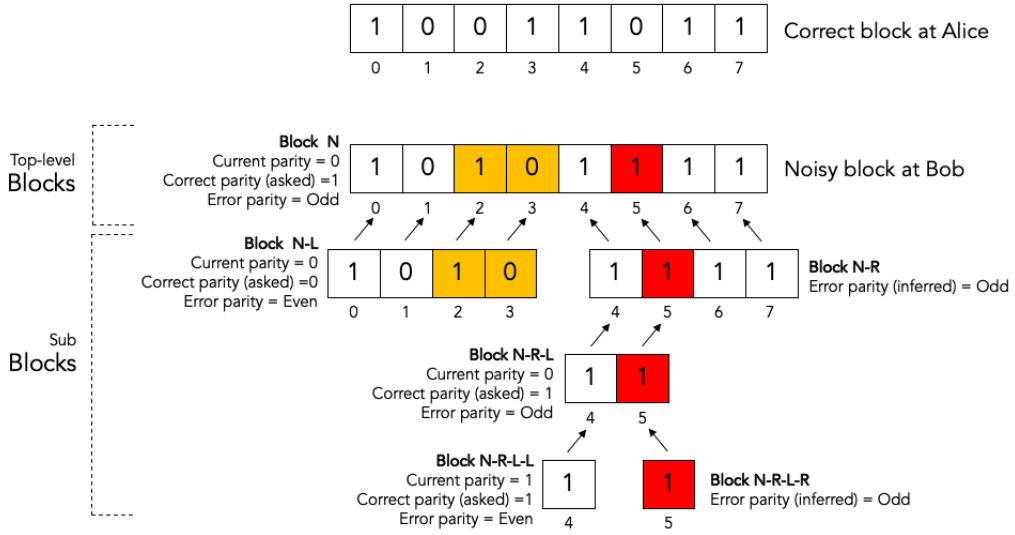
Bob ezután csak azokkal a blokkokkal foglalkozik, amelyekben pártalan számú hiba található, mivel a páros számú hibát tartalmazó blokkban van lehetősége arra, hogy 0 hiba van, így ezeket nem fogja megpróbálni kijavítani. A páratlan blokkokra végrehajtja a *binary* algoritmust, ami pontosan egy bithiba kijavítására képes. A *binary* algoritmus lefutása után a blokkban található hibák száma páros lesz, ha pedig elfogyott az összes páratlan hibát tartalmazó blokk, akkor a felek a következő iterációba lépnek.

A *binary* algoritmus egy olyan blokkon dolgozik, amiben páratlan számú hiba található. Az első lépés, hogy a blokkot felosztja két egyenlő részre (bal és jobb rész). Ha esetleg nem páros hosszú a blokk, akkor a baloldali egy bittel többet fog tartalmazni. Ezután Bob kiszámolja a két fél paritását. Mivel a teljes blokk paritása páratlan, ezért a hibás bitek elhelyezkedése a következő lehet: vagy az egyikben található páros számú hiba, míg a másikban páratlan vagy fordítva. Olyan lehetőség így nem állhat fent, amikor mindkét fél páros vagy páratlan hibát tartalmaz. Bobnak ezután már csak azt kell kiderítenie, hogy pontosan melyik fél tartalmazza a páratlan számú hibát. Ehhez ismét megkéri Alice-t, hogy a két fél közül valamelyikhez küldje el a helyes paritás értéket. Teljesen mindegy, hogy melyik félhez tartozó értéket kéri Bob, mivel az egyik páros, a másik pedig páratlan számú hibát tartalmaz, így bármelyik paritását a helyes értékkel összehasonlítva megkapja, hogy pontosan melyikben van a páratlan számú hiba.

Miután Bob megtalálta a páratlan számú hibát tartalmazó felet, lefuttatja rajta a *binary* algoritmust. Ez a rekurzív végrehajtás pontosan addig tud működni, amíg a páratlan hibát tartalmazó blokk mérete 1 nem lesz. Ebben az esetben viszont pontosan tudjuk, hogy ez az egy bit hibás, így könnyen visszabillenthető a megfelelő állapotba. A rekurzív hívások segítségével Bob sikeresen kijavított pontosan 1 hibát a pártalan hibát tartalmazó blokkban, tehát a hibák száma páros lett. A *binary* algoritmus rekurzív meghívására látható egy példa a 4.1 ábrán. A felső sorozat a helyes blokk Alice-nál, alatta a zajos változat látható, ami Bob-nál található. A hibás bitek sárgával és pirossal vannak jelezve. Látható, hogy először két részre lesz osztva a blokk, majd miután kiderült, hogy melyik félben van páratlan számú hiba (jobb oldali), az egész folyamat lejátszódik újra ezen a félblokkon (az ábrán ezután csak az a fél látszik, ami a páratlan hibát tartalmazza). A folyamat legvégén Bob megtalál egy hibás bitet, amit ki tud javítani.

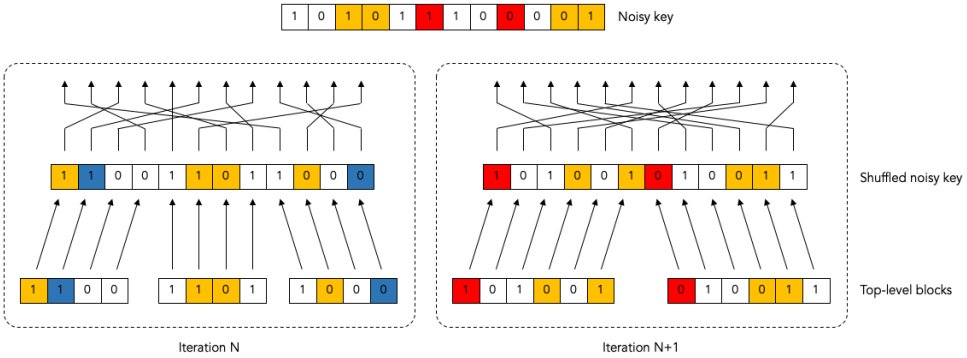
Az adott iteráció tehát akkor zárul le, amikor minden blokk páros számú hibát tartalmaz. Természetesen ezt nem azt jelenti, hogy már nincsen hiba a Bob oldalán lévő kulcsban (hiszen lehet bennünk 2, 4, 6, stb.), sem azt, hogy ezeket a blokkokat ne lehetne tovább javítani. Az ilyen blokkban lévő hibákat egyrészt a következő iteráció elején végbemenő átrendezés olyan blokkba is rakhatja, ami páratlan hibát fog tartalmazni, így ott Bob ki fogja tudni javítani. Másrészt a protokoll nevét is adó Cascade effektus is lejátszódik az iterációk alatt, ami nagyban befolyásolja a páros blokkok kezelését.

A Cascade effektus a következőképpen működik: Az N. iterációban Bob kijavít egy bitet egy páratlan hibát tartalmazó blokkban, így ezen iteráció után ennek a bitnek az értéke már helyesen szerepel az ő oldalán. Ezt viszont a korábbi összes iterációban még nem tudta, így ha esetleg pont emiatt a hiba miatt tartalmazott az

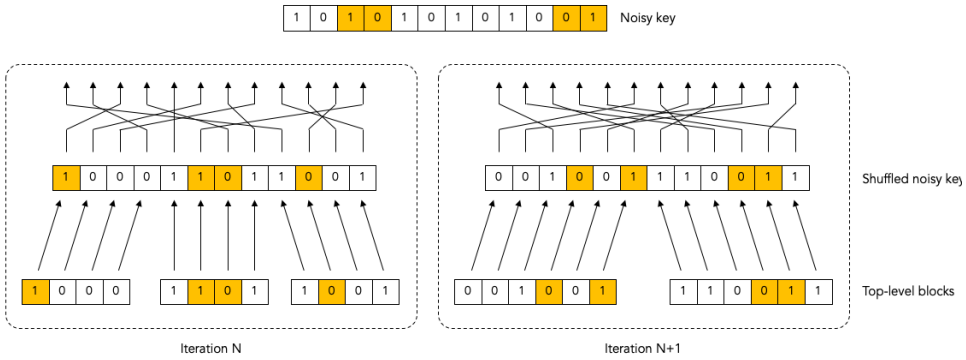


4.1. ábra. A *binary* algoritmus egy kisméretű blokkon. Forrás: [21]

egyik korábbi blokk páros számú hibát, akkor abban az iterációban nem foglalkozott vele. Azáltal viszont, hogy most már tudja Bob, hogy az a bit hibás volt, a korábbi összes iterációban előálló páros hibát tartalmazó blokkot meg tudja vizsgálni és ha emiatt változik a hibák száma, akkor a *binary* algoritmus segítségével ki tud javítani még egy hibát. Ennek a bithibának a kijavítása viszont hasonlóképpen kihat a többi iterációra és nem csak azokra, amelyek korábban futottak le. A Cascade effektus hatására a hibák javítása további lehetőségeket hoz létre és ezáltal hatékonyan tudja Bob a nála lévő kulcsot a megfelelő állapotra hozni. Természetesen újra le kell szögezni, hogy a Cascade protokoll nem biztosítja, hogy minden hiba ki lesz javítva. A Cascade effektust mutatja be a 4.2 és a 4.3 ábra. Az 4.2 ábra felső sorában látható a zajos kulcs Bob oldalán az $N + 1$. iteráció előtt, ahol a hibás biteket a piros és a sárga szín jelöli. A pirossal jelzett bitek azok, melyeket a *binary* algoritmus segítségével Bob ki tud javítani az $N+1$. iterációban, miután az iteráció elején megkeverte a biteket, blokkokat képzett és meghatározta, hogy melyik blokk tartalmaz páratlan számú hibát (ezeket a lépéseket az ábra jobb oldalán lehet látni bekeretezve). Az ábra bal oldalán látható az N. iteráció. Ahogy észrevehető, a bitkeverés után nem volt olyan blokk, ami páratlan bithibát tartalmazott volna, így Bob ebben az iterációban nem tudott javítani. Viszont ha megjelöljük azokat a biteket, amiket a következő iterációban ki tudott javítani és a helyes értékeket írjuk be hozzájuk (ezek a kék bitek), akkor a 4.3 ábrán látható eredményt kapjuk és két olyan blokk is keletkezik, melyek most már páratlan hibát tartalmaznak. Ezekben újra javíthat Bob egy-egy hibát, melyek hatást gyakorolnak a többi iteráció blokkjaira és ez így folytatódik tovább.



4.2. ábra. A Cascade effektus előtti állapot, az $N+1$. iteráció eredménye után. Forrás: [21]



4.3. ábra. A Cascade effektus hatása. Forrás: [21]

Összefoglalásként megállapítható, hogy a Cascade protokoll képes hatékonyan elérni, hogy Bob oldalán nagy valószínűséggel az Alice által létrehozott kulccsal megegyező bitsorozat alakuljon ki. Ahogy az a protokoll működéséből látszik, a Cascade-t lehetséges átalakítani például az iterációk száma és ehhez kapcsolódóan a blokkméretek módosításával, amire több megoldás is született az évek során [22] [23]. A protokollt a QKD rendszerekben is előszeretettel használják, de főleg a diszkrét esetben, mivel ott bináris adatokat kapunk a mérés után. A mi rendszerünk viszont folytonos értékekkel dolgozik, így ahhoz, hogy a Cascade használható legyen diszkretizálni kell a folytonos értékeket, ami felvet kérdéseket azzal kapcsolatban, hogy hogyan lehet pl. a QBER értéket meghatározni. A másik probléma, hogy ez a protokoll *forward reconciliation* módon működik, ami felső korlátot ad az elérhető távolságra, ahogy azt korábban említettem.

4.1.2. A Sliced error correction

A Cascade protokoll egyik problémája volt, hogy csak bináris adatokon tudott dolgozni. A CV-QKD rendszerek megjelenésével kidolgoztak olyan megoldásokat is, amik már magukba foglalják a folytonos értékek diszkretizációját is. Az egyik ilyen megoldás a *Sliced error correction - SEC*, amely segítségével a CV-QKD rendszerekben is könnyűszerrel elvégezhető az *information reconciliation* lépés [19]. Ebben az alfejezetben bemutatom a SEC működését felhasználva a [19]-t.

Mint ahogy azt minden *information reconciliation* megoldás teszi, a SEC is a kvantumos fázis után lép be a képbe. Mivel folytonos változójú rendszerről beszélünk, ezért Alice és Bob ekkor folytonos értékekkel rendelkezik, amelyek nagy valószínűséggel nem egyeznek meg, de korrelálnak egymással. A protokoll először egy bináris sorozatra képzi le a két félnél elérhető adatokat és utána egy tetszőleges bináris hibajavító kód segítségével végzi el a Bob-nál lévő adatsorozat javítását. Tehát ilyen értelemben a SEC inkább egy általános keretrendszert biztosít, amiben felhasználható tetszőleges bináris hibajavító kód annak ellenére, hogy nem egy olyan csatorna áll fent a két fél között, melyre a hibajavító kód készült.

Ahhoz, hogy a protokoll lépéseit leírjam, be kell hogy vezessek definíciókat, melyek a protokoll sajátjai. Az első kettő a *slice függvény* és a hozzá tartozó *slice becslő függvény*. A *slice függvény* a folytonos értékek diszkretizálásáért felel és felfogható a távközlésben használt kvantálásnak is. Ezt a függvényt Alice használja és a jelölése $S(x)$, ahol x Alice folytonos értékeinek egy eleme. A függvény ezt az értéket képzi le a bináris térbe ($GF(2)$-be). Ezeket a $S(x)$ értékeket összefoghatjuk egy vektorba:

$$S_{1\ldots m}(x) = (S_1(x), \ldots, S_m(x)),$$

ami így 2^m lehetséges értékkel rendelkezik. Egy ilyen vektort egyként kezelve $K(X) = S_{1\ldots m}(x)$ megkapjuk a nyers kulcs bináris változatát, ahol X az Alice oldalán található valószínűségi változó, melyből az x értékek származnak.

A Bob oldalán található értékek a X' valószínűségi változóból érkeznek és x'-vel jelöljük őket. Bob a *slice becslő függvény*eket fog használni a következő felépítéssel:

$$\tilde{S}_1(x'), \tilde{S}_2(x', S_1(x)), \ldots, \tilde{S}_m(x', S_1(x), \ldots, S_{m-1}(x))$$

. Azaz a *slice becslő függvény* a Bob-nál megtalálható folytonos adatot képzi le $GF(2)$-re felhasználva az Alice-nál elérhető ezen adathoz tartozó helyes érték $GF(2)$-re vett leképezését. Bob ezekkel az értékkel tud egy becslést adni az $S(X)$ kimenetére. Természetesen a *slice becslő függvény*eket is lehet egy vektorba összefogni. Ezen függvények által kapott értékekből álló bináris sorozatot lehet az Alice-nál található helyes, nyers kulcs zajos változatának tekinteni. Természetesen mind a *slice függvény*, mind a *slice becslő függvény* megalkotása attól függ, hogy milyen X és X' eloszlása. A későbbiekben ki fogok térni arra, hogy az esetünkben használt normális (Gauss) eloszlás esetében milyen *slice* és *slice becslő függvény*eket lehet használni.

Ha Alice és Bob megegyeztek abban, hogy milyen *slice* és *slice becslő függvény*eket fognak használni, akkor a protokoll működése a következőképpen zajlik: Alice a nála található l hosszú nyers, folytonos adatából $x_1, x_2, \ldots, x_l$ előállít m darab l hosszú vektort a *slice függvény*ek segítségével:

$$(S_1(x_1), \ldots, S_1(x_l)), \ldots, (S_m(x_1), \ldots, S_m(x_l)).$$

A kommunikáció az első ilyen l hosszú vektorral kezdődik. Bob létrehoz egy bitsorozatot felhasználva a nála található $x'_1, \ldots, x'_l$ nyers, folytonos adatokat és az Alice-tól kapott $(S_1(x_1), \ldots, S_1(x_l))$ vektort. Ezután egy bináris hibajavító kódot használva Alice és Bob kijavítják a Bob-nál lévő bitsorozatot (Bob ezt akár egyedül is megteheti, ha megfelelő hibajavító kódot használ). A következő $2 \leq i \leq m$ körben Alice felhasználja a $(S_i(x_1), \ldots, S_i(x_l))$ vektort, Bob pedig használva az $\tilde{S}_i$ *slice becslő függvény*t elkészíti a becslését a helyes bitsorozatra kihasználva a $x'_1, \ldots, x'_l$ és a korábbi $S_1(x_1), S_2(x_2), \ldots, S_{i-1}(x_l)$ értékeket. Ezután a választott hibajavító kódot futtatva kijavítják Bob becslését. A folyamat végén Alice egymás után írja az m darab l hosszú vektort és így megkapja a nyers bináris kulcsot. Bob hasonlóképpen tesz a javított értékekkel.

A SEC célja, hogy ezeket a lépéseket úgy hajtsa végre, hogy a lehető legkevesebb információt szivárogtassa ki a folyamat során egy harmadik félnek. A [19]-ben adott bizonyítás szerint belátható, hogy elég csak megadni az első r *slice*-ot, azaz $S_{1\ldots r}(X^{(d)})$-t megadni, hogy a maradék $m - r$ *slice*-t Bob meg tudja becsül-

ni és a segítségükkel vissza tudja állítani a saját adait úgy, hogy megegyezzenek $S_{1\ldots m}(X(^{(d)})$-vel, ahol d a dimenziók száma (a bemeneti ábécé mérete). A képlet r-re pedig a következő:

$$r = \lfloor dH(K(X^{(1)} \mid X'^{(1)}) + 1) \rfloor,$$

ahol $H(K(X^{(1)} \mid X'^{(1)})$ a feltételes entrópia.

A megfelelő *slice* és *slice becslő függvény* létrehozása két részből áll. Először a lehetséges értékek intervallumát kell felosztani kisebb intervallumokra ($T(X)$), majd egy m, bináris értéket kell hozzájuk rendelni. Mindezt úgy érdemes elvégezni, hogy $I(T(X), X')$-ot maximalizáljuk, hiszen ez segíti Bob-ot abban, hogy jól tudjon becsülni. A *slice becslő függvény*ek a *most likelihood* megoldást követik. A normál eloszlása esetében a valós számokat t intervallumra bontjuk fel $t-1$ változó segítségével (ezek lesznek a határok): $\tau_1, \tau_2, \ldots, \tau_{t-1}$ és $\tau_0 = -\infty$, $\tau_t = +\infty$. Egy a intervallumra $(1 \leq a \leq t)$ igaz, hogy $\{x : \tau_{a-1} \leq x \leq a\}$. Ahhoz, hogy maximalizáljuk az említett közös információt, szimmetrikus határokat kell használni $\tau_a = \tau_{t-a}$.

A *slice*-okhoz való kód rendelése pedig [19] alapján akkor hozza a legjobb eredményt (feltéve, hogy $t = 2^m$ alakú), ha $a-1$ bináris reprezentációjának legkisebb helyi értékű bitjét rendeljük $S_1(x)$-hez, ha $\tau_{a-1} \leq x \leq a$ és ezután a maradék biteket a legnagyobb helyi érték felé haladva hozzárendeljük a *slice*-okhoz egészen $S_m(x)$-ig.

A SEC-ről elmondható, hogy megoldja azt a problémát, amire a Cascade nem adott megoldást és biztosít egy olyan keretet, amiben tetszőleges bináris hibajavító kódot lehet használni (akár a Cascade-t is). Az említett bizonyítás eredményeként a kiszivárgott információ is csökken és akár 3 dB-es jel-zaj viszony mellett is képes hatékonyan működni. De mivel *forward reconciliation*-ről van szó és bizonyították, hogy 3 dB-nél lejjebb nem tud működni [15], ezért a kitűzött célt, miszerint nagy távolságokat érjünk el a készülő rendszerrel nem tudja támogatni. Emiatt nem ezt a protokollt választottam.

4.1.3. A megfelelő protokoll kiválasztása

Ahogy a fejezet elején is említettem, elsődleges célom egy olyan *information reconciliation* protokoll megtalálása volt, ami jól használható alacsony jel-zaj viszony mellett is (így nagy távolság érhető el a rendszerrel) és képes hatékonyan javítani a hibákat.

A bemutatott két protokoll bár rendelkezik jó tulajdonságokkal, de vagy a folytonos értékek megléte (Cascade) vagy az alacsony SNR elérése (sliced error correction) problémát okozhat számukra, ezért úgy döntöttem, hogy egy harmadik megoldást választok.

A következő fejezetekben bemutatott *information reconciliation* protokoll [24]-ben került leírásra, de használ eredményeket [20]-ból is. A protokoll működésének megértéséhez elengedhetetlen, hogy először bevezessem az LDPC kódok területét [25] alapján. A 4.2. fejezetben ezt fogom megtenni, majd azután áttérek a protokoll tényleges működésére a 4.3. fejezetben.

4.2. Az LDPC kódok

Az LDPC kódok a hibajavító kódok családjába tartoznak. Első megjelenésük az 1960-as években volt, amikor Robert G. Gallager megalkotta doktori disszertációjának részeként [26]. Ekkor még nem terjedtek el és egészen a 90-es évekig kellett várni arra, hogy újra felfedezzék őket [27]. Azóta igen gyakran használják őket például szatelitek kommunikációja során vagy mélyűri hálózatok létrehozásánál. Képesek a szintén közkedvelt turbó kódoknál is jobban teljesíteni bizonyos körülmények között. Az LDPC kódok bemutatását [26] és [25] alapján készítettem el.

Az LDPC a *Low Density Parity Check*-ből képzett betűszó, ami alacsony sűrűségű paritásellenőrzést jelent. Ahhoz, hogy ezt a kifejezést megértsük be kell vezetni a paritásellenőrzés fogalmát. A paritásellenőrzés a hibajavító blokk kódok világából ered, melyben a kódolás nem karakterenként, hanem karakterek csoportjára, blokkonként történik. A blokknak fix hossza van és a blokkban található karakterekre fel tudunk írni bizonyos kapcsolatokat. Ha az átküldésre szánt információ bináris, akkor a karakterek megfelelnek a 0-s és 1-es bitnek. Ekkor ezek a kapcsolatok az egyes bitek között felírhatóak egy egyenlettel, melynek kimenetét mi adhatjuk meg. Ha például a K blokk $x_1x_2x_3x_4$ bitekből épül fel, akkor például az $x_1 + x_2 = 0$ és $x_2 + x_3 = 1$ egyenletekkel (természetesen modulo 2) meghatározhatunk kapcsolatokat az egyes bitek között. Az első egyenlet azt fejezi ki, hogy a x_1 és x_2 lehetséges értékei az $(x_1 = 0, x_2 = 0)$ és az $(x_1 = 1, x_2 = 1)$ párok lehetnek, mivel csak akkor teljesül, hogy az összegük 0 lesz. Hasonló leírás adható a második egyenletre. Sőt, mivel x_2 mindkét egyenletben szerepel, így ha megtudjuk x_3 értéket (vagy x_1-ét), akkor megkaphatjuk x_1 (vagy x_3) értéket x_2-n "keresztül". Ez a technikát hívják paritás ellenőrzésnek. A gyakorlatban azt várjuk el, hogy az összegek 0-t adjanak minden egyenletre. Ezeket az egyenleteket arra tudjuk felhasználni, hogy az információ átvitele során bekövetkezett bithibákat javítsuk. Ha az átvitt blokkra a vevő oldalán nem teljesülnek a leírt egyenletek, akkor tudjuk, hogy bizonyos bitek megváltoztak és az egyenletek segítségével helyre tudjuk őket állítani. Természetesen ehhez az kell, hogy minden bit szerepeljen megfelelő mennyiségű egyenletben.

Ezeket az egyenleteket tömörebben egy mátrix segítségével tudjuk reprezentálni, amelyben minden oszlop megfeleltethető az egyik bitnek és a mátrix sorai pedig

az egyes egyenleteknek. A példában megadott K blokkra és az egyenletekre felírható a következő mátrix:

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}.$$

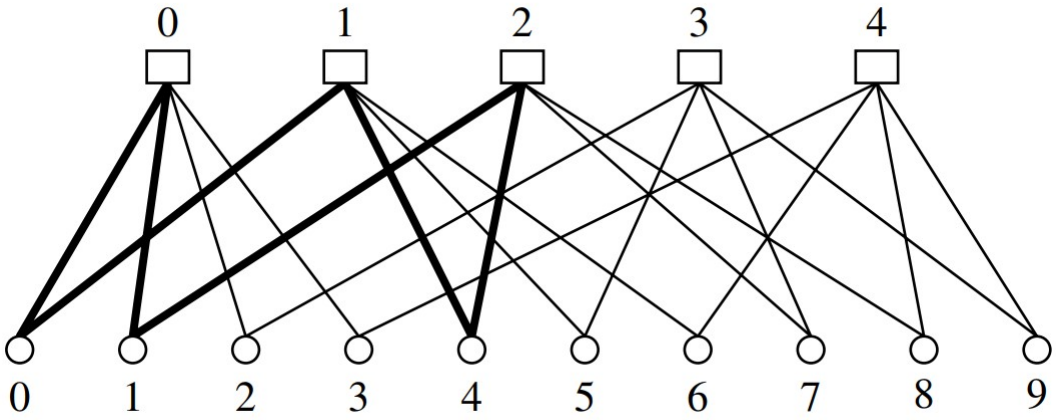
A paritásellenőrző mátrixokat a generátor mátrixból lehet kiszámolni, melyek arra használhatóak, hogy előállítsuk a különböző kódszavakat. A kódolás folyamata ezen kódok esetében nagyon egyszerű. A kódolandó blokkot, mint vektort kell értelmezni és megszorozni a generátor mátrixszal. Az így kapott kódszót küldjük el a másik félnek. A kódszavakra igaz, hogy ha vesszük a szorzatukat a paritásellenőrző mátrixszal, akkor a csupa 0 vektort kapjuk. Így például remekül lehet ellenőrizni, hogy egy adott bitvektor lehet-e kódszó (ez a dekódolásnál fontos lesz). Ahhoz, hogy a kommunikáció megfelelően működjön természetesen szükség van arra, hogy mindkét fél rendelkezzen a generátor mátrixszal (és így a paritás ellenőrző mátrixszal is). A hibajavító kódok fejlődése során sok konstrukció született arra, hogy pontosan mely bitek között is kéne felírni ilyen kapcsolatokat attól függően, hogy milyen csatornát használunk a kommunikációra vagy hogy milyen adatot szeretnénk átvinni. Érdekes módon azt sikerült bebizonyítani, hogy akkor a leghatékonyabb ez a módszer, ha véletlenszerűen kötjük össze az egyes biteket.

Az LDPC kódok feloldásában szereplő *Low Density* azt jelenti, hogy a kódhoz felírt paritásellenőrző mátrixban az 1-esek száma alacsony (alacsony a sűrűségük). Ez azért nagyon fontos, mivel ha nagy mennyiségű 1-es szerepel a mátrixban, akkor azt jelenti, hogy az egyenletekben sok változó szerepel, ami komplexebbé teszi ezen egyenletek kielégítését és így hosszabb ideig tarthat a dekódolás.

Az LDPC kódok reprezentálása kétfajtaképpen történhet, melyek természetesen ekvivalensek egymással. A most bemutatott paritásellenőrző mátrix mellett, melyet **H**-val szoktak jelölni, egy a bitek közötti kapcsolatokat talán jobban bemutató megoldás a Tanner-gráf, mely a kitalálójáról Michael Tanner-ről kapta a nevét. Ez egy páros gráf, melynek csomópontjai két csoportba tartoznak, az egyik a változó csomópontok (*variable node*), másik pedig az ellenőrző csomópontok (*check node*). Az ellenőrző csomópontok megfeleltethetőek egy egyenletnek a paritásellenőrző mátrixban: Ha egy változó csomópont és egy ellenőrző csomópont között van él, akkor az a változó csomópont benne van az adott egyenletben. Egy Tanner-gráfra és a hozzá tartozó paritásellenőrző mátrixra látható egy példa a 4.4. és a 4.5. ábrán.

4.2.1. Az LDPC kódok dekódolása

Az LDPC kódok dekódolásához iteratív dekódolókat használnak, melyek közel optimálisak tudnak lenni, ha a kódhoz tartozó parításellenőrző mátrixot megfelelően hoztuk lére (azaz ügyelve arra, hogy milyen csatornán és milyen jel-zaj viszony



4.4. ábra. Példa egy Tanner gráfra Forrás: [25]

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

4.5. ábra. A Tanner gráfhoz tartozó paritás ellenőrző **H** mátrix Forrás: [25]

mellett lesz használatban). Az LDPC kódok dekódolásához a legtöbbet használt megoldás az *SPA - sum-product algorithm* és ennek különböző fajtái. Mielőtt az algoritmus működését bemutatnám, bevezetem a *message passing* típusú dekódolás ötletét, ami az SPA alapját adja. A leírást és a használt jelölésrendszert [25]-t felhasználva készítettem.

A *message passing* során sok kisebb méretű dekódoló működik együtt, hogy végrehajtsák a dekódolás folyamatát. Mindegyiknek egy kisebb feladata van, de egy megfelelően kidolgozott üzenetváltási stratégia (időzítés és kapcsolatok) mentén információkat küldenek egymásnak és így születik meg a végső eredmény. Az egyik legfontosabb feladat az ilyen megoldásoknál az egyes, kis dekódolók közötti kapcsolatok megtervezése. Egyáltalán nem mindegy, hogy milyen formában kommunikálnak egymással a dekódolók, hiszen ezen múlik, hogy hatékonyan megy-e végbe a folyamat. Ha például egy kör van jelen a dekódolók kapcsolati gráfjában, akkor ez egyfajta visszacsatolásként funkcionál, hiszen visszaviszi ugyanazt az információt (vagy pontosabb az adott információ által kialakult eredményt) oda, ahonnan az érkezett. Ez viszont hátráltatja az eredmény kialakulását. Természetesen ez nem azt jelenti, hogy egyáltalán nem lehet kör a kapcsolati gráfban, mivel ha a kör mérete akkora, hogy az iterációk során nem járja be az információk áradása, akkor nem okoz ilyen problémát (ha a kör a méretéből fakadóan bejárásra kerül egyszer, akkor ez még mindig jobb eset, mintha többször járnánk be).

Egy másik fontos pont, hogy hogyan kezeljük az információkat, melyeket a szomszédainktól kapunk és hogyan állítsuk össze ezekből és a nálunk lévő információból azt, amit később továbbküldünk az egyes szomszédoknak. A fent említett kör problémája miatt úgy kell eljárnunk, hogy a szomszédunknak küldött információ a következő iterációban ne tartalmazza azt, amit ő az előzőben küldött, hiszen ekkor létrehozunk egy 2 hosszú kört közte és köztünk. Összefoglalva, ha adott az X dekódoló, aki a szomszédjának Y-nak szeretne küldeni információt $I_{X\to Y}$, akkor az a következőképpen fog kinézni:

$$I_{X\to Y} = \sum_{Z\in N(X)} I_{Z\to X} - I_{Y\to X} + I_X = \sum_{Z\in N(X)-\{Y\}} I_{Z\to X} + I_X,$$

ahol $N(X)$ a szomszédokat jelöli. Ahogy említettem a kapcsolatok kialakítása mellett az üzenetek küldésének időzítése is fontos. Ez adja meg, hogy hányszor küldjenek az egyes dekódolók a szomszédjainak üzeneteket. Az egyik legegyszerűbb megoldás, ha mindig küldünk az összes szomszédnak függetlenül az iteráció számától és más faktoroktól üzenetet.

Ahhoz, hogy megadjuk az LDPC kódokhoz szükséges dekódolót már csak azt kell definiálni, hogy milyen információt adnak át egymásnak az egyes dekódolók,

hogy milyen kapcsolati háló van közöttük valamint, hogy mikor kell megállnia a folyamatnak. A LDPC kódok reprezentációjánál használt Tanner-gráf tökéletesen megfelel a kapcsolati hálónak és a csomópontokat tekinthetjük egyszerű dekódolóknak is. Az SPA esetében a dekódolók az *a posteriori* valószínűségét fogják számolni annak, hogy az átküldött kódszó adott bitpozíciójában 1-es szerepelt, feltéve, hogy milyen érték érkezett abban a pozícióban, azaz a következő valószínűségre vagyunk kíváncsiak:

$$P(v_j = 1 \mid \mathbf{y}),$$

ahol $\mathbf{y}$ a kapott zajos kódszó, v_j pedig a j-edik bit az elküldött kódszóban. Ezenkívül még szükségünk van a *likelihood ratio*-ra és ennek logaritmusára is (LLR):

$$l(v_j \mid \mathbf{y}) = \frac{P(v_j = 0 \mid \mathbf{y})}{P(v_j = 1 \mid \mathbf{y})} \text{ és } L(v_j \mid \mathbf{y}) = \log(\frac{P(v_j = 0 \mid \mathbf{y})}{P(v_j = 1 \mid \mathbf{y})}).$$

Az LLR információt küldik el egymásnak a csomópontok azt a szabályt felhasználva, amiről korábban írtam. Mivel egy páros gráfról van szó, ezért ellenőrző csomópont csak változó csomópontnak küldhet információt és fordítva. A csomópontok ezután felhasználva a saját tudásukat és a szomszédoktól kapott információkat újraszámolják az LLR értékeket és elküldik a szomszédoknak úgy, hogy kihagyják a tőle kapott információt. Ez egy iterációnak feleltethető meg. Ha az iterációk maximális számát elértük, akkor az LLR-ek alapján az SPA kiadja azt a kódszót, ami szerinte a legnagyobb valószínűséggel került elküldésre. A dekódolási folyamat az iterációk maximális számának elérése előtt is megállhat, hiszen tudjuk, hogy egy helyes kódszót a paritásellenőrző mátrix-szal megszorozva a csupa nulla vektort kapjuk, így ezt használhatjuk megállási feltételnek.

Arról még nem esett szó, hogy a dekódoló bemenetét mivel kell inicializálni. Nem magát a zajos kódszót kell megadni, hanem az LLR-eket az egyes bitpozíciókban. Ez az érték viszont a csatorna típusától függ. A mi rendszerünknél érdekes BIAWGN csatorna esetében a következőképpen számíthatóak ki az LLR-ek: Legyen $x_j = (-1)^{v_j}$ a j-edik átküldött bináris érték, amire rárakódik n_j, ami egy független normál eloszlásból származik $\mathcal{N}(0, \sigma^2)$. A fogadónál tehát az $y_j = x_j + n_j$ érték jelenik meg. Ezekből az *a posteriori* valószínűség:

$$Pr(x_j = x \mid y_j) = [1 + \exp(-2y_j x/\sigma^2)].$$

Ezt felhasználva az LLR pedig:

$$L(v_j \mid y_j) = 2y_j/\sigma^2.$$

Ebből is látható, hogy a zaj szórásnégyzetére rendelkezni kell valamilyen becsléssel a dekódolás megkezdése előtt. Ezt a paraméterbecslés során tudják megtenni a felek.

4.2.2. Az LDPC kódok csoportosítása és megadása

Az LDPC kódokat két csoportra lehet osztani azon tulajdonságuk alapján, hogy milyenek lehetnek egymáshoz képest az egyes csúcsok fokszámai a gráfban. Reguláris LDPC kódról beszélünk, ha adott, hogy a változó és az ellenőrző csomópontoknak mi a fokszáma. Irreguláris LDPC kódról beszélünk, ha a csomópontok fokszámai változhatnak egymáshoz képest. A irreguláris kódokkal többfajta konstrukció érhető el, ezért a gyakorlatban ezeket használják. Ahhoz, hogy leírjunk egy irreguláris LDPC kódot meg kell adni az úgynevezett fokszámeloszlás polinomot a változó és az ellenőrző csomópontokra. Ebben a polinomban meg kell adni, hogy adott fokszámú csúcsok (d fokúakat az x^d-en jelöli) hányadrészét teszik ki az összes csúcsnak (ω_d a változó és ψ_d az ellenőrző csomópontokhoz). Természetesen a λ_d-k és a ψ_d-k összege 1 kell hogy legyen.

$$\Omega(x) = \sum_d \omega_d \cdot x^d \text{ és } \Psi(x) = \sum_d \psi_d \cdot x^d$$

A megfelelő eloszlást különböző paraméterek mentén a *differential evolution* segítségével lehet megtalálni. Az így kapott eloszlást pedig a *density evolution* eljárással elemezni és megfigyelni olyan tekintetben, hogy a kapott eloszlás alapján létrehozott kód adott környezetben hogyan fog várhatóan viselkedni. A megfelelő eloszlás megtalálásához nagy szakértelem és hosszú idő szükséges.

4.3. A választott protokoll működése

A protokoll azután kapcsolódik be a klasszikus kommunikációba, miután a két fél egyeztette, hogy Bob melyik kvadratúrákat választotta. Alice az elküldött adatokat 0 várható értékű és adott szórású normális eloszlás szerint választotta, így ekkor Alice és Bob rendelkezik ugyanazon kvadratúrákhoz tartozó adatokkal, melyeket normálnak. Ezen adatok - bár egy kvadratúrához tartoznak -, de a kvantumos csatorna, a mérési eszközök zaja, illetve egy lehetséges támadó jelenléte miatt eltérnek, viszont ennek ellenére korrelálnak egymással. Alice-nál található a $X \sim N(0,1)$-ből képzett vektor, amire a kvantumos csatornán a $Z \sim N(0, \sigma_Z^2)$ zaj került, így Bob-nál az $Y = X + Z$ található, azaz $Y \sim N(0, 1 + \sigma_Z^2)$. X és Y hossza is n. Bob generál egy véletlen n hosszú bitsorozatot S-t, melyhez előállít egy LDPC kódszót C-t. A

két fél előre megbeszélte az LDPC kódhoz tartozó paritásellenőrző mátrixot. Ezt a kódszót és az általa mért adatokat felhasználva létrehozza az M üzenet:

$$M_i = (-1)^{C_i} Y_i \text{ , ahol } i = 1, 2, \ldots, n$$

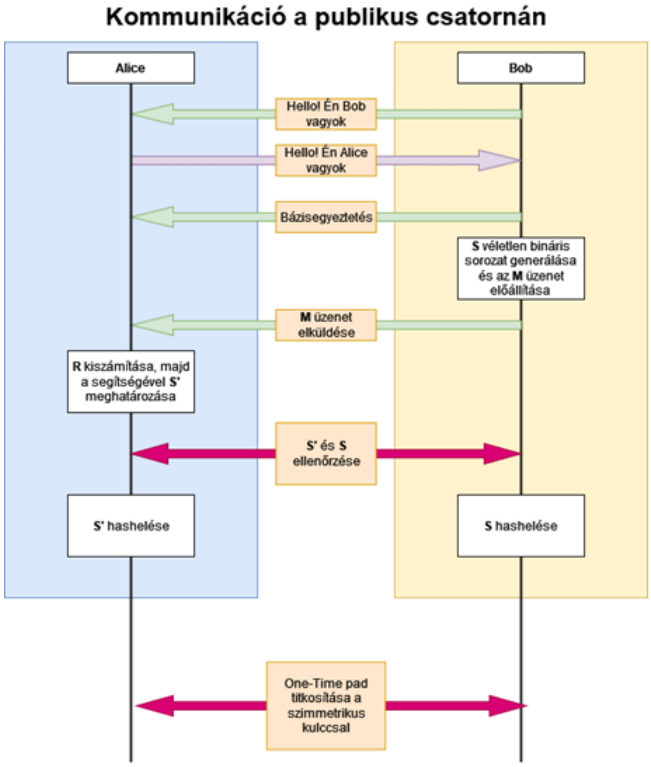
és azt elküldi Alice-nek. Alice az M üzenetből a saját adatait felhasználva kiszámolja R-t:

$$R_i = \frac{M_i}{X_i} = \frac{(-1)^{C_i} Y_i}{X_i} = \frac{(-1)^{C_i}(X_i + Z_i)}{X_i} = (-1)^{C_i} + (-1)^{C_i} \frac{Z_i}{X_i}$$

Ez pedig Bob C kódszójának egy zajjal torzított változata és megfigyelhető, hogy a két fél között létrejött egy virtuális BIAWGNC (*Binary Input Additive Gaussian Noise Channel*). Viszont Alice ismeri az X_i értékeket minden i-re, ezért ismeri annak normáját is. Emiatt a csatorna zaja is ismert lesz: Gauss-os zaj 0 várható értékkel és $\sigma_{N_i}^2 = \sigma_Z^2 / |X_i|^2$ varianciával. Alice R-ből egy LDPC dekódoló segítségével előállítja S'-t, ami nagy valószínűséggel megegyezik a Bob által létrehozott S-sel. A csatornában jelenlévő nagy mértékű zaj (pl. egy támadó miatt) torzíthatja R-t olyannyira, hogy a dekódoló nem képes tökéletesen visszaállítani C-t és ezáltal magát S-t. Ahhoz, hogy ezt ellenőrizni tudják a *confirmation* lépésben összevetik az Alice által kiszámolt S'-t és a Bob-nál lévő S-t. A [24] cikkben megtalálható ennek a megoldásnak a magasabb (2, 4 és 8) dimenziókba való átalakítása.

4.4. Az elkészült program bemutatása

Az előbbi fejezetben ismertetett protokoll implementálásához egy programot készítettem, amit ebben a fejezetben fogok bemutatni. A létrehozott kulcs felhasználására pedig egy olyan chat-alkalmazás készült, aminek a segítéségével Alice és Bob titkosítottan tud szöveges üzeneteket és képeket küldeni egymásnak, melyeket a létrehozott kulccsal titkosítanak. Először ismertetem azon részeit a programnak, melyeket építőkövekként használ fel a program többi része (ilyen pl. az LDPC kódolást kezelő függvények) és utána a grafikus felületet is magába foglaló, a program belépési pontját adó részt mutatom be. Az egyes elemeknél kitérek a használt megoldásokra és könyvtárakra, ezenkívül ismertetem, hogy a fejlesztés során milyen megoldásokat próbáltam ki és melyek voltak azok, amiken változtatni kellett az előzetes tesztelések alapján.



4.6. ábra. Kommunikáció a publikus csatornán Forrás: Saját ábra

4.4.1. A LDPC kódok kezelése

Az első blokk, amit be fogok mutatni az az LDPC kódokkal foglalkozó programrészlet. Ahogy azt egy korábbi fejezetben is említettem az LDPC kódoknál a kódolás egy hatékonyan megoldható feladat, de a dekódolás már egy komplexebb algoritmust kíván, amelynek hatékony implementálásétól nagyban függ a teljes folyamat sebessége. Ahhoz, hogy kezelni tudjam az LDPC kódokat két lehetőségem volt. Az egyik, hogy egy saját kódoló és dekódoló programot írok, amely olyan programnyelvben vagy hardveren készül, ahol a sebességre tudok koncentrálni (ilyen lehet a programnyelvek esetében a C/C++ hardveres esetben pedig egy GPU és a hozzá tartozó megfelelő programnyelv). Ezt az utat azért nem tartottam jó megoldásnak az esetemben, mivel az LDPC kóddal kapcsolatos programrész adja a szűk keresztmetszetet, ezért az implementálás minőségétől nagyban függ az elérhető teljesítmény és biztos szerettem volna lenni azzal kapcsolatban, hogy a dekódolás megfelelőn és hatékonyan működik. Ezenkívül pedig az elsődleges célom az volt, hogy a klasszikus lépések jó ütemben elkészüljenek és megbízhatóan fel tudjuk használni a készülő rendszer tesztelése során. A dolgozat végén található részben, ahol a lehetséges továbbfejlesztésekkel és javításokkal foglalkozom, ki fogok térni természetesen arra, hogy a LDPC kezelésén mit lehet javítani.

Ezen blokkal kapcsolatban az első probléma - amivel rögtön szembesültem - a parításellenőrző mátrixhoz kapcsolódott, mivel mind [24], mind [20] ahhoz hogy nagy távolság és alacsony jel-zaj viszony mellett is működőképes rendszert hozzon létre, olyan LDPC kódokat használt, melyek egyrészt alacsony kódrátájúak ([24] esetében akár 0.02), másrészt pedig nagyon nagy méretűek (a mátrixok 10^6 darab sort is tartalmazhatnak) voltak. Ezt azt vonja maga után, hogy a dekódolásnak nemcsak hogy hatékonyan kell történnie, de olyan hardveren kell végezni, ami képes kihasználni az esetleges párhuzamosítási megoldásokat is. Mindkettő munkában a kutatók GPU-kat használtak az LDPC kódok kezeléséhez, ez viszont számunkra nem volt lehetőség, csak egy CPU-t és egy integrált videókártyát tartalmazó rendszert tudtunk használni. Ezen okok miatt ekkora mátrixokat nem tudtunk kezelni, mivel az jelentősen lelassította volna a klasszikus kommunikáció lefutását.

Miután eldöntöttem, hogy egy kisebb méretű mátrixot fogunk tudni csak kezelni már nem volt lehetőséges, hogy a hivatkozott forrásokban ismertetett és fokszámeloszlás polinomokkal megadott mátrixokat használjuk fel. Az LDPC kódokhoz szükséges parításellenőrző mátrix létrehozására több módszer is ismert. Ilyenek a Gallager-féle A és B módszer vagy a *Progressigve Edge Growth - PEG* algoritmus, ahol felhasználhatjuk a fokszámeloszlás polinomot is. Viszont ahhoz, hogy ezen megoldások segítségével adott csatornán jó tulajdonságokkal rendelkező LDPC kódot hozzunk létre jelentős tapasztalat és tudás szükséges, amelyekkel nem rendelkeztem, ezért megpróbáltam keresni kezelhető méretű, de jó hibajavító képességgel rendelkező és publikusan elérhető LDPC kódokat. Választásom pedig a DVBS-2 szabványra esett, mely több, változó kódrátájú LDPC kódot is biztosít. Természetesen találnom kellett olyan megoldásokat, melyekkel ezek a kódok kezelhetőek.

Először tehát olyan könyvtárakat vagy programokat kerestem, melyek biztosítják az LDPC kódok használatához szükséges funkcionalitást és fel lehet használni őket úgy, hogy kommunikáljanak egy másik programmal is. Az első ilyen könyvtár, amit megvizsgáltam a pyldpc Python csomag volt, mely függvényeket biztosít paritásellenőrző mátrixok létrehozásához (például a Gallager-féle A módszerrel) és természetesen a kódolás és dekódolás lépésekhez is. Elsőként ezt a könyvtárat felhasználva próbáltam meg elkészíteni az ismertetett protokoll lépéseit (a bázisegyeztetés utáni üzenetekkel). Sajnos gyorsan szembesülnöm kellett vele, hogy mivel tisztán pythonos implementációról van szó, ezért a sebessége nem feltétlenül elfogadható nagy mátrix esetén. Azt tapasztaltam, hogy a dekódolást végző függvény a pszeudokód implementálását tartalmazza, így nem a legoptimálisabb megoldásokat vonultatja fel. A tesztelések alapján már egy 20x40-es mátrix esetén is a dekódolás több másodpercbe telt, ezért ezt a könyvtárat elvetettem.

A következő megközelítésem a DVBS-2 irányából indult. Megpróbáltam olyan szoftverrádiók létrehozásához használható programot vagy könyvtárat keresni, ami támogatja ezt a szabványt, hiszen ekkor nagy valószínűséggel az abban található LDPC kódokat is támogatnia kell. Így találtam a GNURadio nevű nyílt forráskódú programot, amely könyvtárként is használható. A GNURadio segítségével létre lehet hozni egy szoftveres rádiót különböző modulok összekapcsolásával (ilyen például egy forrás, kódoló-dekódoló vagy akár egy amplitúdómodulátor). Nagy szerencse volt, hogy bár a könyvtár elsődlegesen C++-szal használható, de létezik pythonos binding, aminek a segítségével könnyen létre tudtam hozni a modulokat és a köztük lévő kapcsolatokat úgy, hogy a fontos lépéseket a C++-ban megírt kód hatja végre. Az LDPC kódolóhoz és dekódolóhoz is léteztek a megfelelő modulok, vektorforrásokat felhasználva az S és a C bináris vektorok kezelése is egyszerű volt. A szükséges átalakításokat (M és R létrehozása) szorzásokat végrehajtó modulokkal tudtam megalkotni. Sajnálatos módon annak ellenére, hogy a DVBS-2 szabvány támogatása hivatalosan a GNURadio része, a szabványhoz tartozó LDPC kódok ebbe még nem tartoznak bele, így nem tudtam használni az ott elérhető mátrixokat. Szerencsére a GNURadio is biztosít megoldást LDPC kódhoz tartozó mátrix generálására, sőt a projekthez tartozó *repository* még példákat is tartalmaz az így generált mátrixokból és ezeket fel is tudtam használni. Az ezzel a könyvtárral elkészült implementációban a mátrix méretei 902 és 1800 voltak. A tesztelése során a kulcsot sikeresen létre tudtam ugyan hozni, de a dekódolás bizonyult a szűk keresztmetszetnek, mivel ez a lépés akár 2-4 másodpercig is tarthatott, ami bár gyorsabb volt mint a pyldpc esetében, de még így is nagyon lassúnak bizonyult, így ezt a megoldást is el kellett vetnem.

Az LDPC kódok kezeléséhez végül az AFF3CT keretrendszert és könyvtárat használtam [28]. Ez egy hibajavító kódokra specializálódott keretrendszer, ami főleg arra használható, hogy a különböző kódokat teszteljük változó paraméterek mellett (ilyen lehet pl. a csatorna vagy a moduláció típusa). Elsődlegesen így is használható, parancssorból indítva és megadva a szükséges argumentumokat, majd a kiválasztott hibajavító kód tesztelése történik különböző csatornatulajdonságok mellett. Szerencsére az AFF3CT-et könyvtárként is lehet használni. Elsőként a Python binding-okat próbáltam meg beépíteni a programba, de ezek fejlesztése még nem áll olyan szinten, hogy a könyvtár minden egyes funkcióját el lehessen érni velük. Így végül azt a megoldást választottam, hogy a könyvtár által használt nyelvben, ami C++, készítem el a kódolást és a dekódolást végző függvényeket és ezeket hívom meg a Python programból a megfelelő könyvtárak segítségével. Az AFF3CT olyan szempontból is jó választásnak bizonyult, hogy egy jól optimalizált LDPC dekódoló implementá-

ciót tartalmaz és a DVBS-2 szabvány által támogatott LDPC kódokat is magába foglalja, így nem kell külön létrehozni a mátrixokat hozzá.

A C++-ban megírt program az AFF3CT-ben elérhető modulokat használja, amelyeket nagyon hasonlóan lehet egymáshoz kapcsolni, mint a GNURadio esetében. A legfontosabb modulokat és eszközöket a 4.1. programrészlet mutatja. Látható, hogy van egy kódolónk és dekódolónk, illetve a DVBS-2 és az LDPC kódok kezeléséhez szükséges objektumok és eszközök is megjelennek.

```
struct modules
{
  std::unique_ptr<module::Encoder_LDPC_DVBS2<>> encoder;
  std::unique_ptr<module::Decoder_SIHO<>> decoder;
};

struct utils
{
  std::unique_ptr<tools::dvbs2_values>      dvbs2_vals;
  tools::Sparse_matrix              H;
  tools::LDPC_matrix_handler::Positions_vector  info_bits_pos;
};
```

4.1. lista. Az LDPC kódok kezeléshez használt modulok és eszközök

Ezeket a modulokat természetesen inicializálni kell, amelyet `init_modules` függvény végez és a 4.2. részlet mutat. Ahogy látható, itt kerül felhasználásra az információs bitek száma K és a kódszavak hossza N is. Ezek jelölik ki, hogy melyik LDPC kódot fogjuk használni. Utána kiválasztásra kerül a megfelelő dekódoló implementáció és a hozzá tartozó kódoló is.

```
void init_modules(const params &p, modules &m, utils &u) {
  u.dvbs2_vals  = tools::build_dvbs2(p.K, p.N);
  u.H       = tools::build_H(*u.dvbs2_vals);

  const auto max_CN_degree = (unsigned int)(u.H.get_cols_max_degree());

  u.info_bits_pos.resize(p.K);
  std::iota(u.info_bits_pos.begin(), u.info_bits_pos.end(), 0);

  module::Decoder_SIHO<B,Q>* modulePtr = NULL;
  modulePtr = (module::Decoder_SIHO<>*) new module::Decoder_LDPC_BP_flooding<B,Q,tools::
   Update_rule_OMS<Q>>(p.K, p.N, p.n_ite, u.H, u.info_bits_pos, tools::Update_rule_OMS <Q>((Q)p.
   offset), p.enable_syndrome, p.syndrome_depth);

  m.encoder = std::unique_ptr<module::Encoder_LDPC_DVBS2<>>(new module::Encoder_LDPC_DVBS2<>(*u.
   dvbs2_vals));
  m.decoder = std::unique_ptr<module::Decoder_SIHO<>>(modulePtr);
};
```

4.2. lista. A modulok inicilizációja

A kódolás maga az ehhez tartozó modul segítségével nagyon egyszerű, csupán csak egy függvényhívás, de az ezt tartalmazó függvényt úgy kell létrehozni, hogy kívülről meghívható legyen (ezért szükséges az `extern "C"` a fejléc elején). Beme-

netként a kódolandó információ és a használni kívánt K és N értékek érkeznek, majd a kódszó előállítása után visszaadásra kerül a hívónak. A teljes függvényt a 4.3. részlet tartalmazza.

```
extern "C" int* encode_bits(int bits_to_encode[], int k, int n) {
  params p;
  modules m;
  buffers b;
  utils u;

  p.K = k;
  p.N = n;

  init_params (p   );
  init_modules(p, m, u);
  init_buffers(p, b);

  int* encoded_bits= new int[p.N * count];

  std::vector<int> bits(bits_to_encode + p.K * i, bits_to_encode + p.K * (i+1));
  m.encoder->encode(bits, b.enc_bits);
  std::copy(b.enc_bits.begin(), b.enc_bits.end(), encoded_bits + p.N * i);

  return encoded_bits;
}
```

4.3. lista. A kódolást megvalósító függvény

A dekódolás folyamata nagyon hasonló a kódoláséhoz, ahogy az a 4.4. programrészletből is látszik. Ugyanúgy figyelni kell arra, hogy a függvény kívülről is meghívható legyen és hogy a hívó meg tudja adni, hogy melyik LDPC kódot szeretné használni.

```
extern "C" int* decode_bits(float noisy_bits[], int k, int n) {
  params p;
  modules m;
  buffers b;
  utils u;

  p.K = k;
  p.N = n;

  init_params (p   );
  init_modules(p, m, u);
  init_buffers(p, b);

  int* decoded_bits= new int[p.K * count];
  std::vector<float> noisy(noisy_bits + p.N * i, noisy_bits + p.N * (i+1));
  m.decoder->decode_siho(noisy, b.dec_bits);
  std::copy(b.dec_bits.begin(), b.dec_bits.end(), decoded_bits + p.K * i);

  return decoded_bits;
}
```

4.4. lista. A dekódolást megvalósító függvény

Miután a program elkészült egy *shared library*-t hoztam létre belőle, amit később a `ctypes` Python könyvtár segítségével tudtam meghívni a saját programomban. Mivel a C++-ban megírt program csak a kódolást és a dekódolást valósítja meg, ezért a fennmaradó lépéseket már Python-ban készítettem el.

Az első, amit szeretnék bemutatni a Bob oldalán végbemenő lépések, amit a 4.5. részlet mutat be. A függvény megkapja `y`-t, ami a Bob-nál lévő mérések eredménye a bázisegyeztetés és az adatok normálása után. Első lépésként véletlenszerűen sorsolunk egy `s` bináris vektort, melynek a hossza `K` lesz, aminek az értéke előzetesen beállításra kerül. Ezután a `cytpes` könyvtár segítségével a korábban már betöltött *shared library* megfelelő függvénye kerül meghívásra úgy, hogy előtte a paraméterek típusa a megfelelő értékre lesz állítva. A visszatérési értéknél is hasonlóan járunk el és a segítségével elkészítjük az Alice-nak küldendő `m` üzenetet.

```
def encodeRandomS(self, y):
  y = y.astype(ctypes.c_float)
  s = np.random.choice([0, 1], size=(self.K, ), p=[0.5, 0.5])

  sAsCTypesArray = s.astype(ctypes.c_int)
  sAsCTypesList = sAsCTypesArray.tolist()


  sAsFunctionArgument = (ctypes.c_int* (self.K))(*sAsCTypesList)
  sAsFunctionArgumentPointer = ctypes.cast(sAsFunctionArgument, ctypes.POINTER(ctypes.c_int))
  c = self.libC.encode_func(sAsFunctionArgumentPointer, self.K, self.N)
  cAsList = c[:self.N]
  cAsFloatArray = np.array(cAsList, dtype=ctypes.c_float)
  self.libC.delete_func(c)

  m = ((-1) ** cAsFloatArray) * y

  return sAsCTypesArray, m
```

4.5. lista. A Bob oldalán történő LDPC-t használó lépések

Az Alice oldalán lejátszódó és az LDPC kódokkal kapcsolatos lépéseket mutatja a 4.6. programrészlet. Ahhoz, hogy megkapjuk S'-ot szükségünk van természetesen a Bob által küldött `m` üzenetre, de a paraméterbecslés során a jel-zaj viszonyra kapott számítás eredményére is (`theta_z`). Ezeket felhasználva létrehozzuk `r`-t és a [24]-ben és a [25]-ben írtak alapján a dekódoló bemenetére a csatorna típusa (BIAGWNC) és a zaj varianciájának ismeretében számolt *likelihood*okat adjuk. A visszakapott érték egy lehetséges kódszó lesz, amit visszaadunk a hívónak.

```
def decodeSFromGivenM(self, receivedM: np.ndarray, x, theta_z: float):
  x = x.astype(ctypes.c_float)
  r = receivedM / x

  channelVariance = theta_z / (np.abs(x)) # As per Channel Codes: Classic and Modern (2009) page
   219.

  r = 2 * (r / channelVariance)
```

```
    rAsFloatList = r.tolist()
    rAsDecodeArgument = (ctypes.c_float*(self.N))(*rAsFloatList)
    rAsDecodeArgumentPointer = ctypes.cast(rAsDecodeArgument, ctypes.POINTER(ctypes.c_float))
    sAsCTypesPointer = self.libC.decode_func(rAsDecodeArgumentPointer, self.K, self.N)
    sAsPythonList = sAsCTypesPointer[:self.K]
    sAsIntArray = np.array(sAsPythonList, dtype=ctypes.c_int)
    self.libC.delete_func(sAsCTypesPointer)

    return sAsIntArray
```

4.6. lista. Az Alice oldalán történő LDPC-t használó lépések

4.4.2. A privacy amplification lépése

A *privacy amplification*-t nagyon egyszerűen oldottam meg, felhasználva a `SHA512` algoritmust. A sikeres *confirmation* után előálló kulcsot felbontom adott nagyságú részekre és egyenként lehashelem őket, majd az eredményeket egy közös tömbben tárolom el, hogy később tetszőleges módon fel lehessen használni a titkosítás során. Az ehhez a lépéshez tartozó függvények láthatóak a 4.7. részletben.

```
def bytesStringFromS(S) -> bytes:
  numberOfBytesInS = len(S) // 8
  sAsString = ''.join(map(str, S[:(numberOfBytesInS*8)]))
  resBytes = bytes()
  for i in range(numberOfBytesInS):
    partOfS = sAsString[i*8:(i+1)*8]
    resBytes += int(partOfS, 2).to_bytes(1, 'big')
  return resBytes

def hashGivenBytes(bytesToHash: bytes) -> bytes:
  hasher = SHA512.new(data=bytesToHash)
  return hasher.digest()

def hashSPerPart(S, numberOfParts):
  sParts = np.split(S[:numberOfParts*512], numberOfParts)
  sByteArray = bytearray()
  for i in range(0, len(sParts)):
    sPartByteString = bytesStringFromS(sParts[i])
    sByteArray.extend(hashGivenBytes(sPartByteString))
  return sByteArray
```

4.7. lista. A privacy amplification megvalósítása

4.4.3. Az interneten folyó kommunikációt megvalósító blokk

A program másik fontos része a publikus csatornán történő kommunikáció megoldása volt. Publikus csatornának az internetet választottam, így döntenem kellett, hogy melyik szállítási rétegbéli protokollt fogom használni. A 4.3. fejezetben leírt lépések alapján rögtön arra gondolhatunk, hogy egy peer-to-peer protokollal állunk szemben, hiszen Bob és Alice egyenrangú, nincs semmifajta megkötés arra, hogy va-

lamelyikük szerepe eltérne a másiktól (kommunikációs szempontból). Azonban úgy is meg lehet közelíteni a helyzetet, hogy mivel *direct reconciliation*-ről van szó Bob információt küld Alice-nak, aki ezekre válaszol. Ez a felállás viszont jobban kezelhető a kliens-szerver modellel, így én is ezt használtam. Alice szerverként várja a kapcsolatokat, melyek Bob-tól érkeznek. Ezután Bob üzeneteire Alice válaszokat küld. Ez a megközelítés akár jobban is beleillik egy olyan környezetbe, amikor egy városi QKD rendszerben pl. egy bank központi állomása több fiókkal is kapcsolatban áll csillag elrendezésben. Mivel a kliens-szerver megoldás mellett döntöttem kézenfekvő volt a TCP választása a szállítási rétegben, mert egy adott klienshez tartozó kapcsolat könnyebben kezelhető, mint az UDP-ben, sőt biztosak lehetünk benne, hogy az átküldött adat meg is érkezik a másik félhez. Ez kifejezetten fontos, amikor a pl. a protokollban az M üzenetet küldjük vagy a *confirmation* lépésre kerül sor.

Mivel a programot Python-ban készítettem, így kézenfekvő lehetőség volt, hogy a nyelvhez tartozó `sockets` könyvtárat használva valósítsam meg a kommunikáció alapját, de nem e megoldás mellett döntöttem. A TCP kommunikációnál az UDP-vel ellentétben fontos megfelelően kezelni az adatokra való várakozást, hiszen itt csak egy adatfolyam áll rendelkezésre, ahol nincs explicite jelölve, hogy hol van a küldött adat vége. Ezt a várakozást természetesen meg lehet oldani blokkoló módon, de az aszinkron megközelítés egy sokkal jobban működő megoldáshoz vezet. Ahhoz, hogy a TCP kezelését és az aszinkron megközelítést egybe tudjam kezelni a Twisted könyvtárt használtam, amely segítségével könnyen lehet protokollokat megvalósítani. A Twisted a reaktor tervezési mintát valósítja meg, amelynek központi eleme egy reaktor, aminél regisztrálásra kerülnek a különböző eseménykezelők. A reaktor folyamatosan ellenőrzi, mely események történtek meg és ekkor a megfelelő eseménykezelőt fogja meghívni, melynek futása után a végrehajtás visszatér a reaktorhoz.

A Twisted másik nagy előnye, hogy kész támogatással rendelkezik a TCP-t használó protokollok kezelésére és lehetőséget biztosít állapotalapú protokollok definiáláshoz is. Ahhoz, hogy a megfelelő típusú protokollt használjuk elég csak leszármaznunk egy speciális osztályból pl. állapotalapú protokoll esetén a `StatefulProtocol` osztályt tudjuk használni. A kliens-szerver modell használatához pedig factory-kat lehet definiálni, melyek a szerver esetén minden új kapcsolathoz létrehoznak egy szerveroldali protokoll objektumot.

Bár a két fél eltérő üzeneteket fog küldeni egymásnak és más állapottal fog rendelkezni, de abban megegyeznek, hogy szükségük van adatok elküldésére oly módon, hogy a TCP folyamban tudni lehessen, mennyi adat érkezik még. Ehhez használható a *framing*, melynek segítségével meg lehet adni, hogy meddig tart egy csomag és milyen részekből áll. A megoldásban egy nagyon egyszerű *framing*-et használtam.

```
class BaseCvQkdProtocol(StatefulProtocol):

  def sendMsg(self, plainMsg: bytes, encrypted: bool, forceSendOK: bool = False):
    numberOfChunksToSend = math.ceil(len(plainMsg) / MAXCHUNKSIZE)
    for i in range(numberOfChunksToSend):
      msgChunk = plainMsg[i * MAXCHUNKSIZE:(i + 1) * MAXCHUNKSIZE]
      msgChunkLength = len(msgChunk).to_bytes(2, 'big')
      if encrypted:
        msgChunk = encryptMsg(msgChunk, self.cipher)
      self.transport.write(msgChunkLength + msgChunk)
    if (numberOfChunksToSend > 1 or forceSendOK):
      self.sendOK()
    if encrypted:
      if plainMsg[0] == 1:
        self.factory.handleMsg("Sent img", data=[plainMsg[1:], str(len(plainMsg)),str(len(self.S))
   , bytearray(self.salsa20Key)])
      elif plainMsg[0] == 0:
        self.factory.handleMsg("Sent msg:", data=[plainMsg[1:].decode(), str(len(plainMsg)),str(
   len(self.S)), bytearray(self.salsa20Key)])

  def sendOK(self):
    self.transport.write(int(0).to_bytes(2, 'big'))

  def closeConnection(self):
    self.transport.loseConnection()
```

4.8. lista. A prokollok közös őse

A küldő a fejlécben elküldi a küldendő adat méretét (a fejléc hossza fixen 2 bájt hosszú) és ezután érkezik az adat. Így a másik oldalon a feldolgozás mindig az első két bájt kiolvasásával kezdődik, utána pedig az abban leírt mennyiségű adatot várja be a fogadó fél. A küldés egységes kezelését a `BaseCvQkdProtocol` függvényei végzik. Itt feldarabolásra kerül az elküldendő adat egy megadott méret szerint és attól függően, hogy titkosított-e a kommunikáció (ez majd a chat esetében lesz fontos) a titkosítás is megtörténik. Legvégül pedig a hossz alapján a fejléc is generálásra kerül és így a csomag elküldhető. Ezeket a lépéseket lehet látni a 4.8. részletben. Ha több részben történik az adat átküldése, akkor a küldőnek valahogyan jeleznie kell a másik fél felé, hogy nem jön több részlet. Ezt úgy oldottam meg, hogy lezárásként egy nulla hosszú csomagot küldünk.

Ezután viszont a két oldal kommunikációját implementáló rész különválik, hiszen más feladatai vannak az egyes szereplőknek. A Twisted-ben az állapotok között könnyen lehet váltani. Az egyes állapotok egy függvényként hozhatóak létre és a visszatérési értékükben kell megadni a következő állapotot és hogy mennyi adat beérkezése után kell átlépni oda (az adat mérete bájtban van megadva). A protokoll első lépése az autentikálás és így az implementációban is ez az üzenetváltás játszódik le először. Mivel az elsődleges cél a program készítése során az volt, hogy megfelelően együtt tudjon működni a rendszerrel és képes legyen létrehozni a kulcsot a nyers adatból, azaz implementálja a klasszikus csatornán végzett kommunikációt, így az autentikáció és az adatok integritásának védelme nem tartozik a programba. Ezen megoldásokat a tényleges felhasználás szempontjából kell megtervezni és a kiválasz-

```
def generateSandCalculateM(self):
  self.S, self.M = self.LDPC_Util.encodeRandomS(y=self.factory.y)
  mAsBytes = self.M.tobytes()
  self.sendMsg(mAsBytes, encrypted=False)
```

4.9. lista. Bob elküldi M-et

```
def calculateRandS(self, M):
  receivedM = np.frombuffer(M, dtype=ctypes.c_float)
  sPrime = self.LDPC_Util.decodeSFromGivenM(receivedM, x=self.factory.x,
  self.theta_z)
  self.sPrimeBytes = sPrime.tobytes()
  crcObj = crcmod.predefined.PredefinedCrc("crc-32")
  crcObj.update(self.sPrimeBytes)
  sPrimeCRC = crcObj.digest()
  self.sendMsg(sPrimeCRC, encrypted=False, forceSendOK=True)
```

4.10. lista. S' kiszámolása és CRC számolás

tani, hiszen itt olyan kérdések merülnek fel, mint pl. autentikálásnál elfogadható-e, ha publikus kulcsokat használunk vagy szigorúan csak PQC (*Post Quantum Cryography*) eljárásokat vehetünk figyelembe, melyek bizonyítottan képesek ellenállni kvantumszámítógép általi támadásnak. Ezen okok miatt a programban csak egy nagyon egyszerű ellenőrzés van a másik fél kilétével kapcsolatban. Ezt a folyamatot Bob kezdeményezi rögtön azután, hogy létrejött a TCP kapcsolat. Ezt feldolgozza Alice és ellenőrzi, hogy megfelelő-e az üzenet, majd visszaküldi a hozzá tartozó információt, amit Bob tud majd ellenőrizni.

Ha sikeresen lezajlott az ”autentikálás” a felek megkezdik a 4.3. fejezetben megadott protokoll lépéseinek végrehajtását. Bob először generálja S-t és a hozzá tartozó M üzenet felhasználva a mérési eredményeit és ezt elküldi Alice-nak. Ezeket a lépéseket a 4.9. részlet mutatja.

Alice ezt követő lépéseit a 4.10. programrészlet mutatja. Először bevárja, hogy megérkezzen a teljes M-et tartalmazó üzenet és elkezdi annak feldolgozását. A korábban bemutatott függvények segítségével megkapja S'-t és hogy a két fél össze tudja hasonlítani a nyers kulcsot elkészíti ennek CRC-jét és elküldi Bob-nak.

Bob bevárja az S'-höz tartozó CRC értéket és ellenőrzi, hogy megegyezik-e a nála lévő értékkel. Ha a kettő megegyezik, akkor erről értesíti Alice-t és elvégzi a nyers kulcs adott blokkonkénti hashelését a korábban bemutatott függvény segítségével. Ezután a végleges kulcs felhasználásával létrehoz egy Salsa20 folyamkódolót, melynek a kulcsa a végleges kulcs első 256 bitje lesz. Az így kapott Salsa20 kódolónak szükség van a *nonce*-jára, hiszen enélkül nem tudná dekódolni az adatokat a másik fél, ezért ezt elküldi Alice-nak.

Természetesen felvetődik a kérdés, hogy miért nem a One-Time Pad titkosítást használjuk a későbbi chat üzenetek elküldése során. Azért döntöttünk végül a Salsa20 mellett, mert a One-Time Pad-nek ugyanannyi kulcsbitre van szüksége, mint az

```
def readEncryptedMsgFromClient(self, data: bytes):
  receivedEncryptedMsg = data
  plainMsg = decryptMsg(receivedEncryptedMsg, self.cipher)
  if plainMsg[0] == 1: # Kép típusa
    self.factory.handleMsg("Got image", data=[str(len(plainMsg)), str(len(self.S)),plainMsg[1:],
   bytearray(self.salsa20Key)])
  elif plainMsg[0] == 0: # Szöveg típusa
    self.factory.handleMsg("Got msg:", data=[str(len(plainMsg)),str(len(self.S)),plainMsg[1:].
   decode(), bytearray(self.salsa20Key)])
  elif plainMsg[0] == 3: # Rekey kezdetét őjelz típus
    self.factory.handleMsg("Rekey")
    self.currentMsg = bytearray()
    rekey_ack_packet = bytearray(int(4).to_bytes(1, "big"))
    rekey_ack_packet.extend(bytes("Rekey", "utf-8"))
    self.sendMsg(bytes(rekey_ack_packet), True, True)
    self.rekeyACKed = True
  elif plainMsg[0] == 4: # Rekey nyugtázásra szolgáló típus
    self.rekeyACKed = True
```

4.11. lista. Az különböző üzenetítpusok kezelése Alice oldalán

elküldendő adat és emiatt ha egy nagyobb képet szeretnénk elküldeni, akkor egy hosszú kulcsra van szükségünk. A tesztelés során ki fogok rá térni, de a rendszer még nem tud olyan állapotban működni, hogy folyamatosan hozzon létre kulcsokat, így csak eseti jelleggel tudunk kulcsot generálni. Ez viszont azt jelenti, hogy a One-Time Pad használatához szükség nagyobb mennyiségű kulcsmérettel nem tudtam tervezni. Ezen indok miatt választottam a Salsa20 folyamkódolót, amelyhez egy 256 bites kulcs szükséges és nagy mennyiségű bitet lehet vele titkosítani. Mivel a protokoll egyszeri lefutása alatt ennél jóval nagyobb a végleges kulcs mérete, ezért egy *rekey* mechanizmust is beépítettem, aminek a segítségével a végleges kulcsot 256 bitenként fel tudják használni arra, hogy bármelyik fél kérésére új Salsa20 kódolót indítanak a következő 256 bittel (amennyiben ez lehetséges).

Ahogy megkapja a *nonce*-t Alice, fel tudja azt használni, hogy létrehozza ő is a Salsa20 kódolóját. Ehhez pedig szükséges még a nyers kulcs hashelt változatának első 256 bitje. Természetesen ezt csak akkor teheti meg, ha Bob tudatta vele, hogy a két nyers kulcs (S és S') megegyezik.

Ezután mindkét fél rendelkezik a végleges kulccsal, amivel létrehoztak egy megegyező *nonce*-szal rendelkező Salsa20 kódolót, így már tudnak titkosítottan kommunikálni egymással. Innentől már a chat üzenetek kezelésén van a hangsúly. Minden üzenet tartalmaz egy típusazonosítót, ami azt hivatott megadni, hogy az üzenetben lévő adat szöveges, kép vagy a *rekey* mechanizmushoz tartozik-e. Ez alapján történik a feldolgozásuk is, amit a 4.11. programrészlet szemléltet. A *rekey* mechanizmus pedig a 4.12-ben látható módon indul a megfelelő típusú üzenet elküldésével, amire egy speciális válasz érkezik a másik féltől.

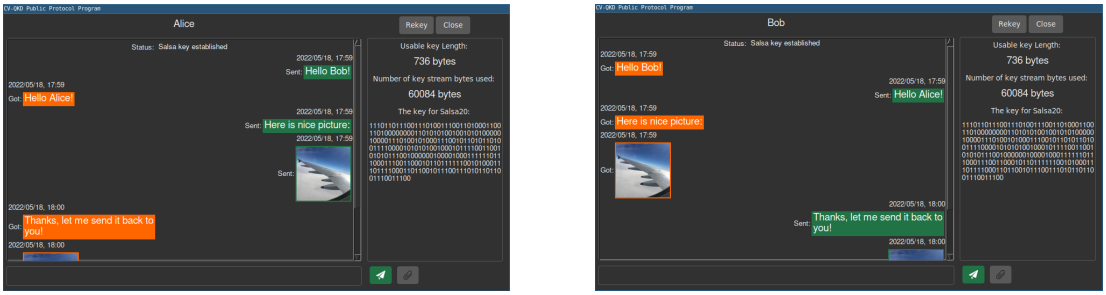
```
def rekey(self):
  rekey_packet = bytearray(int(3).to_bytes(1, "big"))
  rekey_packet.extend(bytes("Rekey", "utf-8"))
  self.sendMsg(bytes(rekey_packet), True, True)
```

4.12. lista. A rekey mechanizmus indítása

4.4.4. A chat-alkalmazás és a grafikus felület

Ahhoz, hogy a létrejött kulcs használatát bemutassam készítettem egy chat-alkalmazást is. Itt a felek arra tudják használni a kvantumosan megosztott kulcsot, hogy titkosítottan kommunikáljanak szöveges üzenetek és képek formájában. A grafikus felületet tartalmazó programfájl a belépési pont, ahol megadhatóak a kapcsolat létrejöttéhez szükséges információk (a hosztnév és a portszám).

A grafikus felületet a Python Tk könyvtárának segítségével hoztam létre. Nagy segítség volt, hogy könnyen integrálható volt a Twisted által használt reaktorba. Az ablakot a program működése közben a 4.7. ábrán lehet látni. A két fél éppen üzeneteket és képéket cserél egymással. A felületen helyet kaptak a chat-alkalmazásokban szokásos gombok és panelek. Egy központi részben az üzenetek láthatóak két oldalra rendezve a feladótól függően. A lenti részben pedig egy szövegdoboz található, ahová az üzenetet lehet írni. Kép csatolásához is elhelyezésre került egy gomb ezen doboz mellett. A felület jobb oldalán pedig a kulcsról és az eddig titkosított adatok mennyiségéről lehet információkat találni. A *rekey* mechanizmust pedig egy gombbal lehet elindítani, ami szintén helyet kapott az ablakon.



4.7. ábra. A chat-alkalmazás működés közben.

5. fejezet

A rendszer tesztelése

Ebben a fejezetben fogom bemutatni a rendszer tesztelése során kapott eredményeket és levonni az ezekből szerzett tapasztalatokat. Először kitérek arra, hogy miben különbözik a 3.2.1. fejezetben bemutatott működéstől a gyakorlatban megvalósított eszköz. Majd bemutatom a tesztelések alatt mért adatokat és a fejezet végén értékelem a kapott eredményeket. Ahhoz, hogy szemléltetni tudjam, hogy milyen változások jelentkeztek a rendszer működésében a hálózati tesztek során a laboratórium környezetben mért eredményeket is röviden áttekintem.

A rendszer összeszerelt állapotban egy standard rack méretű házban kapott helyett. Az optikai eszközök mellett a feldolgozáshoz szükséges készülékek (pl. a PC) és ezek tápellátásukhoz, hűtésükhöz elengedhetetlen berendezések is elhelyezésre kerültek. Ez a megoldás hatással van a rendszer működésére, de erre a tesztek értékelésénél fogok kitérni. A korábban bemutatott ábrán (3.5. ábra), ami a felépítést mutatja, látható, hogy a lézer Alice oldalán a rendszer közvetlen része. A gyakorlati megvalósításban a lézer kikerült az Alice oldalát tartalmazó házból és egy optikai kábellel csatlakozik annak bemenetére. A 3.2.2.1. fejezetben bemutatott paraméterbecslés a klasszikus fázis egyik legfontosabb lépése, hiszen az itt kiszámolt és megbecsült értékek alapján tudunk kulcsot létrehozni. Ebben a lépésben az optikai csatornán elküldött adatok egy részét használjuk fel. Ezek az elkészült rendszerben a *decoy* vagy segédimpulzusok és minden második elküldött impulzus ilyen típusú. A klasszikus lépésekhez szükséges adatok kiszámolása mellett ezeket a segédimpulzusokat tudja Alice és Bob felhasználni arra, hogy a köztük lévő optikai összeköttetésen történő utazás során a referencia és a hasznos jel elfordulását megkapják és ennek segítségével korrigálni tudják az adatokat úgy, hogy a Bob méréséből kapott eredmény és az Alice által küldött adat azonos koordináta-rendszerben legyen.

A klasszikus fázis során egy autentikált csatornára van szükség. Ahogy azt a 4.4. fejezetben is említettem, a megvalósult program nem végez biztonságosnak tekinthető autentikálást a kapcsolat létrejötte után és az ezután küldött üzenetek

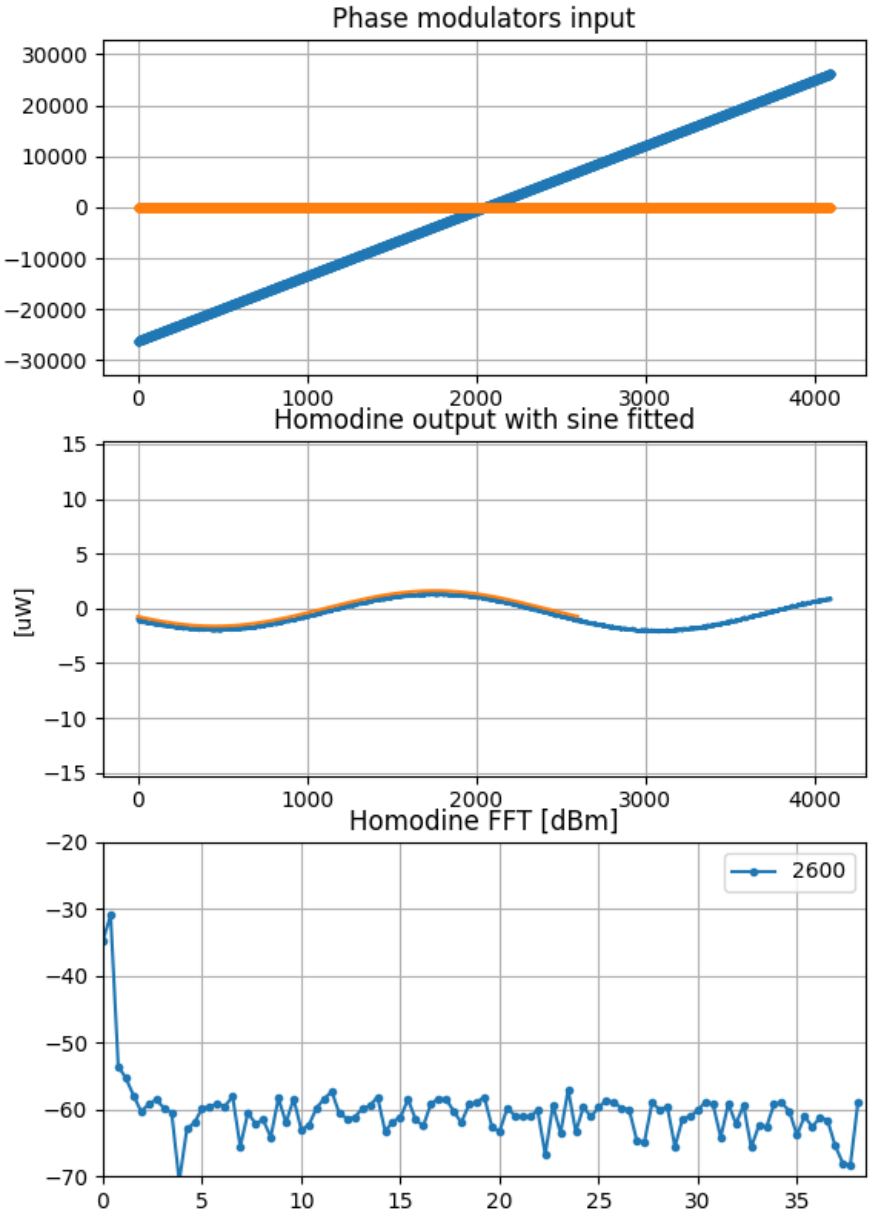
sincsenek autentikálva. A tesztelés során ezeket a problémákat úgy oldottuk meg, hogy a rendszerhez tartozó minden TCP kommunikáció SSH-tunneling megoldással került megvalósításra és publikus kulcsú azonosítást használunk. Ahogy azt korábban említettem a rendszer még nem állít elő folyamatosan kulcsokat, hanem eseti jelleggel képes arra, hogy megpróbálja létrehozni azokat, így a tesztelések során az elérhető kulcsráta meghatározására nem volt lehetőség, az eredményeknél a sikeres vagy sikertelen kulcsgenerálás fog majd csak szerepelni. A tesztelés során egy 6480x16200-as LDPC paritásellenőrző mátrixot használtunk, így az egyes futások során keletkezett kulcs hossza 6480 bit volt. Ahhoz, hogy ez létrejöjjön Alice-nak és Bob-nak 16200 (vagy annál több) mérési adattal kellett rendelkezniük.

Az Alice és Bob oldalát megvalósító eszközök először laboratóriumi környezetben kerültek tesztelésre egy optikai padon összeállított felépítéssel, majd összeszerelés után újra megvizsgáltuk őket. A optikai hálózaton történő tesztelésre két különböző hosszúságú szakaszon került sor. A optikai összeköttetést és a hálózatot a Magyar Telekom biztosította. A két tesztelt szakasz a Budapesti Műszaki Egyetem épületében található labor és a Magyar Telekom kelenföldi központja, illetve a labor és Wigner Fizikai Kutatóközpont (Csillebércen) között volt kialakítva. A szakaszok esetében nem közvetlen optikai szálas kapcsolat lett létrehozva a két végpont között, hanem a Magyar Telekom állomásai által kijelölt kisebb szakaszok összekapcsolásával jöttek létre a tesztelések során használt összeköttetések. Ennek természetesen hatása volt a végpontok között mért csillapításra is, amelyre ki fogok térni az egyes tesztek bemutatása során.

5.1. A rendszer tesztelése laboratóriumi környezetben

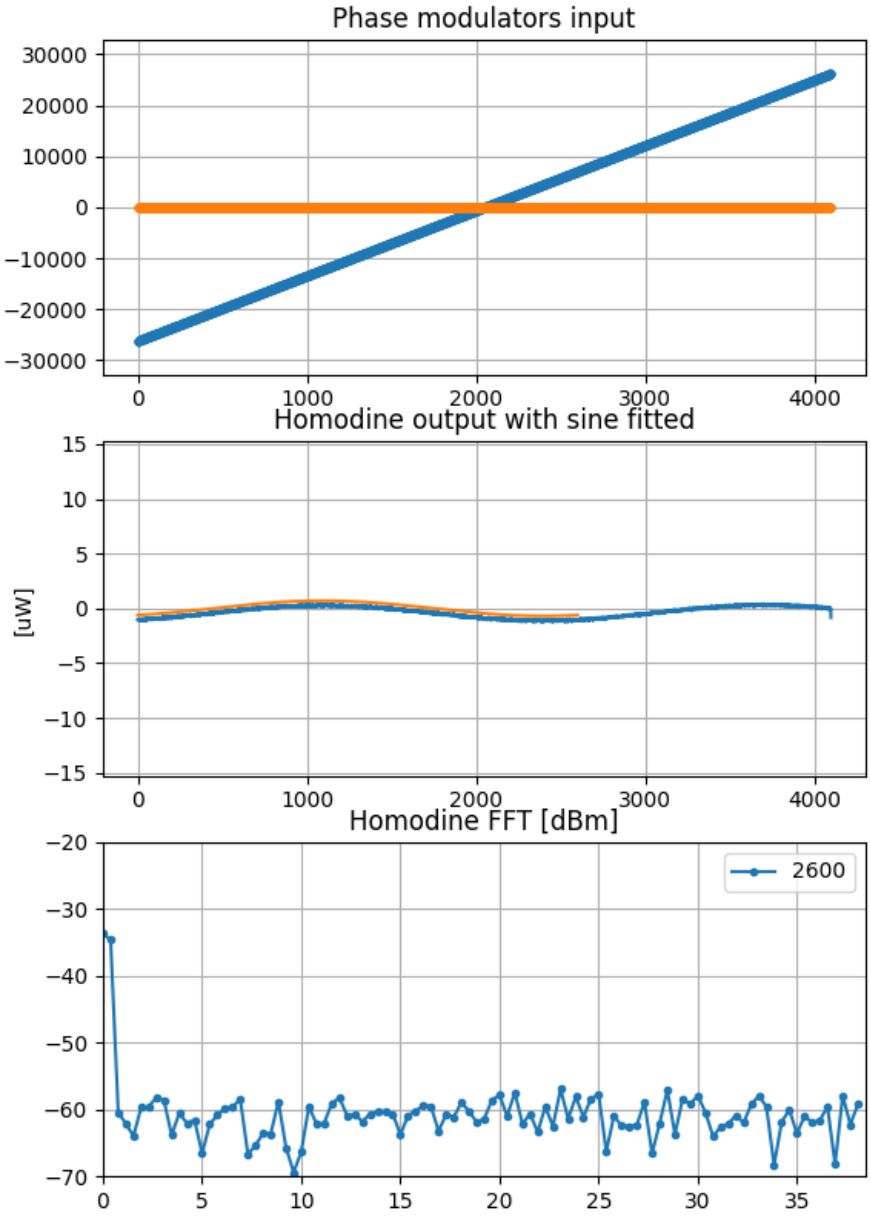
Ebben az elrendezésben az Alice és Bob oldalát megvalósító eszközök egy optikai padon helyezkedtek el. A tesztelés során Alice csak a fázismodulátorát használta, aminek a segítségével egy szinusz hullámot küldött Bob-nak (Bob ekkor csak az egyik kvadratúrát mérte). Több átlagos intenzitásérték mellett is megvizsgálására került a rendszer. A 5.1. ábra középső részén látható a szinuszhullám 1 nW-os átlagos intenzitás mellett. Ennél az értéknél nagyjából 7500 fotont tartalmazott egy impulzus. A jel-zaj viszony ebben az esetben 26.83 dB volt, ami nagyon jónak tekinthető.

A 5.2. ábra pedig a 0.2 nW-os átlagos intenzitás mellett mutatja a szinuszhullámot, ami ebben esetben már sokkal nehezebben látható. Ilyen átlagos intenzitásérték mellett már csak kb. 1500 foton található egy impulzusban és a jel-zaj viszony a mérések alapján 19.54 dB volt.



5.1. ábra. A szinusz 1 nW-os átlagos intenzitásnál.

Az átlagos intenzitás csökkentésével el tudtuk érni, hogy egy impulzusba 100-as nagyságrendű foton kerüljön, de ezekben az esetekben az ábrákon már nem figyelhető meg lényegi különbség. A laboratóriumban elvégzett mérések alapján látható, hogy a rendszer alacsony intenzitásérték mellett is képes működni, ami nagyon fontos ahhoz, hogy a kommunikáció kvantumosnak legyen tekinthető, de erről a tesztek



5.2. ábra. A szinusz 0.2 nW-os átlagos intenzitásnál.

eredményeinek összegzésében fogok írni. A rendszert az optikai asztalon 5.3. ábra szemlélteti.

5.3. ábra. A teljes rendszer az optikai asztalon

5.2. A rendszer tesztelése a Budapest Műszaki Egyetem és a Magyar Telekom kelenföldi állomása között

A tesztelés során a laborból a Bob-hoz tartozó eszköz került elszállításra a kelenföldi állomásra, mivel az Alice oldalán található lézert így sokkal könnyebben és biztonságosabban lehetett kezelni. A tesztelés azzal kezdődött, hogy megmértük az összeköttetés csillapítását az Alice oldalán kilépő és a Bob-hoz megérkező teljesítmény összehasonlításával. A mérések alapján a szakasz csillapítása 4.76 dB volt.

Ezután az Alice oldalán található lézeráram különböző értékei mellett a referencia és a hasznos jel intenzitását mértük meg Bob oldalán. A hasznos jel intenzitásának mérése esetében természetesen a referencia jelet lecsatoltuk a kimenetről. A mérési eredményeket a 5.1. táblázatban lehet látni.

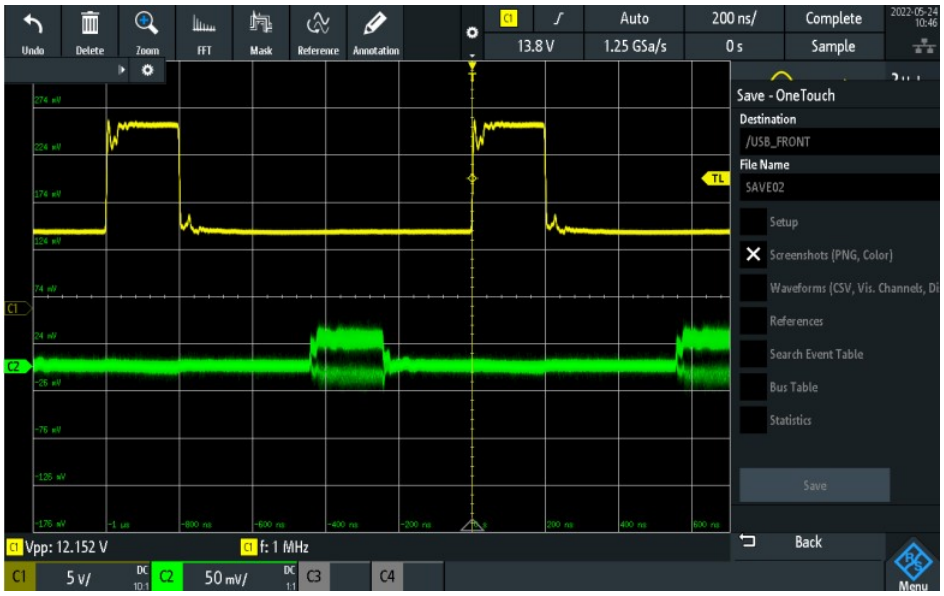
Lézeráram (mA)	Ref. Sig. intenzitás (mW)	Átlagos sig. intenzitás (nW)	Kulcsátvitel fotonszám Alice oldalán (kerekített érték)	eredményessége
140	1	2.1	48000	többnyire sikeres
120	0.8	1.9	43400	többnyire sikeres
100	0.5	1.7	38800	ritkán sikeres
100	0.5	2.4	54800	stabil kulcsátvitel

5.1. táblázat. A rövidebb szakasz tesztelés során mért értékek

Kiszámolható, hogy az impulzusok átlagosan 7800 fotont tartalmaznak (felhasználva, hogy mennyi energia található az $1\mu s$ idejű 1 nW-os lézerimpulzusban és hogy mekkora egy $1.5\mu m$ hullámhosszú foton energiája). Ebből vissza lehet számolni, hogy Alice-tól átlagosan hány foton érkezett egy impulzusban. Látható, hogy alapvetően nem a referencia lézerimplulzusok teljesítménye határozza meg a kulcsátvitel sikerességét, hanem a hasznos jelé. Az értékek jóval magasabbak, mint korábban, az optikai asztalos összeállításban alkalmazott teljesítmények. Ennek okait az 5.4. fejezetben elemzem röviden.

5.3. A rendszer tesztelése a Budapesti Műszaki Egyetemen és a Wigner Fizikai Kutatóközpont között

Ez a szakasz már sokkal hosszabb volt, mint az előző fejezetben bemutatott. Körülbelül 20 km volt az összeköttetés a két eszköz között. Az első lépés itt is a csillapítás meghatározása volt. A mérések alapján ez 8 dB volt. A mérték értékeket a 5.2. táblázat mutatja. Ennél a tesztnél a maximális 190 mA-es lézeráramot használtunk, mivel csak így tudtunk Bob oldalán elegendő referenciajel teljesítményt elérni. A 5.4. ábrán látható az oszcilloszkóp képernyője, amin a referencia és a hasznos jel látható. Ahogy arra a 3.2.1. fejezetben kitértem, a két jel egymástól eltolva utazik ugyanazon szálon, ami az ábrán is látható. A hasznos jel 4-5 nagyságrenddel is kisebb, mint a referencia.



5.4. ábra. Az oszcilloszkóp kimenete a tesztelés során.

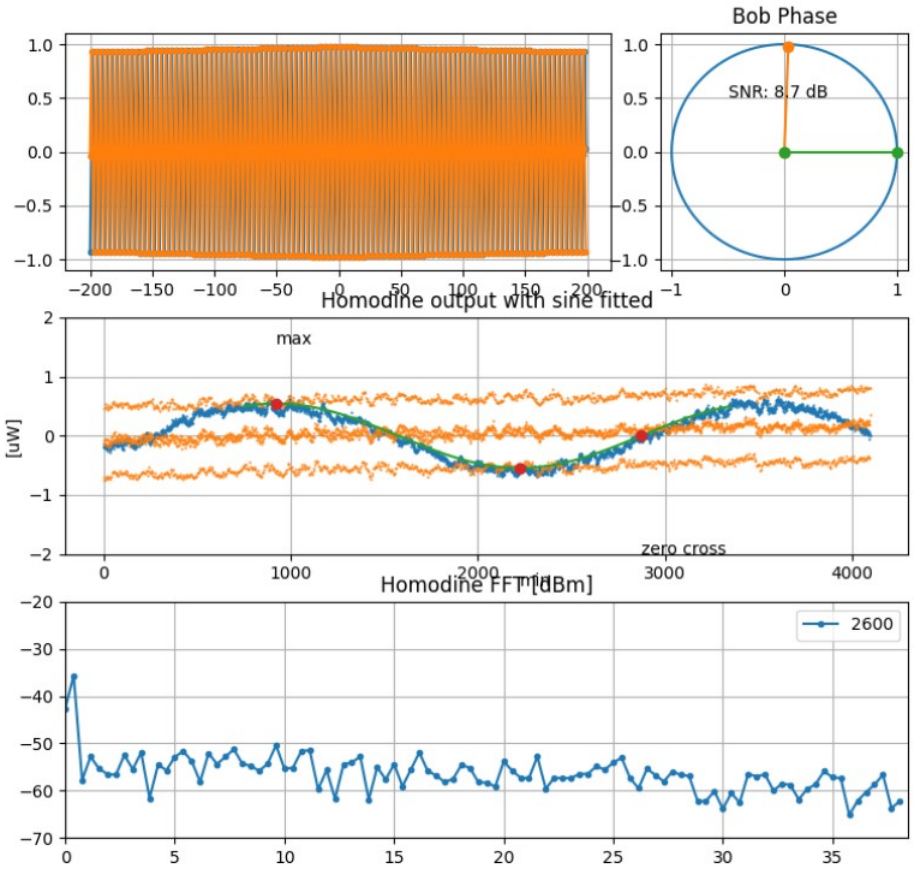
Lézeráram (mA)	Ref. Sig. intenzitás (mW)	Átlagos sig. intenzitás (nW)	Kulcsátvitel fotonszám Alice oldalán (kerekített érték)	eredményessége
190	0.35	1.8	88500	többnyire sikeres
190	0.35	2.8	137700	stabil kulcsátvitel

5.2. táblázat. A hosszabb szakasz tesztelés során mért értékek

5.4. A tesztek eredményeinek összegzése

A legfontosabb kérdés a tesztekkel kapcsolatban, hogy kvantumosnak tekinthető-e az átvitel. Ahhoz, hogy az legyen nagyjából < 100 foton/impulzus-ra van szükség átlagosan, amit az eredmények alapján a hálózati tesztelések során nem sikerült elérni. A laborban elvégzett, még az összeszerelések előtt megvalósított kísérletekben azonban elértük ezt a szintet. Az egyik lehetséges oka annak, hogy ezek az értékek a két teszt során ilyen magasak voltak az, hogy a rendszert nagy mértékben befolyásolják a klasszikus környezeti zajok. Ilyen lehet egy hangos eszköz a szobában vagy a feldolgozáshoz használt PC hűtésére szolgáló ventilátorok rezgése is. Ahhoz, hogy össze lehessen hasonlítani a laboratóriumi teszteket a Telekom hálózatán végzettekkel, az utóbbi esetben is végrehajtottuk a szinuszhullám elküldését. Ennek az eredménye látható a 5.5. ábrán. A szinuszhullám sokkal zajosabb, mint a laborban elvégzett mérések során, ez pedig a jelre rárakódó különféle zajoknak (pl. az említett klasszikus zajoknak) köszönhető.

A másik fontos eredmény a klasszikus kommunikációs résszel és azon belül a hibajavítással kapcsolatos. A tesztek során az LDPC kód csak 5 dB-nél nagyobb jel-zaj viszony mellett tudta kijavítani a Bob által küldött adatot. Mivel a használt kód



5.5. ábra. A szinusz átküldése az optikai hálózaton.

képes lenne BIAWGN csatornán alacsonyabb SNR mellett is hatékonyan működni, ezért a feltételezésünk az, hogy az említett klasszikus zaj miatt az elküldött adatokra nem csak normál eloszlásból származó értékek rakódnak rá.

6. fejezet

Összefoglalás

Dolgozatomban bemutattam, hogy mi az a kvantumos kulcsszétosztás és hogy miért egyre fontosabb a használata a ma meglévő titkosítások fényében. Kitértem rá, hogy milyen típusai vannak és mind a diszkrét, mind a folytonos változatából bemutattam példákat. Ezen példák működését lépésről lépésre ismertettem és a hozzájuk kapcsolódó háttértudást is tárgyaltam. Ezután kitértem a Budapest Műszaki Egyetemen épülő rendszer működésére és részletesen bemutattam, hogy a kvantumos kommunikáció után a klasszikus csatornán milyen lépések szükségesek ahhoz, hogy létrejöjjön a folyamat végén csak a két fél által ismert titkos kulcs, amit fel tudnak használni pl. a köztük lévő kommunikáció titkosítására.

Ezután rátértem arra a munkára, amit a rendszer kapcsán végeztem. Bemutattam, hogy milyen lehetséges *information reconciliation* megoldásokat vizsgáltam meg, amelyek széleskörűen használtak a QKD rendszerekben. Leírtam működésüket és az épülő rendszerben való felhasználhatóságukra is kitértem. Ezt követően részletesen foglalkoztam az általam választott protokoll lefutásához szükséges LDPC kódokkal, majd bemutattam annak működését. Következő lépésként a választott megoldást megvalósító, általam készített programot mutattam be kitérve arra, hogy milyen megoldásokat próbáltam ki és végül melyeket választottam és miért. A dolgozat végén pedig ismertettem a rendszer tesztelése során szerzett adatokat, melyek két szakaszon folytak egy magyar szolgáltató optikai hálózatán.

6.1. Továbbfejlesztési lehetőségek

Ahogy arra a tesztek összegzésében is kitértem, a rendszer még nem tökéletes, így bőven van még lehetőség annak továbbfejlesztésére és javítására. Számomra a legfontosabb a klasszikus kommunikáció javítása. Ezen a területen az elsődleges feladat a hibajavítás felgyorsítása és a minél alacsonyabb jel-zaj viszony melletti hatékonyság. Ehhez egyrészt az LDPC dekódoló algoritmus sebességén kell javítani, amely vagy

egy jobban optimalizált implementációval érhető el vagy egy erősebb feldolgozóegység segítségével (pl. egy GPU). Másrészt az alacsonyabb SNR kezeléséhez kisebb kódrátájú LDPC kódokat kell használni, azaz olyan mátrixokat, ahol a kódszavak hossza jóval nagyobb, mint az elküldendő információ, ami azt jelenti, hogy több a paritásbit, így jobban kijavíthatóak a hibák. Az ilyen kódokhoz tartozó mátrixok viszont nagy méretűek (ahogy azt a LDPC kódokkal foglalkozó programrészletet bemutató fejezetben említettem), így a kezelésükhöz másfajta hozzáállás szükséges, ami szintén a dekódolás javítását jelenti. Másrészről az is érdekes továbbfejlesztés lenne, ha a rendszerhez specifikusan hoznánk létre egy mátrixot ekkora méretben.

A rendszer másik fontos hiányossága ebben a pillanatban, hogy nem képes folyamatosan kulcsokat generálni. Ennek megoldásához szükséges, hogy rendelkezzünk egy kulcstárral, amit folyamatosan fel lehet tölteni a frissen generált kulcsokkal, így később bármikor felhasználhatóak lesznek. Ehhez természetesen szükséges szoftveres fejlesztés is, de a rendszer fizikai megvalósításában is módosításokat kell végezni, hogy stabilan tudjon működni huzamosabb ideig. A most használt *information reconciliation* megoldásban az egyik lépés két normál eloszlásból származó értéket oszt el. Mivel mindkét valószínűségi változó nulla várható értékű, így nagy a valószínűsége, hogy két nullához nagyon közeli számot osztunk el. Ennek kiküszöbölésére fel lehet használni a szferikus kódolást [15].

A generált kulcs felhasználásának tekintetében egy olyan példát is el lehet készíteni, amely jobban felkelti a rendszert megismerők figyelmét, mint a titkosított chat-alkalmazás. Egy ilyen alkalmazás lehet egy valósidejű videókapcsolat, amely a generált kulccsal kerül titkosításra.

Köszönetnyilvánítás

A kutatást az Innovációs és Technológiai Minisztérium és a Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal támogatta a Kvantuminformatika Nemzeti Laboratórium keretében.

Szeretném megköszönni Dr. Bacsárdi Lászlónak és Dr. Kis Zsoltnak azt a sok segítséget, amit a munkám és a dolgozat megírás során kaptam tőlük. Köszönöm szépen Belső Zoltánnak, hogy segített az implementációm javításában az ötleteivel és a tanácsaival.

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[2] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[3] Charles H Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *arXiv preprint arXiv:2003.06557*, 2020.

[4] Artur K. Ekert. Quantum cryptography based on bell's theorem. *Phys. Rev. Lett.*, 67:661–663, 1991.

[5] T. C. Ralph. Continuous variable quantum cryptography. *Phys. Rev. A*, 61:010303, 1999.

[6] Mark Hillery. Quantum cryptography with squeezed states. *Phys. Rev. A*, 61:022309, 2000.

[7] N. J. Cerf, M. Lévy, and G. Van Assche. Quantum distribution of gaussian keys using squeezed states. *Phys. Rev. A*, 63:052311, 2001.

[8] Frédéric Grosshans and Philippe Grangier. Continuous variable quantum cryptography using coherent states. *Phys. Rev. Lett.*, 88(5):057902, 2002. Number: 5.

[9] Paul Jouguet, Sébastien Kunz-Jacques, Anthony Leverrier, Philippe Grangier, and Eleni Diamanti. Experimental demonstration of long-distance continuous-variable quantum key distribution. *Nature Photonics*, 7(5):378–381, 2013.

[10] Duan Huang, Peng Huang, Dakai Lin, and Guihua Zeng. Long-distance continuous-variable quantum key distribution by controlling excess noise. *Scientific Reports*, 6(1):1–9, 2016.

[11] Alberto Carrasco-Casado, Verónica Fernández, and Natalia Denisenko. *Free-space quantum key distribution*, pages 589–607. Springer, 2016.

[12] Dr. Kis Zsolt and Belső Zoltán. Folytonos változójú kvantum alapú kulcs-szétosztó rendszer (cvqkd) működésének bemutatása. Technical report, BME, 2019.

[13] Fabian Laudenbach, Christoph Pacher, Chi-Hang Fred Fung, Andreas Poppe, Momtchil Peev, Bernhard Schrenk, Michael Hentschel, Philip Walther, and Hannes Hübel. Continuous-variable quantum key distribution with gaussian modulation – the theory of practical implementations. *Advanced Quantum Technologies*, 1(1):1800011, 2018.

[14] Gilles Brassard and Louis Salvail. Secret-key reconciliation by public discussion. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, volume 765, pages 410–423. Springer Berlin Heidelberg, 1994. Series Title: Lecture Notes in Computer Science.

[15] Anthony Leverrier, Romain Alléaume, Joseph Boutros, Gilles Zémor, and Philippe Grangier. Multidimensional reconciliation for continuous-variable quantum key distribution. *Phys. Rev. A*, 77(4):042325, 2008.

[16] Charles H. Bennett, Gilles Brassard, and Jean-Marc Robert. Privacy amplification by public discussion. *SIAM J. Comput.*, 17:210–229, 1988.

[17] Alexander Semenovich Holevo. Bounds for the quantity of information transmitted by a quantum communication channel. *Problemy Peredachi Informatsii*, 9(3):3–11, 1973.

[18] W. T. Buttler, S. K. Lamoreaux, J. R. Torgerson, G. H. Nickel, C. H. Donahue, and C. G. Peterson. Fast, efficient error reconciliation for quantum cryptography. *Phys. Rev. A*, 67(5):052303, 2003.

[19] G. Van Assche, J. Cardinal, and N. J. Cerf. Reconciliation of a quantum-distributed gaussian key. *IEEE Trans. Inform. Theory*, 50(2):394–400, 2004.

[20] Paul Jouguet, Sébastien Kunz-Jacques, and Anthony Leverrier. Long-distance continuous-variable quantum key distribution with a gaussian modulation. *Physical Review A*, 84(6):062317, 2011.

[21] Bruno Rijsman. A cascade information reconciliation tutorial. (utolsó elérés: 2022.10.08). https://hikingandcoding.wordpress.com/2020/01/15/a-cascade-information-reconciliation-tutorial/.

[22] Christoph Pacher, Philipp Grabenweger, Jesus Martinez-Mateo, and Vicente Martin. An information reconciliation protocol for secret-key agreement with small leakage. In *2015 IEEE International Symposium on Information Theory (ISIT)*, pages 730–734. IEEE, 2015.

[23] Jesus Martinez-Mateo, Christoph Pacher, Momtchil Peev, Alex Ciurana, and Vicente Martin. Demystifying the information reconciliation protocol cascade. *arXiv preprint arXiv:1407.3257*, 2014.

[24] Mario Milicevic, Chen Feng, Lei M Zhang, and P Glenn Gulak. Quasi-cyclic multi-edge ldpc codes for long-distance quantum cryptography. *NPJ Quantum Information*, 4(1):1–9, 2018.

[25] William Ryan and Shu Lin. *Channel codes: classical and modern*. Cambridge university press, 2009.

[26] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[27] David JC MacKay and Radford M Neal. Near shannon limit performance of low density parity check codes. *Electronics letters*, 32(18):1645, 1996.

[28] A. Cassagne, O. Hartmann, M. Léonardon, K. He, C. Leroux, R. Tajan, O. Aumage, D. Barthou, T. Tonnellier, V. Pignoly, B. Le Gal, and C. Jégo. Aff3ct: A fast forward error correction toolbox! *Elsevier SoftwareX*, 10:100345, 2019.