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

Istenes Dóra Márton Botond László

# KVANTUM ALAPÚ VÉLETLENSZÁM-GENERÁTOROK MŰKÖDÉSI HATÉKONYSÁGÁNAK NÖVELÉSE

KONZULENS

Dr. Bacsárdi László

BUDAPEST, 2020

# Tartalomjegyzék

**Kivonat** .... **4**

**Abstract** .... **6**

**1 Bevezetés és motiváció** .... **7**

1.1 Jó véletlenszámok szükségessége .... 7

1.2 Véletlenszámok előállítása .... 8

1.3 Célkitűzéseink .... 9

1.4 A dolgozat felépítése .... 9

**2 Kvantum véletlenszám generátorok** .... **11**

2.1 A Műegyetemen épülő kvantumvéletlenszám- generátorok .... 11

2.1.1 Beérkezési időn alapuló QRNG .... 12

2.1.2 Spontán emisszión alapuló QRNG .... 13

2.2 A QRNG-k ismertsége .... 13

2.2.1 A kvantuminformatika és -kommunikáció fontossága .... 13

2.2.2 QRNG-k az üzleti világban .... 14

**3 A generátorok tesztelése és az extraktorok** .... **16**

3.1 A számok minőségének mérése .... 16

3.2 A Dieharder tesztek .... 18

3.3 Extraktorok .... 19

3.3.1 A determinisztikus extraktorok .... 20

3.3.2 A seeded extraktorok .... 20

**4 A Műegyetemen épült generátorok tesztelése és vizsgálata utófeldolgozással** .... **22**

4.1 A dolgozat során használt architektúra .... 22

4.2 Az utófeldolgozás során használt exraktorok értékelésük .... 23

4.2.1 Az XOR művelet, mint extraktor .... 23

4.2.2 A Neumann-féle extraktor .... 24

4.2.3 Az iteráló Neumann-féle extraktor .... 26

4.2.4 A N-bites Neumann-féle extraktor .... 27

4.2.5 A H-függvény .... 29

4.2.6 Az S-boxot használó extraktor .... 31

4.2.7 A Toeplitz-mátrixot használó extraktor .... 33

4.2.8 A hash függvények mint extraktorok ........ 34
4.3 A generátorok tesztelése ........ 35
4.3.1 A spontán emisszión alapuló generátor vizsgálata ........ 36
4.3.2 A beérkezési időn alapuló generátor vizsgálata ........ 38
4.3.3 A tesztek eredményeinek összegzése ........ 40
4.4 A kimenet vizualizálása ........ 40
4.4.1 Miért jó egy webes szoftver? ........ 41
4.4.2 A megvalósítás ........ 42
**5 Összefoglalás ........ 44**
**Irodalomjegyzék ........ 45**

# Kivonat

A jó minőségű véletlen számok egyre fontosabbak a mai világ kriptográfia rendszereiben, folyamataiban. Az internetes kommunikációtól a kriptovalutákon át egészen az online szerencsejátékig nagy szükség van rájuk. A legelterjedtebb álvéletlen véletlenszám generátorok könnyen elkészíthetőek, de a determinisztikusság problémájával küzdenek. Ezért nagyon fontosak az olyan generátorok, amelyek a természetből eredő véletlenséget használják ki, ezek a valódi véletlenszám generátorok. Egy változatuk – a kvantum alapú véletlenszám-generátor, QRNG – a kvantumfizika szabályait felhasználva állítja elő a számokat.

Ezek a valódi véletlenséget kihasználó generátorok már kereskedelmi forgalomban is kaphatóak. Nagyon fontos szempont a különböző generátoroknál (legyen az álvéletlen vagy valódi) az előállított számok minősége. Ennek a tulajdonságnak a meghatározásához különböző statisztikai teszteket lehet használni, melyek azt hivatottak megállapítani, hogy a számok, mint bitsorozat mennyire tér el a teljes véletlen bitsorozattól. Általános esetben a generátorok által létrehozott számok nem egyből tökéletes minőségűek, ezért utófeldogozás szükség a számokon. Ez a folyamat extrataktorok beiktatásával történik, melyek segítségével a minőség (a számok véletlensége) növelhető. Ezek a megoldások viszont csökkentik a generátor kimeneti sebességet.

Munkánkban egy, a Műegyetemen fejlesztett kvantum véletlenszám generátorral dolgozzunk. Tesztek futtatásával megállapítottuk, hogy a generátor által létrehozott számok önmagukban még nem megfelelő minőségűek, ha a nagy bitsebesség a cél. Azért, hogy ezt növelni tudjuk, különböző utófeldolgozási eljárásokat alkalmaztunk. Ezeknek az eljárásoknak az a célja, hogy a kapott számokot, azokon műveletek elvégzése révén, javítsunk. Ezáltal a forrásból, a generátorból, fakadó hibákat kiküszöböljük. Ennek a folyamatnak a neve az extrakció, melynek során a kapott számokból "kivonjuk" a véletlenséget és azt felerősítve javítják a generátor kimenetét. Több különböző elven működő extraktort összehasonlítottunk. Az eredményük alapján pedig javaslatot teszünk arra, hogy melyiket érdemes használni az általunk vizsgált generátoron. Végül pedig

létrehoztunk egy webes felületet, melyen elérhetőek a generátoron végzett tesztek eredményei az utófeldogozási eljárások előtt, illetve után. Ezenkívül a felületen elérhetőek a generátor által előállított számok is valós időben, így mindenki számára elérhetőek a jó minőségű kvantum véletlenszámok.

# Abstract

Good quality random numbers play a more significant role in today's world's cryptographic systems and processes than ever before. From the communication on the internet, through crypto currencies to online gambling they are much needed. The most common way to generate these numbers are the pseudorandom generators, but they suffer from the problem of determinism. Therefore, generators which produce numbers based on natural randomness are high in demand. They are called true random number generators. A particular version of them – called quantum random number generators, QRNG – uses the laws of quantum physics to produce random numbers.

These true random number generators are already available on the commercial market. The most important question regarding them (pseudo or true) is the quality of the numbers they are creating. To measure this attribute one can use statistical tests, to determine how far the numbers (represented as a stream of bits) are from a truly random bitstream. In general, we can tell that the output of a given generator is not always of good quality. Therefore, we need some kind of postprocessing on the numbers. One way to do this is by using extractors. With their help we can improve the quality (the randomness) of the produced output. Unfortunately, these kinds of operations lower the output rate of a generator, sometimes by a significant amount

In our work, we examined a quantum random number generator built in BME. When we used the mentioned statistical tests, we concluded, that if a high output bitrate is important, we couldn't produce numbers with good enough quality. To meet the quality requirements, we used a different number of postprocessing methods. These technics, called extractors, are aimed to increase the randomness of the numbers by using special operations on them and therefore mitigating the problem coming from the source (the generators). They extract the "randomness" from a the bitstream and then amplify it to make the output of a generator better. We collected a number different extractors and compared them to each other. With the help of the result we propose a possible postprocessing architecture for the used generator. Lastly, we developed a web-based platform, where one can access the result of the tests on the numbers before and after applying postprocessing. Besides that, the platform enables one to get numbers from the generator in real time. Thus, making it accessible for a wider audience.

# 1 Bevezetés és motiváció

Ha beírja az ember a Google keresőbe, hogy Random Number Generator ,akkor nagyon sok keresési találat tárul a szeme elé, különböző féle generátorokkal, elsőként a Google sajátjával. Az, hogy ennyi féle és fajta random generátor készült az évek során, egy bizonyíték arra, hogy az embereknek szüksége van véletlen számokra. Akár csak egy rövid eldöntendő vitához, vagy akár egy nagy ipari felhasználáshoz. Azonban rengeteg különbség létezik a generált számok minőségében.

## 1.1 Jó véletlenszámok szükségessége

Jó véletlenszámokra nagyon sok területen van szükségünk. A legalapvetőbb és a legkisebb fontosságú felhasználás a lokális, emberek közötti interakció. Különböző viták eldöntésére, játékok hiányzó alkatrészeihez használnak naponta rengetegen dobókocka szimulátort vagy hasonló alkalmazást A második felhasználás számítógépes játékok logikájához kapcsolható. A mai világban a legtöbb játék tartalmaz valamiféle véletlenszerűséget, ami egy fontos működési elem, és bár felhasználói elégedettség szempontjából nem elhanyagolható ezeknek a véletlenszámoknak az ideálissága, közel sem annyira fontos, mint a harmadik csoporté. Ezek azok az alkalmazások, amelyek vagy kifejezetten sok embert érintenek, vagy a véletlenszámok nagyon kritikus pontokon vannak használatban. Ilyen területek: a kriptográfia, különböző valószínűség számítási metódusok, vagy egyéb ipari alkalmazások mint a lottóhúzás és más szerencsejátékok [1]. A szerencsejátékokban (mint a lottóhúzás) régebben megoldható volt a véletlen faktor kézzel történő előállítása, ami régebben úgy valósult meg, hogy egy játékmester kihúzta egy gömbből a számokat. Ám a mai világban egyrészt rengeteg féle lottóhúzási koncepció alakult ki, valamint az érdeklődés a szerencsejátékok iránt is jobban elterjedt. Ezen tényezők miatt sokkal nagyobb mennyiségű véletlen számra lett szükség, ami megnehezíti a klasszikus lottóhúzási ceremóniát. Ezen a területen rendkívül fontosak a jó randomságú számsorozatok.

A kriptográfia az egyik legfontosabb alkalmazási területe a véletlen számoknak [1]. Különböző titkosítási eljárások léteznek. Manapság rendkívül népszerű az RSA

algoritmus, amelyen alapul a nyílt kulcsú titkosítás. Ezen elterjedt kriptográfiai módszernek viszont megvan a saját hátránya, ami a technológia gyorsuló fejlődésével egyre inkább kiaknázhatónak tűnik. Ha viszont szimmetrikus kulcsú titkosítást szeretnénk használni, akkor a különböző kulcsmegosztó protokollok működéséhez jó minőségű véletlen számokra van szükségünk. Szimmetrikus kulcsú titkosítás során a felek ugyanazt a kulcsot használják az üzenet kódolására és dekódolására, a különböző üzenetekhez pedig - ha például One Time Pad titkosítóról beszélünk - lecserélik. Ezen metódus olajos működéséhez nagyon sok gyors ütemben generálódó, és legfőképpen nem kikövetkeztethető véletlen számokra van szükség.
De hasonló a helyzet a különböző digitális aláírások világában is, az egyszer használatos véletlen számoknak (nonce) fontos részét képezik a biztonságos működésnek.

## 1.2 Véletlenszámok előállítása

Véletlenszámok tökféleképpen előállíthatóak, és a különböző módokban sok féle párhuzam vonható. Alapvetően háromféle kategóriába lehet sorolni a generátorokat: PRNG, vagyis Pszeudo Random Number Generator, TRNG vagyis True Random Number Generator, és a QRNG (Quantum Random Number Generator).
A PRNG-k manapság a legjobban elterjedt formája a random szám generátoroknak. Mint a nevében is látszik, ezeknek a generátoroknak a kimenete nem igazi random számok, vagyis nem annyira mint egy kocka vagy pénz érme dobásának eredménye. Ezeket a számokat egy matematikai képlet, algoritmus generálja vagy előre kiszámolt táblázatok alapján számolják ki őket. A fentebb említett módszerek eredménye, bár véletlennek tűnő számokat ad vissza, bizonyos adatok birtokában könnyen visszafejthetőek. A pszeudo random számok legnagyobb hátránya a determinisztikusságuk hiszen a generálás kiindulási értéke és a generálási algoritmus tudatában, mindig kiszámítható a következő elem. A pszeudo random számok előnye a hatékonyságukban rejlik. Bár a generált számok nem tökéletesen randomak, sőt akár nagyon hosszú periódusonként ismétlődhetnek is, rövid idő alatt rendkívül sok generálható belőlük. így olyan alkalmazási területeken nagyon hasznosak, ahol sok szám generálására van szükség rövid idő alatt, és amelyben az alkalmazások akár ki is tudják használni a periodicitásukat.

Ilyenek a különféle szimulációk, és modellezési applikációk. Vannak alkalmazási területek, ahol viszont a PRNG-k hátránya komoly biztonsági rést okoz a rendszerben, így ezen területeken inkább TRNG-ket alkalmaznak [2]. A TRNG-k olyan generátorok, amelyek egy természetüknél fogva random eseményt vesznek a generálás alapjául. Ezek az események általánosságban fizikai eseményeket jelentenek, például a radioaktív bomlás, atmoszférikus zaj. Idetartoznak a kvantum alapú véletlen szám generátorok is, de ők kiemelt helyzetben vannak a TRNG-k között. Általános esetben egy TRNG nagyon jó számokat generál, viszont lassan, és nagyon körülményesen. Mind az atmoszférikus rajzhoz, mind a radioaktív bomlás méréséhez pontos műszerekre, és ideális körülményekre van szükség. Az alacsony hatékonysága ellenére mégis rengeteg területen inkább TRNG-t használnak a számok valódi randomsága miatt.

## 1.3 Célkitűzéseink

Miután megismerkedtünk különböző optikai alapú kvantum véletlenszám-generátor architektúrákkal, célul tűztük ki az ezek által előállított véletlenszámok minőségének vizsgálatát, és a véletlenszámok minőségének javítását különböző extraktorok segítségével. Ehhez saját programokat fejlesztettünk. Python nyelven implementáltunk különböző extraktorokat, melyekkel feldolgoztuk a tesztelendő generátorok kimenetét. Vizsgálataink során az egyes extraktorok alkalmazási lehetőségeit mérlegeltük, részletesen elemezve azok működését. Végül a vizualizációra is hangsúlyt helyeztünk. Létrehoztunk egy webhelyet amely streameli egy generátor kimenetét, valamint lehetővé teszi az extraktorok alkalmazását ezen a kimeneten. Az általunk készített platform szintén lehetővé teszi a kimenetek tesztelését is.

## 1.4 A dolgozat felépítése

A dolgozat felépítése a következő. A 2. fejezetben bemutatjuk a Műegyetemen épülő kvantum alapú véletlenszám-generátorok működési elvét, valamint ismertetjük ezek üzleti felhasználásai lehetőségeit. A 3. fejezetben bemutatjuk, hogy mit jelent, hogy egy véletlenszám-generátor által előállított számok jó minőségűek, megmutatjuk, hogy melyek a legtöbbet alkalmazott módszerek ezen minőség tesztelésére és be is mutatjuk azok működést. Kitérünk arra, hogy melyek a legismertebb olyan tesztcsomagok, melyek segítségével lehet véletlenszám-generátorokat tesztelni és az egyik ilyen csomag

bizonyos tesztjeit be is mutatjuk. Bevezetjük az extraktorok fogalmát, részletesen elmagyarázzuk, hogy mi működésük alapelve és milyen típusaik léteznek. A 4. fejezetben ismertetjük, hogy milyen módszerrel teszteljük a Műegyetemen készült két generátort és végig vesszük, hogy a tesztelés során milyen extraktorokat fogunk használni. Ezen extraktorok működését részletesen tárgyaljuk és értékeljük azok felhasználhatóságát QRNG esetében. Ezután bemutatjuk a generátorok tesztelése során kapott eredményeinket és azokat értékeljük. Majd a fejezet végén bemutatjuk a webes alkalmazásunk főbb pontjait és allűrjeit.

# 2 Kvantum véletlenszám generátorok

A QRNG-k egy különleges alosztálya a TRNG-k csoportjának. Ezek a generátorok a kvantumos tulajdonság eredendő megjósolhatatlanságát használják ki. A kvantum alapú RNG-k abban különböznek a többi TRNG-től, hogy sok esetben egyszerűbb a felépítésük, anyagköltségük, a környezeti igényük. Fontos tulajdonságuk még, ami az egyik legnagyobb előnyük, hogy rövid idő alatt képesek nagymennyiségű adatot kibocsátani magukból. Hasonlóan a kockadobáshoz, vagy a pénzfeldobáshoz ideális generátoroknak tűnnek, ám rendkívül sok fizikai tényező szólhat közbe, ami torzíthatja az eredményeket. A generátorok működése kvantumfizikai eseményeken alapszik, amelyek a tiszta elméleti síkon tökéletes eredményeket biztosítanak, azonban ezek technológiai megvalósítása torzítja az eredményeket. Ezek a torzítások sok esetben az eszközök eredendő hibáin múlnak, például nem létezik tökéletes nyalábosztó, nem létezik tökéletes detektor, nem létezik tökéletes vezeték, sőt, sok esetben két felhasznált eszköz tökéletlensége nem egyezik. Példaként az útelágazáson alapuló generátorban (melynek működése azon alapszik, hogy két külön út végén két külön detektor áll, az utakra pedig egy nyalábosztó tereli a fotonokat) elméleti esetben 50%-ban az egyik, 50%-ban a másik detektor jelzi egy fotont, így a detektorokhoz rendelve a 0, 1 értékeket készen áll a véletlenszám-generátor. Gyakorlatban azonban a nyalábelosztó nem teljesen 50-50%-ban osztja a fotonokat, a detektorok nem minden beérkező fotont érzékelnek, így máris nem egyenletes sorozatot fogunk sorsolni.

## 2.1 A Műegyetemen épülő kvantumvéletlenszám- generátorok

Technológiájukat tekintve a QRNG-nek rengeteg fajtája létezik. Az egyik ilyen az optikai QRNG, viszont ez is magába foglal rendkívül nagyszámú különböző topológiát. Ezek közül kettő épül a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Karán: egy spontán emisszión alapuló generátor és egy beérkezési időn alapuló architektúra. Ezek működésének kifejtése a következő két fejezetben történik.

### 2.1.1 Beérkezési időn alapuló QRNG

A 2.1. ábrán is bemutatott architektúra nagyvonalakban arra épül, hogy koherens, stabilis időbeli optikai teljesítménnyel rendelkező fényforrásból származó fotonok detektálásának közötti időt vesszük a generáció alapjának. A beérkezési időn alapuló generátor egyetlen egy darab detektort használ, így kiküszöbölve a különböző detektorok közötti eltérés problematikáját [3][4] .

Kétféle detektort szoktak általában használni, SPAD-t (Single Photon Avalanche Detector) vagy pedig fotonsokszorozót (másnéven elektronsokszorozónak is hívják). A fotonsokszorozó előnye, hogy nincs regenerálódási holt ideje, ellentétben a SPAD-al ami két foton beérkezése között egy rövid ideig inaktív.

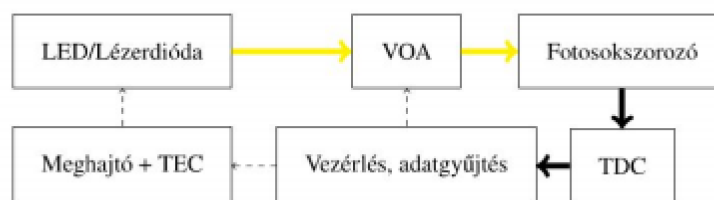
A fotonok beérkezése exponenciális eloszlású, így könnyen belátható, hogy a beérkezési idők eloszlása is exponenciális eloszlású $\lambda e^{-\lambda t}$ , ahol lambda az egységnyi idő alatt beérkezett fotonok várható értéke.

A $t_1, t_2, t_3, t_4$ beérkezési időket, párosával hasonlítjuk össze. Ha $t_1 > t_2$ akkor 0-ás ként, ha pedig $t_2 > t_1$, akkor pedig 1-es ként értelmezzük a mérés eredményét. Ezzel bár lecsökken a bitsebesség, kiküszöböljük az adatok legevidensebb korrelációját, hiszen minden adatot így pontosan egyszer fogunk felhasználni.

Nem tökéletes még az adat extrakciónk, ugyanis az időértékeket digitális jellel közelítjük. Ez két problémát vet fel: a közelítés során gyakrabban felléphetnek egyenlő értékek, amik valószínűsége egyéb esetben megközelíti a nullát. Ezeket az értékeket el kell vetnünk, így hozzájárulva a bitsebesség további csökkenéséhez. Másik probléma a digitális órajel folytonossága. Amennyiben az órajel folytonos az elejétől a mérés végéig, az egy újfajta korrelációhoz vezet. Ezen korreláció csökkenthető nagyon gyors vagy nagyon lassú órajellel, de amíg a nagyon lassú órajel, használhatatlan szintre csökkenti a bitsebességet, a nagyon gyors órajel a gyakorlatban sokszor nem kivitelezhető, a hardver komplexitása miatt.

A fentebb említett hibákra van nagyvonalakban megoldás, ám van néhány egyéb környezeti változó, ami ronthatja a generált számok minőségét. Ezek közé tartozik a fotondetektor sötétzaja (dark count rate), ami a detektor nem dezignált fotonokból származó jelzéseinek mutatója, hanem egyéb külső tényezők hatása. A másik hasonló a

vezeték idealitásának hiánya, melynek hatására néhol elnyelődnek fotonok, így rontva a mérés eredményét.




**2.1. ábra: A beérkezési időn alapuló generátor sematikus ábrája [3]**

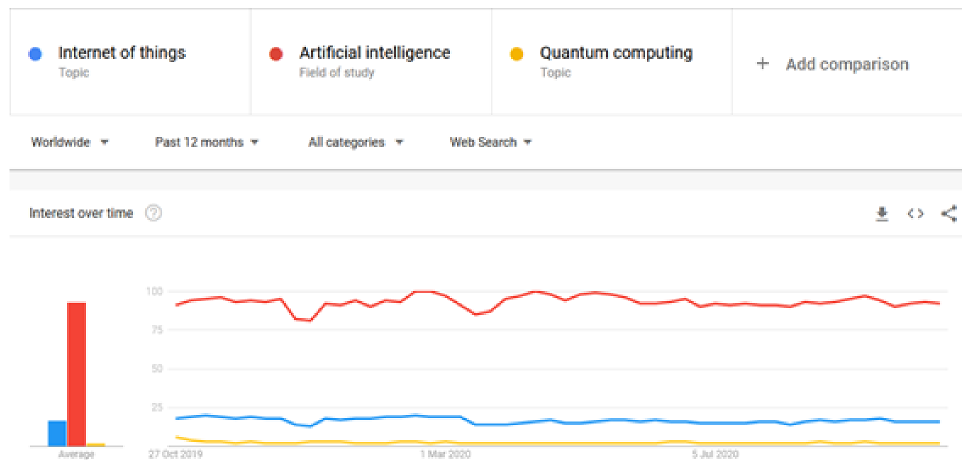
### 2.1.2 Spontán emisszión alapuló QRNG

Az erősített spontán emisszión (ASE) alapuló megoldások a spontán emissziót (egyikét az anyag-fény kölcsönhatásoknak) és rokonát az indukált emissziót használják ki.. Az indukált emisszió során egy foton gerjesztett állapotban yan. Ha ez kölcsönhatásba lép egy másik fotonnal akkor kiesik a gerjesztett állapotból, és kibocsát egy olyan tulajdonságokkal rendelkező fotont mint amilyennek ütközött. A spontán emisszió esetében az alapállapotba történő visszaesés spontán jön létre, viszont ez is foton kibocsátással jár, ami zajként jelenik meg a kimenő jelben. A két kölcsönhatás együttes kihasználásából történik az erősített spontán emisszió, vagyis a spontán emisszió által keltett zaj felerősítése indukált emisszióval [3][4].

## 2.2 A QRNG-k ismertsége

### 2.2.1 A kvantuminformatika és -kommunikáció fontossága

A kvantuminformatika és -kommunikáció bár egy rendkívül fontos terület, az emberek többsége számára részben ismeretlen terület. Mint a 2.2. ábrán is szemléltetett Google trend statisztikából is látszik, kevésbé népszerű hasonló enyhén „buzzword”-nek nevezhető témákhoz képest mint például a mesterséges intelligencia (Artificial Intelligence, AI) vagy a dolgok hálózata (Internet of Things, IoT). A kvantum, mint terület annyira idegen a többségtől, hogy sok tudományos fantasztikus filmben szerepel egy kvantum előtaggal rendelkező futurisztikus szerkezet. Ezen apró példából is érezhető a misztikum, ami körbeveszi a területet.




**2.2. ábra: Google trend statisztika három népszerű kifejezésre**

A kvantumkommunikáció témakörén belül a kvantum alapú kulcsszétosztás (quantum key distribution, QKD) és a kvantum véletlenszám-generátorok azok a területek, amelynek ismertsége és jelentősége - különösen üzleti szinten - egyre nő az utóbbi években. Az Inside Quantum Technology jelentése alapján, az egyik olyan terület amiben a legtöbb jövedelem növekedés várható az elkövetkező években [5].

### 2.2.2 QRNG-k az üzleti világban

Korábban említettük, hogy relatíve ismeretlenek kvantum alapú véletlenszám-generátorok, de igyekeztünk felderíteni hogy piaci környezetben milyen generátorok fordulnak elő és érhetőek el.

Az Az ID Quantique (IDQ) egy svájci cég, amely a Genovai egyetem spin-off cégeként kezdte, majd kinőtte magát a mai napig az egyik legdominánsabb céggé a területen, kutató intézetekkel Angliában és Koreában (Svájc mellett) [6].

Quantum Numbers Corp (QNC) egy kanadai cég aki specializálták magukat a QRNG alapú technológiákban. A QNC QN2 generátora az alagúthatást használja véletlen számok előállításához.

A két cég QRNG technológiáit felhasználják nagyon széles körben, a banki szférától kezdve egészen a katonai alkalmazásokig. [7].

Mind a két cég jeleskedik mikrochip méretű generátorok fejlesztésébe n[8][9]. A QNC eredménye volt az első chip méretű generátor, viszont az IDQ fejlesztési érdeme a legkisebb QRNG chip, a Quantis QRNG chip, ami szintén egy spontán emisszión alapuló megoldást használ, aminek lényege, egy LED dióda spontán foton kibocsátását nézzük, a kvantum zaj hatására. Ezek az adatok továbbítódnak egy CMOS sensorba, ami "megszámolja" ezeket a fotonokat. Ennek a kimenetét, közvetlenül el lehet érni, de ez továbbítódik egy NIST 800-90 szabvány által elvárt DRBG-be (deterministic random number generator algorithm) és azt is le lehet kérdezni.[10]

Érdekes módon, a rendszer érzékeli a saját hibáit és ezt vissza vezetve önmagába ezeket korrigálja.

Az IDQ generátoraival dolgozik a Loterie Romande, Française des Jeux, Novomatic, BetCruise, A Bet A, Poker Match szerencse és számítógépes játékokkal foglalkozó cégek, valamint az IDQ-t bízta meg a az NS&I brit nemzeti kincstár az ERNIE (Electronic Random Number Indica-tor Equipment) 5. verziójának elkészítésével[11], kvantum alapokra helyezve a rendszert. Alkalmazási területei még az IDQ chipeknek az Iot és a telekommunikáció, ami többek között egy hosszútávú együtt működésben mutatkozik meg a koreai SK Telecommal.

# 3 A generátorok tesztelése és az extraktorok

## 3.1 A számok minőségének mérése

Különböző módon tudjuk tesztelni a véletlenszámok minőségét [5,12], a statisztikai tesztek [13,14] a számok minőségének meghatározásában segítenek. Láthattuk az előző fejezetben sok, különböző típusú véletlenszám-generátor létezik. Ezen generátorok eltérő megoldásokat használnak arra, hogy előállítsák a számokat, amelyeket később fel lehet használni. Bizonyos módszerek ezek közül függenek egy természeti jelenségtől vagy egy olyan hasonló folyamattól, amely az időben változik. Ezek a változások nagy mértékben tudják befolyásolni az előállított számokat. Azért, hogy a generátort használó tudja, az adott időszakban milyen állapotban van az eszköz, fontos egy olyan módszer, amivel el lehet dönteni, megfelelően működik-e a generátor. Azaz tényleg véletlenszámokat állít-e elő. Természetesen, amikor nem egy olyan fizikai forrást használunk, amiről bizonyított, hogy véletlen (bizonyos feltételek mellett), hanem egy álvéletlen generátort, akkor az a legfontosabb, hogy tudjuk, hogy maga a generátor elég jó minőségű-e.

A minőség megállapításához teszteket szokás alkalmazni. Ezen tesztek a nagyon egyszerűtől és könnyen számíthatótól a komplexig és hosszúig terjednek. A tesztek magukon a generált számokon dolgoznak, amelyeket vagy bitsorozatként kezelnek, vagy fix mennyiségű bitenként számokat képeznek belőlük és azokat vizsgálják. Bár a tesztek komplexitásban egymástól eltérhetnek a céljuk mindig az, hogy megtalálják: a generált számsor mennyiben tér egy tényleges véletlen forrástól. Ezekkel a tesztekkel a véletlent magát szeretnénk megmérni, ami nem mérhető mennyiség, ezért a tesztek valószínűségi alapon működnek. Az összes generált számot nem tudjuk kezelni, csak egy részüket. Így az eredmény attól függ, hogy mely számokat vizsgáljuk. Ezért a teszteket egymás után többször is elvégezhetjük.

A legegyszerűbb ilyen teszt egy egyszerű statisztikai teszt, ami azt vizsgálja, hogy az előállított bitsorozatban mennyi az egyesek és a nullások száma. Ezt a tesztet frekvenciatesztnek hívják. Alapja, hogy ha teljesen véletlen a bitsorozat, akkor az egyesek és nullák száma közel azonos kell, hogy legyen. Ehhez hasonló teszt például a résteszt. Ebben az esetben az vizsgáljuk, hogy például két nulla között mennyi bitnyi távolság van.

Fontos megemlíteni, hogy ennél a két tesztnél beszélhetünk a teljes bitsorozatról vagy annak egy szakaszáról. Hiszen, ha például csak egy kis szakaszát vesszük a teljes sorozatnak és abban csak egyeseket találunk, akkor bár az a szakasz nem teljesít jól a teszten, de a teljes sorozat még szerepelhet megfelelőn. Ez természetesen a másik irányba is érvényes. Egy komplexebb teszt például a Kolmogorov–Szmirnov-próba [15], mellyel össze lehet hasonlítani két minta eloszlását.

Általánosságban elmondható, hogy a használt tesztek nagy része a statisztikai hipotézisvizsgálat csoportjába esik. A hipotézisvizsgálat során egy állítást, a nullhipotézist ($H_0$) szeretnénk elfogadni, vagy megcáfolni. Amikor egy ilyen tesztet alkalmazunk egy véletlenszám-generátorra, akkor a nullhipotézis az, hogy a generátor teljesen véletlen számokat állít elő. A másik állítás az alternatív hipotézis, amit $H_a$-val jelölünk és a nullhipotézis tagadása. Egy generátor esetében ez azt jelenti, hogy a generátor nem véletlen számokat állít elő. Az egyes tesztekhez megadható egy valószínűségi változó, amihez előállítható egy eloszlásfüggvény, amennyiben feltesszük, hogy a nullhipotézis igaz. Ezután kiválasztunk egy $\alpha$ kritikus értéket ezen az eloszlásfüggvényen (ez nagyon alacsony érték szokott lenni). Majd kiszámoljuk a statisztikai értéket a generátor által adott bitsorozaton. Ha ez az érték a kritikus szint ($\alpha$) alatt van, akkor nem utasítjuk el a nullhipotézist. Ha fölötte, akkor elutasítjuk. A vizsgálatnak négy kimenetele lehet. Ha $H_0$ tényleg igaz és mi elfogadtuk, akkor jól döntöttünk (ennek a valószínűsége 1-$\alpha$). Ha $H_0$ igaz volt, de mi ennek ellenére elvetettük, akkor rosszul döntöttünk (ennek a valószínűsége $\alpha$). Ez hívják 1. típusú hibának (false positive). Ha $H_0$ hamis és mi elvetettük, akkor jól döntöttünk (ennek a valószínűsége 1-$\beta$). Ha $H_0$ hamis volt és mi nem vettettük el, akkor nem döntöttünk jól (ennek a valószínűsége $\beta$). Ezt az esetet nevezik 2. típusú hibának (false negative). A két hibatípus értékét mindig fontos figyelemmel követni. Az 1. típusút $\alpha$ helyes megválasztásával lehet befolyásolni, amelyet általában 0.05-re, azaz 5 %-ra szoktak állítani. A 2. típusú hibát a minta méretével és $\alpha$ értékével lehet változtatni. Egy hipotézisvizsgálatnak a 2. típusú hiba minimalizálása a fontosabb. Például, ha egy bírósági tárgyalásra gondolunk, ahol a nullhipotézis az, hogy a vádlott ártatlan, de valójában bűnös, de mi nem vetjük el a nullhipotézist, akkor felmentünk egy bűnözőt. Vagy esetünkben a generátorról azt mondtuk, hogy megfelelt, pedig a valóság nem ez. Az 1. típusú hiba sokkal kevesebb problémát hoz magával, hiszen akkor egy megfelelő generátorra mondjuk azt, hogy nem megfelelő. Ezt későbbi tesztekkel még helyre lehet hozni.

A hipotézisvizsgálat kimenetelét (a döntést) meg lehet határozni az úgynevezett p-érték és az α összehasonlításából is. A p-érték 0 és 1 között mozog és azt adja meg, hogy a nullhipotézis mellett felsorakoztatott érveink mennyire erősek. Más szavakkal: azt a valószínűséget adja meg, hogy legalább annyira extrém teszteredményeket érünk el, mintha a teszteket úgy hajtottuk volna végre, hogy a nullhipotézist igaznak véltük. Ez az érték annál kisebb, minél távolabb kerülünk a nullhipotézisből eredő eredményektől. Ha a p-érték kisebb, mint α, akkor nem fogadjuk el a nullhipotézist, míg, ha magasabb annál akkor elfogadjuk. Tehát egy generátornál 1-hez közeli p-érték az jelenti, hogy egy teljesen véletlenszám-generátor kimentétől csak nagyon kis mértékben tér el az övé, azaz teljesen jó minőségű véletlenszám-generátornak mondható.

Természetesen sok ilyen teszt létezik és ezeket általában különféle csoportokba szokták szervezni és egymás után végrehajtani őket a tesztelendő generátoron (egy tesztet akár többször is egymás után). Sok ilyen tesztcsomag létezik, az egyik leghíresebb az Amerikai Egyesült Államokban működő National Institute of Standards and Technology (NIST) által létrehozott Statistical Test Suite (STS) egy 15 tesztből álló gyűjtemény, amit azért adtak ki, hogy bárki tudja tesztelni a véletlenszám-generátorát. A másik pedig a Dieharder (illetve ennek elődje a Diehard) tesztcsomag, amiről a következő fejezet beszél részletesebben.

## 3.2 A Dieharder tesztek

A Dieharder tesztcsomag elődjének tekinthető Diehard teszteket (12 db) 1995-ben adta ki George Marsaglia egy CD-n, amin mellékelt hozzájuk nagyobb mennyiségű véletlen számot is. A Dieharder csomag tartalmazza ezen teszteket, a korábban említett NIST STS-nek bizonyos tesztjeit, illetve még több mások által készített tesztet is. Népszerűségét annak köszönheti, hogy sok jó megbízható teszt található benne, képes kezelni bemeneti fájlok mellett azt is, ha a tesztelendő számok egy folyamként érkeznek az általános bementén. Ezenkívül nagyon könnyen használható, mivel elérhető Linux rendszerekben a parancssorból és működése könnyen paraméterezhető a kívánt cél elérése érdekében. A dolgozatban ennek a tesztcsomagnak a tesztjeit fogjuk használni, ezért érdemes bemutatni bizonyos tesztek működését.

Az overlapping 5-permutation tesztben (vagy a Dieharder által OPERM5-nek hívott tesztben) 1 millió 32 bit-es számot vizsgálunk meg. A 32-bites számokat ötösével

vesszük és megnézzük, hogy milyen sorrendben vannak. A 120 lehetséges sorrend közül mindegyiknek egyenletesen kellene megjelennie.

A 32x32 binary rank teszt megvalósítása következő: a 32 véletlenül kiválasztott egész számból kivesszük a 32 legbaloldalibb biteket és mátrixokat építünk belőlük, majd ezeknek a mátrixnak kiszámoljuk a rangját és megvizsgáljuk a megtalált rangok számát és összevetjük a lehetséges értékekkel.

## 3.3 Extraktorok

Az előző fejezetekben elhangzott, hogy sok olyan forrás van, amit fel lehet használni egy véletlenszám-generátor elkészítéséhez. Bár ezek a források jó alapot biztosítanak egy generátorhoz, nem lehet azt mondani, hogy az őket használó architektúra kimenetén megjelenő számok magától értetődően teljesen véletlenek lesznek. A bitek közötti korreláció vagy a forrás alacsony entrópiája azt vonja maga után, hogy a korábban említett teszteken nem fog megfelelni a generátor. Az ilyesfajta hibák jöhetnek magától a forrástól - ha esetleg nem megfelelőt választottunk -, de a még magas entrópiával bíró számítások alapján tökéletes forrást használó generátor is elbukhat a teszteken, ha a használt hardver (pl. a szenzorok) nem megfelelőek és elrontják a kimenetet. Az említett tesztekkel az ilyen hibák észlelhetőek, de ez nem elég. Az a cél, hogy egy olyan algoritmust használjunk, amely a generátor által előállított számokat használja fel és a segítségükkel egy olyan számsorozatot (vagy másik nézetben bitsorozatot) állít elő, amelyben nem szerepel korreláció és megközelíti az eredeti forrás entrópiáját. Ezek az algoritmusok az extraktorok [4, 16].

A generátor (illetve a generátor által előállított számok) minőségének mérésénél azt a kijelentést tettük, hogy akkor tekinthető egy generátor teljesen véletlennek, ha erősen közelít a teljesen véletlen forráshoz. Ez azt jelenti, hogy a forrás nem különböztethető meg az egyenletes eloszlástól, $U$-tól, vagy nagyon kis mértékben tér el tőle. Ezt az eltérést valahogyan mérhetővé kell tennünk, ezért bevezetjük a következő jelölést:

$$d(\mathrm{X,Y}) = \max_{a \in A} |P_X(a) - P_Y(a)|,$$

ahol X és Y két valószínűségi változó ugyanazon az A eseménytér felett. Tehát a legnagyobb eltérését keressük ugyanazon esemény valószínűségei között. Ebből a képletből megadható, hogyan tudjuk mérni az U egyenletes eloszlástól vett távolságot. Ez pedig a következő:

$$d(\mathrm{X}, \mathrm{U}) = \max_{a \in A} |P_X(a) - P_U(a)| \leq \varepsilon.$$

Ezt úgy tudjuk megfogalmazni, hogy X ε-uniform. Ezzel tehát meg tudunk adni egy olyan határt az egyenletes eloszlástól vett eltávolodásra, amit még elfogadhatónak tartunk.

Egy másik fontos kérdés az extraktoroknál, hogy mi történik a forrás entrópiájával miután alkalmaztuk az algoritmust. Ahhoz, hogy ezt meg tudjuk mondani valahogy meg kell tudunk mérni az entrópiát. Entrópiából több fajta is létezik pl. a Shannon-féle entrópia, de az extraktoroknál a min-entrópiát szokták használni. A min-entrópia kiszámítása a következő:

$$H_\infty(X) = \min_x \left\{ \log\left( \frac{1}{P(X = x)} \right) \right\}$$

Ahol a *log* a kettes alapú logaritmust jelenti. Ha szeretnénk egy *m* hosszúságú ε-uniform sorozatot kinyerni a forrásból, akkor az entrópiának legalább *m*-nek kell lennie, mivel a fenti képletet használva az egyenletes eloszlásnál az összes esemény valószínűsége egyenként $2^{-m}$, ha *m* lehetséges esemény van, ezáltal a min-entrópia *m* lesz. Ezért a forrásra kiszámolt min-entrópiát szeretnénk megközelíteni az extraktorokkal.

### 3.3.1 A determinisztikus extraktorok

A determinisztikus extraktorok kizárólag a bemenetüket használják a működésük során. Ahhoz, hogy megadjunk egy determinisztikus extraktort, meg kell adni egy forrást (C), melynek van egy min-entrópiája. Ezen kívül meg kell adni a bemenet hosszát (*n*), a kimenet hosszát (*k*) és ennek természetesen ε-uniformnak kell lennie. Mivel ezek az extraktorok csak attól függenek, hogy milyen bemenetet kapnak, ezért könnyen használhatóak és elkészíthetőek.

### 3.3.2 A seeded extraktorok

A seeded extraktorok abban különböznek a determinisztikus társaiktól, hogy esetükben a bemenetükön a forrástól származó biteken kívül szükségük van még egy *d* hosszúságú seed-re, azaz magra. Ezt úgy lehet elképzelni, mint egy PRNG-nél a

inicializáló vektorr. Ez a seed-et az extraktor működése során használja, ezért nagyon fontos, hogy jó minőségű és megfelelő hosszú legyen. Természetesen fontos, hogy $d$-t megpróbáljuk minimalizálni, miközben a kimenet hosszát ($k$) megpróbáljuk maximalizálni.

# 4 A Műegyetemen épült generátorok tesztelése és vizsgálata utófeldolgozással

Ebben a fejezetben először szeretnénk bemutatni, hogy hogyan vizsgáltuk meg a 2. fejezeben bemutatott két generátort, milyen eredményre jutottunk, ez alapján milyen utófeldolgozási eljárásokat (extraktorokat) alkalmaztunk és ezek milyen eredményre vezettek. Ezután bemutatjuk azt a webes felületet, ahol az érdeklődök hozzájuthatnak az egyik generátor által létrehozott számokhoz és a rajta futtatott tesztek eredményét is meg tudják tekinteni.

## 4.1 A dolgozat során használt architektúra

Az extraktorok generátorokon való alkalmazását, az azt megelőző és követő teszteléséket, illetve a webes felület elkészítését egymástól függetlenül végeztük, de a folyamat nagyon hasonló. Az extraktorok hatásának tanulmányozása során a két generátor által korábban előállított számokkal dolgoztunk. A két fájl formátuma különböző volt. Míg az egyik egy szöveges fájl volt, amiben egymás után voltak megtalálhatóak a bitek számként ábrázolva, amik megjelentek generátor kimentén, addig a másik típus egy bináris fájl volt, ami tartalmazta a generátor által készített számokat.

A fájlok méretüket tekintve 1 GB-osak (a szöveges fájl esetében) és kb. 400 MB-osak (a bináris állomány esetében) voltak. A 3.2. fejezetben bemutatott Dieharder tesztcsomag Linux rendszereken elérhető, parancssori programját használtuk a számok teszteléséhez, melynél lehetőség van a tesztelendő számok forrásaként mind bináris, mint szöveges fájlt megadni. Ha a szöveges fájl mellett döntünk, akkor szükséges, hogy a fájl kövessen egy magadott struktúrát. Egy megfelelően szerkesztett fejléccel kell kezdődnie és a számok, amelyek 32 bites pozitív egész számonként vannak értelmezve a program által, egymás alatt kell, hogy szerepeljenek. Mivel a szöveges állomány nagy területet foglal el ennyi szám esetén, ezért úgy döntöttünk, hogy a bináris formátumot használjuk a bemeneti fájlok esetében. Ennek a fájlnak nem kell rendelkezni semmilyen különleges felépítéssel, csak tartalmaznia kell a biteket, amiket a program 32 bites pozitív egész számok sorozatként értelmez.

A következőkben felsorolt extraktorok az eredeti számokon dolgoztak és mindegyik előállított egy saját fájlt (binárist), amit fel lehetett használni a teszteléshez.

## 4.2 Az utófeldolgozás során használt exraktorok értékelésük

### 4.2.1 Az XOR művelet, mint extraktor

Az első extraktor, amit felhasználtunk a lehető legegyszerűbb. A kizáró vagy, másnéven XOR, egy logikai művelet, amely a számítástechnika különféle területein sokszínű használatnak örvend. Legyen ez a kriptográfia, a számítógépes grafika, RAID technika vagy esetünkben a véletlenszám-generátorok utófeldolgozása [17].

Az XOR az egyik legegyszerűbb logikai művelet, melynek az igazságtábláját a 4.1. táblázat szemlélteti. Látható, hogy csak akkor ad igaz értéket, ha a két bemenete különbözik. Az XOR operáció remekül használható arra, hogy a generátorban levő bias-t csökkentse, de csak akkor, ha az egyes bitek függetlenek egymástól. A bias azt jelent, hogy a generátor a kimeneti bit két lehetséges állapota közül az egyiket jobban preferálja, elfogult az irányában. Azaz például, ha a keletkező biteket egy $X$ valószínűsége változóként tekintjük, akkor annak a valószínűsége, hogy $X = 1$, vagy más formában $P(X = 1)$ nagyobb, mint az a valószínűség, hogy $X = 0$. Ekkor, ha megvizsgáljuk a kimeneti bitsorozatot, akkor azt fogjuk tapasztalni, hogy több 1-es szerepel benne, min 0-s, ami azt jelenti, hogy nem megfelelő, mivel közel egyenlő számban kell, hogy jelen legyenek. A bias-t jól lehet mérni az említett $X$ valószínűségi változó várható értékének $\frac{1}{2}$-től vett eltérésével. Az $X$ valószínűségi változó várható értékének jelölése $E(X)$ és megadható következő képlettel: $E(X) = P(X = 1)$. Ha a valószínűségi változó közel véletlen, akkor a várható értéke az $\frac{1}{2}$-t megközelíti (vagy eléri), így a bias közel 0 lesz. A bitek függetlensége azt jelenti, hogy az első bit értéke nem befolyásolja a második bit értékét.

Ha megvizsgálunk két egymástól független bitet, amelyekhez $X\ és\ Y$ valószínűségi változókat rendelünk, melyeknek a várható értéke μ és α, akkor az XOR operátort alkalmazva rajtuk a várható érték a következőképpen alakul:

$$E(X\ XOR\ Y) = \mu + \alpha - 2\mu\alpha = \frac{1}{2} - 2(\mu - \frac{1}{2})(\alpha - \frac{1}{2}).$$

Így látható, hogy $\left|E(X\ XOR\ Y) - \frac{1}{2}\right| \leq \min(\mu - \frac{1}{2}, \alpha - \frac{1}{2})$, azaz az XOR operáció tényleg csökkenti a bias-t.

Alkalmazása egy generátor kimenetén könnyen megoldható, mivel a generátor által képzett biteket kell kettesével elhelyezni a bementén és az extraktor kimenetére az operátor eredményét elhelyezni. Lehetséges, hogy több XOR operátort használunk a biteken olyan formában, hogy például két XOR kimenetét egy harmadikra kötjük. Mivel az egyik legfontosabb logikai kapuról van szó, ezért fizikai megvalósítása is könnyen elkészíthető és az egyszerűségből fakadóan gyorsan elvégezhető a művelet. Hátránya viszont, hogy a kimeneti bitek számát a felére csökkenti, így nagy mértékben befolyásolja az elérhető bitsebességet. Ha az előbb említett módon, többet használunk belőle és az első szint bemeneti bitjeinek a száma $n$, akkor a generátor kimeneti bitjeinek száma az $n$-ed részére csökken. Ezenkívül, ahogy korábban említettük, hatásfoka nem megfelelő olyan generátornál, ahol a bitek nem függetlenek egymástól.

**4.1. táblázat: Az XOR logikai művelet igazságtáblája**

| **A** | **B** | $\boldsymbol{A \oplus B}$ |
|---|---|---|
| H | H | H |
| H | I | I |
| I | H | I |
| I | I | H |

### 4.2.2 A Neumann-féle extraktor

A Neumann-féle extraktor egyike az első olyan algoritmusoknak, amelyeket kifejezetten azzal a céllal hoztak létre, hogy javítsák egy bitsorozat minőségét. Megalkotója Neumann János volt, innen is kapta a nevét [18].

Mivel ez a megoldás korai eredmény, ezért az elsődleges célja az az algoritmusnak, hogy az XOR-hoz hasonlóan csökkentse a bias-t, azaz az 1-esek és a 0-sok száma az által kiadott bitsorozatban megegyezzen. Ehhez viszont itt is fontos, hogy az egyes bitek függetlenek legyenek egymástól. Ha ez teljesül, akkor teljes mértékben képes előállítani egy olyan bitsorozatot, ami egyenletes eloszlású. Működése hasonlít az

XOR-hoz, ugyanúgy két bitet vár a bemenetén, de a kimenet előállítása más logikával történik.

**4.2. táblázat: A Neumann-féle extraktor lehetséges kimenetei**

| Bemeneti bitpár | Kimeneti bitpár |
|---|---|
| 00 | A bitek eldobása |
| 01 | 0 |
| 10 | 1 |
| 11 | A bitek eldobása |

Az algoritmusnak a lehetséges bemenetekre adott válaszai a 4.2. táblázatban láthatóak, a működése pedig a következő:

- Vizsgáljuk meg a bementi bitpárt:
    - Ha „00” vagy „11”, akkor nem lesz kimenet, a bitpárt eldobjuk
    - Ha „01” vagy „10”, akkor a kimenet a pár első bitje lesz

Ezáltal láthatjuk, hogy ha a bitek függetlenek egymástól és nem változik annak a valószínűsége, hogy a bit értéke 1 lesz ($p$), akkor a „01” és az „10” pár előfordulásának valószínűsége ugyanaz, így az előállított sorozatban is azonos valószínűségekkel fog szerepelni az 1-es és a 0-s. Vegyük például azt az esetet, amikor az algoritmus kimenete 1-es.

$$P(kimenet = 1) = P(kimenet = 1 \mid a\ bemenet\ nem\ kerül\ eldobásra)$$
$$= \frac{P(\ kimenet = 1\ \text{és}\ a\ bemenet\ nem\ kerül\ eldobásra)}{P(a\ bemenet\ nem\ kerül\ eldobásra)}$$
$$= \frac{P(bemenet = \text{"10"})}{P(bemenet = \text{"10" vagy bemenet = "01"})}$$
$$= \frac{(\frac{1}{2}+\varepsilon)(\frac{1}{2}-\varepsilon)}{\left(\frac{1}{2}+\varepsilon\right)\left(\frac{1}{2}-\varepsilon\right)+(\frac{1}{2}+\varepsilon)(\frac{1}{2}-\varepsilon)} = \frac{\frac{1}{4}-\varepsilon^2}{2(\frac{1}{4}-\varepsilon^2)} = \frac{1}{2}$$

A fenti egyenletben $\varepsilon$ a korábban említett bias-t jelenti és értéke $\frac{1}{2} - p$. Ebből látható, hogy ha a bitek tényleg függetlenek, akkor a Neumann-féle extraktor által előállított bitsorozat teljesen véletlennek tekinthető, egyenletes eloszlású.

Hasonlóan az XOR művelethez a Neumann-féle extraktor is nagy mértékben csökkenti a kimeneti bitsorozat hosszát, azaz a bitsebességet. Negyedeli az előállított biteket és bitsebességet. A kimeneti sebessége függ magától a generátortól, nem konstans, ez csökkenti hasznosságát. Mivel, ahogy említettük, ez az egyik első ilyen algoritmus, ezért a későbbi évek során a tovább gondolásai is megjelentek. Ilyen például az $N$-bites Neumman-féle extraktor, vagy az iteráló Neumann-féle extraktor, amelyeket a következő fejezetekben fogunk bemutatni.

### 4.2.3 Az iteráló Neumann-féle extraktor

Ahogy láthattuk az előző fejezetben, az eredeti Neumann-féle extraktor bár célját eléri, de közben nagy mennyiségű bitet dob el és ezáltal szignifikánsan csökkenti a kimeneti bitsebességet. Ezen gondolatmenet során született meg az iteráló Von-Neumann extraktor [19].

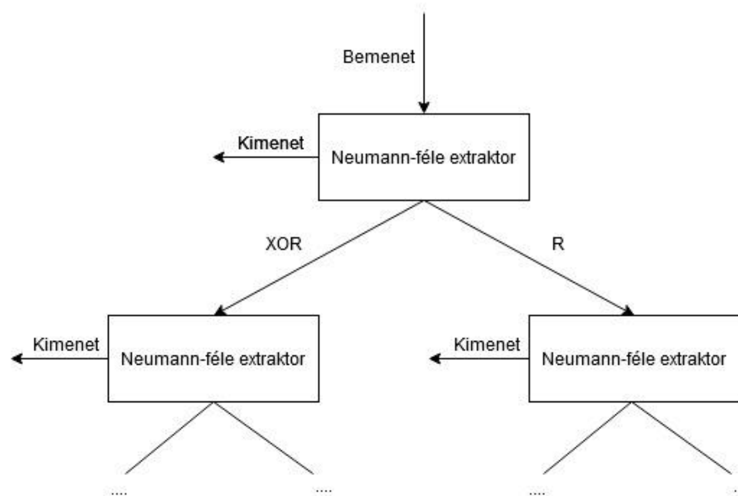
Az alapötlete ennek az algoritmusnak az, hogy a Neumann-féle extraktor működése során elvesztett biteket valamilyen módon újra fel tudjuk használni és ezáltal növeljük a kimeneti bitsebességet, pontosabban csökkentjük az arra ható negatív hatást. Ez a negatív hatás az eredeti Neumann-féle extraktornál 25%.

Az algoritmus a következőképpen működik:

1. Hajtsuk végre a bemeneti bitsorozaton a Neumann-féle extraktort és kimeneten jelenítsük meg a kapott biteket.
2. Ugyanezen a bitsorozaton alkalmazzuk az XOR és az R operátort. Az így kapott bitsorozatokat pedig használjuk fel bemenetként az első pontban.

A XOR operátor a 4.2.1. fejezetben leírtak szerint működik. Az R operátor működése pedig a következő: Ha a bemenetén a „00” bitpár jelenik meg, akkor a kimenete 0. Ha a bemenetén az „11” bitpár jelenik meg, akkor a kimenete 1. Minden más bemenetre nem állít elő kimenetet. Ahogy láthatjuk az extraktor nevében található iteráló megnevezés onnan ered, hogy ezt a folyamat többször is meg tudjuk ismételni egymás után. Ezáltal változtatva a kimenethez tartozó bias-t és a bitsebességet. Az algoritmus működését egy

adott bemeneten a 4.2. ábrán lehet megtekinteni, a működésének ábrázolás pedig a 4.1. ábrán látható.




**4.1. ábra: Az iteráló Neumann-féle extraktor működése**

Az iteráló Neumann-féle algoritmus megvalósítása még mindig könnyűnek mondható, de fontos megjegyezni, hogy a bias alakulása az R függvény hatására növekedésnek indul, bár még így is ki lehet vonni belőle bizonyos mennyiségű entrópiát.

| Az eredeti bitsorozat | 00 | 10 | 11 | 01 | 00 | 11 | 10 | 01 | 11 |
|---|---|---|---|---|---|---|---|---|---|
| Neumann-féle | | 1 | | 0 | | | 1 | 0 | |
| XOR | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| R | 0 | | 1 | | 0 | 1 | | | 1 |

**4.2. ábra: Az iteráló Neumann-féle extraktor működése egy bitsorozaton**

## 4.2.4 A N-bites Neumann-féle extraktor

Az iteráló Neumann-féle extraktorhoz hasonlóan az n-bites Neumann-féle extraktor is azon gondolat mentén lett kialakítva, hogy az eredeti algoritmus túlzottan nagy mértékben csökkenti a generátor bitsebességét [20,21].

Ahogy az extraktor nevéből következtetni lehet rá, az alapvető gondolat, amellyel növelni szeretnénk az elérhető bitsebességet az, hogy az eredeti algoritmussal ellentétben nem 2 bitet dolgozunk fel bemenetként, hanem $N$ darabot. Fontos leszögezni, hogy ahogy

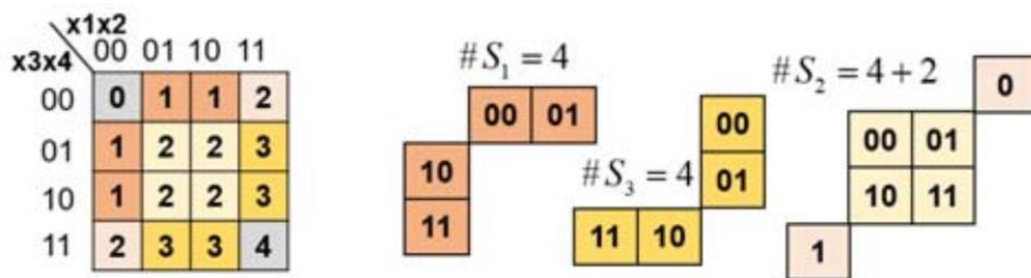
az eredeti Neumann-féle megoldásnál, úgy itt is nagyon fontos, hogy az egyes bitek egymástól függetlenek legyenek.

Az $N$-bites Neumann-féle extrakor $N$-bit hosszú bemenetének $2^N$ lehetséges permutációja lehet. Ezt a $2^N$ lehetséges permutációt $N+1$ csoportba osztjuk, melyeket azalapján hozunk létre, hogy az adott permutációban mennyi 1-es bit található. Ezeket $S_k$-val jelöljük, ahol $k$ 0-től $N$-ig terjedhet. Mivel az egyes bitek között feltételeztük, hogy nincsen függés, ezért egy csoporton belül az egyes permutációk előfordulásának valószínűsége megegyezik: $p^k q^{N-k}$, ahol $p$ annak a valószínűsége. hogy az adott bit 1-es, $q$ pedig az, hogy nem $(1-p)$. Ezért a csoportokhoz külön-külön rendelhetünk egy kódot és az így keletkező sorozatban már nem lesz bias. Az adott csoporthoz rendelt kód hossza $\log_2 C_N^k$, ahol $C_N^k$ az adott $S_k$ csoport tagjainak a száma. Az $N=4$ esetben kialakuló csoportokra és a hozzájuk rendelt kódokra a 4.3. ábrán láthatunk egy példát. A jobb oldali ábrán $x_k$ a $k.$ bitet jelenti.

Az $N$-bites Neumann-féle extraktorral elérhető elméleti bitsebesség a következőképpen alakul:

$$\sum_{k=0}^{N} C_N^k \log_2 C_N^k \, p^k q^{N-k} / N \ .$$

Az $N$-bites Neumann-féle extraktorok segítségével nagy mértékben lehet csökkenti az eredeti algoritmus bitsebességre gyakorolt hatását amellett, hogy megtartjuk az annak kedvező tulajdonságait. A 4-bites eset tényleges fizikai megvalósításánál a kimeneti sebességre gyakorolt hatás érték elérheti a 40,6%-ot, de $N$ növelésével a 90% is megközelíthető.




**4.3. ábra: Jobbra: A kialakított csoportok [21] Balra: A csoportokhoz tartozó kódok [21]**

### 4.2.5 A H-függvény

A korábban bemutattotakhoz hasonlóan ez az algoritmus is alacsony szintű, logikai kapukkal könnyen megvalósítható, de mégis képes nagy mértékben csökkenteni a bias-t egy véletlenszám-generátor esetében. A H-függvényt Markus Dichtl dolgozta ki [22], munkáját többek között egy másik extraktor, az E-függvénye, hibái inspirálták.

Az algoritmus, hasonlóan a korábban megismertekhez, a generátorban fellépő bias-t szeretné csökkenti. Fontos, hogy a bementként használt generátorban az egyes bitek függetlenek legyenek. A tervezés során fontos kikötés volt, hogy nem szabad az algoritmusnak attól a $p$ valószínűségtól függenie, amely azt adja meg, hogy mekkora valószínűséggel lesz a kimeneti bit 1-es. A másik fontos cél az volt, hogy az algoritmus magas entrópiájú bájtokat állítson elő a kimenetén.

Bemenetként a H-függvény 16 bitet vár és ebből 8 bitet állít elő a kimenetén. Ahogy láthatjuk tehát, a bitsebesség felére csökken a H-függvényt használva. A bemenet bitjeit $a_0, a_{1,}, a_2, \dots, a_{15}$-ként hivatkozzuk a későbbiekben. Az első lépés azelőtt, hogy elkészítenénk a kimeneti 8 bitet az, hogy ebből a 16 bitből két 8-as csoportot képzünk a következő módon:

- Az első csoport a bemenet első 8 bitje, azaz $a_0 - a_7$
- A második csoport bitjeit, $b_0 - b_7$, a következőképpen állítjuk el:

  $b_i = a_i \, XOR \, a_{i+1 \, mod \, 8}$ .

Fontos megjegyezni, hogy a $b_i$ bitek előállítása nem bijektív folyamat, hiszen az XOR-t használjuk és csak 128 lehetséges értéket vehetnek fel.

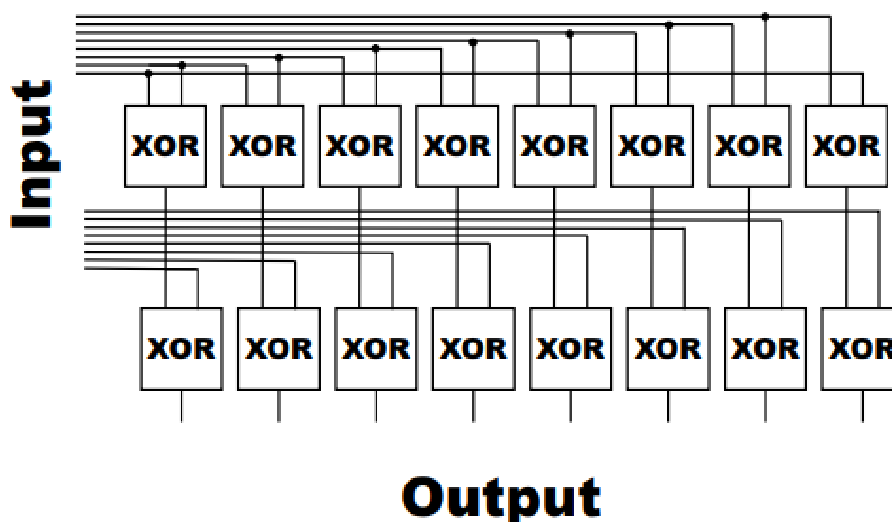
Ezután történik a kimenetként használt 8 bit elkészítése, melyeket $c_0, c_1, \dots, c_7$ -ként fogunk nevezni. Az $i$. bit előállítása a következőképpen történik:

$$c_i = b_i \, XOR \, a_{i+8} .$$

Ezt a függvényt nevezzük H-függvények. Ha szeretnénk az előkészítést is belevenni a működésebe és az egész folyamatot leírni pseudo-kód jelleggel, akkor következő leírást adhatjuk az extraktorra:

$$H(a_1, a_2) = \left(a_1 \, XOR \, elforgatásbalra(a_1, 1)\right) XOR \, a_2 .$$

Ahol $a_1$ és $a_2$ a bemeneti 16 bit első és második 8 bitjét jelentik, az $elforgatásbalra(a_1, 1)$ pedig a $a_1$ elforgatását jelenti balra 1-gyel oly módon, hogy a forgatás során kiesett bitek megjelennek a szám jobb oldalán. A H-függvény logikai kapukkal való realizációját a 4.4. ábrán láthatjuk.




**4.4. ábra: A H-függvény megvalósítása logikai kapukkal [22]**

Fontos utána járni a H-függvény esetében annak, hogy az XOR-hoz képest milyen mértékben tudja befolyásolni a bias változását. Ha a korábban kimondott bitek között függetlenségből indulunk ki, akkor annak a valószínűséget, hogy $n$ darab 1-es bitet tartalmazó bájtot kapunk $(\frac{1}{2}+\varepsilon)^n(\frac{1}{2}-\varepsilon)^{8-n}$, ahol $\varepsilon$ maga a bias, azaz $p-\frac{1}{2}$. Ha kiszámoljuk ezeket a valószínűségeket $n$ lehetséges értékeire, akkor utána megkaphatjuk annak a valószínűséget, hogy az XOR operáció kimenteinek mekkora a valószínűség két 8 bites bemenetre. Például $n=0$ (azaz a csupa nulla XOR kimenet) esetén ez:

$$\frac{1}{256}-\frac{\varepsilon^2}{8}+\frac{7\varepsilon^4}{4}-14\varepsilon^6+70\varepsilon^8-224\varepsilon^{10}+448\varepsilon^{12}-512\varepsilon^{14}+256\varepsilon^{16}.$$

Mivel $\varepsilon$ értéke egy generátor esetében kicsi, ezért az alacsony kitevőn lévő $\varepsilon$ hatványok befolyásolják a legjobban az eredményt.

Ha kiszámoljuk a H-függvény egyik kimenetének valószínűséget, akkor ezt az értéket kapjuk:

$$\frac{1}{256}+\frac{\varepsilon^3}{16}-\frac{\varepsilon^4}{4}-\frac{3\varepsilon^5}{4}+\frac{\varepsilon^6}{2}+3\varepsilon^7+3\varepsilon^8-4\varepsilon^9-8\varepsilon^{10}.$$

Amiben eltűnt a négyzetes tag, így elmondható, hogy a H-függvény jobban teljesít az XOR-nál. A H-függvény tovább lehet finomítani annak érdekében, hogy a harmadik, negyedik és további hatványok eltűnjenek, ezáltal javítva az algoritmust.

### 4.2.6 Az S-boxot használó extraktor

Ennek az extrakornak az alapötlete az, hogy egy már meglévő, máshol működő megoldást felhasználva készüljön el egy olyan algoritmus, amely használható extrakcióra. Ezt a algoritmust két török kutató készítette el [23].

Az extraktor alapvető építőeleme az S-boksz, amelyet kriptográfia megoldásokban szoktak használni. Neve az angol substitution szóból származik, ami helyettesítést jelent. Az S-bokszot a szimmetrikus kulcsú titkosítás során szokták használni azzal a céllal, hogy a titkosítás során felhasznált kulcs és a titkosított szöveg közötti kapcsolatot elfedjük. Ha az S-bokszra mint függvényre tekintünk, akkor elmondhatjuk, hogy bementként egy $n$ bites számot kap, amelyből egy $m$ bites számot állít elő. A helyettesítés abban merül ki, hogy a lehetséges $2^n$ bemenethez megfelel egy kimenet. A bement és kimenet hossza eltérő is lehet. Az S-boksz egyik lehetséges reprezentációja egy táblázat, melyben megfelelő cellát a bement segítségével lehet kiválasztani és tartalmazza a megfelelő kimenetet.

Ahhoz, hogy egy megfelelő minőségű S-bokszot készítsünk el meg kell felelni bizonyos követelményeknek. Az első ilyen követelmény a szigorú lavina, amely kimondja, hogy a bementek egyetlen bitjének megváltozása a kimeneti bitek legalább a felének megváltozásával kell, hogy járnia. A második a bitek függetlensége, mely hasonlóan korábbiakhoz azt várja el, hogy egy bit értékeinek valószínűsége ne függjön a többi bittől. A harmadik követelmény az, hogy a S-boksz ne legyen lineáris. Ezt azt jelenti, hogy például két bemenet összegének a kimenete nem egyezik meg az egyes bemenetekre külön-külön kiadott kimenetek összegével. Az utolsó követelmény a S-boksz felé, hogy kiegyenlített legyen. Ez követelmény azt várja el, hogy magát a bokszot alkotó vektorok egyenlő számban tartalmazzanak 1-es és 0-s biteket. Minden követelmény nem lehet egyszerre teljesíteni, mivel van olyan, amely ellentmond egy másiknak.

Az egyik leghíresebb szimmetrikus kulcsú titkosítást megvalósító rendszer, amely S-bokszokat használ az a DES és az AES algoritmus. A DES esetében fix bokszokat

használnak, mely azt jelenti, hogy azok már előre el vannak készítve és nem változnak az algoritmus működése során. Itt szám szerint 8 S-bokszot készítettek el, melyek 6 bites bemeneteket képeznek le 4-bites kimenetekre. Az $S_5$ jelzésű bokszot a 4.3. táblázatban lehet látni. Van olyan alkalmazásuk is a bokszoknak, ahol kulcs alapján dinamikusan változnak, ezáltal növelve a hatásukat.

**4.3. táblázat: A DES-nél használt ötös számú S-boksz**

| **$S_5$** | | A középső 4 bit | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | **0000** | **0001** | **0010** | **0011** | **0100** | **0101** | **0110** | **0111** | **1000** | **1001** | **1010** | **1011** | **1100** | **1101** | **1110** | **1111** |
| A két szélső bit | **00** | 0010 | 1100 | 0100 | 0001 | 0111 | 1010 | 1011 | 0110 | 1000 | 0101 | 0011 | 1111 | 1101 | 0000 | 1110 | 1001 |
| | **01** | 1110 | 1011 | 0010 | 1100 | 0100 | 0111 | 1101 | 0001 | 0101 | 0000 | 1111 | 1010 | 0011 | 1001 | 1000 | 0110 |
| | **10** | 0100 | 0010 | 0001 | 1011 | 1010 | 1101 | 0111 | 1000 | 1111 | 1001 | 1100 | 0101 | 0110 | 0011 | 0000 | 1110 |
| | **11** | 1011 | 1000 | 1100 | 0111 | 0001 | 1110 | 0010 | 1101 | 0110 | 1111 | 0000 | 1001 | 1010 | 0100 | 0101 | 0011 |

Az extrakor a DES-ben használt S-bokszokat fogja használni és működése hasonlít az ott használt folyamatokhoz. Bemeneteként az algoritmus 48 bitet vár, amiből egy 32 bites kimenetet állít elő, melyből látható, hogy $\frac{1}{3}$-os csökkenést jelent. Ez kevesebb, mint a korábban bemutatott extraktorok esetében, így ez az algoritmus kevésbé csökkenti a bitsebességet. Az extraktornak a működése a következő:

1. A 48 bites bemenetet osszuk fel 8 darab 6 bites csoportra. A csoport sorszáma eldönti, hogy melyik sorszámú S-bokszot fogjuk használni az algoritmusnál.
2. A 6 bites csoport két szélső (az első és az utolsó) bitje eldönti, hogy melyik sort kell használnunk az S-bokszból.
3. A középső 4 bit kijelöli, hogy melyik oszlopot fogjuk használni.
4. A kijelölt sor és oszlop eldönti, hogy melyik cella tartozik az adott bitcsoporthoz és a cellában található érték fogja képezni a kimenet 4 bitjét.
5. Ha van még 6 bites csoport, akkor lépjünk a következőre, ha nincs akkor az így keletkezett 8 darab 4 bites érték lesz az extraktor kimenete.

Így, ha például a bemenet 5. 6 bites csoportja az „100011” sorozat, akkor a 3. táblázatban látható $S_5$ boksz 3-es sorának (fentről a negyedik, mivel 0-tól kezdődik a számozás) és 1-es oszlopának (balról a második, szintén a 0-tól kezdődő számozás miatt) a metszetében

lévő cella fogja tartalmazni, hogy mi kerül a kimenet 5. 4 bites csoportjába. Ebben az esetben ez a „1000” bitsorozat, azaz a 8 lesz.

Összeségében elmondható, hogy az S-bokszok felhasználása a extraktorokként egy jó minőségű megoldást jelent, ami egy korábban több kriptográfia rendszerben kipróbált megoldást használ és kisebb méretekben csökkenti a végső bitsebességet, mint az előző fejezetekben felsorolt társai.

### 4.2.7 A Toeplitz-mátrixot használó extraktor

Ez az extraktor nevéből adódóan Toeplitz-mátrixot fog használni az utófeldolgozás során. Ezen mátrixok fontos helyet foglalnak el a lineáris algebrában, mivel azok az egyenletrendszerek, amelyek megadhatóak egy Toeplitz-mátrix segítségével könnyebben megoldhatóak mint egy általános egyenletrendszer.

A Toeplitz-mátrixok olyan mátrixok, melyeknek az átlóikban (a bal fentről jobb lentiig) ugyanaz a szám található. Egy ilyen mátrixra példa a következő:

$$\begin{matrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{matrix}$$

A Toeplitz-mátrixokat használó extraktor bemenetének hossza a használt mátrixtól függ. Az algoritmus működése egyszerűen megadható. A bemeneti bitsorozatot bináris vektorként értelmezzük és a lineáris algebra szabályait követve megszorozzuk a mátrixot a bemeneti vektorral (természetes modulo 2-n értelmezve a műveleteket). Az így kapott vektor lesz az algoritmus kimenete. Bár az algoritmus alapgondolata egyszűrnek tűnhet, de a mátrix méretének és a benne található értékek megfelelő megválasztásával jó minőségű számok állíthatók elő. Ha a mátrix méretéről már döntöttünk, akkor a tartalmával kapcsolatban, hasonlóan az S-bokszokhoz, két választásunk lehet. Ha úgy döntünk, hogy fix értékeket használunk, akkor az algoritmus teljes működése során ugyanazok az értékek fogják benépesíteni a mátrixot, de úgy is dönthetünk, hogy minden egyes bemeneti vektor után megváltoztatjuk a mátrix tartalmát például úgy, hogy egy LFSR (Linear Feed Forward Shift Register) adagolja a mátrixot felépítő számokat.

Megvalósítás szempontjából jó helyet foglal el a Toeplitz-mátrixot használó megoldás, mivel a mátrixműveletek elvégzésre hatékony hardvereket lehet építeni és a

Toeplitz-mátrix tulajdonságai alapján ahhoz, hogy tároljuk a mátrix elemeit nem szükséges az összeset értéket ismerni, hanem elég csak az első sort és az első oszlopot (pontosabban, az első két elem itt megegyezik, így azt nem kell kétszer).

### 4.2.8 A hash függvények mint extraktorok

A hash függvények sokszínű alkalmazásnak örvendenek a számítástechnika különböző területein. Számos helyen alkalmazzák őket a kriptográfiától az adattárolásig. Előnyös tulajdonságaik miatt extraktorként is kifejezetten jól használhatóak.

A hash függvények determinisztikus függvények, a bemenetükön megkapnak egy sorozatot, amihez kimenetükön kiadnak egy hasht-t vagy másnéven lenyomatot, ami adott bemenetre mindig ugyanaz. A bemenet hossza természetesen véges és a kimenet és a bemenet hossza között lehet eltérés. Ahhoz, hogy egy adott függvényt hash függvénynek nevezzük meg kell felelnie bizonyos kritériumoknak. Az első követelmény az első őskép elleni védelem. Ez alatt azt értjük, hogy ha adott egy lenyomat, amelyről nem tudjuk, hogy mely bemenet alapján készült, akkor csak a lenyomat birtokában nehéz legyen kiszámolni azt, hogy mi volt a tényleges bemenet. A második követelmény a második őskép elleni védelem. Ez a kritérium azt mondja ki, hogy ha a birtokunkban van egy bemenet $b_1$, akkor nehéz találni egy másik bemenetet $b_2$-t, hogy a két bemenet lenyomata megegyezzen, azaz $hash(b_1) = hash(b_2)$. A harmadik az ütközés elleni védelem. Ez a követelmény az jelenti, hogy nehéz találni két olyan bemenetet, melynek a lenyomata megegyezik. Ez az előző követelmény erősebb változata. Természetesen, mivel a kimenet hosszából adódóan véges számú a lehetséges érétkek halmaza, ezért van lehetőség arra, hogy két bemenetnek a lenyomat megegyezik. A következő követelmény az S-bokszoknál már említett lavinával kapcsolatos. A hash függvényeknél is fennáll az az elvárás, hogy a bemeneten egy bit megváltoztatása a lenyomat olyan mértékű megváltozását igényli, amiből már nem lehet kiszámolni, hogy a két bemenet között csak egy bit eltérés volt.

A imént felsorolt tulajdonságok alapján a hash függvények jól használhatók extrakciós célokra. A lavina-hatás következményeként, ha a bemenetre két kis mértékben eltérő bitsorozat érkezik, akkor a hozzájuk tartozó lenyomatok nagy mértékben el fognak térni. Mivel a kimenet hossza akár rövidebb is lehet bemenetnél, ezért ekkor tömörítés is történi, ami a bitsebesség csökkenéséhez is vezethet.

A hash függvények fizikai extraktorként való használatát nagyban segíti, hogy ahogy a fejezet elején említettük számos kriptográfia protokoll használja őket, ezért fontos, hogy számításük hatékony legyen. Ezért sok készülékben (pl.: okostelefonok vagy számítógépek) megtalálható egy dedikált hardver arra célra, hogy lenyomatokat számoljon. Így használtuk hatékonyan megoldható.

## 4.3 A generátorok tesztelése

A generátorok teszteléséhez az elsődleges feladat az volt, hogy az előző fejezetekben bemutatott extraktoroknak elkészítsük az implementációját. A megvalósított extraktorok Python nyelvben íródtak, a fájlkezelést is ennek a nyelvnek a segítségével valósítottuk meg. Az extraktorok implementációja után a következő feladat a tesztelés lebonyolítása, automatizálása és ezen tesztek eredményeinek kiértékelése volt. Teszteket a korábban már említett Dieharder tesztcsomaghoz tartozó Linux környezetben használható dieharder parancssori program segítségével végeztük, az eredmények összegyűjtése és kiértékelése pedig szintén a Python nyelv segítségével történt.

A következőkben bemutatjuk, hogy az egyes extrakorok milyen eredményt értek el a két generátor által előállított számokon összevetve az utófeldolgozás nélküli és a többi extrakor által létrehozott számokkal. Fontos megjegyezni, hogy a két generátor által előállított számok, amelyeket használtunk, eltérő hosszúságúak voltak. A másik fontos információ, hogy az iteráló Neumann-féle extrakor esetében csak 1 iterációt használtunk, az N-bites Neumann-féle extraktor esetében a bemutatott 4 bites változatot implementáltuk, a Toeplitz-mátrixot használó extraktor esetében egy fix értékű mátrixot használtunk, melynek a tartalmát egy PRNG-vel generáltuk és a hash függvények közül pedig a SHA256-ot használtuk.

A tesztelés során fontos kérdés volt, hogy a Dieharder tesztcsomagból mely tesztek használjuk. A tesztcsomagban jelen állapotban 30 teszt található, melyek változó mennyiségű számot várnak és futási idejük is nagyban eltérő. A megfelelő tesztek kiválasztásának folyamatában fontos szempont volt, hogy a két generátor által előállított számok mennyisége eltérő volt tesztelés során. A beérkezési időn alapuló generátor esetében kb. 400 MB-nyi szám állt rendelkezésre, míg a spontán emisszión alapuló generátortól kb. 120 MB-nyi számot használtunk (Ennek a fájlnak a szöveges változat volt kb. 1 GB). Ezenkívül szem előtt tartottuk, hogy egyrészt a választott tesztek

bemutassák, hogy a Dieharder tesztcsomagra jellemző teszteket, másrészt, hogy a már korábban említett NIST-féle tesztcsomagból is használjunk bizonyos teszteket. Ezen célok figyelembevételével 19 tesztet választottunk ki, amelyekből 16 a Dieharder saját tesztje és viszonylag kisebb mennyiségű bemenetet várnak, így gyorsan végre lehet őket hajtani. A maradék három teszt pedig a NIST-féle STS-ből került kiválasztás és a Dieharderben tesztcsomagban megtalálhatóak.

A két generátornál táblázatos formában gyűjtöttök össze az eredményeket, melyek tartalmazzák az első oszlopukban az egyes tesztek neveit, a további oszlopokban pedig a források és az adott teszten elért eredményük tekinthető meg. A teszt eredménye kétfajta lehet: PASSED, ez azt jelenti, hogy átment a teszten az adott bemenet, ekkor szerepel a teszthez számított p-érték is a táblázatban. A másik lehetőség a FAILED, ami azt jelenti, hogy a teszten nem ment át a bemenet.

### 4.3.1 A spontán emisszión alapuló generátor vizsgálata

**4.4. táblázat: A spontán emisszión alapuló generátor tesztelése 1. rész**

| **Teszt neve** | **Eredeti** | **XOR** | **Neumann-féle** | **Iteráló Neumann-féle** | **4 bites Neumann-féle** |
|---|---|---|---|---|---|
| *diehard_birthdays* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_operm5* | FAILED | PASSED (0.07985303) | FAILED | FAILED | PASSED (0.07837971) |
| *diehard_rank_32x32* | PASSED (0.5654664) | PASSED (0.04205843) | PASSED (0.03589123) | PASSED (0.47921886) | PASSED (0.21459865) |
| *diehard_rank_6x8* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_bitstream* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_opso* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_oqso* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_dna* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_count_1s_str* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_count_1s_byt* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_parking_lot* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_2dsphere* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_3dsphere* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_squeeze* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *diehard_runs* | FAILED | PASSED (0.63112649) | PASSED (0.766932065) | PASSED (0.3633577500) | PASSED (0.617240975) |
| *diehard_craps* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *sts_monobit* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *sts_runs* | FAILED | FAILED | FAILED | FAILED | FAILED |
| *sts_serial* | FAILED | FAILED | FAILED | FAILED | FAILED |

A spontán emisszión alapuló generátor (ASE) által létrehozott számoknál fontos megemlíteni, hogy az elkészítésük során felülmintavételezés történt, a bitsebesség növelése céljából. Ennek következtében sokkal rosszabb minőségű számok keletkeztek, mintha a megfelelő mintavételi frekvencia mellett történt volna az előállításuk.

A 4.4. táblázatban láthatjuk az extraktorok első csoportját és magát az eredeti fájlt is. Ahogy láthatjuk a felülmintavételezés nagy hatást gyakorolt a számok minőségére. Az eredeti, utófeldolgozás nélküli számok csak egy teszten mentek át. Megfigyelhetjük továbbá, hogy az első csoportba tartozó, egyszerűbb extraktorok nem voltak képesek nagy mértékben javítani a számok minőségét. Ennek oka az, hogy ezen extraktorok az eredeti bitek mennyiségét nagy mértékben csökkentik, így a tesztekhez ajánlott mértet nem minden esetben fogják elérni a kimeneteiken megjelenő bitsorozatok.

**4.5. táblázat: A spontán emisszión alapuló generátor tesztelése 2. rész**

| **Teszt neve** | **H-függvény** | **S-boksz** | **Toeplitz-matrix** | **SHA256** |
|---|---|---|---|---|
| *diehard_birthdays* | PASSED (0.94641277) | PASSED (0.13644905) | FAILED | FAILED |
| *diehard_operm5* | PASSED (0.18692743) | PASSED (0.35893748) | FAILED | PASSED (0.41383291) |
| *diehard_rank_32x32* | PASSED (0.46696257) | PASSED (0.10086222) | FAILED | PASSED (0.86138959) |
| *diehard_rank_6x8* | PASSED (0.41878989) | FAILED | FAILED | PASSED (0.49746324) |
| *diehard_bitstream* | FAILED | FAILED | FAILED | PASSED (0.82243865) |
| *diehard_opso* | FAILED | FAILED | FAILED | PASSED (0.23104535) |
| *diehard_oqso* | PASSED (0.00975851) | FAILED | FAILED | FAILED |
| *diehard_dna* | PASSED (0.03534096) | FAILED | FAILED | PASSED (0.38228438) |
| *diehard_count_1s_str* | FAILED | FAILED | FAILED | PASSED (0.36054751) |
| *diehard_count_1s_byt* | PASSED (0.19563835) | FAILED | FAILED | PASSED (0.40564059) |
| *diehard_parking_lot* | PASSED (0.06065386) | FAILED | FAILED | PASSED (0.0105223) |
| *diehard_2dsphere* | PASSED (0.87195165) | FAILED | FAILED | PASSED (0.56409577) |
| *diehard_3dsphere* | PASSED (0.84350498) | PASSED (0.09747995) | FAILED | PASSED (0.82336762) |
| *diehard_squeeze* | FAILED | FAILED | FAILED | PASSED (0.9536816) |
| *diehard_runs* | FAILED | PASSED (0.474623119) | PASSED (0.52170489) | PASSED (0.19764922) |
| *diehard_craps* | FAILED | FAILED | FAILED | FAILED |
| *sts_monobit* | FAILED | FAILED | FAILED | PASSED (0.06209802) |
| *sts_runs* | FAILED | FAILED | FAILED | PASSED (0.32245618) |
| *sts_serial* | FAILED | FAILED | FAILED | PASSED (0.562838076) |

Az 4.5. táblázatban tekinthetjük meg az extraktorok második csoportjának eredményeit. Az első csoporttal ellentétben itt már olyan extraktorok szerepelnek, melyek

megvalósítása bonyolultabb vagy egy olyan már létező megoldást használnak, mely jól alkalmazható az utófeldolgozás során. Ahogy azt a H-függvényt bemutató fejezetben részleteztük, az XOR-nél jobb eredményt tud elérni, mivel a bias kisebb hatványokon szerepel az egyes kimenetek valószínűségében. Ezt a megállapítást nyomon lehet követni a tesztek eredményében is. Egy fontos adat, ami leolvasható a táblázatból, hogy a hash függvények (esetünkben az SHA256) nagyon jól használhatóak extraktorként a korábban említett tulajdonságaik alapján..

### 4.3.2 A beérkezési időn alapuló generátor vizsgálata

**4.6. táblázat: A beérkezési időn alapuló generátor tesztelése 1. rész**

| Teszt neve | Eredeti | XOR | Neumann-féle | Iteráló Neumann-féle | 4 bites Neumann-féle |
|---|---|---|---|---|---|
| *diehard_birthdays* | PASSED (0.56380095) | PASSED (0.80955637) | PASSED (0.53579783) | PASSED (0.09976293) | PASSED (0.85316244) |
| *diehard_operm5* | PASSED (0.95248039) | PASSED (0.16290198) | PASSED (0.06569084) | PASSED (0.46296909) | PASSED (0.35997334) |
| *diehard_rank_32x32* | PASSED (0.31256319) | PASSED (0.94823215) | PASSED (0.57589196) | PASSED (0.3269735) | PASSED (0.91692869) |
| *diehard_rank_6x8* | PASSED (0.12138697) | PASSED (0.815641) | PASSED (0.60621254) | PASSED (0.88352827) | FAILED |
| *diehard_bitstream* | PASSED (0.30524513) | PASSED (0.12084341) | PASSED (0.52386711) | PASSED (0.29699942) | FAILED |
| *diehard_opso* | FAILED | PASSED (0.44808817) | PASSED (0.22856026) | PASSED (0.06781771) | FAILED |
| *diehard_oqso* | PASSED (0.92727833) | PASSED (0.6905901) | PASSED (0.98686054) | PASSED (0.1342607) | FAILED |
| *diehard_dna* | PASSED (0.54914012) | PASSED (0.60171674) | PASSED (0.97191448) | PASSED (0.53275766) | FAILED |
| *diehard_count_1s_str* | PASSED (0.92717366) | PASSED (0.9761014) | PASSED (0.93533938) | PASSED (0.54008877) | FAILED |
| *diehard_count_1s_byt* | PASSED (0.94124367) | PASSED (0.87502784) | PASSED (0.67379598) | PASSED (0.82126527) | FAILED |
| *diehard_parking_lot* | PASSED (0.86257036) | PASSED (0.10028111) | PASSED (0.27343195) | PASSED (0.01230735) | FAILED |
| *diehard_2dsphere* | PASSED (0.57568022) | PASSED (0.33609543) | PASSED (0.75439102) | FAILED | FAILED |
| *diehard_3dsphere* | PASSED (0.57416686) | PASSED (0.98169563) | PASSED (0.5750182) | PASSED (0.03108173) | PASSED (0.24640272) |
| *diehard_squeeze* | PASSED (0.11427901) | PASSED (0.49774833) | PASSED (0.04321115) | PASSED (0.01315746) | FAILED |
| *diehard_runs* | PASSED (0.176144785) | PASSED (0.80503291) | PASSED (0.68403767) | PASSED (0.284208195) | PASSED (0.419221465) |
| *diehard_craps* | PASSED (0.30668977) | PASSED (0.71111101) | FAILED | PASSED (0.73704351) | FAILED |
| *sts_monobit* | PASSED (0.35956167) | FAILED | PASSED (0.76260061) | PASSED (0.24898388) | FAILED |
| *sts_runs* | FAILED | PASSED (0.89174607) | FAILED | FAILED | FAILED |
| *sts_serial* | PASSED (0.569525278) | PASSED (0.45898010) | PASSED (0.548110869) | PASSED (0.453456344) | FAILED |

A beérkezési időn alapuló generátor által létrehozott számok elkészítése során nem volt jelen felülmintavételezés és a spontán emisszióhoz képest több szám állt rendelkezésre. A generátorról szóló fejezetben bemutattuk, hogy az általa előállított számok annak működéséből fakadóan jó minőségűek, így arra számíthatunk, hogy ha nincs jelen felülmintavételezés, akkor teszteken is jól fog szerepelni a generátor.

A 4.6. táblázatban látható a generátor utófeldolgozás nélküli kimenetének eredményei illetve, az hogy az extraktorok első csoportja által előállított kimenetek hogy szerepeltek a teszteken. Megfigyelhető, hogy már az eredeti kimenet is nagyon jó szerepelt, melynek az eredményét az extraktorok nagy része bár nem rontotta el, de nem is tudta tökéletes végeredményre vinni.

**4.7. táblázat: A beérkezési időn alapuló generátor tesztelése 2.rész**

| Teszt neve | H-függvény | S-boksz | Toeplitz-matrix | SHA256 |
|---|---|---|---|---|
| *diehard_birthdays* | PASSED (0.30569157) | PASSED (0.73200915) | PASSED (0.99108387) | PASSED (0.06504442) |
| *diehard_operm5* | PASSED (0.35943401) | PASSED (0.91221516) | PASSED (0.33589583) | PASSED (0.24067412) |
| *diehard_rank_32x32* | PASSED (0.24320712) | PASSED (0.38279651) | PASSED (0.31256319) | PASSED (0.00696166) |
| *diehard_rank_6x8* | PASSED (0.30411421) | FAILED | PASSED (0.12138697) | PASSED (0.43825249) |
| *diehard_bitstream* | PASSED (0.11502841) | FAILED | PASSED (0.14495013) | PASSED (0.12008882) |
| *diehard_opso* | PASSED (0.88000537) | FAILED | PASSED (0.21949196) | PASSED (0.35419291) |
| *diehard_oqso* | PASSED (0.8580282) | FAILED | PASSED (0.52389989) | PASSED (0.41221714) |
| *diehard_dna* | PASSED (0.48249739) | PASSED (0.35056086) | PASSED (0.36789757) | PASSED (0.73359341) |
| *diehard_count_1s_str* | PASSED (0.57482847) | FAILED | PASSED (0.28002494) | PASSED (0.57678484) |
| *diehard_count_1s_byt* | PASSED (0.03680136) | FAILED | PASSED (0.52029866) | PASSED (0.28005155) |
| *diehard_parking_lot* | PASSED (0.54735549) | PASSED (0.10932071) | PASSED (0.57621075) | PASSED (0.81757364) |
| *diehard_2dsphere* | PASSED (0.78449851) | PASSED (0.12646859) | PASSED (0.3035199) | PASSED (0.79167004) |
| *diehard_3dsphere* | PASSED (0.8766563) | PASSED (0.52355523) | PASSED (0.56511482) | PASSED (0.50061305) |
| *diehard_squeeze* | PASSED (0.90470803) | FAILED | PASSED (0.34757657) | PASSED (0.27386875) |
| *diehard_runs* | PASSED (0.780351395) | PASSED (0.554690155) | PASSED (0.46266088) | PASSED (0.518447625) |
| *diehard_craps* | PASSED (0.644233265) | PASSED (0.39155297) | PASSED (0.732811945) | PASSED (0.38418817) |
| *sts_monobit* | PASSED (0.89790989) | PASSED (0.04143217) | PASSED (0.40325347) | PASSED (0.98915878) |
| *sts_runs* | PASSED (0.63195098) | FAILED | FAILED | PASSED (0.52011109) |
| *sts_serial* | PASSED (0.577786352) | FAILED | PASSED (0.4684227147) | PASSED (0.565093908) |

A 4.7. táblázatban található meg, hogy az extraktorok második csoportja által létrehozott számok hogyan teljesítettek a teszteken. Látható, hogy mind a H-függvény, mind a hash függvény képes volt olyan mértékben javítani a számokon, hogy azok megfeleljenek mind a 19 teszten. Ezenkívül megállapítható, hogy a Toeplitz-mátrix is jól teljesített.

### 4.3.3 A tesztek eredményeinek összegzése

A két generátoron végzett tesztek eredményeit megvizsgálva fontos tanulságokat vonhatunk le mind a generátorok tekintetében, mind az extraktorokra nézve. A generátorokon fontos szempont a bitsebesség, melynek minél magasabb értéke bizonyos alkalmazások részéről szigorú elvárás. A tesztek eredményeiből látható, hogy azzal, hogy növeljük a bitsebességet a generátor által előállított számok minősége romlásnak indulhat. Ilyenkor fontos, hogy használjunk utófeldolgozási eljárásokat és folyamatos teszteket, melyek biztosítják, hogy a generátor kimenetén megjelenő bitek megfelelőek lesznek. A végrehajtott tesztek alapján megállapíthatjuk, hogy akkor is érdemes használni extraktorokat, ha a tesztelt generátornak az utófeldolgozás nélküli kimenete is jó eredménykét ér el, mivel azt is lehet javítani. Végül láthatjuk, hogy az egyes extraktorok között is vannak különbségek, különböző tulajdonságokkal bírnak, így a célnak megfelelően kell kiválasztani őket.

## 4.4 A kimenet vizualizálása

A QRNG-ken alkalmazott extraktorok bemutatására létrehoztunk egy weblapot. Ezzel a weblappal több célunk is volt. Mint a korábbi fejezetekben említettük, a kvantumos témakör nem a legismertebb területe az informatikának. Hihetetlen mennyiségű felfedezni való téma tartozik ide. Az egyik célunk a weblappal, hogy közelebb hozzuk az emberekhez a kvantumrandom generátorokat, mind fizikai mind elméleti síkon. Így a weblap képes egy generátor kimenetét konstans streamelni. Másik oldalról viszont szerettük volna bemutatni az extraktorok hatását is. Ezen két gondolat találkozásából született az oldal másik funkciója, mely alapján a generált számokat lehet tesztelni a különböző dieharderes teszteseteken, mind magukban, mind különböző extraktorok alkalmazásával. A harmadik funkció, ami szerepel a weblapon, az a generátor online illetve offline állapotának a jelzése. Mivel a weblap alapjául szolgáló QRNG által előállított véletlenszámokat nem csak mi használjuk, hanem más kutatók is igénybe

veszik, , így a kimenete nem szükségszerűen lesz rákötve a weblapra, ezért szerettük volna jelezni a felhasználók felé, mikor aktív a generátor és mikor offline. Ez a tulajdonság nem kapcsolja ki a tesztelés/extraktorok opciót, hiszen a legutolsó adat vizsgálata nem áll le.

Általános esetben, egy komplexebb téma esetében, a vizualizáció nagyon sokat segít a megértésben. Ehhez hasonlóan, egy ismeretlen téma esetében, egy könnyen elérhető, figyelem felhívó formátum közelebb hozhatja a gondolatkört.

### 4.4.1 Miért jó egy webes szoftver?

A projekt tervezése során gondolkodtunk, hogy mi lenne egy ideális felület, a téma vizualizálására, de az opciók közül egy webes szoftver teljesít a legtöbbet az általunk megcélzott kritériumokból. Fontos a célunk szempontjából, hogy sok ember számára elérhető legyen, ezért esett a választásunk egy weblapra. Vannak akik specifikusan ezt vagy ehhez hasonló oldalakat, szoftvert keresnek, számukra egy bizonyos szintig indifferens az hogy mennyi erőfeszítésbe kerül neki elérnie a célt, míg valaki aki csak véletlenül akad rá, nem feltétlenül lép át egy határt hogy tovább lépjen. Erre egy jó példa egy alternatíva a sok emberhez való eljuttatáshoz egy Androidos alkalmazás. Ebben az esetben ha valaki aki amúgy is egy valódi random generátort keresett le fogja tölteni az applikációt, míg valaki aki csak rátalált, nem fogja venni a fáradtságot, hogy letöltse, mert általános esetben annyira nem érdekli a téma.

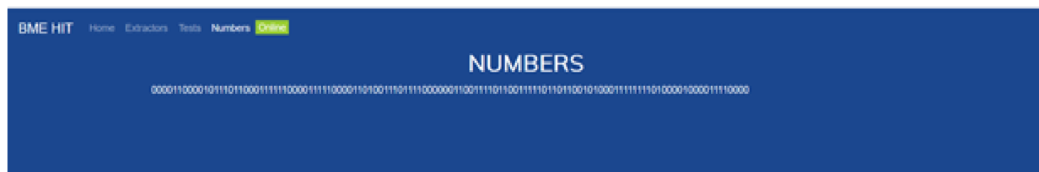
Egy letöltendő szoftvernél nagyon hasonló problémák jelentkeznek mint egy telefonos alkalmazásnál.

A weblap esetében, ha már eljut odáig a felhasználó, hogy megtalálja a linket, nagy valószínűséggel, rá is kattint. Természetesen ennek is meg vannak a veszélyei. A felhasználónak a megfelelő környezetben kell találkoznia a linkkel, viszont még így is nagyobb eséllyel látogat el az oldalra, mint ellenkező esetben tölt le egy alkalmazást.

Egy másik pont amiben jeleskedik a webes felület, a platform függetlensége. A felület mobilon való elérhetősége két fontos előnnyel rendelkezik. A plusz platform újabb lehetőséget ad, hogy megismertesse magát a felhasználókkal, valamint rendelkezik az áthelyezhetőség tulajdonságával.

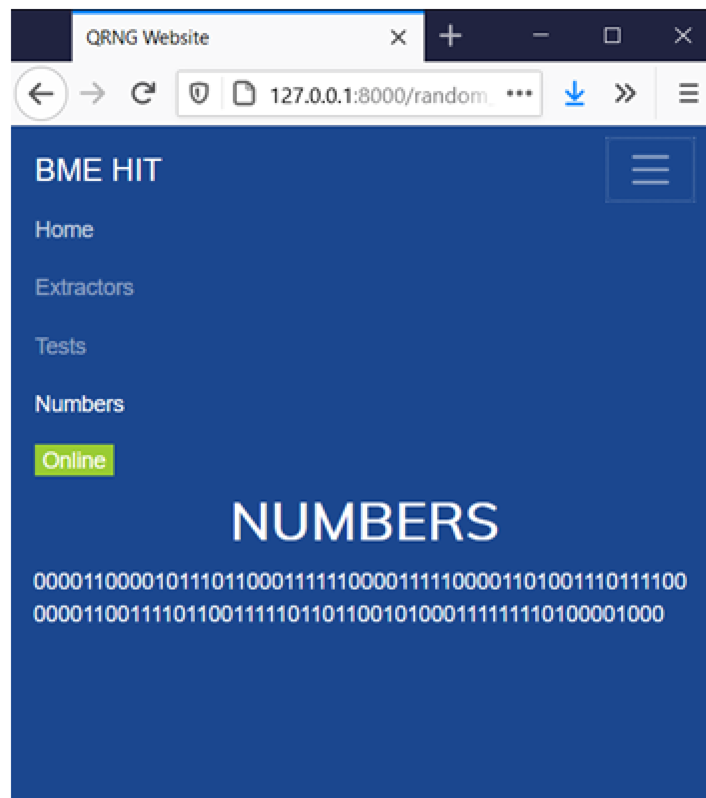
### 4.4.2 A megvalósítás

A program alapvetően egy Django alapú weblap. Sok mérlegelés után a Django keretrendszert választottuk, hiszen ez tartalmazta a legtöbb olyan lehetőséget amire nekünk szükségünk volt. A Django framework egy magas-szintű python alapú framework ami nagyban elősegíti strukturált webes alkalmazások fejlesztését. A Django egyik nagy előnye, hogy a Django magában tartalmazza az api-t is így a fájlokkal való kommunikáció nagy mértékben egyszerűbb más framewokokhoz képest.




**4.5. ábra: Az elkészített oldal működés közben**

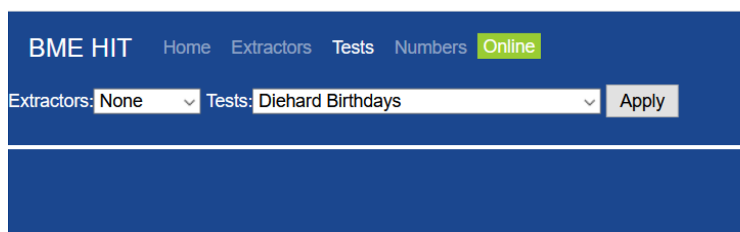
A weblap - ahogy korábban is említettük - három fő tartalommal rendelkezik. A véletlen számokat megjelenítő funkció, amely - ha éppen online üzemmódban van a QRNG -, akkor az általa generált bitsorozatot jelenti meg. Fizikailag a QRNG által




**4.6. ábra: Az elkészített weblap működése kisképernyős készülékeken**

generált kimenetet egy mappában tároljuk, ebből dolgozza fel a weblap a legfrissebb elemet. Mivel ezek a fájlok túl nagyok ahhoz hogy egyszerre legyenek a memóriába olvasva, a backend soronként olvassa a fájlt, és továbbítja a frontend felé. A frontend eltárolja a local storage-ban és innen bufferelve olvassa ki és írja ki a felületre. A weblap működés közben a 4.5 ábrán tekinthető meg, ahol látni lehet az éppen online generátor által előállított számokat.

A weblap reszponzívan átmérezhető, hogy kicsi képernyőn is értelmezhető legyen a felület. Ebben az esetben a szöveg tördelve jelenik meg, a navigációs sor pedig egy hamburger menüvé zsugorodik össze, ami lefelé lenyitható. A weblap kisképernyős megjelenése a 4.6 ábrán látható. A weblap a terveink szerint a későbbiek során minden műegyetemi polgár számára elérhető lesz az egyetem belső IP-tartományán, de jelen pillanatban még localhoston fut a webszerver, így az okostelefonon keresztüli megjelenésre optimailizált mobilképernyőt az ablak kicsinyítésével szimuláltuk.




**4.7. ábra: Az extraktorok és tesztek kiválasztása**

A másik fő pont a QRNG által generált kimenet tesztelése és a különböző extraktorok kiválasztásával a kimenet minőségének további javítása. Lehetővé tesszük ugyanis a weblap felhasználói számára, hogy a korábbi fejezetekben bemutatott teszteket és extraktorokat saját maguk is kipróbálják a QRNG által generált kimeneten. A legördülő elemek segítségével lehet kiválasztani a kívánt tesztet valamint az alkalmazni kívánt extraktort, ahogy az a 4.7. ábrán látszik. Az űrlaptovábbítja a backendnek a kívánt adatokat, és ezt követően megtörténik a tesztek futtatása és a kiválasztott extraktor alkalmazása

# 5 Összefoglalás

Dolgozatunkban felvezettük a kvantum alapú random generátorok alapvető működését és koncepcióját valamint a felhasználási lehetőségeiket. Majd bemutattuk a két Műegyetemen készülő kvantum alapú véletlenszám-generátor működési elvét. Ezután megvizsgáltuk a generátorok helyzetét az iparban és a piacon. Bemutattuk, hogy melyek a legismertebb módszerek egy véletlenszám-generátor tesztelése és hogy ezek hogyan működnek. Megemlítettük, hogy melyek a legtöbb helyen használt tesztcsomagok és az egyikről részeletesebben is beszámoltunk, bemutatva annak történetét röviden és elmagyarázva néhány benne található teszt működését. Ismertettük, hogy mit jelent egy generátor szempontjából az utófeldolgozás, mit értünk extraktor alatt és hogy ennek milyen típusait ismerjük. Bemutattuk a Műegyetemen épülő két generátor segítségével, hogy hogyan lehet a gyakorlatban is elvégezni egy QRNG tesztelését és az azon alkalmazott extraktorok hatékonyságát. A tesztelés során használt extraktorok működését és ahhoz szükséges kiegészítő információkat ismertettük, kitértünk ezen algoritmusok előnyeire és hátrányaira. Ezután a tesztelés eredményeit kiértékeltük és következtetéseket vontunk le belőle, valamint röviden bemutattuk a vizualizációs megoldásnak választott kapcsolódó weboldalt is.

*Köszönetnyilvánítás*

A munka a Kvantumbitek előállítása, megosztása és kvantuminformációs hálózatok fejlesztése nevű, 2017-1.2.1-NKP-2017-00001 számú projekt a Nemzeti Kutatási Fejlesztési és Innovációs Alapból biztosított támogatással, a "Nemzeti kiválósági program" pályázati program finanszírozásában valósult meg.

# Irodalomjegyzék


[1] Mario Stipcevic, “Quantum random number generators and their applications in cryptography” , Proc. SPIE 8375, Advanced Photon Counting Techniques VI, 837504 (2012); https://doi.org/10.1117/12.919920

[2] Mads Haahr: *Introduction to randomness and randomnumbers*, https://www.random.org/randomness/ (Utolsó megtekintés: 2020. október 27.)

[3] Bacsárdi László, Gerhátné Udvary Eszter, Imre Sándor, Schranz Ágoston, Zsolczai Viktor *“Optikai elven működő kvantum alapú véletlenszám-generátor architektúra tervezése” (Tech. report, 2018)*

[4] Gerhátné Udvary Eszter,Schranz Ágoston, Matolcsy Balázs, Mészáros Gergely, Petkovics Ármin, Wiandt Bernát,Rucz Péter “*Áttekintés a kvantum alapú véletlenszám-generátorokról” (Tech. report, 2018)*

[5] George Marsaglia „*The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness*”, https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/ (Utolsó megtekintés: 2020. október 18.)

[6] QRNG’s: Real Market Drivers https://www.insidequantumtechnology.com/qrngs-real-market-drivers/ (Utolsó megtekintés: 2020. október 27.)

[7] IDQ Company Profile https://www.idquantique.com/about-idq/company-profile/ (Utolsó megtekintés: 2020. október 27.)

[8] IDQ QRNGs applications in gaming and lotteries https://www.idquantique.com/random-number-generation/applications/gaming-and-lotteries/ (Utolsó megtekintés: 2020. október 27.)

[9] IDQ Banking Sector Use Case https://marketing.idquantique.com/acton/attachment/11868/f-7d9b3e78-4571-4549-964b-1209cb2bd6ac/1/-/-/-/-/Banking_NS%26I%20Use%20Case.pdf (Utolsó megtekintés: 2020. október 27.)

[10] IDQ Quantis QRNG Chip Brochurehttps://marketing.idquantique.com/acton/attachment/11868/f-025e/1/-/-/-/-/Quantis%20QRNG%20Chip_Brochure.pdf (Utolsó megtekintés: 2020. október 27.)

[11] QNC Products descriptions https://www.quantumnumberscorp.com/products/ (Utolsó megtekintés: 2020. október 27.)

[12] Robert G. Brown: „*Robert G. Brown's General Tools Page*”, https://webhome.phy.duke.edu/~rgb/General/dieharder.php, (Utolsó megtekintés: 2020. október 18.)

[13] Triola, Mario F., „*Elementray Statistics*", Addison-Wesley, 2001

[14] Martyn Shuttleworth, Lyndsay T Wilson, „*Type I Error and Type II Error*" https://explorable.com/type-i-error (Utolsó megtekintés: 2020. október 18.)

[15] NIST/SEMATECH „*e-Handbook of Statistical Methods: Kolmogorov-Smirnov Goodness-of-Fit Test*", https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm (Utolsó megtekintés: 2020. október 18.)

[16] Salil P. Vadhan, "*Pseudorandomness*", Foundations and Trends in Theoretical Computer Science: Vol. 7: No. 1–3, 2012, pp. 1-336.

[17] R. B. Davies, „*Exclusive OR (XOR) and hardware random number generators*", Tech. Rep., 2002. [Online], Available: http://www.robertnz. net/pdf/xor2.pdf (Utolsó megtekintés: 2020. október 21.)

[18] John Von Neumann, „*Various techniques used in connection with random digits*", National Bureau of Standards Applied Math Series 12, pp. 36–38., 1951

[19] Y. Peres, „*Iterating Von Neumann's Procedure for Extracting Random Bits*", Ann. Statist., pp. 590–597., 1992

[20] P. Elias, „*The efficient construction of an unbiased random sequence*" Ann. Math. Statist., pp. 865–870., 1972

[21] Ruilin Zhang, Sijia Chen, Chao Wan, Hirofumi Shinohara, „*High-Throughput Von Neumann Post-Processing for Random Number Generator*", 2018 International Symposium on VLSI Design Automation and Test (VLSI-DAT), pp. 1-4, 2018.

[22] Markus Dichtl, „*Bad and Good Ways of Post-Processing Biased Physical Random Numbers*" 14th International Workshop, FSE2007, Luxembourg, Luxembourg, March 26-28, 2007

[23] Erdinç Avaroğlu, Taner Tuncer, „*A novel S-box-based postprocessing method for true random number generation*", Turkish Journal of Electrical Engineering & Computer Sciences, 28: 288 – 301, 2020