SAPIENTIA ERDÉLYI MAGYAR TUDOMÁNYEGYETEM
MAROSVÁSÁRHELYI KAR,
INFORMATIKA SZAK

SAPIENTIA
ERDÉLYI MAGYAR
TUDOMÁNYEGYETEM

A dolgozat címe

DIPLOMADOLGOZAT

Témavezető:
Dr. Márton Gyöngyvér,
Egyetemi tanár

Végzős hallgató:
Kondert Matyas

2024

Declaraţie

Subsemnata/ul ....., absolvent(ă) al/a specializării ……., promoţia…… cunoscând prevederile Legii Educaţiei Naţionale 1/2011 şi a Codului de etică şi deontologie profesională a Universităţii Sapientia cu privire la furt intelectual declar pe propria răspundere că prezenta lucrare de licenţă/proiect de diplomă/disertaţie se bazează pe activitatea personală, cercetarea/proiectarea este efectuată de mine, informaţiile şi datele preluate din literatura de specialitate sunt citate în mod corespunzător.

Localitatea,

Data: Absolvent

Semnătura……

Kivonat

Dolgozatom témája a véletlenszerű számok generálása és ezek felhasználása kriptográfiában. A dolgozat keretén belül vizsgálni fogom: a véletlenszerű szám generáló algoritmusok alkalmazását különböző szakterületeken, a fontosságát a minőségi véletlenszerű szám generáló algoritmusoknak a kriptográfiában, az ilyen algoritmusok fajtái és a különböző fajta algoritmusok használati esetei.

Megvizsgálom a különböző programozási nyelvek által használt algoritmusok müködését, összehasonlítom ezeket egymás között, megvizsgálom a programozási nyelvek által biztosított kripto csomagok véletlenszerű szám generálo algoritmusokat és összehasonlítom őket az alapértelmezett algoritmussal

Meg fogom vizsgálni a fent említett algoritmusokat a NIST (National Institute of Standards and Technology) által biztosított statisztikai tesztcsomag segítségével ezáltal megvizsgálva a különböző algoritmusok előnyeit és hátrányait.

Be fogom mutatni a véletlenszerű szám generáló algoritmusok fontosságát valamint néhány helyzetete ahol az ilyen algoritmusok helytelen használata problémákat okozott a múltban.

Be fogom mutatni az általam fejlesztett véletlenszerű szám generátort, ennek fejlesztését, nehézségeket amibe ütköztem a fejlesztés során valamint alávetem a fent emlitet statisztikai tesztcsomagnak.

Tartalomjegyzék

1. fejezet

Bevezető

A véletlenszerűség egy olyan része a világunknak amit az emberiség már az őskorban megfigyelt és felhasznált az emberek szórakoztatására. Ebben az időben a véletlenszerűséget dobokockákal probálták szimulálni. Régészek felfedeztek dobókockákat a IE 2500-as évekből egy táblé(backgammon)-szerű játék része ként. Hosszú ideig a véletlenszerüség egyetlen haszna a különböző szerencsére alapuló játékok játszása volt. A 16. században Olasz matematikusok elkezdték tanulmányozni a szerencsejátékokat és a hozzájuk tartozó valószínűségeket. Ezt a kutatást. Az évszázadok során a véletlenszerűség háttér gondolat volt a legtöbb akadémiai körben azonban a 20. században a számítástechnika kialakulása után rájöttünk, hogy a véletlenszerűséget több dologra is fel tudjuk használni mint szerencsejátékokra.

A modern korban a véletlenszerűséget számos helyen felhasználjuk mint például: Monte Carlo módszer (egy determinisztikus problémát véletlenszerüség segítségével old meg), véletlenszerű mintavétel a modellek statisztikai vizsgálatához, számítógépes játékok futtatásához ahol azt szeretnénk, hogy a játék menete véletlenszerű legyen minden új játék indításakor, gépi tanulás futtatásakor a kezdeti sulyok beállításakor, számítógépes rendszerek streszteszelésekor.

A fentieken kívül talán a legfontosabb alkalmazása a véletlenszerüségnek a kriptográfia területén belül találjuk. A véletlenszerűség kritikus része a legtöbb modern kriptográfiai rendszernek. Míg a fenti példákban a véletlenszerüség minősége nem kritikus az alkalmazások helyes működéséhez (nem tekintjük végzetes hibának ha egy számítógépes játék ugyanazt a pályát többször generálja) a kriptográfiában létfontosságú, hogy a véletlenszerűség ténylegesen kiszámíthatatlanul viselkedjen.

2. fejezet

Véletlenszerű szám generáló algoritmusok fajtái és működése

A modern számítógépek működésének egyik alapelve az a tény hogy minden algoritmus kimenete kiszámítható ismerve a bemenetet és az algoritmust. Ilyen körülmények között elméletileg lehetetlen kéne legyen hogy egy számítógépes algoritmus egy véletlenszerű számot térítsen vissza kimenetként. Szóval, hogy tudunk véletlenszerű számokat generálni egy számítógépen? A válasz az hogy csalunk. Kétféleképpen csalhatunk ami által definiálhatjuk a véletlenszerű szám generáló algoritmusok. Ezek a valós és a pszeudo véletlenszerű szám generátorok.

A valós véletlenszerű szám generátorok egy kiszámíthatatlan fizikai jelenséget átfordítunk egy formátumba amit a számítógép tud értelmezni és ezt használjuk véletlenszerű számok generálására. A modern valós véletlenszerű szám generáló algoritmusok különböző fizikai jelenségeket használnak mint például a photonok (fényrészecskék) viselkedését, egy radioaktív anyag bomlásának gyakorisága, a számítógép processzorának a hőmérsékle. Ezek az algoritmusok igazán kiszámíthatatlanok tudnak lenni a megfelelő alkalmazásban és emiatt biztonságosabnak tekinthetőek mint a pszeudo véletlenszerü szám generátorok de lassabak is lehetnek mivel az algoritmusok várnia kell addig amíg a fizikai jelenség elég változáson ment át hogy egy új számot tudjon generálni. Emiatt a tulajdonság miatt hajlamosak a “bottleneck” problémára.

A másik megoldás az úgynevezett pszeudo véletlenszerű szám generátorok. Két fontos tulajdonsága a ezeknek az algoritmusoknak a determinisztikusság (az algoritmus egy adott bemenetre mindig ugyanazt a kimenetet fogja visszatéríteni) és a véletlenszerűségtől megkülömbözhetetlenség(az algoritmus kimenete egy külső szemlélő számára megkülönbözhetetlen kell legye egy véletlenszerű számsor tól). Ezek az algoritmusok egy kulcsból, ami egy az algoritmusnak megadott szám, generálnak egy látszólag véletlenszerű számot. Ennek a tulajdonságnak köszönhetően az, hogy az ilyen fajta algoritmusok kevésbé biztonságosak mivel, hogy ha egy támadó tudja az algoritmus bemenetét és az algoritmust akkor kiszámíthatja az algoritmus kimenetét és ezzel veszélyeztetve a rendszert. Az ilyen algoritmusokat általában egy láncolt rendszerben használjuk ahol a generálás kezdetekor megadunk egy kezdő értéket, az algoritmus ezt a kezdőértéket átalakítja egy látszólag véletlenszerű számmá majd mikor egy új számra van szükségünk az algoritmus az előzőleg generált számot felhasználva kezdő értékként generál egy új véletlenszerű számot. Fontos észben tartani hogy az eljárás amivel a számokat generáljunk olyan kell legyen

ami megakadályozza azt, hogy egy generált számból a szám generálására használt kezdő értékként meghatározhassa egy rosszindulatú felhasználó. Ezt a tulajdonságot forward security-nek nevezzük. Ezen felül amennyiben egy rosszindulatú felhasználó megtudja a lánc egyik állapotát az abból származó állapotokat is ki tudja számolni ezért érdemes az algoritmusnak időnként új kezdő értéket adni. Ezt a tulajdonságát az algoritmusnak backward security-nek nevezzük.

2.1. NIST statisztikai tesztcsomag

Mielőtt megvizsgálnánk a különböző véletlenszerű szám generáló algoritmusokat nézzük meg hogyan lehet összehasonlítani és megvizsgálni ezeket. Az egyik elfogadott módszer amit a véletlenszerüség vizsgálatára használunk az a NIST által publikált véletlenszerűség tesztelésére használt csomag. A tesztcsomag szerepe, hogy egy objektív tesztet biztosítson a generátorok felmérésére. Fontos észben tarttani, hogy ha egy generátor jól teljesít a tesztcsomagon ez nem jelenti azt, hogy a véletlenszerű szám generátor biztonságos hanem azt hogy a algoritmus által generált számok statisztikai szempontból az elvárt paraméterek közé esnek. A tesztek során az algoritmus egy 1-esekből és 0-sokból álló számsort kell generáljon amit 15 teszten kell átmenjen ahhoz hogy biztonságosnak számítson (statisztikai szempontból). Ezek a tesztek a következők:

- Gyakoriság (Monobit) Teszt: Gyakoriság (Monobit) Teszt: Ez a teszt megvizsgálja hogy a számsorban lévő 1-esek és 0-sok száma elég közel van-e egymáshoz. Egy véletlenszerű számsornak közel fele kell 1-esből és fele kell 0-sból álljon.

- Blokkonkénti Gyakoriság Teszt: Ez a teszt a számsort felosztja egyenlő méretű blokkokra és megvizsgálja hogy ezekben a blokokban az 1-esek és 0-sok aránya megegyezik e egy random sorozattól elvárt arányokkal.

- Homogén Sorozat Teszt: Ez a teszt a ugyanolyan számokból álló részsorokat vizsgálja a számsorban. Egy sorozat egy csak 1-eseket vagy 0-sokat tartalmazó része a számsornak aminek az elején és végén az ellenkező bit található. A tesz azt vizsgálja hogy a számsor mien gyakran vált 1-esről 0-ásra. Nem jó ha ez túl gyakran történik de az se jó ha túl ritkán.

- Leghosszabb 1-eseket Tartalmazo Részsor Teszt: Ez a teszt blokkokra osztja a számsort majd megállapítja hogy a leghosszabb 1-esekből álló rész hossza elvárt paraméterek közé esik e minden blokban.

- Bináris Mátrix Rang Teszt: Ez a teszt mátrixokba rendezi a számsort majd megvizsgálja ezeknek a mátrixoknak a rangját ezzel elenörizve, hogy a sorozatban lévő számok függetlenek e egymástól.

- Diskrét Fourier Transform Teszt: Ez a teszt egy Fourier transform segítségével ellenőrzi le hogy vannak e ismétlődő minták a számsorban.

- Átfedésmentes Sablon Teszt: Azt ellenőrzi hogy egy sablon nem e jelenik meg a mintában többször mint azt egy véletlenszerű mintától elvárnánk

- Átfedéses Sablon Teszt: Ugyanaz mint az előző teszt csupán az átfedéseket is leelenőrzi.
- Maurer's Univerzális Statisztikai Test: Ez a teszt azt vizsgálja, hogy milyen hosszúak az ismétlődő minták közötti szakaszok. Ez azt mutatja meg hogy mennyire lehet lerövidíteni a számsort. Ha egy számsort lényegesen le lehet rövidíteni az arra utal, hogy a számsor nem véletlenszerű.
- Lineáris Bonyolultság Teszt: Ez a teszt blokkokra osztja a számsort majd megvizsgálja hogy milyen hosszú kell legyen egy Lineáris Visszacsatolási Eltolási Regiszter (Linear Feedback Shift Register vagy LFSR) ahhoz, hogy a blokkban szereplő számsort létrehozza. Az LFSR-ek olyan algoritmusok amik egy bináris számsor és néhány szabály segítségével egy új bináris számsort tudnak kenerálni. Minél hosszabbak az LFSR-ok amiket a blokok generálására használunk annál véletlenszerű a számsor.
- Minta teszt: Megvizsgálja hogy hányszor szerepel a számsorban az összes lehetséges m hosszúságú minta. Ha a számsor véletlenszerű akkor a minden minta szinte ugyan annyiszor kell, hogy szerepeljen a számsorban. m=1 esetén ez a teszt megegyezik a Monobit tesztel
- Entropia Teszt: A teszt összehasonlítja az m hosszú minták előfordulásának számát az m+1 hosszúságú mintál előfordulását hogy kiszámítsa az eredeti számsor megközelítő entropiáját
- Osszeg Teszt: Ez a teszt átalakítja a 0-sokat -1-esekké majd végig lépeget a számsoron a és kiszámolja az összegete a jelenlegi elemnek az összes többi előtte lévő elemmel. Megvizsgálva az így kiszámolt összegek közül a legnagyobbat meghatározhatjuk, hogy mennyire véletlenszerű a a számsor.
- Véletlenszerű Út Teszt: Ez a teszt átalakítja a 0-sokat -1-esekké majd végig lépeget a számsoron összeadva az értékeket és ezeket az értékeket ábrázolva egy gráfon. Ezt a gráfot véletlenszerű sétának (random walk) nevezzük. A gráf segítségével a számsort ciklusokra bontjuk ahol egy ciklus elejét és végét azok a pontok határozzák meg ahol a gráf értéke 0. Ezekben a ciklusokban megvizsgáljuk, hogy a ciklusban szereplő állapotok előfordulásának száma megegyezik-e egy véletszerű szám sorban megtalálható állapotok számával.
- Véletlenszerű Út Teszt Változata: Ebben a tesztben az előzőhöz hasonlóan létrehozunk egy véletlenszerű séta gráfot amiben megszámoljuk a különböző állapotok előfordulásának számát. Ezeket az megvizsgálva megállapíthatjuk hogy számsor úgy viselkedik-e mint ahogy azt egy véletlenszerű számsortól elvárnánk.

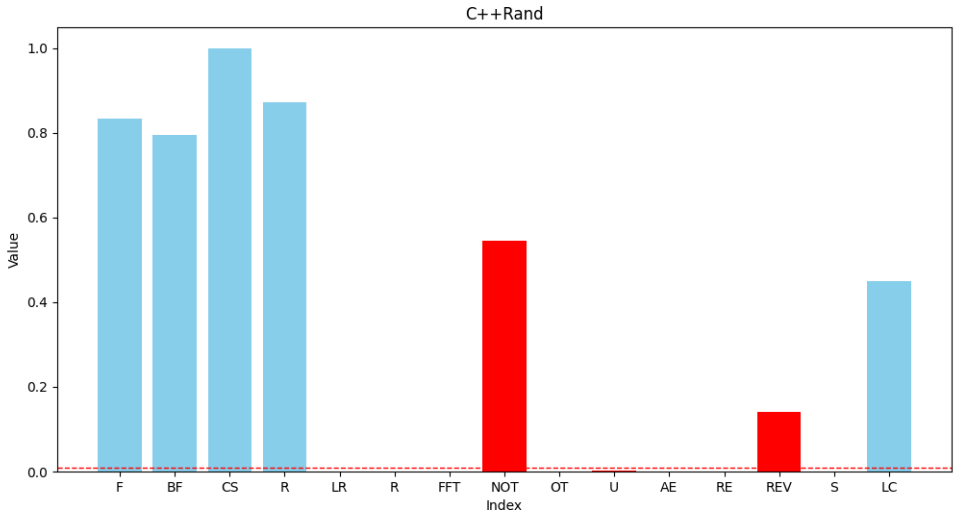
2.2. C++ Véletlenszerű szám generáló

A C++ egyike a legismertebb programozási nyelveknek a világon. Népszerűségét a gyorsaságának valamint a nyelv standard könyvtárában elérhető számos eszköznek köszönheti. Az egyik ilyen eszköz a stdlib nevű konyvtárban található rand függvény. Mindenki aki véletlenszerű számokat akar generálni egy C++ programban szóval vizsgáljuk

meg hogyan is működik. A függvény egy lineáris kongruenciális generátort használ a számok generálására menynek keplete:

$$X_{n+1} = (aX_n + c) \bmod m \tag{2.1}$$

Amennyiben többször hívjuk meg a függvényt ez az előző hívásnál generált értéket fogja felhasználni az új szám generálására. Mielőtt ezt a függvényt használni tudnánk azonban ajánlott beállítani egy kezdeti értéket az srand függvény segítségével. Gyakori megoldás hogy az srand függvénynek a jelenlegi idő unix timestamp-jét megadni mint paraméter ezzel elkerülve azt, hogy a rand függvény ugyanazokat az értékeket térítse vissza minden futtatásnál. Ez a megoldás első ránézésre elég primitivnek tűnik de lássuk, hogy teljesít a NIST statisztikai tesztcsomagával vizsgálva.

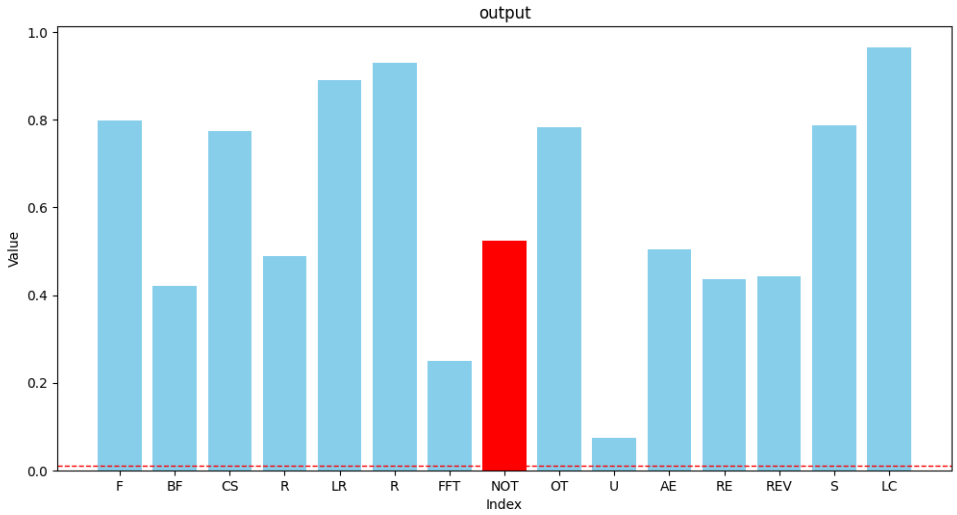


2.1. ábra. C++ Rand Függvény Eredmények

Mint a fenti ábrán is láthatjuk ez a módszer elég gyengén teljesít a teszten. A szaggatott piros vonal jelzi a küszöböt amit az algoritmus el kell érjen ahhoz, hogy sikeresnek számítson. Ez a véletlenszerű szám generáló egyértelműen nem elég biztonságos hogy kriptográfiai rendszerekben felhasználásuk.

2.3. Java Véletlenszerű szám generáló

A Java programozási nyelv egy a C++-nál újabb népszerű nyelv melyet főként web-alkalmazások és mobil alkalmazások fejlesztésére használják. Java-ban a C++ hasonlóan sok kiegészítő könyvtárhoz van hozzáférésünk melynek egyike a import java.util.Random könyvtár ami véletlenszerű számok generálására használják. Ez a könyvtár a C++-ban véletlenszrű szám generálásra használt lineáris kongruenciális generátort használ. A különbség a két könyvtár között az generátor paramétereinek értékei. Lássuk hogy teljesít lineáris kongruenciális generátort az új paraméterekkel:

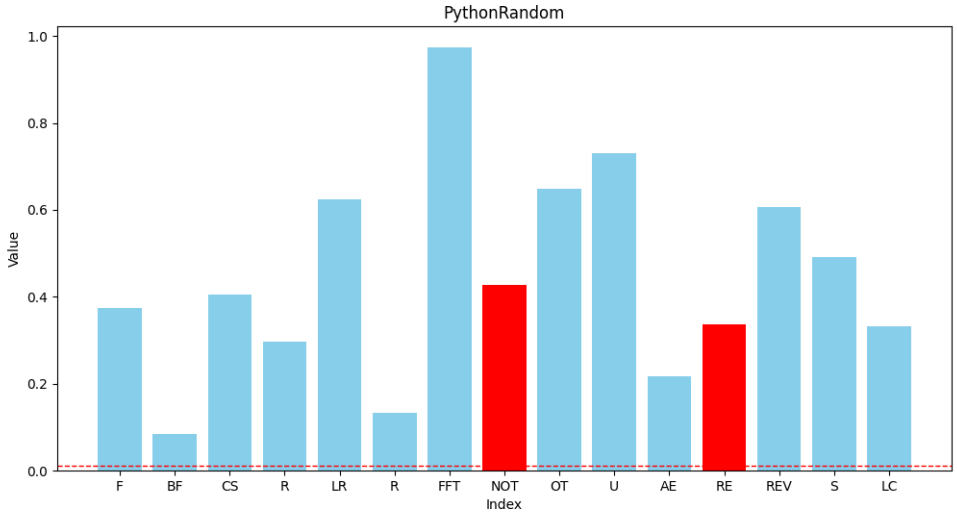


2.2. ábra. Java Random Függvény Eredmények

Az fenti ábrán láthatjuk, hogy ez az algoritmus jobban teljesít a C++ os verziónál. Ez valószínűleg az generátor paramétereinek köszönhető.

2.4. Python Véletlenszerű szám generáló

A python a C++-al és Java-val ellentétben nem lineáris kongruenciális generátort használ hanem az 1997-ben kifejlesztett Mersenne Twister algoritmust. Ez az algoritmus egy úgynevezett állapot tömböt használ a számok generálására. Ezt a tömböt a inicializálás során feltölti számokkal majd a bit csúsztatások segítségével egy új állapot tömböt generál majd ennek az állapot tömbnek a segítségével generál számokat. A tulajdonság ami megkülönbözteti a Mersenne Twister algoritmust a többi véletlenszerű szám generáló algoritmustól az az elképesztöen hosszu periodusa. Egy véletlenszerű szám generátor periódusa az a szám amit ki tud generálni mielőtt a számok ismétlődni kezdenek. Ez a periódus a Mersenne Twister esetében $2^{19937} - 1$. Lássuk hogy teljesít ez az algoritmus a NITS tesztjén:



2.3. ábra. Python Random Függvény Eredmények

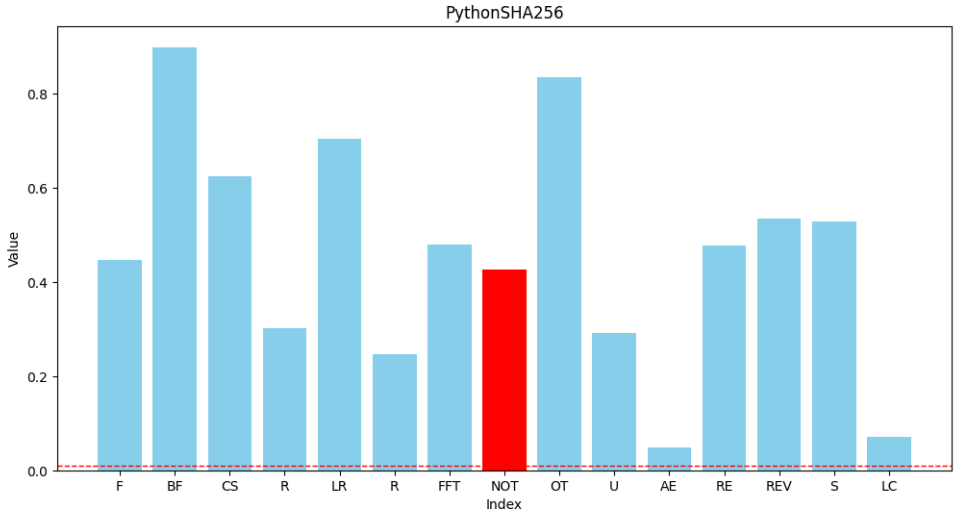
Mint ahogy az ábrán is láthatjuk ez az algoritmus szintén elég jól teljesít a fenti teszteken azonban hasonlóan a Java algoritmusához ez az algoritmus is elbukik néhányat NOT(Non-overlapping Template) tesztek közül.

2.5. SHA-256 algoritmus felhasználása véletlenszerű számok generálására

A SHA-256 egy hash algoritmus amit 2001-ben publikált az Amerikai NSA(National Security Agency). A hash algoritmusok lényege hogy egy tetszőleges méretű adathalmazt átalakít egy 256 bites “hash”-é. Ahhoz hogy egy hash algoritmus biztonságos legyen 3 tulajdonsággal kell rendelkezzen:

- Az első az úgynevezett “pre-image resistance” ami azt biztosítja, hogy egy hash-ből nem lehet visszanyerni az eredeti adathalmazt.
- A második az úgynevezett “second pre-image resistance” ami azt biztosítja hogy ha valaki tudja a forrását egy hash-nek és a hash-et nem tud egy új bemenetet adni a hash algoritmusnak amire ugyanazt a hash-et térítenek vissza.
- A harmadik az úgynevezett “collision resistance” ami azt mondja hogy nem létezhet két bemeneti adathalmaz amire a hash algoritmus ugyanazt a hash-t adja vissza.

Ezeket a tulajdonságokat figyelembe véve a hash algoritmusok alkalmasak kéne legyenek véletlenszerű számok generálására. Vegyük a “Államvizsga” karakterláncot mint seed és generáljunk belőle hash-eket. Ezeknek a hasheknek a hasheknek az utolso bitjét egymáshoz fűzve generáljunk egy véletlenszerű számot és vizsgáljuk meg hogy ez a szám hogy teljesít a NIST tesztjén.

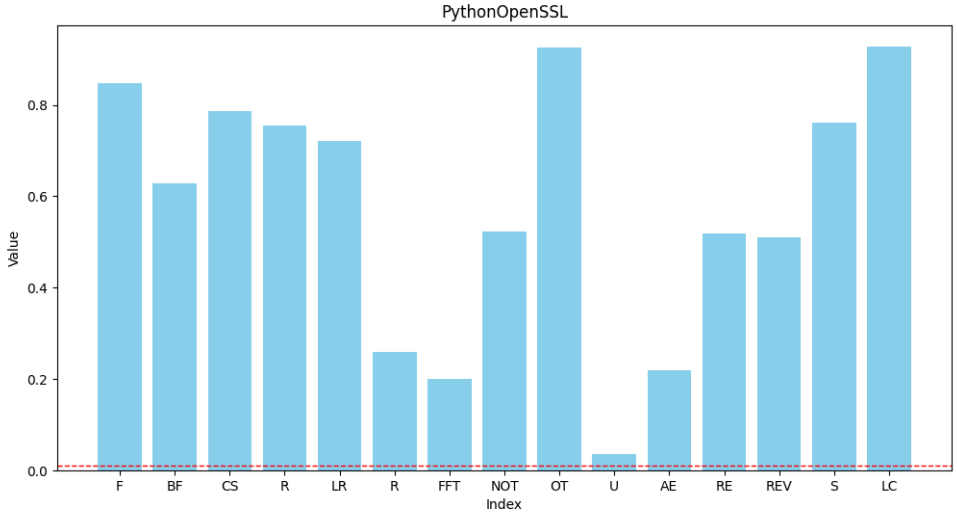


2.4. ábra. SHA-256 Függvény Eredmények

Ahogy láthatjuk az előző két algoritmushoz hasonlóan a ez az algoritmus is elbukik néhányan a NOT(Non-overlapping Template) tesztek közül.

2.6. Véletlenszerű számok generálása OpenSSL-el

Az OpenSSL egy open-source könyvtár amit 1990-es években fejlesztettek ki arra a célra, hogy egy erős és biztonságos eszköz legyen kriptográfiai felhasználásra. Ez a könyvtár lehetőséget ad a véletlenszerű számok generálására. Erre a célra az adott számítógépen futó operációs rendszer által szolgáltatott generátort használja fel számok generálására. Ez a generátor általában különböző forrásokból kinyert entropia segítségével generál véletlenszerű számokat. Ezek a források lehetnek a számítógép termális információja, a háttérben futó folyamatok azonosítói, a számítógép indításnál eltelt idő. Ezekből a forrásokból az operációs rendszer képes gyorsan jó minőségű véletlenszerű számokat generálni. Lássuk milyen eredményeket ér el az algoritmus a NIST teszteken:



2.5. ábra. Python Random Függvény Eredmények

Amint azt láthatjuk ez az algoritmus teljesíti minden tesztjét a NIST teszt csomagnak. A tényező ami erőssé teszi ezt az algoritmus a többihez képest az az a tény, hogy ötvözi egy valós véletlenszerű számokat generáló algoritmust egy pseudo véletlenszerü számokat generáló algorimussal.