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

Kvantum alapú véletlenszámgenerátor valós idejű tesztelése

TDK dolgozat

Készítette:

Iván Attila Gyula
Solymos Balázs

Konzulens:

dr. Bacsárdi László

2019

Tartalomjegyzék

Kivonat

Napjaink kriptográfiai megoldásai gyakran támaszkodnak véletlenszámokra, melyek minősége ezek helyes működéséhez sok esetben kritikus. A leggyakrabban használt véletlenszám generátorok ún. pszeudo véletlenek, ezeket valamilyen nem véletlen forrás (például a rendszer idő), segítségével generáljuk. Különböző fizikai jelenségek, például kvantummechanikai mérések kihasználásával lehetőség nyílik bizonyítottan nemdeterminisztikus viselkedésű generátorok készítésére. Az ilyenek elméleti működése sokszor viszonylag könnyen megérthető, azonban technológia korlátok miatt fizikai megvalósításuk és üzemeltetésük már korántsem problémamentes, így szükséges megfelelő felügyelő és hitelesítő rendszer alkalmazása.

A generátor véletlen kimenetéből adódóan, az eszköz ellenőrzésére korlátozott lehetőség adódik, mivel teljes biztonsággal semmilyen kimeneti formára nem állítható, hogy hibás működésből származna. Megfelelő statisztikai tesztek alkalmazásával azonban a gyakorlatban észlelhetők a szokásostól eltérő, nem megfelelő minőségű kimenetet létrehozó állapotok. Tökéletlen rendszer hibáinak megfelelő ismeretében, állítható egy jelformáról, hogy sokkal nagyobb valószínűséggel hibából származik, mint helyes működésből.

Dolgozatunkban egy hazai kvantumtechnológiai projekt részeként általunk jelenleg is fejlesztés alatt álló feldolgozó és felügyelő rendszert ismertetjük. Ennek feladata a fizikai eszközből származó nyers adatfolyam valós idejű feldolgozása és ellenőrzése, valamint a feldolgozott, jó minőségű kimenet külvilág felé történő biztosítása. A jelenleg elérhető statisztikai teszt környezetek nem képesek valós időben futni, ezek átalakítása sajátos szoftver-architektúrát igényelt, amely egyszerre képes stabilan és flexibilisen működni. A valós idejű feldolgozás hatékony megvalósításához, véges rendelkezésre álló erőforrás miatt, megfelelő eszközkészlet választása szükséges. Ehhez ideálisan, az ellenőrző rendszert a fizikaihoz igazíthatóvá, jellegzetes hibákra érzékennyé kell tenni. Munkánk során teszteltük a rendszer működését különböző általános véletlenforrásokkal, és vizsgáltuk az egyes rendelkezésünkre álló eszközök alkalmazhatóságát.

Abstract

Many cryptographic use cases today rely on random numbers. Quality of these numbers is usually critical for security. Most algorithms used for creating random numbers are so called pseudo random number generators. These often use some non-random source (e.g., system time) during their operations, making them deterministic in nature. There are multiple ways to build a proven non-deterministic "true" random number generator, for example relying on quantum measurements. Their operation principles often seem to be simple in theory, but building and operating them has multiple challenges arising from current limitations of technology, resulting in the need for an appropriate monitoring system.

Due to the output's random nature, detecting a failure using the collected data cannot be done with absolute certainty. Every possible sequence can occur during normal operations, meaning no state can correspond to an error without ambiguity. Certain statistical tests, however, can be used to provide useful observations even in this case. Sufficiently knowing the underlying system, one can conclude that a given sequence is much more likely to be the result of an error rather than a rare occurrence from nominal operations.

In our work, we present a system which is currently under development in the framework of a Hungarian quantum technological project. This system is responsible for the real-time monitoring and validation of the data coming from the soon-to-be-built physical hardware. Currently available statistical test suites cannot efficiently satisfy the real-time requirement, therefore we designed a custom software that can, while allowing us a greater degree of flexibility. This added flexibility is also crucial for our intended use case. The system has access to finite computational power, resulting in it being the limiting factor for the depth of analysis applicable. Therefore, picking the right toolset for this analysis is paramount. Ideally, the monitoring system should be flexible enough to allow itself to be custom fitted to the hardware it is monitoring, resulting in greater sensitivity. We have tested our system with various available random number generators, while investigating the applicability of individual analytical tools.

1. fejezet

Bevezetés

Az információs technológiák fejlődésével, egyre növekszik a véletlen eseményeket valamilyen formában hasznosító alkalmazások száma. Ezen események által biztosított kiszámíthatatlanságra az őket használó alkalmazások szemszögéből, a számítási kapacitáshoz és átviteli sebességhez hasonlóan működés során felhasznált erőforrásként tekinthetünk, mely két fontos mérőszámmal bír. Egyik a rendelkezésre álló bitek száma (erőforrás mennyisége), másik pedig az ezek által tartalmazott bitenkénti entrópia (erőforrás minősége). Ez utóbbi megfelelősége különös jelentőséggel bír kriptográfiai eljárásokban.

Véletlenség előállítására igen sokféle módszert alkalmazhatunk. Tipikus mindennapi életből származó példa, a különböző játékokhoz használt dobókocka, vagy egy pénzérme feldobása. Informatikai alkalmazásokban pedig jellemzően valamilyen nehezen megjósolható algoritmus kimenetét, vagy egy kiszámíthatatlan fizikai folyamat mintavételezésének eredményét használjuk. Közös jellemzőjük ezeknek, hogy felhasználóként elvárjuk, hogy a résztvevő felek közül senki se rendelkezhessen a folyamat várható kimenetéről (legyen az kockadobás, vagy egy bitsorozat) többlet információval a többiekhez képest. Míg a való életben egy cinkelt kockát használó játékos a társai nyerési esélyeit aránytalanul rontja (ezzel elrontva a játékot), titkosításhoz felhasznált véletlen esemény támadó általi előzetes ismerete, a titkosított információ titkosságát szüntetheti meg. Ezen szituációk elkerülése végett használat előtt mind a kocka, mind a véletlen információt szolgáltató forrás megfelelőségéről meg kell bizonyosodnunk.

Az ellenőrzésnek egy lehetséges módja a kocka esetében, annak tömegeloszlásának pontos meghatározása, a használni kívánt eszköz megfelelő átvizsgálása, bizonyság szerzése arról, hogy minden eleme az elvártnak megfelelően működik. Ez informatikai felhasználások esetében a teljes rendszer megfelelő analízisének feleltethető meg. Egy másik megközelítést jelenthet, ha tapasztalati úton ellenőrizzük a kívánt viselkedést. Dobunk a kockával sokat és ha azt látjuk, hogy meglepően sokszor hatost kaptunk, arra gyanakszunk, hogy valami nincs rendben. A kimeneten egyfajta statisztikai tesztet végzünk.

Statisztikai tesztek használata kifejezetten hasznos lehet másképp nehezen detektálható hibák felderítésére, azonban a feldolgozott adatfolyam véletlen jellegéből adódóan egy nagy hiányosságuk, hogy teljes biztonsággal nem állíthatjuk semmilyen vizsgált folyamról, hogy az ténylegesen véletlen-e vagy sem. Bizonyos állításokat viszont így is tehetünk. Egy ilyen lehet például, hogy a vizsgált bemeneti forma sokkal nagyobb valószínűséggel valamilyen működési hiba eredménye, semhogy megfelelő működésből származna. Mindig van valamekorra esély még így is hamis hiba detektálására, azonban megfelelő paramé-

terek választásával ez kellően alacsonyra csökkenthető, így ezekkel a tesztekkel is nagy megbízhatóságú eredményekhez juthatunk.

Egy hazai kvantumos elven működő véletlenszám generátor építésén dolgozó projekt részeként a mi feladatunk a készülő rendszer által szolgáltatott kimenet feldolgozása, validálása és felügyelete, valamint külvilág felé történő kimenet biztosítása. Léteznek kifejezetten véletlenszám generátorok ellenőrzésére különböző szoftvercsomagok, azonban ezek a feladat által támasztott valós idejűséget nem tudják teljesíteni. Megoldásként saját szoftver-architektúrát készítettünk, ami kielégíti mind ezt a kritériumot, mind a feladat által támasztott egyéb elvárásokat (nyers egyenetlen adatfolyam feldolgozása, hosszú távú adatgyűjtés validációhoz). Figyelembe véve a rendelkezésre álló számítás kapacitás végességét, külön figyelmet fordítottunk a rendszer flexibilisségére, megadva a lehetőséget arra, hogy a hardver elkészültével, annak jellegzetességeihez a készített környezet igazítható legyen, ezáltal a célnak jobban megfelelő, összességében hatékonyabb megoldást adva. A flexibilis és könnyen használható rendszer további előnye, hogy lehetőséget biztosít meglévő, vagy akár egyedi új eszközök egyszerű környezetbe illesztésére, valamint működésük tesztelésére is.

1.1. A dolgozat felépítése

A 2. fejezetben bemutatjuk véletlenszám generátorok különböző fajáit, kitérva a tervezett fizikai hardver lehetséges kvantumos alapú megvalósításaira. Ezután megismerkedünk a generátorok és véletlen viselkedés ellenőrzésének témakörével, jelenleg is elérhető megoldások áttekintésével zárva a szekciót. Az általunk készített architektúra a 3. fejezetben, míg az ebben a rendszerben elérhető vizsgálatra használható eszköztár a 4. fejezetben kerül bemutatásra. Ezek folyamán ismertetjük a tervezés és kivitelezés során hozott döntéseinket, azok hátterét, illetve az így keletkezett környezet képességeit. A rendszer és hozzá tartozó eszköztár helyes működését az 5. fejezetben különböző tesztesetekkel vizsgáljuk, majd a 6. fejezetben ismertetjük rendszerünk tervezett, valamint jövőbeni lehetséges felhasználási területeit.

2. fejezet

Véletlenszám generátorok

A fejezetben röviden ismertetésre kerülnek a különböző típusú generátorok, valamint ezek jellemzői, külön kitérve az általunk készített rendszer által felügyelendő hardver tervezett működési módjaira. Ezután megvizsgáljuk a generátorokkal szemben támasztott elvárásokat és az ellenőrzésükhöz rendelkezésre álló eszköztárat, amik vizsgálatára az általunk készített rendszernek is képesnek kell lennie.

2.1. Véletlenszám generátorokról általában

A mindennapi életben számos helyen használunk véletlen eseményeket, elég csak a különböző szerencsére is alapuló játékokra gondolni, melyek közül egyeseket már a korai civilizációk korában is játszottak. Az informatika fejlődésével azonban mostanra megjelentek más felhasználási területek is, melyek közül az egyik talán legjelentősebb a biztonságtechnika. A legtöbb titkosítási eljárás keményen támaszkodik véletlen számsorokra, kihasználva ezeknek azt a fontos tulajdonságát, hogy a véletlen jelleg miatt előre megjósolhatatlanok, kizárólag tippelhetők. Ebből következik például, hogy amennyiben egy ilyen sorozatot használunk titkos kulcsnak, ezen kulcs kitalálására egy potenciális támadó számára sincs hatékonyabb algoritmus az egyszerű tippelésnél, amihez a szükséges számítási kapacitás a sorozat hosszával exponenciálisan nő (kellően hosszú sorozat esetén ezt a kapacitást közel lehetetlen biztosítani). Ezeknél a felhasználásoknál különös fontossággal bír a szolgáltatott véletlenség minősége, mivel ez biztosítja a támadhatatlanságot. Kommunikációs esetekben azonban fontos metrika még az alkalmazott generátor sebessége is, mivel újabb üzenetek titkosításához újabb véletlen számokra van szükség. Ezen sorok lassú előállítása akár a teljes biztonságos kommunikáció sebességét is korlátozhatja.

A generátoroknak általánosan két fő fajtáját különböztetjük meg a kimenet előállításához felhasznált kezdeti entrópiaforrás szerint. Ennek megfelelően beszélünk pszeudo (PRNG - Pszeudo Random Number Generator) és valós (TRNG - True Random Number Generator) véletlenszám generátorokról. A pszeudo véletlenszám generátoroknál a „pszeudo” arra vonatkozik, hogy ezek a generátorok valójában nem igazi véletlen számokat szolgáltatnak, mivel működésük determinisztikus.

2.1.1. Pszeudo véletlenszám generátorok

Generátorok ezen csoportja determinisztikus működésű algoritmusokat takar, melyek belső állapotuk alapján szolgáltatnak látszatra véletlen kimenetet. Indításukkor a kiinduló belső állapotot (seed) érdemes valamilyen valódi entrópiaforrás segítségével megadni, ezzel nehezítve egy az algoritmust ismerő támadó dolgát. Ez azonban sokszor nem elégséges védelem. Elegendő adat gyűjtésével a belső állapot becsülhető, ezáltal sebezhetővé téve a generátort. Emiatt érdemes hozzájuk társítani egy akár rosszabb minőségű, lassabb, de valós véletlenszám generátort, amivel a PRNG belső állapota periodikusan frissíthető. Következményképp egy valódi entrópiaforrás entrópiája kerül szétterítésre a jellemzően hosszabb kimeneti bitsorozaton. Ebben az üzemmódban a determinisztikus algoritmus a valós forrás feldolgozójaként, entrópia extractoraként működik. Kellően gyors forrás hiányában használatuk előnyös lehet ezen hiányosságok részbeni elfedésére, mivel képesek a legtöbb felhasználás számára elfogadható minőségű kimenet előállítására. Különös figyelmet igényel azonban a megfelelő állapot inicializáció, ennek hiánya komoly sebezhetőségekhez vezet. [1][2] Jellemzően ilyen algoritmusok egyes titkos kommunikációhoz használt cypher-ek, (a titkosított üzenet véletlennek tűnik), valamint amerikai NIST (National Institute of Standards and Technology (USA)) is adott ki ajánlást arról szólóan, hogy milyen kritériumoknak kell megfelelni egy ilyen konstrukciónak.

2.1.2. Valós véletlenszám generátorok

Olyan fizikai folyamatok mintavételezésével, melyeknek eredményét pontosan megjósolni nem tudjuk is készíthetőek generátorok, ezáltal közvetlen egy megfelelően felügyelt és kondicionált környezeti eseményből entrópiát szerezve. Nagy előnyük, hogy működésük nemdeterminisztikus, ezáltal nincs olyan algoritmus amely ismeretében támadásuk egy belső állapot becslésére lenne visszavezethető. Mivel az entrópiát a környezetből szerzik, az ehhez szükséges fizikai folyamat előidézését, mérését azonban megfelelően biztosítani kell. Ez megoldandó problémák egy új osztályát jelenti. A méréshez használt eszközök a valóságban nem ideálisak, a mért rendszerre hatással lehet a környezet. A fizikai rendszer által szolgáltatot véletlenség ezen felül általában a kívánttól eltérő eloszlású, kimenet nyújtása előtt további feldolgozó lépések szükségesek. A megvalósítás szükségszerű tökéletlenségeinek hatását ellensúlyozni kell, ami körültekintő kivitelezést, kalibrációt és felügyeletet igényel.

2.2. Kvantum véletlenszám generátorok

Kvantum véletlenszám generátorok nagy előnye, hogy az általuk mintavételezett fizikai folyamatok bizonyítottan véletlen viselkedésűek, ezt a viselkedést kívülről befolyásolni nem lehet. Emiatt elméleti megvalósításuk egyszerű, hiszen elég csak ezt a kvantumos információt hordozó folyamatot jól mintavételezni.

2.2.1. Kvantuminformatikai bevezető

Kvantumos információ különböző fizikai formákban megjelenhet. Ilyen lehet például elektronok spinje, fotonok polarizációja, vagy akár az is, hogy egy foton a számára elérhető

utak közül, melyiken haladt (Ezt használja a 2.2.2 fejezet tervei közül az első). Elméleti szempontból ennek pontos megjelenése közömbös, az információ időfejlődése a fontos, amit általánosan kvantumbitekkel és azokon végzett műveletekkel leírhatunk. Diszkrét időben ezt az alábbi posztulátumok megfogalmazása segíti.

Posztulátumok

Diszkrét időfejlődést leíró kvantuminformatika az alábbi négy posztulátumból indul ki [3, 4]:

1. **posztulátum:** Egy zárt fizikai rendszer állapota leírható egy komplex valószínűségi amplitúdókkal jellemzett egységnyi hosszú vektorral, amelyet a Hilbert-térben értelmezünk (egy komplex vektortér, amelyben definiált a skaláris szorzás).

2. **posztulátum:** Egy zárt fizikai rendszer két időpont közti változása leírható egy unitér transzformációval, amely csak a két időponttól függ.

$$|\varphi'\rangle = U\,|\varphi\rangle \tag{2.1}$$

3. **posztulátum:** A kvantum mérések leírhatók ún. mérési operátorok halmazával. Egy adott m állapot mérésének valószínűsége:

$$p(m) = \langle\psi|\, M_m^\dagger M_m\,|\psi\rangle \tag{2.2}$$

a mérés után a rendszer a alábbi állapotba kerül:

$$\frac{M_m\,|\psi\rangle}{\sqrt{\langle\psi|\, M_m^\dagger M_m\,|\psi\rangle}} \tag{2.3}$$

A méréseknek teljesíteniük kell a teljességi relációt,

$$\sum_m M_m^\dagger M_m = I \tag{2.4}$$

vagyis az összes valószínűség összegének 1-nek kell lennie, a mérésnek le kell fednie minden lehetséges állapotot.

$$\sum_m p_m = 1 \tag{2.5}$$

4. **posztulátum:** Egy összetett rendszer állapota leírható az egyes rendszerek állapotának tenzor szorzatával,

$$W = |\varphi_0\rangle \otimes |\varphi_1\rangle \otimes |\varphi_2\rangle \otimes \cdots \otimes |\varphi_n\rangle \tag{2.6}$$

A Bra-ket jelölés

A kvantumbitek (vagy qubitek) leírására leggyakrabban Bra-ket jelölést használunk [3] (mivel ezt a jelölést Dirac vezette be ezért néha Dirac-jelölésként is hivatkozott). Egy rendszer állapotát általában $|\psi\rangle$-vel [„ket szí”] jelöljük, amely egy komplex oszlopvektor.

Fontos jelölés még $\langle\psi|$ [„bre szí”], $|\psi\rangle$ transzponáltjának konjugáltja, vagyis (komplex) adjungáltja[1].

$$|\psi\rangle = \begin{bmatrix}\psi_1\\ \psi_2\\ \vdots\\ \psi_n\end{bmatrix}$$
$$\langle\psi| = |\psi\rangle^\dagger = |\psi\rangle^H = |\psi\rangle^{-T} = \begin{bmatrix}\overline{\psi_1} & \overline{\psi_2} & \dots & \overline{\psi_n}\end{bmatrix}$$

Az állapotok egyszerűbb kezeléséhez érdemes definiálnunk két bázis vektort, melyek általában $|0\rangle$ és $|1\rangle$:

$$|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}, |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \tag{2.7}$$

A bázis vektorok felhasználásával, bármilyen tiszta állapot leírható az alábbi módon:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\\beta\end{bmatrix} \quad (\alpha,\beta\in\mathbb{C};\quad |\alpha|^2+|\beta|^2=1) \tag{2.8}$$

Projektív mérés

Projektív mérés esetén két ortonormált állapot között döntünk, például $|0\rangle$ és $|1\rangle$ között. A mérési operátorok ebben az esetben [4]:

$$M_0 = |0\rangle\langle 0| = \begin{pmatrix}1&0\\0&0\end{pmatrix}$$
$$M_1 = |1\rangle\langle 1| = \begin{pmatrix}0&0\\0&1\end{pmatrix}$$

Ezek teljesítik a teljességi relációt:

$$\sum_m M_m^\dagger M_m = \begin{pmatrix}1&0\\0&0\end{pmatrix} + \begin{pmatrix}0&0\\0&1\end{pmatrix} = \begin{pmatrix}1&0\\0&1\end{pmatrix} = I \tag{2.9}$$

Ekkor a posztulátumok alapján számolható a két állapot mérésének valószínűsége, amelyek a következők:

$$P(0) = \langle\phi|\begin{pmatrix}1&0\\0&0\end{pmatrix}\begin{pmatrix}\alpha\\\beta\end{pmatrix} = \begin{pmatrix}\overline{\alpha} & \overline{\beta}\end{pmatrix}\begin{pmatrix}\alpha\\0\end{pmatrix} = |\alpha|^2 \tag{2.10}$$

[1]Ez a fogalom nem azonos a mátrixok invertálásánál használt adjungálttal.

$$P(1) = \langle\phi| \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \left(\alpha ß \beta\right) = \begin{pmatrix} \overline{\alpha} & \overline{\beta} \end{pmatrix} \begin{pmatrix} 0 \\ \beta \end{pmatrix} = |\beta|^2 \tag{2.11}$$

Egyszerű kvantum alapú véletlenszám generátor elméletben

Az eddig ismertetett kvantum elemekből könnyen vázolhatjuk egy kvantum alapú véletlenszám generátor legegyszerűbb elméleti működését. Induljunk ki $|\psi\rangle = |0\rangle$ állapotból, alkalmazzunk egy ún. Hadamard kaput, ez az állapotot az alábbi $|\psi\rangle$ állapotba viszi.

$$|\psi\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{2.12}$$

Alkalmazzuk ezután az előzőekben ismertetett projektív mérést:

$$P(0) = |\alpha|^2 = \frac{1}{\sqrt{2}}^2 = \frac{1}{2}$$
$$P(1) = |\beta|^2 = \frac{1}{\sqrt{2}}^2 = \frac{1}{2}$$

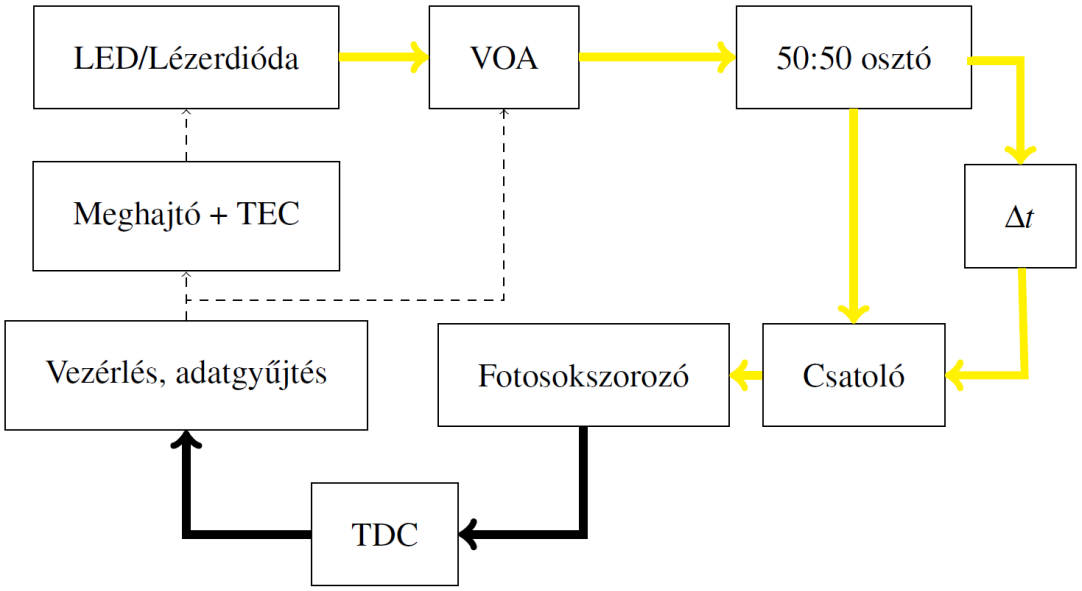
P(0) és P(1) is $\frac{1}{2}$ vagyis a mérés eredménye pontosan az esetek felében lesz 0 és az esetek felében 1 elegendően sok kísérlet esetén. Az elrendezés tehát valódi véletlenszám generátorként tud funkcionálni, feltéve, hogy meg tudjuk építeni fizikailag.

2.2.2. Tervezett fizikai architektúrák

A gyakorlatban egy kvantumbitnek sokféle fizikai megvalósulása lehet. Ezek közül alkalmazhatósága miatt az egyik legnépszerűbb alternatíva fotonok használata, már meglévő optikai eszközökkel való kompatibilitásuk miatt. A [5] dokumentumban bemutatásra került három tervezett fizikai architektúrák esetében is ez a reprezentáció lett a választott megoldás.

Útelágazáson alapuló QRNG

A 2.1 ábrán látható blokkvázlat a tárgyalt három architektúra közül a legegyszerűbben megérthető elrendezést ábrázolja. Egy fényforrás minden egyes véletlen bit előállításához periodikusan kibocsát egy-egy fényimpulzust, melyek egy-egy fotont tartalmaznak. A fényforrást egy lézervezérlő áramkör vezérli, valamint egy Peltier-elemből, termisztorból és a vezérlő szabályozóköréből álló termoelektromos hűtőrendszer felel az állandó hőmérsékletéért. A foton optikai szálon halad tovább egy nyalábosztó felé. A nyalábosztó két kimeneti ágának egyikébe egy késleltetőszakaszt helyezünk, ez az ág tehát hosszabb. A két ágat egy 2:1-es optikai csatolóval újra egyesítjük, majd egy egyfoton-detektort helyezünk annak kimenetére. Annak függvényében, hogy a foton „melyik úton érkezett” (ez itt a kvantumos információ) , a lézerimpulzus kibocsátásához képest rövidebb vagy hosszabb idő elteltével fog jelezni a detektor, az út-szuperpoziciót idő-szuperpozícióvá alakítva át. A beérkezési idő és a kibocsátási idő különbségének alapján (amit egy TDC, time-to-digital

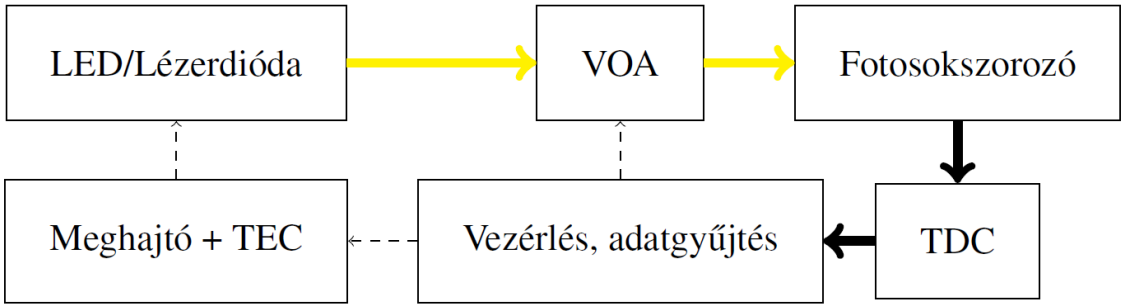


2.1. ábra. Az útelágazáson alapuló QRNG blokkvázlata. TEC: termoelektromos hűtésszabályzó; VOA: változtatható optikai csillapító; Δt: késleltetőszakasz; TDC: time-to-digital converter.

konverter méri) rendelhetünk hozzá az egyes jelzésekhez nullás vagy egyes biteket, így létrehozva a véletlen bitsorozatot.

A felépítésre jellemző hibák nagy része csak a bitsebesség csökkenésével jár. Kimeneti eloszlásra az osztó tökéletlensége, valamint a hosszabb úton jelentkező nagyobb csillapítás (a hosszabb útról kevesebb detekció lesz) lehet hatással. Mindkét eset a detektált nullás és egyes bitek relatív gyakoriságára van csak hatással.

Az ilyen felépítésű generátorokkal néhány Mbit/s sebességet lehet elérni. [6][7]



2.2. ábra. A fotonszámláláson alapuló QRNG blokkvázlata. TEC: termoelektromos hűtésszabályzó; VOA: változtatható optikai csillapító; TDC: time-to-digital converter.

Fotonszámláláson alapuló QRNG

A második architektúra (2.2 ábra) az előzővel szemben nem impulzusüzemű, hanem folytonos fényű forrást használ. Erre a célra megfelelhet akár egy fénykibocsátó dióda (LED), akár egy lézerdióda. A fényforrást konstans előfeszítő árammal hajtjuk meg, a kimenő optikai teljesítmény időbeli stabilitását pedig az előzőekhez hasonlóan egy Peltier-elemből, termisztorból és a lézervezérlő szabályozóköréből felépülő termoelektromos hűtőberendezés biztosítja. A fény optikai szálon keresztül halad, majd egy vezérelhető optikai csillapító végzi el a detektorra jutó teljesítmény finomhangolását. Detektorként ez az architektúra is egy fotonsokszorozót használ, ami egy vagy több foton beérkezését egy kimeneti impulzussal jelzi.

A véletlenség alapját ebben az esetben az adott T idő alatt beérkező fotonok (pontosabban az időablak során történő detekciók) száma jelenti. Ismert, hogy a fix idő alatt beérkező fotonok száma folytonos fényforrás esetében Poisson-eloszlást követ λT paraméterrel, ahol λ az időegység alatt kibocsátott fotonok várható értéke. Az így kapott értékekből többféle módszerrel állíthatunk elő véletlen biteket.

- Detekciók paritásából biteket előállítva ugyan 1-es bitgenerálási hatásfokot érünk el (egy időablak által generált véletlen bitek száma), de a Poisson-eloszlás aszimmetriájából fakadóan lesz egy eredő kiegyenlítetlenség. A bias az eloszlás várható értékének, tehát az átlagos fényteljesítménynek növelésével fokozatosan csökken. Továbbá előnyös tulajdonságú fotosokszorozók használatával beállítható úgy a teljesítmény, hogy a bias teljesen kiküszöbölhető. [8]

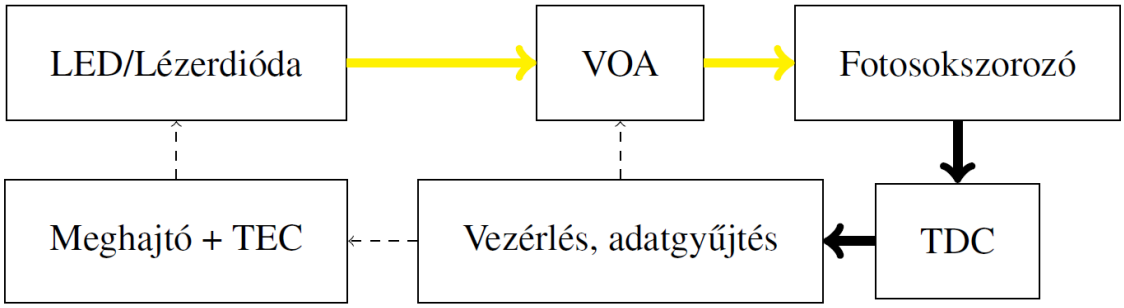
- Léteznek olyan módszerek, amelyek lényegében a radioaktivitás-alapú QRNG-k lassú órajeles módszerével [9] működnek. Ennek során két egymást követő időablakban mért detekciókat (n_1 és n_2) hasonlítunk össze, majd a komparálás eredményéhez rendelünk egyetlen bitet. Egyetlen időablak mért értékét csak egy összehasonlításban használhatjuk fel. Ennek a megoldásnak előnye, hogy nem okoz kiegyenlítetlenséget vagy korrelációt az egymást követő bitek között, de hátránya az alacsonyabb bitgenerálási hatásfok.

- Harmadik alternatívát jelenthet, ha egy-egy méréshez nem csak egy bitet rendelünk, hanem többet is. Ez a bitgenerálási hatásfokot és a bitsebességet megnöveli, viszont a feldolgozás kérdése bonyolultabb, mert az eloszlás nem egyenletes, valamint használatához körültekintő kalibráció szükséges

Az ilyen elven működő generátorok már 50 Mbit/s körüli sebességeket is képesek nyújtani [8], ami pontos működési módtól illetve használt eszközöktől függően akár ennél még gyorsabb is lehet.

Beérkezési időn alapuló QRNG

A harmadik választott architektúra (2.3 ábra) folytonos fényű fényforrást használ. Erre a célra megfelelhet akár egy fénykibocsátó dióda (LED), akár egy lézerdióda. A fényforrást konstans előfeszítő árammal hajtjuk meg, a kimenő optikai teljesítmény időbeli stabilitását pedig az előzőekhez hasonlóan egy Peltier-elemből, termisztorból és a lézervezérlő szabályozóköréből felépülő termoelektromos hűtőberendezés biztosítja. Detektorként ez



2.3. ábra. A beérkezési időn alapuló QRNG blokkvázlata. TEC: termoelektromos hűtésszabályzó; VOA: változtatható optikai csillapító; TDC: time-to-digital converter.

az architektúra is egy fotosokszorozót használ, ami egy vagy több foton beérkezését egy kimeneti impulzussal jelzi.

A véletlenség alapját ebben az esetben a detekciók közti időkülönbségek adják, melyeket egy TDC segítségével mérünk. Mivel Poisson-folyamatról beszélünk, a beérkezési idők különbségei exponenciális eloszlást követnek.Az így kapott digitális, kettes számrendszerben értelmezett értékeket közvetlenül nem tudjuk felhasználni, mert ez nagy kiegyenlítetlenséget eredményezne, ezért a fotonszámláláshoz hasonló komparatív módszert használhatunk, összehasonlítva az első–második, illetve második–harmadik detekció között eltelt időket. Kétféle órajelet használhatunk: folytonosat, valamint olyat, mely minden detekció után újraindul. Bizonyított, hogy az utóbbi előnyösebb, mert a folytonos órajel használata esetén korreláció léphet fel. [10]

A beérkezési időn alapuló QRNG hardver szempontjából nem különbözik a fotonszámláláson alapuló generátortól, az architektúrák különbsége a detekciók értelmezésében rejlik.

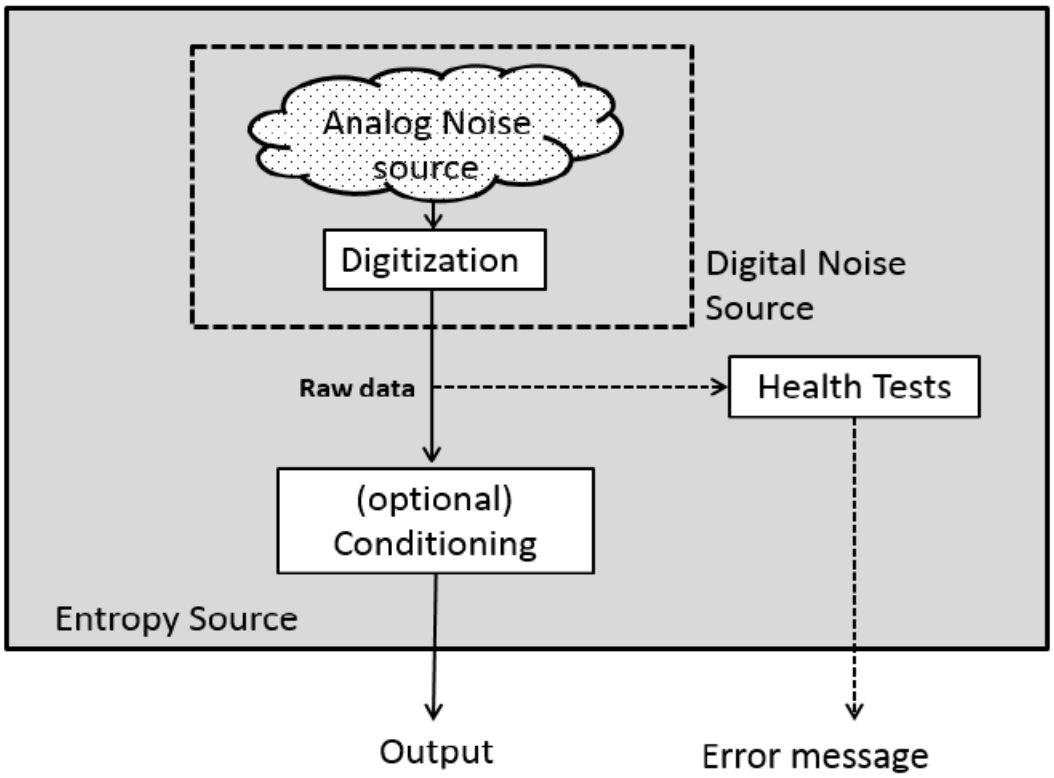
2.3. Generátortervezési ajánlások

Sok biztonságtechnikai eljárás keményen támaszkodik véletlen számokra, ezért a használt generátorokkal szemben is elvárt a megbízható, helyes működés. A kiadott adatfolyam ellenőrzése azonban éppen az elvárt véletlen jelleg miatt nehézkes. Egy lehetséges megközelítés statisztikai tesztek, próbák végzése, azonban ezek megbízhatóságával óvatosan kell bánni. Az előzőleg már említett véletlen jellegből adódik, hogy az adatsornak elvárt viselkedése például, hogy időről időre egy-egy teszten megbukjon. Ezen felül egy adott teszt csak egyfajta „minta” meglétét keresi a folyamban, ami miatt egy olyan ténylegesen nem véletlen viselkedés amit éppen nem vizsgálunk észrevétlen maradhat. Ebből adódóan valós rendszerek tervezésénél és hitelesítésénél érdemes a kimenet ellenőrzésén túl az egész rendszert is átvizsgálni az esetleges hibalehetőségek kiküszöböléséhez. Ennek megvalósításához nyújtanak egyfajta iránymutatást az amerikai NIST témába vágó ajánlásai. Az első dokumentum [11] determinisztikus generátorokkal, a második [12] entrópiaforrásokkal, míg a harmadik [13] generátorok felépítésével, a teljes rendszer konstrukciójával foglalkozik. Ezeken felül készítettek még egy statisztikai tesztcsomagot (STS - Statistical Test Suite) [14] ami 15 különböző tesztet tartalmaz a kimenetek vizsgálatához. Ezen felül

természetesen léteznek még különböző más tesztek, tesztgyűjtemények is. Az egyik talán legismertebb ilyen a „dieharder” csomag [15] , ami az 1995-ös ún.„diehard” tesztek bővítésének indult a 2000-es években. Azóta egy több száz tesztet tartalmazó csomaggá nőtte ki magát, többek közt tartalmazza az STS-ben használt 15 tesztet is.

Entrópiaforrások általános vizsgálata

Mivel a projektben tervezett generátor egy valódi entrópiaforrás, ezért az alábbiakban az ajánlások ide vonatkozó részeit vizsgáljuk. Feltételezzük, hogy a fizikai kivitelhez közvetlen kapcsolódó kihívások nagy része már kezelve van, a feldolgozóegységnek csak a beérkező nyers adatfolyamot kell kezelni. A 800-90B NIST ajánlás [12] is ezzel a helyzettel foglalkozik. Részletesen leír egy hitelesítési eljárást, amit a 2.4 ábrán látható általánosított ajánlott struktúra segítségével mutat be.



2.4. ábra. Általános entrópia forrás modellje a NIST ajánlásban

A hitelesítési eljárás célja, hogy a lehetséges hibákat, támadási felületeket kiszúrja. Ennek megfelelően, vannak benne tesztek a forrás működésére vonatkozóan (pl.:futás közbeni, újraindítási), valamint a forrással kapcsolatos egyéb feltételezéseink igazságáról való megbizonyosodáshoz is. Az általánosított struktúra 3 fő elemből áll. Egy zajforrásból, a forrás működését ellenőrző tesztekből, és egy opcionális a nyers adatok kondicionálását végző egységből. A zajforrás tulajdonképpen maga a fizikai entrópiát biztosító konstrukció. Mivel ezek általában analóg kimenettel rendelkeznek szükség van egy digitalizáló lépésre is. Ideális esetben ennek a kimenete tisztán lehetne az egész forrás kimenete. A

valós működés során azonban számolni kell a lehetséges hibákkal, képesnek kell lenni ezek észlelésére és ellensúlyozására.

A forrás egészségét vizsgáló tesztek ebből a szempontból különösen fontosak. A NIST ajánlásban megfogalmazottak szerint ezeknek három fő esetben kell jelezniük:

1. Ha a kimenet entrópiája jelentősen csökken.
2. Zajforrás hiba esetén
3. Hardver hiba esetén és nem megfelelő működésű implementációnál

Az ajánlás ezen túl három fajta ilyen tesztet különböztet meg, elvégzésük módja és ideje szerint:

Indítási tesztek a forrás kezdeti indításánál, valamint újraindításnál használatosak. Bizonyságot adnak arról, hogy minden az elvártak szerint működik még a forrás tényleges használata előtt, valamint arról is, hogy a legutóbbi indulási tesztek elvégzése óta semmi sem ment tönkre. Fontos, hogy a teszthez használt mintákat nem bocsájtjuk általános használathoz rendelkezésre amíg a tesztek le nem futottak. Ha ezek semmilyen hibát nem mutattak ki, utána a minták igény szerint elvethetők, vagy felhasználhatók.

Folyamatos tesztek, normál működés közben folyamatosan futnak a kimeneti adatokon. Ezeknek a célja, hogy lehetőséget adjanak a zajforrásban előforduló különböző lehetséges hibák észlelésére. Mivel ezeket a teszteket valós időben minden mintán elvégezzük, a téves pozitív jelzések valószínűségét érdemes alacsonyan tartani. Emiatt ezek általában durvább hibákat detektálnak csak nagy valószínűséggel. Továbbá itt korlátot jelent a támasztott erőforrásigény is éppen a folyamatos jelleg miatt. Megemlítendő még, hogy a tesztek egy adatfolyamon dolgoznak és mivel elvégzésük idejére itt nem szakítjuk meg a minták kiküldését, könnyen lehet, hogy mire elegendő bizonyosságot szereztünk a hibás működés beálltáról, a forrás már eme hibás működésből származó adatokat is szolgáltatott a külvilágnak.

Igény szerinti teszteknek elvégezhetőnek kell lenniük tetszőleges időben. Az indítási tesztekhez hasonlóak minőségben, valamint abban is, hogy a tesztek alatt használt mintákat csak az eredmények megléte után lehet a további felhasználásra bocsájtani. Az ajánlás megengedi ezen tesztek elvégzését újraindítással is, amennyiben az így használt indulási tesztek azonnal elvégzésre kerülnek.

Kondicionáló elem egy determinisztikus működésű elem. Használható például a kimeneti folyamban lévő bias eltávolítására, esetleges entrópia ráta növelésre, amennyiben erre szükség van. A megbízható működéshez biztosítani kell többek közt ezen algoritmusok megfelelőségét, valamint azt is, hogy az általuk felhasznált bemeneti adatokat kizárólag ezek az algoritmusok kapják meg, mivel ezek kiadásával a végső kimenet ebből támadhatóvá válna.

Az általunk fejlesztett rendszernek célja, hogy ennek az ajánlásnak megfelelően eleget tudjon tenni, valamint az egyéb működtetés során felmerülő igények megfelelő ellátása.

2.4. Statisztikai tesztek

A feldolgozó rendszerrel szemben elvárás, hogy a fizikai eszközt valamelyest monitorozza, annak meghibásodása esetén ezt jelezni tudja, valamint helyes működésnél erről valami-

lyen szintű bizonyosságot adjon. Ezt tipikusan, ha a hardver maga nem küld valamilyen hibajelzést, a beérkező adatfolyam folyamatos vizsgálatával, az így kapott eredmények figyelésével valósíthatjuk meg.

2.4.1. Véletlen számok statisztikai tesztelése általában

Az előzőekben már említett kimeneten elvégezhető tesztek, statisztikai tesztek. A véletlenségre vonatkozó konkrét mérhető mennyiség hiányában ezekre támaszkodhatunk, hogy a forrás esetleges működési hibáját a kimenet alapján detektálhassuk. Fontos megemlíteni, hogy pont a véletlen jelleg miatt a hiba tényét így is csak valószínűsíthetjük, biztosra nem tudhatjuk. Egy jól működő véletlen generátortól elvárható, hogy időről időre egy ilyen teszten megbukjon, éppen az lenne a kirívó, ha egyáltalán nem tenné ezt. Ahhoz, hogy egy hibáról viszonylag nagy biztonsággal állíthassuk, hogy tényleg hiba, jellemzően többfajta statisztikai tesztet is végzünk párhuzamosan. Így már jóval kisebb annak az esélye, hogy az elvárt kimenet éppen egyszerre több ilyenen bukik meg.

Egy teszt során az adatsort az ideális véletlen adatsorhoz próbáljuk hasonlítani valamilyen szempont szerint. A véletlenség egy valószínűségi tulajdonság, ezért ezt valószínűségi módszerekkel tudjuk tenni. Továbbá mivel végtelen sok minta létezik, amire az illeszkedést vizsgálni lehet, emiatt egy teszt sem tekinthető teljesnek. A statisztikai tesztek úgy vannak megfogalmazva, hogy egy kezdeti null hipotézist vizsgálunk velük. Esetünkben ez az, hogy a generátor igazi véletlen generátor. Ehhez köthető még az alternatív hipotézis, ami ennek tagadása: a generátor nem igazi véletlen generátor. Minden teszthez választható valamilyen releváns valószínűségi változó, amihez a véletlen feltételezés segítségével egy elméleti eloszlás számolható. Az eloszlásból meghatározható egy kritikus érték (tipikusan valamilyen messzi külső érték például a 99%-os pontnál), ami a teszt végén az adotsorból számított statisztikai értékkel kerül összehasonlításra. Ha a statisztikai érték a kritikus értéket túllépi, a null hipotézis elutasításra kerül. Statisztikai hipotézises tesztelés lehetséges kimeneteleit szemlélteti a 2.5 ábra.

TRUE SITUATION	CONCLUSION	
	Accept H_0	Accept H_a (reject H_0)
Data is random (H_0 is true)	No error	Type I error
Data is not random (H_a is true)	Type II error	No error

2.5. ábra. Statisztikai hipotézises tesztelés lehetséges kimenetei

Ha az adat véletlen, az esetek egy kis részében a null hipotézis elutasításáról születik következtetés. Ezt nevezzük 1. típusú hibának. Ha az adat nem véletlen kis valószínűséggel a null hipotézis elfogadásáról születhet döntés. Ezt nevezzük 2. típusú hibának. A maradék két esetben a levont következtetés a valóságnak megfelelő, ott nem történik hiba.

Az 1. típusú hiba valószínűségét szokták a teszt jelentőségi a szintjének is nevezni. Ez beállítható tesztelés előtt, jelölése α. Kriptográfiai alkalmazásokban ennek jellemző értéke 0.01 körüli.

A 2. típusú hiba valószínűségének jelölése β. Ellentétben α-val β nem egy fix érték. Egy adatfolyam többféleképpen is lehet nem véletlen és a különböző módokhoz különböző β tartozhat. Emiatt a számolása is jóval nehezebb mint α meghatározása.

A tesztek egyik fő célja a 2. típusú hiba valószínűségének minimalizálása, mivel biztonsági szempontból is az jelenti a legnagyobb kockázatot, ha egy rosszul működő generátorról azt hisszük, hogy mégis rendben üzemel. Az egyes hibavalószínűségek (α és β) függenek egymástól, valamint a mintavételezett sorozat hosszától. Ha ezekből kettő ismert, a harmadik is következtethető. Gyakorlatban általában mintahosszt és α-t választanak, majd ehhez olyan kritikus értéket keresnek, ahol legkisebb a 2. típusú hiba valószínűsége. A teszt statisztikai adatiból számolják ki az ún. P értéket, ami a lényegében az adott hipotézishez tartozó bizonyítékok erősségét összegzi. Jelen esetben ez a P érték annak a valószínűsége, hogy egy tökéletes véletlenszám generátor kevésbé véletlen sorozatot szolgáltat, mint a tesztelt adatfolyam az aktuális körülmények mellett. Így például ha $P = 0$, akkor a sorozotat teljes mértékben determinisztikusnak tűnik, míg ha $P = 1$, akkor pedig teljes mértékben véletlenszerűnek.

2.4.2. Elérhető statisztikai tesztcsomagok

Kifejezetten véletlenszám generátorok vizsgálatához jelenleg két nagyobb statisztikai tesztcsomag érhető el. Ezek a korábban már említett NIST által az ajánlásuk mellé kiadott STS (Statistical Test Suite) [14], valamint az 1995-ös Diehard tesztek [16] folytatásaként induló, mára viszont több száz különböző tesztet tartalmazó Dieharder [15] tesztcsomag. Mindkét csomagra igaz, hogy a megfelelő alaposságot a tesztek sokszínűségével igyekszik elérni. Tekintve, hogy az egyes statisztikai tesztek bonyolultabb matematikai számításokat (pl.: inverz gamma függvény számolása) is tartalmaznak, használatukkor és eredményeik értékelésénél figyelmet kell fordítani ezek numerikus stabilitására is.

NIST Statistical Test Suite

A NIST Statistical Test Suite [14], egy a NIST által kiadott statisztikai tesztcsomag, ami 15 tesztet tartalmaz, C-ben implementálva részletes dokumentációval. A csomag célja, hogy eszközként szolgáljon véletlen adatsorok teszteléséhez. A tartalmazott tesztek a következők:

1. Frequency (Monobit) Test
2. Frequency Test within a Block
3. Runs Test
4. Tests for the Longest-Run-of-Ones in a Block
5. Binary Matrix Rank Test
6. Discrete Fourier Transform (Spectral) Test
7. Non-overlapping Template Matching Test
8. Overlapping Template Matching Test
9. Maurer's "Universal Statistical" Test
10. Linear Complexity Test

11. Serial Test
12. Approximate Entropy Test
13. Cumulative Sums (Cusums) Test
14. Random Excursions Test
15. Random Excursions Variant Test

Látható, hogy a tesztek igen sokfélék, hogy a végtelen lehetséges nem véletlen mintából minél többet lefedjenek. Az első monobit teszt például csupán a sorozatban lévő egyesek és nullák számát vizsgálja, míg Mauer tesztje a sorozat tömöríthetőségét, vagy a DCT a spektrális eloszlást. Összességében egy viszonylag nagy területet átfogó tesztcsomag, amivel a generátorok jellemzőbb hibáit már szűrni lehet.

Dieharder tesztek

A Dieharder tesztcsomag eredetileg az 1995-be publikált diehard tesztek [16] bővítéseként indult, azóta viszont egy több száz tesztet tartalmazó csomaggá nőtte ki magát. Többek közt mostanra tartalmazza már a NIST STS-ben található teszteket is rengeteg más a szakirodalomban fellelhető további teszttel együtt. Célja, hogy a lehető legalaposabb csomagot nyújtsa, ezt tükrözi az is, hogy sok felhasználók által írt egyéb tesztet is tartalmaz. Maga a csomag Linux rendszereken egyszerűen telepíthető, parancssorból elérhető „dieharder" néven. Egyik további hasznos tulajdonsága továbbá, hogy támogatja generátorok tesztelését is (bitfolyamot olvas be), a megszokott már előre egy fájlba összegyűjtött nyers adatokon történő tesztelésen felül. Megjegyzendő még, hogy futása során nagy mennyiségű adatot használ fel eredményei kiértékeléséhez, ezért az STS csomagnál erősebb állítást jelent a tesztjein való bukás. Sok esetben az általa szolgáltatott eredmény már egy adott teszt többszöri futtatásából származó eredmény (lsd. KS tesztek a későbbiekben). Ennek hátránya hogy futtatása meglehetősen erőforrásigényes, az általunk épített rendszer valós idejű ellenőrzésére alkalmatlan.

3. fejezet

A fejlesztett rendszer

A fejezetben bemutatásra kerülnek a rendszer alapvető komponensei, ezek funkciója, illetve egyes különleges megoldások oka és módja. A fejezet elején ismertetjük a rendszer céjait és az előzetes terveket bevezetésként. Ezután vázoljuk a tényleges megvalósításra került architektúrát.

3.1. A rendszer céljai

A rendszernek valós időben kell teszteket futtatnia a bejövő adatokon, amennyiben valami meghibásodik azt jeleznie kell tudni. Esetleg indításkor adatokat kell szolgáltatnia a hardver kalibrációjának segítéséhez. Amennyiben a tesztek hibát jeleznek a kimenetet le kell tiltani, a felhasználók csak jó minőségű, ténylegesen véletlen kimenet kaphatnak meg. Tesztekre elérhetőek viszonylag jó implementációk (Dieharder [15], NIST [12]), viszont ezek nincsenek felkészítve valós időben futásra, ezt valamilyen módon meg kell oldani az architektúrának.

A hardver szükségképpen rendelkezik valamilyen bias-al (valamilyen hibával, amely miatt az eloszlás nem lesz teljesen egyenletes), ez rontja a kimenet minőségét, ennek kiküszöbölésére extractort kell alkalmaznunk. Ezek az extractorok olyan függvények, amelyek képesek egy bias-al terhelt bemenetből eltávolítani a biast vagy mérsékelni azt.

A kimenetet célszerű egy webszerveren keresztül szolgáltatni a felhasználóknak, így a rendszer később akár publikusan is elérhetővé tehető az interneten. Az adatokhoz gyorsabb, könnyebben feldolgozható alternatíva lehet egy egyszerű UDP szerver.

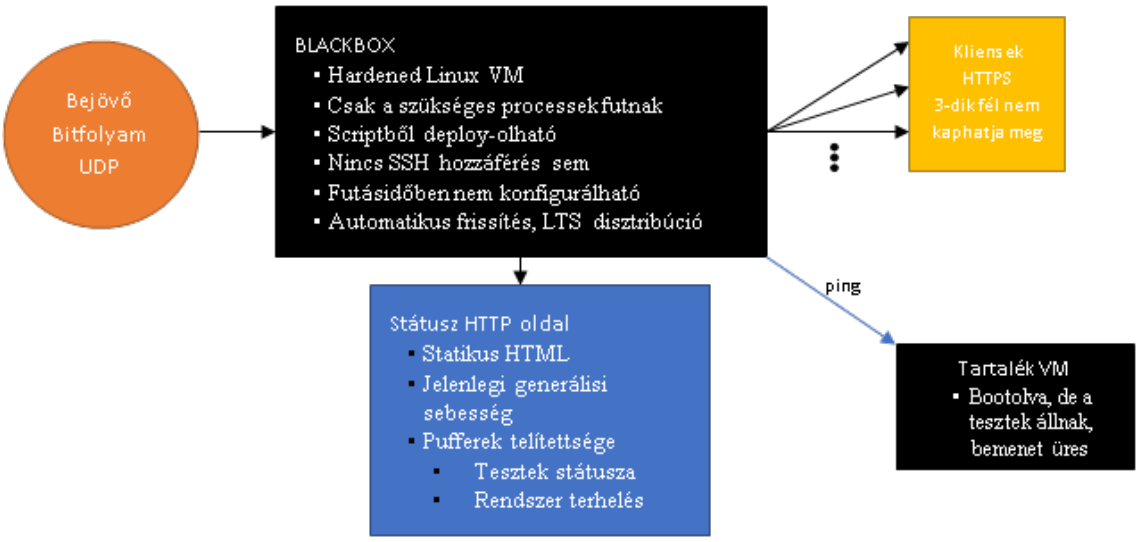
Tesztelési és fejlesztési szempontokat figyelembe véve is hatékony lehet egy virtuális gépen futtatott Linuxot alkalmazni. Ennek előnye, hogy jól tesztelhető, illetve minden fejlesztő futtathat belőle egy saját példányt. A publikusan elérhető tesztek általában nyílt forráskódú szoftverek C-ben vagy C++-ban írva. Ezek lefordítása is sokkal könnyebben megoldható Linux alatt.

A virtuális gép felvet egy olyan fejlesztési problémát, hogy egy idő után a szoftver futtatása más környezetben lehetetlenné válik, később nem lehet frissíteni az operációs rendszert. Ez a probléma úgy iktatható ki, hogy a szoftvert önmagában fordíthatóvá és installálhatóvá tesszük, átlátható és jól dokumentált módon. Ezzel a megoldással a hardver jeleit feldolgozó számítógépen akár natívan is futhat a rendszer.

A fejlesztés átláthatósága érdekében szükséges valamilyen verziókezelő rendszer al-

kalmazása. Mivel a fejlesztés Linuxon folyik ezért gitet alkalmazunk erre a feladatra. Az egyszerűbb fordítás és telepítés érdekében az egész projekt kódját egy git repository-ban tároljuk.

3.2. Tervezett és megvalósított architektúra



3.1. ábra. **Áttekintő rajz** – A főbb rendszerkomponensek áttekintő rajza

A detektor kimenetéről egy véletlen bitfolyam érkezik, de ez nem lesz tökéletesen egyenletes eloszlású, ezért szükséges mindenképpen valamilyen utófeldolgozás. Ezen egység három különböző feladatot kell ellásson:

- A bejövő bitfolyam tesztelése különböző erre alkalmas tesztekkel
  - Folyamatosan
  - Indításkor
- Extractor alkalmazása a kimenet minél „véletlenebbé” tétele érdekében és ennek tesztelése
- A kimenet eljuttatása több különböző felhasználó számára

A feladatra viszonylag jól alkalmazható egy Linux alapú virtuális gép, ennek fejlesztés, üzemeltetés és stabilitás szempontjából is vannak előnyei. Fejlesztési szempontból bármilyen laptopon vagy asztali gépen futtatható a feldolgozó prototípusa és könnyen tesztelhető az elérhető sebesség. Szintén gyorsítja a fejlesztést, hogy az elérhető véletlenszám generátor tesztkörnyezetek Linuxra könnyen lefordítható nyílt forráskódú szoftverek. Üzemeltetés szempontjából gyakorlatilag érzéketlen az alatta futó hardverre. Stabilitás szempontjából a szoftver működése jól tervezhető a disztribúció kiadásának életciklusán belül, mivel megfelelő célú kiadást választva a könyvtárak verziója évekig állandó marad.

A virtuális gép használata miatt könnyen implementálható hideg vagy meleg tartalék is, amely az eredeti rendszer sérülése, hibája esetén átveszi a feladatait. Ez akár fizikailag másik gépen is futhat. (Egy esetleges későbbi állandóan futó rendszer esetén lehet ez fontosabb.) A kiválasztott disztribúció az openSUSE Leap 15 lett. Előnye a viszonylag hosszú támogatási idő, a jól kiszámítható és stabil szoftveres környezet, illetve a systemd szoftver amelyet a rendszer nagymértékben használ.

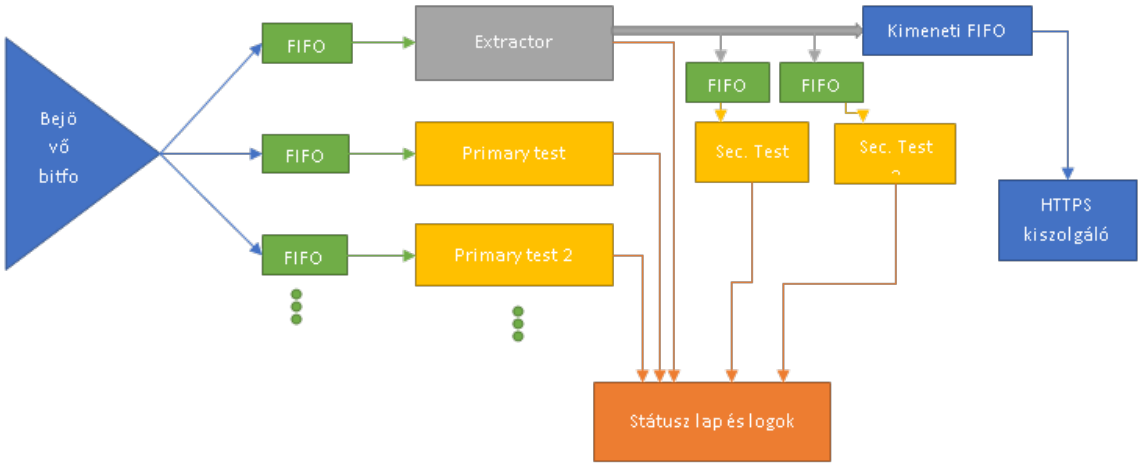
A bemenet érkezhet UDP-n, mivel egy véletlen bitfolyam esetén teljesen felesleges újraküldeni az elvesző biteket amennyiben folyamatosan keletkeznek újak, illetve nem szükséges titkosítás, mivel az a lokális hálózaton, akár egy közvetlen kapcsolaton érkezik. Ezzel ellentétben a nagy nehézségekkel előállított kimenő bitfolyamot érdemes újraküldeni, valamint ebben az esetben titkosítás is szükségessé válik. Utóbbira standard és könnyen lekérhető megoldás egy HTTPS kiszolgáló. A kliensek egyszerű kérésekkel tudnak ebben az esetben lekérni csak hozzájuk elküldött valódi véletlen számokat, ezzel lehetővé téve a kriptográfiai alkalmazást is. (Érdekes lehet a kiszolgálón magán is felhasználni véletlenszám generálásra a kimenetet, ez a kernel módosításával viszonylag könnyen megvalósítható lehet).

Közvetlenül hiba jelzésre mindenképpen szükség van, illetve a bejövő és a kimenő bitsebesség elérése is érdekes lehet. Erre létrehozható egy HTML lap, amely jelzi a tesztek státuszát, a sebességeket, illetve a rendszer terhelését. A lassabb nem valós idejű tesztek kézzel történő futtatására is itt lehet lehetőséget adni, természetesen megfelelő azonosítás mellett. Ezt a feladatot a monitord komponens valósítja meg, ez képes összegyűjteni a tesztek eredményeit és összesíteni azokat.

A rendszer állapotának naplózása is fontos szempont, mindenképpen szükséges valamilyen napló fájl alkalmazása. Alapvetően a rendszerben minden komponens egyenesen syslog-ba logol, ezzel igen könnyűvé téve az összesítést és a hiba keresést. A tesztek számára egy másik hibajelzési lehetőség megfelelő signalok küldése a monitord-nek amely ezeket összesíti.

Az architektúra viszonylag egyszerűvé teszi egy tartalék VM alkalmazását, ha erre szükség lenne. Jelenleg a fejlesztés még nem tart olyan szakaszban, amely ezt szükségessé tenné. Amennyiben a jövőben erre igény keletkezik, csak egy másik virtuális gép szükséges, ami ugyanúgy megkapja a bemenetet és az első hibája esetén elkezd működni.

3.3. Előzetes tervek



3.2. ábra. **Tervezett architektúra** – Az először tervezett architektúra. A FIFO pufferek kerültek lecserélésre egy stabilabb megoldásra.

Az elkészült megoldás jobb megértéséhez először ismertetjük az eredeti terveket, hogy utána ezek alapján könnyebben megérthető legyen az elkészült rendszer.

A virtuális gépen belüli szoftveregységekből (mivel elképzelhető osztály és külön program alapú implementáció is) alapvetően háromfajtára van szükség. A forrást analizáló elsődleges tesztek bemenetükön egy FIFO listán kapják meg a bitfolyamot, kimenetük valamilyen a státuszukat leíró mérőszám(ok), vagy egyszerűen egy igaz, hamis érték (átment, nem ment át). A második fajta a tényleges feldolgozó, kondicionáló lépéseket végző egységek. Erre alkalmasak lehetnek ún. extractor algoritmusok. A feldolgozás bemenete szintén a bejövő bitfolyam, kimenete viszont egy feldolgozott bitfolyam, amely általában jóval rövidebb, mint a bemenet. A harmadik fajta ilyen egység a másodlagos teszt, amely a kimenetet teszteli. Ezek hasonlóak az elsődleges tesztekhez, csak bemenetük más. Fontos azonban megjegyezni, hogy a másodlagos tesztek inkább szoftveres hibától védenek, valamint validálásra használjuk őket, ezért itt nem szükséges rájuk annyi processzor időt fordítani, mint a bemeneten. Nem szükséges különbséget tenni továbbá a különböző fajta tesztek között se (indulás utáni tesztek, folyamatos tesztek, időszaki manuálisan indítható tesztek), ezek csak abban kell különbözzenek, hogy mikor futtatjuk őket.

Ezen rendszer létrehozására két fajta megoldás is szóba jöhet. Ezek egyike a monolitikus, egy nagyobb program típusú megoldás (a részegységek osztályok például egy java környezetben). A másik lehetőség az inkább klasszikus UNIX megoldásnak tekinthető sok kis program, melyek külön processzekként futnak. Az első megoldás előnye, hogy nagyobb sebességet lehet vele elérni, feltételezett azonos tervezési szint és munkaóra mellett, viszont kevésbé stabil, mivel bármilyen nagyobb hiba összedönti az egész rendszert. Ezen megoldás előnye még, hogy az osztályok közötti kommunikáció viszonylag egyszerűen megoldható, illetve a naplózás is. Amennyiben nem szükséges a hardver maximális kihasználása a második megoldás sok szempontból előnyösebb lehet. Egy processz hibája esetén a többi képes tovább futni, a különböző egységeket kernel szinten lehet prioritizálni, befolyásolva az egész rendszer működését. Itt problémát jelenthet a programok közötti

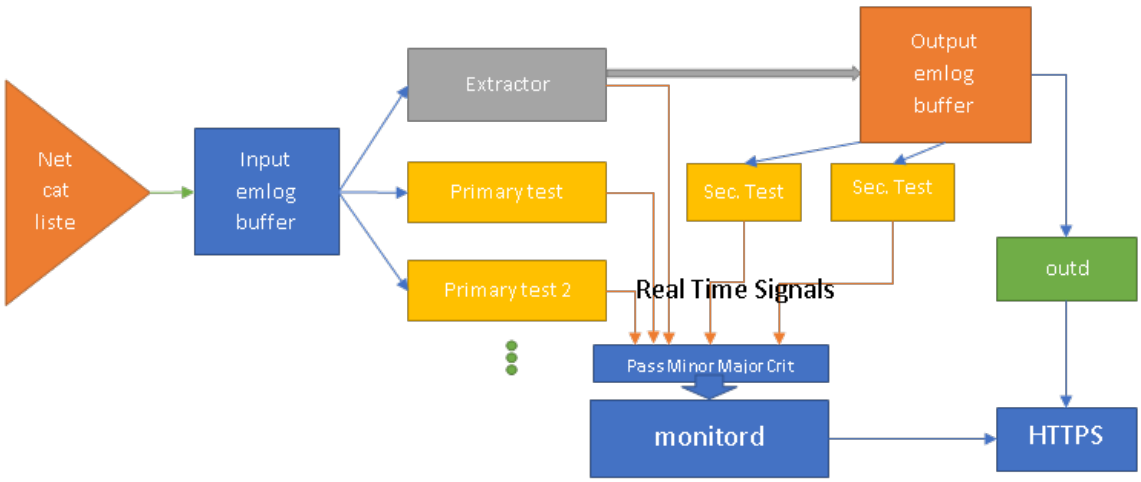
kommunikáció és a naplózás, de szerencsére ezekre mind léteznek stabil, jól használható standard megoldások.

A rendszer státuszának megjelenítésére és a véletlenszám generátor kimenetének lekérésére szükséges webszerver futtatása a rendszeren. Valószínűleg túl sok kapcsolatra nem kell számítani, ezért egy egyszerű Apache szerver, megfelelő szerver oldali scriptekkel alkalmas lehet a feladatra.

Mindenképpen problémát okoz, hogy az egyes tesztek és extractorok nem azonos sebességgel fognak futni. Erre kézenfekvő megoldás lenne igazodni a leglassabbhoz. Amennyiben ez az extractor azzal nem lenne probléma, ha viszont ez egy teszt, akkor lassítani fogja a feldolgozást, ami nem feltétlen szerencsés, ha a bejövő bitsebesség alá lassul. Erre megoldás lehet az esetleges lassabb teszteket kizárólag a bitfolyamból kivágott részletekre futtatni.

3.4. Az elkészült architektúra

A továbbiakban a tényleges, megvalósításra került architektúra részletes bemutatásával folytatjuk, a beérkező adat által bejárt adatútnak megfelelően végighaladva ennek alkotólemein.



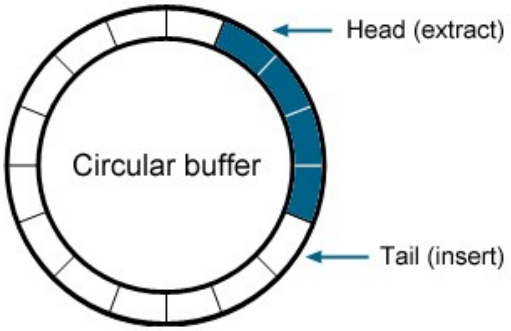
3.3. ábra. **Az elkészült architektúra** – A fő változást az emlog kernel alapú körpuffer bevezetése jelentette, így nem szükséges külön FIFO pufferek alkalmazása. Az egyes komponensek Unixos daemonként futnak. Az emlog miatt szükségessé vált egy plusz komponens az outd bevezetése.

3.4.1. netcat

Az UDP-n bemenetre várakozást a netcat alapprogram daemonként való futtatásával oldottuk meg. Ez kapott egy saját systemd service fájlt így automatikusan indul a rendszer indításával. A netcat kimenetét egy tee parancsba pipeolva írjuk a bemeneti pufferbe.

3.4.2. emlog

Az eredeti tervekben szereplő több FIFO puffert szerencsére sikerült kiváltani az *emlog* nevű kernel alapú körpufferrel. Ezt a megoldást eredetileg nagyon kevés memóriával rendelkező beágyazott eszközökhöz fejlesztették ki naplózásra. Hasonlóan működik, mint a kernel log (gyakran csak *dmesg*, a lekéréséhez használható parancs neve után), vagyis egy körpuffert alkalmaz, amely túlcsordulás esetén körbe fordul és felül írja a régebbi bejegyzéseket. Az emlog kernel modul egy körpuffert valósít meg karakteres eszközként (*character device*).



3.4. ábra. **Körpuffer szemléltetés** – A rendszerben az egyes tesztek külön nyitják meg a puffert, ezért mindegyik külön független olvasó pointerrel rendelkezik.

Az emlog használatával nincs szükség több, csak egy pufferre, nem kell minden egyes teszthez vagy extracktorhoz külön FIFO puffer. Ennek oka, hogy a különböző processzek saját olvasó pointerrel rendelkeznek, így egymástól függetlenül tudják olvasni a bemetet. Az emlog a tesztek sebességkülönbségének problémáját is megoldja, egy lassabb teszt nem feltétlen fogja tesztelni a bemenet egy részét, mert korábban körbefordul a puffer. Jó buffer méret választással megoldható, hogy ne legyenek a leglassabb tesztek se sokkal lemaradva az extractortól.

Az emlog sajnos sohasem került beolvasztásra a mainline kernelbe (a *mainline kernel* a kernel.org-ról elérhető hivatalos Linux kernel verzió elterjedt megnevezése), külső modulként fordul, ezért lefordítása nehézkes. A kernel modul használat ezen kívül kernel verzió és Linux függővé teszi a rendszert, de tekintettel arra, hogy milyen előnyöket nyújt ezen modul használata ez elfogadható veszteség. Hasonló fájl alapú körpuffert sajnos userspace-ben (*userspace* – azok a programok, amelyek felhasználói módban futnak ezért memória hozzáférésük korlátozott) nem lehetséges készíteni.

A rendszer által használt emlog verzió elérhető githubon [17]. Ez ellentétben az eredetivel fordítható az openSUSE Leap 15 viszonylag friss 4.12 verziójú kernelével is.

3.4.3. Tesztek és extractorok

Az emlog segítségével viszonylag könnyen megvalósítható a tervekben vázolt több processzes architektúra. Az extractorok illetve a tesztek daemonként futnak (A daemon olyan process amely a háttérben fut, a szokásostól eltérően van indítva, ezzel a szülő processze az init lesz.) Ennek a megoldásnak több előnye is van:

- Az egyes binárisok teljesen függetlenek, ezért egyenként cserélgethetők
- Amennyiben programhiba miatt az egyik teszt leáll a többi tovább fut
- Az egyes daemonok prioritása függetlenül állítható
- Bármikor hozzáadható új daemon a korábbiak módosítása nélkül

Alapvetően a UNIX daemonok naplózásra a standard syslogot szokták használni [18]. Ezt a szokást mi is átvettük, és minden extractor és teszt a syslogba logol. Ezzel központi helyen elérhető minden fontosabb bejegyzés időbélyeggel, a régebbi bejegyzések időszakos törlése is megoldott. A tesztek az eredmények naplózására használnak saját fájlokat is, ezek felhasználhatók például a rendszer helyes működésének bizonyítására.

A több komponens által használt függvényeket két osztott könyvtárba a *libqrng*-be és a *libtest*-be helyeztük el. A *libqrng* tartarmazza a daemonokhoz szükséges függvényeket, a *libtest* pedig a tesztekhez szükségeseket.

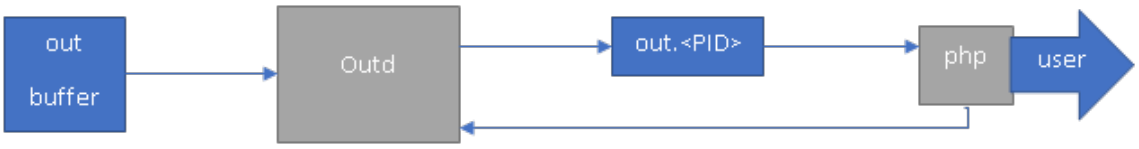
Az indítás során az adott daemon megnyitja a számára szükséges emlog puffereket, ezután daemonizálja magát és a tényleges feldolgozást egy végtelen ciklusban végzi. A felesleges processzor használat minimalizálása érdekében a daemonok egyre növekvő időt várakoznak, amennyiben a puffer üres. Minden egyes ciklus végeztével egy sched_yield() hívással [18] elengedjük a processzort a többi teszt számára.

3.5. outd

Az outd az emlog puffer miatt vált szükségessé. Amennyiben a kimenet nem rendelkezik egy saját pufferrel, szükséges egy köztes réteg bevezetése, amely lehetővé teszi azt, hogy a kimeneten egy adott bájt sorozat soha ne jelenjen meg kétszer (ez hatalmas biztonsági probléma lenne egy véletlenszám generátor esetén).

Az outd-ből adatot kell juttatnunk a HTML kimenetre. Ezt nagyban nehezítette, hogy az outd egy C-ben írt daemon, aminél jellegéből fogva az stdout, az stdin és az stderr át van irányítva a /dev/null-ba. A legfontosabb követelmény a rendszerrel szemben, hogy a kiolvasott kimenetet csak egy felhasználónak küldjük el.

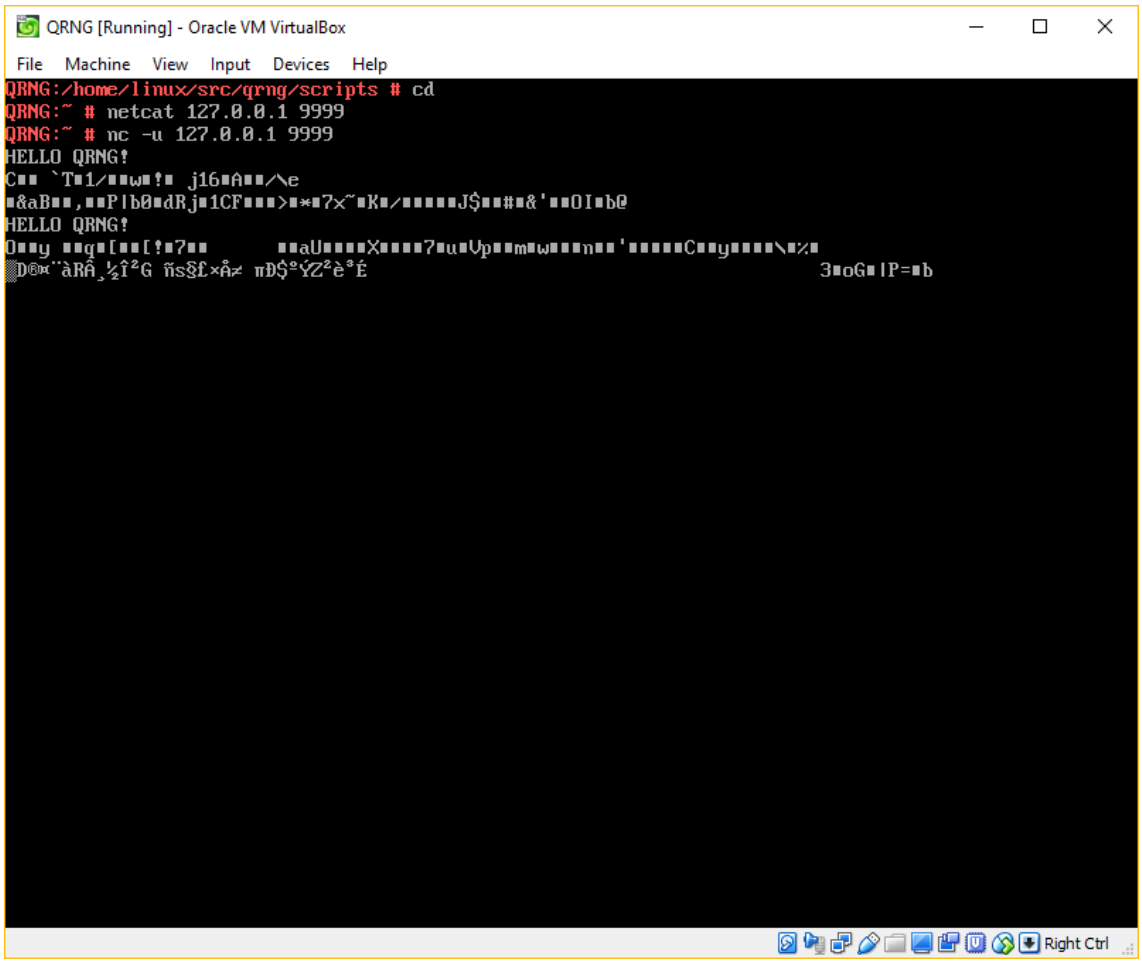
Ennek a problémának több lehetséges megoldása is van. Az első megoldás az 5. ábrán látható módszer volt. A php script egy SIGUSR1 signalt küldött az outd-nek, amely erre generált egy fájlt, aminek a fájlnevében szerepelt a php script PID-je. Ez elég körülményes megoldás volt, de ehhez képest meglepően jól működött.



3.5. ábra. **Az első megoldás**

A jelenlegi ennél jóval fejlettebb és szebb megoldás egy UDP szerver alkalmazása. Az outd minden olyan UDP csomagra, aminek tartalma a „HELLO QRNG!” karakterlánc

választ egy véletlen karakterlánccal. Ezt akár a netcat parancs segítségével is kipróbálhatjuk.



3.6. ábra. **Kimenet lekérése nc segítségével** – A HELLO QRNG! üzenet küldése után az outd válasza látható

3.6. monitord

A monitord feladata a rendszer állapotának figyelése, amire ún. *real time signalokat*[1] használ. Definiálva lett négy signal, amelyekkel egy teszt jelezni tudja az esetleges hibákat, illetve a sikeres futást is. Amennyiben a teszt bukik a Minor signalt használjuk, a Critical szoftver hibák jelzésére alkalmazható. A Majort jelenleg a tesztek általánosan nem használják, viszont a monitord beállítható számú (akár tesztenként) egymást követő Minor után generál egy Majort.

A monitord tartalmaz egy bináris fát az ismert processzek neveivel és PID-jeivel, ezekhez az interruptokat külön számolja, illetve percenként egy összesítést is készít. Abban

[1] A Linux operációs rendszer támogat ún. real time signalokat. Ezeket ellentétben a szokásos UNIX signalokkal a kernel egy várakozási sorba állítja. Ez jelen alkalmazáshoz elengedhetetlen, mivel minden signalnak meg kell érkeznie a monitord-hez [18].

az esetben, ha egy eddig ismeretlen processz küld signalt lekérjük a nevét egy system hívással és hozzáadjuk a listához. Ha ez valamiért nem sikerül a gyökér elembe logolunk.

A monitordtől az outdhez hasonlóan kérhetünk kimenetet. A kimenet HTML formátumú, ezért érdemesebb webböngészőből lekérdezni. Ez tartalmazza az egyes tesztek állapotát, illetve a státuszt jelző számlálókat. Amennyiben túl sok hiba gyűlik össze, a monitord képes blokkolni az outd kimenetét, illetve újra engedélyezni azt, ha a rendszer helyreáll.

Jelenleg 4 értéket vesz figyelembe ehhez a monitord, amik fájlban konfigurálhatók akár tesztenként is (amennyiben nem adunk meg az adott teszthez értéket, akkor az alapértelmezettet használja).

1. **Unstable** – Daemon újraindulás vagy critical signal esetén
2. **MinInc** – Az adott intervallumban a minor százalék kiugróan magas az indulástól számítotthoz képest (egy szorzó konfigurálható)
3. **MinPc** – A minor százalék egy előre beállított kritikus szint fölé megy
4. **MajPc** – A major százalék egy előre beállított kritikus szint fölé megy

A fentebbi négy értéket összeadva a monitord megkapja a blokk értéket és amennyiben ez a konfigurált érték fölé kerül blokkolja az outd kimenetét (ez megoldható egy signal használatával). Ha az érték a küszöb alá csökken újra engedélyezhetjük a kimenet. Fontos, hogy blokkolás esetén az outd végigolvassa a kimeneti puffert hibás kimenetek későbbi kiküldését elkerülendő. Az újonnan kiküldött értékeknek a blokkolás után kell érkezniük a kimeneti pufferbe.

Name	PID	I Pass	I Min	I Maj	I % min	T % min	T Crit
approxentropy	890	58	2	0	3.3%	5.6%	0
dft	891	456	26	2	5.4%	5.1%	0
linearcomplex	893	1	0	0	0.0%	0.0%	0
complex	894	701	45	4	6.0%	5.0%	0
monobit	895	523	21	1	3.9%	5.0%	0
serial	896	445	38	3	7.9%	7.2%	0
mauer	897	19	0	0	0.0%	6.0%	0
monobitblock	3436	499	21	1	4.0%	4.8%	0
runs	3437	459	25	2	5.2%	5.1%	0

Up&R	NoSig	Unstable	MinInc	MinPc	MajPc	Block
9	0	0	0	1	0	1

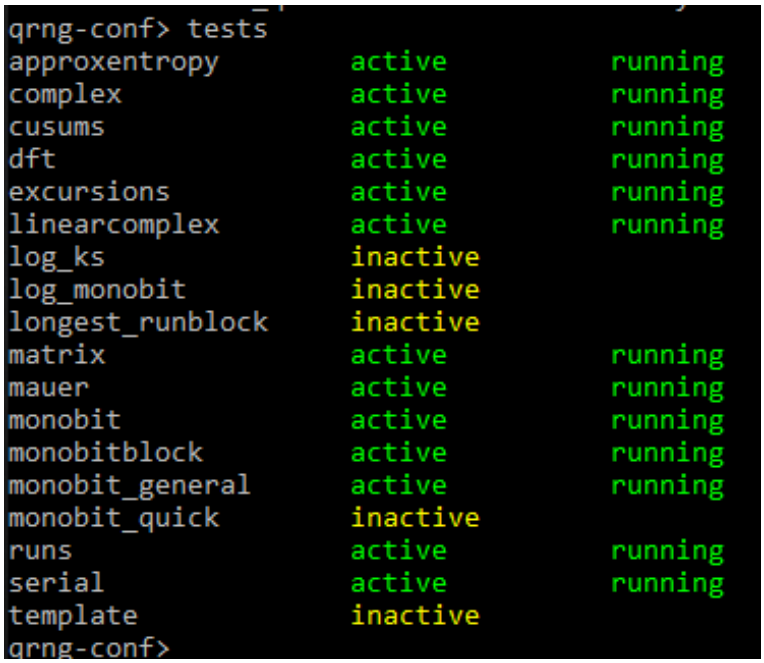
3.7. ábra. **A monitord kimenete a weböngészőben** – Az I-vel jelzett értékek az utolsó 1 perces intervallumra vonatkoznak, ezeket percenként nullázzuk. A T-vel jelzett értékek a monitord indulásától számolódnak. Alul láthatók az állapotot jelző számlálók. Ha minden rendben a tesztkktel akkor az Up&R értékét növeli. Ha még nem érkezett semmilyen signal, akkor a NoSig 1 lesz. Az Unstable számlálót növelik az újraindult, vagy critical signalt küldő daemonok. A MinInc akkor növekszik, ha a monitord configban megadott szorzónál nagyobb I % min / T % min. MinPc a kritikus minor százalékot elért daemonokat jelöli, MajPc hasonlóan csak major százalékra. A Block mező Unstable + MinInc + MinPc + MajPc, ez alapján dönt a monitord a kimenet blokkolásáról.

3.7. Egyéb segédprogramok

3.7.1. qrng-conf

A qrng-conf szkript egy köztes rétegként működik a systemd [19] felett. Segítségével állítható be, hogy milyen tesztek fussanak, illetve lekérdezhetjük az egyes kompenensek állapotát.

A tesztek listájának megjelenítéséhez a qrng-conf tests parancsát használhatjuk (akár qrng-conf tests alakban hívva, akár qrng-conf után a megjelenő parancssorba beírva működik). Az egyes teszteket activate, illetve disable paranccsal kapcsolhatjuk ki, be.



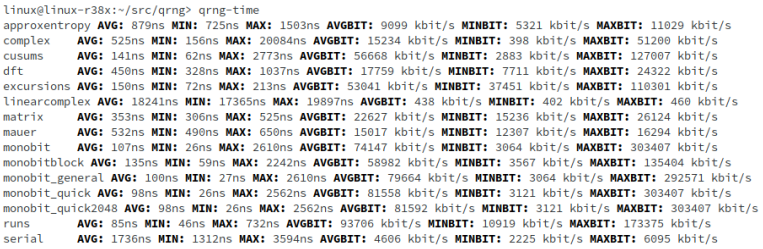
3.8. ábra. **A qrng-conf kimenete** – A tesztek listájában inactive címkével jelennek meg a kikapcsolt tesztek, active running címkével a futó tesztek. Amennyiben lenne olyan teszt, ami hibával állt le azt failed címkével jelezné.

A qrng-conf képes elindítani teszt véletlenszám generátorokat és ezekkel táplálni a bemenetet. Jelenleg ebből 4 fajta támogatott:

1. `generator zero` – Ez kizárólag nullát ír a bemenetre
2. `generator urandom` – A /dev/urandom segítségével írja a bemenetet
3. `generator random` – Hasonlóan az urandomhoz, csak a /dev/random-ot használja
4. `generator randfile@ <fájlnév>` - Ennek megadhatunk egy a /usr/share/qrng/blobs mappában található fájlt, aminek tartalmát írja a bemenetre, ha a fájl végére ér újrakezdi

3.7.2. qrng-time

A megfelelő tesztek kiválasztásához és a rendszer futásidejének méréséhez feltétlenül szükséges az egyes tesztek teljesítményének összehasonlítása. Ezt úgy oldottuk meg, hogy egy konstans (`MEASURE_TIME`) definiálása után minden egyes teszt a syslogba naplózza az 1 bájtra számolt futásidejét. A qrng-time szkript képes ezeket az értékeket összesíteni és tesztenként számol egy átlag, maximum és minimum értéket. Fontos azonban, hogy normál esetben ezt a funkciót kikapcsoljuk, mivel sebesség csökkenéssel jár.



3.9. ábra. **A qrng-time kimenete** – A qrng-time listázza az egyes tesztek átlagos, minimum és maximum futásidejét egy bájtra vetítve, illetve a bitsebességet kbit/s-ban.

4. fejezet

Rendszerben használt eszközök

A fejezetben bemutatjuk a rendszerhez készített feldolgozó eszköztárat, röviden ismertetve az általános teszteléshez implementált extractorokat, majd tárgyalva a jelenleg futtatható teszteket, valamint ezek megvalósítását a saját architektúrában. Kitérünk még rendhagyó igényeket kielégítő tesztek készítésének módjára, végül pedig a hosszútávú adatgyűjtést támogató megoldások kerülnek bemutatásra.

4.1. Extractorok

Az extractorok olyan algoritmusok, amelyek képesek egy gyengén véletlen forrásból egy erősebben véletlen kimenetet generálni, entrópiát nyernek ki egy adatsorból. Erre sokféle megoldás létezik (A legtöbb determinisztikus PRNG algoritmus is képes extractorként funkcionálni), az alábbiakban ismertetjük az általunk használtakat. Mivel a készülő fizikai hardver felől alapvetően jó minőségű bemenetre számíthatunk, így célszerűbb törekedni az egyszerű és stabil megoldásokra. Jelenleg legjobbank a NIST ajánlásokban [20] is szereplő SHA hash tűnik erre megfelelőnek, ezt a modern x86-os processzorok képesek gyorsítani is [21].

4.1.1. Von Neumann extractor

Az első extractor jellegű algoritmust Neumann János írta le 1951-ben [22]. Ez csak az egyesek és nullák számát képes kiegyenlíteni a bemenetben, más hibák ellen nem véd, ezért éles rendszerben nem érdemes használni. Maga az algoritmus nagyon egyszerű:

- Ha két egymást követő bejövő bit egyezik (00 vagy 11), akkor nem írunk a kimenetre
- Ha nem egyeznek, akkor az elsőt a kettő közül írjuk a kimenetre (10 esetén egyet, 01 esetén nullát)

A rendszerben ezt a *neumannext* program valósítja meg. Amennyiben ezt szeretnénk használni bekapcsolhatjuk a `qrng-conf activate neumannext` paranccsal.

4.1.2. SHA hash alapú extractor

Az `shaext` egy SHA-256 hasht használó extractor. Az OpenSSL könyvtárt használja, a bejövő adatokra egy SHA256 hast futtat és ezt küldi a kimenetre. A könyvtár képes alkalmazni a processzor specifikus hardveres gyorsítást, így a viszonylag bonyolult algoritmus ellenére gyorsan futhat.

4.2. Tesztek a fejlesztett rendszerben

Az általunk fejlesztett rendszerben a különböző tesztek az előzőleg már bemutatott módon képesek külön folyamatokként futni. Ilyenkor mindegyikük a hozzájuk tartozó bementi körbufferből olvas be (mintavételez). Így alapvetően a beérkező adatok igény szerinti mintavételezése minden egyes így futó tesztfolyamatra automatikusan megvalósul. Amennyiben azonban ugyanazon adatsoron szeretnénk végrehajtani több tesztet (például indításnál), arra is lehetőség van, hiszen semmi nem korlátoz arra, hogy egy ilyen futó folyamat csak egy bizonyos tesztet tartalmazzon. Az említett eset így megoldható egy bemeneti körbuffer egy kiválasztott összetettebb feldolgozó folyamathoz rendelésével, szigorúbb esetben esetleg még arra is ügyelve, hogy a bufferből minden adat feldolgozásra kerüljön. Az egyes folyamatok folyamatosan logolnak a monitorozó rendszerbe, így a feldolgozás állapota és eredménye egy központi helyről mindig elérhető, igény szerint szűrhető. Ennek megfelelően a rendszeren futó tesztekkel szemben támasztott elvárások a következők:

- Az egyes folyamatban futó tesztek egy bemeneti körbufferből olvasnak be adatokat, majd ezeket egy kimeneti körbufferbe továbbadják.
- A tesztek során kapott eredményeket a központi monitorozó rendszerbe adják tovább, a kapott eredmények logfile-ban is tárolásra kerülnek.
- Az eredmény tartalmazza a teszt kimenetét (ez az üzenet típusát is meghatározhatja a könnyebb szűrés érdekében), a kapott p-értéket és esetlegesen más, az adott tesztre jellemző értékeket további feldolgozás céljából.

Bár a 2.4.2-ben már említett tesztcsomagok formájában léteznek már implementált tesztek, ezek többnyire már meglévő nagy hosszúságú véletlen sorozatok, vagy a dieharder esetében akár generátorok egyszeri tesztelésére lettek tervezve. Ennek ellenére lehetne őket használni akár az általunk használandó forrás adatainak tesztelésére is, például a csomagok folytonos újrafuttatásával, azonban ezzel a feldolgozó rendszer pontosabb állíthatóságát és optimalizálhatóságát fel kéne adnunk. Egész csomagként való használatuktól emiatt a kritikusabb ellenőrzést végző részeknél eltekintünk. Megemlítendő, hogy mivel a dieharder csomag képes bitfolyamok feldolgozására, futtatása a mi rendszerünkben is lehetséges amennyiben a kimeneti folyamot szolgáltatjuk neki bemenetként. Számítási igénye miatt valós idejű ellenőrzésre nem alkalmas, azonban a sorozat megfelelően kis sebességű részletén futtatható, amennyiben igény van rá.

A csomagok által alkalmazott tesztekkel ezen felül még a fő probléma, hogy a beérkező véletlen adatokat a két implementáció általánosságban más módon tárolja, manipulálja a tesztjei elvégzése során. Emiatt, valamint az egyes adatkezelési módszerek adaptálásából

adódó feladat bonyolultsága miatt merült fel az igény egy alternatív, saját rendszerünkhöz igazított bit, illetve adatkezelés megírására.

4.3. Bitkezelés

A rendszer az adatokat bájtonként kezeli, ennek következtében számunkra is a legelőnyösebb és legegyszerűbb megoldás, ha a tesztjeink is képesek erre. Teljesen új bitkezelés készítésének nagy hátránya természetesen, hogy így a csomagokban használt teszt implementációk nem lesznek kompatibilisek ezzel és emiatt a teszteket újra kell írni a megváltoztatott környezetnek megfelelően. Ezt valamelyest ellensúlyozza viszont, hogy megfelelően elkészített támogató adatkezeléshez a tesztek megírása könnyebb, valamint a teljes mértékben saját megoldás további előnye a célhoz igazíthatóság és a későbbi könnyebb optimalizálhatóság is.

Az új bitkezelés készítésénél fontos szempont volt így a tesztek elkészíthetőségének könnyítése, valamint, hogy használhatóságában megfelelően igazodjon a rendszer többi részéhez.

Ez végül a `Bitblock` c++ osztály segítségével lett megvalósítva. Ennek feladata a körbufferből beolvasott adatok tárolása, valamint a tesztek számára egy könnyen kezelhető interface kialakítása az általuk ebből használni kívánt információ gyors elérésére.

Az így tárolt adat egy választható hosszúságú bájttömb. Emiatt el kell tárolni magát a bájttömböt és ennek a hosszát. Ezt ennek megfelelően a `chars` és `length` belső változók valósítják meg. Az ezzel járó memóriamenedzsent is az osztály feladata.

Az adat eltárolásán túl az osztály jelenlegi formájában biztosít a későbbi feldolgozáshoz és manipulációhoz is támogatást. Ennek megfelelően a következő segédfüggvényeket is tartalmazza:

- Definiálva van az `operator=` `operator+` és `operator+=` egyes bájttömbök egyszerű kezeléséhez. `operator=` esetén egyszerű másolat készül, míg `operator+` a két tömb konkatenálásának és `operator+=` konkatenálásnak és azonnali megfeleltetésnek felel meg.
- `.getbit(index)` függvény egy bit információjának kinyerésére. Itt „index” a kinyerendő bit indexe.
- `.bitstochar()` főleg debug célokra. A tárolt bitfolyam 0 és 1 karakterekként való könnyű megjelenítéséhez.
- `.count_ones()` a bájttömbben lévő egyesek számolására, opcionálisan megadható kezdő és vég bitindexszel. Mivel sok teszt felhasználja ezt az információt, ezért az osztály része lett.

4.4. Statisztikai tesztek megvalósítása

A fejlesztett rendszer lehetőséget ad tesztek egymás utáni vagy akár külön történő párhuzamos futtatására, megadott be- és kimeneti bufferrel. Ez a már korábban említett módon külön folyamatokkal valósítható meg. Ehhez a futtatandó tesztek egy egyszerűen

kezelhető egységes formára hozása célszerű. Ennek első lépése volt a saját bitkezelés írása. Használhatóság szempontjából cél a tesztek könnyű és flexibilis beillesztése a futtatandó folyamatok kódjába. Az egyes teszteket külön függvények tartalmazzák ennek biztosítása érdekében. Ezen függvények argumentumai is a lehetőségekhez mérten standardizálva vannak. (az állítható paraméterek és esetenként a kimenetek száma tesztenként változó). Egy tesztet megvalósító függvénynek a három első argumentuma így kötelezően az alábbi:

- `Bitblock& bits` referencia a bitekre, amin a tesztet futtatni szeretnénk. Ez az objektum tartalmazza a tesztelendő információt, a sorozat hosszát, valamint elérhetőek általa a leggyakoribb műveletek a sorozattal kapcsolatban. (Optimalizálásnál egy helyen elég változtatni)
- `double failcrit` a teszt sikerességének vizsgálatához a küszöbértéket adja meg, ami felett a teszt sikeresnek minősül.
- `double& pvalue` referencia a változóra ami a teszt számolt eredményét fogja tárolni.

Ezeken felül a legtöbb tesztnek vannak még jellegükből következő opcionális argumentumai. A teszt függvény visszatérési értéke `int`, melynek pozitív értéke a teszt sikerességét, nulla a standard működés melletti bukást, a negatív értékek pedig az esetleges hibákat jelzik. Ebben a formában egy teszt hozzáadása egy folyamathoz a 4.1 ábrán látható egyszerű kódrészlettel megvalósítható.

```
int processInput(int outfd, char* buff, int buffsize)
{
    if(log_reopen_scheduled) {                                  //logfile kezelés
        qrng_reopenlog(NAME, &logfile);
        log_reopen_scheduled = false;
    }

    //init static
    static Bitblock bits(0);                                    //indulási inicializáció
    //do stuff
    bits += Bitblock(buff, buffsize);                           //bitek gyűjtése

    if (bits.length < TESTBYTELENGTH) return 0;
    int stat = 0;
    int count = 0;
    double pvalue=0;
    int result = monobit(bits, FAILCRIT, pvalue, &stat, &count); //maga a teszt
    if (result == 0) {                                           //eredmények kezelése
        send_minor(&mondpid);
    }
    else if(result >0) {
        send_pass(&mondpid);
    }
    else {
        syslog(LOG_CRIT,"Example crit (failcrit<0)");
        send_crit(&mondpid);
    }
    //log result                                                //logfile-ba írás
    char s[128];

    sprintf(s, "%lf %d %d ", pvalue, count, stat);
    qrng_logstring(logfile, s);


    int tested = bits.length;
    //reset bit container
    bits = Bitblock(0);
    return tested;
}
//process indítása, demonizálás stb...
int main()
{
```

4.1. ábra. Példa tesztkód beillesztésére.

Ezen követelmények megvalósításához saját tesztek írása, a már meglévő tesztek megfelelő átírása, adaptálása szükséges.

Ehhez szerencsére támogatást nyújt a már említett bitkezelés, valamint a már meglévő statisztika tesztcsomagok dokumentációja is útmutatóként szolgálhat. Mivel a rendszernek célja, hogy a NIST által készített ajánlásnak is megfeleljen, valamint az ő általuk készített STS-nek alaposabb dokumentációja van, ezért a tesztcsomagban található algoritmusok mindegyike adaptálásra került. Az új formára adaptált teszteknek a fő felhasználási területe a rendszer valós idejű ellenőrzése folyamatos futás mellett, ezért az összes lehetséges teszt egyszerre történő használata a nagy számítási igény miatt nem célszerű. Helyette egy körültekintően megválasztott, a feladathoz optimális gyűjteményre van szükség.

A továbbiakban ismertetésre kerülő tesztek könnyebb megértését elősegítendő folytassuk egy egyszerűbb eset részletesebb vizsgálatával. Statisztikai tesztek futtatásához szükséges alapvető követelmények megfogalmazásához tekintsük a legegyszerűbb tesztet az STS-ből. Ez a „monobit” teszt.

4.4.1. Monobit teszt megvalósítása

A teszt célja, a véletlennek vélt bitfolyamban az 1-esek és 0-ák arányának összehasonlítása. Teljesen véletlen sorozat esetén ez statisztikailag közel esik 1/2-hez, ami azt jelenti, hogy közel azonos számú 0-át és 1-est tartalmaz a sorozat. A teszt ehhez az arányhoz való közelséget vizsgálja.

A többi teszthez hasonlóan itt is lényegében egy statisztikai próbát végzünk. A véletlen generátorhoz van egy alapfeltevésünk, miszerint feltéve, hogy a generátor véletlen (H_0), egy az általa szolgáltatott adatok felhasználásával kapott mértét mérésével, ennek értékei valamilyen meghatározott eloszlást kell, hogy kövessenek. Ez a meghatározott eloszlás a monobit teszt esetében az, hogy az 1-esek és 0-ák aránya 1/2. Ezután statisztikai eszközökkel kiszámítható valamilyen ε biztonsággal, hogy az adatok ennek megfelelőek-e vagy sem. Ha igen, azzal nem jutottunk sokkal előrébb, annyit mondhatunk csak, hogy nem biztos ε biztonsággal, hogy a bitfolyam nem véletlen, tehát nincs ok arra, hogy H_0-t cáfoljuk. Ha nem megfelelőek ennek az adatok az valamivel többet mond, mivel ilyenkor az alternatív hipotézist (H_a) fogadjuk el, ami azt állítja, hogy a generátor nem véletlen. Egy teszt gyakorlati kiértékelése általában a következőkből áll:

1. előfeldolgozás, esetleges reprezentáció váltás
2. teszt statisztika számolása
3. p-érték számolása
4. döntés valamilyen küszöbbel

Ez a monobit teszt esetében megvalósítva a következő: [1]

1. S_n összeget képzünk, mégpedig úgy, hogy végighaladva a bitfolyamon 1-es esetén +1-et, 0-ás esetén -1-et adunk az összeghez.
2. Teszt statisztika számolása: $s_{obs} = \frac{|S_n|}{\sqrt{n}}$ (n a bitsorozat hossza).
3. p-érték számítása: p-érték = $erfc(\frac{s_{obs}}{\sqrt{2}})$
4. Döntés: ha p-érték < küszöb akkor a teszt megbukott.

4.4.2. Matematikai eszköztár választás

Látható, hogy a számolások elvégzése, különösen a p-érték számolása egy komolyabb matematikai művelet. Jelen esetben ez egy „erfc” függvényhívást jelentett, mivel végül normális eloszláshoz viszonyítottunk. Más teszteknél ahol más elvárt eloszláshoz számolunk p-értékeket ez lehet más függvény. Tipikus ilyen még az „igamc” függvény ami χ^2 eloszlásokhoz tartozik. Emiatt szükséges megfelelő matematikai könyvtár választása. Főbb megfontolandó szempontok ezekkel szemben a pontosság és a sebesség. Az egyszerűbb és megbízható átemelés érdekében az STS-ben is használt cephes C matematikai függvénykönyvtárat használjuk. Későbbi optimalizáció szempontjából felmerülhetnek még a c++ „boost” könyvtárában található függvények is, csere előtt azonban egy körültekintő összehasonlító teszt szükséges.

[1] Az eset pontos matematikai hátterének ismertetésével a 4.5.1 részben még foglalkozunk. A többi statisztikai teszt pontos technikai leírása pedig [14] dokumentumban található.

4.4.3. NIST tesztek adaptálása

A NIST által kiadott STS-ben található tesztek átírása két fő tényező miatt volt szükséges.

- Bemenő sorozat tárolási módja: Az eredeti implementációban egy globális struktúra tárolja a teljes vizsgálandó sorozatot. Ez számunkra valós idejű vizsgálatok végzéséhez alkalmatlan, nehézkes a folyamatos frissítését kezelni, különösen több ezt használó, párhuzamosan futó folyamat esetén.

- Kimeneti formátum: Az eredeti csomag azonnal fájlokba írja az eredményeket, valamint helyenként olyan adatokat is gyűjt ami számunkra közömbös. Ennél egy egyszerűbb és használathoz jobban illeszkedő módra volt szükségünk.

Ezekre a problémákra adott megoldást a tesztek korábban már említett formára történő átírása. Megemlítendő, hogy a dieharder [15] csomaggal kapcsolatban felmerülő problémák is ehhez hasonlóak.

A már meglévő tesztcsomagokról elmondható, hogy a bennük implementált tesztek tényleges matematikai, algoritmikai tartalmán történő módosítás szükségtelen, a rendszer megbízhatósága, könnyebb tesztelhetősége szempontjából még akár kerülendő is. Ennek biztosításához szerencsére segítséget nyújt a csomagok részletes dokumentációja, valamint szükséges az átírt tesztek megfelelő összehasonlító tesztelése az eredeti csomagbeli implementációjukkal. Ez a tesztcsomagok forráskódjának elérhetősége, valamint az átírt forma modularitása miatt esetünkben könnyen elvégezhető volt. Az STS-ben található tesztek előnye, hogy összeválogatásuknál törekedtek az ortogonalitásra, ebből adódóan az egyes tesztek a sorozat megfelelően különböző tulajdonságait vizsgálják. A fennmaradó (a monobit teszt már részletes bemutatásra került) átemelt tesztek rövid leírása [14]:

Monobit teszt egy blokkon belül:

Ez a teszt az egyszerű monobit teszttel nagyrészt megegyezik, a fő különbség, hogy míg az egyszerű monobit teszt az egész sorozatot vizsgálja, itt annak M bit hosszúságú darabjait nézzük és ezek összesítéséből alkotunk statisztikát. Ennek megfelelően referencia eloszlás jelen esetben már nem egy normál eloszlás, hanem χ^2.

Opcionális argumentuma: M - a vizsgálandó blokkok hossza.

Futamok tesztje (Runs test)

Az egy sorozaton belüli futamok számát vizsgálja. Egy futam csupa 1-esek vagy csupa 0-ák megszakítatlan sorozata. Egy k hosszú futam például k egymás utáni 0-át jelent, ahol a k+1-edik bit már 1-es. Célja, hogy megállapítsa, a sorozatban található futamok száma egy véletlen sorozattól elvárhatónak megfelel-e. Megállapítható vele, hogy a két állapot (1-es futam vagy 0-ás futam) közti oszcilláció esetlegesen túl gyors-e vagy lassú. A teszt által számolt statisztika a talált futamok száma, aminek a referencia eloszlása a χ^2 eloszlás.

Opcionális argumentuma nincs.

Leghosszabb 1-es futam egy blokkon belül

Blokkon belüli 1-es futamok hosszát vizsgálja. Célja, hogy a leghosszabb ilyenről eldöntse, hogy a véletlen sorozattól elvárhatónak megfelel-e. Megjegyzendő, hogy az 1-es futamok hosszában történő eltérés eltérést jelent a 0-ás futamok hosszában is, így elég

csak az 1-esek vizsgálata. Az egyes blokkokban tapasztalt hosszakat a teszt eltárolja, ezen eredményekből kapott eloszlást viszonyítja a referencia χ^2 eloszláshoz.

A tesztnek nincsenek opcionális argumentumai a vizsgált blokkok hosszúságát automatikusan beállítja a vizsgálandó sorozat hosszának megfelelően. Megfelelő futásához legalább 128 bit hosszú tesztelendő sorozat elvárt.

Bináris mátrix rang teszt

A sorozatból készített diszjunkt mátrixok rangjait vizsgálja. Fix hosszúságú bitsorok lineáris függetlenségének vizsgálata. A sorozatból diszjunkt 32x32-es bináris mátrixokat készít, majd ezek rangjainak eloszlását hasonlítja a referencia χ^2 eloszláshoz. Az esetleges túllógó biteket (amikből már nem tud újabb 32x32-es mátrixot készíteni) eldobja.

Opcionális argumentuma nincs. Megfelelő futáshoz legalább 38912 bit hosszúságú sorozat ajánlott.

Diszkrét Fourier transzformált teszt

A sorozat Fourier transzformáltjában megjelenő csúcsokat figyeli. Célja a periodikusan megjelenő mintázatok detektálása, ami a feltételezett véletlen működésnek ellentmondana. Azt vizsgálja, hogy a 95%-os limitet meghaladó csúcsok száma mennyire tér el a várt 5%-tól. Referencia eloszlása a normális eloszlás.

Opcionális argumentuma nincs, 1000 bitnél hosszabb tesztsorozat ajánlott.

Nem átlapolódó mintaillesztéses teszt

Előre megadott minták előfordulási gyakoriságát számolja. Olyan generátorok kiszűrésére, amik egy adott aperiodikus mintát túl sokszor ismételnek. Mindig egy m bit hosszúságú ablakban vizsgálja a bitfolyamot. Ha nem találja meg a mintát az ablakot egy bittel elcsúsztatja. Amennyiben megtalálja a keresett mintát a minta után bitre csúsztatja az ablakot és folytatja a keresést. A megtalált minták előfordulási gyakoriságait viszonyítja a referencia χ^2 eloszláshoz.

Opcionális argumentuma: m - keresendő minta hossza.

A teszthez van egy (`/templates`) mintákat tartalmazó könyvtár a rendszerben.

Átlapolódó mintaillesztéses teszt

Hasonló az előző teszthez, azzal a különbséggel, hogy minta találatnál is csak egy bitet lépteti tovább a vizsgált ablakot. A megtalált minták előfordulási gyakoriságait viszonyítja a referencia χ^2 eloszláshoz.

Opcionális argumentuma: m - keresendő minta hossza.

Ez a teszt is az előzőleg említett (`/templates`) mintákat tartalmazó könyvtárat használja. Futtatásához legalább 1 millió bit hosszúságú tesztsorozat ajánlott.

Mauer „Univerzális" statisztikai tesztje

Ez a teszt a talált minták közti bitbeli távolságokat nézni. Célja, hogy eldöntse, a sorozat lényegesen tömöríthető-e. Egy lényegesen tömöríthető sorozatot nem tartunk véletlenszerűnek. A sorozatot blokkokra osztja, amiken belül a blokk elejét minták inicializálására, a fennmaradó részét pedig a statisztikai adatok gyűjtésére használja. Az így

összegyűjtött távolságok adják a teszt statisztikát, aminek referencia eloszlása normális eloszlás.

Opcionális argumentuma nincs, 1 millió bitnél hosszabb tesztsorozat ajánlott.

Lineáris komplexitás tesztje

Célja, hogy meghatározza a sorozathoz tartozó lineáris visszacsatolt shift regiszter hosszát. Ha ez túl rövid a sorozatot nem fogadjuk el véletlennek. A beérkező adatot blokkokra osztja, ezeken belül keresi a hozzájuk tartozó legrövidebb lineáris visszacsatolt shift regisztereket. Ezeknek a hosszát viszonyítja a referencia χ^2 eloszláshoz.

Opcionális argumentuma: M - a vizsgálandó blokkok hossza.

Futtatásához legalább 1 millió bit ajánlott, valamint M ajánlott értéke 500 és 5000 közötti.

Serial teszt

A különböző, akár átlapolódó m bit hosszúságú minták előfordulási gyakoriságait nézi. Célja, hogy megmondja a lehetséges 2^m bitminta gyakoriságainak eloszlása mennyire közelíti a véletlenszerű működés esetén elvárt uniformitást. A gyakoriságokból számolt végső statisztika referencia eloszlása a χ^2 eloszlás.

Opcionális paramétere: m - a vizsgálandó minták bithossza.

Futtatásához legalább n bit ajánlott, ahol n teljesíti a következő egyenlőtlenséget:

$$m < \lfloor log_2 n - 2 \rfloor$$

Becsült entrópia teszt

Az előző teszthez hasonlóan m bit hosszúságú minták előfordulási gyakoriságát vizsgálja, azzal a különbséggel, hogy két szomszédos hosszúságú (m és $m+1$) mintacsoport egymáshoz képesti frekvenciáját vizsgálja. Az ebből készített tesztstatisztika referencia eloszlása itt is a χ^2 eloszlás.

Opcionális paramétere: m - az első vizsgálandó mintahossz. A másik vizsgált hossz $m+1$ lesz.

Futtatásához legalább n bit ajánlott, ahol n teljesíti a következő egyenlőtlenséget:

$$m < \lfloor log_2 n - 5 \rfloor$$

Halmozott összegek tesztje

A nullától való maximális eltérést a véletlen sorozat által meghatározott véletlen séta esetében, ahol 1-s bitnél +1-et, 0-ásnál -1-ed adunk az összeghez. Ettől a maximális kitéréstől véletlen sorozatoknál elvárt, hogy nullához közel maradjon (sorozat hosszától függő valamilyen szórással természetesen). A figyelt eltérés referencia eloszlása egy megfelelően megválasztott Gauss eloszlás.

Opcionális paramétere: mode - Megadja, hogy elölről vagy hátulról indítjuk a sétát.

Ajánlott minimális sorozathossz: 100 bit.

Véletlen körutak tesztje

Az előző tesztnél már ismertetett véletlen sétát használva, azt figyeli, hogy ez a séta bizonyos állapotokat hányszor látogat meg. Ezen látogatások számából készít tesztstatisztikát, aminek referencia eloszlása a χ^2 eloszlás. A sorozatot külön vizsgálandó sétákra

bontja, ahol egy vizsgált rész kezdeti pontja a 0 állapot és végpontja a következő előforduló 0 állapot. Nyolc kitüntett állapothoz gyűjti a látogatások számát, ezek a következők: -4, -3, -2, -1, 1, 2, 3, 4.

Opcionális argumentuma nincs, futtatásához legalább 1 millió bit hosszúságú bemeneti sorozat ajánlott.

Módosított véletlen körutak tesztje

Az előző teszttel nagyrészt megegyezik, azzal a különbséggel, hogy a tesztsorozat vizsgálat szempontjából itt nincs részsorozatokra bontva, valamint a vizsgált állapotok száma nagyobb (-9-től 9-ig a 0 állapot kivételével). Referencia eloszlása egy megfelelően megválasztott Gauss eloszlás.

Opcionális argumentuma nincs, futtatásához legalább 1 millió bit hosszúságú bemeneti sorozat ajánlott.

4.5. Módosított statisztikai tesztek

Az eddig bemutatott és más meglévő statisztikai csomagok tesztjei is azzal a feltételezéssel élnek, hogy az elérni kívánt ideális eloszlás az egyenletes eloszlás. Az alkalmazási módok túlnyomó részében ez igaz, azonban lehetnek ez alól kivételek. Esetünkben egy ilyen kivételt jelenthet amikor feldolgozatlan közvetlen a fizikai eszköztől érkező bitfolyamot vizsgálunk. A legtöbb véletlenség előállításához mintavételezett folyamat nyers mintái nem egyenletes eloszlást követnek, ezt később a feldolgozás során a nyers adatfolyam megfelelő algoritmusokkal való transzformálásával érik el. Ez azonban egy további réteget jelent a felügyelő és a fizikai rendszer között, ami hátráltathatja a kellően gyors hibadetekciót. Felmerül tehát az igény olyan tesztek használatára, amikben az ideális eloszlás nem egyenletes, hanem ettől eltérő. Mivel a vizsgált tulajdonság (pl.: egyesek relatív száma, frekvenciatartománybeli kép stb..) nem, csak a referencia eloszlás változik, az előzőleg már bemutatott tesztek alapján megfelelő módosításokkal új, ezek módosított céleloszlásra értelmezett változatai készíthetőek. Ennek egy lehetséges megvalósítását vizsgáljuk a következő példával.

4.5.1. Monobit teszt általános Bernoulli-eloszlás esetén

Tételezzük fel, hogy a 2.2.2 során tárgyalt fizikai rendszerünkben egy nem ideális elemmel rendelkezünk aminek következtében az előállított bitek 45-55%-os arányban oszlanak el. Ezt a hibát méréssel már igazoltuk, létezését ismerjük, javítását a feldolgozási lépések során végezzük. A fizikai rendszer felügyeletéhez viszont szeretnénk monobit tesztet futtatni közvetlen a nyers kimeneten is. Ehhez az eredeti teszt egy módosított változatát használhatjuk. A módosításnak több módja is lehet:

1. Módosított referencia eloszlást használunk p-érték számításhoz.
2. A teszt statisztika számolási módján változtatunk úgy, hogy a számolt érték végső eloszlása a korábban alkalmazottat kövesse.

Az első eset megvalósítása a monobit esetében a 4.4.1 folyamán ismertett lépések közül a harmadik módosításával jár. Az $erfc()$ függvény (standard normális eloszláshoz tartozó

kiegészítő hibafüggvény) helyett az általunk kívánt eloszláshoz tartozó hibafüggvényt kell használni.

A második esetben az első és második lépésen kell úgy módosítani, hogy a létrejövő statisztika az egyenletes eset statisztikájával egyezzen. Az általunk készített példatesztben ezt a megoldást alkalmaztunk.

Az eredeti monobit teszt a Bernoulli-eloszlást követő változók összegeként keletkező binomiális összeg normális eloszlással való közelítését használja. Jelölje b_i a vizsgált bitsorozat i-edik bitjét. Ekkor a 4.4.1 lépéseinek megfelelően a bitsor -1 és +1 értékeket tartalmazó átalakítottja $X = 2b - 1$, a képzett összeg pedig: $S_n = X_1 + X_2 + ... + X_n = 2(b_1 + b_2 + ... + b_n) - n$.

Feltételezve, hogy az egyes bitek eloszlása $Bern(p = 1/2)$ Bernoulli-eloszlást követ, a belőlük képzett $B(n, p)$ összeg a De Moivre-Laplace tétel szerint nagy n esetén általánosan $\mathcal{N}(np, np(1-p))$ normális eloszlással közelíthető, ami az S_n által leírt véletlen séta esetére azt jelenti, hogy S_n $\sqrt{n}$-el normálva standard normális eloszlással közelíthető. Ez alapjánl a számolt statisztikai érték:

$$\frac{S_n}{\sqrt{n}} \sim \mathcal{N}(0,1) \rightarrow s_{obs} = \frac{|S_n|}{\sqrt{n}} \sim 2\mathcal{N}(0,1) - 1 \tag{4.1}$$

Amiből a p-érték már $erfc(\frac{s_{obs}}{\sqrt{2}})$-ként számolható.

Felhasználva a Bernoulli-eloszlás közelítését, készíthető monobit teszt tetszőleges $p \in (0,1)$ értékre. Az eredeti esetet tovább egyszerűsítve legyen a számolt összegünk egyszerűen a talált 1-esek száma(0-nál +0, 1-nél +1). Így $S_n \sim \mathcal{N}(np, np(1-p))$. Ebből az eloszlásból célunk egy $s_{obs} \sim \mathcal{N}(0,1)$ (most az abszolútérték képzést egy lépéssel későbbre hagyjuk) készítése. Ez könnyen megtehető S_n következő átalakításával:

$$s_{obs} = \frac{(S_n - np)}{\sqrt{np(1-p)}} \tag{4.2}$$

Innentől kezdve már csak p-értéket kell számolni ami abszolútérték képzés után már az eredeti esettel azonos.

Az általunk fejlesztett rendszerben ehhez az esethez tartozó teszt implementálásra került. Más tesztek más céleloszláshoz történő módosítása ehhez az előzőekhez analóg módon elvégezhető, azonban mivel mind az eloszlás mind a teszt a fizikai hardvertől közvetlen függ, ezek megvalósítása már annak pontos ismeretében ajánlott.

4.6. Teszteredmények feldolgozása

A futó tesztek eredményeit a monitorozó rendszer kezeli, valamint minden tesztekből nyert információ log formájában fájlokban is tárolásra kerül. Ezek tartalmazzák a feldolgozott bitmennyiségen és a kapott p-értéken túl, az esetlegesen gyűjtött tesztenként változó többlet statisztikákat is.

4.6.1. Tesztek sikeressége

A legalapvetőbb információ minden tesztről a megfelelés ténye, a tesztelt sorozat sikeresen átment-e az adott teszten vagy sem. Ennek központi gyűjtése indokolt, tekintve az így kapott statisztika egyszerű értelmezhetőségét, valamint azt, hogy már ez is egy áttekintő képet ad az egész rendszer állapotáról. Egy-egy teszt sikertelensége a monitorozó rendszerben külön hibaszinten jelenik meg. Túlságosan sok bukásnál indokolt valamilyen működési hibára gyanakodni, ilyenkor a rendszertől elvárt, hogy a kimenetet letiltsa, akciót hajtson végre a hiba elhárítására (értesítés, esetleges újraindítás). A statisztika tesztenkénti gyűjtésével szerencsés esetben (ha a tesztünk pont erre szűr) még egy adott hiba típusára is lehet következtetni.

4.6.2. p-értékek gyűjtése és KS tesztek

A sikerességen kívül a tesztekből eltárolásra kerül a döntéshez használt ún. p-érték is. Ez az az általános statisztika ami megmondja a végzett próba szerint mennyire valószínű, hogy az adott bitsorozat egy tökéletes véletlenszám generátorból származik. Ezeknek a p-értékeknek is van egy elvárt eloszlása (egyenletes), ennél fogva lehet rajtuk további statisztikai próbákat végezni. Az ilyen tesztek az ún. „KS tesztek ” [23][24]. Ezek során azt vizsgáljuk hogy a kapott p-értékeink eloszlása mennyire tér el az egyenletestől 0 és 1 között. Végrehajtva egy ilyet, egy újabb p-értéket kapunk, így ezeket megfelelő körültekintéssel (nem szabad megfeledkezni a numerikus pontatlanságról sem) rekurzívan ismételgetve elméletben lehetséges az eddig gyűjtött adatainkról egy átfogóbb eredmény számolása. Ennek előnye, hogy amennyiben a generátorunk nem véletlen működést produkál, az így kapott statisztika tart a nullához. Számolása viszont meglehetősen erőforrásigényes. Készítettünk egy általános, a log fájlokon KS próbát végezni tudó programot. Ezzel igény szerint egy-egy fájlból tesztre jellemző összesített p-értékek számoltathatók, valamint automatikusan is hívható logkezelés folyamán, régebbi log fájlok tömörítése előtt. Így hosszútávú működésnél használható nagyobb mennyiségű teszteredmény aggregálására. Már aggregált és még nem aggregált p-értékek keverését elkerülendő egy próba elvégzése után a számolt p-értéken túl az ehhez felhasznált összesített bitszámot is eltároljuk.

4.6.3. Tesztek egyedi adatai

Bizonyos tesztekből egyéb kiegészítő információ gyűjtésével lehetséges azoknak egy kiterjesztettebb változatát használni.

A monobit tesztet példának használva:

Amennyiben a teszt végeztével elmentjük a teszt során számolt összeget is (1-esek és 0-ák számának különbsége), valamint a tesztelt bitsorozat hosszát, ezek felhasználásával egy megfelelően módosított monobit teszt képes két ilyen elmentett teszt eredmény egyszerű összevonására, hiszen a tesztstatisztika számolásához szükséges adatok a két előzetes tesztből lementésre kerültek ($s_{obs} = \frac{|S_n|}{\sqrt{n}}$). Egy ilyen tesztnek így maga a bitsorozat nem is szükséges bemenete. Ezt a kiterjesztett tesztet rendszerünkben „monobit_statconcat” néven implementáltuk. Segítségével a monobit teszt által generált logfájlon egy ilyen átfogó tesztet végezhetünk el.

Jelenleg csak ez az említett eset került megvalósításra, azonban megfelelő többlet

adatok gyűjtésével, hasonló elgondolás szerint az alábbi tesztekhez lehet még igény szerint összesítő tesztet készíteni:

1. Futamok tesztje: a tesztelt bithossz, egyesek aránya és a futamok számának gyűjtésével

2. Leghosszabb 1-es futam egy blokkon belül tesztje: tesztelt blokkok számának, hosszának, valamint a készített cellák értékeinek elmentésével

3. Bináris mátrix rang teszt: blokkok számának, teljes rangú és teljes-1 rangú mátrixok számának gyűjtésével

4. Halmozott összegek tesztje: tesztelt bithossz, maximális eltérés, bitsorozat végén lévő eltérés gyűjtésével

5. Véletlen körutak tesztje: a bithossz, és az egyes állapotokhoz tartozó belső statisztikák, valamint bitsorozat végén lévő eltérés gyűjtésével

5. fejezet

Rendszer tesztelése

A fejezetben különböző tesztesetekkel bizonyosodunk meg rendszerünk, és részei helyes működéséről. Ehhez először szoftveres generátorok vizsgálatával referencia generátort választunk, majd ennek felhasználásával, egy hibás működésű generátor szimulált esetét tekintjük. A fejezetet az adaptált eszköztár elemenkénti futási teljesítményeinek vizsgálatával zárjuk.

5.1. Tesztelés célja

Mivel a felügyelendő fizikai hardver jelenleg még csak az építési fázisban tart, ezért a rendszerek együttműködését még nem állt módunkban vizsgálni. Más generátorok használatával azonban lehetőségünk van mind a rendszer, mind az ebben használt eszközök általános vizsgálatára. Az ebből származó eredmények későbbi személyre szabás esetén is hasznosak lehetnek. A következőkben a rendszer képességeinek szemléltetésére, vizsgálatára használjuk őket.

5.2. Referencia generátor választás

Már széleskörben használt, könnyen elérhető generátorok használata előnyös a rendszer teszteléséhez. Mivel ezek viselkedése ismert, vizsgálni lehet vele, hogy a feldolgozó rendszerben a feldolgozó lépések megfelelően működnek-e, az ismert, előzetesen elvárt eredményeket kapjuk-e. Másik előnyük a megfelelő hozzáférhetőség. Ez lehetőséget ad arra, hogy miután a feldolgozórendszer helyes működéséről megbizonyosodtunk, annak különböző szimulált hibákra adott viselkedését vizsgáljuk. Egyszerű használatuk és testreszabhatóságuk miatt a Crypto++ [25] c++ könyvtár által biztosított véletlenszám generátorokat választottuk erre a feladatra. Ezek a generátorok a 2.1.1 során már ismertetett csoportba tartoznak. Determinisztikus algoritmust használnak, aminek folyamatos inicializálására lehetőség van akár a számítógép belső entrópiaforrásának használatával is. A későbbi tesztek elvégzése előtt így a következő generátorokat vizsgáltuk:

- AutoSeededRandomPool
- AES Cypher PRNG-ként használva beépített entrópiaforrással nem blokkoló üzemmódban seedelve

- AES Cypher PRNG-ként használva számlálóval seedelve
- AES Cypher PRNG-ként használva beépített entrópiaforrással blokkoló üzemmódban seedelve

5.2.1. Tesztelési elrendezés

A generátorok a host operációs rendszerben egy egyszerű küldő programba építve generálják az udp-n elküldésre kerülő véletlen adatfolyamot. A rendszer egy virtuális gépen fut, ami egy virtuális hálózaton van a host géppel, a 666-os porton várja a véletlen bitfolyamot. A rendszerben a 15 STS-ben definiált statisztikai teszt megfelelőjét futtatjuk, előfeldolgozás nélkül, 0.01-es bukási küszöbbel. Az architektúra pontos működését a 3.4 fejezetben már tárgyaltuk. Eredményként a monitorozó rendszer által gyűjtött statisztikát, valamint a fájlokba logolt adatokat tekintjük.

5.2.2. Eredmények

Az egyes generátorok kellően hosszú ideig történő vizsgálata után (több GB-nyi adat) a monitorozó rendszer által gyűjtött statisztika, valamint a logként tárolt eredményeken futtatozz átfogó tesztek eredményeit összehasonlítva döntöttünk a további tesztekhez referenciaforrásként használni kívánt megoldásról.

A vizsgált generátorok nem valós véletlenséget szolgáltatnak, emiatt a hosszútávú vizsgálattól elvárható, hogy valamilyen gyengeséget mutasson ki. Rövidtávon, illetve kevésbé mélyebb vizsgálatokra, már elvárható, hogy véletlen jelleget mutassanak. Ezek az előfeltevéseink.

Monitorozó rendszer statisztikái

A monitorozó rendszer gyűjti, hogy egy teszt hányszor volt sikeres (Pass), illetve hányszor bukott (Min). Öt egymás utáni bukást külön számlálóval jelez (Maj), valamint van egy számláló fenntartva egyéb működési hibák számára (Crit). A futtatott tesztek között vannak kombinált tesztek, amelyek futásuk során egyszerre több tesztet valósítanak meg. Ezek a complex, template és excursions tesztek. A tesztelt bitsorozat hossza a 4.4.3 során ismertetett ajánlásoknak megfelelően tesztenként változó. Különleges teszt a serial, mivel futása során két p értéket számol, emiatt csak akkor nem bukik, ha mind a két érték küszöb feletti, valamint az excursions tesztek, amelyek futásához egy statisztikai minimum elérése szükséges. Ennek hiányában csak a Maj. számlálót növelik.

AutoSeededRandomPool

Az AutoSeededRandomPool a Crypto++ könyvtár egy beépített generátora. A 5.1 ábrán található eredményeken jól látszik a beállított bukási arány hatása. A futtatott tesztek közel ilyen arányban voltak sikeresek. Ez az általunk is a generátortól elvárt eredmény. Ettől eltérést a serial teszt esetében látunk a 5.2.2 részben tárgyaltak miatt. Enyhe eltérés tapasztalható még a dft és mauer teszteknél is. Ez esetleges későbbi mélyebb vizsgálatra adhat okot, azonban lényegi következtetést ebből még nem érdemes levonni. A

Name	PID	T % min	T Pass	T Min	T Maj	T Crit
serial	867	1.6%	394262	6580	0	0
monobit	868	0.9%	5725214	54527	1477	0
complex	869	1.0%	3238052	31864	286	0
runs	870	1.0%	197322	2002	0	0
approxentropy	871	0.9%	12345	118	0	0
linearcomplex	872	0.8%	467	4	0	0
monobitblock	873	1.0%	1604819	15474	7	0
dft	875	1.3%	49154	630	0	0
mauer	876	1.3%	1292	17	0	0
excursions	1790	1.0%	7543	75	877	0
longest_runbloc	1793	1.0%	395367	4152	0	0
matrix	1795	1.1%	1294	15	0	0
template	1802	1.3%	1117	15	0	0
cusums	1804	1.0%	1604660	15782	7	0

5.1. ábra. AutoSeededRandomPool generátor statisztikája

Major számláló esetében látható, hogy a monobit, complex és excursions teszteknél viszonylag magas értékek születtek. Ez a complex teszt esetében azzal magyarázható, hogy benne több teszt fut együtt egyszerre, így ezek bukása gyenge minőségű bitfolyamrészeknél korrelált, ezzel az vizsgált bukások egymásutániságának eloszlását is befolyásolva. Ezt támasztja alá, hogy az általa tartalmazott tesztek külön történő futtatása (cusums és longest_runblock) nem mutat ilyen viselkedést. A monobit teszt is váratlanul magas statisztikát mutat ezen a téren, az összes bukás 2.7%-át. Ebből arra lehet következtetni, hogy egyes bukások egymással korreláltak, csoportosan jelentkeznek. Ezt támasztja alá, hogy amennyiben egyszerre hosszabb bitsorokat vizsgáló monobit tesztet(az ábrán nem látszik, utólagosan futattuk) alkalmazunk ez már nem figyelhető meg. Ez a jelenség valószínűleg a számítógép nem jó minőségű belső entrópiaforrásának tudható be, az általunk használt sebesség mellett kifogy az entrópiából. Az excursions tesztek esetében a Maj jelzések túlnyomóan (Több Maj mint bukott eset) egy beépített statisztikai küszöb el nem éréséből származnak, aminek jelzésére ezt a számlálót használjuk. A feltételt a teszt elvégzése előtt vizsgáljuk, teljesülése hiányában pedig a tesztet nem végezzük el. A feltétel jellegének ismeretében a monobit eseténél már említett időszakos minőségromlásra lehet következtetni. [1]

AES Cypher PRNG-ként használva beépített entrópiaforrással nem blokkoló üzemmódban seedelve

Titkosításhoz használt AES titkosító megfelelően seedelve használható PRNG-ként. Jelen esetben ezt a seedelést a számítógép beépített entrópiaforrása által szolgáltatott szekvenciával végezzük. A forrástól nem blokkoló üzemmódban kérünk adatokat, ami azt jelenti, hogy amennyiben a forrás entrópiából kimerül, abban ez esetben is próbál valamilyen sort szolgáltatni a kérőnek. A blokkoló mód ennek ellentettje, kimerülésnél addig blokkolja a kimenetét amíg elégséges entrópiát nem gyűjt. Ennek pontos megvalósítása azonban eszközönként változik, minőségi garanciát nem ad. Az 5.2 ábra eredményei az előző generátoréhoz hasonlóak. Különbséget jelent viszont a monobit tesztek csoportos bukásának aránya, ez itt csak az összes bukás 0.4%-a. Ebből arra lehet következtetni, hogy az AES cypher hatékonyabban fedi el a beépített entrópiaforrás hibáit.

AES Cypher PRNG-ként használva számlálóval seedelve

Az AES cyphert lehet használni determinisztikus inicializálási értékekkel is. Bár ez biztonságos kommunikációhoz nem ajánlot, a determinisztikus kimenet miatt, azonban esetünkben visszafejthetőség kiküszöbölése nem, csupán véletlennek tűnő sorozat előállítása a cél. A seedelést egy egyszerű számlálóval valósítjuk meg. A 5.3 ábra eredményei hasonlítalak az eddigiekhez. Különbséget jelent, hogy a csoportos bukásokat detektáltuk két új teszt esetében is (monobitblock és cusums). További változás még, hogy a monobit csoportos bukások aránya az entrópiaforrásból seedelt eset kétszerese itt.

[1] Az excursions tesztek véletlen körutak során felvett állapotok eloszlását vizsgálják. A kérdéses feltétel ezeknek a körutaknak a megfelelő számossága. Sorozatba kiegyenlítetlenséget (monobit pont ezt figyeli) hozó hibák, az adott sétákat az egyik irányba eltolhatják, így csökkentve a kezdőállapotba való visszatérés esélyét, ami vizsgált statisztika csökkenéséhez vezet.

Name	PID	T % min	T Pass	T Min	T Maj	T Crit
linearcomplex	883	0.4%	664	3	0	0
approxentropy	884	1.0%	19980	200	0	0
monobitblock	885	0.9%	2605931	24505	2	0
monobit	886	0.9%	5461796	51907	251	0
runs	887	1.0%	319577	3204	0	0
mauer	888	1.0%	2099	21	0	0
serial	889	1.6%	638400	10708	0	0
dft	890	1.2%	79636	968	0	0
complex	891	1.0%	4847569	47963	237	0
excursions	1794	1.0%	12461	123	1145	0
longest_runbloc	1797	1.0%	640512	6416	0	0
matrix	1799	0.8%	2103	17	0	0
template	1806	1.2%	1605	19	0	0
cusums	1808	1.0%	2604830	25695	7	0

5.2. ábra. AES Cypher statisztikája PRNG-ként használva beépített entrópiaforrással nem blokkoló üzemmódban seedelve

Name	PID	T % min	T Pass	T Min	T Maj	T Crit
approxentropy	1793	0.9%	11077	104	0	0
complex	1795	1.0%	1770380	17716	199	0
dft	1797	1.1%	44151	507	0	0
excursions	1799	0.9%	6804	60	731	0
linearcomplex	1801	1.0%	289	3	0	0
longest_runbloc	1803	1.0%	355029	3530	1	0
matrix	1805	1.0%	1163	12	0	0
mauer	1807	1.2%	1161	14	0	0
monobit	1809	1.0%	2002116	19295	206	0
monobitblock	1811	0.9%	1390530	13239	39	0
runs	1813	1.0%	177069	1775	0	0
serial	1815	1.7%	353754	5987	1	0
template	1817	1.1%	726	8	0	0
cusums	1819	1.0%	1384523	13866	46	0

5.3. ábra. AES Cypher statisztikája PRNG-ként használva számlálóval seedelve

AES Cypher PRNG-ként használva beépített entrópiaforrással blokkoló üzemmódban seedelve

AES cyphert ebben az esetben is a beépített entrópiaforrással seedeljük, azzal a különbséggel, hogy az üzemmód választásnál blokkolást állítunk be. Az így kapott eredményeket az 5.4 ábra mutatja. A gyűjtött adatok, az előzőekhez hasonlóak. Kirívó a linearcomplex teszt nagyobb bukási aránya, azonban a kevesebb minta miatt ennek biztonsága kérdéses lehet. Másik eltérés a monobit csoportos bukások megnőtt aránya. Egy lehetséges magyarázat lehet erre, hogy mivel itt blokkoló üzemmódban kértünk adatokat a beépített entrópiaforrástól, ezért annak működésére vártunk. Amennyiben a csoportos bukások ennek általános üzemű működéséből származnak, blokkoló üzemmód bekapcsolásával ezeket a hibákat úgymond bevárjuk, a kapott bitsornak nagyobb részét teszi ki a forrás általános működésének eredménye. Tekintve, hogy ennek belső megvalósítását körültekintő analízisnek nem vetettük alá, valamint későbbi felhasználásainkhoz ennek ismerete nem feltétel, a jelenség további vizsgálatától eltekintünk.

Hosszútávú összesített statisztikák

Az egyes tesztek által gyűjtött logfájlok alapján KS tesztek futtatásával aggregált eredményeket készíthetünk. Egy-egy ilyen 10000 teszt eredményét összesíti, így egy újabb tömörebb naplófájlt adva. Ez az előzőnél már mélyebb vizsgálatot jelent, itt elvárható már a bukás bizonyos teszteken. A kapott összesítő eredmények összefoglalása az 5.1 táblázatban található. Mivel minden 10000 teszteredményből lesz egy KS teszt eredménye, ezért egy teszthez még így is általában egynél több adat tartozik. Ezen felül ezen összesítő állítások is statisztikai tesztek, ezt az eredmények értékelésénél is figyelembe kell venni. Elvárás így ezek felé is, hogy egyenletes eloszlást mutassanak. A táblázatban négyféle minősítést osztottunk ki egyes tesztek eredményeire, aszerint, hogy a KS tesztek kimenete az elvárt tulajdonságokat mennyire követik.

1. OK - A tesztstatisztika az elvárásoknak megfelelő.
2. Nem erős - A tesztstatisztikán az elvárásoktól eltérő tendencia látszik (Például minden p-érték 0.4 alatti).
3. Gyenge - A tesztstatisztika erősen eltér az elvárásoktól, azonban nem konstans 0.
4. Bukott - A tesztstatisztika konstans 0.

A hosszútávú teszteken a monobit teszt minden esetben megbukott, valamint több más teszt is van, melyek eredményei általánosan bukás fele tendálnak. Ez valamelyest elvárható volt, tekintve, hogy alacsony minőségű entrópiaforráson végeztünk elemzést nagy mennyiségű adat gyűjtésével. A vizsgált esetek közül a legjobban a blokkoló seedelésű AES cypher teljesített.

Referencia választás

Későbbi eseteink vizsgálatához, a generátor rövid távon történő legjobb működése a cél, a normális működés során detektált hibák minimalizálása a fontos. Ez alapján a nem blokkoló üzemű AES cypher által megvalósított megoldást választottuk, mivel itt a legkisebb

Name	PID	T % min	T Pass	T Min	T Maj	T Crit
monobit	2068	0.9%	2045434	19609	397	0
approxentropy	2094	1.0%	5228	51	0	0
complex	2099	1.0%	1362579	13701	135	0
dft	2101	1.1%	20845	241	0	0
excursions	2103	0.9%	3194	30	364	0
linearcomplex	2105	2.5%	198	5	0	0
longest_runbloc	2107	1.0%	167540	1674	0	0
matrix	2109	0.9%	549	5	0	0
mauer	2111	0.9%	549	5	0	0
monobitblock	2114	0.9%	680569	6407	4	0
runs	2116	1.0%	83588	846	0	0
serial	2118	1.7%	166939	2834	0	0
template	2120	1.2%	482	6	0	0
cusums	2122	1.0%	680385	6693	6	0

5.4. ábra. AES Cypher statisztikája PRNG-ként használva beépített entrópiaforrással blokkoló üzemmódban seedelve

KS teszt eredmények				
Teszt neve	AutoSeeded RandomPool	Nem blokkoló AES	Számláló AES	Blokkoló AES
monobit	Bukott	Bukott	Bukott	Bukott
monobitblock	OK	OK	OK	OK
runstest	OK	OK	OK	OK
longest_runblock	Nem erős	Nem erős	Nem erős	Nem erős
matrix	OK	OK	OK	OK
dft	Gyenge	Nem erős	Gyenge	OK
template.nonoverlap	OK	OK	OK	OK
template.overlap	OK	OK	OK	OK
mauer	OK	OK	OK	OK
linearcomplex	OK	OK	OK	OK
serial	OK	OK	OK	OK
approxentropy	OK	OK	OK	OK
cusums	Gyenge	Gyenge	Gyenge	Gyenge
excursions.standard	Gyenge	Gyenge	Gyenge	Gyenge
excursions.variant	Gyenge	Gyenge	Gyenge	Gyenge
complex.runblock	OK	OK	OK	OK
complex.cusums	Gyenge	Gyenge	Gyenge	Gyenge

5.1. táblázat. KS teszt eredmények

a csoportos bukások aránya, ezáltal rövidebb vizsgálatokhoz ez adja a legjobb referenciát a vizsgált megoldások közül.

5.3. Módosított eloszlás esete

Ebben szekcióban a 4.5.1-ben bemutatott szimulált esetet vizsgáljuk a különböző tesztesetekkel. Ennek megfelelően az 5.2-ben választott generátor kimenetét egy egyszerű előfeldolgozásnak küldés előtt alávetve szimuláljuk a 45-55%-os megváltozott arányt. Mivel a sebesség ebben az esetben közömbös, egyszerűen a generátor által szolgáltatott eredeti kimenet bájtonkénti vizsgálatával elő tudjuk ezt állítani. A vizsgált bájtot egész számként kezelve, ha ez a szám az alsó 45%-nak megfeleltetett régióba esik (<115) 0-ás bitet ha nem, akkor pedig 1-es bitet küldünk.

5.3.1. Megváltozott eloszlás detektálása

Először az 5.2-ben ismertetett mérési elrendezés ezen a megváltoztatott eloszláson történő futási eredményeit vizsgáltuk. Célunk a hibafajtára érzékeny tesztek meghatározása.

Name	PID	I Pass	I Min	I Maj	I % min	T % min	T Pass	T Min	T Maj	T Crit
approxentropy	2057	0	1	0	100.0%	100.0%	0	3	0	0
complex	2059	7	175	31	96.2%	96.8%	30	908	121	0
dft	2061	3	0	0	0.0%	42.9%	8	6	0	0
longest_runbloc	2067	0	23	0	100.0%	100.0%	0	118	0	0
monobit	2073	113	245	36	68.4%	66.1%	633	1236	142	0
monobitblock	2075	36	55	0	60.4%	62.1%	179	293	5	0
runs	2077	12	0	0	0.0%	0.0%	59	0	0	0
serial	2079	0	22	0	100.0%	100.0%	0	117	0	0
cusums	2083	0	90	0	100.0%	100.0%	0	472	0	0
monobit_general	2086	352	5	0	1.4%	1.3%	1851	25	0	0
monobit_long	2088	0	12	0	100.0%	100.0%	0	59	0	0

Up&R	NoSig	Unstable	MinInc	MinPc	MajPc	Block
3	0	0	0	9	2	11

5.5. ábra. 45-55 arányban módosított eloszlás szerinti generátor eredményei

45-55% szerint módosított eloszlás

Működési hiba észlelése esetén a rendszernek a kimenetet automatikusan blokkolnia kell. Mivel a tesztek egyenletes eloszlást feltételeznek egy így módosított generátor a rendszer számára hibás működésűnek kell, hogy tűnjön. A 5.5 ábrán láthatóak ennek az esetnek az első néhány perc futásából származó eredményei. Így látható az is, melyek azok a tesztek amelyek bemenetük mérete miatt gyors detekcióra képesek lehetnek. Látható, hogy a monitorozó rendszer a kimenetet automatikusan blokkolja (Block nem 0), tehát sikeres a hibadetekció. Ekkora eloszlásbeli eltérésre a legtöbb teszt bukással reagál, emiatt vizsgáljuk egy kevésbé drasztikus változtatás hatását is.

49-51% szerint módosított eloszlás

Az előző mérést hajtjuk végre újra, azzal a módosítással, hogy most 49-51%-osra módosítjuk a generátor által küldött bitek eloszlását. Az így kapott eredményeket az 5.6 ábra mutatja. A hibadetekció itt is sikeres, azonban erre a változásra már kevesebb teszt érzékeny. A legérzékenyebb (a gyors tesztek közül) a monobit_long teszt. Érdemes megfigyelni, hogy a sima monobit teszt ezt a hibát csak az átmeneti statisztikát használva (utófeldolgozással a p-értékekből a hiba látható) nem veszi észre, míg hosszabb bitsorokra futó változata igen. Ez annak tudható be, hogy a rövidebb tesztben a kevesebb tesztelt bit miatt az átlagos elvárttól való eltérés még nem elég nagy megbízható bukás okozá-

Name	PID	I Pass	I Min	I Maj	I % min	T % min	T Pass	T Min	T Maj	T Crit
complex	1811	78	2	0	2.5%	3.1%	217	7	0	0
dft	1813	1	0	0	0.0%	0.0%	3	0	0	0
longest_runbloc	1819	11	1	0	8.3%	6.7%	28	2	0	0
monobitblock	1827	40	0	0	0.0%	0.9%	111	1	0	0
runs	1829	6	0	0	0.0%	0.0%	15	0	0	0
serial	1831	10	0	0	0.0%	3.6%	27	1	0	0
cusums	1835	38	2	0	5.0%	2.7%	109	3	0	0
monobit	1877	157	3	0	1.9%	1.6%	441	7	0	0
monobit_long	1885	3	3	0	50.0%	60.0%	6	9	0	0
monobit_general	1887	158	2	0	1.2%	1.3%	442	6	0	0

Up&R	NoSig	Unstable	MinInc	MinPc	MajPc	Block
10	0	0	0	1	0	1

5.6. ábra. 45-55 arányban módosított eloszlás szerinti generátor eredményei

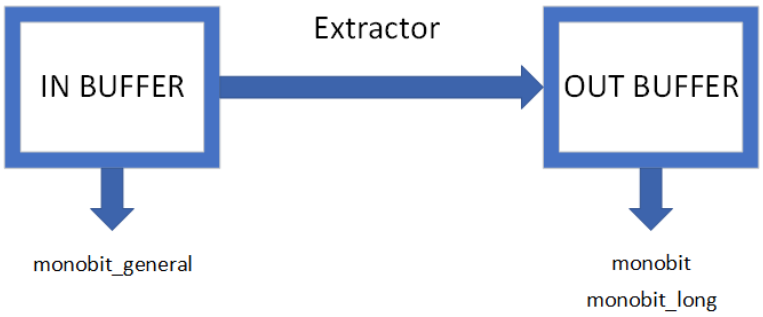
sához. Ez bizonyítja, hogy a tesztenként vizsgált bithossz is hatással tud lenni az egyes hibafajtákra való érzékenységre. A monobit esetében ez azt mutatja, hogy míg a rövidebb tesztekkel detektálni lehet bizonyos időszakos romlásokat 5.2.2, ezek jelen módosított eloszlást használó példánkban nem elégségesek. Ezzel szemben a hosszabb változat a jelenlegi esetet tudja detektálni, míg a korábbit nem. Megjegyzendő, hogy a kiterjesztett monobit_statconcat segédprogrammal a rövidebb teszt logfájlaiból a hosszabb teszttel egyenértékű tesztek készíthetőek, azonban valós idejű ellenőrzésre ez nem optimális.

5.3.2. Teszt módosított eloszláshoz

A 5.5 és 5.6 ábrákon bemutatott eredményeken láthatóak a monobit_general nevű teszt futtatásának is az eredményei. Ez a teszt mindkét esetben az éppen aktuális (45-55 vagy 49-51) eloszláshoz 4.5.1 során már leírt általános monobit teszt megvalósítása. Várható kimenete ugyanaz mint a monobit kimenete egyenletes eloszlás esetén. A kapott eredmények ennek megfelelnek.

5.3.3. Extraktorok tesztje

A 4.1 fejezetben bemutatott algoritmusok segítségével az így kapott hibás bemenet feldolgozható. Mivel a hibánk egyenetlenséggel jár, mindkét ismertett algoritmust vizsgálhatjuk vele. (Mindkettőnek javítania kell tudni.) A következő esettel ezek működését vizsgáljuk.



5.7. ábra. Extractorok vizsgálatához használt elrendezés

Name	PID	I Pass	I Min	I Maj	I % min	T % min	T Pass	T Min	T Maj	T Crit
monobit	2077	43	0	0	0.0%	1.0%	1727	17	0	0
monobit_long	2081	1	0	0	0.0%	0.0%	54	0	0	0
monobit_general	2085	410	5	0	1.2%	1.2%	17218	207	0	0

5.8. ábra. Von Neumann alapú extractor működtetésének eredménye

A mérési elrendezésen annyit változtatunk, hogy a bejövő adatfolyamon közvetlen csak a monobit_general tesztet futtatjuk, ezután egy extractor segítségével ezt feldolgozva a feldolgozott adatokat egy új bufferbe írjuk, amin a maradék teszt fut, így vizsgálva, hogy az extraktor meg tudta-e szüntetni a kiegyenlítetlenséget. Mivel a kiegyenlítetlenségre legjobban a monobit_long teszt érzékeny, ezért a monobit_general teszten kívül ezt, valamint az eredeti monobit tesztet futtatjuk, hogy ellenőrizni tudjuk a hiba megszűntét, valamint alapot kapjunk a lefuttatott tesztek számának összehasonlításához. (A monobit_general és monobit az eredeti elrendezésben közel azonos alkalommal fut le.)

Az 5.8 és 5.9 ábrákon láthatóak az egyes extractorokhoz tartozó futások eredményei. Látható, hogy a kiegyenlítetlenségből származó hiba mindkét esetben megszűnt. A kime-

Name	PID	I Pass	I Min	I Maj	I % min	T % min	T Pass	T Min	T Maj	T Crit
monobit	1802	7	0	0	0.0%	1.1%	4700	54	0	0
monobit_long	1806	1	0	0	0.0%	2.0%	146	3	0	0
monobit_general	1810	58	1	0	1.7%	1.1%	32406	358	3	0

Up&R	NoSig	Unstable	MinInc	MinPc	MajPc	Block
4	0	0	0	0	0	0

5.9. ábra. SHA hash alapú extractor működtetésének eredménye

neten elvégzett tesztek száma, azonban csökkent. Ez annak az eredménye, hogy ezek az extractorok működésük során bár entrópiát nyernek ki, ez sok esetben a kimeneti sebesség kárára történik. Erre a csökkenésre az elvégzett tesztek arányából lehet következtetni. A Von Neumann extractor esetében ez az arány 1:10, míg az SHA megoldás ennél egy kicsit jobban teljesít, itt az arány 1:7 körüli.

5.3.4. Megjegyzés a mintavételezésről

Felmerülhet a kérdés, hogy milyen hatással van a teszteredményekre a körbufferből történő beolvasás. A 3.4.2 fejezetnek megfelelően a tesztek egymásra gyakorolt hatásával nem kell foglalkozni, mivel minden folyamathoz saját olvasópointer tartozik, nem lép fel kiéheztetés. A külső eszköztől kapott bufferbe beolvasott és a tesztek által bufferből kiolvasott adatsorok így csak abban az esetben különbözhetnek, ha a beolvasási sebesség a feldolgozás sebességénél gyorsabb, két beolvasás között a körbuffer új adattal írja felül az olvasópointer által éppen jelölt területet. A mintavételezés hatása kiküszöbölhető a buffer kellően nagy méretűre választásával. Ez akár MB-os nagyságúra is állítható, ami elégséges ahhoz, hogy az összes teszt esetén eredményenkét elegendő egy a tesztenként megadott értékeknek megfelelő, kellően hosszú összefüggő blokkot használó olvasási művelet. Következményként minden folyamat úgy kapja meg a számára érdekes adatrészt, ahogy az a generátorból érkezett.

5.4. A tesztek futási sebessége

A végleges rendszerben az egyes tesztek futási sebessége kritikus szempont. Nem érdemes olyan teszteket választani, amelyek túl kevés bitet mintavételeznek, vagy túlságosan kihasználják a processzort. Emiatt fontos az egyes tesztek futási sebességének mérése, amelyet a már ismertetett `qrng-time` szkripttel tehetünk meg.

Fontos, hogy a rendszerkomponenseket a lehető legjobb sebesség érdekében a gcc `-O3` és `-march=native` beállításaival fordítjuk. Az `-O3` a lehető legmagasabb optimalizációs szint, ami minden esetben biztonságos (nem okoz eltéréseket a szabványos C/C++ viselkedéstől). Az `-march=native` flag bekapcsolja az összes olyan speciális funkció használatát (pl. vektor utasítások, titkosító algoritmusok stb.), amelyek az adott CPU-n elérhetőek [26]. Ez a binárist processzorcsalád specifikussá teszi, de gyorsíthat rajta. Emiatt a mért teljesítményadatok tájékoztató jellegűek, csak az adott processzorcsaládra igazak teljes egészében, de nagyságrendi eltérésekre nem érdemes számítani az egyes tesztek futásidejének arányaiban más rendszereken sem.

Az időmérés során az előzőekben bemutatott teszteket futtattuk, a bemenetet a `/dev/urandom` eszköz szolgáltatta (ez pseudo véletlen számokat generál kernel szinten). A virtuális gép egy Intel Xeon X5650 [27] processzor 8 szálát kapta meg a 12-ből.

Az eredmények alapján a következő megfigyeléseket tehetjük:

- A *linearcomplex* teszt nagyon lassú, nem biztos, hogy érdemes futtatni éles rendszeren.
- Az egyes tesztek néha belassulnak (l. min. kbit/s), ezeket valószínűleg az okozza, hogy a rendszer éppen túlterhelt és elveszi a processzort futás közben.

Test	átl. t	min. t	max. t	átl. kbit/s	min. kbit/s	max. kbit/s
approxentropy	879ns	725ns	1503ns	9099	5321	11029
complex	525ns	156ns	20084ns	15234	398	51200
cusums	141ns	62ns	2773ns	56668	2883	127007
dft	450ns	328ns	1037ns	17759	7711	24322
excursions	150ns	72ns	213ns	53041	37451	110301
linearcomplex	18241ns	17365ns	19897ns	438	402	460
matrix	353ns	306ns	525ns	22627	15236	26124
mauer	532ns	490ns	650ns	15017	12307	16294
monobit	107ns	26ns	2610ns	74147	3064	303407
monobitblock	135ns	59ns	2242ns	58982	3567	135404
monobit_general	100ns	27ns	2610ns	79664	3064	292571
monobit_quick	98ns	26ns	2562ns	81558	3121	303407
monobit_quick2048	98ns	26ns	2562ns	81592	3121	303407
runs	85ns	46ns	732ns	93706	10919	173375
serial	1736ns	1312ns	3594ns	4606	2225	6095

5.2. táblázat. A qrng-time kimenetéből készített táblázat. Tartalmazza az átlagos mért, bitsebességeket kbit/s-ban, illetve az egy bájtra leosztott futásidőt nanoszekundumban. A rendszer a futtatás során egy Intel Xeon X5650 [27] processzor 12 logikai szálából 8-at használt.

- A *cusums, runs, monobit és runs* tesztek kifejezetten gyorsak (50Mbit/s felett), ezeket érdemes lehet alkalmazni akár a feldolgozott kimenet tesztelésére is. Ezen tesztek több erőforrás esetén még gyorsabbak lehetnek, mivel a max. kbit/s értékeik jelentősen nagyobbak az átlagnál.

- Igazából két érték érdekes, az átlag sebesség amely tükrözi az adott processzoron ténylegesen elérhető rendszer sebességet, illetve a maximum, amely megadja, hogy az adott teszt nagyjából milyen gyorsan futna önmagában.

- A *monobit_quick és monobit_quick2048* tesztek az eredeti *monobit*-el közel azonosan teljesítenek. A változatok közti különbség annyi, hogy „quick” megjelelölés tesztekben p-érték számoláshoz a küszöbhöz tartozó megfigyelt statisztika értéke a kódba be van égetve, így az erfc() függvény hívását elkerüljük. Ebből arra lehet következtetni, hogy a legszámításigényesebb rész a tesztstatisztika számolása (1-esek megszámolása).

Összefoglalva a fenti értékek alapján megállapíthatjuk az egyes tesztek egymáshoz képeszti sebességét, hiszen a minden egyes feldolgozás után meghívott `sched_yield()` hívással megadjuk a lehetőséget minden tesztnek a processzor használatára [18], nem lép fel kiéheztetés. Az elért sebesség megfelelőnek tűnik, ahhoz képest, hogy a futtatáshoz egy viszonylag régi (8 éves) processzort használtunk, a *monobit* teszt maximum bitsebessége elérte a a 300Mbit/s-ot.

6. fejezet

Összefoglalás, jövőbeli tervek

Dolgozatunkban bemutattuk az általunk véletlenszám generátorok kimenetének feldolgozására, felügyeletére, és validációjára készített megoldást. Ismertettük a felénk támaszott elvárásokat, majd bemutattuk az ezek teljesítésére képes általunk fejlesztett rendszert.

6.1. Dolgozat összefoglalása

Bevezetésként bemutattuk véletlenszámok előállításának főbb módjait. Ezek közül részletesen ismertettük ennek kvantummechanikai jelenségekre épülő változatát. Ismertettünk három különböző fizikai megvalósítást, melyek várhatóan annak a nagyobb kvantumtechnológiai projektnek a részeként kerülnek elkészítésre, melynek az általunk tervezett rendszer is része.

Bemutattuk az ilyen generátorok felé támasztott elvárásokat, ezáltal elvárást támasztva felügyelő rendszerük felé is, mivel igazolnia kell tudni ezen elvárások teljesülését. Ezután statisztikai tesztek ilyen célokra történő általános használhatóságával foglalkoztunk, ismertetve a jelenleg is elérhető megoldásokat. Beláttuk, hogy ezek az általunk támasztott kritériumok teljesítésére alkalmatlanok, így indokolva a továbbiakban saját, egyedi architektúra fejlesztését.

Részletesen bemutattuk a tervezett rendszert, a tervezés során felmerülő kérdéseket, az általunk választott megoldásokat és választásaink okait. Ezt követte az ez alapján megvalósított környezet leírása, komponensenkénti működésének részletes ismertetése.

Ennek ismeretében következő lépésként a működés során használt eszköztár bemutatásával folytattuk. Az egyedi rendszer ezen tár, magához való igazítását követelte. Az extractorok esetében ez viszonylag kevés módosítást jelentett. Statisztikai tesztek adaptálásához ellenben szükség volt saját bitkezelés készítésére, valamint az egyes tesztek eszerint történő újraírására. Az összes NIST STS-ben található statisztikai teszt ilyen módon átvételre került. Vizsgáltuk továbbá a már meglévő tesztek általánosabb és egyéb specifikus igényekre használható változatait, ennek eredményeként ilyenekre példaimplementációt is készítettünk. Lezárásként bemutattuk a rendszerben található hosszútávon gyűjtött nagy adatmennyiség elemzésére alkalmas megoldásokat.

Az elkészült rendszert szimulált teszteseteknek vetettük alá, így ellenőrizve megfelelő viselkedését. Egyszerűen elérhető szoftveres generátorok kimenetét elemeztük vele. Az így kapott adatok felhasználásával hibás bemeneti folyamot szimuláltunk, bemenet

javításának módjait, az elérhető eszközök erre való érzékenységét vizsgáltuk. Végül ezek teljesítményét is összehasonlítottuk.

Megállapítottuk, hogy a készített rendszer megfelelően működik, pontos generátorhoz készített egyedi eszköztár futtatásához platformként tud szolgálni, ilyen gyűjtemény készítéséhez eszközök vizsgálatára alkalmas.

6.2. Jövőbeli tervek

Logikus következő lépés a fizikai hardver elkészültével a rendszer ehhez történő pontos illesztése. Végső cél egy, az egyetemi hallgatók számára véletlenszámokat biztosító kvantumos architektúrán alapú megoldás készítése, melyhez a nyers adatfolyam felügyeletét, feldolgozását (így külvilág fele a kimenetet is) és validációját is ez a rendszer végzi.

A környezet platformként történő használhatóságának további fejlesztése is különböző lehetőségeket rejt magában. Mivel felügyeletre képes kizárólag beérkező bitfolyam alapján, a ténylegesen megvalósított generátor milyensége közömbös. Emiatt a rendszer egyszerűen használható és telepíthető csomaggá történő alakításával, egy általános, a jelenleg elérhető megoldásoknál jobban személyre szabható platform készíthető. Eszközök könnyű hozzáadása miatt új kísérleti tesztek és extractorok teszteléséhez is ideális.

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.

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] Ian Goldberg – David Wagner: Randomness and the netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21. évf. (1996) 1. sz., 66–71. p.

[2] CVE-2008-0166 OpenSSL 0.9.8c-1 up to versions before 0.9.8g-9 on Debian-based operating systems uses a random number generator that generates predictable numbers, which makes it easier for remote attackers to conduct brute force guessing attacks against cryptographic keys. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-0166`, 2008. Jan.

[3] Michael A. Nielsen – Isaac L. Chuang: *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. kiad. New York, NY, USA, 2011, Cambridge University Press. ISBN 1107002176, 9781107002173.

[4] S. Imre – B. Ferenc: *Quantum Computing and Communications An Engineering Approach*. Hoboken, NJ, USA, 2005, John Wiley & Sons, Inc.

[5] Schranz Ágoston: Szakmai beszámoló a HunQuTech-projekt keretében készülő QRNG architektúráról, 2018 (projektbeszámoló).

[6] Thomas Jennewein – Ulrich Achleitner – Gregor Weihs – Harald Weinfurter – Anton Zeilinger: A fast and compact quantum random number generator. *Review of Scientific Instruments*, 71. évf. (2000) 4. sz., 1675–1680. p.

[7] André Stefanov – Nicolas Gisin – Olivier Guinnard – Laurent Guinnard – Hugo Zbinden: Optical quantum random number generator. *Journal of Modern Optics*, 47. évf. (2000) 4. sz., 595–598. p.

[8] Harald Fürst – Henning Weier – Sebastian Nauerth – Davide G Marangon – Christian Kurtsiefer – Harald Weinfurter: High speed optical quantum random number generation. *Optics express*, 18. évf. (2010) 12. sz., 13029–13037. p.

[9] Miguel Herrero-Collantes – Juan Carlos Garcia-Escartin: Quantum random number generators. *Reviews of Modern Physics*, 89. évf. (2017) 1. sz., 015004. p.

[10] Mario Stipčević – B Medved Rogina: Quantum random number generator based on photonic emission in semiconductors. *Review of scientific instruments*, 78. évf. (2007) 4. sz., 045104. p.

[11] Sp 800-90a rev. 1 recommendation for random number generation using deterministic random bit generators. https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final.

[12] Sp 800-90b recommendation for the entropy sources used for random bit generation. https://csrc.nist.gov/publications/detail/sp/800-90b/final.

[13] Sp 800-90c (draft) recommendation for random bit generator (rbg) constructions. https://csrc.nist.gov/publications/detail/sp/800-90c/draft.

[14] Nist sp 800-22: Documentation and software. https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software.

[15] dieharder by robert g. brown, duke university physics department, durham, nc 27708-0305 copyright robert g. brown, 2018. https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[16] G. Marsaglia: "the marsaglia random number cdrom including the diehard battery of tests of randomness". florida state university. 1995. archived from the original on 2016-01-25. https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/.

[17] emlog github. https://github.com/nicupavel/emlog.

[18] The open group base specifications issue 7, 2018 edition, ieee std 1003.1™-2017 (revision of ieee std 1003.1-2008). https://pubs.opengroup.org/onlinepubs/9699919799/.

[19] Freedesktop.org systemd wiki. https://www.freedesktop.org/wiki/Software/systemd/.

[20] Elaine Barker – John Kelsey – John Bryson Secretary: Nist draft special publication 800-90b recommendation for the entropy sources used for random bit generation, 2012.

[21] Intel sha extensions. https://software.intel.com/en-us/articles/intel-sha-extensions.

[22] John von Neumann: Various techniques used in connection with random digits. In A. S. Householder – G. E. Forsythe – H. H. Germond (szerk.): *Monte Carlo Method*. National Bureau of Standards Applied Mathematics Series sorozat, 12. köt. 13 fejezet. Washington, DC, 1951, US Government Printing Office, 36–38. p.

[23] Andrey Kolmogorov: Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 4. évf. (1933), 83–91. p.

[24] Nickolay Smirnov: Table for estimating the goodness of fit of empirical distributions. *The annals of mathematical statistics*, 19. évf. (1948) 2. sz., 279–281. p.

[25] Crypto++ library 8.2, a free c++ class library of cryptographic schemes. https://www.cryptopp.com/.

[26] Gcc 7 manual. https://gcc.gnu.org/onlinedocs/gcc-7.4.0/gcc/.

[27] Intel xeon x5650 specifications. https://ark.intel.com/content/www/us/en/ark/products/47922/intel-xeon-processor-x5650-12m-cache-2-66-ghz-6-40-gt-s-in html.