Eötvös Loránd Tudományegyetem

Informatikai Kar

Komputeralgebra Tanszék

Pszeudovéletlen sorozatok mértékei és konstrukciói

Szerző:

Kovács Levente

programtervező informatikus BSc

Témavezető:

Dr. Tóth Viktória

egyetemi adjunktus

Komputeralgebra Tanszék

Budapest, 2019

Tartalomjegyzék

1 Bevezető

1.1 Történeti betekintés

Véletlen sorozatok generálására sok modern területnek igénye van (pl.: statisztika, szimulációk, kriptográfia). Számítógéppel azonban nem lehetséges valódi véletlen számokat generálni, ezért pszeudovéletlen eljárásokra hagyatkozunk. Cél az, hogy olyan eljárásokat használjunk melyek statisztikailag véletlennek tűnnek, attól függetlenül, hogy egy determinisztikus rendszer generálja az eredményt.

Az ilyen sorozatokra való igény először Gilbert Vernam munkájának következménye. 1917-ben Vernam feltalálta a one-time pad titkosítási eljárást. Az üzeneteket telegráfon bitenként továbbították, viszont az eredeti adatot valamilyen kulcs segítségével megváltoztatták (amit aztán a fogadó fél a kulcs ismeretében vissza tud fejteni). Ha le is hallgatták a küldött adatot, akkor a kulcs nélkül nem lehetett tudni, hogy mi volt az eredeti üzenet.

1949-ben Claude Shannon, matematikus, bebizonyította a fenti eljárásról, hogyha egy jó sorozatot használunk kulcsnak, akkor az eredeti és a titkosított bitsorozat között nem lehet semmiféle összefüggést találni. Ennek következménye, hogy a Vernam-féle one-time pad eljárást nem lehet feltörni, ha megfelelő, egyszer használatos kulcsokat használunk.

Az is igaz azonban, hogyha nem megfelelő kulcsokat használunk (pl. többször használjuk a kulcsokat), akkor a titkosítás könnyen törhető. A fő nehézség tehát a one-time pad használatánál, hogy nagy méretű véletlen bitsorozatokat kell előállítani kulcsoknak.

Az ilyen sorozatok előállításához régen fizikai eszközöket használtak, pl. diódákat. A problémája az ilyen eszközöknek, hogy külső tényezők is befolyásolják a véletlenséget, és meghibásodás is történhet. Ezért nagyon fontos, a véletlenség vizsgálása különböző statisztikai tesztekkel, ami azonban időigényes. Emiatt jobb, hogyha "csak" pszeudovéletlen sorozatokat állítunk elő. Így elkerülhető a statisztikai tesztelés, mivel

bizonyítottan jó véletlen tulajdonságokkal rendelkező sorozatot kapunk.

A pszeudovéletlen tulajdonságok mérésével sok évtizede foglalkoznak matematikusok. Elsőként Émile Borel definiálta végtelen bináris sorozatokra a pszeudovéletlenség normalitás mértékét. Később Solomon Wolf Golomb és Donald Knuth próbálkoztak a pszeudovéletlenség matematikai fogalmának definiálására. Ennek a definíciónak alapvető gyengeségei voltak, ezért ezt ma már nem nagyon használják. Ezután Andrej Nyikolajevics Kolmogorov és Gregory John Chaitin bonyolultságelméleti irányból közelítették meg a problémát, de a gyakorlatban ezek a fogalmak nem kaptak nagy jelentőséget, mivel alkalmazásuk nem jól kivitelezhető.

Ezen próbálkozások inspiráltak egy új, konstruktív megközelítést a pszeudovéletlenség vizsgálatára. Christian Manduit és Sárközy András megközelítése az volt, hogy léteznek olyan sorozatok, melyekről bizonyítható, hogy statisztikailag erős véletlen tulajdonságokkal rendelkeznek (más szóval: erős pszeudovéletlen tulajdonságúak), így az utólagos tesztelés elkerülhető. Ez a fajta vizsgálat az évek során egyre szélesebb körben alkalmazott.

1.2 A program célja

A programom célja, hogy a fent említett konstrukció keretén belül vizsgáljak különböző pszeudovéletlen sorozatokat. A vizsgált konstrukciókat két csoportból válogattam. Egyik kategóriába matematikai hátterű sorozatok tartoznak, melyeknek működése főként számelméleti ismereteken alapulnak. A másik csoport az iparban széles körben használt pszeudovéletlen generátorok, amik nem feltétlenül matematikai hátterűek.

A programban lehetőség nyílik a pszeudovéletlen konstrukciók kipróbálására, és a generált sorozatok vizsgálatára a statisztikai mértékek szerint. A sorozatokat ki is próbálhatjuk a Vernam-féle one-time pad titkosítási eljárással.

2 Felhasználói dokumentáció

A program pszeudovéletlen sorozatkonstrukciók kipróbálására és a sorozatok vizsgálatára készült a statisztikai mértékekkel.

2.1 Rendszerkövetelmények

2.2 Futattás

2.3 Menürendszer

A menürendszer pontjainak 3 fő csoportja van, ezek:

- Alapvető funkciók: Kilépés, vissza a főmenübe
- Konstrukciók: A konstrukciók kipróbálása
- Tesztelés: Mértékek és titkosítás

2.3.1 Főmenü

Amikor a program elindul a felhasználót a főmenü fogadja. Itt elérhető az összes megvalósított konstrukció almenüpontja és a mértékek, illetve titkosításé is.

Kilépni a program jobb felső sarkában található "X" gombbal, vagy a főmenüben található "Kilépés" gombbal lehet.

Konstrukciók

Ez a pont csak alapvető segítséget nyújt a különböző konstrukciók kipróbálásához. Részletes leírása az algoritmusoknak a fejlesztői dokumentációban találhatóak.

2.3.2 Legendre szimbólumos konstrukció

Ez a konstrukció a Legendere szimbólumon alapul, melynek véletlen tulajdonságait régóta vizsgálják. Mauduit és Sárközy ezen a sorozaton vizsgálta először 1997-ben a véges bináris sorozatokra készített jól-eloszlás és korreláció-mértékeket [C. Mauduit, A. Sárközy, On finite pseudorandom binary sequences I: The measures of pseudorandomness, the Legendre symbol].

Szükséges paraméterek:

1. **A sorozat hossza**

   A sorozat hosszát bájtokban kell megadni, tehát az 1 bájt hosszú sorozat 8 bitet foglal magába. Bármennyi lehet.

2. **Prímszám**

   Egy olyan prímszámra van szükségünk a konstrukcióhoz, mely legalább 16-szor nagyobb mint a sorozat hossza (bitben megadott méret esetén 2-szer nagyobb). Az is kell, hogy a 2 legyen primitív gyök modulo prím. A "Generálás" gombbal a program megkeresi a legkisebb olyan prímet mely a fenti feltételeknek megfelel. A "Következő" gombbal ugorhatunk a következő legkisebb jó prímre.

3. **A konstrukcióhoz használt polinom fokszáma**

   A fokszám nem lehet túl nagy függően a prímszámtól, de túl kicsi sem. Az alsó határ jelen megvalósításban 2-nek, a felső $5 \cdot \sqrt[10]{p}$-nek lett megszabva, ahol p a használt prímszám. Polinom fokszámot generálhatunk a fenti intervallumban a "Generálás" gombbal.

4. **A polinom**

   A konstrukció jó működéséhez szükséges, hogy olyan polinomot válasszunk, amelynek csak egyszeres gyökei vannak. Emiatt a polinomot a gyökeinek felsorolásával reprezentáljuk, ezeket a szóközzel elválasztva kell felsorolni a szövegmezőben. A fokszámmal megegyező számú gyököt kell megadni. Lehetőség van polinom generálására is, ekkor a program legenerál egy az előző pontban megadott fokszámú polinomot.

Ha fenti paramétereket megadtuk, akkor a "Sorozat generálása" gombra kattintva megkapjuk az eredményét a Legendre konstrukciónak a kapott bemenetre.

2.3.3 RC4 konstrukció

Az RC4 konstrukciót Ron Rivest fejlesztette ki az 1987-ben viszont eredetileg üzleti titok volt. 1994-ben azonban kiszivárgott az algoritmus leírása. A működése egyszerű és gyors, ezért használata széles körben elterjedt, viszont vannak rossz tulajdonságai.

Szükséges paraméterek:

1. **A sorozat hossza**
   A sorozat hosszát bájtokban kell megadni. Bármennyi lehet.
2. **A kulcs**
   A kulcs változó méretű lehet, ez alapján fog történni a generálás. Tipikusan 40-2048 bit hosszú a kulcs. Ez a megvalósítás egy szöveget vár kulcsnak, így átfogalmazva a feltételt legalább 5, maximum 256 karakter hosszú lehet a kulcs.

A fenti paraméterek megadása után megtekinthetjük az RC4 konstrukció által generált sorozatot a "Sorozat generálása" gombbal.

2.3.4 Additív karakteres konstrukció

Ezt a konstrukciót Mauduit, Rivat és Sárközy vezették be [C.Mauduit, J.Rivat, A.Sárközy, Construction of pseudorandom binary sequences using additive characters].

Szükséges paraméterek:

1. **A sorozat hossza**
   A sorozat hosszát bájtokban kell megadni. Bármennyi lehet.

2. **Prímszám**

   Ebben a konstrukcióban a prímszám nagyságára nincs igazi nagyságrendi követelmény. A "Generálás" gombbal a legkisebb, sorozat hosszánál nagyobb, prímszámot adja vissza. A "Következő" gomb megkeresi az első jelenleginél nagyobb prímet.

3. **A konstrukcióhoz használt polinom fokszáma**

   Hasonlóan a Legendre konstrukcióhoz, a fokszámtól elvárjuk, hogy ne legyen túl nagy vagy túl kicsi. Az alsó határ jelen megvalósításban 2-nek, a felső $5 \cdot \sqrt[10]{p}$-nek lett megszabva, ahol p a használt prímszám. A "Generálás" gombbal egy fokszámot kapunk a fenti intervallumból.

4. **A polinom**

   A Legendre konstrukcióhoz hasonlóan fontos, hogy a polinomnak csak egyszeres gyökei legyenek. Ezért a polinomot legenerálhatjuk úgy, hogy a gyökeit soroljuk fel, szóközzel elválasztva. A fokszámmal megegyező számú gyököt kell megadni. Lehetőség van polinomot generálni a fenti követelményeknek eleget téve a "Polinom generálása" gombbal.

2.3.5 ChaCha20 konstrukció

A ChaCha20 egy stream cipher eljárás, mely a Salsa20 egy változata. Mindkét algoritmust Daniel J. Bernstein német-amerikai matematikus, kriptológus fejlesztette ki. A Salsa változatot 2005-ben tervezte, 2007-ben publikálták először [https://cr.yp.to/snuffle/s 20071225.pdf], az eSTREAM projekt keretében belül. A ChaCha változat 1 évvel később, 2008-ban lett publikálva, célja az volt, hogy növelje a lavinahatást és ugyanolyan, vagy picit jobb teljesítményű legyen mint a Salsa.

Mindkét eljárás eszközei a bitenkénti XOR (kizáró vagy), a 32-bites összeadás mod 2^{32}, és a konstans távolságú forgatás egy belső állapoton, ahol az állapot 16 db 32-bites adatból (szóból) áll. Csak ezen műveletekkel az algoritmus kivédi az időzítéses támadásokat. Ez olyan támadás, ahol a támadó fél a futtató rendszer hiányosságait kihasználva (tehát nem a szoftver implementációja a baj) a kriptográfiai algoritmus futási idejéből próbálja feltörni a rendszert, mivel a be-

menettől függhet a futási idő. Csak fix 32-bites műveleteket használva a futási idő megegyezik minden bemenetre, mivel az algoritmus fixen 20 menetben végzi fel a fenti műveleteket és a műveletek száma is fix minden menetben.

A ChaCha algoritmus állapotrendszerének és algoritmusa részletes leírása a fejlesztői dokumentációban található. Jelen megvalósítás az RFC 8439-ben [https://tools.ietf.org/h definiáltakat követi melyet az IRTF publikált 2018-ban, de ezen kívül több változata van az algoritmusnak, főként a körök számára vonatkozóan.

A szükséges bemenetek:

1. **A sorozat hossza**

   A sorozat hosszát bájtokban kell megadni. Bármennyi lehet.

2. **Kezdőállapot**

   A kezdőállapot 16 darab 32 bites szóból áll, jelen implementációban az egyszerűbb használat miatt 16 darab előjel nélküli 32 bites egész számot jelent. Az állapot 4 részre bomlik, ebből az egyik kategória konstans, 4 darab szóval, így ezt nem kell megadni. A maradék 3 csoport 12 szavát kategóriánként kell megadni:

   (a) *Kulcs*

   Ez 8 darab állapotot fed le. Jelen esetben 8 darab számot kell megadni a $[0, 2^{32} - 1]$ intervallumban. Tudunk kulcsot generálni a fenti intervallumból vett számokkal a "Generálás" gombra kattintva.

   (b) *Számláló*

   A jelen megvalósításban ez egy szót fed le (van verzió, ahol 64 bites a számláló, vagyis 2 szót foglal le), a megvalósításban ez minden "körében" az algoritmusnak növelődik, mondhatni számolja az iterációkat. Egy számot kell megadni a $[0, 2^{32} - 1]$ intervallumban.

   (c) *Egyszer használatos kulcs (nonce)*

   Ez a maradék 3 szót foglalja magában. Ez a kulcs azért van külön kategóriában mert ezt a kulcsot csak egyszer lehet használni egy titkosított kommunikációban, hogy kivédje az újraküldéses támadásokat. Ez általában egy tetszőleges szám, de mindig más. Jelen megvalósításban ez 3 darab

szám megadása a $[0, 2^{32} - 1]$ intervallumban. Generálhatunk ilyen egyszer használatos kulcsot a fenti intervallumból vett számokkal a "Generálás" gombra kattintva.

3 Fejlesztői dokumentáció

3.1 Konstrukciók

3.1.1 Legendre konstrukció

Ismertető

Def.: Ha p egy prímszám, és $a \in \mathbb{Z}$, akkor az $\left(\frac{a}{p}\right)$ Legendre-szimbólum értéke:

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{ha } a \text{ kvadratikus maradék modulo } p \text{ és } a \not\equiv 0 \pmod{p} \\ -1 & \text{ha } a \text{ nem kvadratikus maradék modulo } p \\ 0 & \text{ha } a \equiv 0 \pmod{p} \end{cases}$$

A Legendre-szimbólum véletlenségét már régóta vizsgálták, ez volt a motiváció a konstrukció megvalósítására.

Algoritmus

1. Vegyünk egy olyan p prímet, hogy a 2 legyen primitív gyök modulo p
2. Vegyünk egy olyan $f \in \mathbb{F}_p[x]$ polinomot, melynek csak egyszeres gyökei vannak, és foka nem túl nagy p-hez képest
3. k hosszú sorozat esetén a $(e_0, e_1, ..., e_{k-1})$ sorozat n. tagja:

$$e_n = \begin{cases} \left(\frac{f(i)}{p}\right) & \text{, ha } f(i) \nmid p \\ 1 & \text{, ha } f(i) \mid p \end{cases}$$

Extra követelmények

Ha egy n hosszú sorozatot szeretnénk generálni, akkor a használt p prímre legyen igaz, hogy $p > 2n$. A polinom fokszáma szeretnénk, ha nem lenne túl kicsi, és a

felső határt p-hez viszonyítva kell megadni. Jelen megvalósításban ha k a polinom fokszáma, akkor legyen $2 \leq k \leq 5 \cdot \sqrt[10]{p}$.

3.1.2 RC4 konstrukció

Ismertető

Az RC4 stream cipher algoritmus a bájt hosszúságú bitsorozatok egy permutációjának előállításával kezdődik, ami egy kulcssorozat segítségével történik. Ez a key-scheduling fázisa az algoritmusnak.

Ez után következik a tényleges sorozat generálás. Ameddig kell a sorozatot generálni (pl. hossz alapján), addig az algoritmus lépésenként keveri a bájtok sorrendjét, és közben egy bájtnyi kimenetet generál.

Ebben a megvalósításban a klasszikus, módosítások nélküli algoritmust valósítottam meg, hogy ez hogyan viszonyul a megvalósított mértékekhez. A módszernek több változata van, amik javíthatják a véletlenségét a generált sorozatnak.

Algoritmus

1. Töltsünk fel egy 256 elemű S tömböt úgy, hogy i. indexére: $S[i] := i$. 0-tól indexelünk, és gyakorlatilag az összes 8 hosszúságú bitsorozatot kell előállítani
2. Adjunk meg egy n méretű K (ebben a megvalósításban egy n karakterű ASCII kódolású szöveg) kulcstömböt, ahol $1 \leq n \leq 256$
3. Ez az algoritmus key-scheduling fázisa. Legyen $i, j := 0$, majd:
   (a) Legyen $j := (j + S[i] + K[i \bmod n]) \bmod 256$
   (b) Cseréljük ki $S[i]$ és $S[j]$ értékét
   (c) Inkrementáljuk i-t, ha $i < 256$, akkor ugorjunk vissza (a)-ra
4. Ez a sorozatgeneráló fázis. Legyen $i, j := 0$, n-szer kell generálni 1 bájtnyi kimenetet, tehát ennek követésére legyen $l := 0$, majd:

(a) Legyen: $i := (i + 1) \bmod 256$, majd $j := (j + S[i]) \bmod 256$

(b) Cseréljük ki $S[i]$ és $S[j]$ értékét

(c) Legyen $T := S[(S[i] + S[j]) \bmod 256]$, T lesz a mostani generálási lépés 1 bájtnyi kimenete

(d) Inkrementáljuk l-t, ha $l < n$, akkor ugorjunk vissza (a)-ra

Extra követelmények

Elvárjuk, hogy a kulcssorozat ne legyen túl rövid (általában 40 bit az alsó határ). Jelen megvalósításban: $n \geq 5$, vagyis legalább 5 karakter hosszú kulcs kell.

3.1.3 Additív karakteres konstrukció

Bevezető

Az additív karakteres konstrukció a Legendre szimbólumoshoz áll legközelebb. A Legendre szimbólumos konstrukció jobb véletlenségi tulajdonságokat mutat. Az additív karakteres sorozat azonban nagyobb sorozatcsaládot képes előállítani, és implementációja egyszerűbb.

Nevét a konstrukció jó tulajdonságait bizonyító tételben használt additív karakterekről kapta.

Algoritmus

1. Vegyünk egy páratlan p prímet
2. Vegyünk egy olyan $f \in \mathbb{F}_p[x]$ polinomot, melynek csak egyszeres gyökei vannak, és foka nem túl nagy p-hez képest
3. k hosszú sorozat esetén a $(e_0, e_1, ..., e_{k-1})$ sorozat n. tagja:

$$e_n = \begin{cases} 1 & , \text{ ha } f(i) \bmod p < \frac{p}{2} \\ 0 & , \text{ ha } \frac{p}{2} \leq f(i) \bmod p \end{cases}$$

Extra követelmények

A polinom fokszáma szeretnénk, ha nem lenne túl kicsi, és a felső határt p-hez viszonyítva kell megadni. Jelen megvalósításban ha k a polinom fokszáma, akkor legyen $2 \leq k \leq 5 \cdot \sqrt[10]{p}$.

3.1.4 ChaCha20 konstrukció

Bevezető

A ChaCha20 stream cipher eljárás egy belső állapoton végzett bitenkénti XOR (kizáró vagy), összeadás (mod 2^{32}) és konstans távolságú forgatásokkal működik. Az állapot 16 darab 32 bites szóból tevődik össze, melynek kezdeti felépítése:

Const	Const	Const	Const
Key	Key	Key	Key
Key	Key	Key	Key
Counter	Nonce	Nonce	Nonce

A konstans mezők együttes értéke az "expand 32-byte k" szöveget adja ki. A fő művelet az úgynevezett negyedkör (quarter round), ez egy 4 paraméteres eljárás, melynek minden paramétere egy 32-bites szó változója. Jelöljük ezt QR(a, b, c, d)-vel, ekkor a QR(a, b, c, d) eljárás (C stílusú pszeudokódként):

$$a \mathrel{+}= b;\ d \mathrel{\hat{}}= a;\ d \mathrel{<<<}= 16;$$
$$c \mathrel{+}= d;\ b \mathrel{\hat{}}= c;\ b \mathrel{<<<}= 12;$$
$$a \mathrel{+}= b;\ d \mathrel{\hat{}}= a;\ d \mathrel{<<<}= 8;$$
$$c \mathrel{+}= d;\ b \mathrel{\hat{}}= c;\ b \mathrel{<<<}= 7;$$

Ahol az $a \mathrel{+}= b$ az a és b bitenkénti összeadását jelenti mod 2^{32}. A moduláris összeadás a megvalósításban simán összeadást jelent kettő előjel nélküli 32 bites egész szám között ahol a maradékot az overflow kezeli. Az eredmény az a változóba kerül.

Az a ^= b a bitenkéni XOR, vagyis kizáró vagy műveletet jelöli. Az eredmény az a változóba kerül.

Az a <<<= b az a 32 bites változó balra forgatását jelöli b-vel. A b-vel való balra forgatás egy bitsorozat balra shiftelését (a magasabb helyi értékek felé shiftelünk) jelenti b-szer úgy, hogy a magas helyi értéken túlmenő bitek visszakerülnek a legkisebb helyi értékre (más néven: ciklikus shiftelés). A forgatás eredménye szintén az a változóba kerül.

Algoritmus

Az algoritmus 20 kört hajt végre, egy körben 4 negyedkört csinálva, váltogatva a körök csoportját, hogy éppen melyik indexen hajtódnak végre a negyedkörök. Kétfajta csoport van, az egyik az oszlopos, másik a diagonális kör. Ha a 16 szavas állapot 4x4-es mátrixát az alábbiak szerint indexeljük:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Akkor a két fajta negyedkör az alábbiak szerint hajtódik végre:

1. **Oszlopos kör**

   Mindegyik negyedkör az egyik oszlopban lévő elemeken hajtódik végre, sorban az 1. oszloptól kezdve, tehát a pszeudokódja (a fent definiált QR eljárással):
   QR(0, 4, 8, 12);
   QR(1, 5, 9, 13);
   QR(2, 6, 10, 14);
   QR(3, 7, 11, 15);

2. **Diagonális kör**

   Mindegyik negyedkör az egyik diagonálison álló elemeken hajtódik végre a főátlótól jobbra haladva. A pszeudokódja:

QR(0, 5, 10, 15);

QR(1, 6, 11, 12);

QR(2, 7, 8, 13);

QR(3, 4, 9, 14);

Az oszlopos körök csoportját megfeleltetjük a páratlan, a diagonális körökét a páros sorszámú körökkel. A 20 kör tehát 10 darab dupla körből áll, ahol az első kör oszlopos, a második diagonális. Tehát 10 iterációnyi dupla kört kell futtatni, ezzel együtt a ChaCha20 precíz algoritmusa:

1. A ChaCha20 állapotát elő kell állítani, ezt a programban számgenerálással, illetve manuálisan lehet megadni. A kezdeti belső ChaCha állapot a fent leírt mátrixos alakban legyen I, ezt le kell másolnunk egy átmeneti X állapotba. $I[i]$ az I állapot i. indexű elemére hivatkozik a fenti indexeléssel.

2. Legyen $i := 1$, majd hajtsunk végre egy dupla kört az X, másolt állapoton, vagyis:

   (a) Oszlopos kör:

   QR($X[0]$, $X[4]$, $X[8]$, $X[12]$);

   QR($X[1]$, $X[5]$, $X[9]$, $X[13]$);

   QR($X[2]$, $X[6]$, $X[10]$, $X[14]$);

   QR($X[3]$, $X[7]$, $X[11]$, $X[15]$);

   (b) Diagonális kör:

   QR($X[0]$, $X[5]$, $X[10]$, $X[15]$);

   QR($X[1]$, $X[6]$, $X[11]$, $X[12]$);

   QR($X[2]$, $X[7]$, $X[8]$, $X[13]$);

   QR($X[3]$, $X[4]$, $X[9]$, $X[14]$);

   (c) Inkrementáljuk i-t, ha $i <= 10$, akkor ugorjunk vissza (a)-ra.

3. Egy menet kimenete egy 512 bites ($16 \cdot 32$, az állapot mérete) sorozat, legyen ez O, ugyanolyan felépítéssel mint az X és I (16 méretű 32 bites szavak tömbje). Legyen ekkor $\forall\ i \in \{0, 1, ..., 15\}$-re:

$$O[i] = X[i] + I[i]$$

Ezzel előállítottuk a kimeneti O bitsorozatot.

4. Növeljük az belső állapot számlálóját, vagyis inkrementáljuk $I[12]$-őt.

Nyilvánvalóan a tetszőlegesen hosszú stream generáláshoz az O bitsorozat nem elég. A fenti algoritmust $ChaChaRound()$-al jelölve a stream generálás algoritmusa a következőféleképpen alakul:

1. Legyen l a generálandó sorozat mérete, ezt fogjuk használni az algoritmusban arra is, hogy számon tartsuk még mennyi bájtnyi adatot kell előállítani. A kimenetnek megfelelő bitsorozat legyen O.
2. Amíg $l \geq 64$, addig:
   (a) Hajtsuk végre a $ChaChaRound()$ eljárást. Az eljárás teljes kimenetét fűzzük az O kimeneti bitsorozathoz.
   (b) Csökkentsük l-t 64-el, vagyis legyen $l := l - 64$.
3. Ha $l > 0$: Még egyszer hajtsuk végre a $ChaChaRound()$ eljárást. A maradék l bájtnyi kimenetet az eljárás eredményeként kapott első l bájtjából állítjuk össze, a mátrixos indexeléssel élve. Ha már egy teljes szónyi adat nem fér bele O-ba, akkor az utolsó szóból ami benne van az első l bájtban már csak az első $l \bmod 4$ bájtot fűzzük a kimenethez.

Jelen körülmények között extra követelményt nem támasztunk az algoritmus felé, de a kezdeti állapotot kifejezetten kell ellenőrizni, ha sokat akarunk az algoritmussal generálni. Ha sok kört fut be ugyanaz az állapot, akkor a számláló körbeér, és ismétlődni fognak a generált értékek. Ekkor mindenképpen új állapotot kell választani, tehát maximálisan $2^{32} - 1$ darab $ChaChaRound()$ hívás legyen mielőtt új állapotot készítünk.

Ezen kívül a nonce részét az állapotnak minden titkosított adatvitel esetén csak egyszer használjuk egy adott kulccsal, hogy kikerüljük a visszajátszásos támadásokat.

A ChaCha algoritmust még nem vizsgálták kimerítően kriptográfiailag, csak a Salsa változatot. A jelenlegi legjobb ismert támadás 2^{109} művelettel töri fel a 7

menetes, és 2^{250} művelettel a 8 menetes Salsa titkosítást.

Azt is megmutatta 2013-ban Nicky Mouha és Bart Prencel [https://eprint.iacr.org/2013/328.pdf]
hogy a Salsa 128-bites biztonságot jelent a differenciális kriptoanalízises támadásokkal
szemben. Ezek olyan támadásokat jelentenek, mely a bemenet megváltoztatásából
a kimenet változásaira próbálnak következtetéseket levonni.