

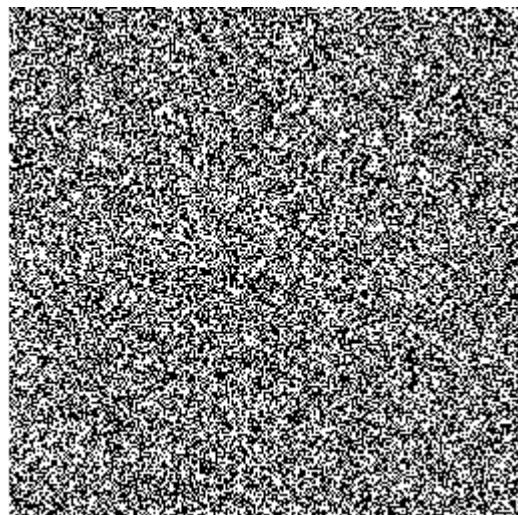
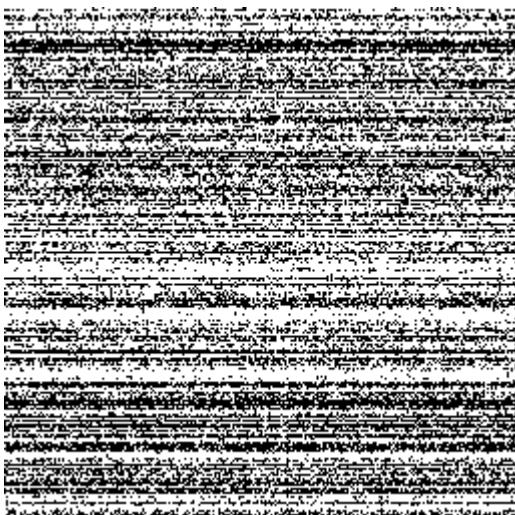
# Knækning af Pseudotilfældige Tal i Browseren

Kryptoanalyse af Math.random(): Udnyttelse af xorshift128+-algoritmen gennem Symbolsk Solver til at Forudsige og Manipulere Spil i Browseren

Asger Finding | L3ak 23/26

Studieområdprojektet 2025-2026

Matematik	A
Teknikfag (DDU)	B



TEC - H. C. Ørsted Gymnasiet Lyngby

Vejledere: Christian Bøge-Rasmussen (cbr), Mikkel Christensen Lund (mclu)

Afleveringsfrist: 12. dec. 2025

Tegn: 47 411

## RESUME

Dette studieområdeprojektet undersøger generering af tilfældige tal gennem to metoder: TRNG og PRNG. Da binære computere er deterministiske, skelnes der mellem Ægte Tilfældige Talgeneratorer (TRNG) samt Kryptografisk Sikre Pseudo-Tilfældige Talgeneratorer (CSPRNG), der er kryptografisk sikre, men langsomme, og Pseudo-Tilfældige Talgeneratorer (PRNG), der er hurtige, men forudsigelige, da de udelukkende afhænger af en starttilstand.

Opgaven analyserer den konkrete PRNG xorshift128+, der anvendes i moderne browser-engines til implementeringen af `Math.random()`. Der redegøres for algoritmens matematiske principper, og dens styrker og svagheder vurderes gennem statistiske tests om ensartethed og uafhængighed. Opgavens hovedfokus er at bryde algoritmen ved at udnytte dens deterministiske natur.

Gennem anvendelsen af den symbolske solver Z3, demonstreres hvordan xorshift128+'s interne tilstand kan rekonstrueres ud fra en sekvens på 5 observerede outputs med 52-bit præcision. Metoden anvendes i et praktisk eksempel i en naivt designet digital roulette, hvor alle forudsagte vindertal matcher de faktiske resultater, hvilket bekræfter metodens effektivitet.

De statistiske tests ( $\chi^2$ -test og autokorrelation) viser, at xorshift128+ producerer ensartede og tilsyneladende uafhængige output-sekvenser i modsætning til en simpelere PRNG'er som LCG, der konsekvent fejler begge.

Afslutningsvis vurderes det, at xorshift128+ opfylder sit designmål for ikke-sikkerhedskritiske formål, men at den er fundamentalt uegnet til kryptografiske formål, hvor CSPRNG'er som Web Crypto API eller `/dev/urandom` er strengt nødvendige for at undgå tilstandsrekonstruktion og talforudsigelse.

## INDHOLD

Resume	2
Indhold	3
Introduktion	4
Teoretisk Grundlag for TRNG, PRNG, CSPRNG og Statistiske Tests	4
A. TRNG	4
1. Entropi i en TRNG - Entropikilder, Conditioning & Seeding	4
2. Helbredsværdig af Entropi - RCT & APT	4
B. PRNG	7
1. Periode og Intern Tilstand (State)	7
2. CSPRNG (Kryptografisk Sikkerhed) vs. PRNG (Hastighed)	7
C. Determinisme	7
1. Determinisme i Kryptografiske Sikkerhedsapplikationer	7
2. Anvendelse af en CSPRNG i Kryptering	8
D. Statistiske Tests	8
1. $\chi^2$ -test for Ensartethed	8
2. Autokorrelation for Uafhængighed	9
Analysen af PRNG'en xorshift128+	10
Statistisk Analyse af Tilfældighed	12
E. Eksperimentdesign	12
F. Hypoteseramme	12
G. Resultater	13
H. Analyse og Fortolkning	13
I. Begrensninger	13
Kryptoanalyse på xorshift128+	13
Real-world Udnyttelse af Output-forudsigelse	15
Diskussion og Perspektivering	18
J. Resultater af Forsøg	18
K. Talsgenerators' Implikation for Sikkerhed	18
L. Udvidelse af Angreb til Andre Applikationer	18
Konklusion	19

## INTRODUKTION

Tilfældige tal er en grundlæggende byggesten i moderne sikkerhed og design, og anvendes i en bred vifte af teknologier. Alt fra sikkerhed; kryptering og autentificering, til simulering, til processuel generering (f.eks. støjmønstre som Perlin Noise) og computergrafik, foreslæede kodeord, spil og netværksprotokoller. Fra onlineservere til den mindste mikrochip, er behovet for talfølger, der opfører sig tilfældigt, konstant. Computere er dog binære af natur, og dermed altid forudsigelige, og ydermere kan de ikke generere et reelt tilfældigt tal. Derfor, når vi har brug for et tal i en kryptografisk sikker sammenhæng, indsamles der først ægte tilfældighed fra en fysisk hardwarelæsning (*True Random Number Generator*, “**TRNG**”), som konditioneres og anvendes til at seede en kryptografisk sikker pseudotilfældig talgenerator (*Cryptographically Secure PseudoRandom Number Generator*, “**CSPRNG**”). CSPRNG’en strækker derfor denne højkvalitets-entropi over en stor mængde sikre pseudotilfældige tal. En CSPRNG er imidlertid en betydelig langsom operation, som ikke er egnet til sammenhænge, hvor vi har brug for tilfældige tal hurtigt og i store mængder - og hvor sikkerheden ikke spiller hovedrollen (European Information Technologies Certification Academy [EITCA], 2014).

Softwareudviklere og matematikere har udviklet “tilfældige” talalgoritmer, der kun er afhængige af én indledende værdi (et “**Seed**”), og som derefter forsyner sig selv med en ny intern tilstand: en pseudo-tilfældig talgenerator (*Pseudo Random Number Generator* “**PRNG**”) (European Information Technologies Certification Academy [EITCA], 2014). Ulempen ved en PRNG er imidlertid, at den er baseret på en fast algoritme, der er afhængig af én indledende tilstand, og kan derfor være forudsigelig, når den analyseres.

Dette studieområdaprojektet vil undersøge forskellen mellem pseudo-tilfældige tal og ægte tilfældige tal, hvordan et pseudo-tilfældigt tal *kan* genereres, og hvilken rolle det spiller i et moderne softwaremiljø. Først redegøres for det teoretiske grundlag omkring en TRNG, CSPRNG og en PRNG. Derefter analyseres xorshift128+, en PRNG, der implementeres i alle moderne browser engines (De Mooij, 2015): V8 (Chromium), SpiderMonkey (Firefox) og WebKit (Safari). Der redegøres for matematikken og principperne bag algoritmen, og der vurderes, hvordan vi kan analysere en tilfældig talgenerators styrke og svagheder, ud fra statistiske tests. Vi vil så analysere xorshift128+ i dybden, og bryde algoritmen gennem den symbolske solver Z3, så vi kan genskabe den interne state og forudsige det næste tal i den “tilfældige” talfølge (et “**Output**”), når vi har lært en række successive outputs. Vores løsning vil anvendes i et praktisk eksempel, hvor vi kan udnytte forudsigeligheden af xorshift128+ til at opnå en fordel eller hacke et program. Vi vil vurdere styrken af xorshift128+, samt dens forgænger xorshift128, og en simpel selvskreven PRNG, imod vores række statistiske tests, for at vurdere ensartetheden og uafhængigheden af talfølgen. Til sidst vil vi diskutere i hvilke tilfælde PRNG’er egner sig i praksis, og hvornår en TRNG foretrækkes - samt hvilke designvalg vi skal træffe for at sikre platformssikkerhed.

## TEORETISK GRUNDLAG FOR TRNG, PRNG, CSPRNG OG STATISTISKE TESTS

### A. TRNG

#### 1. ENTROPI I EN TRNG - ENTROPIKILDER, CONDITIONING & SEEDING

Kvaliteten af output fra forskellige RNG’er varierer. Det kan forekomme, at en tilfældighedsalgoritme har ulemper i sit output. For at beskrive RNG’ers kvalitet anvendes forskellige målsætninger afhængig af dens type:

- En *TRNG* evalueres på entropi; dens *kvalitet* (graden af ægte tilfældighed)
- En *PRNG* evalueres på periode; dens *kvantitet* (længden før gentagelse)

Entropi repræsenterer graden af uorden, tilfældighed og uforudsigelighed, som TRNG’en producerer, og er deskriptivt for målet af rå tilfældighed af vores totalt tilfældige talserie. Man indsamler entropi fra fysiske processer i den virkelige verden, der er svære eller umulige at forudsige (Cao et al., 2022). Fysiske eksempler på entropi kan være termisk støj, radioaktivt henfald eller fysiske interaktioner (f.eks. interaktionstidspunkter eller musebevægelser). En TRNG fungerer ved at måle denne naturlige støj, kendt som entropikilden, og så konvertere den til en binær strøm.

Den binære strøm gennemgår en process kaldet konditionering (“**conditioning**”), hvor entropien forstærkes for at opnå en højere grad af tilfældighed (Barker, 2025). Conditioning anvender kryptografiske hashfunktioner eller extraction-algoritmer til at kondensere den rå entropi. Dette fjerner bias og korrelationer i det fysiske støj, så output har en ensartet fordeling. Højkvalitetsentropi er fundamental for initialiseringen (“**seeding**”) af en CSPRNG (Barker, 2025), som derefter kan producere en stor mængde sikre og hurtige pseudotilfældige tal. Hvis den indledende entropi er svag - eksempelvis kun 32 bits i stedet for 256 bits, reduceres sikkerhedsniveauet tilsvarende, uanset algoritmens teoretiske styrke, fordi det svageste led i CSPRNG’en bestemmer dens kvalitet.

#### 2. HELBREDSVALIDERING AF ENTROPI - RCT & APT

En TRNG kan godt fejle i sin entropi, og derfor er der defineret “Health Checks” (et helbredstjek) til at vurdere kvaliteten af ens

entropi. Det kan godt være svært at vurdere om ens TRNG er kompromitteret af den ene eller anden grund, men SP 800-90B (Turan et al., 2018) definerer to helbredstjek:

Entropikilden i en TRNG bør ikke have for mange konsekutive gentagelser af samme værdi, da dette indikerer potentiel hardwarefejl, og er forudsigtlig. Repetition Count Test ("RCT") er designet til at detektere katastrofale fejl i ens entropikilde, hvor den producerer identiske værdier gentagne gange i stedet for tilfældige uafhængige prøver ("samples"). Det vil ske, hvis entropikilden af en årsag er blevet fejlbehaftet. Testens grundprincip er at vurdere hvert output i serien, og for hvert output, der har samme værdi som det forrige, tæller den op. Hvis en anden værdi optræder, vil tælleren nulstilles. For hver iteration sammenlignes tællerens værdi med en tærskelværdi ( $C$ ) for at bedømme, om vi har overskredet den maksimale grænse for repetition.

Testens funktion beskrives i følgende diagram:

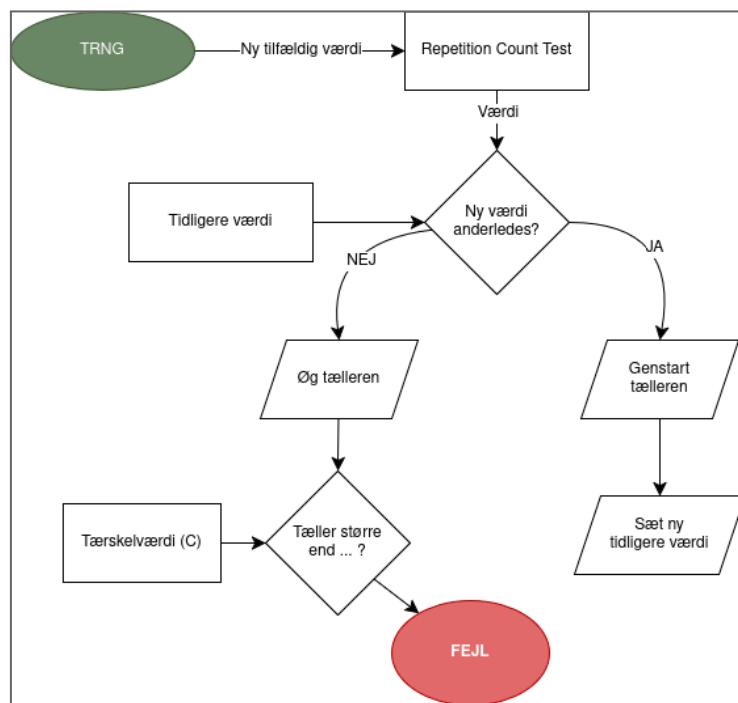


Fig. 1 Flowchart for Repetition Count Test (RCT) algoritme ifølge NIST SP 800-90B. Testen evaluerer hvert sample fra entropikilden og tæller konsekutive gentagelser af samme værdi; hvis tælleren når tærskelværdien  $C$  fejler testen, ellers nulstilles tælleren ved hver ny værdi og næste sample evalueres.

Tærskelværdien beregnes således:

$$C = 1 + \lceil \frac{-\log_2(\alpha)}{H} \rceil$$

Hvor at:

$\alpha \in ]0, 1]$ : den acceptable falsk-positive rate (f.eks.  $2^{-20} \approx 1$  pr. million)

$H \in [1, 8]$ : den estimerede minimumsentropi pr. sample (i bits).  $H = 4$  bits/byte betyder at hver byte effektivt har 4 bits entropi. Det vil sige, at hver byte bærer samme mængde uforudsigelighed som en uniform 4-bit fordeling ( $2^4 = 16$  lige sandsynlige værdier). En reduceret entropi kan skynde bias eller reduceret støjkvalitet i den fysiske entropikilde. Minimum entropi er defineret ud fra  $p$ :

$$p = 2^{-H} \Leftrightarrow H = -\log_2(p).$$

Det er okay, hvis  $p$  ikke er 0,5 (en 50/50 spredning af 0 og 1), da den rå entropi fra TRNG'en gennemgår conditioning, som transformerer dataen til en uniform distribution ved  $p = 0,5$ .

Man anvender Adaptive Proportion Test ("APT") til at vurdere, om der er sket en ændring i støjstrukturen, så frekvensen ændrer sig. I binær sammenhæng (*kun bits*) betyder det, at fordelingen mellem 0'er og 1'er er forskudt fra den forventede 50/50-fordeling. APT arbejder ved at evaluere glidende blokke (et "Vindue") af  $W$  samples. For hvert vindue tælles hvor mange gange første værdi (referencen) forekommer. Hvis den forekommer  $C$  eller flere gange, indikerer det en bias. Testen består kun, hvis alle vinduer overholder grænsen for tærskelværdien  $C$ .

APT tager højde for, om entropien er ufuldstændig, hvis der for eksempel er flere af en bestemt bit eller værdi, end der er af en

anden. Så længe entropiværdiernes fordeling er bevidst og allerede kendt, vil testen stadig bestå. Testen kaldes "adaptiv" fordi referenceværdien skifter med hvert vindue. Det er altid første sample i det aktuelle vindue, der vurderes ud fra, hvilket er vigtigt, når ens entropipulje består af ikke-binære outputs.

Formålet med APT er at overvåge algoritmens helbredstilstand på længere sigt for at se, om der er sket en uventet forskydning i entropiværdierne.

Testen kan beskrives med følgende diagram:



Fig. 2 APT-algoritme. Testen evaluerer glidende W-vinduer ved at tælle forekomster af første værdi; fejler hvis antal  $\geq C$ .

Man kan beskrive APT matematisk *på et enkelt vindue af en binær sekvens* sådan:

1. Lad en binær sekvens af bits være:

$$S = (b_1, b_2, \dots, b_n), b_i \in \{0, 1\}$$

2. Lad  $W \in \mathbb{N} = 1024$  være vinduesstørrelsen, som ifølge SP 800-90B skal være 1024 for binære sekvenser  
Antal fulde vinduer:

$$J = \lfloor \frac{S_{\text{langde}}}{W} \rfloor$$

3. Lad det  $j$ -te vindue ( $j = (0, 1, \dots, J - 1)$ ) være:

$$S_j = (S_{jW+1}, S_{jW+2}, \dots, S_{jW+W})$$

Så kan vi definere tællingen:

$$X_j = (\sum_{i=1}^W S_{jW+i})$$

4. Vi opskriver fejlbetingelsen for ét vindue.

$C \in \mathbb{N}$  er tærskelværdien, som er sandsynligheden for den mest sandsynlige værdi.

$$F(j) = \{1 \text{ hvis } X_j \geq C \vee X_j \leq W - C, 0 \text{ ellers}\}$$

APT tester begge veje. Både om referenceværdien forekommer for ofte ( $X_j \geq C$ ) eller for sjældent ( $X_j \leq C$ ). Begge scenarier indikerer et bias i fordelingen.

Sekvensen består APT, hvis ingen vinduer fejler:

$$APT(S, W, C) = \{1 hvis \sum_{j=0}^{J-1} F(j) = 0, 0 ellers\}$$

Tærskelværdien vil variere ud fra ens sandsynlighed  $p$ . SP 800-90B angiver igen en anbefalet falsk-positive rate på:

$$\alpha = 2^{-20} \approx 10^{-6}$$

Den perfekte TRNG har en minimumsentropi pr. sample,  $H$ , til 1 bit pr. bit. Det vil også sige, at  $p = \frac{1}{2}$ . Entropikvaliteten i en TRNG kan selvfølgelig variere, og man skal justere  $p$  i takt.

Sandsynligheden for præcis  $k$  forekomster af 1'ere i en serie hvor  $p$  er bias-parameteren, bestemmes gennem binomialfordelingen (Hill, 2018):

$$P(X = k) = \frac{(W \text{ over } k) \cdot p^k \cdot (1 - p)^{W-k}}{\frac{W!}{k!(W-k)!}}$$

APT tester mod den kumulative sandsynlighed. Derfor skal man etablere en tærskelværdi  $C$ . For at bestemme tærskelværdien  $C$  summerer man forekomsten op til falsk-positive raten  $\alpha$ :

$$C = \min(c \in \mathbb{N}: \sum_{k=c \text{ til } W}^W \left( \frac{W!}{k!(W-k)!} \cdot p^k \cdot (1 - p)^{W-k} \right) \leq \alpha) \Rightarrow C = 589$$

## B. PRNG

### 1. PERIODE OG INTERN TILSTAND (STATE)

Mens entropi sikrer kvaliteten af det indledende seed, bestemmer perioden hvor mange tal der kan genereres sikkert fra dette seed, før reseeding bliver nødvendigt.

Perioden i en PRNG eller CSPRNG er defineret som det antal tal, algoritmen genererer, før talfølgen og den interne tilstand begynder at gentage sig selv (cyklisk struktur). En lang periode er essentiel for at sikre sig, at brugeren ikke ser den samme sekvens af tal gentage sig. I en CSPRNG er en lang periode afgørende for sikkerheden. Længden af den interne tilstand beskrives i antal bits og er nogle gange et suffiks til algoritmens navn, som f.eks. HMAC\_DRBG med SHA-256 (Greendow, u.d.) eller xorshift-128+. Den interne periode for SHA-256-CSPRNG'en er således  $2^{256}$ , eller  $1,157 \cdot 10^{77}$  muligheder, før vi cirkler tilbage. Et brute-force-angreb på seedet involverer et forsøg på at gætte den aktuelle  $N$ -bitttilstand ved at prøve alle  $2^N$ -output-muligheder. For en CSPRNG med 256-bit seed ville det hele kræve  $2^{256}$  iterationer, hvilket betragtes som beregningsmæssigt urealistisk. Det er fundamentalt, at en CSPRNG har en stor tilstandsstørrelse, og deraf følgende stor periode, så den overstiger alle realistiske tidsrammer for angrebet.

### 2. CSPRNG (KRYPTOGRAFISK SIKKERHED) VS. PRNG (HASTIGHED)

Når man arbejder i en kryptografisk sikker sammenhæng, kræver det stærk tilfældighed i generering af nøgler, nonces og session tokens. Operativsystemet samler fysisk entropi fra hardwaren, som tilføjes til en pulje, som aflæses for øgte tilfældige bits. I stedet for at anvende dataen direkte fra puljen når der skal genereres noget kryptografisk sikkert, så anvender man en kryptografisk sikker pseudotilfældig talgenerator, CSPRNG'en, som kan "trække" entropien over en større mængde bits. Det er denne CSPRNG der bruges i praksis til at skabe nøglemateriale, engangsværdier til netværksprotokoller, session-ID'er og andre værdier, der ikke må være forudsigelige (European Information Technologies Certification Academy [EITCA], 2014).

Et godt eksempel på en CSPRNG findes i Linux-kernen (*Random Number Generation - ArchWiki*, u.d.), som vedligeholder en entropipulje der konstant og kontinuerligt fordres med ny tilfældighed indsamlet fra entropikilder i hardwaren, bl.a. CPU RDRAND (Intel) og RDSEED (AMD), interrupt timings, etc. Selve indsamlingen fodrer en CSPRNG, der kører i kernen, som der producerer den kryptografisk sikre talfølge. Outputtet sendes løbende til `/dev/urandom`, som leverer en konstant strøm af pseudotilfældige bytes uden blokering. `/dev/urandom` er i sig selv en yderst robust CSPRNG, der bruger den initiale højkvalitets-entropi som sit seed, og stream cipheren ChaCha20 som sin PRNG, og opdaterer sin interne tilstand periodisk med frisk entropi fra puljen for at modstå kryptoanalytiske angreb.

## C. DETERMINISME

### 1. DETERMINISME I KRYPTOGRAFISKE SIKKERHEDSAPPLIKATIONER

Det skal bemærkes, at der er en forskel mellem validering gennem sikker tilfældighed, og anvendelsen af *hashfunktioner*, der tager en indgangsværdi, og permanent transformerer den (Brandon, 2024). Når du indsender en *string* (eksempelvis et password) til en hashfunktion (f.eks. MD5 eller SHA-256) (*Hash Functions*, u.d.), er den ikke afhængig af tilfældighed, men anvender i stedet en algoritme på indholdet, som fører til et irreversibelt men konsistent output (f.eks. en adgangskode) for at omdanne den til en ny information, som vi kan gemme og sammenligne med. Når du opretter en konto i en tjeneste, er det korrekt praksis at *hashe* adgangskoden og ikke gemme den i direkte tekst, så hvis en hacker får adgang til din database, kan de ikke lække alle dine brugeres adgangskoder, men får i stedet kun irreversibel data, der er ubrugeligt. Når du logger ind, sender du din adgangskode til serveren, som derefter kan hashe den indtastede adgangskode mod den tilsvarende hashede adgangskode i databasen for at validere brugeren.

### 2. ANVENDELSE AF EN CSPRNG I KRYPTERING

Når man skal validere en bruger i en server, bruger man håndtryksprincippet (Mol, 2020). Når en forbindelse etableres (evt. ved at en bruger logger ind på et website) genereres der et langt, uniform, uforudsigligt tal via sin CSPRNG. Dette bliver klientens identifikator, som den skal vedhæfte, ofte via en *cookie*, med hver anmodning til serveren, for at serveren sammenknytter anmodningen med den rigtige bruger, og validerer anmodningen. Hvis man i stedet sender en ugyldig token, bliver ens anmodning afvist på serveren. Det er standardpraksis at tildele en token en udløbsdato, så brugeren skal periodisk logge ind på tjenesten igen. Dette er for at forhindre, at aktører stjæler en token og logger ind som brugeren uden at kende vedkommendes adgangskode.

Det er også nødvendigt at bruge en CSPRNG når man arbejder med nøglepar (Smith & Community, 2025). Offentlige og private nøgler er baseret på asymmetrisk kryptografi, hvor den private nøgle holdes hemmelig, og bruges til dekryptering af data og til at lave digitale signaturer, og den offentlige nøgle må distribueres frit og bruges til kryptering og signaturvalidering. Sikkerheden bygger på en envejsfunktion, hvor man frit kan anvende den offentlige nøgle, men hvor det er beregningsmæssigt urealistisk at genskabe den private nøgle ud fra den, fordi algoritmens matematiske struktur gør inversion praktisk umulig (GeeksforGeeks, 2025).

PGP bruger nøglepar til både kryptering og signering af beskeder. Afsenderen krypterer beskeden med modtagerens offentlige nøgle, og kun modtager kan dekryptere beskeden igen med deres private nøgle. Sikkerhed sikres gennem konceptet *web of trust*, hvor man etablerer autenticitet ved at nøglerne er bundet kun til ejer (Judge, 2024).

## D. STATISTISKE TESTS

### 1. $\chi^2$ -TEST FOR ENSARTETHED

Vi anvender Pearson's  $\chi^2$ -test (Chi i anden test) for at vurdere, om outputtet fra en PRNG er ensartet fordelt over det forventede interval (H. G., u.d.). Testen er fundamental for at vurdere algoritmens kvalitet, fordi ægte tilfældige tal altid skal forekomme med næsten samme hyppighed, så der ikke er bias eller favorisering imod en særlig værdi.  $\chi^2$ -testen giver en størrelse for talseriens ensartethed imod en forventet fordeling, som kan vurderes imod en tabelværdi (*den kritiske værdi*), der fortæller os om vi får en uniform fordeling eller et bias i vores algoritme.  $\chi^2$ -testen er en højresidet test, som betyder, at den kun vurderer den øvre tærskel/en dårlig tilpasning. I virkeligheden bør der også tages højde for den venstre side, hvor en  $\chi^2$ -værdi tæt på 0 indikerer en *for god* tilpasning; hvilket peger på, at der er potentiel determinisme i PRNG'en.

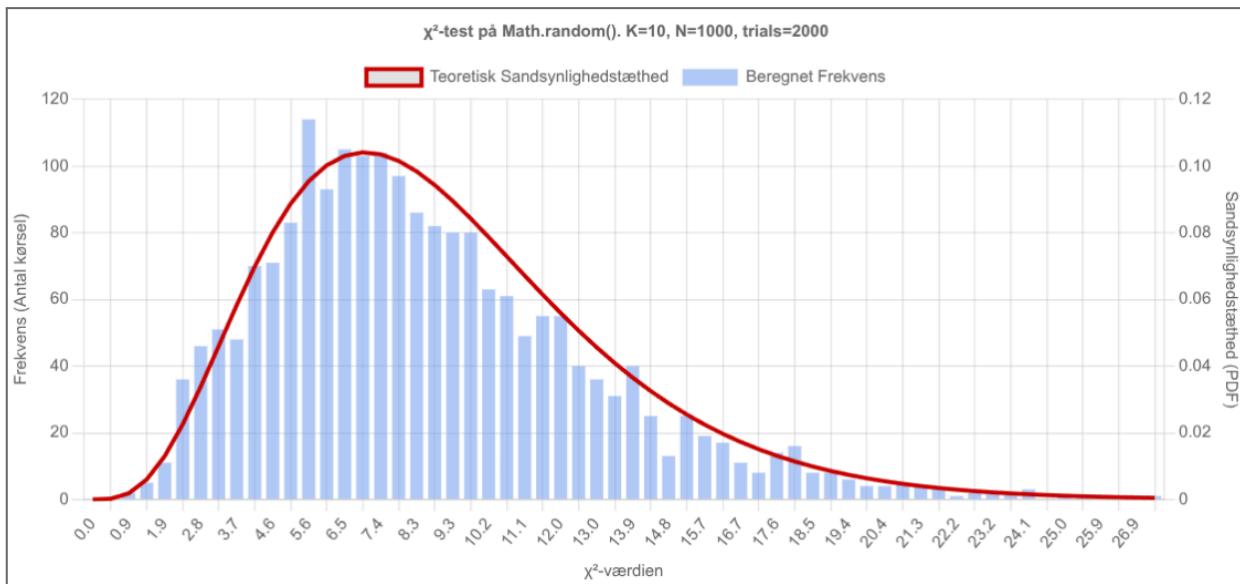


Fig. 3 Sammenligning af den beregnede frekvens (blå søjler, venstre akse) af  $\chi^2$ -værdier fra 2000 kørsler med den teoretiske sandsynlighedstæthed (rød kurve, højre akse). Det tætte overlap bekræfter, at algoritmen genererer tal, der følger en ensartet fordeling.

Ideen er, at man opdeler ens output i  $k$  antal lige store intervaller ("Bins") af outputtets minimum- og maksimumsværdier (f.eks. [0.1, 1.0] for `Math.random()`). Man bestemmer en iterationsmængde  $N$ , og i et perfekt scenario vil fordelingen af output-værdier være uniform over alle bins. Vi kan beskrive vores bins således:

- 1 Bin 1: [0.0, 0.1[
- 2 Bin 2: [0.1, 0.2[
- ...
- $k$  Bin 10: [0.9, 1.0[

Da vi forventer en uniform fordeling over alle bins (samme hyppighed for alle værdier), kan vi beskrive den forventede hyppighed ud fra det genererede antal tal  $n$  og antallet af bins  $k$ :

$$E_i = E = \frac{N}{k}$$

Den observerede hyppighed ( $O_i$ ) er optællingen af outputs imod vores bins; altså hvor vores resultater lander i mængderne. Med den forventede hyppighed og den observerede hyppighed bestemmer vi  $\chi^2$ -størrelsen som summen af den kvadrate forskel mellem den observerede og den forventede hyppighed, normaliseret med den forventede hyppighed:

$$\chi^2 = \sum_{i=1}^k \frac{(O_i - E_i)^2}{E_i}$$

En lille  $\chi^2$ -værdi indikerer, at den observerede hyppighed ( $O_i$ ) er tæt på de forventede hyppigheder ( $E_i$ ), hvilket understøtter ensartetheden af algoritmen. På den anden hånd vil en stor  $\chi^2$ -værdi indikere en stor afvigelse fra ensartetheden.

Når vi har en  $\chi^2$ -værdi, kører vi en hypotesetest.  $H_0$  (nulhypotesen), angiver at vi har en uniform fordeling (H. G., u.d.). Alternativet, kaldet  $H_A$ , fortæller os at kørslen har et bias. Vi vurderer vores hypotese imod  $\chi^2_{kritisk}$  ("den kritiske værdi"), som angiver, ud fra en signifikansniveauet  $\alpha$  (f.eks. 5 % / 0,05) og frihedsgraderne ( $k - 1$ ) (DF), om vores data er uniform eller har et bias. Hvis den beregnede  $\chi^2$ -værdi er større end den kritiske værdi, forkastes  $H_0$ . Det giver mening at lave mange kørsler af  $\chi^2$ -testen på en PRNG, for at vurdere resultaterne imod signifikansniveauet. Under mange kørsler ( $N \rightarrow \infty$ ), hvis vores algoritme er uniform, vil den beregnede  $\chi^2$ -værdi følge asymptoten af den teoretiske  $\chi^2$  ved  $k - 1$  frihedsgrader.

## 2. AUTOKORRELATION FOR UAFHÆNGIGHED

Vi anvender autokorrelationstesten for at vurdere, om tallene genereret af en algoritme er uafhængige af hinanden. Den sikrer, at der ikke går nogle talserier igen i sekvensen, eller at der er glatte overgange mellem outputs, som kan udnyttes i en forudsigelse. En talgenerator kan godt virke uafhængig ved værdier tæt på hinanden, men der kan opstå mønstre eller svingninger, som bryder

uafhængigheden og gør tilfældigheden forudsigelig i det længere løb. Givet en sekvens af tal  $x_1, x_2, \dots, x_N$  vælger man et lag  $k$ . Man undersøger om værdierne  $k$  trin fra hinanden følger hinandens udsving - altså, hvis den ene værdi afviger opad fra middelværdien samtidig med at den anden også gør det, bevæger de sig sammen ( $r_k > 0$ ). Hvis den ene går op mens den anden går ned, bevæger de sig modsat ( $r_k < 0$ ). Hvis der ingen sammenhæng er, kan vi konkludere, at de er uafhængige ( $r_k = 0$ ) (Leonardo, u.d.), (Nguyen, 2021).

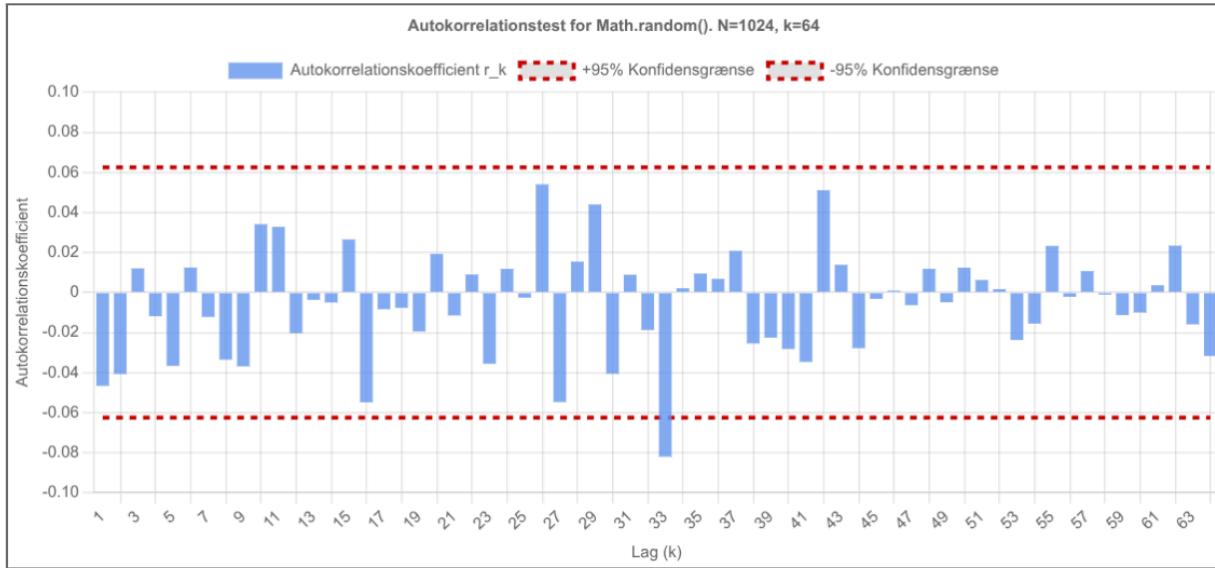


Fig. 4 Autokorrelation af en PRNG med  $r_k$  fra 64 par med talmængde 1064. Værdier sammenlignes med  $\pm 95\%$  konfidensgrænsen. Det kan observeres at kun én autokorrelationsværdi ved  $k = 33$  overgår grænsen. Det bekræfter, at algoritmen genererer tal, der er uafhængige af hinanden.

$$r_k = \frac{\sum_{i=1}^{N-k} (x_i - \bar{x})(x_{i+k} - \bar{x})}{\sum_{i=1}^N (x_i - \bar{x})^2}$$

Hvor at:

$\bar{x}$  er middelværdien af sekvensen.

Tælleren mäter, hvor meget parrene  $(x_i, x_{i+k})$  bevæger sig sammen. Hvis store tal ofte følges af store tal med en  $k$  forskydning, vil tælleren blive positiv. På den anden side, hvis store tal typisk følges af små tal, bliver den negativ. Udtrykket i nævneren normaliserer tælleren, så resultatet ligger mellem  $[-1, 1]$ .

Da udtrykket kun vurderer tallene ud fra én værdi af  $k$ , giver det mening at køre autokorrelationstesten over flere bølgelængder/en større mængder af  $k$ -værdier (f.eks.  $k = \{1, \frac{N}{10}\}$ ).

En ideel PRNG har  $r_k \approx 0$  for alle  $k > 0$ . Ethvert afvigende  $r_k$  vil antyde et gentagende mønster ved  $k$ 's forskydning, og sekvensen vil ikke være uafhængig.

#### ANALYSE AF PRNG'EN XORSHIFT128+

xorshift128+ er den nuværende PRNG-algoritme, som implementeres i alle primære browser-engines, herunder V8 for Chromium/Google Chrome ("random-number-generator.h," 2025), SpiderMonkey for Firefox ("XorShift128PlusRNG.h," 2018) og WebKit for Safari ("WeakRandom.h," 2025). xorshift128+ var først implementeret i V8 ved commit 085fed0 den 24. november 2015. Den afløser PRNG'en MWC1616 (*Multiply With Carry*, der kombinerer to 16-bit dele), som var i stand til at generere et 32-bit værdi (dvs. den havde maksimalt en  $2^{32}$  periodelængde) (Guo, 2015). JavaScript's `Number`-primitiv følger IEEE 754-standarden for dobbeltpræcision 64-bit binært format, som tildeler 1 bit til fortegnet (er tallet positivt eller negativt?), 11 bits til eksponenten (mellem -1022 til 1023) og 52 bits til mantissen (tallets repræsentation som et decimaltal mellem 0 og 1) (Mozilla, u.d.-a). Derfor blev det anset for utilstrækkeligt med 32 bits, fordi JavaScript i virkeligheden er i stand til at repræsentere  $2^{52}$  i et decimaltal. Siden `Math.random()` returnerer værdier i  $[0, 1]$ , udnyttes kun mantissens 52 bits præcision. Det betyder dog, at `Math.random()` kan generere  $2^{52} = 4,5 \cdot 10^{15}$  forskellige værdier, mens V8 med MWC1616 før kun havde  $2^{32} = 4,3 \cdot 10^9$  mulige værdier - en forskel

på over 1 million gange.

Lad os kigge på xorshift128+'s implementering i V8:

```
uint64_t state0_;
uint64_t state1_;

// Static and exposed for external use.
// Generate random numbers using xorshift128+.

static inline uint64_t XorShift128(uint64_t* state0, uint64_t* state1) {
    uint64_t s1 = *state0;
    uint64_t s0 = *state1;
    *state0 = s0;
    s1 ^= s1 << 23;
    s1 ^= s1 >> 17;
    s1 ^= s0;
    s1 ^= s0 >> 26;
    *state1 = s1;
    return s0 + s1;
}
```

Fig. 4 funktionsdefintionen for xorshift128+ i V8. Kodesprog: C++. ("random-number-generator.h," 2025)

xorshift128+ har en 128-bit intern tilstand fordelt over to 64-bit variable ("State"(s)), hhv `uint64_t state0` (`s0`) og `uint64_t state1` (`s1`).

Outputtet fra xorshift128+ er faktisk 64-bit, men Number (IEEE 754) bruger kun 52 bits præcision, så `Math.random()` tager produktet af xorshift128+, returnerer de øverste 52 bits som et tal i intervallet [0, 1[ og kasserer de 12 mindst signifikante bits i slutningen.

xorshift128+ bruger state rotation:  $\text{state0} \leftarrow \text{state1}$ ,  $\text{state1} \leftarrow \text{transformeret state0}$

Alt arbejde sker med 64-bit unsigned integers.

Der er 3 serie forbundne XOR-operationer med forskellige shifts, hhv. 23, 17 og 26. Shift-værdierne er valgt for at maksimere bit-spredning og minimere korrelationen mellem outputs. Disse værdier er fundet empirisk gennem omfattende statistik og trial-and-error testing af forskellige kombinationer (Vigna & Elsevier, 2017).

Outputtet fra xorshift128+ er en sum af de to state-værdier, ikke en XOR. Dette står i modsætning til den originale xorshift128, som kun lavede trin af XOR og bitshifts, med operationen til returværdien:  $(w = (w \wedge (w \gg 19)) \wedge (t \wedge (t \gg 8)))$ ; (Marsaglia, 2003)

'+'-operationen i xorshift128+ øger ikke-lineariteten af algoritmen. Addition giver *carry bits*, bits som der skubbes mod venstre ligesom i normal addition, så man mister værdifuld information, hvis man vil forsøge at invitere algoritmen.

Lad os gennemgå xorshift128+'s komplette operation med et eksempel. `0x` er et præfix for at indikere, at det er en hex-værdi (base-16):

Vi har to begyndende states, de 2 64-bit variable. I vores eksempel bruger vi:

- `state0: 0x14cefb416bc7915` (i base-10: 1499346506157422869)
- `state1: 0x16d961281c79ab21` (i base-10: 1646453963684948769)

De to states fås efter 30 iteration af xorshift128+ med de indledende states `0x0000000000000001` og `0x0000000000000002`:

1. `s1 = state0`  
Vi gemmer parameter 1 til en lokal variabel  
`s1 = 0x14cefb416bc7915`
2. `s0 = state1`

Vi gemmer parameter 2 til en lokal variabel  
`s0 = 0x16d961281c79ab21`

3. `state0 = s0`

Vi sætter en ny global `state0` fra `s0` (nu afledt af `state1`)  
`state0 = 0x16d961281c79ab21`

4. `s1 ^= s1 << 23`

Her laver vi en bit shift-operation, der rykker `s1` 23 bits til venstre. Det skal tydeliggøres, at det er den 64-bit binære værdi, der forskydes med 23 pladser, og ikke er hex-værdien, der rykkes 23 pladser. Det er de fire underliggende bits i en byte ( $4 \cdot 16 = 64$ ), der udgør talværdien, der rykkes.

`s1 << 23 = 0xda0b5e3c8a800000`

Vi laver en XOR-operation på bits af det gamle tal og det nye bitforskudte tal. En XOR-operation giver et true output, når et af de to inputs er forskellige (dvs. et ulige antal inputs er true). Ellers giver den false:

Input		Output
s1	<code>s1 &lt;&lt; 23</code>	<code>s1 ^ s1 &lt;&lt; 23</code>
0	0	0
0	1	1
1	0	1
1	1	0

Tab. 1 XOR gate-logik for xorshift128+

`s1 (new) = 0xcec5e1889c3c7915`

5. `s1 ^= s1 >> 17`

Lignende operation, igen på `s1`. Her laver vi en højreforskydning i stedet for venstre, og laver en XOR-operation på den gamle og nye værdi.

`s1 >> 17 = 0x00006762f0c44e1e`  
`s1 (new) = 0xcec586ea6cf8370b`

6. `s1 ^= s0`

XOR-operation mellem `s0`, den gamle state, og vores modificerede `s1`.

`s0 = 0x16d961281c79ab21`  
`s1 (new) = 0xd81ce7c270819c2a`

7. `s1 ^= s0 >> 26`

Bitshift til højre med 26 af `s0`, XOR over `s1`.

`s0 >> 26 = 0x00000005b6584a07`  
`s1 (new) = 0xd81ce7c7c6d9d62d`

8. `state1 = s1`

Vi sætter en ny `state1`-reference til næste iteration  
`state1 = 0xd81ce7c7c6d9d62d`

9. `output = s0 + s1`

Returnerer summen af `s0` og `s1`. Operationen er implicit en modulo( $2^{64}$ ) på grund af integer overflow i C++'s `uint64_t` datatype.

Hvis summen overstiger  $2^{64} - 1$ , wrappes den automatisk rundt til 0. Returværdien er de pseudotilfældige bytes.  
`output = 0xeeef648efe353814e` (i base-10: 17219030420445954382)

Nye state0: `0x16d961281c79ab21`

Nye state1: `0xd81ce7c7c6d9d62d`

Output: `0xeeef648efe353814e`

## STATISTISK ANALYSE AF TILFÆLDIGHED

## E. EKSPERIMENTDESIGN

Vi vil vurdere styrkeerne af tre forskellige PRNG'er, henholdsvis xorshift128+, xorshift128 og en Lineær Kongruentiel Generator (*Linear Congruential Generator*, "LCG"). En LCG er en meget simpel PRNG, som kun laver tre operationer for at beslutte en pseudotilfældig sekvens. Dens ulempe er, at den ikke er særlig stærk og meget let at forudsige. En LCG har en multiplikationsoperation af sin tidligere tilstand og en additionsoperation, der lægges til resultatet, og der derefter vikles tilbage omkring en anden værdi med moduloperatoren (*Random Number Information*, u.d.). Vores LCG vil bruge parametrene fra glibc som bruges i GNU Compiler Collection ("gcc"). Værdierne er 1103515245 for multiplikation, 12345 for addition og  $2^{32}$  for modulus (*Randomness in Computation: Random Number Generators*, 2012).

Testparametre fastsættes sådan: for  $\chi^2$ -testen anvendes  $K = 10$  bins,  $N = 1000$  samples og 2000 gentagelser (kørsler).  $K = 10$  giver 9 frihedsgrader, hvilket er en acceptabel balance af følsomhed og beregningsomkostning.  $N = 1000$  sikrer en tilstrækkelig statistisk styrke med en forventet beregnet frekvens på  $\sim 100 \frac{\text{samples}}{\text{bin}}$ . 2000 gentagelser sikrer en stabil fordeling, der vil mindske de normale statistiske afvigelser, så resultatet kan sammenlignes med den teoretiske  $\chi^2$ -densitet.

For autokorrelationstesten anvendes  $N = 1024$  samples og  $N = 16384$  samples med maksimalt lag  $k = 20$ .  $N = 1024$  giver konfidensgrænser på  $\pm \frac{2}{\sqrt{1024}} = \pm 0,0625$ . Lag op til  $k = 20$  vil afsløre mønstre over en kort periode, uden at udvande grafen ved højere k-værdier. Vi vil analysere både små og store N-værdier, fordi det kan afsløre forskellig bias, der kan forekomme ved forskellige størrelser.

## F. HYPOTESERAMMME

Vores nulhypotese  $H_0$  er, at PRNG-outputtet vil være uafhængig og have en ensartet fordeling under intervallet  $[0, 1]$ .

For  $\chi^2$ -testen skal den beregnede  $\chi^2$ -fordeling matche den teoretiske  $\chi^2$ -fordeling med  $DF = K - 1$ . Hypotesen forkastes, hvis værdier konsekvent afviger fra den kritiske værdi ved  $\alpha = 0,05$ .

For autokorrelationstesten skal koefficienten  $r_k$  ligge inden for  $\pm \frac{2}{\sqrt{N}}$  for alle  $k > 0$ , 95 % af tiden. Hypotesen forkastes, hvis søgerne bryder konfidensgrænsen for ofte.

For xorshift-varianterne (hhv. xorshift128+ og xorshift128) forventes nulhypotesen at blive opfyldt. LCG forventes at fejle begge tests, fordi dens multiplikation og addition skaber en lineær udvikling i modulo-rummet.  $\chi^2$ -testen forventes at fejle, fordi generatoren ikke spredter værdierne jævnt. Den bevæger sig i et lille antal lineære baner gennem  $2^{32}$ -modulo-rummet, som giver bias. Autokorrelationen forventes at fejle, fordi værdierne hænger lineært sammen ( $x_{n+1} = (ax_n + c) \bmod m$ ).

## G. RESULTATER

Resultaterne ses i Bilag 2, 3. og 4. For selv at køre statistikken, hent [statistics/autokorrelation.html](#) og [statistics/chisq.html](#) fra Bilag 1 eller se Bilag 5.2 og 5.3.

## H. ANALYSE OG FORTOLKNING

Med hensyn til autokorrelationstesten ligger fejlprocenten for xorshift-algoritmerne (både den originale og den additionsbaserede) inden for 5 %-tærsklen efter flere kørsler. En lille  $k$ -værdi betyder, at den er begrænset til test af uafhængighed i små intervaller, men ved at justere til kørsler med en højere  $k$ -værdi, opnås samme jævne resultater. Interessant nok, over mange kørsler, virker LCG til at blive forringet over tid. Det skal dog noteres, at algoritmen er fejlbehæftet fra starten, og det bliver blot tydeligt efter mange iterationer. Dette skyldes strukturen i LCG's modulus-aritmetik, hvor fejlene først bliver synlige ved store forskydninger. LCG med modulus  $2^{32}$  har en lav periode i de lave bits, fordi addition af 12345 (et ulige tal) betyder, at bit 0 konsekvent skifter mellem 0 og 1 i et deterministisk XOR-mønster afhængig af multiplikationskonstanten.

Med hensyn til  $\chi^2$ -testen ser vi også ensartede resultater for xorshift128 og xorshift128+. Deres  $\chi^2$ -værdi falder inden for den forventede fordeling over vores serie på 2000 gentagelser. Det betyder, at vi naturligvis vil have afvigende værdier, såsom  $\chi^2 = 27,360$  (se Bilag 1.1, `Math.random()`), men en stor del af værdierne er næsten ligligt fordelt. På den anden side ser vi et meget højreforskudt  $\chi^2$ -resultat for LCG'en. Det forklarer af, at LCG'en ikke fordeler værdierne jævnt i vores bins. Den over- og underrepræsenterer i stedet

bestemte intervaller, fordi af samme grund som den fejler autokorrelationstesten, klumper værdierne sig i bestemte områder mens de mangler i andre. Problemet opstår, fordi følgende værdier ligger på en lineær bane bestemt af rekursionen  $x_{n+1} = (ax_n + c) \text{ mod } m$ .

Vi observerer samme effektivitet i begge tests for både xorshift128+ (gennem `Math.random()`) og xorshift128-implementeringen. Dette er forventet og bekræfter, at tilføjelsen af et additionstrin i xorshift128+ hverken forringer eller forbedrer algoritmens ydeevne i vores benchmark. I stedet tjener det primært til at skjule algoritmens interne tilstand.

xorshift128+ og xorshift128 opfylder nulhypotesen, mens gcc LCG fejler nulhypotesen, fordi den hverken spreder værdierne jævnt eller producerer uafhængige output-par.

## I. BEGRÆNSINGER

$\chi^2$ -testen og autokorrelationstesten opdager kun simple defekter i en PRNG.  $\chi^2$  vurderer om talgeneratorens fordeling af ét output i hele sekvensen af tal afviger fra uniformitet, uden at tage højde til afhængigheder af andre outputs. Vores  $\chi^2$  vurderer kun en-dimensionel fordeling af værdier over intervallet  $[0, 1]$  ved opdeling i bins og sammenligning med den forventede frekvens for uniform fordeling. Man bør vurdere en PRNG over flere dimensioner ved at gruppere sekvenser, således at  $(x_1, x_2, x_3)$  bliver til  $(x_1, \dots, x_d), (x_{d+1}, \dots, x_{2d})$ , og man kan mønsteranalyse dem.

Autokorrelationstesten opdager kun lineær afhængighed mellem  $x_i$  og  $x_{i+k}$ . Ikke-lineære relationer og korrelation gennem flere led af forskelligt compute forbliver skjult.

Testsuiten TestU01 (University of Montreal, u.d.) er en enorm og omfattende testpakke til tilfældige talgeneratorer. Den anvender en lang række tests (SmallCrush, Crush, BigCrush), som kombinerer hundredvis af statistiske analyser af fordeling, afhængighed, dimensionalitet og algebraiske strukturer. Den kan afsløre fejl, der ligger langt uden for både  $\chi^2$  og autokorrelation, og som først bliver synlige, når en RNG udsættes for belastning gennem mange forskellige testmetoder. TestU01 blev anvendt i skabelsen af xorshift128+-algoritmen, hvor man kørte TestU01 på hundredvis af kombinationer af bitshift-værdier, for at fastlægge de bedste parametre. Hver kørsel tog cirka to timer at gennemføre, men xorshift128+ består nu også stort set alle test i testen. (Vigna & Elsevier, 2017)

## KRYPTOANALYSE PÅ XORSHIFT128+

Selvom xorshift128+ har taget skridt imod kryptoanalyse ved at lave et additionstrin til resultatet af xorshift128-kernelogikken, trinnet `output=s0 + s1`, er denne operation ikke tilstrækkelig for at sløre generatorens deterministiske natur. Da alle dens operationer (XOR, bitshifts og addition) er helt definerbare, kan vi bruge en SMT til at genskabe algoritmens interne tilstand. En SMT fungerer som en slags ligningsløsning, der afgør tilfredsstillelsen af dit problem under de givne betingelser og begrænsninger. En SMT sammenlignes med "solve"-funktionen i et CAS-værktøj, men en SMT er ideelt bygget til computerproblemer, mens *solve* bygger på symbolske løsninger ud fra matematiske principper. Vi anvender Microsofts Z3 Theorem Prover ("Z3") (*Z3Prover/Z3: The Z3 Theorem Prover*, u.d.).

Vi anvender Z3 til at løse xorshift128+. Med Z3 kan vi:

- Definere de ukendte variabler
- Tilføje betingelser, der skal opfyldes for at løse problemet ("**constraints**")
- Finde en numerisk løsning til vores problem

I V8 (Chromium/Google Chrome/Node.js) kører `Math.random()` faktisk baglæns. Den opretter en cache pool, der fylder en pulje af størrelse `kCacheSize` ("random-number-generator.h," 2025), som der trækkes værdier fra. Dette gøres af hensyn til ydeevne, så den hurtigt kan hente værdier, og puljen kan genfyldes, mens funktionen er inaktiv. Det betyder dog også, at hvis vi ville løse xorshift128+ på traditionel vis, skal vi bestemme det *foregående tal*, ikke det næste. Vi skal derfor lave en baglæns implementering.

Den forward xorshift128+-algoritme ser sådan ud:

```
uint64_t s1 = state0;
uint64_t s0 = state1;
state0 = s0;
s1 ^= s1 << 23;
s1 ^= s1 >> 17;
s1 ^= s0;
s1 ^= s0 >> 26;
state1 = s1;
return s0 + s1;
```

Fig 5 Den væsentlige del af xorshift128+ i V8. Kodesprog: C++. ("random-number-generator.h," 2025)

Før vi kan genere nogle tal, skal vi først bestemme `state0` og `state1` for at kunne forudsige det næste tal i talfølgen.

Vi kender allerede hele algoritmen fra V8's kildekode, så vi kan lave implementeringen nemt i Python. Vi kan tilføje hele xorshift128+-operationen til Z3 som et constraint, hvor vi tager højde for, at vi kun har 52 bits til løsningen (dvs. mantissen). Hvis vi har nok bits til at finde en løsning (typisk er en sekvens af 5 outputs i træk fra `Math.random()` tilstrækkeligt), bestemmer vi `state0` og `state1`.

Vi skriver vores Z3-løsning i Python:

```
class V8Solver:
    def __init__(self, sekvens: List[float]):
        self.state0, self.state1 = None, None
        self.intern_sekvens = sekvens[::-1] # Vi omvender listen pga. cache-puljen
        self.maske = 0xFFFFFFFFFFFFFF # Anvendes senere

        se_state0, se_state1 = BitVecs("se_state0 se_state1", 64) # Tilføj states til Z3
        t0_ref, t1_ref = se_state0, se_state1
        løser = Solver()

        for i in range(len(self.intern_sekvens)):
            # xorshift128+ - her udfører vi selve algoritmen på vores Z3 variabler
            se_s1 = se_state0
            se_s0 = se_state1
            se_state0 = se_s0
            se_s1 ^= se_s1 << 23
            se_s1 ^= LShR(se_s1, 17)
            se_s1 ^= se_s0
            se_s1 ^= LShR(se_s0, 26)
            se_state1 = se_s1

            # 52-bit mantisse - her sikrer vi, at der kun vurderes på de sidste 52 bits
            original_random = self.intern_sekvens[i]
            mantisse = int(original_random * (1 << 53))

            # Tilføj constraint til Z3 løseren
            løser.add(mantisse == LShR(se_state0, 11))

        # Vi vurderer, om der kan findes en løsning til vores constraints
        if løser.check() != sat:
            return None

        # Her beregner og assigner vores konkrete løsning
        model = løser.model()
        self.state0 = model[t0_ref].as_long()
        self.state1 = model[t1_ref].as_long()
```

Fig 6 State-retrival af xorshift128+ i initialiseringsfasen af V8Solver-klassen. Kodesprog: Python. (se Bilag 1; solver/V8Solver.py, eller Bilag 5.6)

Nu har vi bestemt `state0` og `state1` ud fra xorshift128+ og den sekvens, vi giver modellen. Vi skal nu bestemme det næste (eller *foregående*) tal i sekvensen.

Når vi inverterer xorshift128+, starter vi med `state1` (det vi kender), og arbejder tilbage for at finde den originale `s1` (som var `state0` før transformationen).

I baglænsfunktionen repræsenterer `ps0` den værdi vi rekonstruerer (tidligere `state0`), og `self.state1` er det vi starter med (den nuværende `state1`, som er resultatet af transformationen). Gennem baglænsfunktionen vil `ps0` gradvist blive transformert tilbage til den originale `state0`.

For at invertere denne sekvens skal hver operation vendes til omvendt rækkefølge. XOR-operationen er sig selv invers (dvs.  $a \oplus b = b \oplus a \wedge a \oplus b \oplus b = a$ ), men når der er bitforskydninger involveret, kræver det ekstra behandling. Problemet er, at hvis vi har en XOR-operation som f.eks. `s1 ^= s1 >> 17`, tager vi `s1` og shifter den til højre med 17 bits, vi XOR'er resultatet med den originale `s1`, og overskriver `s1` med den nye værdi. Vi har nu brug for den originale `s1`, men vi har kun resultatet. Den højreshiftede værdi påvirker kun bits af resultatet, mens de øverste 17 bits forbliver uændrede. Vi kan bygge den originale værdi op bit for bit:

```
ps0 = ps0 ^ (ps0 >> 17) ^ (ps0 >> 34) ^ (ps0 >> 51) (en forskydning af de 17 hver gang indtil vi fylder ud)
```

Tilsvarende for `s1 ^= s1 << 23` (et venstre-shift), hvor bits 23-63 påvirkes:

```
ps0 = (ps0 ^ (ps0 << 23) ^ (ps0 << 46)) (en forskydning af de 23, hver gang indtil vi fylder ud)
```

I vores Pythonimplementering bemærkes også en `& self.maske` (hvor `self.maske = 0xFFFFFFFFFFFFFFFFF`). I Python kan integers vokse ubegrænset, men den originale implementering er en `uint64_t` (unsigned 64-bit integer). Operationen her sikrer, at resultatet forbliver et 64-bit tal. Uden masken kunne left-shift operationer producere tal større end 64-bits, hvilket ville ødelægge løsningen. Hele implementeringen kan ses på GitHub i solver/V8Solver.py (Bilag 1) el. Bilag 5.6.

Vi har tilmed også lavet en implementering til SpiderMonkey/Firefox, som minder meget om V8's, men som anvender en forward xorshift128+ i sin næste forudsigelse. Den ligger på GitHub i solver/SpiderMonkeySolver.py (Bilag 1) el. Bilag 5.7.

#### REAL-WORLD UDNYTTELSE AF OUTPUT-FORUDSIGELSE

Vi vil eftervise metodens anvendelighed i en online roulettesimulator. Miljøet er sat op således:

- Backend: node.js (v24.7.0)
- Frontend: browser (enhver)

Backenden kører gennem node.js på localhost på port 3000. Når <http://localhost:3000/> tilgås, leveres index.html. Der er ét endpoint til backenden, `/spin` (se Bilag 1; web\_exploit/server.js el. Bilag 5.8). node.js er en serversidet runtime bygget på V8, og bruger derfor samme `Math.random()` implementering som vi løser i V8Solver.py.

Man kan sende en GET request til `/spin`, eventuelt med en bet-parameter (bet: int [1, 36]). Backenden beregner et resultat med `Math.random()`, et vindtal (`Math.floor(result * 36) + 1`), hjulets rotation med (`result * 360`), om om vi taber eller vinder vores bet (`bet === number`).

Som angriber har vi mulige angrebsvektorer (*attack vectors*) at vælge imellem. Vi vurderer de tre, `number`, `rotation` og `won`, på deres informationsindhold og forholdet til `Math.random()`, for at bestemme, hvilken vi bør anvende til vores angreb. Vi bruger feltet med den højeste præcision og det mindste informationstab.

Den bedste angrebsvektor er `rotation`, da den giver en direkte float-værdi. `rotation` mister en smule information ved afrundningsstøjen fra multiplikationstrinnet, men størstedelen af mantissen består; `bits ≈ 52`.

Den næstbedste vil være `number`, som ligger i intervallet [1, 36]. Vi kan derfor udlede, at det sandsynligvis beregnes med operationen `Math.floor(Math.random() * 36) + 1`. Floor-operationen gør til gengæld, at vi mister meget af vores information. Vi står tilbage med ca.  $\log_2(36) \approx 5,17$  bits ud af de fuldkomne 52 mantisse-bits. Den vil stadig være brugbar, men kræver meget større beregning og mange flere samples i sekvensen for at kunne få en korrekt forudsigelse.

Sidst er `won`. Den er også afledt af den originale `Math.random()`, men den fortæller os kun, om det bet vi har suppleret, rammer et bestemt tal; vi får dermed kun én binær oplysning pr. spin, og den oplysning kræver derudover, at vi indfører et bet.

Vores angrebsmetode er som følger:

- Indsaml bits gennem spins uden et bet
- Få solver til at løse states til sekvensforudsigelse
- Bekræft korrekt forudsigelse
- Udførelse på rouletten

Vi skal først indsamle vores data. Vi laver seks calls til `/spin` gennem Developer Tools-konsollen på roulette-websiden:

```
const rotations = [];
let forventet_forudsagte_rotation = 0;
for (let i = 0; i < 6; i++) {
  const { rotation } = await fetch('/spin').then(result => result.json());
  if (i < 5) rotations.push(rotation);
  else forventet_forudsagte_rotation = rotation;
}
console.log(`Sekvens: ${rotations.join(',')}`);
console.log(`Næste rotation i sekvensen (til validering af resultat): ${forventet_forudsagte_rotation}`)
```

Fig 7 Kode-snippet til indsamling af 6 pseudo-tilfældige tal fra roulette-simulatorens backend. Kodesprog: JavaScript.

Når vi kører programmet på websiden, får vi denne fem lange sekvens, samt det næste tal til at bekræfte vores solvers løsning:

```
Sekvens:
239.0349524765034, 176.2443549996164, 314.95871733959734, 314.1772910202531, 152.4193564892859
Næste rotation i sekvensen (til validering af resultat): 217.39300471851814
```

For at vores solver vil acceptere et tal, der ligger uden for `Math.random()`'s `[0, 1[`, skal vi lave nogle ændringer til vores kildekode, så vi har en brugerdefinerbar multiplikationskonstant:

```
def __init__(self, sekvens: List[float]):
    def __init__(self, sekvens: List[float], transform: float = 1.0, bits_tilgængelige: int = 52):
        [...]
        original_random = self.intern_sekvens[i]
        original_random = self.intern_sekvens[i] / self.transform
        [...]
        løser.add(mantisze == LShR(se_state0, 11))
        shift_amount = 53 - self.bits_tilgængelige
        mantisze_masked = mantisze >> shift_amount
        state_masked = LShR(LShR(se_state0, 11), shift_amount)
        løser.add(mantisze_masked == state_masked)
        [...]
```

Fig 8 Ændring af kode i V8Solver for at understøtte transform og tab af præcision. Kodesprog: Python (diff). (se Bilag 1; solver/V8Solver.py, eller Bilag 4.6)

Vi kan nu tildele en multiplikationskonstant til V8Solver som parameteren `transform`. Vi ved derudover, at hvis vi ganger op eller ned, risikerer vi at miste noget af matissen. Vi har derfor introduceret endnu en parameter, `bits_tilgængelige`, som vi justerer, hvis vi ved, at vi mister nogle af de laveste bits til vores transformation.

Vi anvender sekvensen i vores solver med browser engine V8, multiplikationskonstanten (`transform`) 360, og alle 52 bits tilgængelige. Det skal noteres, at det ikke er altid, at solveren kan lykkedes med alle 52 bits - i så fald skal bits reduceres. Det er bedst at have så mange bits tilgængeligt som muligt, da der kan opstå falske positiver (flere mulige korrekte states) ved at bruge færre bits præcision.

Se figuren for den praktiske udførelse:

```

zoidberg@ConnectON:~/Documents/Uddannelse/Gymnasium/3.G/SOP/SOP/solver
zoidberg@ConnectON:~/Documents/Uddannelse/Gymnasium/3.G/SOP/SOP/web_exploit — node
zoidberg@ConnectON:~/Documents/Uddannelse/Gymnasium/3.G/SOP/SOP/solver$ ls
prng_example solver statistics web_exploit
zoidberg@ConnectON:~/Documents/Uddannelse/Gymnasium/3.G/SOP/SOP$ cd solver
zoidberg@ConnectON:~/Documents/Uddannelse/Gymnasium/3.G/SOP/SOP/solver$ python xorshift128pSolver.py 239.0349524765034,176.2443549996164,314.95871733959734,
314.1772910202531,152.4193564892859 engine=v8 transform=360.0 bits=52
Sekvens: [239.0349524765034, 176.2443549996164, 314.95871733959734, 314.1772910202531, 152.4193564892859]
Browser Engine: v8
Transform: 360.0
Bits: 52

Forudsagte tal:
[217.39300471851814, 1.1073429914517785, 144.58119334694592, 161.76437254301743, 89.96820679314996, 30.79881254071224, 221.82425239437194, 174.5327548408608
, 333.80473354721937, 244.4140535774851]
zoidberg@ConnectON:~/Documents/Uddannelse/Gymnasium/3.G/SOP/SOP/solver$ 

```

Fig. 9 Roulette-forudsigelse kørt med vores generede sekvens. engine=v8, transform=360.0, bits=52. (se Bilag 1; solver/xorshift128pSolver, eller Bilag 5.5)

Vi kan se på Fig. 9, at det næste forudsagte tal i vores talserie stemmer overens med `forventet_forudsagte_rotation` (den næste rotation i sekvensen) = 217.39300471851814. Dette indikerer, at vi har gendannet den korrekte tilstand.

Vi regner med at vindertallet beregnes som  $v = \lfloor r \cdot 36 \rfloor + 1$ , hvor  $r = \text{Math.random}()$ . Vi har nu også bekræftet, at  $rot = r \cdot 360$ . Det må derfor betyde, at  $v = \lfloor \frac{rot}{360} \cdot 36 \rfloor + 1 \Leftrightarrow v = \lfloor \frac{rot_{forudsagt}}{10} \rfloor + 1$ . Vi kan dermed genskabe de næste mange vindertal ud fra vores forudsigelser. Se tabel 2.

Forudsagt rotation	Vindertal
1.1073429914517785	1
144.58119334694592	15
161.76437254301743	17
89.96820679314996	19

Tab. 2 Forudsagte rotationsgrader omregnet til næste vindertal i rouletten

Notér, at vi ikke har inkluderet rotationen 217.xxx, fordi dens cyklus allerede er opbrugt. Vi kører de næste fire forudsagte tal på roulettespillet, og vinder alle sammen. Se figuren herunder.

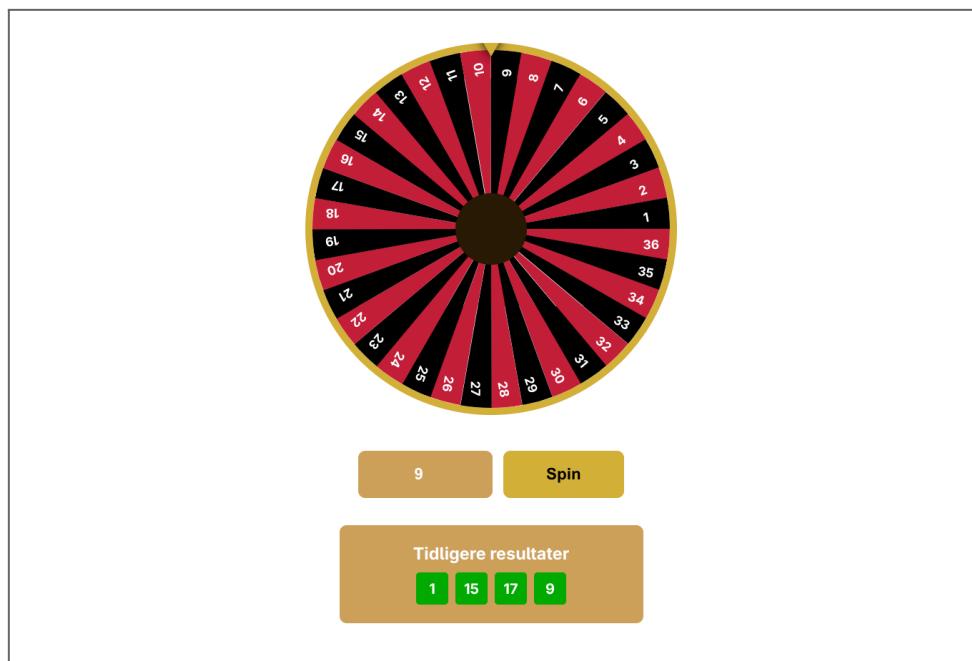


Fig. 10 Bets 1, 15, 17, og 9 korrekt gættet på roulettensimuleringen. (se Bilag 1; web\_exploit/, eller Bilag 5.8 og Bilag 5.9)

Fig. 10 viser, at vores solver succesfuldt har rekonstrueret den interne tilstand i xorshift128+, hvilket tillader forudsigelse af de næste rotationsgrader, og dermed vindertallene i roulettesimuleringen. Dette bekræfter, at determinismen i en PRNG kan udnyttes, når vi har tilstrækkelig information tilgængelig.

## DISKUSSION OG PERSPEKTIVERING

### J. RESULTATER AF FORSØG

Forsøget har vist, at det med 5 observerende outputs er muligt at genskabe den interne tilstand i PRNG'en xorshift128+. Dette gælder også for den oprindelige xorshift128, som endda er nemmere at knække, da den ikke bruger addition, men kun XOR-operationer.

xorshift128+ er ikke designet til at være kryptografisk sikker (som en CSPRNG), men formår at være hurtig og kan effektivt dække hele alle  $2^{52}$  mantisser, som tal i JavaScript kan repræsentere, fordi algoritmens fulde periode er  $2^{128} - 1$ .

Vores forsøg krævede mange observerende outputs med en rimelig præcision (ikke kun vindertallet, men næsten fuld `Math.random()`-præcision) for at opfyldes i rimelig tid. Hvis vi kun havde vindertallet, ville det stadig være muligt at genskabe tilstanden, men kræve meget mere beregningstid og optimering af vores metode.

Derudover var angrebet en succes, fordi rouletten anvender en "naiv" tilgang til at generere sit næste træk. I virkeligheden bør programmet anvende node.js' `crypto.getRandomValues()` eller `crypto.randomBytes()` (*Crypto | Node.js Documentation*, u.d.), som begge anvender operativsystemets CSPRNG (f.eks. /dev/urandom på Linux). Eksperimentet illustrerer en central pointé: PRNG'er som xorshift128+ er fundamentalt forudsigelige, når deres output kan observeres, og bør derfor aldrig anvendes i sikkerhedskritiske sammenhænge.

Sidst skal det noteres, at vi har arbejdet i et kontrolleret miljø, ikke med et real-world angreb. Den primære fordel ved dette er, at vi ikke har nogle forstyrrelser i vores sekvens fra andre brugere, der genererede deres egne tilfældige tal. I praksis ville vi være nødt til at finde et angrebspunkt, der kan generere en tilstrækkelig serie af kontinuerlige bits uden forstyrrelser, der er uden for vores viden og kontrol.

## K. TALGENERATORS' IMPLIKATION FOR SIKKERHED

Det er afgørende, at udviklere træffer de rigtige beslutninger, når de skriver kode. PRNG'er og CSPRNG'er har hver deres anvendelsesområde, og fordele og ulemper ved hver af dem bør altid vurderes, når de integreres i en protokol. I klient-idet infrastruktur, hvor en bruger vil have direkte adgang til outputtet fra en tilfældig talgenerator - f.eks. ved valg af et vindertal i et roulettespil, er en PRNG fundamentalt usikker. På serversiden er `Math.random()` stadig sårbar, bare sværere at observere.

I tilfælde, hvor sikkerhed og reproducerbarhed *ikke* er et primært hensyn, vil `Math.random()` eller en anden PRNG ofte være tilstrækkelig. Dette kan være tilfældet for spil-AI'er, der skal træffe tilfældige beslutninger og er afhængige af hurtige operationer, eller seed-baseret terrængenerering som i Minecraft, eller til brug i simuleringer (f.eks. oprette partikler), hvor styrken af den tilfældige talgenerator ikke har en betydelig indflydelse på resultatet.

Til gengæld, hvis operationen finder sted i en kritisk del af en kodebase - for eksempel i autentificering (såsom at lave cookies) eller ved valg af tilfældigt tal med sikkerhedsmæssige konsekvenser eller anden betydelig værdi - bør en CSPRNG altid foretrækkes. Hvis `Math.random()` anvendes til at generere session tokens, kan en angriber observere tidligere tokens, rekonstruere PRNG-tilstanden, og forudsige fremtidige tokens, og dermed autentificere sig som andre brugere. De fleste browsere har siden 2011 implementeret Web Crypto API, der kan generere kryptografisk sikre tilfældige tal. (Mozilla, u.d.-b) Node.js har crypto-modulet til sine serverside operationer.

## L. UDVIDELSE AF ANGREB TIL ANDRE APPLIKATIONER

Selvom dette forsøg fokuserede på et simpelt roulettespil, er de samme principper direkte anvendelige i mange andre applikationer. Og selvom der i dag findes mange API'er til korrekt håndtering af kryptografi, er det i sidste ende op til udvikleren at implementere den bedste praksis.

Det kunne for eksempel være muligt at opnå en fordel i multiplayer-spil, hvis der er huller i implementeringen af tilfældige beslutninger, såsom spiller- og fjendens spawn-placeringer eller loot box-resultater.

Derudover, hvis en applikation for eksempel seeder en PRNG (eller CSPRNG) med usikker entropi (f.eks. timestamp, bruger-ID), kan angriberne bestemme seed-værdi. Hvis en PRNG seedes med `Date.now()` (millisekunder siden 1970), kan en angriber lave et brute-force på kort tid, for der er i alt ca.  $1,8 \cdot 10^{12}$  muligheder, og  $(1,8 \cdot 10^{12}) \ll (2^{64} \approx 1,8 \cdot 10^{19})$ . En angriber kan initialisere sin egen talgenerator med samme seed, og generere en identisk sekvens. Denne angrebsform er for eksempel blevet anvendt til at gendanne en adgangskode og få adgang til et låst bitcoin-wallet (Joe Grand, 2024).

#### KONKLUSION

Denne opgave har demonstreret xorshift128+-algoritmens fundamentale sårbarhed gennem kryptoanalyse. Ved anvendelse af Z3 Theorem Prover blev algoritmens 128-bit interne tilstand rekonstrueret, og angrebet blev anvendt imod et praktisk eksempel med succes. Ud fra en sekvens med 5 observerende outputs, kunne vi forudsige efterfølgende outputs. Den praktiske udnyttelse i roulettesimuleringen bekræftede metodens effektivitet ved at kunne korrekt gætte alle vindertallene.

De statistiske tests ( $\chi^2$ -test og autokorrelation) viste, at xorshift128+ producerer ensartede og tilsyneladende uafhængige outputs. Algoritmens bitshift-metode og store  $2^{128} - 1$ -periode sikrer en tilstrækkelig spredning samt minimal korrelation for at fungere godt i en ikke-sikkerhedskritisk sammenhæng. De statistiske tests efterviste også effektiviteten af xorshift128 - kernen på xorshift128+ - og viste, at additionstrinnet i xorshift128+ primært fungerer til at skjule den interne tilstand. Konsekvent fejlede den simple PRNG LCG, begge tests.

Selvom additionsstrinnet i xorshift128+-algoritmen forbedrede algoritmens ikke-linearitet, fjernede det ikke fuldstændigt algoritmens deterministiske fundament. XOR-operationernes vendbarhed, kombineret med tilstrækkelig observeret præcision, gjorde symbolisk løsning mulig i praksis.

xorshift128+ opfylder sit designformål. At være hurtig og generere statistisk pålidelige pseudotilfældige tal til f.eks. grafik, simuleringer og processuel generering. Den er dog grundlæggende ikke egnet til kryptografiske formål, hvor en ordentlig CSPRNG bør foretrækkes. Sikkerheden bestemmes altid af det svageste led. Seed-kvalitet, valg af algoritme og implementeringskontekst skal vurderes samlet. En CSPRNG med 256-bit entropi forringes til 32-bit sikkerhed med dårlig seedning. Tilsvarende bliver en PRNG med perfekt statistik forudsigtelig, når man observerer output.

## REFERENCER

- Barker, E. (2025). *Recommendation for random bit generator (RBG) constructions*. <https://doi.org/10.6028/nist.sp.800-90c>
- Brandon. (2024, August 23). *What is a password hash: Cryptography Basics*. Tuta. Hentet December 5, 2025, fra <https://tuta.com/blog/what-is-a-password-hash>
- Cao, Y., Liu, W., Qin, L., Liu, B., Chen, S., Ye, J., Xia, X., & Wang, C. (2022). Entropy sources based on silicon chips: true random number generator and physical unclonable function. *Entropy*, 24(11), 1566. <https://doi.org/10.3390/e24111566>
- Crypto | Node.js Documentation*. (u.d.). Hentet December 10, 2025, fra <https://nodejs.org/api/crypto.html>
- De Mooij, J. [jandemooij]. (2015, November 30). *Testing Math.random(): Crushing the browser*. Jan De Mooij Blog. Hentet December 2, 2025, fra <https://jandemooij.nl/blog/testing-math-random-crushing-the-browser/>
- European Information Technologies Certification Academy [EITCA]. (2014, June 12). *What are the key differences between True Random Number Generators (TRNGs), Pseudorandom Number Generators (PRNGs), and Cryptographically Secure Pseudorandom Number Generators (CSPRNGs)?* EITCA. Hentet December 2, 2025, fra <https://eitca.org/cybersecurity/eitc-is-ccf-classical-cryptography-fundamentals/stream-ciphers/stream-ciphers-random-numbers-and-the-one-time-pad/examination-review-stream-ciphers-random-numbers-and-the-one-time-pad/what-are-the-key-differences-between-true-random-number-generators-trngs-pseudorandom-number-generators-prngs-and-cryptographically-secure-pseudorandom-number-generators-csprngs/>
- GeeksforGeeks. (2025, July 23). *Asymmetric key cryptography*. GeeksforGeeks. Hentet December 5, 2025, fra <https://www.geeksforgeeks.org/computer-networks/asymmetric-key-cryptography/>
- Greendow. (u.d.). *GitHub - greendow/Hash-DRBG: The C implementation of Hash\_DRBG in NIST SP 800-90A Rev.1 is provided. Header files and library files of OpenSSL 1.1.1 or higher version are needed while compiling and linking*. GitHub. <https://github.com/greendow/Hash-DRBG>
- Guo, Y. [hashseed]. (2015, December 17). *There's Math.random(), and then there's Math.random()*. V8 Blog. Hentet December 9, 2025, fra <https://v8.dev/blog/math-random>
- H. G., B. (u.d.). *Chi-Square Goodness of Fit Test*. Stat Trek. Hentet December 8, 2025, fra <https://stattrek.com/chi-square-test/goodness-of-fit>
- Hash functions. (u.d.). Devtool Online. Hentet December 5, 2025, fra <https://devtool.online/hash>
- Hill, J. (2018). NIST Special Publication 800-90B Comments. *UL*. <https://www.untruth.org/~josh/sp80090b/UL%20SP800-90B-final%20comments%20v1.4%2020181126.pdf>  
Kommentar til 800-90B
- Joe Grand. (2024, May 28). *I hacked time to recover \$3 million from a Bitcoin software wallet* [Video]. YouTube. Hentet December 10, 2025, fra <https://www.youtube.com/watch?v=o5IySpAkThg>
- Judge. (2024, September 22). *The "Web of Trust" principle or how PGP works*. HackerNoon. Hentet December 5, 2025, fra <https://hackernoon.com/the-web-of-trust-principle-or-how-pgp-works>
- Leonardo, A. (u.d.). *Chi-Square Statistic: How to Calculate It / Distribution*. Statistics How To. Hentet December 8, 2025, fra <https://www.statisticshowto.com/probability-and-statistics/chi-square/>
- Marsaglia, G. (2003). Xorshift RNGs. *Journal of Statistical Software*, 8(14). <https://doi.org/10.18637/jss.v008.i14>
- Mol, G. (2020, December 29). *Session security*. Beagle Security. Hentet December 5, 2025, fra <https://beaglesecurity.com/blog/article/session-security.html>
- Mozilla. (u.d.-a). *Number - JavaScript*. MDN. Hentet December 9, 2025, fra [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Number#number\\_encoding](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number#number_encoding)
- Mozilla. (u.d.-b). *Web Crypto API - Web APIs*. MDN. Hentet December 10, 2025, fra [https://developer.mozilla.org/en-US/docs/Web/API/Web\\_Crypto\\_API#browser\\_compatibility](https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API#browser_compatibility)
- Nguyen, C. [chaunn3502]. (2021, September 5). *Autocorrelation and Partial Autocorrelation in Time Series Data*. Medium. Hentet December 8, 2025, fra <https://medium.com/@chaunn3502/autocorrelation-and-partial-autocorrelation-in-time-series-data-1dfdb683e48e>
- Random number generation - ArchWiki. (u.d.). [https://wiki.archlinux.org/title/Random\\_number\\_generation](https://wiki.archlinux.org/title/Random_number_generation)
- Random Number Information. (u.d.). Random Number Generator. Hentet December 9, 2025, fra <https://www.randomnumbergenerator.com/generation.asp>
- Randomness in Computation: Random Number Generators. (2012, October). [Slide show; PowerPoint]. Carnegie Mellon University. <https://www.cs.cmu.edu/~15110-f12/Unit09PtA-handout.pdf>
- random-number-generator.h. (2025). [Software]. In v8. Google. <https://source.chromium.org/chromium/chromium/src/+main:v8/src/base/utils/random-number-generator.h>
- Smith, S., & Community, K. (2025, October 7). *CSPRNG - KERI Concept*. vLEI.wiki. Hentet December 5, 2025, fra <https://www.vlei.wiki/concept/csprng>
- Turan, M. S., Barker, E., Kelsey, J., McKay, K. A., Baish, M. L., & Boyle, M. (2018). *Recommendation for the entropy sources used for random bit generation*. <https://doi.org/10.6028/nist.sp.800-90b>
- University of Montreal. (u.d.). *Empirical testing of random number generators*. TesU01. Hentet December 9, 2025, fra <https://simul.iro.umontreal.ca/testu01/tu01.html>
- Vigna, S. & Elsevier. (2017). Further scramblings of Marsaglia's xorshift generators. *Elsevier*. <https://www.sciencedirect.com/science/article/pii/S0377042716305301?via%3Dihub>

- WeakRandom.h. (2025). [Software]. In *WebKit*. Apple. <https://github.com/WebKit/WebKit/blob/main/Source/WTF/wtf/WeakRandom.h>
- XorShift128PlusRNG.h. (2018). [Software]. In *SpiderMonkey*. Mozilla. <https://searchfox.org/firefox-main/source/mfbt/XorShift128PlusRNG.h#65>
- Z3Prover/z3: The Z3 Theorem Prover. (u.d.). GitHub. Hentet December 9, 2025, fra <https://github.com/z3prover/z3>
- Zaiontz, C. (u.d.). Autocorrelation Function. Real Statistics Using Excel. Hentet December 8, 2025, fra <https://real-statistics.com/time-series-analysis/stochastic-processes/autocorrelation-function/>

## BILAG

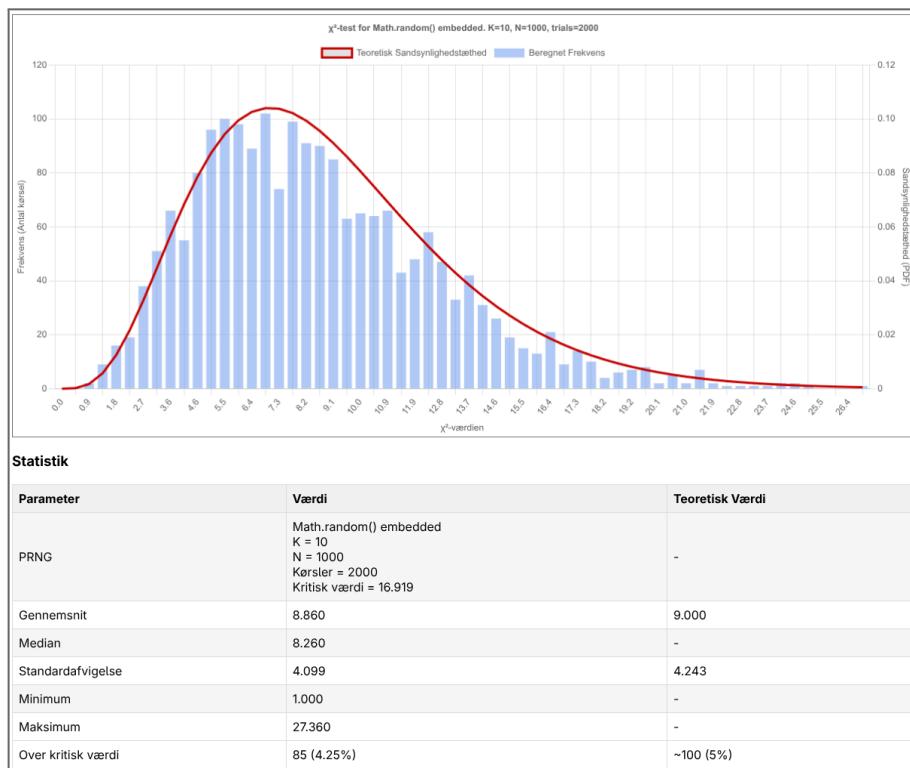
### 1. KILDEKODE

Alt koden bag dette projekt kan tilgås på:

<https://github.com/asger-finding/sop>

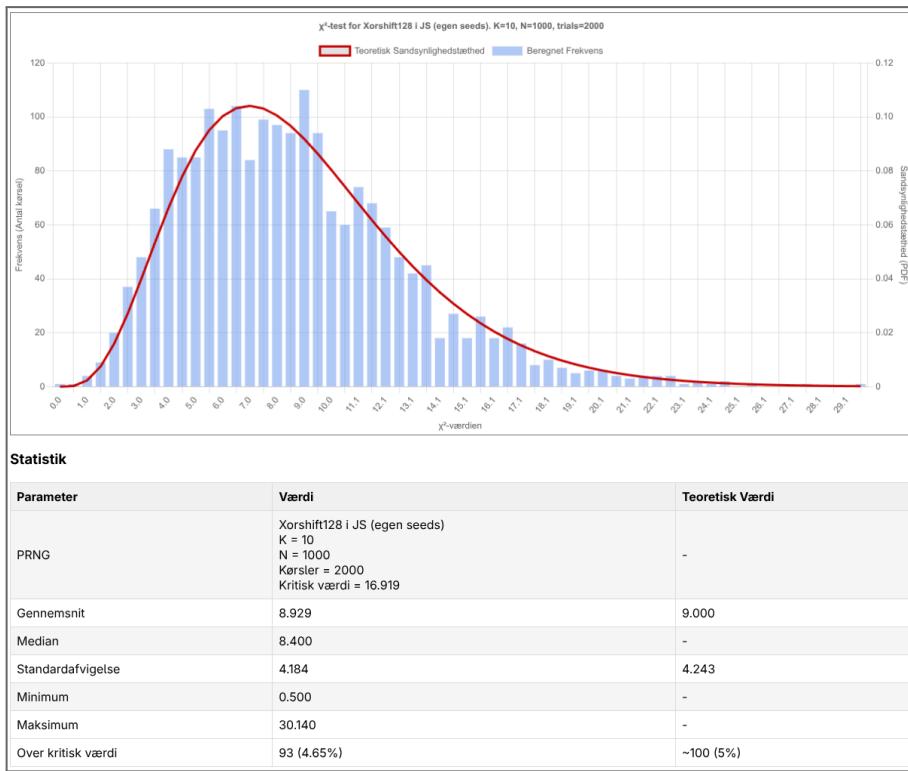
Koden er også vedhæftet som Bilag 5.xx.

### 2. $\chi^2, N = 1024$

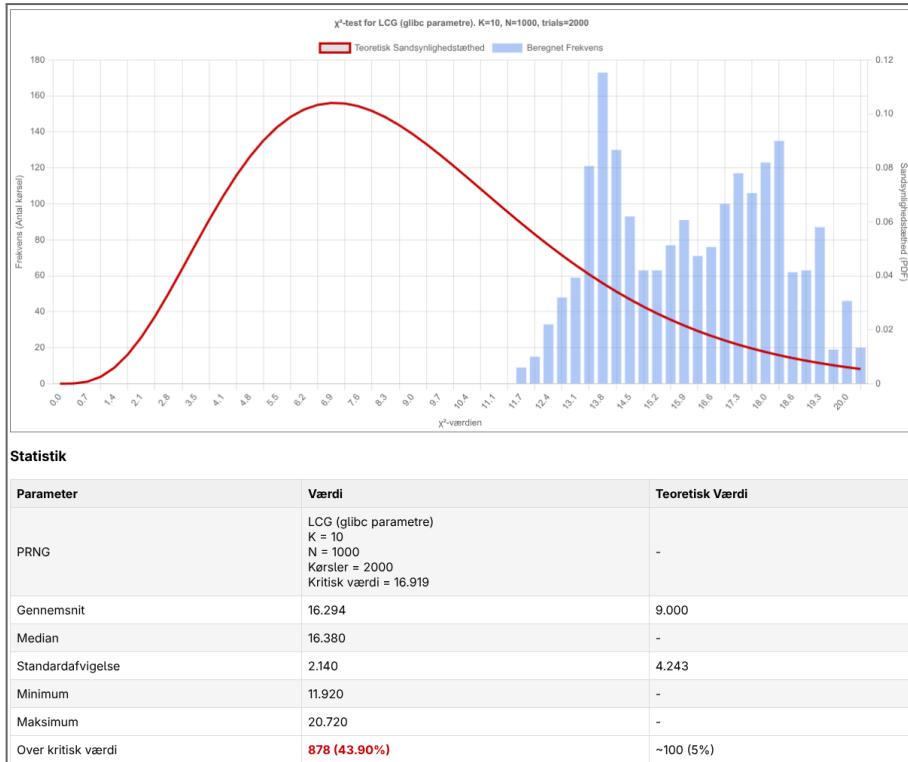


Bilag. 2.1 Math.random()  $\chi^2$ -testresultat ved  $N = 1000, k = 10$ , kørsler = 2000,  $\chi^2_{kritisk} = 16,919$

Autogenereret vha. Bilag 1; statistics/chisq.html el. Bilag 4.2.

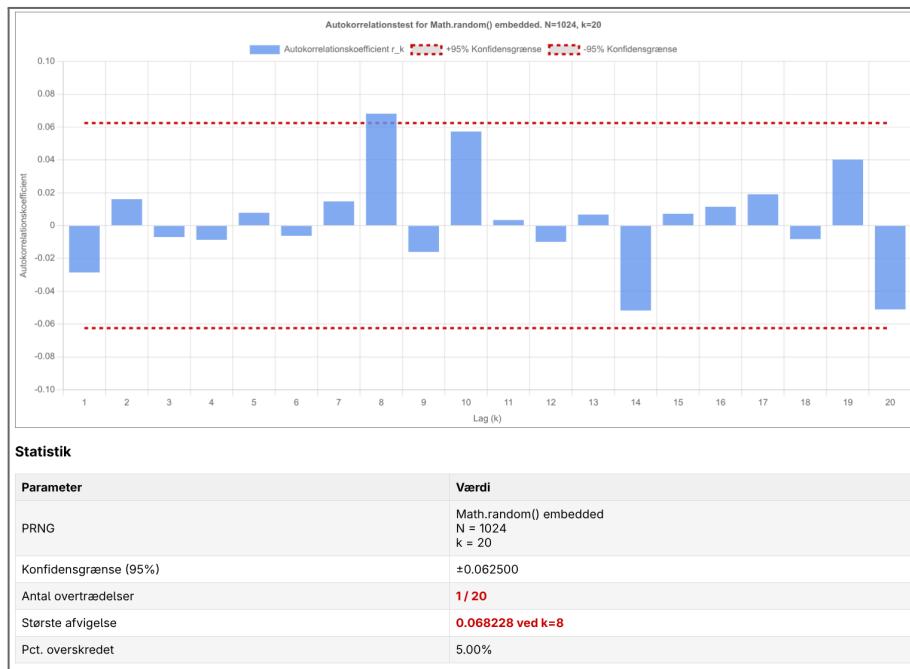


Bilag. 2.2 xorshift128  $\chi^2$ -testresultat ved  $N = 1000$ ,  $k = 10$ , kørsler = 2000,  $\chi^2_{kritisk} = 16,919$ . Autogenereret vha. Bilag 1; statistics/chisquare.html el. Bilag 4.2.

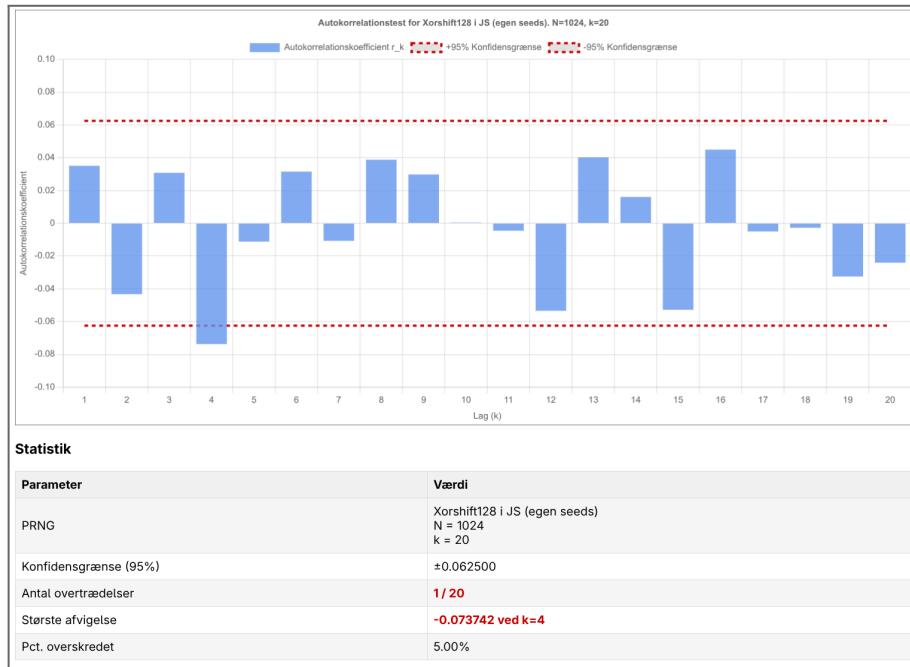


Bilag. 2.3 gcc LGC  $\chi^2$ -testresultat ved  $N = 1000$ ,  $k = 10$ , kørsler = 2000,  $\chi^2_{kritisk} = 16,919$ . Autogenereret vha. Bilag 1; statistics/chisquare.html el. Bilag 4.2.

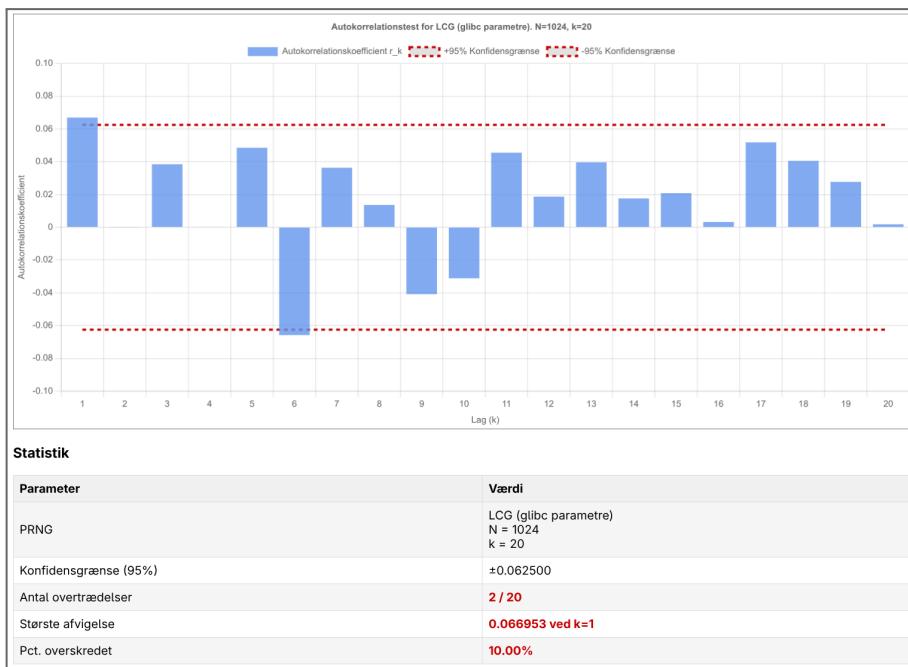
### 3. AUTOKORRELATION, $N = 1024$



Bilag. 3.1 Math.random() autokorrelation ved  $N = 1024, k = 20$ . Autogenereret vha. Bilag 1; statistics/autokorrelation.html el. Bilag 4.3.

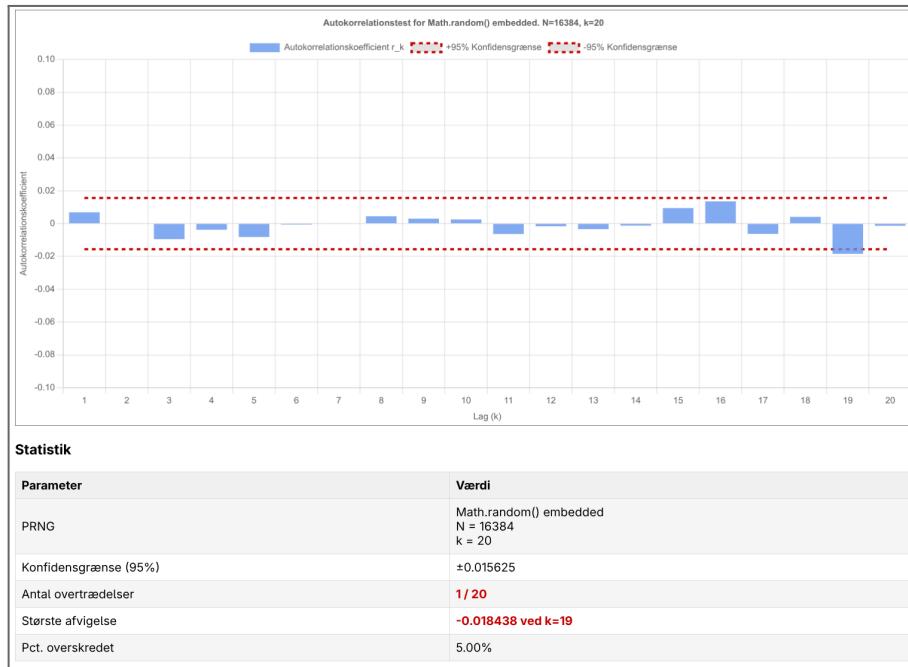


Bilag. 3.2 xorshift128 autokorrelation ved  $N = 1024, k = 20$ . Autogenereret vha. Bilag 1; statistics/autokorrelation.html el. Bilag 4.3.



Bilag. 3.3 gcc LCG autokorrelation ved  $N = 1024$ ,  $k = 20$ . Autogenereret vha. Bilag 1; [statistics/autokorrelation.html](#) el. Bilag 4.3.

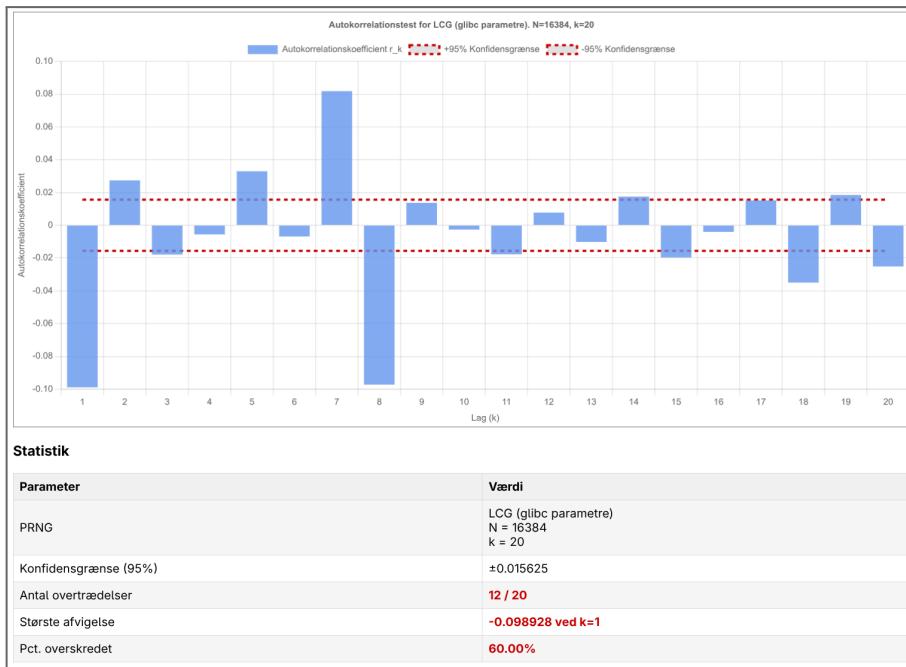
#### 4. AUTOKORRELATION, $N = 16\,384$



Bilag. 4.1 Math.random() autokorrelation ved  $N = 16\,384, k = 20$  Autogenereret vha. Bilag 1; statistics/autokorrelation.html el. Bilag 4.3.



Bilag. 4.2 xorshift128 autokorrelation ved  $N = 16\,384, k = 20$  Autogenereret vha. Bilag 1; statistics/autokorrelation.html el. Bilag 4.3.



Bilag. 4.3 gcc LGC autokorrelation ved  $N = 16\ 384, k = 20$  Autogenereret vha. Bilag 1;  
[statistics/autokorrelation.html](#) el. Bilag 4.3.

## 5. KODEBILAG

```
def xorshift128plus(state0, state1):
    # Python har ikke 64 bit begrænsning på tal,
    # så vi maskerer
    MASK_64 = (1 << 64) - 1

    print("xorshift128+")
    print(f"Initial state0: {state0:#018x} (base-10: {state0})")
    print(f"Initial state1: {state1:#018x} (base-10: {state1})")
    print()

    s1 = state0
    print(f"Trin 1: s1 = state0")
    print(f"        s1 = {s1:#018x}")
    print()

    s0 = state1
    print(f"Trin 2: s0 = state1")
    print(f"        s0 = {s0:#018x}")
    print()

    state0 = s0
    print(f"Trin 3: state0 = s0")
    print(f"        state0 = {state0:#018x}")
    print()

    shifted = (s1 << 23) & MASK_64
```

```
s1 = s1 ^ shifted
s1 &= MASK_64
print(f"Trin 4: s1 ^= s1 << 23")
print(f"          s1 << 23 = {shifted:#018x}")
print(f"          s1 (new) = {s1:#018x}")
print()

shifted = s1 >> 17
s1 = s1 ^ shifted
s1 &= MASK_64
print(f"Trin 5: s1 ^= s1 >> 17")
print(f"          s1 >> 17 = {shifted:#018x}")
print(f"          s1 (new) = {s1:#018x}")
print()

s1 = s1 ^ s0
s1 &= MASK_64
print(f"Trin 6: s1 ^= s0")
print(f"          s0      = {s0:#018x}")
print(f"          s1 (new) = {s1:#018x}")
print()

shifted = s0 >> 26
s1 = s1 ^ shifted
s1 &= MASK_64
print(f"Trin 7: s1 ^= s0 >> 26")
print(f"          s0 >> 26 = {shifted:#018x}")
print(f"          s1 (new) = {s1:#018x}")
print()

state1 = s1
print(f"Trin 8: state1 = s1")
print(f"          state1 = {state1:#018x}")
print()

output = (s0 + s1) & MASK_64
print(f"Trin 9: output = s0 + s1")
print(f"          s0      = {s0:#018x}")
print(f"          s1      = {s1:#018x}")
print(f"          output = {output:#018x} (base-10: {output})")
print()

print(f"Endelig state0: {state0:#018x}")
print(f"Endelig state1: {state1:#018x}")
print(f"Output:         {output:#018x}")
print("\n")

return output, state0, state1
```

```
# Kørsel med simple state-værdier
print("Simple startværdier\n")
output1, new_state0_1, new_state1_1 = xorshift128plus(
    state0=0x0000000000000001,
    state1=0x0000000000000002
)

print("\n\n")

# Kørsel med store state-værdier
print("Større startværdier\n")
output2, new_state0_2, new_state1_2 = xorshift128plus(
    state0=0XBADA55B00B1E5101,
    state1=0xC0FFEE1CEC0FFEE5
)

print("\n\n")

# Kørsel med flere iterationer
iterations = 30
print(f"{iterations} iterationer\n")
state0 = 0x0000000000000001
state1 = 0x0000000000000002

for iteration in range(iterations):
    print(f"\n{'='*50}")
    print(f"Iteration {iteration + 1}")
    print(f"{'='*50}\n")
    output, state0, state1 = xorshift128plus(state0, state1)
```

Bilag 5.1 prng\_example/xorshift128p.py

```
<html>

<head>
    <meta charset="utf-8">
    <title>Chi2 ( $\chi^2$ )-test for PRNG</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js@4.5.1/dist/chart.umd.min.js"></script>
<style>
    body {
        font-family: system-ui;
        margin: 12px
    }

    #main {
        width: 1200px;
    }

    .controls {
        margin-bottom: 8px;
        display: flex;
        justify-content: space-between;
        align-items: center;
        flex-wrap: wrap;
    }

    .control-group {
        display: flex;
        align-items: center;
        gap: 8px;
    }

    label {
        display: inline-block;
    }

    #plot {
        border: 1px solid rgb(118, 118, 118);
        display: block;
        margin-top: 10px
    }

    #results {
        margin-top: 20px;
    }

    #results table {
        width: 100%;
        border-collapse: collapse;
        margin-top: 10px;
    }
</style>
```

```
}

#results th, #results td {
    border: 1px solid #ddd;
    padding: 8px;
    text-align: left;
}

#results th {
    background-color: #f2f2f2;
    font-weight: bold;
}

#results tr:nth-child(even) {
    background-color: #f9f9f9;
}

.violation {
    color: #c00;
    font-weight: bold;
}
</style>

<script type="module">
    class XorShift128Plus {
        constructor(seed1 = 1, seed2 = 2) {
            this.state0 = this.to64bit(seed1);
            this.state1 = this.to64bit(seed2);
        }

        to64bit(value) {
            if (typeof value === 'bigint') {
                return value & 0xFFFFFFFFFFFFFFFFFFn;
            }
            if (Array.isArray(value)) {
                return (BigInt(value[0]) << 32n) | BigInt(value[1]);
            }
            return BigInt(value) & 0xFFFFFFFFFFFFFFFFFFn;
        }

        next() {
            let s1 = this.state0;
            let s0 = this.state1;

            this.state0 = s0;

            s1 ^= (s1 << 23n) & 0xFFFFFFFFFFFFFFFFFFn;
            s1 ^= s1 >> 17n;
            s1 ^= s0;
            s1 ^= s0 >> 26n;
        }
    }
</script>
```

```
        this.state1 = s1 & 0xFFFFFFFFFFFFFFFn;

        const output = (s0 + s1) & 0xFFFFFFFFFFFFFFFn;
        return output;
    }

    random() {
        const output = this.next();
        const top52 = output >> 12n;
        return Number(top52) / 4503599627370496;
    }
}

class XorShift128 {
    constructor(seed1 = 1, seed2 = 2, seed3 = 3, seed4 = 4) {
        this.x = new Uint32Array(4);
        this.x[0] = seed1 >>> 0;
        this.x[1] = seed2 >>> 0;
        this.x[2] = seed3 >>> 0;
        this.x[3] = seed4 >>> 0;
    }

    next() {
        let t = this.x[3];
        let s = this.x[0];

        this.x[3] = this.x[2];
        this.x[2] = this.x[1];
        this.x[1] = s;

        t ^= t << 11;
        t ^= t >>> 8;
        this.x[0] = t ^ s ^ (s >>> 19);

        return this.x[0];
    }

    random() {
        return this.next() / 0x100000000;
    }
}

class LCG {
    constructor(seed = 1) {
        this.a = 1103515245;
        this.c = 12345;
        this.m = 2**32;
        this.state = seed >>> 0;
    }
}
```

```

next() {
    this.state = ((this.a * this.state) + this.c) >>> 0;
    return this.state;
}

random() {
    const value = this.next();
    return value / this.m;
}
}

window.xorshift128plus = new XorShift128Plus(1n, 2n);
window.xorshift128 = new XorShift128(1, 2, 3, 4);
window.LCG = new LCG(1);
</script>

<script>
(() => {
    const g = 7;
    const C = [0.9999999999980993, 676.5203681218851, -1259.1392167224028,
771.32342877765313, -176.61502916214059, 12.507343278686905, -0.13857109526572012,
9.9843695780195716e-6, 1.5056327351493116e-7];
    const gamma = (z) => {
        if (z < 0.5) return Math.PI / (Math.sin(Math.PI * z) * gamma(1 - z));
        else {
            z -= 1;

            let x = C[0];
            for (let i = 1; i < g + 2; i++) x += C[i] / (z + i);

            const t = z + g + 0.5;
            return Math.sqrt(2 * Math.PI) * Math.pow(t, (z + 0.5)) *
Math.exp(-t) * x;
        }
    }

    const chi2Densitet = (værdi, frihedsgrader) => {
        if (værdi < 0) return 0;
        const halv = frihedsgrader / 2;
        return 1 / (2 ** halv * gamma(halv)) * værdi ** (halv - 1) *
Math.exp(-værdi / 2);
    }

    const beregnChi2 = (antalSamples, antalBins, prngFunction) => {
        const tællere = new Array(antalBins).fill(0);
        for (let sample = 0; sample < antalSamples; sample++)
tællere[Math.floor(prngFunction() * antalBins)]++;

        const forventet = antalSamples / antalBins;
    }
}

```

```

let chi2Sum = 0;
for (let bin = 0; bin < antalBins; bin++) {
    const forskel = tællere[bin] - forventet;
    chi2Sum += (forskelse * forskel) / forventet;
}
return chi2Sum;
}

const lavHistogram = (værdier, antalBins, maksimum) => {
    const histogram = new Array(antalBins).fill(0);
    const binBredde = maksimum / antalBins;
    for (const værdi of værdier) {
        const binIndeks = Math.min(antalBins - 1, Math.floor(værdi / binBredde));
        histogram[binIndeks]++;
    }
    return histogram;
}

window.toggleCustomArea = () => {
    const select = document.getElementById("prngSelect");
    const customArea = document.getElementById("customPrngArea");
    if (select.value === 'custom') {
        customArea.style.display = 'block';
    } else {
        customArea.style.display = 'none';
    }
}

let graf;
window.run = () => {
    if (graf) graf.destroy();

    const antalBins = Number(document.getElementById("K").value);
    const antalSamples = Number(document.getElementById("N").value);
    const antalGentagelser = Number(document.getElementById("trials").value);
    const kritiskVærdi = Number(document.getElementById("criticalValue").value);
    const prngType = document.getElementById("prngSelect").value;
    let prngNavn = document.getElementById("prngSelect").options[document.getElementById("prngSelect").selectedIndex].text;

    let prngFunction;

    switch (prngType) {
        case 'mathRandom':
            prngFunction = Math.random;
            break;
    }
}

```

```

        case 'xorshift128plus':
            prngFunction = () => window.xorshift128plus.random();
            break;
        case 'xorshift128':
            prngFunction = () => window.xorshift128.random();
            break;
        case 'lcg':
            prngFunction = () => window.LCG.random();
            break;
        case 'custom':
            if (typeof window.customPRNG === 'function') {
                prngFunction = window.customPRNG;
                prngNavn = 'Custom PRNG';
            } else {
                const customCode =
document.getElementById("customCode").value;
                try {
                    eval(`function() { ${customCode} }()`);
                    if (typeof window.customPRNG === 'function') {
                        prngFunction = window.customPRNG;
                        prngNavn = 'Custom PRNG';
                    } else {
                        alert("customPRNG() er ikke defineret. Bruger
Math.random() som fallback.");
                        prngFunction = Math.random;
                    }
                } catch (e) {
                    alert(`Fejl ved evaluering af Custom PRNG kode:
${e.message}`);
                    return;
                }
            }
            break;
        default:
            prngFunction = Math.random;
    }

    const chi2Værdier = [];
    for (let gentagelse = 0; gentagelse < antalGentagelser; gentagelse++)
chi2Værdier.push(beregnChi2(antalSamples, antalBins, prngFunction));

    const frihedsgrader = antalBins - 1;
    const maksimalChi2 = Math.max(...chi2Værdier);
    const histogramBins = 60;
    const histogram = lavHistogram(chi2Værdier, histogramBins,
maksimalChi2);
    const binLabels = [...Array(histogramBins)].map((_, i) =>
        (i * maksimalChi2 / histogramBins).toFixed(1)
    );

```

```
const teoretiskFordeling = [...Array(histogramBins)].map((_, i) => {
    const xVærdi = i * maksimalChi2 / histogramBins;
    return chi2Densitet(xVærdi, frihedsgrader);
});

graf = new Chart(document.getElementById('plot'), {
    type: 'bar',
    data: {
        labels: binLabels,
        datasets: [
            {
                label: 'Teoretisk Sandsynlighedstæthed',
                data: teoretiskFordeling,
                type: 'line',
                pointRadius: 0,
                borderColor: 'rgba(200, 0, 0, 1.0)',
                yAxisID: 'y1'
            }, {
                label: 'Beregnet Frekvens',
                data: histogram,
                backgroundColor: 'rgba(100, 150, 240, 0.5)',
                borderColor: 'rgba(100, 150, 240, 1)',
                yAxisID: 'y0'
            }
        ]
    },
    options: {
        plugins: {
            title: {
                display: true,
                text: `χ²-test for ${prngNavn}. K=${antalBins}, N=${antalSamples}, trials=${antalGentagelser}`
            },
            legend: {
                display: true,
                position: 'top'
            },
            tooltip: {
                mode: 'index',
                intersect: false
            }
        },
        scales: {
            x: {
                title: {
                    display: true,
                    text: 'χ²-værdien'
                }
            },
            y0: {
                type: 'linear',
                position: 'left',
            }
        }
    }
});
```

```

        title: {
            display: true,
            text: 'Frekvens (Antal kørsel)'
        },
        beginAtZero: true
    },
    y1: {
        type: 'linear',
        position: 'right',
        grid: {
            drawOnChartArea: false
        },
        title: {
            display: true,
            text: 'Sandsynlighedstæthed (PDF)'
        },
        beginAtZero: true
    }
}
}

const gennemsnit = chi2Værdier.reduce((a, b) => a + b, 0) /
chi2Værdier.length;
const sorteret = [...chi2Værdier].sort((a, b) => a - b);
const median = sorteret[Math.floor(sorteret.length / 2)];
const min = Math.min(...chi2Værdier);
const max = Math.max(...chi2Værdier);
const teoretiskGennemsnit = frihedsgrader;
const teoretiskVarians = 2 * frihedsgrader;
const varians = chi2Værdier.reduce((acc, val) => acc + (val -
gennemsnit) ** 2, 0) / chi2Værdier.length;
const standardafvigelse = Math.sqrt(varians);
const antalOverKritisk = chi2Værdier.filter(v => v >
kritiskVærdi).length;
const procentOverKritisk = (antalOverKritisk / chi2Værdier.length * 100).toFixed(2);
console.log(antalOverKritisk, chi2Værdier.length)

document.getElementById('results').innerHTML =
<h3>Statistik</h3>
<table border="1" cellpadding="6" cellspacing="0" style="border-collapse: collapse; margin-top: 10px;">
<tr>
    <th>Parameter</th>
    <th>Værdi</th>
    <th>Teoretisk Værdi</th>
</tr>
<tr>
    <td>PRNG</td>

```

```

<td colspan="1" style="white-space: pre-wrap;">">${prngNavn}\nK      =    ${antalBins}\nN      =    ${antalSamples}\nKørsler      =
${antalGentagelser}\nKritisk værdi = ${kritiskVærdi}</td>
<td>-</td>
</tr>
<tr>
    <td>Gennemsnit</td>
    <td>${gennemsnit.toFixed(3)}</td>
    <td>${teoretiskGennemsnit.toFixed(3)}</td>
</tr>
<tr>
    <td>Median</td>
    <td>${median.toFixed(3)}</td>
    <td>-</td>
</tr>
<tr>
    <td>Standardafvigelse</td>
    <td>${standardafvigelse.toFixed(3)}</td>
    <td>${Math.sqrt(teoretiskVarians).toFixed(3)}</td>
</tr>
<tr>
    <td>Minimum</td>
    <td>${min.toFixed(3)}</td>
    <td>-</td>
</tr>
<tr>
    <td>Maksimum</td>
    <td>${max.toFixed(3)}</td>
    <td>-</td>
</tr>
<tr>
    <td>Over kritisk værdi</td>
    <td class="${antalOverKritisk > antalGentagelser * 0.05 ? 'violation' : ''}">${antalOverKritisk} (${procentOverKritisk}%)</td>
    <td>~${(antalGentagelser * 0.05).toFixed(0)} (5%)</td>
</tr>
</table>
`;
}

document.addEventListener('DOMContentLoaded', () => {
    window.toggleCustomArea();
    window.run();
});
})();
</script>
</head>

<body>

```

```

<div id="main">
    <div class="controls">
        <div class="control-group">
            <label for="K">K</label><input id="K" type="number" value="10">
        </div>
        <div class="control-group">
            <label for="N">N</label><input id="N" type="number" value="1000">
        </div>
        <div class="control-group">
            <label for="trials">Kørsler</label><input id="trials" type="number" value="2000">
        </div>
        <div class="control-group">
            <label for="criticalValue"><a href="https://www.chisquaretable.net/" target="_blank">Kritisk værdi</a></label><input id="criticalValue" type="number" value="16.919">
        </div>
        <div class="control-group">
            <label for="prngSelect">PRNG</label>
            <select id="prngSelect" onchange="window.toggleCustomArea()">
                <option value="mathRandom" selected>Math.random() embedded</option>
                <option value="xorshift128plus">Xorshift128+ i JS (egen seeds)</option>
                <option value="xorshift128">Xorshift128 i JS (egen seeds)</option>
                <option value="lcg">LCG (glibc parametre)</option>
                <option value="custom">Custom</option>
            </select>
        </div>
        <button id="run" onclick="window.run()">Kør</button>
    </div>

    <div id="customPrngArea" style="display:none; margin-top: 8px; width: 100%;>
        <label for="customCode">Custom PRNG-kode</label>
        <textarea id="customCode" rows="4" cols="100">
window.customPRNG = function() {
    // Returner et tal mellem 0 og 1.
    // TODO: jeg har ikke nogen vedholdig state for nyt run
    return Math.random();
}
        </textarea>
    </div>

    <canvas id="plot" width="900" height="420"></canvas>

    <div id="results"></div>

    <div id="footer">
        <p>Tabelværdier for kritiske  $\chi^2$  værdier for højre hale fås ved <a href="https://www.chisquaretable.net/">

```

```
target="_blank">www.chisquaretable.net</a> (f.eks.  
<code>16.919</code> ved <code>α=P=0,05</code>  
      (5%) og <code>DF=9</code>)</p>  
    </div>  
  </div>  
</body>  
  
</html>
```

Bilag 5.2 statistics/chisquaretable.html

```
<html>

<head>
    <meta charset="utf-8">
    <title>Autokorrelationstest for PRNG</title>
    <script src="https://cdn.jsdelivr.net/npm/chart.js@4.5.1/dist/chart.umd.min.js"></script>
<style>
    body {
        font-family: system-ui;
        margin: 12px
    }

    #main {
        width: 1200px;
    }

    .controls {
        margin-bottom: 8px;
        display: flex;
        justify-content: space-between;
        align-items: center;
        flex-wrap: wrap;
    }

    .control-group {
        display: flex;
        align-items: center;
        gap: 8px;
    }

    label {
        display: inline-block;
    }

    #plot {
        border: 1px solid rgb(118, 118, 118);
        display: block;
        margin-top: 10px;
        width: 900px;
        height: 420px;
    }

    #results {
        margin-top: 20px;
    }

    #results table {
        width: 100%;
```

```
border-collapse: collapse;
margin-top: 10px;
}

#results th, #results td {
    border: 1px solid #ddd;
    padding: 8px;
    text-align: left;
}

#results th {
    background-color: #f2f2f2;
    font-weight: bold;
}

#results tr:nth-child(even) {
    background-color: #f9f9f9;
}

.violation {
    color: #c00;
    font-weight: bold;
}

```

```
</style>

<script type="module">
    class XorShift128Plus {
        constructor(seed1 = 1, seed2 = 2) {
            this.state0 = this.to64bit(seed1);
            this.state1 = this.to64bit(seed2);
        }

        to64bit(value) {
            if (typeof value === 'bigint') {
                return value & 0xFFFFFFFFFFFFFFFFFFn;
            }
            if (Array.isArray(value)) {
                return (BigInt(value[0]) << 32n) | BigInt(value[1]);
            }
            return BigInt(value) & 0xFFFFFFFFFFFFFFFFFFn;
        }

        next() {
            let s1 = this.state0;
            let s0 = this.state1;

            this.state0 = s0;

            s1 ^= (s1 << 23n) & 0xFFFFFFFFFFFFFFFFFFn;
            s1 ^= s1 >> 17n;
        }
    }
</script>
```

```
s1 ^= s0;
s1 ^= s0 >> 26n;

this.state1 = s1 & 0xFFFFFFFFFFFFFFFn;

const output = (s0 + s1) & 0xFFFFFFFFFFFFFFFn;
return output;
}

random() {
    const output = this.next();
    const top52 = output >> 12n;
    return Number(top52) / 4503599627370496;
}
}

class XorShift128 {
    constructor(seed1 = 1, seed2 = 2, seed3 = 3, seed4 = 4) {
        this.x = new Uint32Array(4);
        this.x[0] = seed1 >>> 0;
        this.x[1] = seed2 >>> 0;
        this.x[2] = seed3 >>> 0;
        this.x[3] = seed4 >>> 0;
    }

    next() {
        let t = this.x[3];
        let s = this.x[0];

        this.x[3] = this.x[2];
        this.x[2] = this.x[1];
        this.x[1] = s;

        t ^= t << 11;
        t ^= t >>> 8;
        this.x[0] = t ^ s ^ (s >>> 19);

        return this.x[0];
    }

    random() {
        return this.next() / 0x100000000;
    }
}

class LCG {
    constructor(seed = 1) {
        this.a = 1103515245;
        this.c = 12345;
        this.m = 2**32;
    }
}
```

```
        this.state = seed >>> 0;
    }

    next() {
        this.state = ((this.a * this.state) + this.c) >>> 0;
        return this.state;
    }

    random() {
        const value = this.next();
        return value / this.m;
    }
}

window.xorshift128plus = new XorShift128Plus(1n, 2n);
window.xorshift128 = new XorShift128(1, 2, 3, 4);
window.LCG = new LCG(1);
</script>

<script type="module">
    const beregnAutokorrelation = (sekvens, k) => {
        const N = sekvens.length;
        if (k >= N - 1) return 0;

        const sum = sekvens.reduce((acc, val) => acc + val, 0);
        const gennemsnit = sum / N;

        let tæller = 0;
        let nævner = 0;

        for (let i = 0; i < N - k; i++) {
            tæller += (sekvens[i] - gennemsnit) * (sekvens[i + k] - gennemsnit);
        }

        for (let i = 0; i < N; i++) {
            nævner += (sekvens[i] - gennemsnit) ** 2;
        }

        if (nævner === 0) return 0;

        return tæller / nævner;
    }

    const genererSekvens = (antalSamples, prngType) => {
        const sekvens = new Array(antalSamples);
        let prngFunction;

        switch (prngType) {
            case 'mathRandom':
                prngFunction = Math.random;
            case 'xorshift128plus':
                prngFunction = window.xorshift128plus;
            case 'xorshift128':
                prngFunction = window.xorshift128;
            case 'LCG':
                prngFunction = window.LCG;
        }

        for (let i = 0; i < antalSamples; i++) {
            sekvens[i] = prngFunction();
        }

        return sekvens;
    };
</script>
```

```
        break;
    case 'xorshift128plus':
        prngFunction = () => window.xorshift128plus.random();
        break;
    case 'xorshift128':
        prngFunction = () => window.xorshift128.random();
        break;
    case 'lcg':
        prngFunction = () => window.LCG.random();
        break;
    case 'custom':
        if (typeof window.customPRNG === 'function') {
            prngFunction = window.customPRNG;
        } else {
            alert("customPRNG() er ikke defineret. Bruger Math.random() som fallback.");
            prngFunction = Math.random;
        }
        break;
    default:
        prngFunction = Math.random;
    }

    for (let i = 0; i < antalSamples; i++) {
        sekvens[i] = prngFunction();
    }
    return sekvens;
}

window.toggleCustomArea = () => {
    const select = document.getElementById("prngSelect");
    const customArea = document.getElementById("customPrngArea");
    if (select.value === 'custom') {
        customArea.style.display = 'block';
    } else {
        customArea.style.display = 'none';
    }
}

let graf;
window.run = () => {
    if (graf) graf.destroy();

    const antalSamples = Number(document.getElementById("N").value);
    const maxLag = Number(document.getElementById("maxLag").value);
    const prngType = document.getElementById("prngSelect").value;
                                let prngNavn =
document.getElementById("prngSelect").options[document.getElementById("prngSelect").selectedIndex].text;
```

```
if (prngType === 'custom') {
    const customCode = document.getElementById("customCode").value;
    try {
        eval(`function() { ${customCode} }()`);
        prngNavn = 'Custom PRNG';
    } catch (e) {
        alert(`Fejl ved evaluering af Custom PRNG kode: ${e.message}`);
        return;
    }
}

const sekvens = genererSekvens(antalSamples, prngType);
const autokorrelationer = [];
const lagLabels = [];

for (let k = 1; k <= maxLag; k++) {
    autokorrelationer.push(beregnAutokorrelation(sekvens, k));
    lagLabels.push(k);
}

const konfidensGrænse = 1 / Math.sqrt(antalSamples) * 2;
const øvreGrænse = new Array(maxLag).fill(konfidensGrænse);
const nedreGrænse = new Array(maxLag).fill(-konfidensGrænse);

graf = new Chart(document.getElementById('plot'), {
    type: 'bar',
    data: {
        labels: lagLabels,
        datasets: [
            {
                label: 'Autokorrelationskoefficient r_k',
                data: autokorrelationer,
                backgroundColor: 'rgba(100, 150, 240, 0.8)',
                borderColor: 'rgba(100, 150, 240, 1)',
                type: 'bar'
            },
            {
                label: '+95% Konfidensgrænse',
                data: øvreGrænse,
                type: 'line',
                borderColor: 'rgba(200, 0, 0, 1.0)',
                borderDash: [5, 5],
                pointRadius: 0,
                fill: false
            },
            {
                label: '-95% Konfidensgrænse',
                data: nedreGrænse,
                type: 'line',
                borderColor: 'rgba(200, 0, 0, 1.0)',
                borderDash: [5, 5],
                pointRadius: 0,
                fill: false
            }
        ]
    }
})
```

```
        },
      },
      options: {
        plugins: {
          title: {
            display: true,
            text: `Autokorrelationstest for ${prngNavn}.
N=${antalSamples}, k=${maxLag}`
          },
          legend: {
            display: true,
            position: 'top'
          },
          tooltip: {
            mode: 'index',
            intersect: false
          }
        },
        scales: {
          x: {
            title: {
              display: true,
              text: 'Lag (k)'
            }
          },
          y: {
            title: {
              display: true,
              text: 'Autokorrelationskoefficient'
            },
            min: -0.1,
            max: 0.1
          }
        }
      }
    );
  });

let overskridninger = 0;
let maxOverskridning = 0;
let maxOverskridningLag = 0;

for (let i = 0; i < autokorrelationer.length; i++) {
  const r_k = autokorrelationer[i];
  const overskridet = Math.abs(r_k) > konfidensGrænse;
  if (overskridet) {
    overskridninger++;
    if (Math.abs(r_k) > Math.abs(maxOverskridning)) {
      maxOverskridning = r_k;
      maxOverskridningLag = i + 1;
    }
  }
}
```

```
        }

    }

    let overskredetPct = (overskridninger / autokorrelationer.length) * 100;

    document.getElementById('results').innerHTML =
        <h3>Statistik</h3>
        <table>
            <tr>
                <th>Parameter</th>
                <th>Værdi</th>
            </tr>
            <tr>
                <td>PRNG</td>
                <td style="white-space: pre-wrap;">${prngNavn}\nN =
${antalSamples}\nk = ${maxLag}</td>
            </tr>
            <tr>
                <td>Konfidensgrænse (95%)</td>
                <td>±${konfidensGrænse.toFixed(6)}</td>
            </tr>
            <tr>
                <td>Antal overtrædelser</td>
                <td class="${overskridninger > 0 ? 'violation' :
''}">${overskridninger} / ${maxLag}</td>
            </tr>
            <tr>
                <td>Største afvigelse</td>
                <td class="${overskridninger > 0 ? 'violation' :
''}">${maxOverskridning.toFixed(6)} ved k=${maxOverskridningLag}</td>
            </tr>
            <tr>
                <td>Pct. overskredet</td>
                <td class="${overskredetPct > 5 ? 'violation' :
''}">${overskredetPct.toFixed(2)}%</td>
            </tr>
        </table>
    ;
}

document.addEventListener('DOMContentLoaded', () => {
    window.toggleCustomArea();
    window.run();
});
</script>
</head>

<body>
    <div id="main">
        <div class="controls">
```

```

<div class="control-group">
    <label for="N">N</label><input id="N" type="number" value="1024">
</div>
<div class="control-group">
    <label for="maxLag">Maks Lag (k)</label><input id="maxLag" type="number" value="20" min="1">
</div>
<div class="control-group">
    <label for="prngSelect">PRNG</label>
    <select id="prngSelect" onchange="window.toggleCustomArea()">
        <option value="mathRandom" selected>Math.random() embedded</option>
        <option value="xorshift128plus">Xorshift128+ i JS (egen seeds)</option>
        <option value="xorshift128">Xorshift128 i JS (egen seeds)</option>
        <option value="lcg">LCG (glibc parametre)</option>
        <option value="custom">Custom</option>
    </select>
</div>
<button id="run" onclick="window.run()">Kør</button>
</div>

<div id="customPrngArea" style="display:none; margin-top: 8px; width: 100%;>
    <label for="customCode">Custom PRNG-kode</label>
    <textarea id="customCode" rows="4" cols="100">
window.customPRNG = function() {
    // Returner et tal mellem 0 og 1.
    // TODO: jeg har ikke nogen vedholdig state for nyt run
    return Math.random();
}
    </textarea>
</div>

<canvas id="plot" width="900" height="420"></canvas>

<div id="results"></div>

<div id="footer">
    <p>
        Hvis søjlerne overskridet de stipede røde linjer,
        indikerer det en signifikant autokorrelation (afhængighed) ved det
        pågældende lag (k).
    </p>
</div>
</div>
</body>

</html>

```

Bilag 5.3 statistics/autokorrelation.html

```
z3-solver==4.15.4.0
```

Bilag 5.4 solver/requirements.txt

```
"""
USAGE

1. Installer z3: pip install z3-solver

2. I din browser, åben et konsolvindue (Developer Tools)
og indtast og submit: copy(Array.from({ length: 5 }, Math.random).join(","));
Det er fem værdier vi anvender til forudsigelsen

3. Passer værdierne til dette script:
$ python xorshift128pSolver.py <tal1,tal2,tal3,...> [engine=v8|spidermonkey]
[transform=1.0] [bits=52]

Hvis vi har en successful løsning, printes værdierne i et array.
"""

import sys
from z3 import *

from V8Solver import V8Solver
from SpiderMonkeySolver import SpiderMonkeySolver

RANDOM_NUMBERS_TO_GENERATE = 10

def main():
    if len(sys.argv) < 2:
        print("Brug: python xorshift128pSolver.py <tal1,tal2,tal3,...>")
        print("[engine=v8|spidermonkey] [transform=1.0] [bits=52]")
        sys.exit(1)

    tal_input = sys.argv[1]

    # Parse optional parameters
    engine = "v8"
    transform = 1.0
    bits = 52

    for arg in sys.argv[2:]:
        if arg.startswith("engine="):
            engine = arg.split("=", 1)[1]
        elif arg.startswith("transform="):
            try:
                transform = float(arg.split("=", 1)[1])
            except ValueError:
                print("Fejl: transform skal være et gyldigt decimaltal")
                sys.exit(1)
        elif arg.startswith("bits="):
            try:
                bits = int(arg.split("=", 1)[1])
            except ValueError:
                print("Fejl: bits skal være et gyldigt heltal")
                sys.exit(1)
```

```

        except ValueError:
            print("Fejl: bits skal være et gyldigt heltal")
            sys.exit(1)

    try:
        sekvens = [float(x.strip()) for x in tal_input.split(',')]
    except ValueError:
        print("Fejl: Alle inputværdier skal være gyldige decimaltal adskilt af
kommaer.")
        sys.exit(1)

    print("Sekvens: %s" % sekvens)
    print("Browser Engine: %s" % engine)
    print("Transform: %s" % transform)
    print("Bits: %s" % bits)

    if engine == "v8":
        forudsiger = V8Solver(sekvens, transform, bits)
    elif engine == "spidermonkey":
        forudsiger = SpiderMonkeySolver(sekvens)
    else:
        print(f"Fejl: Ukendt engine \'{engine}\'. Skal være \'v8\' eller
\'spidermonkey\'")
        sys.exit(1)

    fremtidige_tal = []
    for _ in range(RANDOM_NUMBERS_TO_GENERATE):
        next = forudsiger.forudsig_næste()
        fremtidige_tal.append(next)

    if None in fremtidige_tal:
        print("Kunne ikke forudsige sekvens")
        sys.exit(0)

    print("\nForudsagte tal:")
    print(fremtidige_tal)

if __name__ == "__main__":
    main()

```

Bilag 5.5 solver/xorshift128pSolver.py

```

from z3 import *
import struct
from typing import List, Optional

class V8Solver:
    def __init__(self, sekvens: List[float], transform: float = 1.0, bits_tilgængelige: int = 52):
        self.state0, self.state1 = None, None
        self.intern_sekvens = sekvens[::-1]
        self.maske = 0xFFFFFFFFFFFFFF
        self.transform = transform
        self.bits_tilgængelige = bits_tilgængelige

        se_state0, se_state1 = BitVecs("se_state0 se_state1", 64)
        t0_ref, t1_ref = se_state0, se_state1
        løser = Solver()

        for i in range(len(self.intern_sekvens)):
            se_s1 = se_state0
            se_s0 = se_state1
            se_state0 = se_s0
            se_s1 ^= se_s1 << 23
            se_s1 ^= LShR(se_s1, 17)
            se_s1 ^= se_s0
            se_s1 ^= LShR(se_s0, 26)
            se_state1 = se_s1

            original_random = self.intern_sekvens[i] / self.transform
            mantisse = int(original_random * (1 << 53))

            shift_amount = 53 - self.bits_tilgængelige
            mantisse_masked = mantisse >> shift_amount
            state_masked = LShR(LShR(se_state0, 11), shift_amount)

            løser.add(mantisse_masked == state_masked)

        if løser.check() != sat:
            return None

        model = løser.model()
        self.state0 = model[t0_ref].as_long()
        self.state1 = model[t1_ref].as_long()

    def __init__(self, sekvens: List[float]):
        self.state0, self.state1 = None, None
        self.intern_sekvens = sekvens[::-1]
        self.maske = 0xFFFFFFFFFFFFFF

        se_state0, se_state1 = BitVecs("se_state0 se_state1", 64)

```

```

t0_ref, t1_ref = se_state0, se_state1
løser = Solver()

for i in range(len(self.intern_sekvens)):
    se_s1 = se_state0
    se_s0 = se_state1
    se_state0 = se_s0
    se_s1 ^= se_s1 << 23
    se_s1 ^= LShR(se_s1, 17)
    se_s1 ^= se_s0
    se_s1 ^= LShR(se_s0, 26)
    se_state1 = se_s1

    original_random = self.intern_sekvens[i]
    mantisse = int(original_random * (1 << 53))

    løser.add(mantisse == LShR(se_state0, 11))

if løser.check() != sat:
    return None

model = løser.model()
self.state0 = model[t0_ref].as_long()
self.state1 = model[t1_ref].as_long()

def forudsig_næste(self) -> Optional[float]:
    if self.state0 is None or self.state1 is None:
        return None
    resultat = self.xorshift128p_baglæns()
    return ((resultat >> 11) / (2**53)) * self.transform

def xorshift128p_baglæns(self):
    resultat = self.state0
    ps1 = self.state0
    ps0 = self.state1 ^ (self.state0 >> 26)
    ps0 = ps0 ^ self.state0
    ps0 = ps0 ^ (ps0 >> 17) ^ (ps0 >> 34) ^ (ps0 >> 51) & self.maske
    ps0 = (ps0 ^ (ps0 << 23) ^ (ps0 << 46)) & self.maske
    self.state0, self.state1 = ps0, ps1
    return resultat

```

Bilag 5.6 solver/V8Solver.py

```

from z3 import *
import struct
from typing import List, Optional

class SpiderMonkeySolver:
    def __init__(self, sekvens: List[float]):
        self.state0, self.state1 = None, None
        self.intern_sekvens = sekvens
        self.maske = 0xFFFFFFFFFFFFFFFFF

        se_state0, se_state1 = BitVecs("se_state0 se_state1", 64)
        t0_ref, t1_ref = se_state0, se_state1

        løsner = Solver()

        for i in range(len(self.intern_sekvens)):
            se_s1 = se_state0
            se_s0 = se_state1
            se_state0 = se_s0
            se_s1 ^= se_s1 << 23
            se_s1 ^= LShR(se_s1, 17)
            se_s1 ^= se_s0
            se_s1 ^= LShR(se_s0, 26)
            se_state1 = se_s1

            mantisse = int(self.intern_sekvens[i] * (1 << 53))

            løsner.add(
                int(mantisse) == ((se_state0 + se_state1) & 0x1FFFFFFFFFFFFF)
            )

        if løsner.check() != sat:
            return None

        model = løsner.model()
        self.state0 = model[t0_ref].as_long()
        self.state1 = model[t1_ref].as_long()

        for i in range(len(self.intern_sekvens)):
            self.xorshift128p()

    def forudsig_næste(self) -> Optional[float]:
        if self.state0 is None or self.state1 is None:
            return None

        resultat = self.xorshift128p()
        return float(resultat & 0x1FFFFFFFFFFFFF) / (1 << 53)

    def xorshift128p(self) -> int:

```

```
s1 = self.state0 & self.maske
s0 = self.state1 & self.maske
self.state0 = s0
s1 ^= (s1 << 23) & self.maske
s1 ^= (s1 >> 17) & self.maske
s1 ^= s0 & self.maske
s1 ^= (s0 >> 26) & self.maske
self.state1 = s1 & self.maske
return (self.state0 + self.state1) & self.maske
```

Bilag 5.7 solver/SpiderMonkeySolver.py

```
const http = require('http');
const fs = require('fs');

const server = http.createServer((req, res) => {
    if (req.url === '/') {
        res.writeHead(200, {
            'Content-Type': 'text/html'
        });

        res.end(fs.readFileSync('index.html'));
        return;
    }

    const url = new URL(req.url, `http://${req.headers.host}`);
    if (url.pathname.startsWith('/spin')) {
        const bet = parseInt(url.searchParams.get('bet')) || null;
        const result = Math.random();
        const number = Math.floor(result * 36) + 1;
        const rotation = result * 360
        const won = bet === number;

        res.writeHead(200, { 'Content-Type': 'application/json' });
        res.end(JSON.stringify({ number, rotation, bet, won }));
        return;
    }

    res.end("{}")
});

server.listen(3000, () => {
    console.log('Server kører på http://localhost:3000');
});
```

Bilag 5.8 web\_exploit/server.js

```
<html>

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Roulette</title>
    <style>
        body {
            margin: 0;
            padding: 40px;
            font-family: system-ui;
            background: #ffffff;
            color: #000000;
            display: flex;
            flex-direction: column;
            align-items: center;
            justify-content: center;
            min-height: 100vh;
            overflow-y: hidden;
        }

        #roulette {
            position: relative;
            width: 416px;
            height: 416px;
            overflow: hidden;
        }

        #wheel {
            position: relative;
            width: 400px;
            height: 400px;
            border-radius: 50%;
            border: 8px solid #d4af37;
            overflow: hidden;
            transition: transform 2s cubic-bezier(0.17, 0.67, 0.12, 0.99);
        }

        .slot {
            position: absolute;
            inset: 0;
            transform-origin: center center;
            transform: rotate(calc(var(--i) * 10deg - 90deg));
            clip-path: polygon(50% 50%, 100% 50%, 100% calc(50% + 35.5px), 50% 50%);
        }
    </style>

```

```
.slot:nth-child(odd) {  
    background: #c41e3a;  
}  
  
.slot:nth-child(even) {  
    background: #000000;  
}  
  
.slot span {  
    color: #ffffff;  
    font-weight: bold;  
    position: absolute;  
    top: 50%;  
    left: 100%;  
    transform: translate(-30px, 10px);  
    transform-origin: 50% 100%;  
    font-size: 14px;  
    pointer-events: none;  
}  
  
#result {  
    position: absolute;  
    top: 50%;  
    left: 50%;  
    transform: translate(-50%, -50%);  
    width: 80px;  
    height: 80px;  
    border-radius: 50%;  
    background: #281a05;  
    display: flex;  
    align-items: center;  
    justify-content: center;  
    font-size: 32px;  
    font-weight: bold;  
    color: #d4af37;  
}  
  
#marker {  
    position: absolute;  
    top: -10px;  
    left: 50%;  
    transform: translateX(-50%);  
    width: 0;  
    height: 0;  
    border-left: 15px solid transparent;  
    border-right: 15px solid transparent;  
    border-top: 25px solid #d4af37;  
    filter: drop-shadow(0 3px 5px #000000bd)  
}
```

```
#betInput {  
    padding: 16px;  
    font-size: 18px;  
    width: 150px;  
    border-radius: 8px;  
    border: none;  
    background: #cda05e;  
    color: #ffffff;  
    font-weight: bold;  
    text-align: center;  
}  
  
#betInput::placeholder {  
    color: #ffffff;  
    opacity: 0.6;  
}  
  
#spinButton {  
    padding: 16px 48px;  
    font-size: 18px;  
    background: #d4af37;  
    color: #000000;  
    border: none;  
    border-radius: 8px;  
    cursor: pointer;  
    font-weight: bold;  
    transition: background 0.2s;  
}  
  
#spinButton:hover:not(:disabled) {  
    background: #e5c048;  
}  
  
#spinButton:disabled {  
    opacity: 0.5;  
    cursor: not-allowed;  
}  
  
.history {  
    min-height: 70px;  
    margin-top: 30px;  
    padding: 20px;  
    background: #cda05e;  
    border-radius: 8px;  
    min-width: 300px;  
    text-align: center;  
}  
  
.history h3 {  
    margin: 0 0 10px 0;
```

```
        color: #ffffff;
    }

.numbers {
    display: flex;
    flex-wrap: wrap;
    gap: 8px;
    justify-content: center;
}

.num {
    color: #ffffff;
    width: 36px;
    height: 36px;
    display: flex;
    align-items: center;
    justify-content: center;
    border-radius: 4px;
    background: #3a3a3a;
    font-weight: bold;
}

.num.win {
    background: #00aa00;
}

.num.loss {
    background: #aa0000;
}
</style>

<script>
    document.addEventListener('DOMContentLoaded', () => {
        const wheel = document.getElementById('wheel');
        const result = document.getElementById('result');
        const spinButton = document.getElementById('spinButton');
        const betInput = document.getElementById('betInput');
        const history = document.getElementById('history');

        let lastRotation = 0;
        let results = [];

        for (let i = 0; i < 36; i++) {
            const slot = document.createElement('div');
            slot.className = 'slot';
            slot.style.setProperty('--i', i);
            slot.innerHTML = `<span>${36 - i}</span>`;

            wheel.appendChild(slot);
        }
    })
}
```

```
spinButton.addEventListener('click', async () => {
    spinButton.disabled = true;
    result.textContent = ' ';

    const bet = parseInt(betInput.value) || null;
    const url = bet ? `/spin?bet=${bet}` : '/spin';

    const response = await fetch(url);
    const data = await response.json();
    const { number, rotation, won } = data;

    // The rotation starts from zero,
    // so we animate with our last culminative rotation
    // with the shift of the previous rotation
    // and some extra rotations for dramatic effect
    const sector = 360 / 36;
    const extraRotation = 3 * 360;
    const computedRotation = ((360 - (results.length - 1)?.rotation
|| 0) + rotation) % 360
        + lastRotation
        + extraRotation;
    wheel.style.transform = `rotate(${computedRotation}deg)`;
    lastRotation = computedRotation;

    setTimeout(() => {
        results.push({ number, rotation, bet, won });
        updateHistory();

        result.textContent = number;
        spinButton.disabled = false;
    }, 2000);
});

const updateHistory = () => history.innerHTML = results.map(response => {
    let className = 'num';
    if (response.bet !== null) className += response.won ? ' win' : ' loss';

    return `<div class="${className}">${response.number}</div>`;
}).join('');

</script>
</head>

<body>

<div id="roulette">
    <div id="wheel"></div>
    <div id="marker"></div>
    <div id="result"> </div>
</div>
```

```
</div>

<div style="display: flex; gap: 12px; align-items: center; margin-top: 40px;">
    <input id="betInput" type="number" id="betInput" min="1" max="36"
placeholder="Gæt (1-36)">
    <button id="spinButton">Spin</button>
</div>

<div class="history">
    <h3>Tidligere resultater</h3>
    <div class="numbers" id="history"></div>
</div>
</body>

</html>
```

Bilag 5.9 web\_exploit/index.html