Analysis of the Linux Random Number Generator

Patrick Lacharme, Andrea Röck, Vincent Stubel, Marion Videau

October 23, 2009 - Rennes

HELSINKI UNIVERSITY OF TECHNOLOGY
Laboratory for Theoretical Computer Science

Nancy-Université
Université Henri Poincaré

ANSSI
Agence nationale de la sécurité des systèmes d'information

Université de Limoges

Outline

- **Random Number Generators**
- **The Linux Random Number Generator**
- **Building Blocks**
  - Entropy Estimation
  - Mixing Function
  - Output Function
- **Security Discussion**
- **Conclusion**

HELSINKI UNIVERSITY OF TECHNOLOGY
Laboratory for Theoretical Computer Science
Nancy-Université
Université Henri Poincaré
ANSSI
Agence nationale de la sécurité des systèmes d'information
Université de Limoges

Part 1

Random Number Generators

Random Numbers in Computer Science

- **Where do we need random numbers ?**
  - Simulation of randomness, e.g. Monte Carlo method
  - Key generation (session key, main key)
  - Protocols
  - IV, Nonce generation
  - Online gambling

- **How can we generate them ?**
  - True Random Number Generators (TRNG)
  - Pseudo Random Number Generators (PRNG)
  - PRNG with entropy input

True Random Number Generators (TRNG) :

- **Properties** :
  - Based on physical effects
  - Needs often post-processing
  - Often slow
  - Needs often extra hardware

True Random Number Generators (TRNG) :

- **Properties** :
  - Based on physical effects
  - Needs often post-processing
  - Often slow
  - Needs often extra hardware

- **Applications**
  - High security keys
  - One-Time Pad

True Random Number Generators (TRNG) :

- **Properties** :
  - Based on physical effects
  - Needs often post-processing
  - Often slow
  - Needs often extra hardware
- **Applications**
  - High security keys
  - One-Time Pad
- **Examples** :
  - Coin flipping, dice
  - Radioactive decay
  - Thermal noise in Zener diodes
  - Quantum random number generator

Pseudo Random Number Generators (PRNG)

- **Properties** :
  - Based on a short seed and a completely deterministic algorithm
  - Allows theoretical analysis
  - Can be fast
  - Entropy not bigger than size of seed

Pseudo Random Number Generators (PRNG)

- **Properties** :
  - Based on a short seed and a completely deterministic algorithm
  - Allows theoretical analysis
  - Can be fast
  - Entropy not bigger than size of seed
- **Applications** :
  - Monte Carlo method
  - Stream cipher

Pseudo Random Number Generators (PRNG)

- **Properties** :
  - Based on a short seed and a completely deterministic algorithm
  - Allows theoretical analysis
  - Can be fast
  - Entropy not bigger than size of seed
- **Applications** :
  - Monte Carlo method
  - Stream cipher
- **Examples** :
  - Linear congruential generators
  - Blum Blum Shub generator
  - Block cipher in counter mode
  - Dedicated stream cipher (eSTREAM project)

PRNG with Entropy Input

- **Properties** :
  - Based on hard to predict events (entropy input)
  - Apply deterministic algorithms
  - Few examples of theoretical models [Barak Halevi 2005]

PRNG with Entropy Input

- **Properties** :
  - Based on hard to predict events (entropy input)
  - Apply deterministic algorithms
  - Few examples of theoretical models [Barak Halevi 2005]
- **Applications** :
  - Fast creation of unpredictable keys
  - When no additional hardware is available

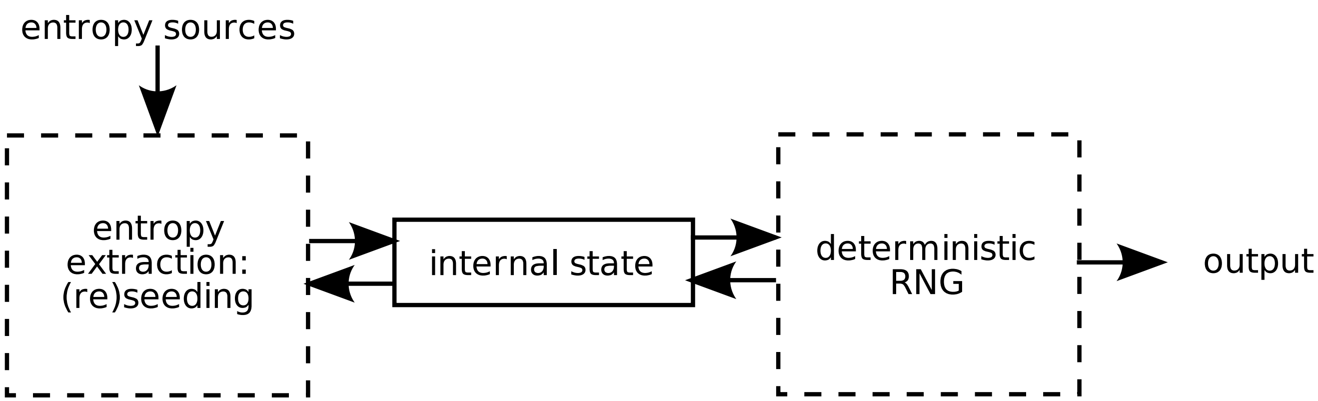
PRNG with Entropy Input

- **Properties** :
  - Based on hard to predict events (entropy input)
  - Apply deterministic algorithms
  - Few examples of theoretical models [Barak Halevi 2005]

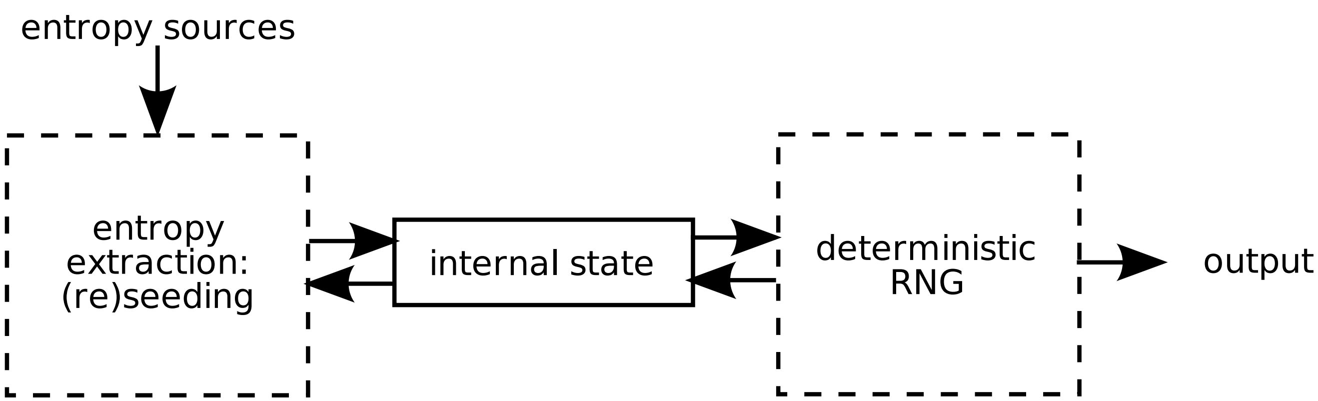
- **Applications** :
  - Fast creation of unpredictable keys
  - When no additional hardware is available

- **Examples** :
  - Linux RNG : /DEV/RANDOM
  - Yarrow, Fortuna
  - HAVEGE

Model of a PRNG with Entropy Input



Model of a PRNG with Entropy Input



- **Resilience/Pseudorandom Security :**
  The output looks random without knowledge of internal state
  - Direct attacks : an attacker has no control on entropy inputs
  - Known input attacks : an attacker knows a part of the entropy inputs
  - Chosen input attacks : an attacker is able to chose a part of entropy inputs

Cryptanalytic Attacks - After Compromised State

Compromised state :
The internal state is compromise if an attacker is able to recover a part of the internal state (for whatever reasons) [Kelsey et al. 1998]

- Forward security/Backtracking resistance :
  - Earlier output looks random with knowledge of current state
- Backward security/Prediction resistance :
  - Future output looks random with knowledge of current state
  - Backward security requires frequent reseeding of the current state

Same Remarks about Entropy (1)

- **(Shannon's) entropy is a measure of unpredictability :**
  Average number of binary questions to guess a value

- Shannon's Entropy for a probability distribution $p_1, p_2, \ldots, p_n$:

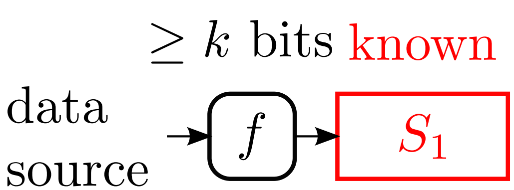
$$H = -\sum_{i=1}^{n} p_i \log_2 p_i \ \leq \log_2(n)$$

- Min-entropy is a worst case entropy :

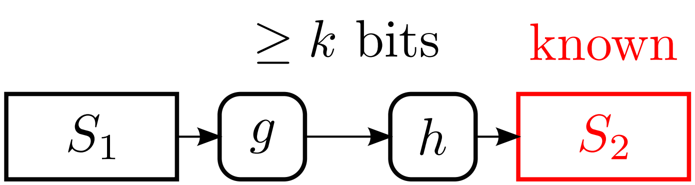
$$H_{\min} = -\log_2\left(\max_{1\leq i\leq n}(p_i)\right) \ \leq H$$

Same Remarks about Entropy (2)

- **Collecting k bits of entropy** :
  After processing the unknown data into a known state S_1, an observer would have to try on average 2^k times to guess the new value of the state.
- **Transferring k bits of entropy from state S_1 to state S_2** :
  After generating data from the unknowing state S_1 and mixing it into the known state S_2 an adversary would have to try on average 2^k times to guess the new value of state S_2.
  By learning the generated data from S_1 an observer would increase his chance by the factor 2^k of guessing the value of S_1.

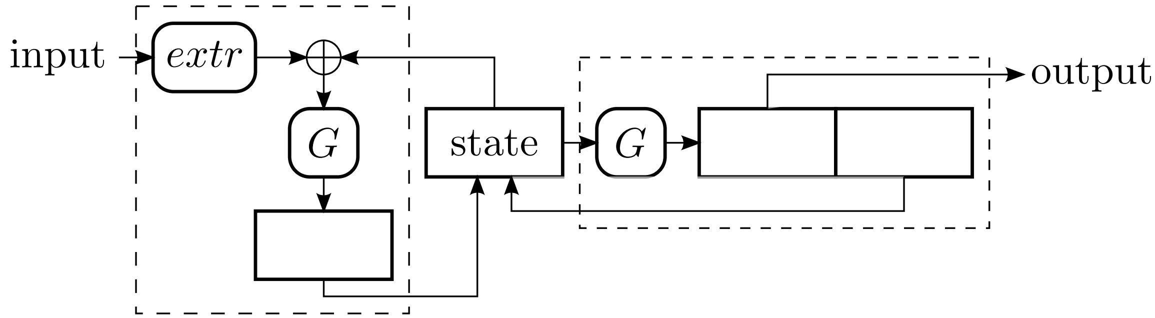


collecting entropy



transferring entropy

Model of [Barak Halevi 2005]



- State of size m
- Extractor for a family $\mathcal{H}$ of probability distributions, such that for any distribution $\mathcal{D} \in \mathcal{H}$ and any $y \in \{0,1\}^m$:

$$2^{-m}(1 - 2^{-m}) \leq Pr[extr(X_{\mathcal{D}}) = y)] \leq 2^{-m}(1 + 2^{-m})$$

- G is a cryptographic PRNG producing $2m$ bits
- Supposes regular input with given minimal entropy
- Proven security in theory, hard to use in practice

Part 2

The Linux Random Number Generator

The Linux Random Number Generator

- Part of the Linux kernel since 1994
- From Theodore Ts'o and Matt Mackall
- Only definition in the code (with comments) :
  - About 1700 lines
  - Underly changes (`www.linuxhq.com/kernel/file/drivers/char/random.c`)
  - We refer to kernel version 2.6.30.7
- Pseudo Random Number Generator (PRNG) with entropy input

Analysis

- **Previous Analysis :**
  - **[Barak Halevi 2005]** :
    Almost no mentioning of the Linux RNG
  - **[Gutterman Pinkas Reinman 2006]** :
    They show some weaknesses of the generator which are now corrected

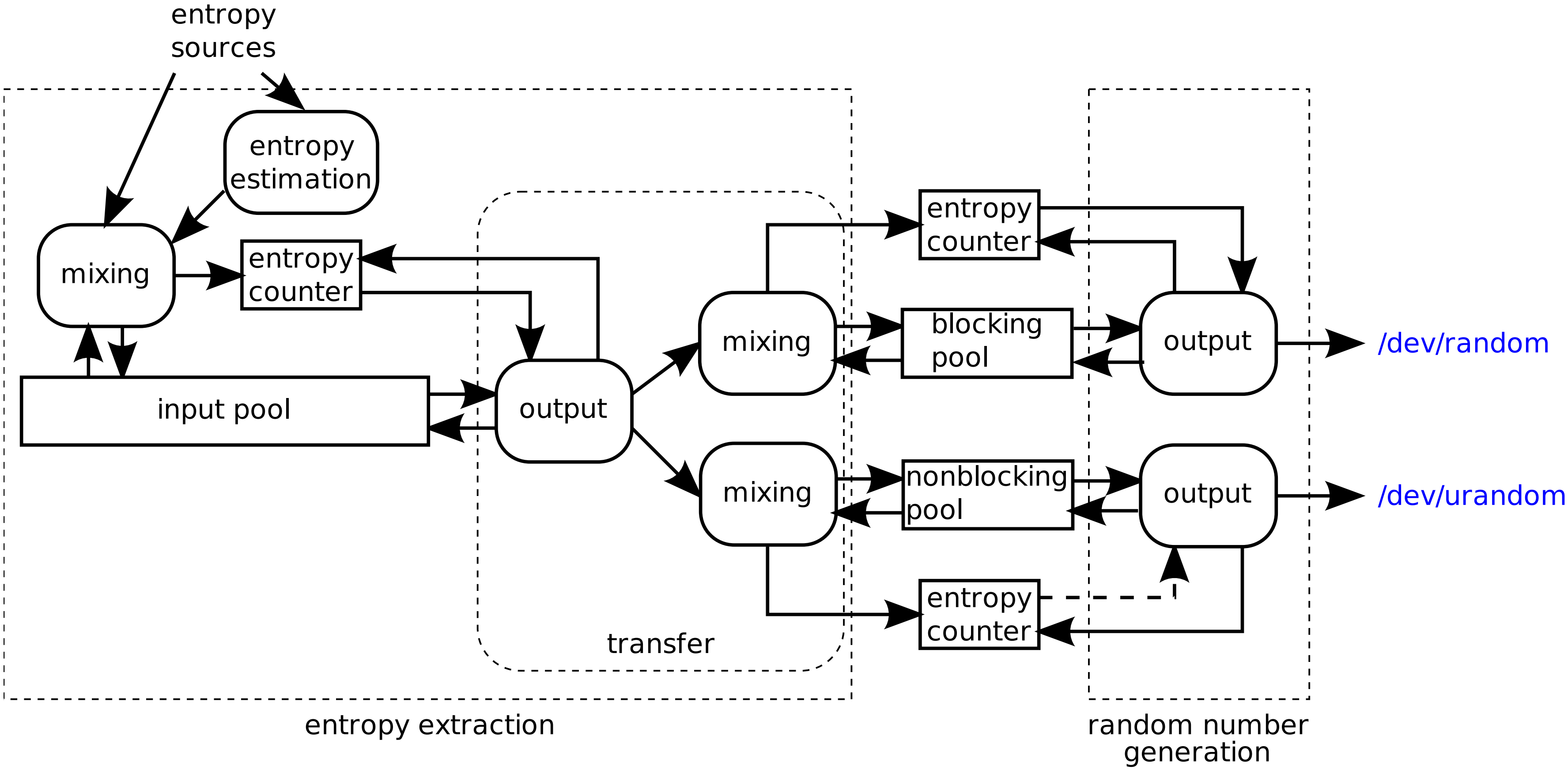
- **Why a new analysis :**
  - As part of the Linux kernel, the RNG is widely used
  - The implementation has changed in the meantime
  - Want to give more details

General

- **Two different versions :**
  - `/dev/random` :
    Limits the number of generated bits by the estimated entropy
  - `/dev/urandom` :
    Generates as many bits as the user asks for

- **Two asynchronous procedures :**
  - The entropy accumulation
  - The random number generation

Structure



- Size of input pool : 128 32-bit words
- Size of blocking/unblocking pool : 32 32-bit words

Functionality (1)

Entropy input :

- Entropy sources :
  - User input like keyboard and mouse movements
  - Disk timing
  - Interrupt timing

- Each event contains 3 values :
  - A number specific to the event
  - Cycle count
  - Jiffies count (count of time ticks of system timer interrupt)

Functionality (2)

Entropy accumulation :

- Independent to the output generation
- Algorithm :
  - Estimate entropy
  - Mix data into input pool
  - Increase entropy count

- Must be fast

Functionality (3)

Output generation

- Generates data in 80 bit steps
- Algorithm to generate n bytes :
  - If not enough entropy in the pool ask input pool for n bytes
  - If necessary, input pool generates data and mixes it into the corresponding output pool
  - Generate random number from output pool
- Differences between the two version :
  - `/dev/random` : Stops and waits if entropy count of its pool is 0
  - `/dev/urandom` : Leaves $\geq$ 128 bits of entropy in the input pool

Functionality (4)

Initialization :

- Boot process does not contain much entropy
- Script recommended that
  - At shutdown :
    Generate data from `/dev/urandom` and save it
  - At startup :
    Write to `/dev/urandom` the saved data
    This mixes the same data into the blocking and nonblocking pool without increasing the entropy count
- Problem for Live CD versions

Part 3

Building Blocks

The Entropy Estimation

- Crucial point for `/dev/random`
- Must be fast (after interrupts)
- Uses the jiffies differences to previous event
- Separate differences for user input, interrupts and disks
- Estimator has no direct connection to Shannon's entropy

The Entropy Estimation - The Estimator

- Let $t^A(n)$ denote the jiffies of the n'th event of source A

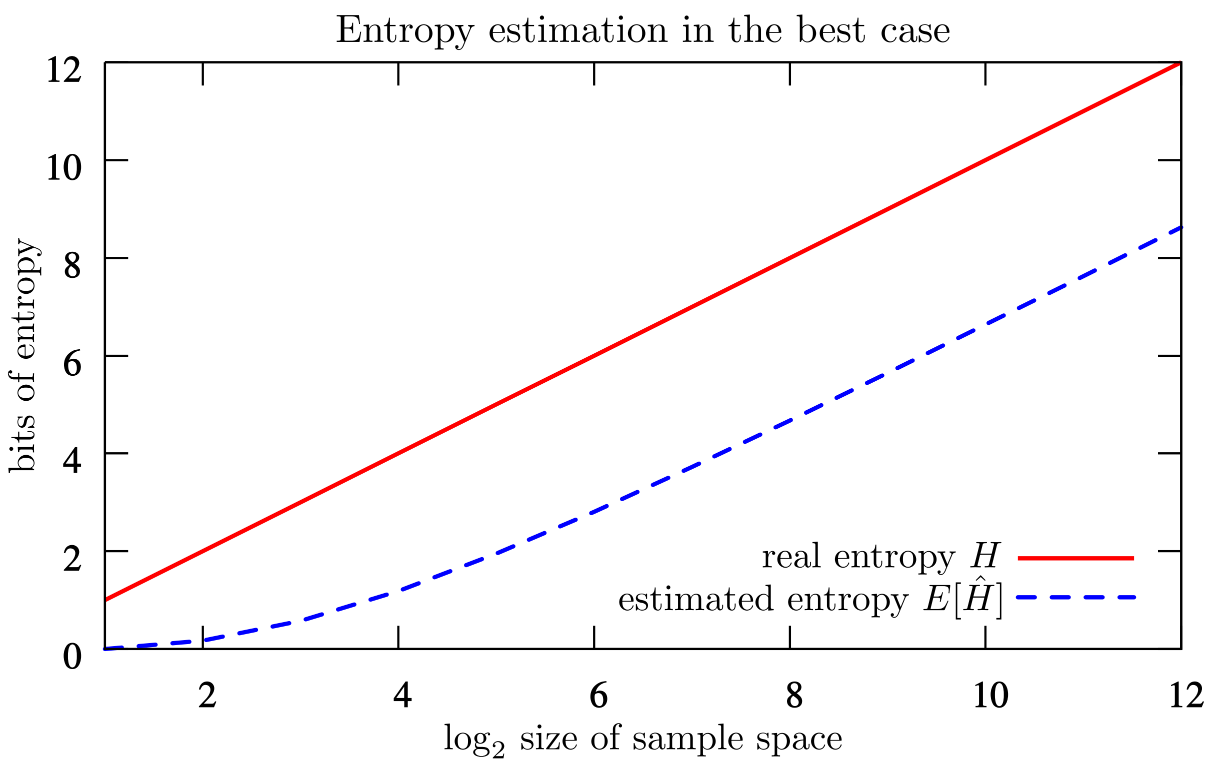
$$\begin{aligned}
\Delta_1^A(n) &= t^A(n) - t^A(n-1) \\
\Delta_2^A(n) &= \Delta_1^A(n) - \Delta_1^A(n-1) \\
\Delta_3^A(n) &= \Delta_2^A(n) - \Delta_2^A(n-1) \\
\Delta^A(n) &= \min\left(|\Delta_1^A(n)|, |\Delta_2^A(n)|, |\Delta_3^A(n)|\right)
\end{aligned}$$

- Estimated Entropy : $\hat{H}^A(n) = \hat{H}\left(\Delta_1^A(n), \Delta_1^A(n-1), \Delta_1^A(n-2)\right)$

$$\hat{H}^A(n) = \begin{cases} 0 & \text{if } \Delta^A(n) = 0 \\ 11 & \text{if } \Delta^A(n) \geq 2^{12} \\ \left\lfloor \log_2\left(\Delta^A(n)\right)\right\rfloor & \text{otherwise} \end{cases}$$

The Entropy Estimation - Uniform Case

- $\Delta_1^{[n]}, \Delta_1^{[n-1]}, \Delta_1^{[n-2]}$ uniformly distributed with support $\{0,1\}^m$ for H $(1 \leq m = H \leq 11)$:
- Compare $E\left[\hat{H}\left(\Delta_1^{[n]}, \Delta_1^{[n-1]}, \Delta_1^{[n-2]}\right)\right]$:



The Entropy Estimation - Worst Case

- Predictable input which maximizes $\hat{H}$:

	$\Delta_1(n)$	$\Delta_2(n)$	$\Delta_3(n)$
$n = 2m - 1$	δ	$-\delta$	-2δ
$n = 2m$	2δ	δ	2δ

- Then for all $n \geq 1$ and $1 \leq \delta < 2^{12}$

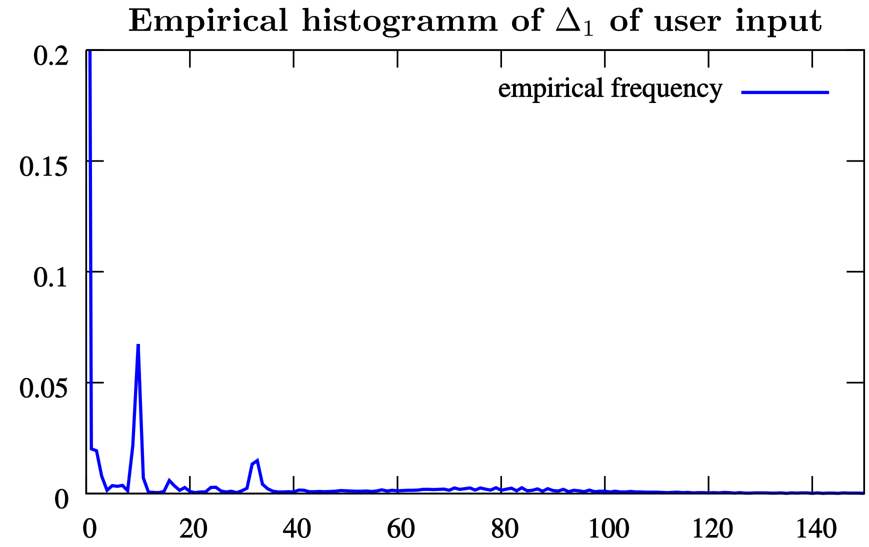
$$\hat{H}(n) = \lfloor \log_2(\delta) \rfloor$$

- For $\Delta_1^{[n]}, \Delta_1^{[n-1]}, \Delta_1^{[n-2]}$ uniformly distributed :

$$E\left[\hat{H}\left(2^c \cdot \Delta_1^{[n]}, 2^c \cdot \Delta_1^{[n-1]}, 2^c \cdot \Delta_1^{[n-2]}\right)\right] = c \cdot E\left[\hat{H}\left(\Delta_1^{[n]}, \Delta_1^{[n-1]}, \Delta_1^{[n-2]}\right)\right]$$

The Entropy Estimation - Empirical Data

- More than 7M of samples of user input events :

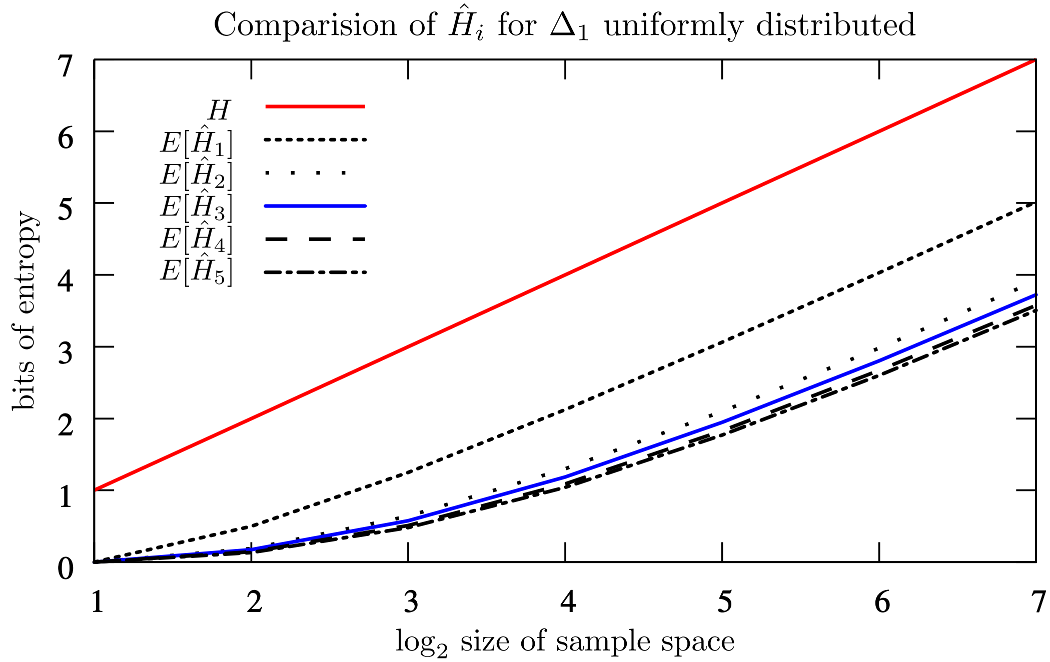


- Comparison (H and $H_{\min}$ based on empirical frequencies) :

	jiffies	cycles	num
$\frac{1}{N-2}\sum_{n=3}^{N}\hat{H}(n)$	1.85	10.62	5.55
H	3.42	14.89	7.31
$H_{\min}$	0.68	9.69	4.97

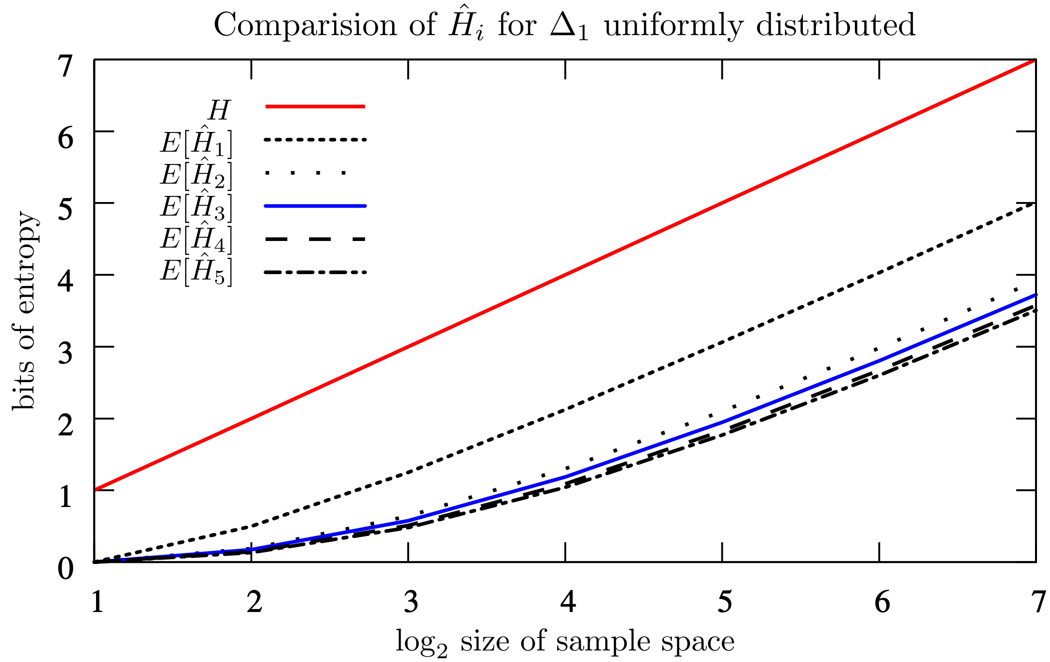
The Entropy Estimation - Levels of Δ

- $\hat{H}_i(n)$: estimator where $\Delta(n)$ depends on i levels of differences.



The Entropy Estimation - Levels of Δ

- $\hat{H}_i(n)$: estimator where $\Delta(n)$ depends on i levels of differences.



- Comparison for empirical data :

	H	$\frac{1}{N}\sum_{n=1}^{N}\hat{H}_1(n)$	$\frac{1}{N-1}\sum_{n=2}^{N}\hat{H}_2(n)$	$\frac{1}{N-2}\sum_{n=3}^{N}\hat{H}_3(n)$	$\frac{1}{N-3}\sum_{n=4}^{N}\hat{H}_4(n)$
jiffies	3.42	1.99	1.99	1.85	1.47

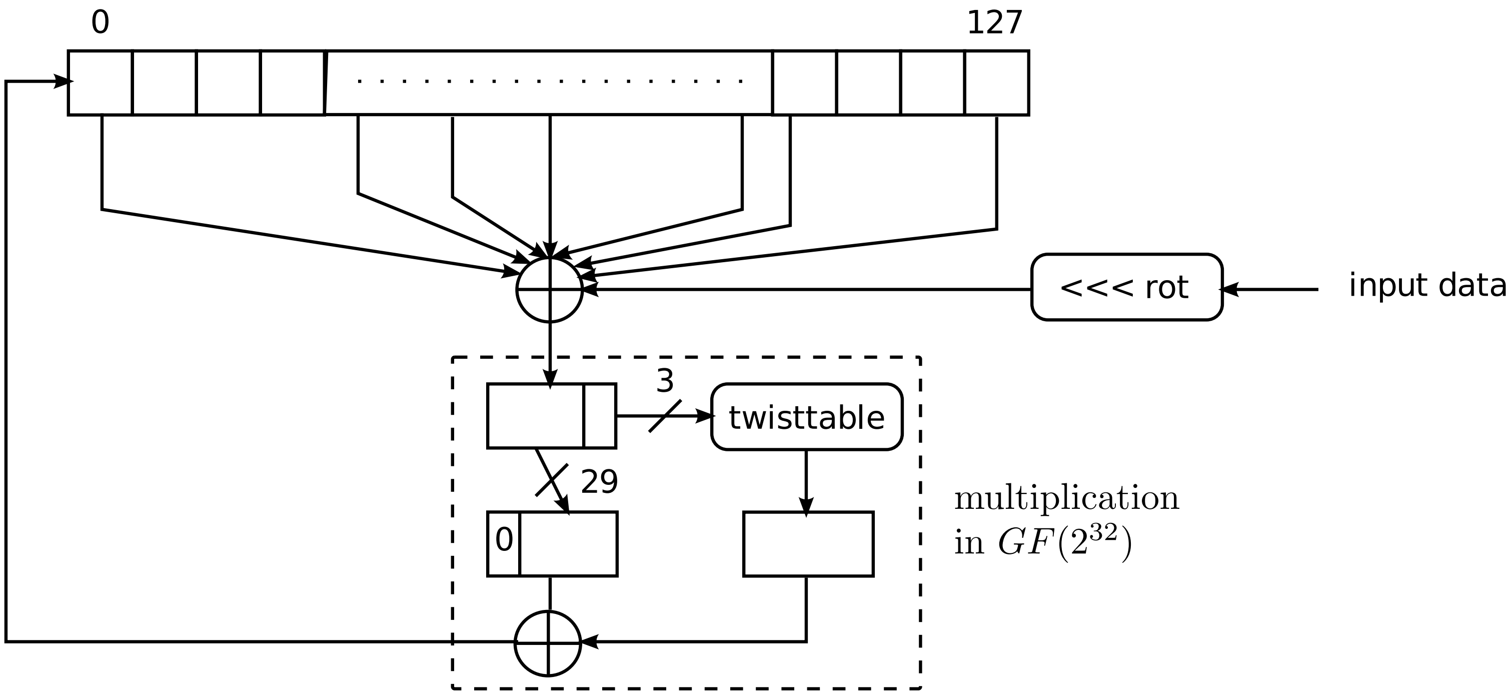
		$\frac{1}{N-4}\sum_{n=5}^{N}\hat{H}_5(n)$	$\frac{1}{N-5}\sum_{n=6}^{N}\hat{H}_6(n)$	$\frac{1}{N-6}\sum_{n=7}^{N}\hat{H}_7(n)$	$\frac{1}{N-7}\sum_{n=8}^{N}\hat{H}_8(n)$
jiffies		1.36	1.27	1.10	0.99

The Mixing Function

- Mixes one byte at a time
  - Completes it to 32 bits and rotates it by a changing factor
- Uses a shift register
- Diffuses entropy in each pool
- Same mechanism for each pool, according to the size of the pool

The Mixing Function - Description

- Inspired by Twisted GFSR [Matsumoto Kurita 1992]
- Applies CRC-32-IEEE 802.3 polynomial in twisted table
- Works on 32-bit words



The Mixing Function - Analysis Without Input (1)

- The Twisted GFSR is defined for trinomials : $X_{\ell+n} + X_{\ell+m} + X_\ell A$
- Uses polynomial on 32-bit words (primitive in $GF(2)$) :

$$P(X) = \begin{cases} X^{128} + X^{103} + X^{76} + X^{51} + X^{25} + X + 1 & \text{input pool} \\ X^{32} + X^{26} + X^{20} + X^{14} + X^{7} + X + 1 & \text{output pool} \end{cases}$$

- Whole method can be written as : $\alpha^3(P(X) - 1) + 1$ where α is from $GF(2^{32})$ defined by the CRC-32 polynomial
- This polynomial is not irreducible in $GF(2^{32})$, thus no maximal period
  - $\leq 2^{92*32} - 1$ instead of $2^{128*32} - 1$ for the input pool
  - $\leq 2^{26*32} - 1$ instead of $2^{32*32} - 1$ for the output pool

The Mixing Function - Analysis Without Input (2)

- We can make it irreducible by just changing one feedback position, e.g. :

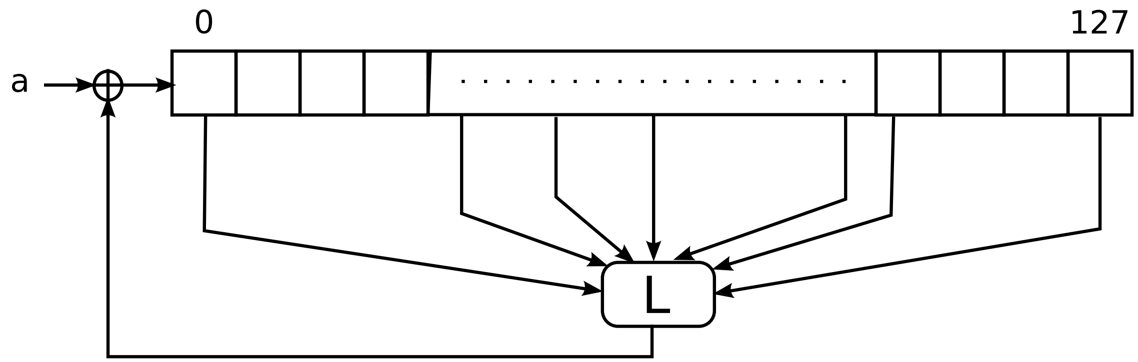
$$P(X) = \begin{cases} X^{128} + X^{104} + X^{76} + X^{51} + X^{25} + X + 1 & \text{input pool} \\ X^{32} + X^{26} + X^{19} + X^{14} + X^{7} + X + 1 & \text{output pool} \end{cases}$$

have respectively periods of $(2^{128*32} - 1)/3$ and $(2^{32*32} - 1)/3$

- We can achieve a primitive polynomial by using $\alpha^i(P(X) - 1) + 1$, with $\gcd(i, 2^{32} - 1) = 1$, e.g. $i = 1, 2, 4, 7, \ldots$

The Mixing Function - Analysis With Input

- The feedback function $L(x_0, x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4}, x_{i_5})$ is linear
- The input can be seen as :



- If we have $x_0 \oplus a$ in the first cell we can write :

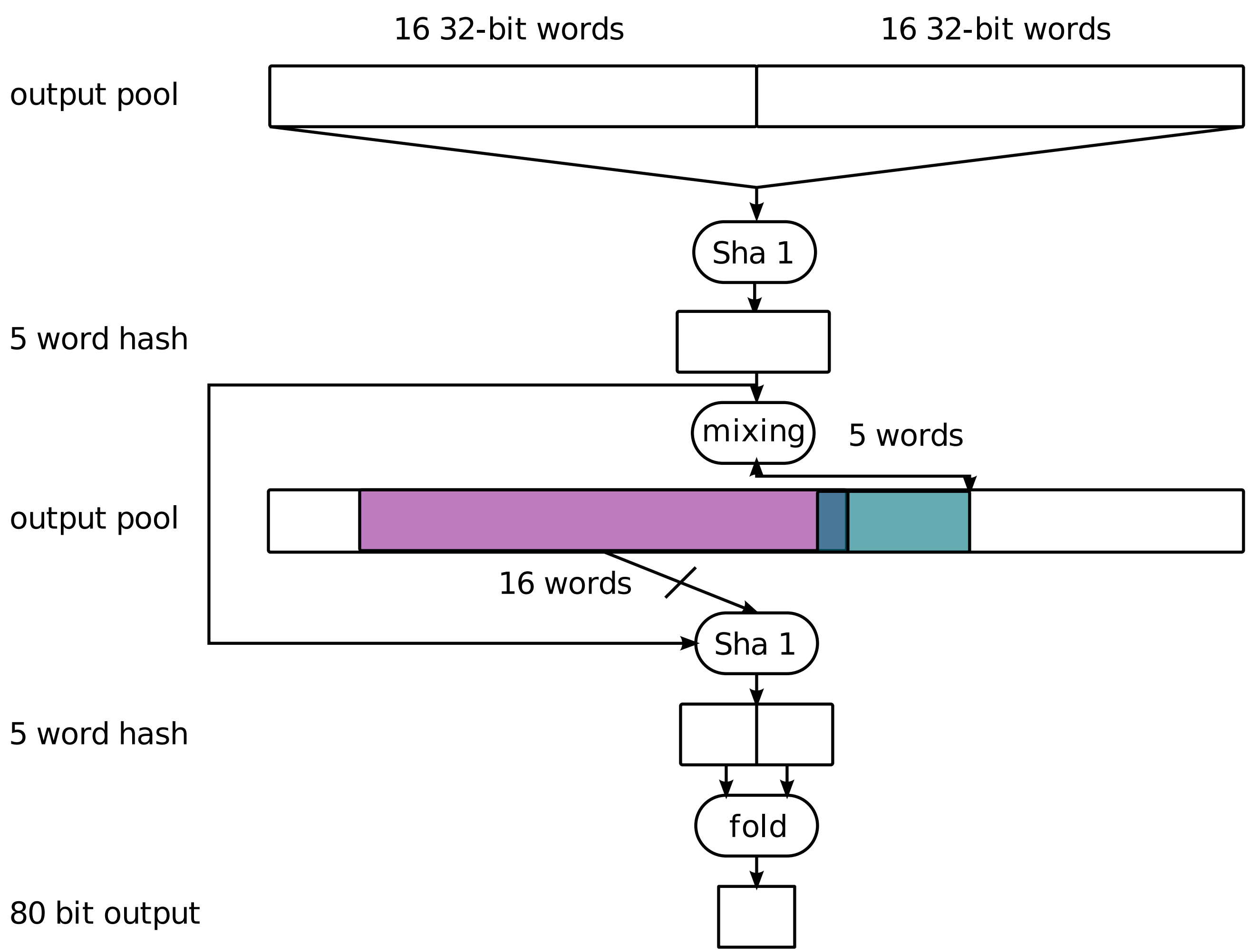
$$L(x_0, x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4}, x_{i_5}) \oplus L(a, x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4}, x_{i_5})$$

- If we know nothing about a or x_0 we cannot guess the next feedback more easily than guessing the unknown value

The Output Function

- Uses Sha-1 with feedback
- Is identical for each pool, according the size of the pool
- Is used for the resilience property
- Is used to avoid cryptanalytic attacks

The Output Function - Description



The Output Function - Analysis

- Changed since paper of Gutterman *et al.*
- Feedback is used for the Forward Security
- Changes $2k$ bits for every k bits of output
- Hard to give a mathematical analysis

Part 4

Security Discussion

Major Changes Since Analysis of Gutterman et al.

- Mixes bytes into the pool and no 32bit words
- Output function mixes all 5 words of the hash back at once and not one word after each hashing of 16 words
- /dev/urandom cannot empty the input pool
- The input is only mixed into the input pool
- Use not only the cycles but also the jiffies as a timestamp and estimate entropy over the jiffies

Forward Security

- Let M be the size of the pool and C the entropy count
- For generating $k \leq \frac{M}{2}$ bits we change $2k$ bits in the pool
  - If we know the state, guessing the previous output is easier than finding the previous state
- `/dev/urandom` : If we have previously generated $k > M$ bits without new entropy input, guessing the previous state might be easier than guessing the previous output
- `/dev/random` : For generating $k > C$ bits we need k bits from the input pool, especially if $k > M$

Backward Security

- If the attacker knows the state and we input 1 unknown word, the attacker looses the knowledge of one word in the register
- If an observer knows the input but not the state, he can not learn anything of the state
- The period of the register without input is not maximal but large

Resilience

- If we assume that there is enough unknown input and a correct entropy estimation, then the output should not be distinguishable from a random sequence
- What happens if there are no good entropy sources ?
- Uses the pseudorandom assumption of a cryptographic hash function
- Both output pools are fed from the same pool but we do not see a concrete way to exploit this fact

The Entropy Estimation

- No direct connection to Shannon's entropy
- Gives no information about knowledge of observer
- Underestimates entropy of a uniform source and of empirical data
- Uses few resources
- Other entropy estimators in literature generally use all samples and need more storage

Comparison with other models (1)

- [Kelsey *et al.* 2000] present the general model Yarrow
  - One output state (key and counter) and two input pools (fast and slow pool)
  - Uses a hash function for entropy extraction and a block cipher for the PRNG
  - Separate entropy count for each pool and each input source
  - Designed to prevent specific attacks

- Their updated version Fortuna does not use entropy estimation anymore

Comparison with other models (2)

- NIST SP 800-90 [Barker Kelsey 2007]
  - Has one state
  - Allows multiple instances
  - Recommends personalization string for initialization
  - Regular tests during generation
  - Specific systems based on one primitive :
    e.g. hash function, HMAC, block cipher, or dual elliptic curves

Part 5

Conclusion

Conclusion

- The Linux random number generator changed a lot since the last analysis
- It is important to have good entropy sources
- The entropy estimator is fast and works not "too bad" for unknown data even if there is no direct connection to the entropy
- The mixing function is a non irreducible polynomial over $GF(2^{32})$ and is not really a twisted GFSR
- The output function resists previous attacks and changes 160 bits in each step

Open Problems

- Is there a better mixing function ?
- Is there a better entropy estimator ?
- Can we say anything more mathematical about the output function ?
- Can we make a proof similar to [Barak Halevi 2005] ?