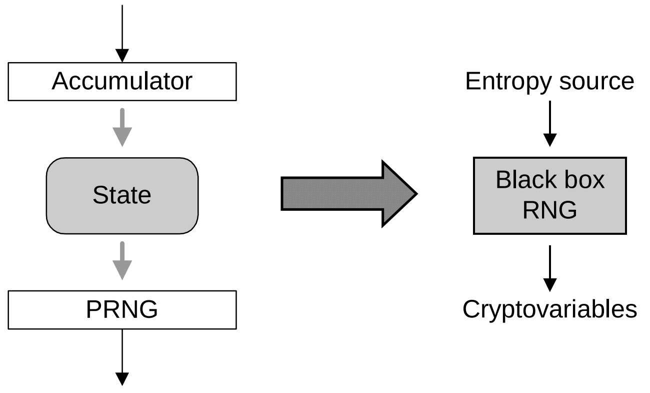
# Testing Issues with OS-based Entropy Sources

Peter Gutmann

University of Auckland

## Generator Abstract Model



Need to analyse/verify

- Entropy source
- Black box RNG (correct implementation)
- Usability

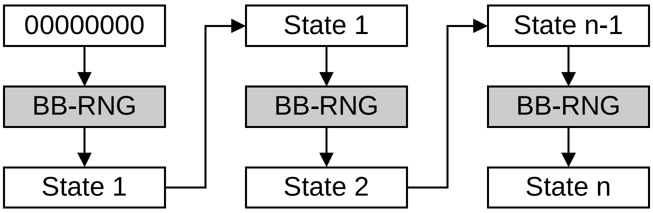
# Verifying Implementation Correctness

Many published generators have contained implementation flaws

- PGP 2.x xorbytes bug
  - `*dest++ = *src++;` (should be ^=)
- GPG 8 years later had the same type of bug
  - Both were present for years in published code
  - Only discovered by coincidence, not as part of a conscious code review
- BSAFE/RSAREF was vulnerable to being fed small data blocks
- OpenSSL/SSLeay only fed a small amount of random pool data (1-10 bytes) into the PRNG

# Verifying Implementation Correctness (ctd)

Use CBC-MAC style verification



- State1 … *n*-1 should be > pool size
  - Verifies correct pool update
- State *n* is verification value, e.g. 128 bits after 10 iterations
  - Great way to find compiler bugs

## Entropy Estimation

Entropy of a set of symbols is given by

$$H = -\sum_{i=1}^{n} p(S_i)\log(p(S_i))$$

Example: English alphabet

H = -( *p*( A ) * log( *p*( A ) ) + *p*( B ) …

= ~4.05 bits/byte (using standard values for *p*( A ) … *p*( Z ))

- Standard entropy measure for data compression is bits/byte (= bits/symbol)

Textbook formula for simple entropy estimation

- Not very good
- 00 01 02 … FE FF 00 01 02 … = perfectly “random”
  - – All symbol frequencies are equal

## Entropy Estimation (ctd)

Need to take inter-symbol influence into account

- Simple model is a memoryless source or zero-order Markov model
- One that considers inter-symbol influence over symbol pairs is a first-order Markov model
- Conditional probability *p*( x | y ) = probability of seeing y once x has been seen
- Conditional entropy of first-order source

$$H = -\sum_{i}\sum_{j} p(x, y)\log(p(y \mid x))$$

- Conditional entropy of second-order source

$$H = -\sum_{i}\sum_{j}\sum_{k} p(x, y, \mathrm{z})\log(p(\mathrm{z} \mid xy))$$

## Entropy Estimation (ctd)

Example: English alphabet

- Zero-order, equiprobable symbols (order –1) = 4.75 bits/byte
- Zero-order, standard probabilities = 4.05 bits/byte
- First order = 3.32 bits/byte
- Second order = 3.10 bits/byte
- Actual figure is probably ~1 bit/byte
  - Determined using an arbitrary-order Markov model via humans who bet on symbol probabilities
  - Even humans can’t get to 1 bit/byte

Example: Pathological “random” data 00 01 02 … FE FF

- Zero-order = 8 bits/byte
- First order = 0 bits/byte

## Entropy Estimation (ctd)

Problems with high-order Markov models

- Consumes huge amounts of memory
  - Order 3 = 4 billion nodes
- Requires large amounts of input to build model
  - 1MB is a good start for order 2-3
- Not exactly real-time performance

Doesn’t work well in embedded environments / crypto HW

- No memory, CPU cycles

Doesn’t work well in standard system using entropy polling

- Not enough data to build model

# Adapting Faster

Statistical compressors

- Allocate codewords to each source symbol to minimise average code length

Dictionary compressors

- Replace a string of symbols with a reference to a previous occurrence of the string

Dictionary compressors adapt to input changes more quickly than statistical compressors

- Only require one repeat occurrence to adapt, not many

# Dictionary Compressors

Two major families of dictionary compressors, LZ77 and LZ78

- Both LZ77 and LZ78 were actually designed as entropy estimators

LZ77 replaces strings with a pointer to a previous occurrence of the string in a sliding window

- Best-known example: Zip

LZ78 enters strings into a dictionary

- Subsequent occurrences are replaced by the position in the dictionary
- Best-known examples: LZW, compress, GIFs, V.42bis

## Dictionary Compressors (ctd)

LZ77 adapts far more quickly than LZ78

- LZ78: What string/substring should be entered into the dictionary?
- LZ77: All substrings are automatically in the sliding window

Dictionary compression acts as a variable-order Markov model

- Sawtooth function, low-order at start, high-order at end
- Demonstration of equivalence of higher-order statistical models and dictionary compressors
  - Algorithm to convert greedy-parsing dictionary compressor to equivalent predictive model

## Dictionary Compressors (ctd)

Dictionary compressors don’t work well for high-entropy data and/or short strings

- Use an enhanced statistical compressor to handle those cases

## Practical higher-order Markov Models

### Making high-order models workable

- Only keep statistics for contexts you’ve seen before
- Use escapes to fall back to shorter contexts

| Context | Order | ‘a’ | ‘b’ | ‘c’ | ESC |
|---|---|---|---|---|---|
| abc | 3 | — | 0.8 | — | 0.2 |
| bc | 2 | 0.1 | — | 0.6 | 0.3 |
| c | 1 | 0.0 | 0.4 | 0.2 | 0.4 |
| — | 0 | 0.2 | — | 0.6 | 0.2 |
| — | -1 | 0.33 | 0.33 | 0.33 | — |

- Replace nodes on an LRU basis

## Hybrid compressors

### Combine dictionary + statistical compressors

- Statistical = order 0…~2, dictionary = order 3…*n*
- Dictionary handles low-entropy data, fast adaptation
- Statistical handles higher-entropy data

### Statistical compressor handles both output of dictionary compressor and literal data that “fell through” the dictionary

- LRU nature of LZ77 makes the output further compressible
- If multiple instances of a string are present in the window, the most recent one (shortest displacement) is used preferentially

## Convergence of Entropy Estimators

The word “universal” as used with entropy estimators doesn’t convey the property you think it does

- A “universal” entropy estimator over-estimates by no more than a constant…

  … which can be arbitrarily large

All “universal” compressors only converge on an ergodic source at infinity

- Markov model has infinite order
- Dictionary compression sawtooth has infinite period
- Very high-order models of English text fed with white noise (almost) reproduce the original text
  - Requires a word-based model because a symbol-based model would take forever to build

## Data Compression as Entropy Estimation

We don’t care about absolute compressibility, only absolute entropy per sample

- Use a compressor to determine the change in entropy from one sample to the next
- Compression estimates (non)randomness in data
- Compression over multiple samples detects amount of new entropy flowing into the system

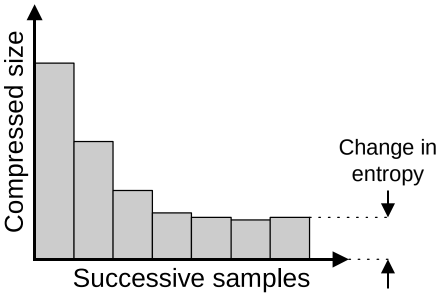
Lossless compression can be viewed as encoding the error signal between the estimated (model) and actual data

Assumes a Markov source

- This is something that you can’t do with a genuine noise source
- (This constraint is both good and bad)

## Data compression as Entropy Estimation (ctd)

Use previous samples to prime (train) the compressor model



- In practice 3 samples are usually enough

## Lossy Data Compression as Entropy Est.

Can perform estimation on a single sample using lossy compression

- Compress sample with strong tracking of signal
  - High-Q JPEG / high PSNR
- Compress sample with weak tracking of signal
  - Low-Q JPEG / low PSNR
- The difference is noise…

  … or loss of detail

## Lossy Data Compression as Entropy Est (ctd)

Analogous to using a low-pass filter on a signal to remove high-frequency sampling noise

- Quantifying “noise” in a non-analogue signal is difficult
- Even in the analogue realm, it only works on select sources

Assumes correlations between nearby samples

- Continuous-tone vs. bi-level / discrete-tone images

Not necessarily a useful estimator of entropy

- Image of sky vs. image of Floyd-Steinberg dithered image

## Lossy Data Compression as Entropy Est (ctd)

Use wavelet transform to identify high-freq components

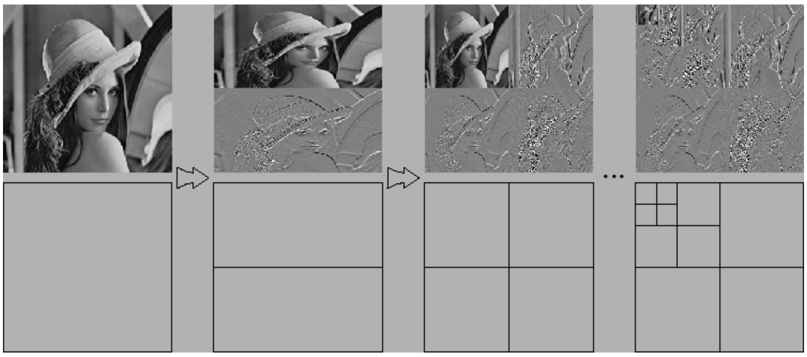


Image courtesy Vicente Ruiz, Universidad de Almería

- Level 1 subbands = high-frequency / unimportant details
  - Quantise heavily
- Level n subbands = low-frequency / important features
  - Quantise minimally

# Entropy Sources

Anything that changes and is somewhat unpredictable

- Disk, memory, VM, thread, process, network statistics in infinite variations
  - Available programmatically or via system commands
- User input (mouse, keyboard, program use)
- Full enumeration of sources is about 10 A4 pages long

Value of data is relative, e.g. network packet stats

- pcap / pf access = nil
- Broadcast LAN access = low
- Switched LAN access = moderate
- WAN access = high

# Entropy Sources (ctd)

System-specific sources

- procfs
- Solaris kstats / WinNT/2K/XP kernel stats
- Tandem heisencounters
- /dev/random / EGD / PRNGD
- MVS OS operation / system call latency
- MBM thermal / power management information
- CPU performance counters
- BeOS `is_computer_on_fire()` (MB temperature)
- VIA C5 hardware RNG
  - Intel, AMD hardware RNGs are dead :-(

Many of the sources have undocumented components

## Entropy Sources (ctd)

Some of the more obscure sources

- CPU cooling fan speed variations
- Vcore drift
- HDD read errors corrected via ECC
- HDD read errors corrected via retry
- Drive head settle time on speculative read

Some sources would require unworkably complex physical models

- Interaction of air current flows, thermal flows, and supply voltage inside PC case
- Change in supply voltage affects fan speed affects air flow affects temperature affects PSU affects supply voltage …

## Entropy Sources (ctd)

Address failure via fault-tolerant design

- Tolerant of faults, not necessarily a formal fault-tolerant design
- Many, many entropy sources
  - Fault-tolerance of entropy sources via massive redundancy
- Redundant generator elements
  - Fortezza RNG
  - cryptlib RNG

Brooklyn Bridge was built seven times as strong as the best available worst-case estimate because the designer knew what he didn’t know

# The PC as PRNG

In effect the computer is acting as a (massively complex) PRNG

- PRNG seeding is provided by user input, physical sources, …
- Complete system is a CSPRNG

Defeating brute-force key-search via PRNG complexity was first suggested in the early ’90s

- Key-crackers use custom hardware to perform many simple operations very quickly
- Defeat by using large amounts of memory, ops that are hard to do efficiently in hardware (32-bit multiply, divide)

# The PC as PRNG (ctd)

Entropy seeding (user input, physical sources) is continuously fed into the meta-PRNG

- Meta-PRNG has enormous state
  - ~100GB, counting all storage media
- Meta-PRNG has enormous complexity
  - Sum of all hardware and software in the system
  - Video, sound, network, disk, USB, system devices, etc etc, not just the CPU

Assuming the meta-PRNG can (somehow) be accurately modelled, attacking it requires a brute-force search of all system states based on given entropy seeding

- Forcing an attacker to resort to brute force is just what we want

## Win16 / Win95 / 98 / ME results

Entropy polling records absolute system state via ToolHelp32

- 99% compression of polled data
- Little change over time
- Minimally-configured machine produces half the entropy of maximally-configured machine

Unexpected behaviour on reboot

- 2½ times larger than static (no-reboot) samples
- 4 times larger than other samples taken after reboot
  - Drivers, devices, support modules, etc are loaded and run in somewhat random order
- Use reboot to destroy state

## WinNT / 2K / XP results

Polling records change in state over time via NtQuerySystemInfo

- Registry performance counters aren’t safe to use
- Time-varying data is less compressible than absolute system state data
  - ~80% compression rather than $99^+$% compression
- Same quantity of input from unloaded machine compressed to $^1/_{10}$ size of loaded machine data
  - With little running, there’s little spinning the kernel counters

Network stats provide almost no entropy

- ~200 bytes compress to only 9 bytes
  - The entropy estimation is quite effective here

# Unix results

Most sources record changes over time (e.g. `*stat`)

- Available on some systems via easier sources (procfs, kstats)
- Characteristics similar to NT / 2K / XP systems
- BSD-ish systems have more sources than SYSV ones
- Results depend on system load
- Reboot behaviour wasn't tested due to self-preservation considerations

# Entropy Polling Example

Entropy estimation process

- Poll entropy data → 40KB of results
- Reduce 10 : 1 (typical) via compression → 4KB
- Assume another 10 : 1 for inability of the model to capture the source characteristics
  - Good safety margin, English language is 2 : 1
- Result: 400 bytes (3200 bits/woozles) of entropy
- Like PSNR, this is only an estimate of goodness, not an absolute measure
- Engineer's rather than mathematician's approach

Only the more tractable OS sources were tested, not the ones based on physical/thermal entropy

## Use in the Field

Compression-based estimation is a lab test, not a field test

- “Please wait while your machine reboots several times…”

Use the lab results to guide field estimation

- Determine entropy for standard sources
- Assign weights to each source based on entropy
- Entropy is sufficient if total exceeds a set threshold

Example

- `netstat –an` produces *n* woozles of entropy per kB of output
- Weight = *m* × kB
  - – Source x provides y% of our requirements
- Entropy is sufficient if total from all sources ≥ 100

## Availability

Available in an open-source implementation

- BSD license or GPL (your choice)
- `http://www.cs.auckland.ac.nz/~pgut001/cryptlib/`, in the `/random` subdirectory
  - – Grab the one tagged “snapshot”
- Meets many/all of the requirements of the proposed standard

See chapter 6 of “Cryptographic Security Architecture Design and Verification” for full design details

# Usability Issues

Providing entropy for the RNG is hard to do right

- Developers can’t solve the problem, so they leave it as an exercise for the user
- Netscape disabled BSAFE safety checks in order to allow the generator to run without proper initialisation

# Usability Issues (ctd)

A simple safety check was added to OpenSSL 0.9.5 to test whether the generator had been properly initialised

An entry was added to the FAQ to explain this

- Later versions of the code were changed to display the URL for the FAQ

User responses…

- Seed the generator with a constant text string
- Seed it with DSA public components (which look random)
- Seed it with output from `rand()`
- Seed it with the executable image

*... more*

## Usability Issues (ctd)

*... continued*

- Seed it with /etc/passwd
- Seed it with /var/syslog
- Seed it with a hash of files in the current directory
- Seed it with a dummy “random” data file
- Seed it with the string “0123456789ABCDEF”
- Seed it with output from the (uninitialised) generator
- Seed it with “string to make the random number generator think it has entropy”
- Downgrade to an older version of the generator that doesn’t perform the check

*... more*

## Usability Issues (ctd)

*... continued*

- Patch the code to disable the check
- Later versions of the code added /dev/random support
  - Replace the /dev/random read with a read of a static disk file

Based on user comments, quite a number of third-party applications had been using low-security cryptovariables from the uninitialised generator

This is not specific to OpenSSL, it’s merely the best-documented case

- Similar advice has been given on bypassing the JCE RNG
- truerand-style RNG that takes while to run

## Usability Issues (ctd)

Crypto purists: If we have a way to evaluate entropy, the device should refuse to work unless sufficient entropy is available

Product developers: We can’t ship an otherwise fully functional device that simply refuses to function in the field

- 0.01% of users (ones with COMSEC training) will have the discipline to handle RNG failures
- 99.99% of users will see an RNG failure as a defective product

## Usability Issues (ctd)

If presented with a “defective” device, the user will

- Use someone else’s product
  - Preferably one that doesn’t warn of entropy problems
- Send in the clear
- Complain / threaten legal action

If an RNG failure appears as a defective product, you’d better make *very* sure that you never get an RNG failure

- “In insufficient entropy → halt” ⇒<br>“At random times, make product defective”
- Address via use of many entropy sources / fault-tolerant design

## Usability Issues (ctd)

Perhaps make it a speed-bump warning

- Warn conscientious users, but don’t fail completely for “just make it work dammit” users

Generate keys, but zero the top 32 bits as a warning to the other side

- “I’m doing the best I can with what I’ve got”
- Less secure than generating a full key if an attacker can tell if full entropy was available

ZKP of entropy state?

## Open Questions

Continuous testing

- How continuous?
  - Every *n* seconds?
  - Before every generation of cryptovariables?
  - Before generation of high-value cryptovariables?
- What if you need keys on a hard deadline?
- What if you’re on an embedded system?
  - No background tasks for continuous tests
  - Can’t afford to run background task for testing

# Open Questions (ctd)

## Entropy polling

- What if there’s nothing to poll?
- Persist state to disk / flash memory
  - randseed.xyz approach already used by many apps
- How to protect the state?
  - Attacker can read to determine past state
  - Attacker can write to affect future state

## Usability

- What to do if there’s insufficient entropy available?
- (Pies will be available in the cafeteria at lunchtime)