Analysis of a strong Random Number Generator

Thomas Biege <thomas@suse.de>

What is the Reason for this Work?

- fun and curiosity...
- ... and last but not least: gaining knowledge!
- for most people /dev/random is a black-box
- why should security-relevant open-source software trust a black-box?
- crypto randomness is a building block for authentication, crypto algorithms, and protocols
- that is why you should know about it’s inner working!

The Lab System

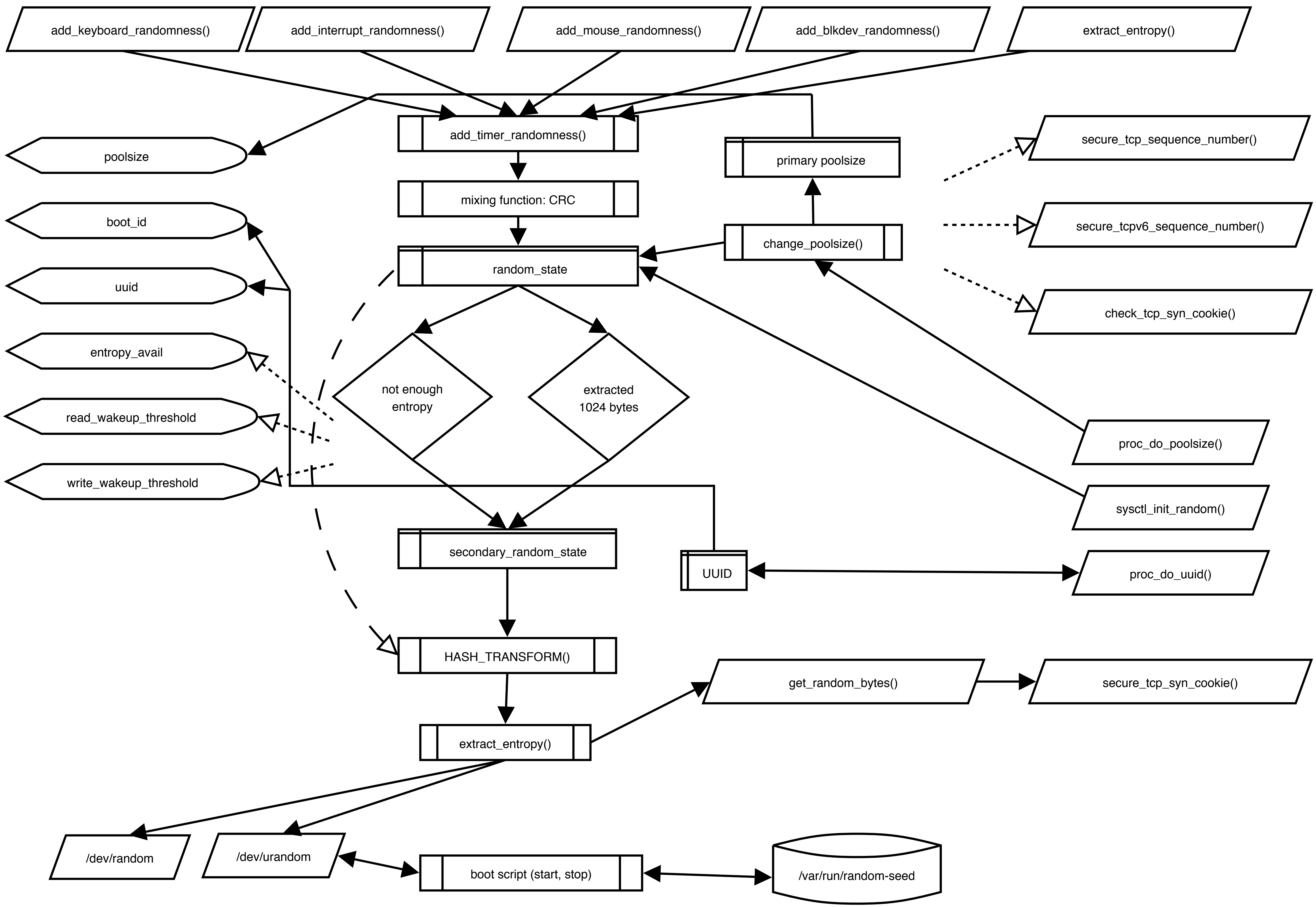
- old laptop with a 400 MHz CPU
- WLAN card for remote logging; <u>no HDD I/O trigger!</u>
- SuSE Linux 9.0 with a 2.4 kernel
- no mouse, no X, no user, no ..., but...
- ... apache + php
- patched kernel to gather observed data in a ringbuffer
- the pool initialization at boot time was disabled when needed

Focus of my Research

- behaviour of data, not of the algorithms
- entropy of events during boot sequence
- entropy consumers
- influences of determinitsitc/malicious entropy sources

The PRNG Design

The Map of Chaos



Entropy Sources

- block-device access
- interrupt occurence
- keyboard typing (desktop system)
- mouse movements/button usage (desktop system)
- (pool extraction)

Entropy Input

- *add_timer_randomness()* measures the event timing $\longrightarrow$ entropy (2 * 32-bit TSC register)

$[0,255]$	keyboard scan codes
$[256,511]$	interrupt number
$[512,\mathrm{UINT\_MAX}]$	block–device major number
$[0,\mathrm{UINT\_MAX}]$	mouse movements
$[0,\mathrm{UINT\_MAX}]$	pool extraction

Entropy Esitmation

- the current and the last two time-periods are used to estimate the entropy
- it's hard to do it right!
- Shannon's entropy:

$$H(S) = \sum_{i=0}^{n} p_i \cdot log_2(p_i)$$

The Pools

- primary pool for data collection (4096 bits)
- secondary pool for extraction (1024 bits)
- 2nd pool is reseeded from 1st pool
- pool contents are mixed with a *Twisted General Feedback Shift Register* (TGFSR) to spread entropy equally in the pool
- pool is implemented as a ringbuffer

Extraction

- /dev/random: blocks until enough entropy is available
- /dev/**u**random does not block but iterates over an initial seed with SHA-1
- pool bits are hashed, 32 bits of the digest are fed back into the pool, the folded half is returned to the caller
- timing is used as entropy source (with entropy 0)
- entropy bits will not be removed, only the estimator is decremented

The Idea!

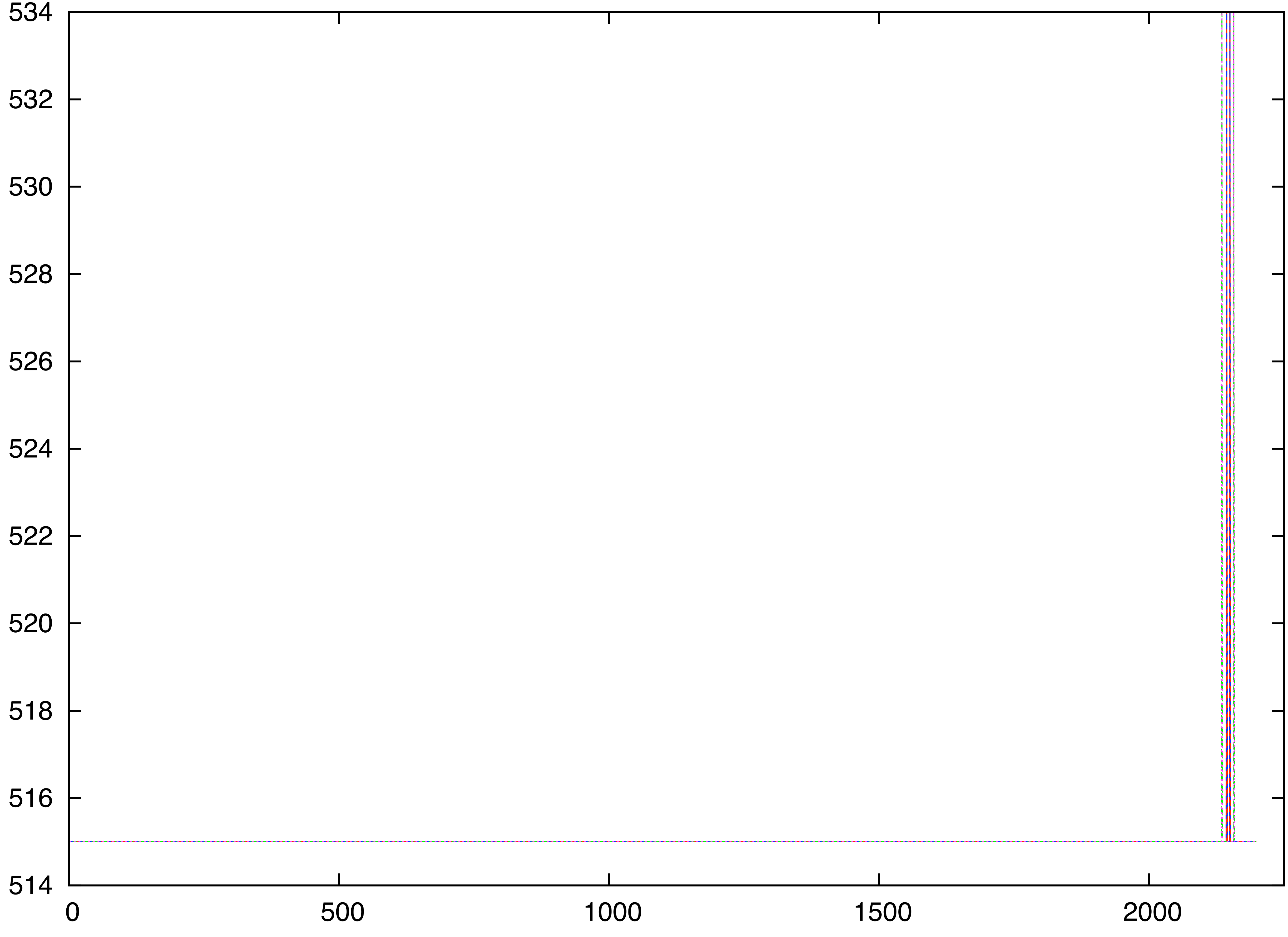
- every algorithm used is deterministic
- **idea:** when the input data is deterministic too, the result can be guessed in a much shorter time
- **proof**: code execution during sytsem boot may be deterministic... or may be not.
- **result 1**: random numbers are not as random as assumed
- **result 2**: crypto systems based on this wrong assumption become weak

Analysis of Entropy Sources

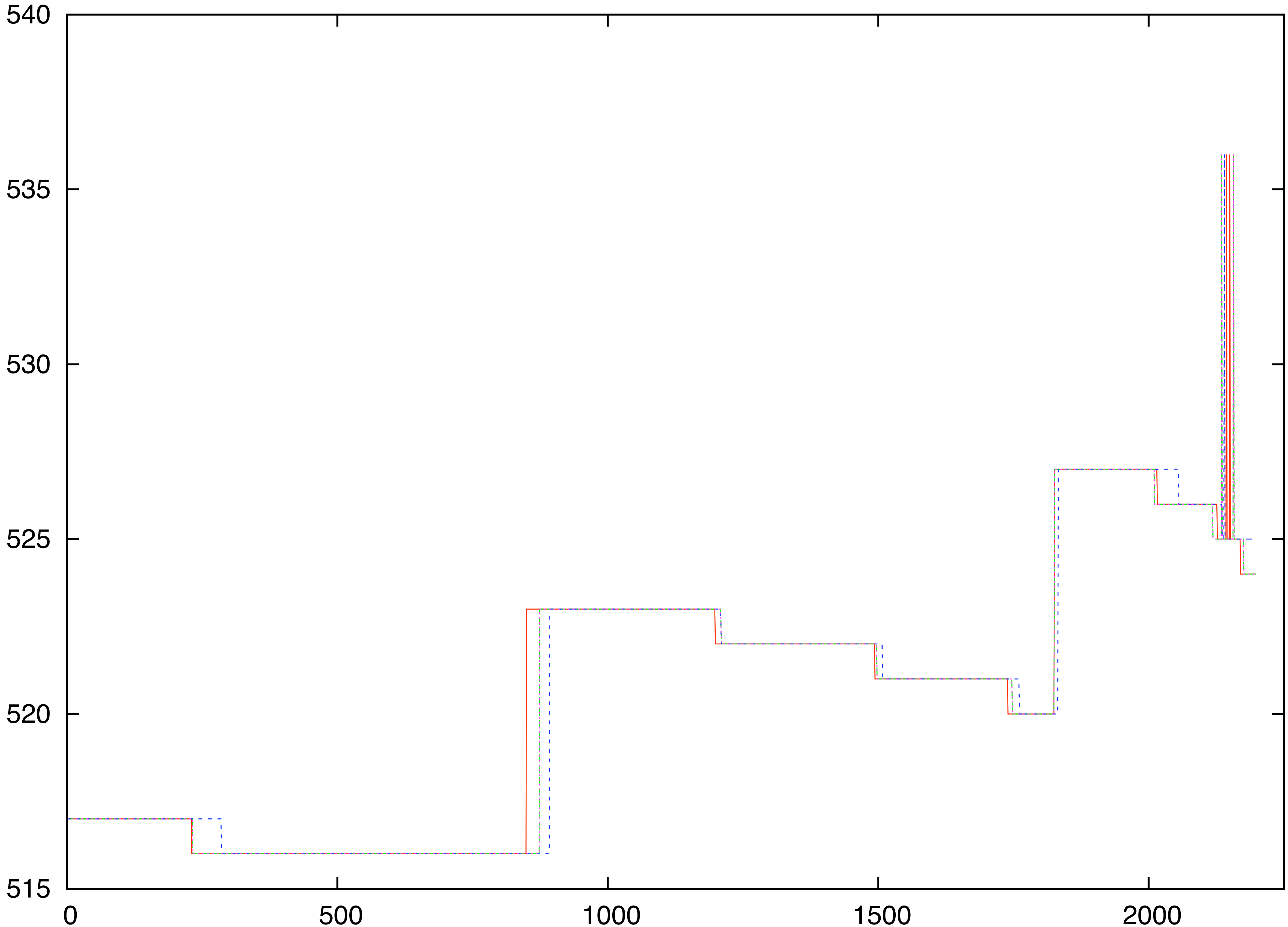
-

Plot the Facts!

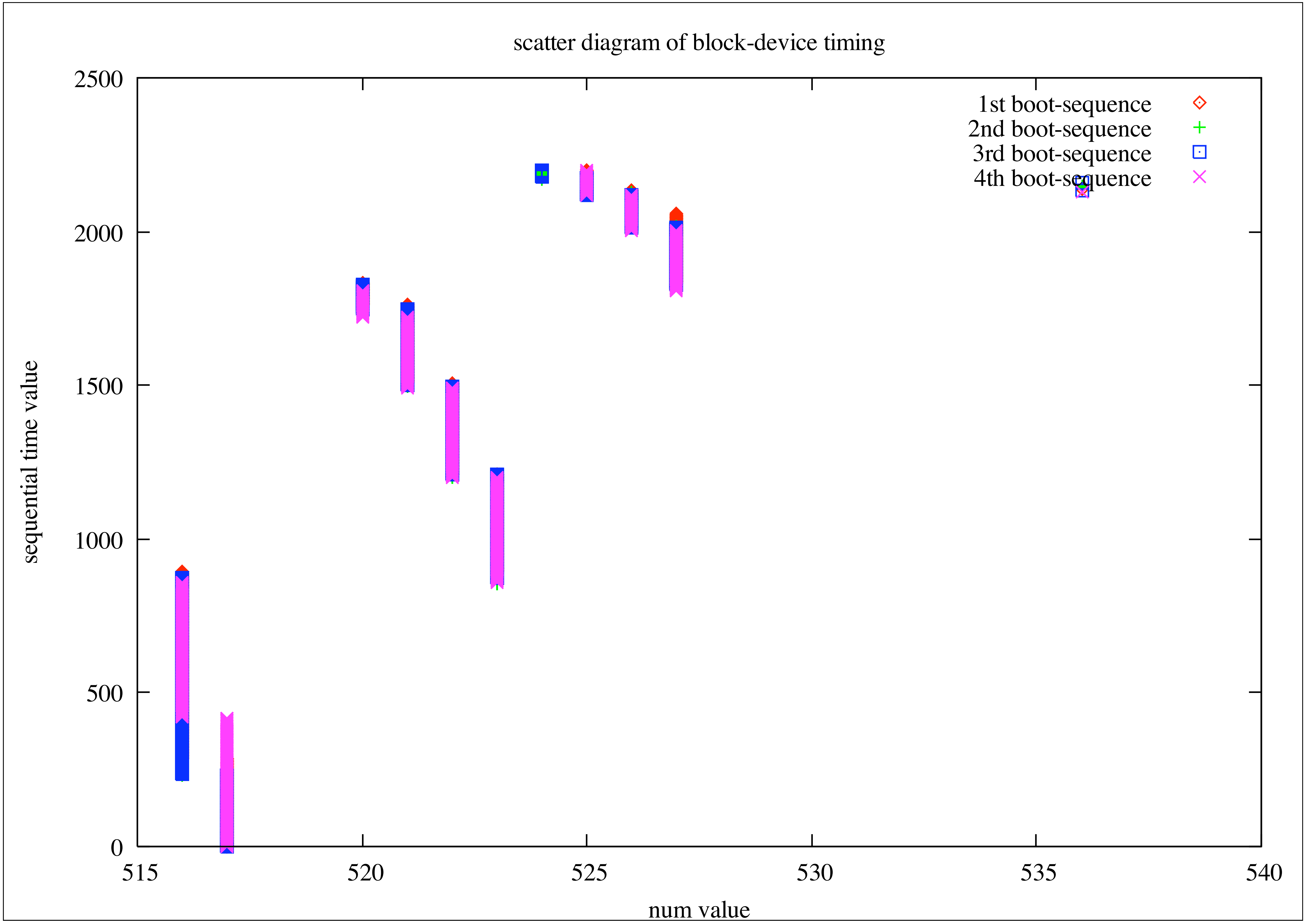
Block-Device Type



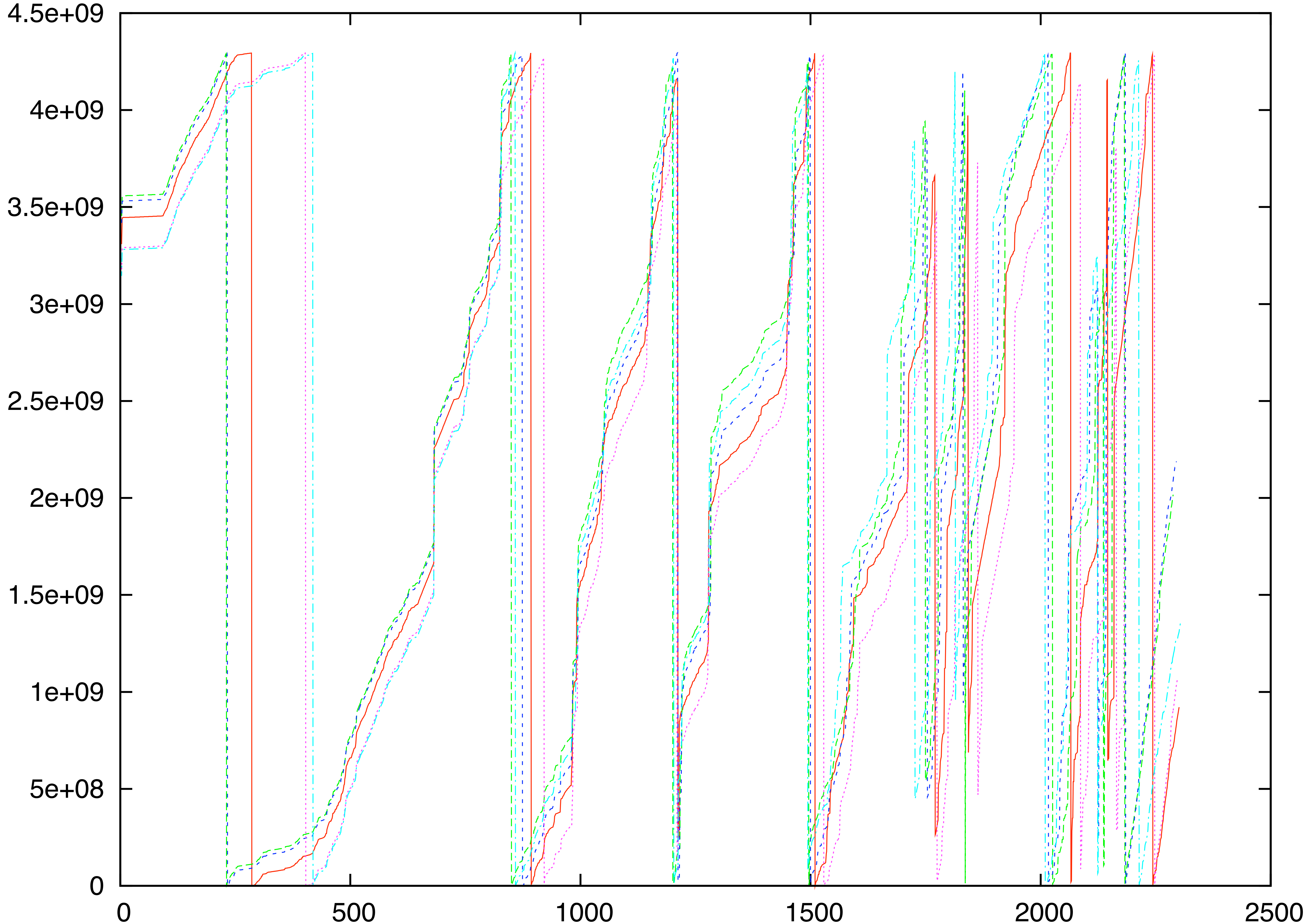
BD-Type XOR'ed with TSC



Same, Axis swapped



Block-Device Timing



result: input behavior is very identical between different boot sequences, even timing!

assumption: entropy during boot process is very low, lower then estimated

let’s check it...

Analysis of Entropy Sources
-
Let Statistics speak!

Oops! Entropy Over-estimation

	Estimated	Calculated	Deviation
Entropy	8	5.77541	2.22459
Mean	127.5	66.91920	60.5808

- entropy overestimation is dangerous for CPRNGs
- a 128-bit key may have less then 90 bits of entropy

Deviation Calculation

	Average	Variance	standard Deviation
num_orig	250	0	0
num	245	0	0

Table 9: Date 14. Oct, process: `hwscan`, source: mouse

	Average	Variance	standard Deviation
num_orig	250	0	0
num	244	0	0

Table 10: Date 15. Oct, process: `hwscan`, source: mouse

	Average	Variance	standard Deviation
num_orig	250	0	0
num	244	0	0

Table 11: Date 16. Oct, process: `hwscan`, source: mouse

Auto-Correlation

	Average	**Variance**	**standard Deviation**
num_orig	250	0	0
num	282.125	14234.9	119.31
time	2.28795e+09	2.77319e+18	1.66529e+09

Table 12: 1st event, process: `hwscan`, source: mouse

	Average	**Variance**	**standard Deviation**
num_orig	250	0	0
num	282.125	14234.9	119.31
time	2.29927e+09	2.77980e+18	1.66727e+09

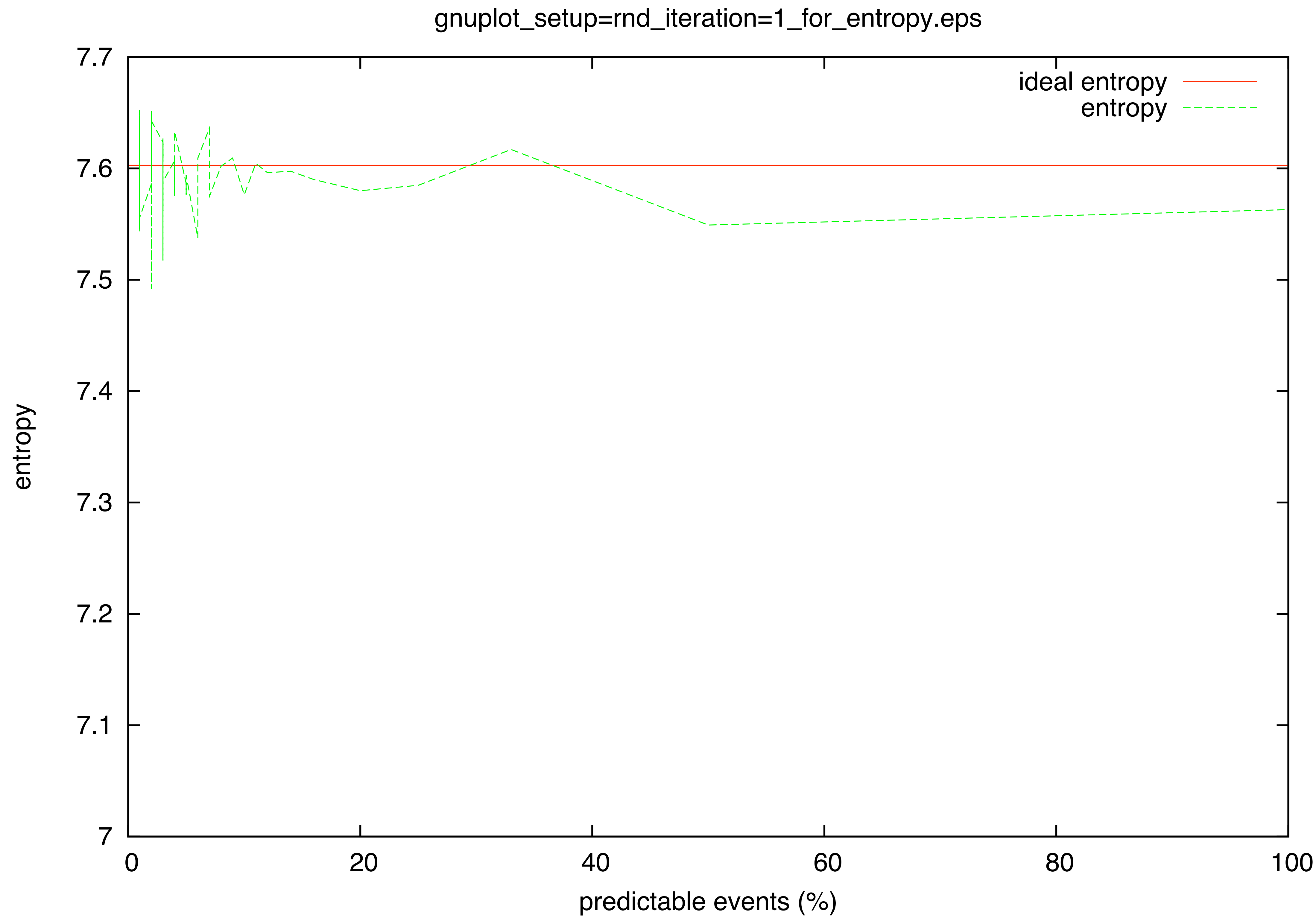
Table 13: 2nd event, process: `hwscan`, source: mouse

result: now we had prooven that input behavior is very identical between different boot sequences

assumption: maybe an attacked system can be “cloned” to get a clue about the CPRNG state

Untrusted Entropy Sources

Low-Quality Source



Malicious Source

LFSR's "pathological state":
all values are zero = output will be zero forever!

a known previously added value can be
"neutralized" and become 0:
TGFSR(0x0000000F, 0xF34015C4) = 0

result I: low-quality source can not dilute existing entropy

result II: a malicious source can put the LFSR into its "pathological state"!

improvement: add weights to input sources based on simple statistical tests

Entropy Consumers

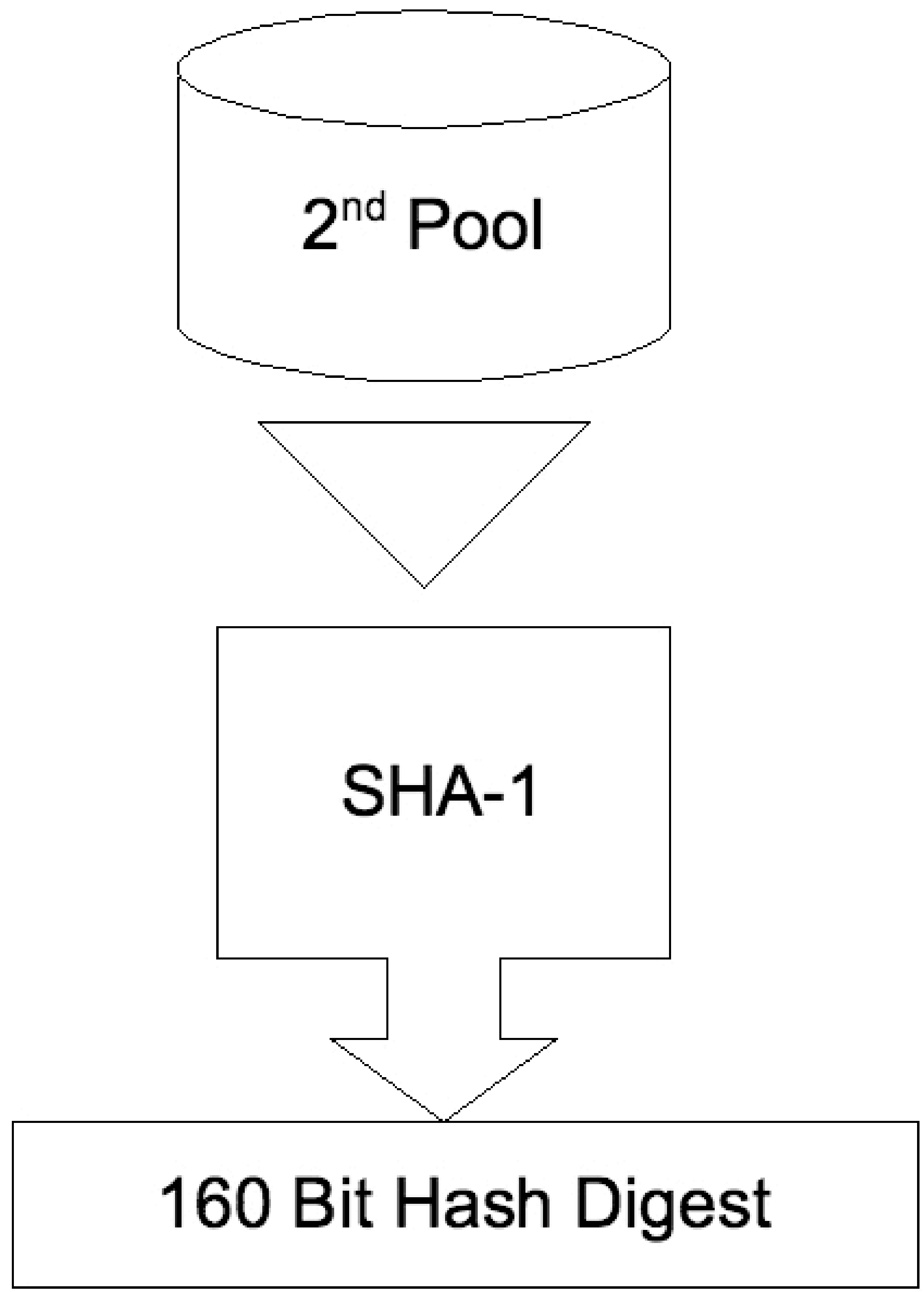
syncppp.c	uses two times 4 random bytes to generate a sequence-number as well as 'magic' values for a WAN interface
smbencrypt.c	8 bytes of random nonce for client authentication and 516 bytes of randomness to just fill a buffer which is used later to store a password. `make_oem_passwd()`, `encode_pw_buffer()`
ip_fragment.c	4 bytes for hashing The function `ipfrag_secret_rebuild()` will be called regularly (every 600 Hz) to update a hash–table.
ip_conntrack_core.c	Netfilter connection state tracking module consumes 4 random bytes per connection to initialize a hash–table. `init_conntrack()` called by `resolve_normal_ct()`
syncookies.c	36 bytes of randomness are used by calling `secure_tcp_syn_cookie` the first time generating IPv4 *SYN–Cookies*. Called in `tcp_ip.c` by `tcp_v4_conn_request()` and `cookie_v4_init_sequence()`
tcp.c	uses 4 bytes for hashing in `tcp_listen_start()`
irlap.c	consumes 4 bytes twice to create a random address.

The Pool is Dripping

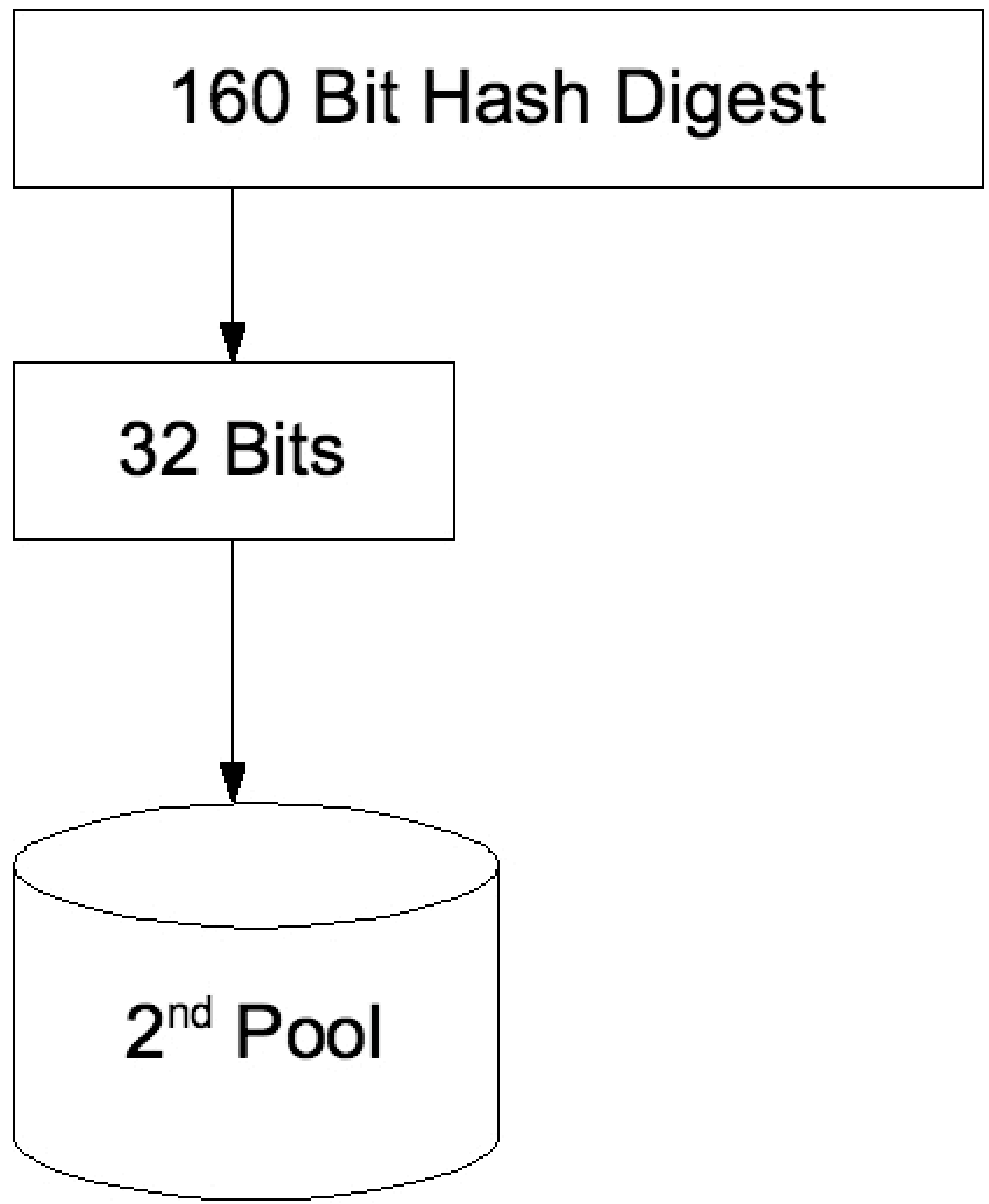
-

Entropy Bits are leaking during Pool Extraction

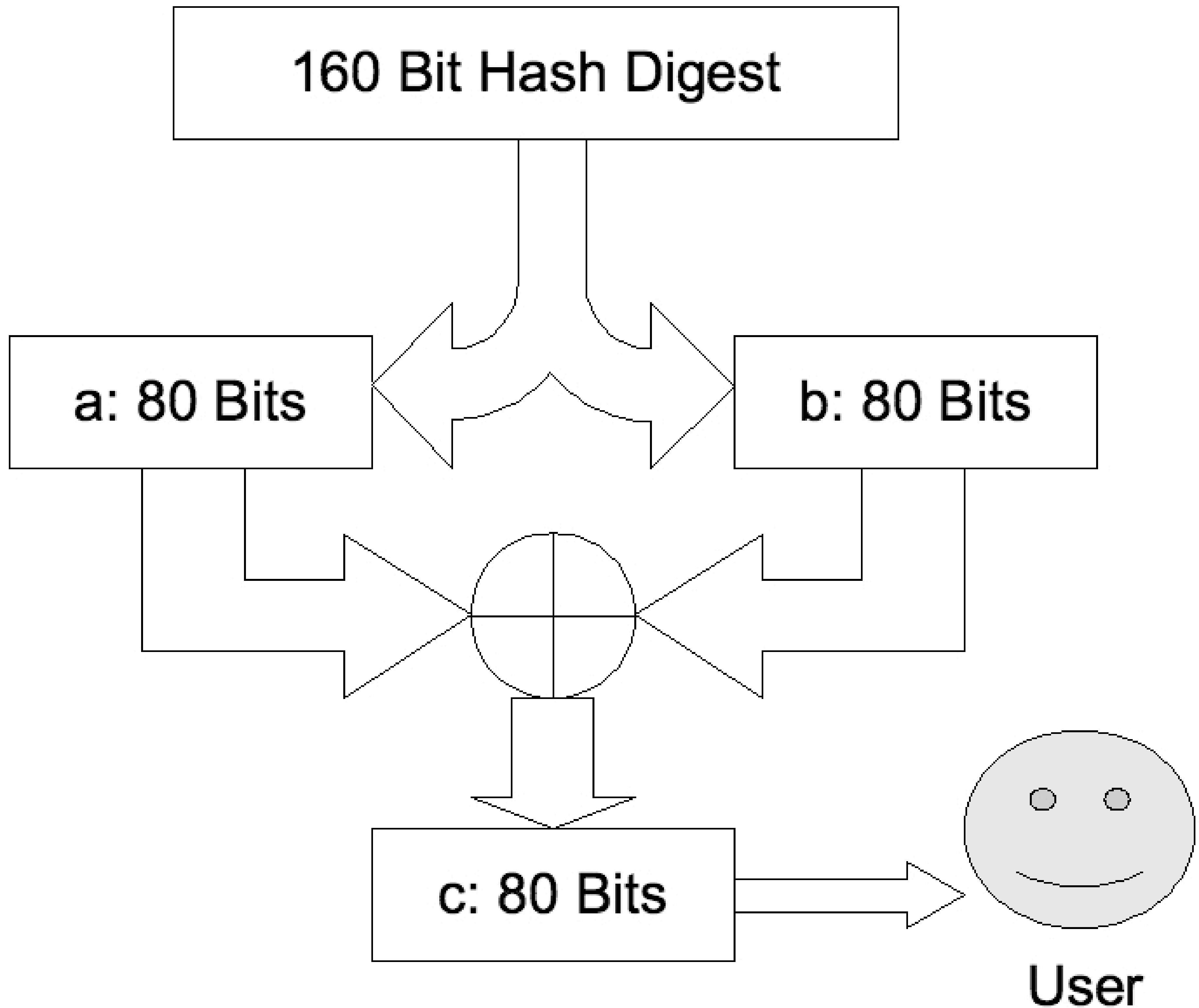
Feedback: Pool Bits hashed



Feedback: 32 Bits go back



Feedback: Fold and Return



Guesswork

to guess the last 32 bits written to the pool an attacker needs 2^{32} steps... no suprise. ;-)

due to the equal distribution of 1s and 0s in a hash digest the <u>search-tree can be reduced by factor 7</u>

So What?

entropy overestimation =
ex. SSH host keys generated during system installation are weak

malicious source =
entropy sources need verification/control

identical input =
systems could be cloned to guess CPRNG state

leakage =
a serious problem?

Questions?

Thomas Biege <thomas@suse.de>

http://www.suse.de/~thomas/