Analysis of a strong Pseudo Random Number Generator

by anatomizing Linux' Random Number Device

Thomas Biege <thomas@suse.de>

November 6, 2006

Abstract

Almost all cryptographic protocols depend on random (unpredictable) values to create keys, cookies, tokens, initialisation vectors, and so on. The Linux (as well as other Unix flavours) kernel provides a character device as a source for randomness. This device represents the essential part needed by various cryptographic protocol implementations for a secure operation (*conditional security*), therefore it needs special attention from security experts.

This paper will give an extract of results taken from analysing the input sources used by Linux' PRNG implementation. The statistical entropy of each source and of the whole pool is calculated to get a better picture of the entropy quality during the boot–process and to spot entropy overestimation by the kernel. Observation taken by process show a repeating behaviour for different system startups. This can be used by an attacker to create profiles and to simulate a more complex system. Even observations of the events generated by the block-device show timing patterns between different boot–sequences. To dispel doubts of developers to add untrusted sources, two kinds of untrusted sources, low-quality and malicious source, were examined. It will be shown that low–quality sources are not able to reduce the entropy in the pool that already exists but can lead to an overestimation. A more dangerous situation exists with the presence of a malicious source which is theoretically able to led the mixing algorithm produce a stream of zeros.

The goal of this work is not to show a practical attack against the random device but to provide more transparency and to ease further analysis.

Acknowledgements

I would like to thank[1] Marc Heuse, Olaf Kirch, Konrad Rieck for proof-reading and discussion.

[1] in alphabetical order

Anyone who considers arithmetical methods of producing random numbers is, of course, in a state of sin.

— *John von Neumann*

1 Anatomy

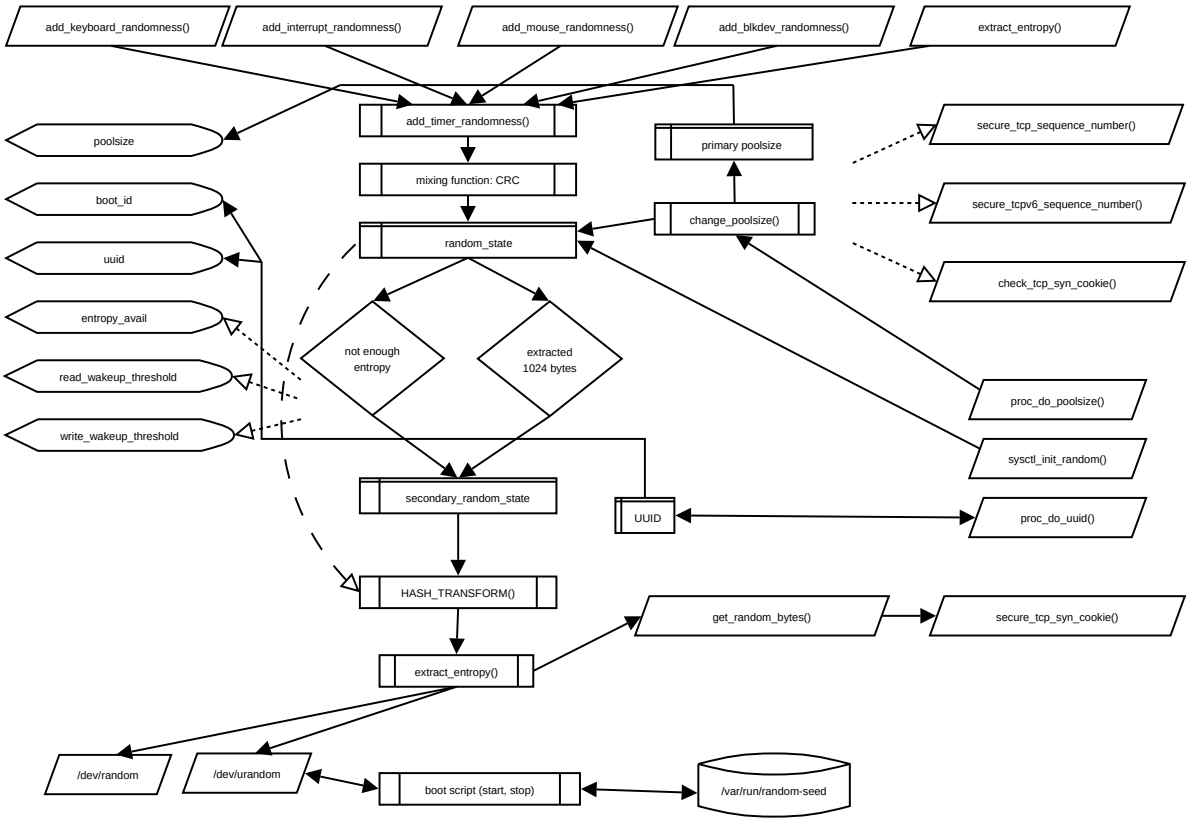

Figure 1: Flowchart of random–device implementation.

The architecture is based on the PRNG of PGP 2.x and PGP 5.x [11] but does not use a hash–algorithm for input mixing because it is too time consuming for a kernel–based approach. Instead mixing is done with a *Linear Feedback Shift Register* and a hash–digest is calculated during extraction of entropy.

1.1 Input Sources

In order to not fall victim of the aforementioned popular quote by John von Neumann the Linux kernel tries to gather random events from five different sources which are not the result of a numeric algorithm but are based on system– and user–behaviour:

- block–device access
- interrupt occurence
- keyboard typing[2]
- mouse movements and button clicking
- pool extraction feedback

[2]Ignores auto-repeat.

The first two sources are mostly triggered by deterministic processes and may also be influenced remotely by an attacker.

The next two sources are based on user–interactions which are considered to be non-deterministic. Unfortunately these input devices are usually not available on a server. Additionally, user input is not always non-deterministic. For example typing passwords can be reconstructed by measuring the transmission–timing of encrypted packets (only for interactive protocols) [7]. Mouse movements may also be deterministic (for a short time–period) because people tend to behave the same way to accomplish repeating tasks. For example when the graphical user interface (GUI) starts up the mouse pointer is adjusted to the centre of the display, then the user moves the mouse pointer to the eMail client application icon on the desktop, double clicks the icon, waits until the application is running, and then moves the pointer to the application's receive/send icon and clicks it. These steps may be the first interaction between the user and the computer every day. User behaviour can also be learned by using techniques from host-based Intrusion Detection Systems (IDS)[3] [8, 9a, 9b]. But this goes far beyond the scope of this paper which is limited to the booting–phase.

The last event, the feedback during extraction, is triggered by deterministic sources (system) as well as non–deterministic sources (user).

Before the input bits are stored in the pool they were mixed with the existing pool content by using a *twisted General Feedback Shift Register* (TGFSR) algorithm [6a, 6b]. To avoid leaking knowledge about the internal state of the pool to the outside and to give away statistically random values only (even if the pool content is not random), the entropy gathered will be mangled further during extraction by using the *Secure Hash Algorithm* (SHA) [1].

The user–land applications can access the entropy–pool by reading from `/dev/random` or `/dev/urandom`. The first interface blocks until the requested amount of entropy bytes is available. The latter one just generates as much bytes as requested by re–computing SHA from an initial random value.

To initialize the random–pool at system startup the kernel adds the result of `gettimeofday()` and the contents of `utsname`. Additionally the previous state of the pool is restored by using a *rc–file* (boot–script). The "start part" writes 512 bytes of a previously saved random–pool state back to the current pool and writes an equal amount (if available) back to the disk. This is useful to avoid initializing the pool with the same state a second time if the system crashs and to remove the already used state from the file–system. The "stop part" just saves about 512 bytes to the disk. Both parts use `/dev/urandom` to not block the boot or shutdown process to the expense of not catching real entropy.

[3] Even on machines other than the one which will be attacked later.

1.2 Event Timing

All entropy source handling functions are calling `add_timer_randomness()` to add timing randomness to the entropy–pool. Therefore a closer look at this function will be provided in this sub–section.

Beside the time delay it also adds a `num` value to the pool which describes the source of the event. This value is encoded in the following way:

[0,255]	keyboard scan codes
[256,511]	interrupt number
[512,UINT_MAX]	block–device major number
[0,UINT_MAX]	mouse movements
[0,UINT_MAX]	pool extraction

If the CPU architecture is x86_64 or i386 with *Time Stamp Counter* (TSC) register support the assembler macro `rdtsc()` is used to get a high–resolution timestamp. This counter is incremented each clock tick[4].

The high part (the one that changes less often) of the timestamp is XOR'ed with `num`. The low part of the 64 bit register is copied to the `time` variable. On other architectures without TSC support the `time` variable is set to the current *jiffies* value and the `num` value is left unchanged.

To calculate the estimated entropy for each type of event, the function determines the current and the last two time delays, then the `delta` variable is set to the minimum absolute difference, rounded down by one bit, and then truncated to a 12 bit value (from 0 to 4095). This value is then used as source for the estimation calculation by using *Euler's summation formula* aka *Euler-Maclaurin formula*[5] to sum up the `delta` bits.

Finally the values are threaded in a queue which is emptied every tick using the function `batch_entropy_store(num, time, entropy)`. The value of `num_orig` can be considered mostly constant, all the randomness of `num` comes from the XOR operation with the high part of the TSC register. Unfortunately the upper 32 bits of the register increment relatively slowly (depending on the CPU speed) and will therefore not change `num` value often enough (see section 2.2).

1.3 Primary and Secondary Pool

The implementation uses two pools, a primary and a secondary pool. The primary pool is used to store new entropy values and the secondary pool is used for extraction. The secondary pool is reseeded from the primary one whenever 1024 bits (size of the pool) were extracted or when not enough entropy (as requested) is in the 2nd pool.

[4]The example system's clock ticks with 400 MHz, which means 1 increment each 2.5 ns.

[5]Which is an approximation of the sum. [2a, pp. 111]

The pool content is mixed using a *twisted General Feedback Shift Register* (TGFSR) [6a, 6b]. The following *tap sequences* are available (pool–size in the left column):

2048	1638	1231	819	411	1	0
1024	817	615	412	204	1	0
512	411	308	208	104	1	0
256	205	155	101	52	1	0
128	103	76	51	25	1	0
64	52	39	26	14	1	0
32	26	20	14	7	1	0

The primary pool uses the *tap sequence* for length 128, the secondary pool uses the *tap sequence* for length 32.

When more than 1024 bits are read from the secondary pool an emergency reseeding will happen to transfer entropy from the primary pool. In the opposite case when the entropy reaches its maximum for a pool the input value will be written in the other pool. If both pools are full, entropy will only be added with half speed.

1.4 System Setup

For sample gathering an i386 Laptop (Celeron 400 MHz, 3GB HDD, 128 MB RAM) with SuSE Linux 9.0 (minimal Software selection), apache2, mod_php4 [6], and a patched 2.4.21 kernel was used.

The kernel needs to be patched to insert a hook in the `add_timer_randomness()` function. The hook logs the process–ID, process–name, jiffy, entropy source, and various other interesting values into a kernel memory buffer. Additional hooks were added to the initialization functions of the random driver to record the *jiffy* value when the random device becomes active in the kernel code. This value can be used as reference for timing analysis but was ignored in this work. Another patch was introduced later to copy the pool whenever the entropy counter reaches its maximum.

The following daemons are started during boot:

- apache2
- cardmgr
- cron
- klogd
- master
- mingetty
- nscd

[6] Installed to keep in touch with a realistic setup.

- pickup
- portmap
- qmgr
- reader
- sshd
- syslogd

Recording the entropy generated by the different events faces the problem that observing a system state influences the system itself. Therefore a big enough kernel–buffer was used that was able to hold about 30.000 events. The contents of the buffer were read by a daemon process from a character device and forwarded to another host via the network. Additionally it was assumed that the measurement inaccuracy is constant over different system runs and does not influence the statistical evaluation in a significant manner.

The recorded entropy values were stored in a file on a remote system and were post–processed by several programs to extract, sort, and format the needed values for the different tests. For example files for the processes running at boot–time as well as for the various input sources were generated. The following file (`hwscan_10_14_1097735021.tab`) content was generated by process `hwscan` running at the 14th of October:

```
pid pname jiffies event source num_orig num time time_diff delta entropy
988 hwscan 86470 mouse 0 250 245 43671843 0 145 7
988 hwscan 86472 mouse 0 250 245 56437833 12765990 1427 10
988 hwscan 86473 mouse 0 250 245 57643113 1205280 528 9
988 hwscan 86474 mouse 0 250 245 58844433 1201320 1980 10
988 hwscan 86475 mouse 0 250 245 60047517 1203084 882 9
```

Files like this are used as source for stochastic and entropy calculations which will be shown and explained later.

This setup does not use a keyboard or a mouse, therefore these two entropy sources did not play a role in the analysis[7].

2 Input Analysis

The input values (the seed) of a PRNG is one of the most important parts in the process of generating unpredictable output values.

The Linux kernel uses a cryptographic hash-algorithm (SHA-1) to create a random looking output value. Normally it is not feasible to attack the hash value other than by guessing the input value (brute–force, average 2^{n-1} steps)[8]. But if the attacker knows some of the input values (by observing a cloned system, for

[7]But they appear because the device–drivers are loaded and used by `hwscan`.

[8]The latest successes in advanced attacks against SHA-1 [10] can be ignored for this work.

example) or just their variance from a constant value, it should be possible to reduce the search space a lot.

By recording the input values used by the random device during several boot–sequences it was possible to determine the entropy gathered in more detail. The results show that almost all values are equal or even constant between several boots. This fact makes clear that generating, for example, SSH host–keys during the installation process of an operating–system is a bad idea because the values used (20 bytes from `/dev/urandom`, time, PID) are highly predictable.

2.1 Sources

As already mentioned above the random–pool is fed with input values from sources like block–devices, the keyboard, the mouse, and other triggered interrupts.

Unfortunately the only two sources of real entropy due to their direct user–interaction are the mouse and the keyboard. Both are not available every time, for example they are not present in server setups. Even if a mouse is attached to the system the corresponding events are not recognized as long as the device is not used by an application like X or `gpm` (device–driver loaded).

The behaviour of block–devices and interrupts can be influenced remotely by sending/receiving data to the system (HTTP, FTP, POP3, ...) for example.

Some events generate values that are limited to a small range of possibilities by nature. The `num_orig` value[9] of a block–device, for example, is the major–number of the device.

The utilization of the functions provided by the random device in the Linux kernel 2.4.21 has some odds. For example some architectures (see the `arch` subtree) lack the hook for `add_interrupt_randomness` in their IRQ handling functions for an unknown reason. No mechanism is in place to verify that the corresponding functions are used in every device–driver for every mouse type, keyboard type, and/or hard drive, everything is based on good will. Furthermore the arguments, which are used as input values for the entropy–pool and passed over to the add-randomness functions, are not standardized. Several device–drivers use different kinds of bit–operations, bit–shifting, and even different values to feed the entropy–pool. This behaviour makes analysis harder for practical attacks because the attacker needs to know the device types used by an user. But in some cases it even ease the work of an attacker because the arguments are values from sources of less entropy.

[9] `num_orig` is the unmodified value, `num` is the result of XOR(`num_orig`, `time`)

2.2 Characteristics of Input Values

The following plots show the values (major number, major number XOR'ed with 32 bits from the TSC upper register, and the timing (TSC lower register)) generated by the block–device during system startup. Each plot contains the graphs of four different boots. The plots do not need much explanation. In general most graphs are very equal (the pointed graphs make it even more clear) and show the predictability of these values[10]. The x–axis is the sequential number of the event occured and the y–axis is the value measured at this time.

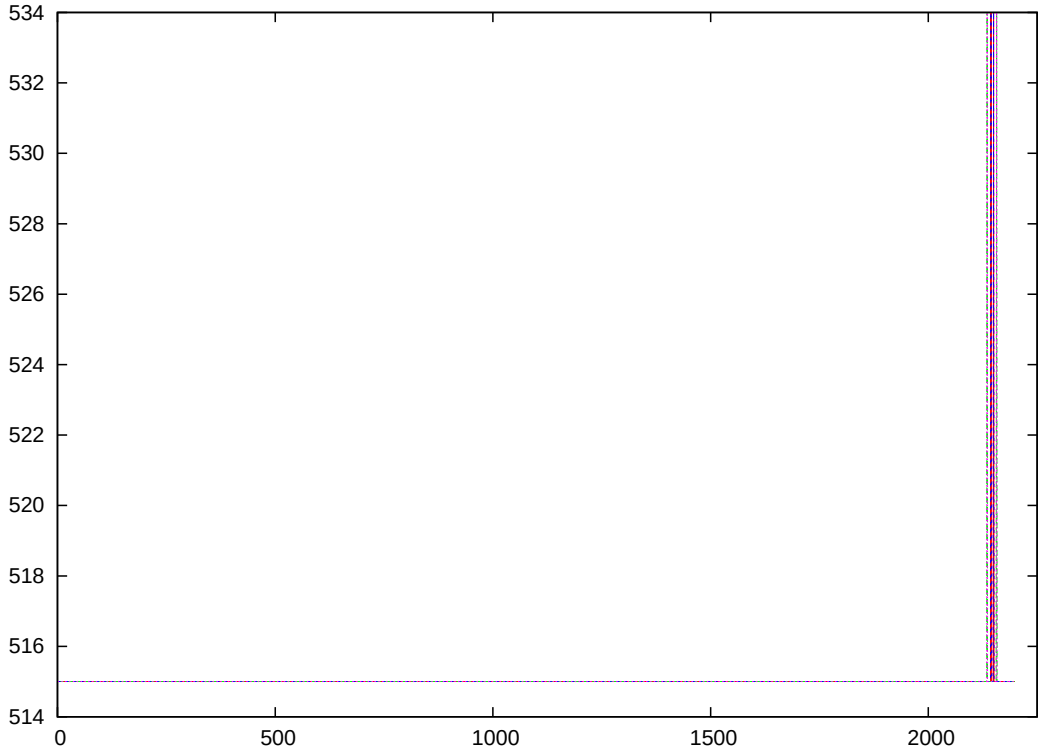

Figure 2: Block–device events of different boot–sequences.

[10]Please note that the values wrap around when they reach 2^{32} and start again from 0. For example, when the lower part of the TSC register wraps around, the upper part gets incremented by one and the low part starts again from zero.

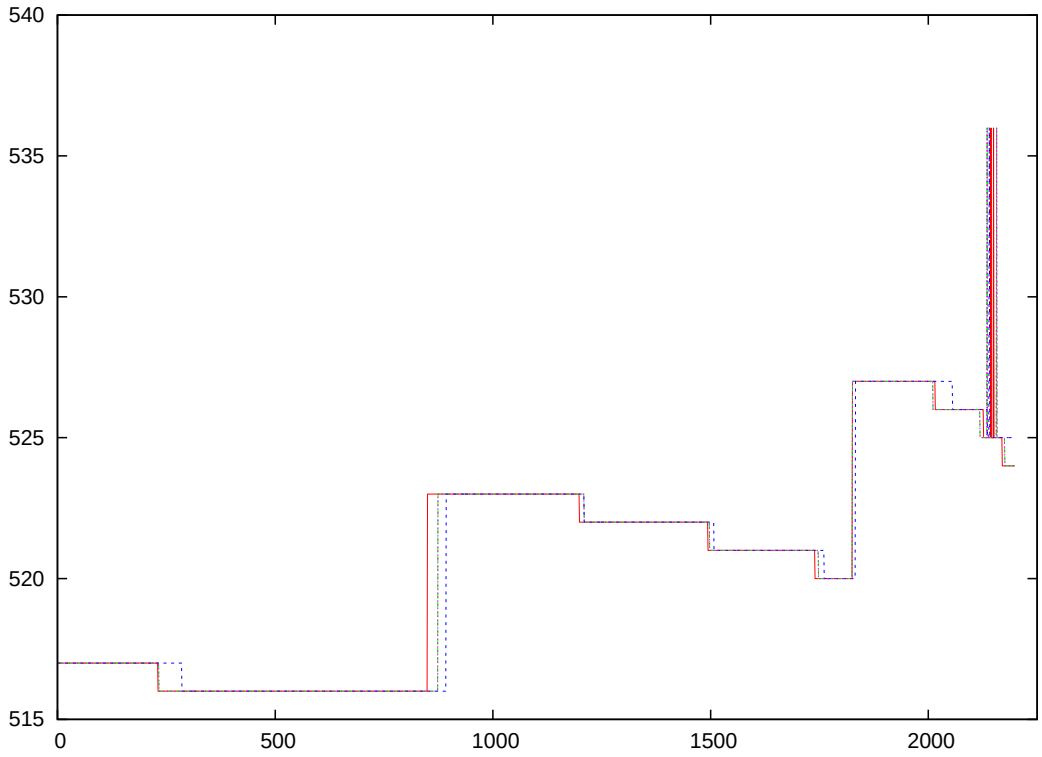

Figure 3: Block–device events (XOR'ed) of different boot–sequences.

Figure 2 shows the block–device actvity during boot (2200 events). This values are almost constant (each graph overlays another) and even the modified values repeat during boot (figure 3). The steps in figure 3 correspond with the wrap–arounds in figure 4. The lower register wraps around and the higher register is incremented by one. The higher part is XOR'ed with the `num_orig` value which results in a new `num` value (the step). The steps in the graph are clustering around the same event number and have the same height. This is an indicator for the repeating timing during independent observations.

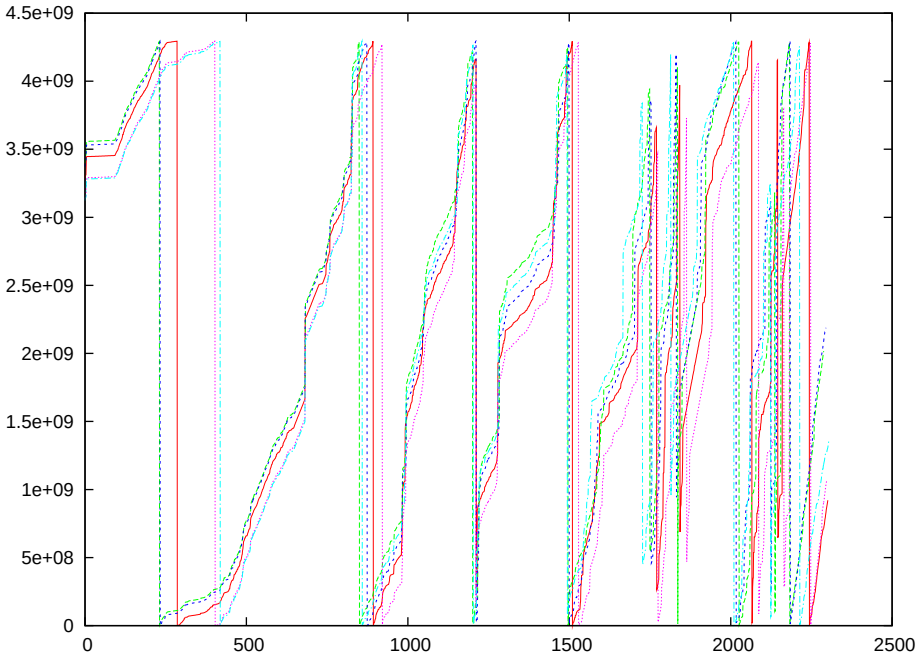

Figure 4: Block–device events of different boot–sequences.

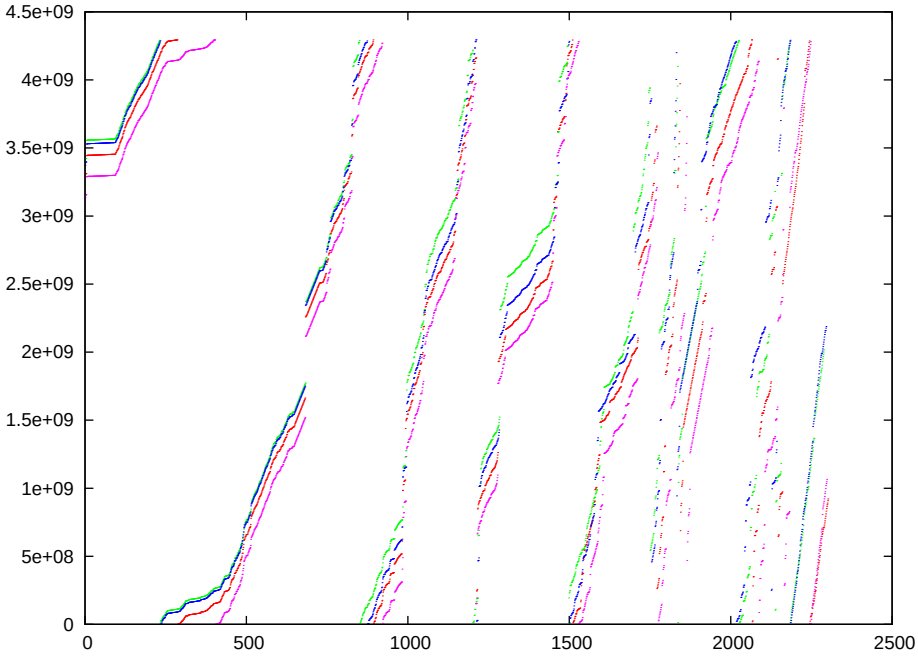

Figure 5: Block–device events of different boot–sequences.

The graph of the timing of the block–device events looks less predictable (figure 4), but the dotted variant clearly shows patterns (figure 5).

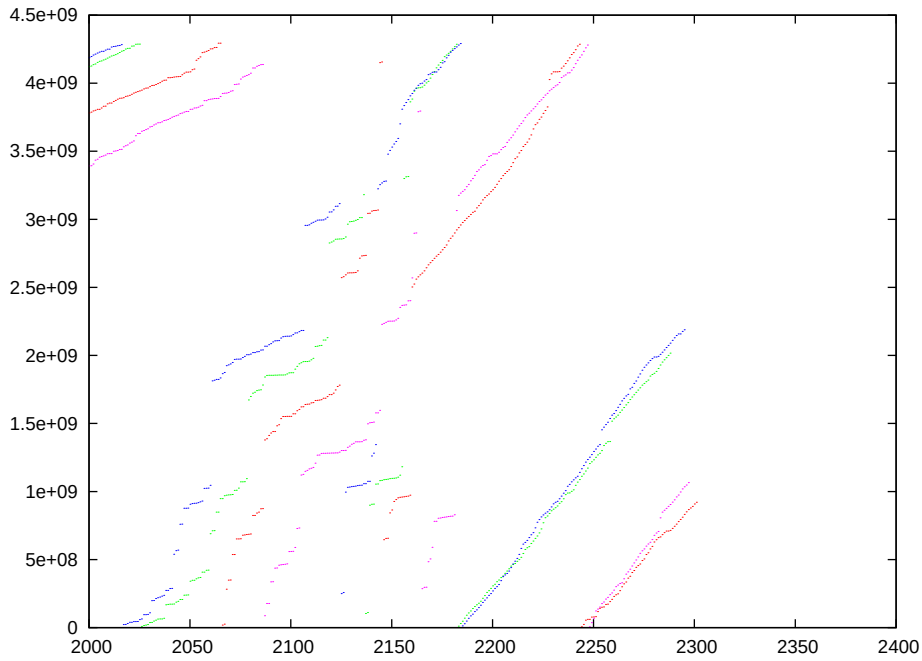

Figure 6: Same event timing as above with limited x-range ($\geq$ 2000).

Starting from event 2000 the timing values seem to be more erratic. A closer look with a limited x–range shows patterns even here (figure 6).

2.3 Probability and Entropy

To get a detailed view of the (statistical) entropy generated, the events and their values (`num_orig`, `num`, `time`) are measured applying a formula based on [3]. Before the results are presented the mathematical basics will be summarized.

A discrete information source S without memory is defined by its symbols s_i and their corresponding probability distribution p_i.

$$S = \begin{pmatrix} s_1 & s_2 & s_3 \cdots s_n \\ p_1 & p_2 & p_3 \cdots p_n \end{pmatrix}$$

The entropy is defined as

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

with unit bit/symbol. H(S) only depends on the probabilities of the symbols not of the symbols itself. Therefore we can also write it as H(p), H(p_1, ..., p_n), or just H. Additionally it is important to find the right probability distribution. For the sake of simplicity the relative frequency was used, which is defined as

E = number of times the event happened / total number of observations made

The sum of the entropy of two different independent sources is:

$$H_{sum} = H_x + H_y$$

A discrete information source without memory containing n symbols has the highest entropy when all symbols have the same probability. This entropy is named H_0 or H_{max}. In other words if $P(s_1, \ldots, s_n) = 1/n$ (uniform discrete distribution, or Laplace-distribution), H_{max} is reached. The system observed here does not follow the rules of a Laplace-experiment because the probability distribution is not uniform. Otherwise the random device would generate real random numbers.

For this work the probability of each event is considered independent, which is a valid assumption but not true in reality. The system–calls (which trigger the input sources) used by the software have a successive dependency, otherwise a program would be useless and makes no sense. With this fact in mind it is possible to calculate more accurate and less uniform probabilities for events measured per process[11] and for the complete pool. Conditional probability $P(x|y)$ or n-th order Markov models can be used for it.

The following tables list the results of the probability and the corresponding entropy H calculated for the values generated by the mouse driver. The mouse driver is triggered by the `hwscan` process during hardware probing. It produces 5 values (total frequency), each value has its own relative frequency (number of occurence of the same value), and the resulting probability (total frequency devided by relatice frequency, E).

value	**total frequency**	**relative frequency**	**probability**	H_{value}
250	5	5	1	0

Table 1: Date 14. Oct, value: `num_orig`, source: mouse

value	**total frequency**	**relative frequency**	**probability**	H_{value}
245	5	5	1	0

Table 2: Date 14. Oct, value: `num` (XOR(`num_orig`,TSC)), source: mouse

Table 2 shows a constant value because the TSC register does not change fast enough.

The `hwscan` process is just one of the worst examples. The block–device activity produces more entropy but are not very random at all.

[11] Existing solutions for host-based IDS may be used for this [8, 9a, 9b].

value	total frequency	relative frequency	probability	H_{value}
515	2200	2199	0.99954500	0.00065562
534	2200	1	0.00045454	0.00504695

Table 3: Date 14. Oct, value: `num_orig`

value	total frequency	relative frequency	probability	H_{value}
517	2200	286	0.130000	0.382644
516	2200	607	0.275909	0.512566
523	2200	317	0.144091	0.402727
522	2200	298	0.135455	0.390667
521	2200	253	0.115000	0.358834
520	2200	72	0.032727	0.161456
527	2200	223	0.101364	0.334742
526	2200	80	0.036364	0.173868
525	2200	63	0.028636	0.146790
536	2200	1	0.000455	0.005047

Table 4: Date 14. Oct, value: `num` (XOR(`num_orig`,TSC))

Table 4 might suggest that the values are relatively well distributed but recalling figure 3 shows the low variance between different boots. Please keep in mind that the examples above just represent a fraction of the input values and that all these events are accumulated in the entropy–pool and the only entity that seems to provide randomness in this simple analysis is the timing of the events (not shown here).

Beside the probability and entropy determination for one observation, the same stochastic quantities for each event of each boot-sequence was determined (*auto-correlation*: *n-th* event of every *m-th* boot–sequence). The following tables show the probability values of each block–device uses for different system startups.

value	total frequency	relative frequency	probability
515	8	8	1

Table 5: 1st event, value: `num_orig`

value	total frequency	relative frequency	probability
517	8	6	0.75
162	8	1	0.125
565	8	1	0.125

Table 6: 1st event, value: `num` (XOR(`num_orig`,TSC))

Table 5 shows a constant value (device major–number) because the system uses only one block–device. In table 6 the XOR'ed value differs but is 517 with a

probability of 0.75. This result shows an equality in the value of the TSC register between different boot–processes.

value	total frequency	relative frequency	probability
3287742141	8	1	0.125
3399780677	8	1	0.125
3373268801	8	1	0.125
3132737397	8	1	0.125
3285032933	8	1	0.125
3123395211	8	1	0.125
433217717	8	1	0.125
365967021	8	1	0.125

Table 7: 1st event, value: `time`, source: interrupt

This table (7) presents the timing values. It is interesting because it illustrates that most values seem to cluster around 3.200.000.000. This underlines the results gained in section 2.2 and table 6. The probability distribution of the other events is nearly the same.

Furthermore the low distribution of values in the pool as shown in figure 9 leads to the assumption that the entropy estimation is too optimistic. After booting the system the entropy–count is 4096 indicating that the pool is at its maximum of entropy (8 bits entropy per byte). After taking nearly 100 independent copies (three copies per boot because the primary pool is filled about three time) of the pool content the entropy and arithmetic mean was calculated. The average of the results and their deviation from the maximum are shown in the following table.

	Estimated	Calculated	Deviation
Entropy	8	5.77541	2.22459
Mean	127.5	66.91920	60.5808

Table 8: Entropy– and mean–deviation from maximum

It is hard for a PRNG to estimate the entropy it can provide, and an overestimation is a serious problem [5]. Therefore it is better to have a more conservative estimation function before more deterministic values are provided then expected. Because of this overestimation an application reading n bits from the random–device directly after boot, gets about 28% less entropy than expected[12]. The entropy value of 5.78 bits/byte can be seen as an upper bound because conditional methods will result in a lower value. The pool should protect himself by applying diffent weights for entropy estimation of a source. It might be useful to adjust this weights during system runtime by using results from simple statistical tests that do not consume much CPU cycles.

[12]During boot the entropy may be less.

2.4 Statistical Quantities of Input Values

Beside the probability and entropy of values there are other simple statistical methods that can help to get a better view of the values added to the pool. First, three noise sphere plots[13] [12] [**?**] are shown to visualize the correlation of random number triples. A correlation is indicated by the unbalanced clustering of points in the sphere. Each sphere is printed from three sites (x–axis, y–axis, z–axis).

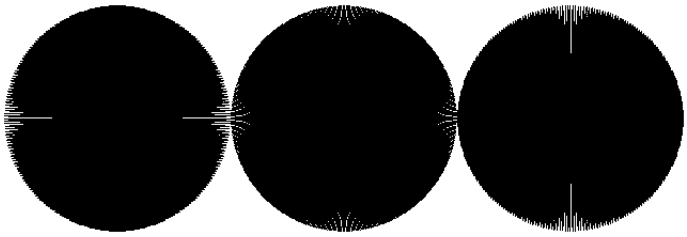

Figure 7: Complete random–number distribution

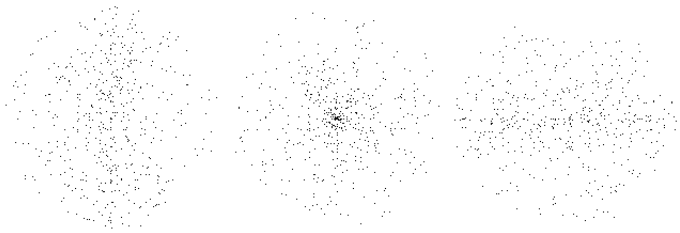

Figure 8: Pool content during boot (with mixing).

[13]Created with a tool called *xnoisesph* based on code from Robert Rothenburg.

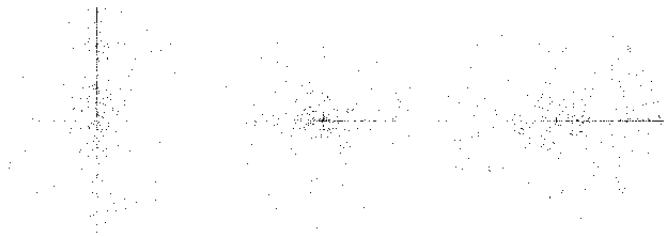

Figure 9: Pool content during boot (without mixing).

In addition to the graphical representation of the input values the calculated characteristics show the low variation of the values. What can be seen here (table 9-11) is the average of each value (sum of values devided by relative frequency), the variance from that average, and derived from that the standard deviation (square root of variance).

	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

An auto–correlation of different system–startups is even more interesting. The two tables below show the *n–th* occurence of `num_orig`, `num`, and `time` generated by the `hwscan` process for m independent observations (boot–sequences.)

	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

Interesting to see is the equality of the variance/standard deviation. The reason why **num** stays the same for the first and second event is that it is the same (as shown in table 1 and 2). The timing patterns shown in figure 4 to 6 are the reason for the very equal deviation seen here.

3 Untrusted Entropy Sources

There are two kind of untrusted sources that can influence the entropy pool: the low-quality sources which may lead to an entropy overestimation, and the malicious source that intentionally changes the pool content to reach an arbitrary goal.

To increase the quality of the entropy–pool during the boot–process it would be nice to have a hardware RNG available. A relatively wide spread and easy to access solution could be the RNG from VIA's x86 *PadLock Security Engine*[14] [4a] or Intel's *Pentium III*[15]. The VIA chip is partly supported in the current Linux kernel. Unfortunately the output of its RNG is not used to feed the pool, which may be due to the lack of trust in the entropy quality provided. Some people also fear that a high output rate of predictable bits can pollute the entropy–pool and makes the cryptographic software, which relies on it, vulnerable. A test report of the VIA CPU can be found in [4b].

3.1 Low–Quality Source

To verify the behaviour of the random–pool under the presence of a low–quality entropy source the code from the kernel was reimplemented in user–space. Each run completely fills up the entropy–pool `iterations` times by using `add_ entropy_ word()`. The entropy–pool can be initialized with random values from `/dev/`

[14]Trademark of *VIA Technologies, Inc*
[15]Trademark of *Intel Corporation*

`urandom` (should have max. entropy because it is the output of a hash–algorithm), could be left empty (zeros), or filled up with the standard data used during boot. Each time the initial pool is modified using the add–function, a user–defined ratio of $1/n$-th values are predictable and the rest being random. Additionally parts of the functions can be disabled (LFSR, twist–table, feedback).

The test was run 100 times with only one pass, the pool was polluted with 1% to 100% of predictable values. To measure how much the non–random values dilute the information–density in the pool the entropy was calculated after each pass. The ideal value of entropy of a fully random and unmodified pool was printed as red line. The experiment was repeated for different setups:

1. adding values, no mixing, no twist table
2. adding values, no mixing, with twist table
3. adding values, with mixing, with twist table (as implemented)

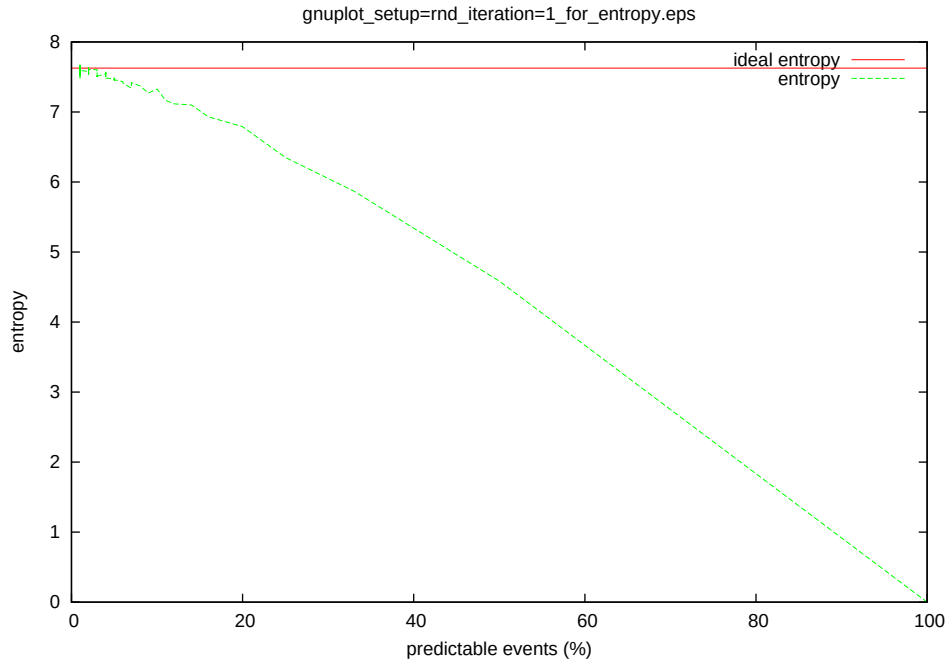

Figure 10: Setup: 1, Entropy from 0% to 100% predictable values.

The entropy falls to zero in a quasi–linear function (figure 10). This result is no surprise because the probability of predictable values rises linear and the values of the random–pool were overwritten.

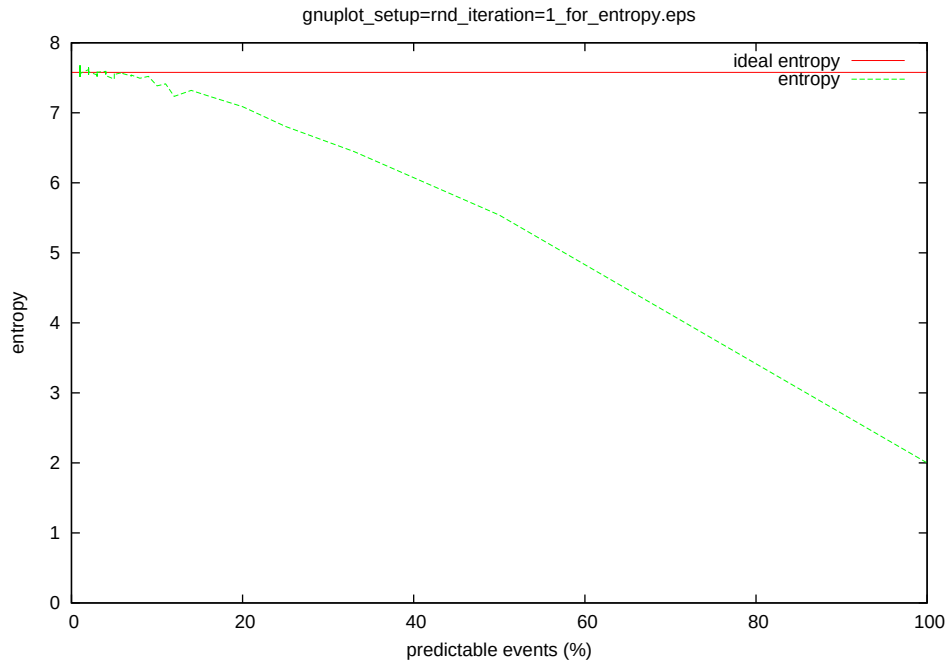

Figure 11: Setup: 2, Entropy from 0% to 100% predictable values.

Figure 11 shows the same situation as in setup 1, with the only exception that the entropy does not drop below 2 due to the twist–table.

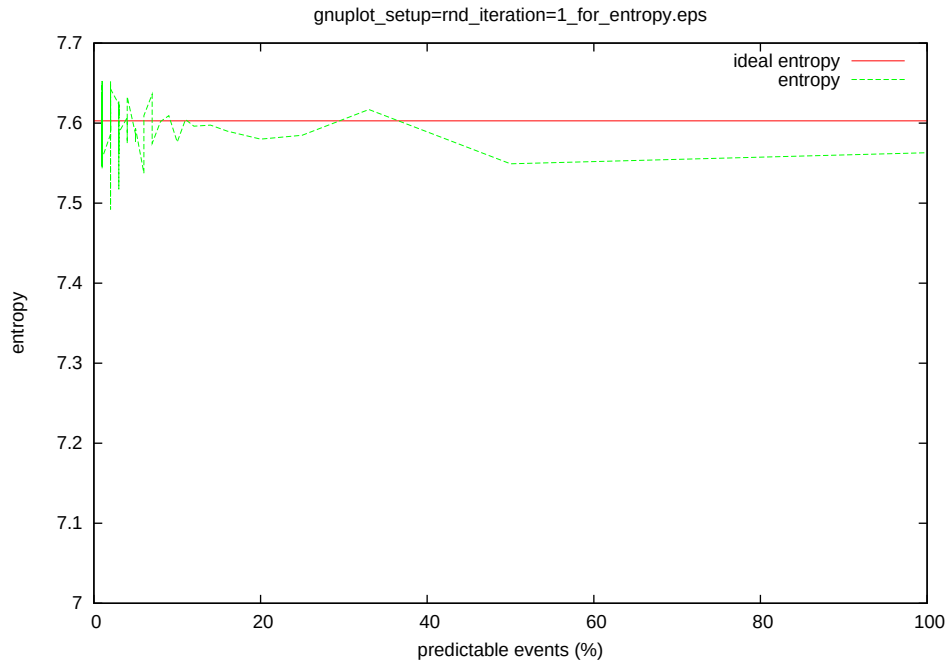

Figure 12: Setup: 3, Entropy from 0% to 100% predictable values.

This is the most interesting plot (figure 12). Due to the TGFSR usage the pool content is mixed with the predictable values and then written to the pool without really lowering the information density. The XOR operation prevents entropy from getting lost even if the value added is constant. Therefore a source of predictable values is not able to dilute the pool. Using this kind of input source does not make the situation worse than it already is, it may only become better.

Because of this behaviour it is important to add input values even if the entropy estimator is at its maximum. The Linux implementation does this by switching between the pools if one of them is "full".

If a low–quality sources generates a lot of input words at an early point in time during startup, maybe even before other sources can be used (driver not loaded), the pool may be filled with no entropy. Applying dynamic weights for this source will protect against an overestimation.

3.2 Malicious Source

The situation looks worse in the case of malicious influence. *Linear Feedback Shift Registers* produce numbers that are uniformly distributed until the upper bound of the period is reached. The length of the period depends on the taps. In the case of a TGFSR the taps are not so important, instead the *twist-table* (or matrix A) is more important [6a]. A LFSR is deterministic, it will produce the same sequence of numbers if it is initialized with the same *seed* (in this case the seed is the pool content). A "pathological" state is encountered if the seed is zero. In this case the output sequence is only zero too. The random device implementation does not check the results written to the pool which may open a vulnerability. But simply adding zeros is not very intelligent and does not differ from a low–quality source, and further non–zero values in the pool are added to the zero value before writing. Another way would be to "neutralize" the values added. The pool is filled–up from the highest position to the lowest and is used like a ring–buffer. The TGFSR uses six taps, four values prior the current position, the position of the last value added, and the current position (which is also the oldest value added).

A malicious source might want to add two words (0x0000000F, 0xF34015C4) to an empty pool. Adding the first word might result in 0xA00AE279. This value is used during mixing for the second word which produces 0x00000000. This can be used by a malicious source the zero the pool if it "knows" the content. To protect against this the values to be written to the pool need to be sanitized first.

4 Entropy Consumers

Several network and file–system components of the kernel use the entropy–pool. This section will summarize the functions exported by the random device–driver

as well as the way they are used. Only functions that can be triggered remotely are represented[16]

To get access to the entropy stored in kernel–memory just a few functions are provided. The mostly used function in the kernel is `get_random_bytes()`. This function reads `nbytes` bytes from the pool - without verifying if enough pool bytes are available - and returns them in a caller–provided buffer. Another function named `secure_tcp_syn_cookie` (which is also based on `get_random_bytes()`) is used for TCP *SYN–Cookie* generation[17]. This section will focus on how these functions can be stimulated remotely[18] and what they do with the entropy.

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.

Fortunately the `extract_entropy()` will not remove entropy from the pool but just decrement the entropy estimator. For this reason an attacker can not make a cryptographic process vulnerable by just triggering entropy consumers over the network. All an attacker may gain is a remote denial–of–service attack or an overestimation by executing input sources in a malicious manner[19].

[16]The user–space can access the entropy–pool through `/dev/random`. and `/dev/urandom` and will be ignored in this section.

[17]The TCP sequence–number generation is not based on bits from the entropy–pool.

[18]Without contemplate the IPv6 and IPsec implementation.

[19]This will not be covered here.

5 Summary and Conclusion

The goal of this work was to test the quality of the input events during the system–startup and to evaluate the modification of these entropy values by the algorithms used in the Linux kernel. To fulfill this task, simple statistical formulas as well as illustrations were used.

The reader should be more familiar with the inner–working of the random–number–generator[20], and should now be able to judge the implementation better than before. Especially application developers should know how far they can trust the entropy from the pool now.

In the **1st section** the implementation was introduced a bit and the sources used or not used were explained.

The values generated by these sources were analyzed in the **2nd section**. Various plots from different boot–sequences showed the equality of the values generated and even timing patterns. To make the low deviation more clear the probability, entropy, and some basic statistical quantities of processes as well as full event generators were calculated. Here the most important part was the analysis of the complete pool content which had shown that the entropy estimation of the kernel is too optimistic during startup and that *at least* $1/3$ *of the entropy values received from the pool are deterministic.* As a rule–of–thumb: "If you need n bits for generating a cryptographic key or a like, read $c * n$ (with $c \geq 2$) bits from the random–device to be on the safe side." By using correlation it should be possible to get a more exact entropy value.

Section 3 may help to reduce the constraints of developers regarding adding support for new and untrusted entropy sources. It was shown that *the entropy gathered from the trusted sources can not be overwritten or removed by sources with less quality* due to the TGFSR mixing. To prevent malicious sources to have a negative impact on the pool, simple tests can be applied to generated events, results written to the pool should be verified, and dynamic weights should be applied for entropy estimation.

In **section 4** some of the entropy consumers in the kernel that can be triggered without having direct access to the machine are visualized. In reality there might be much more possibilities available to an attacker depending on the configuration of the system under attack. Fortunately the implementation does not remove entropy from the pool when it extracts it, only the entropy–counter is decremented. Overestimation might be used to lower the quality of the pool bytes, however this case was not examined.

The two major problems, entropy overestimation and the influences of a malicious source, can be avoided by applying simple stochastic tests. As mentioned earlier, the result of the test can be used to adjust a weight for an entropy source

[20]To repeat myself again: The focus lies in the data modification not too much in the algorithms. The latter field was explained by other peoples work very well.

and to stop writing bad values to the pool during mixing. It may also be useful to change the *tap sequence* to not use the previously gained result when mixing the current input value into the pool. This avoids the accumulation of correlated data by concatenate words from the same input source.

At the end some words should be said about the algorithms. All are deterministic and only SHA-1 is not revertable. But unfortunately function `extract_entropy()` leaks some bits of pool–state. The function uses SHA-1 to hash the complete pool and write the digest back into it.

```
for (i = 0, x = 0; i < r->poolinfo.poolwords; i += 16, x+=2) {
    HASH_TRANSFORM(tmp, r->pool+i);
    add_entropy_words(r, &tmp[x%HASH_BUFFER_SIZE], 1);
}
```

If `poolwords` is 128 the loop iterates eight times and `x` is 16 at the end. The modulo operation for the `tmp` array index results in 1. Therefore the last loop run adds a 32 bit value from position 1 to the pool. Later the 160 bits of the digest are folded into half by XOR'ing the 80-bit-halfs with each other. The intention behind this is to hide possible output patterns from the hash algorithm.

```
for (i = 0; i <  HASH_BUFFER_SIZE/2; i++)
    tmp[i] ^= tmp[i + (HASH_BUFFER_SIZE+1)/2];
// fold the middle-word
x = tmp[HASH_BUFFER_SIZE/2];
x ^= (x >> 16);
((__u16 *)tmp)[HASH_BUFFER_SIZE-1] = (__u16)x;
```

Formalized: a = hash[0-79], b = hash[80-159], c = a XOR b. After folding the halved digest is returned to the caller.

```
/* Copy data to destination buffer */
i = min(nbytes, HASH_BUFFER_SIZE*sizeof(__u32)/2);
if (flags & EXTRACT_ENTROPY_USER) {
    i -= copy_to_user(buf, (__u8 const *)tmp, i);
    if (!i) {
        ret = -EFAULT;
        break;
    }
} else
    memcpy(buf, (__u8 const *)tmp, i);
```

The code copies the lower 80 bits which are the result of the folding to the caller. The attacker can use a search–tree to determine all possible values of a and b that result in c to guess the last 32 bit value added to the pool in the loop. This search results in 2^{32} possible combination of a and b, which is not an improvement

over a brute–force attack. But the equally occurence of zeros and ones in the 160 bits of the digest can be used to prune the search–tree and reduce the possible combinations by factor 7 (about 2^{29}).

This technique does not lead to the real value written to the pool during the last round in the loop but shows that the result of the last iteration should not be used as input because it has strong influences on the value given to the user.

6 Future Work

Only some basic questions are answered by this work and even some new questions came up. The entropy calculation should be refined by using conditional methods. The value gained from this can be used to find a better entropy estimator that dynamically adapts to the system state.

The influence of some parts of the implementation were not analyzed in much detail, for example the cooperation between primary and secondary pool as well as the initialization with a previous entropy pool during system start. These parts may be needed for a better understanding of the inner working of Linux' PRNG.

Furthermore the simulation of the pool on another system would be an interesting but hard task. A method would be needed to exactly describe the deviation of the simulation from the real pool.

References

[1] Bruce Schneier; Angewandte Kryptographie; Addison-Wesley

[2a] Donlad E. Knuth; The Art of Computer Programming, Volume 1; Addison-Wesley

[2b] Donlad E. Knuth; The Art of Computer Programming, Volume 2; Addison-Wesley

[3] Claude E. Shannon; Mathematical Theory of Communication; University of Illinois Press

[4a] VIA x86 PadLock Security Engine, `http://www.via.com.tw/en/initiatives/padlock/hardware.jsp#rng`

[4b] Evaluation of VIA C3 Nehemiah Random Number Generator, `http://www.cryptography.com/resources/whitepapers/VIA_rng.pdf`

[5] John Kelsey, Bruce Schneier, Niels Ferguson; Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator

[6a] M. Matsumoto, Y. Kurita; Twisted GFSR generators; ACM Transactions on Modeling and Computer Simulation

[6b] M. Matsumoto, Y. Kurita; Twisted GFSR generators II; ACM Transactions on Modeling and Computer Simulation

[7] Dawn Xiaodong Song, David Wagner, Xuqing Tian; Timing Analysis of Keystrokes and Timing Attacks on SSH

[8] C. C. Michael, Anup Ghosh; Two State-based Approaches to Program-based anomaly detection

[9a] Anup K. Ghosh, Aaron Schwartzbard, Michael Schatz; Using Program behaviour Profiles for Intrusion Detection

[9b] Anup K. Ghosh, Aaron Schwartzbard, Michael Schatz; Learning Program behaviour Profiles for Intrusion Detection

[10] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu; Finding Collisions in the Full SHA-1

[11] Peter Gutmann; Software Generation of Practically Strong Random Numbers

[12] Noise Sphere `http://mathworld.wolfram.com/NoiseSphere.html`