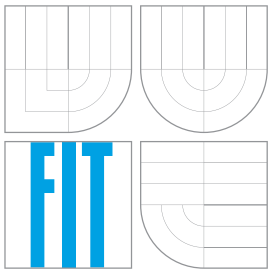
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

ANALYSIS OF ENTROPY LEVELS IN THE ENTROPY POOL OF RANDOM NUMBER GENERATOR

ANALÝZA MNOŽSTVÍ ENTROPIE K DISPOZICI V GENERÁTORU NÁHODNÝCH ČÍSEL

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. PETER KREMPA

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. MAROŠ BARABAS

BRNO 2013

Abstract

The term entropy is in computer science usualy used to refer to a stream of random data. This work sumarizes briefly techniques used to generate random data and describes the random number generator used in the Linux kernel. Later on this work focuses on determining the bit generation speed of the Linux kernel RNG when running as virtual machines under different hypervisors. The work describes the reasons for poor performance of the RNG in virtual environment and proposes steps to overcome them. As a next step, the proposed approach is implemented, tested and the results are compared with the original system. The entropy distribution system is able to improve the level of entropy in the kernel by orders of magnitude when using a fast RNG as a source.

Abstrakt

V informatice je pojem entropie obvykle znám jako nahodný proud dat. Tato práce krátce shrnuje metody generovaní nahodných dat a popisuje generátor náhodnych čísel, jež je obsažen v jádře operačního systému Linux. Dále se práce zabývá určením bitové rychlosti generování nahodných dat tímto generátorem ve virtualizovaném prosředí, které poskytují různé hypervizory. Práce popíše problémy nízkého výkonu generátory nahodných dat ve virtualním prostředí a navrhne postup pro jejich řešení. Poté je nastíňena implementace navržených postupů, které je podrobena testům a její vysledky jsou porovnány s původním systémem. Systém pro distribuci entropie může dále vylepšit množství entropie v sytémovém jádře o několik řádu, pokud je připojen k vykonému generátoru nahodných dat.

Keywords

entropy, generator, virtualization, hypervisor, linux, RNG, VirtualBox, KVM, qemu, Xen

Klíčová slova

entropie, generátor, virtualizace, hypervizor, linux, RNG, VirtualBox, KVM, qemu, Xen

Citation

Peter Krempa: Analysis of entropy levels in the entropy pool of random number generator, diplomová práce, Brno, FIT VUT v Brně, 2013

Analysis of entropy levels in the entropy pool of random number generator

Declaration

I declare that I created this term project independently under the supervision of Ing. Maroš Barabas. I referenced all publications used as sources.

.....
Peter Krempa
May 21, 2013

Contents

Chapter 1

Introduction

With introduction of hardware virtualization support virtual machines have grown from a convenient way to test new features and software and aid kernel developers to a fully fledged solution used in enterprise environments. Virtualization is now commonly used to consolidate hardware infrastructure and reduce costs of the infrastructure by avoiding ownership of physical hardware in favor of virtual machines. Thanks to this more and more services are being migrated to virtual infrastructures.

Modern information systems are trying to balance information security with global availability of the data. One of the key aspects needed to achieve this balance is encryption. Encryption algorithms often require random data to initialize key generating algorithms or for generating of challenges in challenge response authentication. This brings the focus to the random number generator that is available as a part of the operating system.

The path taken to increase the usability of virtualization included solving performance issues of storage and networking subsystems as they are the most critical from a response latency and performance point of view. The previously overlooked and less important subsystems are now becoming bottlenecks. This work will focus on analyzing performance of the random number generator in the kernel of the Linux operating system.

The performance issues of the Linux kernel random number generator are easily ignored as usually only the *urandom* pool is used in applications and the interface of that pool is supplying entropy from a pseudorandom number generator in the case the main random pool is depleted. When strong entropy is needed the regular pool of entropy may be depleted and the application requiring the access to such entropy will block until the kernel is able to fulfill the demand.

This thesis consists of a brief introduction into entropy generation used nowadays with focus on enterprise usage of random number generators followed by a brief introduction into virtualization and available hypervisor solutions. The work will then focus on testing the performance of the Linux kernel random number generator under various circumstances and running inside different hypervisors and compares and analyzes the results. The next chapter is aimed to introduce options that can be used to improve the performance of the Linux kernel random number generator when running inside a virtual machine and choses the approach to be implemented. As the next step the implementation details of the software system created for improving the performance of the random number generator are summarized. The following chapter then analyzes the impact and performance of Linux kernel random number generator when the implemented system is used to pass additional entropy to the guest operating system. In the conclusion, goals of this work are summarized and future work on the software system that was created as a part of this thesis is proposed.

Chapter 2

Entropy

Entropy is commonly described to be the **measure of uncertainity and disorder** in a system.[10]

In information theory the term entropy was defined by Claude E. Shannon as the average unpredictability in a random variable, which is equivalent to its information content. [12]

For purposes of this work the term **entropy** also refers to the contents of the entropy pool - the random bits prepared to be read by applications or more general to a stream of random bits.

2.1 Use of entropy in computer systems

In modern computer systems entropy is used to accomplish various tasks. These may range from simple games, animations, art, random images to generating of cryptographically strong keys and electronic gambling.

Entropy generator in computer systems is normally referred as a random number generator or RNG.

Various tasks have different quality and quantity requirements for entropy. For computer games, computer art, and others, large quantities of random data are needed to simulate various seemingly random physical phenomenons. Those data don't ultimately need to be truly random. A good appearance of randomness is enough for these tasks.

On the other hand for use in creation of cryptographic keys, unique identifiers, password generation and even computer based gambling the requirements are different. These tasks require truly random and unpredictable data otherwise an attacker could take advantage of the knowledge of the random algorithm used by the RNG to gain acces to the system without permission.

Historically RNGs have been implemented in the applications themselves and later in libraries. Currently best random number generators are usually found in cryptographic libraries such as OpenSSL where they are used to generate cryptographic keys.

With a stronger need for information security cryptography is increasingly used to protect user data. Introduction of disk encryption and VPN[1] comes with the need to have good quality entropy generators in the kernels of operating systems for generating of key material. This allows also general purpose applications to have access to good quality and properly seeded RNGs without much extra effort by accessing the common RNG.

[1] Virtual Private Network - a secure tunnel between two separated parts of a network

Additionally current hardware development makes it easy to include components to create a entropy generator on the system boards of computer systems. These use principles described in section 2.2.2. The generators have fast bit speeds with true random behavior and are easily accessible in cheap systems. One of possible disadvantages of this approach is that a bug in the hardware random number generator can't be patched in most cases and may be widely exploitable.

2.2 Random number generators in computer systems

There are multiple approaches to generate streams that have high entropy levels. Each of the approaches has advantages and disadvantages and thus limited usage for some kinds of problems.

The most basic approach is to use a pure software solution. This is described as a pseudo-random number generator. This solution is fast but limited in the entropy level. The entropy level of the sequence is reduced to the entropy of the initial value.

The second approach improves the first by having a pool of entropy that is continually filled with entropy originating from hard-to-predict events, like network, user and disk activity. There are multiple implementations using this approach including the Linux kernel RNG.

The third option to gather entropy is to measure and extract randomness of truly random effects. This depends heavily on physical characteristics of some components and requires special hardware for the generator.

2.2.1 Pseudo-random number generators

Pseudo-random number generators - PRNGs - use an algorithm to create long sequences of numbers that are apparently random.

PRNGs are fast in terms of bitrate as they don't depend on any external input, just on the previous value.

The generating algorithm depends on a starting value called **seed** which may be taken from the current time or other hard-to-predict value or chosen arbitrarily to allow generating the same sequence. Without knowledge of the algorithm the next output value of a PRNG has high entropy for the observer and thus can be considered as a random number. [1]

Advantage and disadvantage depending on field of usage of PRNGs is periodicity. After a certain amount of generated numbers the sequence repeats as the previous value is equal to the seed. The repetition period may be long enough to not appear during normal use. This and the speed of PRNGs make them ideal for modelling and simulation, computer games that might benefit from repeatability[2] or computer art and generation of artificial terrain.

If the algorithm and seed is known to the receiving party the entropy of the stream generated by a PRNG is equal to zero as each bit can be calculated.

The most simple PRNG is the linear congruential random number generator. It's defined by the equation $X_{N+1} = aX_N + c \bmod m$ [17]. The values of a, c and m have to be chosen so that the sequence does appear random enough.

There are better PRNGs in terms of distribution and sequence repetition. One of the popular algorithms is the *Mersene twister*. PRNGs are commonly used with a high entropy

[2]This allows for example to play a game with the same generator. starting point over again.

data source used to seed the PRNG for high security applications that require larger amounts of data but the entropy of the seed is sufficient.

There are various standards describing algorithms used as PRNGs and fields of applications of these generators such as the NIST standard for Deterministic Bit Generators [1].

2.2.2 True random number generators

True random number generators - TRNGs - are based on physical phenomena that are hard or impossible to predict. The entropy is contained within the effect itself and it is extracted by measuring the occurrence, time difference or absolute value of those effects. The quality and speed of entropy bits generated by these generators depends on the phenomenon and also on the physical construction of the detection device.

Phenomenons ideal to be used for entropy generation are radioactive decay, avalanche effects on reverse-biased electronic components and thermal, atmospheric and other sources of noise, detection of photons travelling through semi-transparent mirrors and others. Some of those aren't practical for real implementations. Commonly used approaches are avalanche effects on electronics and thermal and atmospheric noises.

The TRNGs are slower if compared to PRNGs, but their main advantage is that even with full knowledge of the working principle of the generator and all initial conditions, the devices keep high entropy levels. TRNGs aren't commonly used but some designs may be implemented on motherboards of computers with no or low extra cost.

Random number generators based on radioactive decay

Radioactive decay of atoms is considered to be truly random as it's based on quantum events that may or may not happen at a certain point in time. A radioactive decay based RNG uses a small radioactive emitter that is safe to handle and detectors that register decaying of the atoms. The entropy is contained in time between the decays happening.

Noise based RNGs

This type of RNG uses a source of noise, that is amplified and sampled using an analog to digital converter. The noise source is usually a thermistor, a output of a reverse biased semiconductor devices or a circuit based on avalanche breakdown semiconductor devices. These approaches are easy enough to implement cheaply and using regular electronic circuitry and is commonly used in crypto-accelerators and hardware RNG tokens. [13]

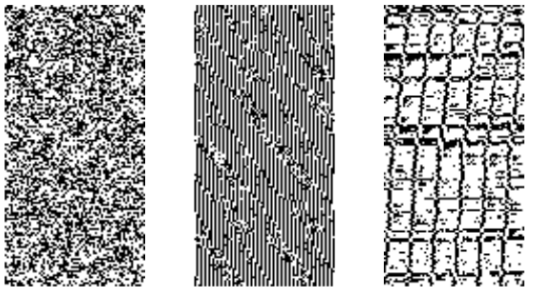
Clock drift detection and ring oscillators

Oscillators used in computer systems tend to drift in their frequency due to tolerances in the components they are built from. This phenomenon can be used to collect entropy data as the variance of the drift is changing due to noise picked up by the components the oscillator is created from.

The usual approach is to have two free running oscillators that contain components to deliberately destabilize their oscillation. One of the oscillators is then used to trigger sampling of the other one creating a bitstream that is considered to be random as it's impossible to simulate.

Figure 2.1: Entropykey: a hardware random number generator USB device. [13]

As the phenomenon that this RNG is based on isn't quantum based this RNG might be tampered to create less random sequences by decreasing the amount of noise picked up by the oscillators. This kind of attack, although theoretically possible, isn't feasible in real applications as systems usually stop operating at temperatures required to decrease the noise enough.



(a) No injection, 70×140 bits

(b) 1.822880 MHz injection, 70×140 bits

(c) 1.929629 MHz injection, 73×146 bits

Figure 2.2: Tampering of oscillator drift based RNGs with power supply signal injection. [9]

Disadvantage of random number generators based on oscillator drift is the susceptibility of power supply noise to alter and lock up the frequency of the free running oscillators and decrease the quality of entropy produced. There's a known attack on RNGs based on this principle used in smartcards. [9]

2.2.3 Random number generators based on hard-to-predict events

Common computer systems require high quality entropy but usually don't have a TRNG to generate it. Apart from PRNGs there's another way to generate entropy only by software means. RNGs based on these principles usually measure disk activity, mouse movements and user activity in the system and extract the entropy of these events. Multiple implementations of this approach exist, where the best known one is the Linux kernel Random number generator that will be mentioned later. Predecessors of this approach were implemented as

user space application that were periodically running statistical commands and extracted entropy from such sources. An example of software used to generate entropy by such an approach is the entropy gathering daemon - `egd`.

2.2.4 The Linux kernel random number generator

The Linux kernel contains a random number generator - the LRNG - that supplies high quality entropy based on entropy gathered from hard-to-predict events. The generator collects information about key-presses, movement of the mouse, jitter of access times to disk and various interrupt sources of the system. The collected data are then stored in the entropy pool until it's requested for usage.[5]

First step of the generator data flow is acquisition of data. The kernel records information about events from the input devices and interrupts along with the timestamp when the event happened. The events are then queued for addition to the pool that is scheduled regularly. The entropy contribution of each of the events isn't known and is approximated at the time the Primary pool is stirred with the contents of the event data. The entropy contribution of the event depends on the type of the event and also on repetition rate of same event[5]

The pools updates take the packets of data prepared by the acquisition code and update the primary pool with them. The pools are constructed using feedback algorithms to stir the contents of the pool while updating the other ones. The secondary and urandom pool are updated in a similar matter from the primary one. Along with this operation the pool volume estimate is updated by the approximate estimate of the contribution.[5]

To access the entropy data from the LRNG userspace applications can use the character device pseudo files `/dev/random` to access the blocking pool and `/dev/urandom`. Kernel tasks can only access the non-blocking pool using the function `get_random_bytes`. The data are extracted from the corresponding pools, either the secondary pool for the blocking output or from the urandom pool for non blocking. The data extraction algorithm includes hashing the extracted values and stirring parts of the data back to the pool to refresh it's contents.[5]

On shutdown of the computer it's recommended to save the state of the LRNG by extracting data from the urandom pool and storing them to disk storage. This data is used in the next boot sequence to seed the LRNG so the state cannot be easily predicted.[5]

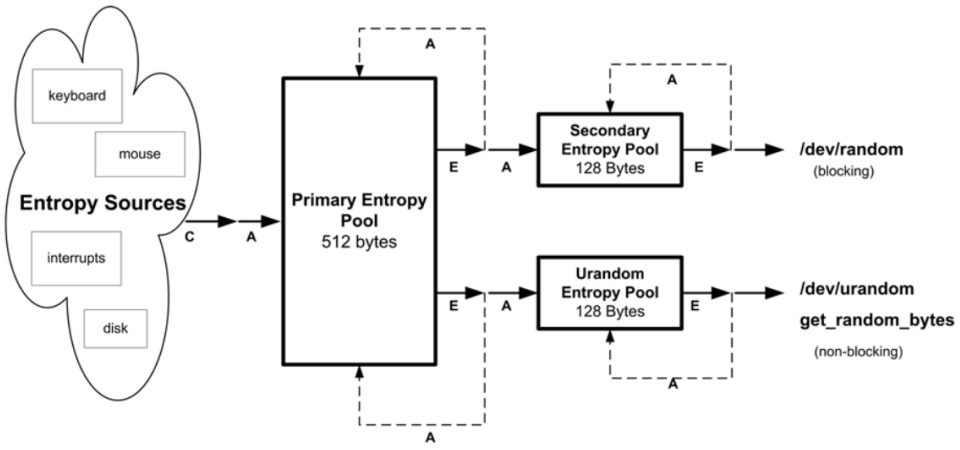


Figure 2.3: Linux kernel random number generator block diagram. [9]

2.3 Testing and certification of random number generators

Computer systems that are used in enterprise environments are required to be trusted. This is mostly true for security measures as encryption where random number generators are used to generate encryption keys. This lead to creation of multiple test suites designed to test the quality of entropy generated by random number generators. These suites measure various statistical values on samples of data from the generators to ensure that the output sequence of the generator can be guessed without the knowledge of the initial state. [11]

The most notable examples of such tests are the Diehard and Dieharder test and the recent NIST 800-22 suite[11].

Bugs in RNGs used for cryptographic purposes may have catastrophic outcomes. An infamous bug in the `OpenSSL` software package in the Debian Linux distribution caused that cryptographic keys generated by the package and used for system authentication and other purposes had only 1 of 32768 possible values making an attack trivial. [16] Commonly used RNG implementations used in computer systems are subject to multiple security audits and are also empirically verified by real-world usage.

Computer systems that are used in enterprise and government applications need to fulfill strict sets of rules to ensure security of the system. As entropy is used in multiple places in such systems in ways that directly influence security of the system, RNG devices are subject security certification too. The standards often list approved random number generation methods. An example is the *Federal Information Processing Standard* that specifies the methods in [4].

2.4 Conclusion

This chapter shows, that there are multiple sources of entropy available in a computer system and with some effort they can be converted to be used as RNGs producing high quality entropy. The summary of advantages and disadvantages should help the user to choose a suitable source of entropy according to the needs, availability and requested security level and certification.

Chapter 3

Virtualization tools and solutions

Virtualization is a fairly modern phenomenon in computer science at first used for experiments, testing and to ease debugging. Now virtualization is increasingly used also in enterprise environments to save costs of physical hardware.

3.1 Anatomy of a virtualized computer system

In virtual system most of the hardware is abstract and emulated by software. The virtual hardware creates an abstraction layer on top of the physical hardware. This is beneficial as it enables to run guests on various hardware platforms, change hosts of the guests - migrate them.

Virtualization allows to share physical hardware that wouldn't be fully utilized by a single task by grouping such cases on one physical machine. The services run in virtual environment aren't different from those running on physical hardware and so aren't the requirements of them.

There aren't many downsides of virtualization solutions. One of them is performance. Virtual machines due to the abstraction layer are slower than physical machines. This issue is being reduced using para-virtual hardware drivers.

3.2 QEMU-kvm and libvirt

One of the many virtualization solutions available today is the QEMU hypervisor used together with libvirt as management. This open source virtualization tool is used both on desktop systems and in enterprise environments.

3.2.1 QEMU

Qemu is a open source hypervisor and emulator that was originally developed by Fabrice Bellard. Qemu supports multiple targets and allows to emulate platforms different from the host platform. Qemu is a full virtualization solution including BIOS or other firmware interfaces and hardware emulation. This allows to run unmodified guest operating systems in such an environment. The QEMU project is available at http://www.qemu.org/. The QEMU project is active and heavily developed and releases are being done regularly.

Qemu supports a wide range of virtual and paravirtual hardware that can be used in the guests with multiple backends for storage and networking. This makes it a good choice for general purpose virtualization.

To configure options for a virtual machine the QEMU hypervisor uses command line arguments for the instance being started. For run time modifications of the state such as change of media in virtual drives, hot plug and unplug of devices or changing the state of the machine an interactive communication channel can be used. This channel is know as *monitor*.

Neither the command line interface nor the monitor are user friendly and QEMU doesn't store any persistent configured state when the virtual machine is not active. This makes QEMU hardly usable by itself and thus a higher level management application is needed which also allows to manage storage and networking for virtual machines.

3.2.2 KVM

Qemu itself doesn't support any virtualization extensions of a CPU that would allow it to run the guests in a more optimized way. For this purpose the kernel based virtual machine extensions, known as **KVM**, were developed.

KVM is no hypervisor itself. KVM is a part of the linux kernel that allows to control the virtualization extensions of a CPU, set up address spaces for guests and channel I/O operations that would originaly be used for interaction with physical hardware. KVM is meant to be used by the hypervisors like QEMU that will benefit from offloading memory management and other crucial data paths into the kernel, while hypervisor itself is implementing device emulation and control of the virtual machine. The interface of KVM is exposed as the `/dev/kvm` device node.

KVM was first introduced for the x86_64 architecture and now the interface is being ported to other architectures such as ARM. KVM was first developed by Avi Kivity at Qumranet.

3.2.3 VirtIO

Full hardware virtualization includes emulation of hardware peripherals on a level where the guest operating system can't distinguish it from a real piece of hardware. The interface of a hardware device was originally designed with respect to the hardware layout and thus isn't usually well suited to be emulated by software. For use in a virtual machine it would be better to design an interface that is meant to be processed by the software emulation layer and thus avoid constructs and approaches including timers and sequencing of the I/O layer. [6]

The VirtIO paravirtual device drivers are trying to solve this issue. The VirtIO infrastructure consists of the emulated hardware that has I/O interface desinged to be used with a software emulator and guest kernel device driver that allows to use such devices. VirtIO drivers are available for disk drives, network cards, serial ports and dedicated communication channels and also for RNG devices. The goal is to achieve near native performance when using the infrastructure to pass through resources from the host system. [6]

VirtIO was developed by Rusty Russel as a acceleration solution for his virtualization solution called `lguest` and is now widely supported in guest operating systems. [6]

3.2.4 libvirt

Libvirt is a open source project that tries to provide a common API and configuration interface for various different hypervisors and the infrastructure needed to support it. Libvirt was started by Daniel Veillard and Daniel Berrange. The project is accessible at `http:`

//www.libvirt.org/. The project is active and under heavy development with support of large IT companies.

Libvirt provides means to configure and manage storage, networking, resources and resource limits, virtual machines and snapshots of them. Libvirt uses XML documents to describe virtual machines and the hardware and provides persistent configuration storage and management of hypervisor processes.

Libvirt also supports remote connection using the libvirt RPC protocol. This allows to manage a hypervisor host from a remote location.

API

Libvirt's api is designed to be a universal interface abstraction for the underlying hypervisor and the operation it can support.The API of libvirt is guaranteed to be backwards compatible and legacy functions are still being maintained although they are obsolete.

QEMU hypervisor driver

The QEMU hypervisor isn't a standalone virtualization solution. Qemu requires instrumentation to start the process with correct command line arguments connect to the monitor and control the virtual machine. This is the purpose of the QEMU driver in libvirt.

The driver manages the running processes and converts information from the libvirt APIs into commands for the QEMU monitor. The QEMU driver is a stateful driver and thus the libvirt daemon is required when managing QEMU virtual machines.

virsh

The virsh – virtualization shell is a basic management application that uses the libvirt API. It was originally developed as a testing tool for the libvirt API but was further developed to be a user friendly basic interface to libvirt to allow simple management tasks. Virsh is distributed along with the libvirt package.

```
$ virsh
Welcome to virsh, the virtualization interactive terminal.

Type:  'help' for help with commands
       'quit' to quit

virsh # list
 Id    Name                           State
----------------------------------------------------
 4     test-vm                        running
```

Figure 3.1: virsh virtualization shell

virt-manager

The virsh shell is only a minimalist interface suitable for minimal tasks. For more complex tasks such as creating a virtual machine virsh requires the knowledge of the libvirt XML format. To ease this type of operations a graphical user interface for the libvirt API was created. [15]

Virt-manager is a separate project that uses the libvirt API to create and manage virtual machines on individual hosts in a user friendly way. Virt-manager is written in python and supports connecting to multiple hosts and provides configuration wizards for creating virtual machines and integrates the graphical console of the machines. [15]

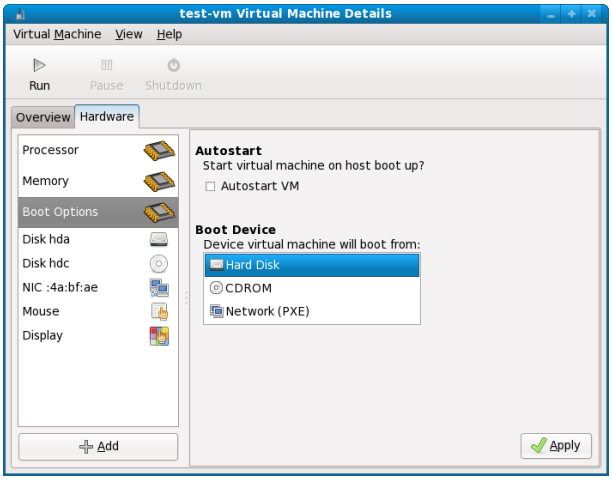


Figure 3.2: virt-manager management interface. [15]

OpenStack, oVirt and others

Libvirt was originally designed to be used on a single node only as a abstraction of the used hypervisor. To allow using it for more sophisticated topologies in datacenters a higher level system needs to be implemented that will manage resources in the datacenter using libvirt on a larger scale.

There are a few open source projects using libvirt as a host based management layer and are building a larger scale application on top of it. OpenStack compute and oVirt are examples of such projects and both are also sold as products with support by large companies.

3.3 VirtualBox

VirtualBox is an all in one virtualization solution including the hypervisor, management interface and guest support drivers. VirtualBox is developed by Oracle and usualy used as a desktop virtualization solution mainly due to a user friendly intreface and easy installation and good support for video acceleration which makes it suitable to virtualize systems with heavy graphical user interaction.

VirtualBox is distributed also as open source project lacking several features described as enterprise such as network booting support and USB passthrough support. Together with the lack of multi host management software and closed source nature of the guest drivers the target segment for use of VirtualBox is on desktop computers rather than servers.

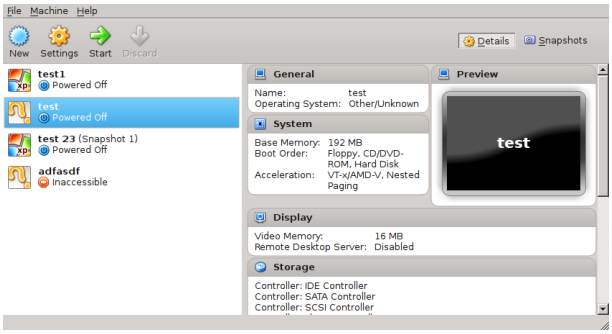


Figure 3.3: VirtualBox management interface.

3.4 Xen

One of the first hypervisors available was the Xen hypervisor. Originaly only paravirtualized Linux guests were supported. Those required modified kernels that replaced hardware drivers and system calls with *hypercalls*. The main reason for this was the need to modify the kernel to run cpu security ring 1 instead of the usual ring 0.

With the introducition of hardware virtualization support into the hardware the Xen hypervisor was upgraded to support it and was one of the firtst to support fully virtualized guests with hardware support.

The Xen hypervisor is now being used less due to the need to do heavy modifications to the host kernel. These modifications are not keeping up with upstream Linux kernel releases. As a replacement, other solutions are used that run on top of the host system and access the virtualization support using a stable interface.

The Xen hypervisor was originaly developed at the University of Cambridge.

Chapter 4

Entropy in virtualized systems

The aim of a virtualized environment is to be indistinguishable from a physical computer system in terms of performance. This is also true for the random number generator. In this chapter I will focus on analyzing of performance of the Linux kernel random number generator under different conditions and various virtualized environments to determine the performance of the generator and I will try identify possible problems with the performance and their causes.

4.1 Available entropy in kernel entropy pools

As described in section 2.2.4 the kernel random number generator stores noise bits collected from the available sources in the entropy pool. The kernel and tasks using the entropy from the pool decrease the amounts of entropy contained in the pool. The available amount of entropy in the kernel pools is an important measure so the following sections will try to research the filling rate and consumption rate of the random number generator in the system under various situations and conditions.

While the entropy pool is filled with entropy all requests including those originating from the non-blocking access methods receive high quality entropy from the primary pool. In cases when there isn't enough entropy in the pool requests for entropy from `/dev/random` block and the non-blocking sources receive entropy from lower quality source. This fact might might lead to starvation of applications trying to generate cryptographic keys and thus requiring access to the higher quailty pool by applications accessing the pseudorandom pool.

Filling of the entropy pool during start and boot phases of a computer system is another source of possible problems. The entropy pool is empty when the kernel is initialized and although the system is under heavy load which increases the entropy generation rate, the starting services may consume a lot of entropy for their initialization purposes. Additionally the services may block the startup if high quality entropy is needed for the startup of the process.

4.2 Impacts of virtualization on performance of random number generator

The virtual hardware due to the nature of emulation may have different properities when compared to the real hardware the emulation is based on. The properities may differ in

terms of timing, interrupt frequency and other variables that are determined by the state of the hardware. As these are the main source that contains entropy in a computer system the changes of the behavior may influence the performance of the generator that is extracting random numbers from the entropy and quality of them.

An example of this difference to real hardware is the disk of the virtual machine being backed by a file on the filesystem of the host. The operating system of the host may apply caching of disk accesses and thus offset the access times for individual blocks and also the count of interrupts in the guest if it is able to provide data in larger chunks. Also fragmented blocks of a filesystem may be present in the cache and avoid the jitter of accessing the disk drive.

Virtual machines also usually don't have input devices directly connected but use a virtual terminals. The input devices are one of the best sources as they are operated by humans in an unexpectable manner

Those are a few examples of differences between a physical computer and a virtual machine. How these affect creation of entropy in the RNG will be discussed in the next section.

4.3 Gathering of Linux kernel RNG performance statistics

The kernel procfs interface in Linux provides means to collect statistical information about the state of the kernel entropy pool. The `/proc/sys/kernel/random/entropy_avail` file will be used to collect information about the estimated volume of entropy contained in the entropy pool. Before the start of each test the entropy pool will be drained using the `/dev/random` device to start the process of filling the pool. This will show the filling rates of the RNG. The guest operating system will need to have minimal impact on negatively affecting the statistics and thus a idle system with minimal amount of started services will be used.

Four sets of experiments will be conducted on multiple hypervisors to portray and quantify the performance of the Linux RNG under various conditions and environments.

4.4 Synthetic tests

The purpose of the synthetic test will be to test the filling rate of the Linux kernel RNG that will be running in virtual systems. To compare the influence of virtualization archigecture on the performance of the LRNG I will use Xen, QEMU/KVM and VirtualBox as hypervisors. These are the most commonly used open source hypervisor solutions available.

As the guest operating system under test I will use Red Hat Enterprise Linux version 6.4. This is a enterprise system that is commonly used and contains support for guest based virtualization support. I will use a minimal installation to reduce the number of running services that might drain the entropy pool and interact with the generation of data. Use of this system will also help in comparing the performance of the Linux RNG when compared to the host system.

As the host system I chose Red Hat Enterprise Linux Server version 5.8 for the Xen hypervisor and Red Hat Enterprise Linux version 6.4 for QEMU/KVM and VirtualBox. The host systems run on a Intel Core 2 Duo processor and the host has one disk drive.

4.5 Real world scenarios

The methodology described above will also be used to collect reference data for real world behavior of the Linux kernel RNG. For this purposes I propose two common usage scenarios:

The first scenario is a virtualized server computer system. The host operating system runs several instances of the QEMU/KVM hypervisor. The guests are production systems running several services and hosting user space for multiple users. All externally accessible services are secured through encrypted channels. The host is a Intel Xeon based machine with a 6 disk raid array and two network connections. As this system is a production machine, performance on various hypervisors cannot be tested. Operating systems used on this machine are Debian Linux installations with custom kernels.

As a second real world example I selected a desktop computer system used by single user for common computer work without the use of virtualization. The system is a Intel Core i7 based machine with Gentoo linux installed. This system has external input devices connected and actively used. Many of the applications used on the system include use of secure channels and the disk storage of the system is encrypted using dm-crypt. This system will try to analyze the long term performance of the LRNG on a desktop system to allow comparison with server systems.

4.6 Virtualization acceleration drivers

A third test scenario will focus on comparing the performance of the LRNG on a guest that is using paravirtual accelerated hardware interface and on a guest that has this advanced interface disabled.

The goal of this test will be to analyze the effect of these drivers on the LRNG speed and compare it with improvement of the performance in the guest.

This test will be based on the synthetic test described above. Along with the LRNG performance testing, the guest will be performing disk reads and the total disk throughput speed will be graphed along with the LRNG performance. To avoid influence of physical hardware a empty sparse file simulating the disk image will be used. This will avoid biasing the results by using a slow hard drive and instead the full potential of the virtualization infrastructure will be utilised.

This test will also allow to compare the performance of the LRNG when used on a system under load, where the disk I/O operations are the primary source of entropy.

4.7 Entropy used up on process start

Early experiments with trying to determine state of the entropy pool in the system shown a strange pattern. After trying to log level of the entropy pool by running a separate process for every data point, the pool drained very fast leaving the minimum value of 128 bits of entropy that is an implementation limit.

Graph 4.1 shows the gradual decrease in the volume of the entropy pool with reference to number of tightly looped started processes. Each start of a process drains 128 bits of entropy from the pool.

Using the `strace` tool to log all system calls by a process has shown that no entropy data is read by the initialization phase of the processes themselves.

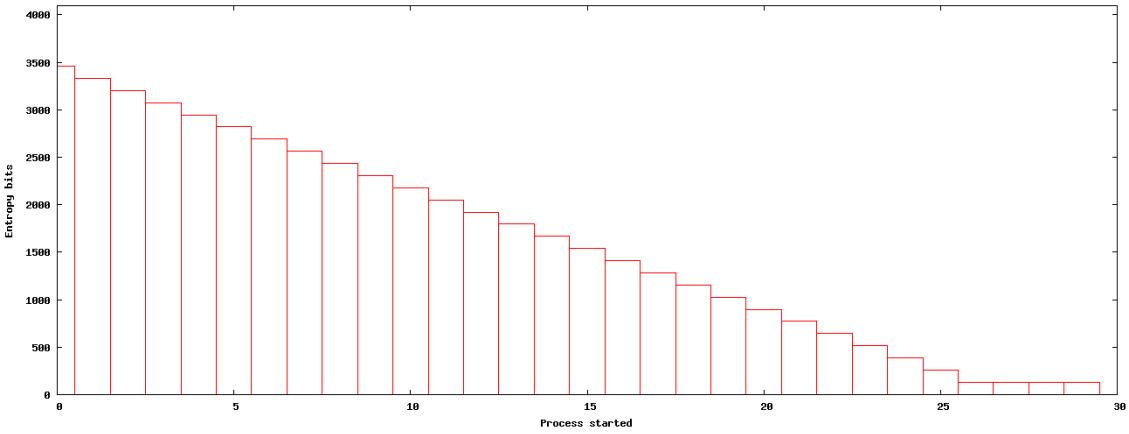


Figure 4.1: Draining of the entropy pool when starting processes.

Further investigation of the Linux kernel showed that 16 bytes of entropy is consumed by the ELF file loader in the kernel when an executable binary is being loaded. The data is stored in the `AT_RANDOM` attribute in the ELF header of the running process. This random initialization vector is used by the glibc library to seed stack protectors and PRNGs. [2]

The workaround used to gather unbiased data is to use a single process that accesses the entropy pool state in a loop by a single process. Together with in-memory caching of the statistics data this minimizes the impact of the analysis software on the tested system.

Starting of processes is a very common task, so this draining might have serious impact on the contents of the entropy pool on busy systems and during boot phases where the init scripts are starting a lot of processes.

4.8 Results

The results were collected by the use of a custom program that was periodicaly reading the state of the random pool and storing the results into memory. After collecting samples for the requested time period, the program wrote the results to a file on the disk. This approach avoided disk writes while the statistics were collected. Afterwards the data were plotted using the gnuplot tool.

The program was also used for gathering statistics while the system was booting. The program may be run as the init process with pid 1. In that case it forks to create a clone that will start gathering statistics, while the original instance executes the sytem startup as usually.

The program is called `entropy_boot` and the source code is on the attached CD.

4.8.1 Synthetic test scenario

As there were two different host systems used to collect statistics graph 4.2 shows the performance of the RNGs depending on the kernel version. The graph shows that the RNGs of both of the hosts have a similar performance, where linux-2.6.18-xen averages roughly 6 bits of entropy data per second and linux-2.6.32 averages 5 bits per second. Both the host systems outperform the guest systems by at least 5 times, where the best guest averages 1 bit per second.

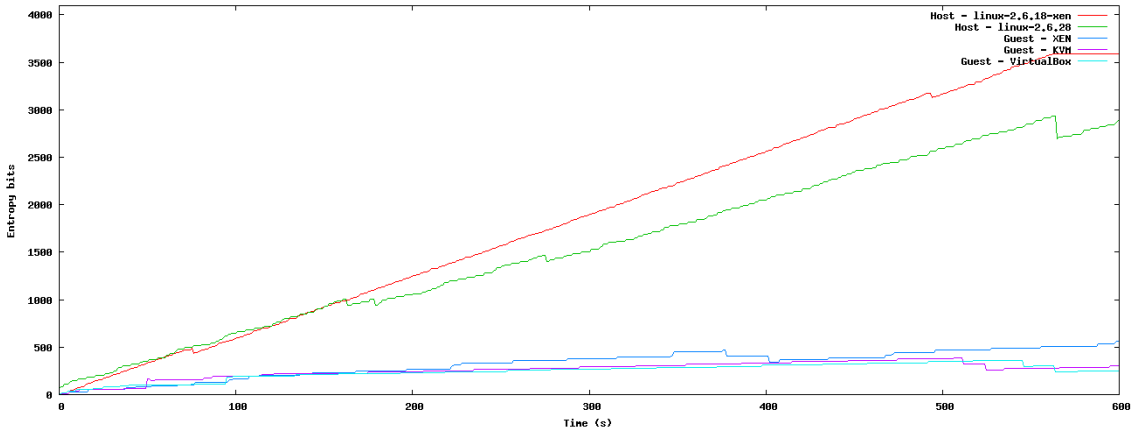


Figure 4.2: Synthetic test - Performance of the RNG in host systems - guest data included for reference.

Graph 4.3 compares the performance of the RNGs in a guest operating system across various hypervisors. It's apparent that the guest achieves the best performance while running on the Xen hypervisor averaging 1 bit per second. Qemu/KVM and VirtualBox with the guest additions both perform worse than Xen averaging roughly 0.4 bits of entropy per second.

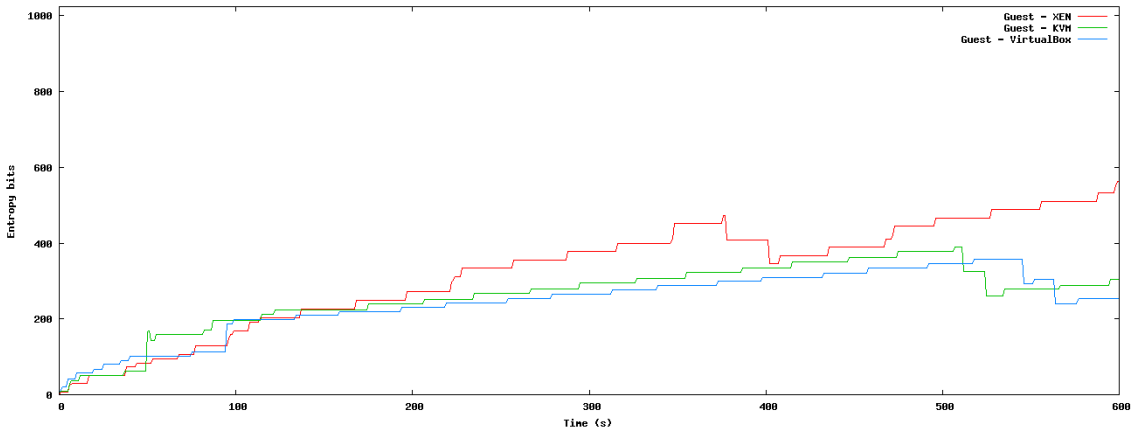


Figure 4.3: Synthetic test - Performance of the RNG in guest systems.

4.8.2 Real world scenario

In the production system the results of the experiment correspond to the synthetic tests. According to graph 4.4 the host system is able to generate high enough amounts of entropy and it's usage isn't affecting the growth of the pool. This contrasts with the situation in the guest system where the kernel isn't able to keep up with consumption of the application and the size of the kernel pool fluctuates around of the minimal read limit value.

The overal performance of the LRNG in an idle and loaded server systems is insufficient and even under load the output rate of the LRNG does not scale well enough to maintain

a steady amount of entropy in the pool.

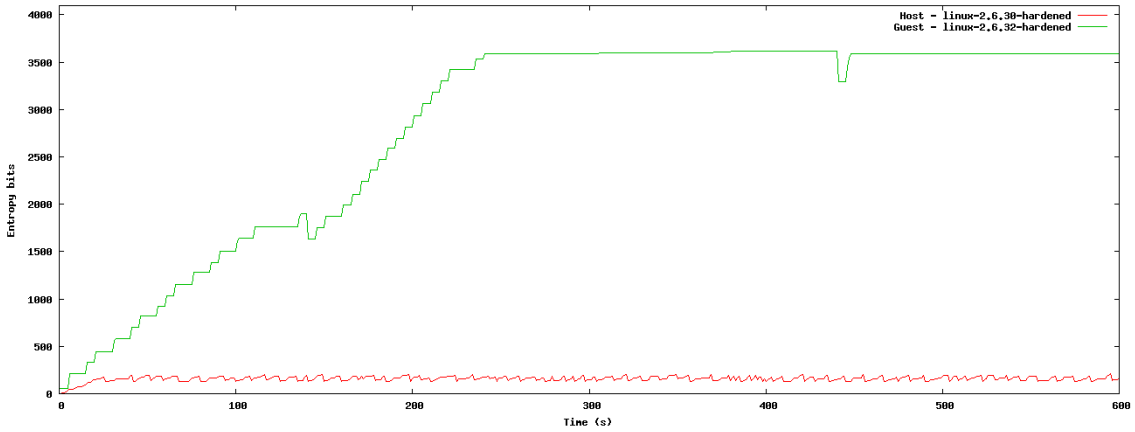


Figure 4.4: Real world scenario - virtualized server system.

Figure 4.5 shows entropy pool level fluctuations on a desktop computer system. Thanks to actively used user input peripherals the RNG is able to generate high amounts of entropy in short periods. On the other hand, as the system is actively used, entropy consumption of the running applications is very high (multiple processes executed, encrypted connections, wireless network encryption keys). These two factors result in sharp changes of the entropy level.

On a desktop system used by a single user the LRNG performes better when compared to server systems due to the existence of actively used peripherals. For a single user use case the LRNG performes well enough.

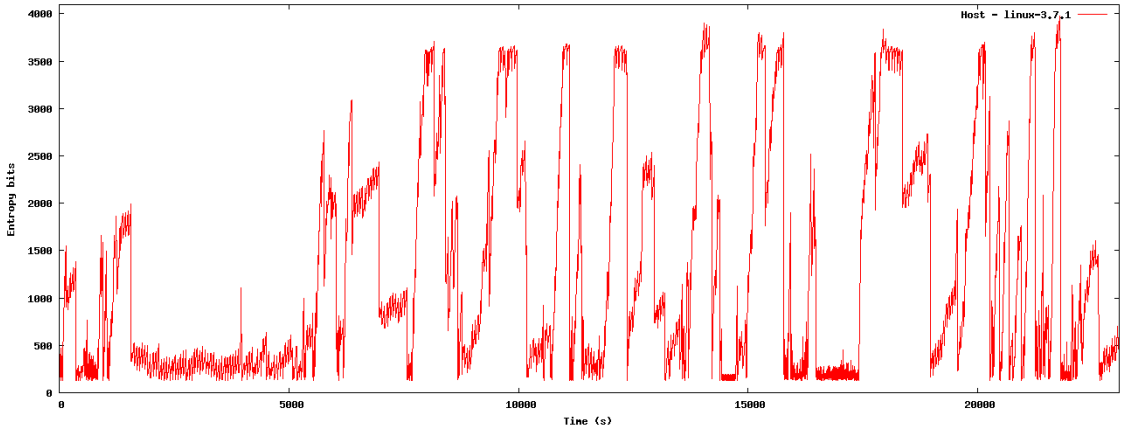


Figure 4.5: Long term usage of a real world system - desktop computer.

4.9 Influence of virtualization drivers

Virtualization acceleration drivers usualy avoid normal code paths to improve throughput of the subsystems with critical performance. The following results analyze the impact of

using accelerated drivers for virtual disk storage on the performance of the LRNG.

4.9.1 QEMU + KVM

From the measurement it's apparent that the VirtIO device driver uses data paths that are comparable to those of the fully virtualized driver. The LRNG was performing similarly when using both the VirtIO and the fully emulated driver. The LRNG on the system under load was producing approximately 5 bits of entropy per second. This result is comparable to the idle state of the host operating system.

The disk I/O performance was approximately twice as good when compared to a system running non accelerated drivers. The accelerated system was able to achieve a sequential read speed of 400MiB/s[1] whereas the guest using the legacy emulated hardware was able to reach 200MiB/s on the tested system. This allows to encourage users to use the accelerated drivers instead of the legacy ones without the need to take the RNG performance into account.

This test also shows that heavy disk I/O improves the performance of the LRNG when using QEMU as a hypervisor when compared with the results from an idle system visible in graph 4.3. On a downside a busy system is more likely to consume more entropy for internal purposes.

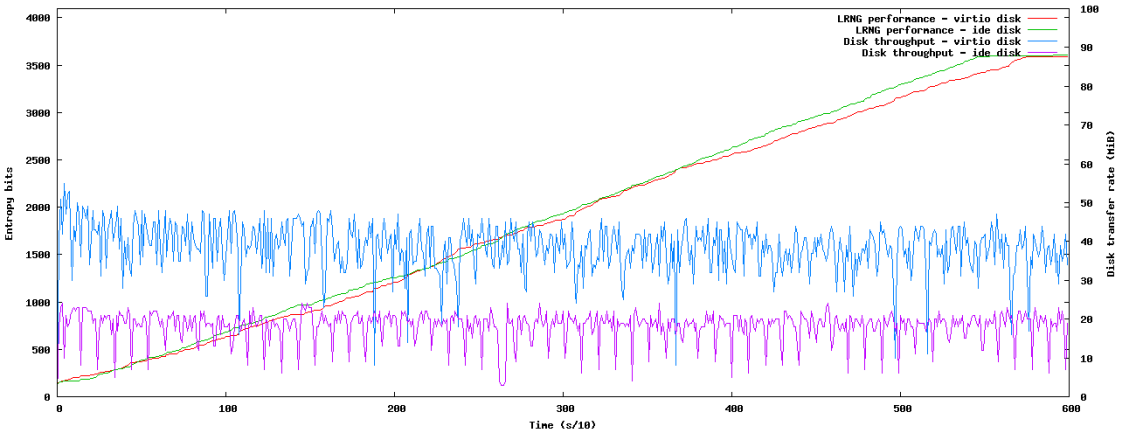


Figure 4.6: QEMU - Influence of disk backend on RNG performance and disk I/O performance.

4.9.2 Xen

When using the Xen hypervisor, the choice of disk driver and backend has direct impact on the performance of the LRNG. With the paravirtualized disk driver the LRNG was performing poorly and it's apparent that the driver is avoiding data paths that are used by the LRNG to gather entropy. The SCSI disk backend is using the standard data paths and thus the kernel is able to extract entropy from these disk reads. The LRNG produced 5 times more entropy when using the SCSI driver compared to the paravirtual driver.

The results of disk bandwidth measurement are unexpected though. The SCSI non-paravirtual disk backend of the Xen hypervisor is performing better than the paravirtual

[1]The graph shows speed in 100ms ticks, thus the throughput per second is 10 times greater

backend. With an SCSI disk the guest was able to read 260MiB/s whereas the paravirtual interface is averaging only 160MiB/s of sequential reading data.

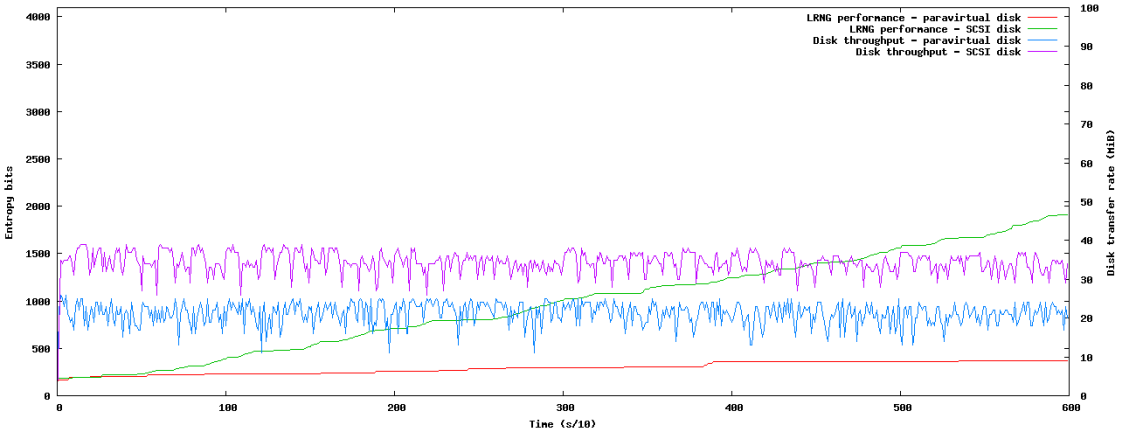


Figure 4.7: xen - Influence of disk backend on RNG performance and disk I/O performance.

4.10 Entropy levels during boot of a Linux system

As expected due to issues described in section 4.7 and the fact that the filling rate of the kernel entropy pool at early boot phases isn't high enough even in cases of heavy disk activity, the entropy pool was almost empty during the boot phase. The filling started after the startup of system services was finished.

Graph 4.8 shows heavy fluctuations of the entropy level during the boot phase. The RNG collects entropy from the heavy disk operations during boot but the init scripts continue draining the pool. After the host boots up at around 25 seconds after the start the entropy generator loses the input but also draining due to process starts ceases. This allows the entropy pool to start slowly filling up.

The data for this test was gathered while running on physical hardware without a hypervisor. As the consumption of entropy is too high during the boot process a test on a virtual environment would yield very similar results.

4.11 Conclusion

According to the test results it's apparent that the performance of the Linux kernel RNG in virtualization guests is poor. There are multiple factors that that cause this.

The first issue is the lack of peripherals in the guest. Virtual machines interact mainly using network services and with exception of maintenance. This drastically reduces the performance of the RNG. Unfortunately this issue cannot be solved in a virtual environment.

The second factor decreasing the performance of the RNG is offloading more and more of the performance critical hardware emulation into paravirtual cooperation with the hypervisor. This has great benefits in improving performance of the guest itself but the guest kernel loses even more sources of noise that are suitable to extract entropy from. Additionally with further progress in speeding up virtualization solutions more and more of the noise sources stop to be suitable. In systems containing multiple guests the offloading procedure might

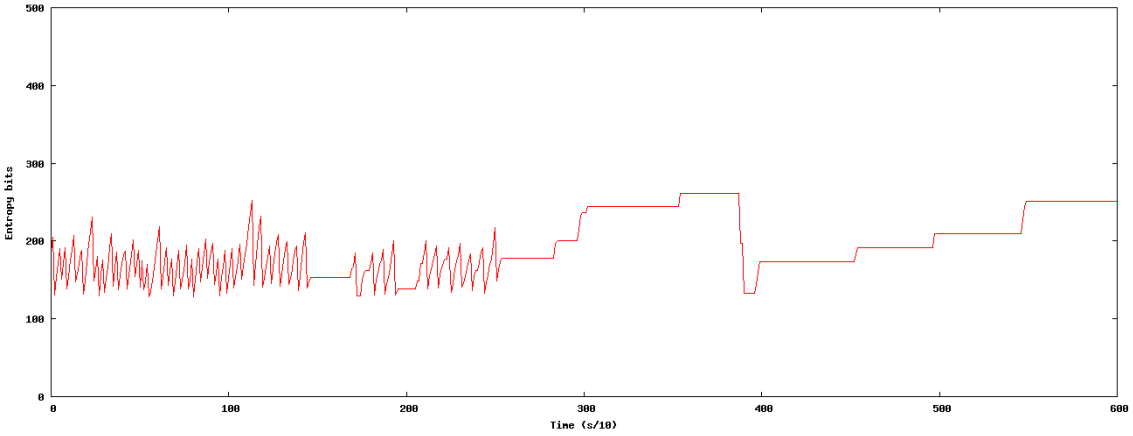


Figure 4.8: Entropy level variations during early booting phases.

start introducing same noise patterns in multiple guests enabling attackers to guess random streams produced by RNGs in other virtual machine running on the same host.

The consumption of entropy needed to initialize every single process in the Linux operating system is the most critical factor that causes very low levels of entropy available in the kernel pools. The entropy is used as a security measure to protect application against stack corruption and is used to seed internal PRNGs. Also address space layout randomization, a technique to avoid buffer overflow and known function pointer attacks requires access to a good entropy source[14].

Chapter 5

Approaches to improve levels of entropy in guests

The research summarized in the previous chapter shows that the performance of the random number generator is poor in systems used today, but the need for entropy data is growing.

This chapter will describe possible approaches we can take to improve the level of entropy in the kernel of guest operating systems. Each of those approaches will be categorized and judged for suitability for various use cases. Then one of the approaches will be chosen to be implemented and the design of the product will be described.

5.1 Gathering of additional entropy in the guest

The most basic approach to improve the slow speed of the LRNG would be to gather entropy from events and user interaction in the system in addition to the LRNG. A carefully designed system would allow to specifically extract entropy from events that are common and random enough. Unfortunately this would require fine tuning the entropy extraction system for a particular purpose and the same system might not perform well in other use cases.

It would be also possible to extract entropy from other hardware devices that are present or passed through to the guest system in addition to those used by the LRNG. On the other hand this hardware can't be used in multiple virtual machines or in the host simultaneously for the same purpose as an attacker could take advantage of the knowledge of the state to reverse engineer the bit stream from the entropy gathering method.

5.2 Passthrough of host's entropy to the guest OS

If the performance of the host's RNG is sufficient the entropy could be passed through to the guest to improve the contents of it's kernel entropy pool. This approach doesn't require adding new sources of entropy or any other complex infrastructure.

As a downside according to section 4.8 this approach doesn't scale well as the performance of the RNG in the host is barely sufficient to cover consumption of the host itself. Also with direct passthrough the guest is also able to easily starve the host operating system of it's entropy.

5.3 Gathering of additional entropy in the host

As an extension of the previous scenario it would be possible to gather additional entropy in the host either from software and user interaction sources or from hardware sources as the host has access to all the physical hardware. The kernel entropy pool can also be seeded from a hardware RNG device present in the host and thus be suitable for passing the entropy to the guest.

5.4 External sources of entropy

Device passthrough would allow passing through a physical hardware random number generator to the guest. This would allow the guest to use it as if it was directly connected to a physical machine. The guest will then able to seed the kernel entropy pool from such a source.

The infrastructure for such passthrough is already existing and hypervisors allow to assign PCI, USB and other devices directly to the guest. The disadvantage of such approach is that the device can't be used from the host or any other virtual machine and migration [1] would be impossible.

5.5 Distribution system for multiple guests

The approaches described above are not fully suitable to be directly used on server systems and infrastructures that run multiple guest virtual machines.

The approach of gathering entropy in the guest may not provide sufficiently random data if the source of entropy is common to all virtual machines. Also the gathering software might unnecessarily load the systems and thus decrease the capacity of the host system.

The other approaches provide a single source of entropy while there are multiple consumers. This creates a competitive environment where one guest could starve others by consuming too much entropy. This creates a need to introduce a distribution system for entropy that will allow to configure flow rates and shape entropy flows to excessive consumers. The goal will be to create a fair environment.

This approach would also allow to use a single hardware RNG device on a host or one central one for a datacenter and distribute entropy to all the guest virtual machines on all the hosts to save costs and resources.

5.6 Design of the system to improve entropy in guests

As a solution to improve the performance of guest's random number generator that will be implemented for the purpose of this thesis the distribution system. The distribution system approach can be used with multiple approaches to generate random data and will allow to use the system in large infrastructures. The following sections describe the desing of the four main parts of such a system.

The platform of cohoice on which this sytem will be implemented is the QEMU hypervisor with libvirt used as the management interface. I chose this platform as it's open source and thus will allow to modify the source code of the components for direct integration.

[1] Migration allows to move a running virtual machine between different physical hosts with no loss of service

Without source code access, this work would be limited to gathering of entropy data in the guests, which wouldn't be useful and couldn't be implemented into an existing project to ensure wide availability of this feature.

The system designed as a part of this thesis will allow to source, control and distribute entropy to guest operating systems running as virtual machines as their native entropy generators are performing poorly.

5.6.1 Entropy source

The system will be designed to support multiple sources according to the needs and configuration of the user. This will allow to use such a system also in enterprise environments as the source can be configured according to the requested certification level as described in section 2.3.

The system will allow to use software entropy sources such as the entropy gathering daemon, hardware RNG integrated into the host or external ones. The system also be prepared to support sourcing of entropy using the RDRAND instruction[3] in modern CPUs or network sources if the user will develop a use case for this functionality.

Possible entropy sources and their description can be found in chapter 2.

5.6.2 Passthrough layer

One of the most critical parts of getting entropy into the guest is the passthrough point to the guest. The hypervisor running on the host computer has to provide means to allow this.

The hypervisor can achieve this in multiple ways. The legacy way to achieve this would be to emulate a physical RNG device in the hypervisor and intercept I/O requests to the devices address space. This is both complex and slow in the result.

Second option is to use paravirtual interface. This introduces the need for a specific driver in the kernel of the guest operating system, but the much simpler interface between the host and guest allows for better performance.

For the purpose of this thesis, the VirtIO RNG virtual hardware and Linux kernel driver will be used as the passthrough layer to the guest. The QEMU hypervisor will then communicate using the character device backend with services in the host to source entropy.

VirtIO RNG was chosen as the kernel driver for the virtual device is already part of the upstream linux kernel for a longer period and thus it can be used in existing systems too. The need to run a modified kernel as a solution is undesirable as it would discourage adoption of this system in production systems.

5.6.3 Distribution layer

The task of the distribution layer will be to request entropy from a entropy source and deliver it to one or more virtual machines. When doing this the layer will have to ensure that the guests are not starving other guests in case of mailicious behavior. This layer will need to interact with the passthrough layer and ensure that it will work even if the management layer is not working for some reason. The distribution layer should be integrated in the virtual machine management software for easy deployment and availability.

There are no open source projects that would implement this functionality thus the distribution layer needs to be implemented from scratch for the purpose of this thesis.

5.6.4 Management layer

The management layer selected for purpose of this thesis is `libvirt`. Libvirt allows to run and easily manage virtual machines run by the QEMU hypervisor.

As a part of this thesis libvirt will be augmented to support management of entropy pools, to allow configuration of virtual RNG devices for virtual machines and integrate the distribution layer instrumentation.

The aim is to integrate all the components into the upstream development tree of libvirt so that the results of this work can be used in production with existing software packages that are commonly used and readily available in Linux distributions.

Chapter 6

Implementation

This chapter will elaborate on the implementation details of the entropy distribution system that was created for purpose of this thesis and described in section 5.6.

6.1 VirtIO RNG

The VirtIO RNG device is a paravirtual device designed to supply entropy to the guest. This will be the passthrough layer of the complete system. VirtIO RNG creates a virtual PCI device in guest's I/O address space that is recognized by the kernel driver and presented as a hardware RNG device. After the driver is loaded, the device is in the Linux operating system available as `/dev/hwrng`. The read requests done on this device invoke the internal backend in the hypervisor that sources the entropy from the configured source and subsequently returns the data to the guest.

6.1.1 QEMU backends

The entropy interface backends are used as internal representation and abstraction of the two possible sources of entropy for a RNG device in the QEMU hypervisor.

`random` backend

This backend is designed to source entropy from character devices like `/dev/random` or similar interfaces. It's the most simple backend. It does not use any specific protocol. When entropy is requested in the guest QEMU accesses the configured character device and reads the entropy data.

Qemu supports reading from arbitrary files including `/dev/urandom`. This is considered not a good idea as the guest doesn't expect to receive pseudo-random entropy from a hardware RNG. [7]

The following command line options are needed to activate this backend along with a RNG device:

```
qemu -object rng-random,filename=/dev/hwrng,id=rng0 \
     -device virtio-rng-pci,rng=rng0
```

EGD backend

The EGD backend is more advanced. The backend uses a simple protocol described in section 6.1.2 to communicate with a suitable network service and uses the protocol to request entropy data.

The communication is accomplished using QEMU's `-chardev` interface. The `-chardev` interface is an abstraction that can be used to communicate over the network, UNIX domain sockets, pipes or be channeled to a file on disk. This allows the backend to be used universally and suitable for multiple use cases.

Command line options used to enable the egd backend are more complex compared to the `random` backend:

```
qemu -chardev socket,host=localhost,port=1024,id=chr0 \
     -object rng-egd,chardev=chr0,id=egd0 \
     -device virtio-rng-pci,rng=egd0
```

6.1.2 EGD protocol

The EGD protocol is a simple network protocol that was designed for entropy distribution in user space. The protocol was created as a part of the entropy gathering daemon project. The protocol is not standardized as an RFC standard, but it's well known and commonly used.

The protocol is simple, binary and stateless. The communication is always initiated by the client. The protocol has 5 commands. Each of the commands is described by the first byte of a message with more optional data according to the message type. The returned message type depends on the command itself.

Command `0x00`

The command has no arguments. The returned message is a 32bit integer in big endian byte ordering containing the number of available entropy bits present in the pool.

Command `0x01`

The 0x01 command is a non-blocking entropy read request. The command has a one byte argument 0xNN for the amount of entropy in bytes that is requested. The returned message is in format `0xMM` followed by 0xMM bytes of entropy where 0xMM is the number of bytes granted by the daemon.

Command `0x02`

Command `0x02` is used for a blocking entropy read request. The argument is one single byte number 0xNN denoting the requested amount of entropy. The returned message contains 0xNN bytes of entropy data and the message is expected to block until the requested amount of entropy can be delivered.

Command `0x03`

The command has multiple arguments 0xMM 0xLL 0xNN followed by 0xNN bytes of entropy. This command denotes a write to the entropy pool of the daemon. The 0xMM

0xLL argument is a 16 bit big endian number of the count of entropy bits contained in the following string. The command has no reply message.

Command `0x04`

To determine the PID of the daemon the 0x04 command can be used. The PID is reported as 0xNN followed by a string of length 0xNN bytes containing the PID of the daemon.

Commands used by QEMU

The implementation of the EGD protocol in the QEMU RNG backend only uses a very limited subset of the protocol. Only the `0x02` command to request blocking entropy is used with an argument of `0x40` bytes of entropy requested.

6.1.3 Rate limiting

The QEMU hypervisor supports basic rate limiting support that allows to limit the flow rate for a single guest. This approach unfortunately doesn't have global knowledge of the system and other virtual machines thus this can't control the flow fairly and similarly doesn't avoid starving the host by multiple guests.

Rate limiting is enabled by adding configuration options to the virtual PCI device definition:

```
qemu -device virtio-rng-pci,max-bytes=1024,period=1000
     ...
```

6.1.4 Availability of VirtIO RNG

The VirtIO RNG device was introduced into the upstream repository by commit `16c915ba42b45` on November 16, 2012 and is available in QEMU-1.0.3 release. The code was written by Amit Shah.

6.2 Basic `libvirt` support for RNG devices

Libvirt stores the machine configuration options in XML documents and then uses them to populate internal structures. Those are then used to generate native configuration options for the hypervisor that are specific for hypervisor drivers.

6.2.1 Configuration file format

Libvirt's domain[1] configuration XML document is described using Relax-ng schema definition. The schema is used to validate configuration documents and serves as a guideline to implement the parser and generator.

The domain XML document contains multiple sections that describe various aspects of the virtual machine. The `<devices>` section is reserved for definitions of virtual hardware devices presented to the system. This section will be the place where the users will be able to add RNG devices to the guest.

[1] Libvirt describes guest machines as domains. This is a legacy name introduced by the XEN hypervisor.

The RNG device is represented with the `<rng>` tag. Configuration options for the RNG device are represented as sub-elements.

To use basic rate limiting that is supported by the QEMU hypervisor, the user may add the `<rate>` element with appropriate values. The entropy consumption limit is configured by the `bytes` attribute. The `period` attribute represents time in milliseconds after which the limit is refreshed.

The `<backend>` element allows to configure the source of entropy for the RNG device in the guest. There are three possible backend models implemented: `random`, `egd` and `pool`.

`random` backend

This backend has only one configurable parameter: the file name of the entropy source. The valid file names for this backends are `/dev/random` and `/dev/hwrng`. This is a subset of the interface provided by QEMU. This limitation was introduced after upstream discussion in the mailing list thread [7] as a workaround to disallow insecure configurations that used might do by mistake. In case the source file name is omitted `/dev/random` is used as the default.

```
<rng model='virtio'>
  <backend model='random'>/dev/random</backend>
</rng>
```

Figure 6.1: Excerpt from guest configuration XML. RNG device with the default `random` backend and `/dev/random` as a source.

`egd` backend

This backend configures the hypervisor to use a configurable character device to communicate with a remote side using the EGD protocol. Libvirt already provides support for configuring and using character devices in virtual machine configuration. This interface was adapted to be reusable and used to parse and generate the EGD backend code. The configuration options include backend type, addresses and file names. The format of the character device XML description is explained in libvirt's documentation.

This allows to use the EGD backend with unix, TCP and UDP connections, log files and POSIX pipes according to the need of the application.

```
<rng model='virtio'>
  <rate period="2000" bytes="1234"/>
  <backend model='egd' type='udp'>
    <source mode='bind' service='1234'>
    <source mode='connect' host='1.2.3.4' service='1234'>
  </backend>
</rng>
```

Figure 6.2: Excerpt from guest configuration XML. RNG device using the `egd` backend with rate limit enabled and using UDP transport

`pool` backend

The `pool` backend configures the guest to source entropy from the entropy pool managed by libvirt. The hypervisor will be automatically configured appropriately to use the entropy pool.

Arguments for this backend allow to configure the pool name used to source the entropy and a distribution class to be used with the host in question. When starting a guest, the hypervisor driver will have to verify that the configured entropy pool is existing and started and the desired class exists in the configuration.

```
<rng model='virtio'>
    <backend model='pool' name='default' class='hostclass1'/>
</rng>
```

Figure 6.3: Excerpt from guest configuration XML. RNG device using the `pool` backend with the `default` pool belonging to the `hostclass1` class.

6.2.2 XML parser and internal structures

The next step in adding a device support into libvirt is to augment the XML parser and formatter and internal data structures to accept the data. As the libvirt library is written in the C language, this step unfortunately isn't automated by parsing the schema definition and generating the data structures according to the definition.

Internal data structure associated with a RNG device

For internal purposes, libvirt stores configuration definitions in internal data structures, while XML files are used for external representation and storing of the state. To describe a RNG device I introduced `struct _virDomainRNGDef`.

This structure holds information about the model of the RNG device, backend type and backend related data. The RNG device type and backend model are described by `enum virDomainRNGModel` and `virDomainRNGBackend`. All of the above data types are defined in `src/conf/domain_conf.h`.

XML parser

To parse the definition of the RNG device multiple XPath queries are used and evaluated using the `libxml2` parser used by libvirt. The queries extract needed information from the document and additional code is used to validate the parsed data. The parser is implemented by the `virDomainRNGDefParseXML` function that is defined in `src/conf/domain_conf.c`.

XML formatter

The XML formatter used in libvirt is created manually similar to the parser. The XML document is created by directly outputting the code instead of generating a DOM tree [2]. The definition of the RNG device is formatted function `virDomainRNGDefFormat` defined in `src/conf/domain_conf.c`.

[2] This complies to the coding guidelines of libvirt.

Cleanups of device handling

When adding a new device type into libvirt there are multiple places that need to be adapted in order to add the support correctly and avoid leaking memory and other problems.

As pre-requisite work, these places were cleaned up and changed so that the compiler produces warnings in cases where a new device type is added but the handler code is not updated. This simplifies future work on libvirt and makes it less bug prone.

6.2.3 Qemu driver support for RNG devices

After libvirt is able to recognize and parse a new device type, the support for this device needs to be implemented into the hypervisor driver. The driver is responsible for creating hypervisor specific native configurations. In case of the QEMU hypervisor the driver translates the internal data structures into command line arguments. The QEMU command line arguments that are used to enable the virtio RNG device are described in section 6.1.1 and are generated by `qemuBuildRNGDeviceArgs` and `qemuBuildRNGBackendArgs` defined in `src/qemu/qemu_command.c`.

6.3 Libvirt support for entropy pools

The next step is to introduce entropy pool support to libvirt. The purpose is to have the ability to configure, control and use the entropy pools as a part of the management interface.

Libvirt uses a modular loadable driver architecture to support multiple approaches for a common task or hypervisor specific approach. Adding entropy pool support results in adding a new driver type and infrastructure to support it and then implementing a driver instance to support `virtentropyd`.

The driver provides APIs used to manage and configure entropy pools. The purpose of the underlying driver implementation is to create specific configuration and start the appropriate services and manage their lifecycle.

6.3.1 API of the entropy pool driver

The public API is the main interface between the user and libvirt. The entropy pool will export the following function in order to allow effectively managing pools from a management application.

`virConnectListAllEntropyPools`

This function is used to list all entropy pools managed by libvirt. The return value contains a list of entropy pool objects that can be used in the API functions manipulating the pools. The legacy listing functions that were implemented by other drivers and are returning a list of names instead of an object list are not implemented by the driver as the design is obsolete.

`virEntropyPoolDefineXML`

This API call is used to create a new persistent entropy pool according to the definition stored in the passed XML. According to common libvirt semantics this API is also used to change the definition of an existing pool by defining an updated XML definition. The updated XML definition has to share the same pool name and UUID.

Defining of a new pool or updating of a existing one will emit an asynchronous libvirt event to notify clients.

`virEntropyPoolUndefine`

To remove a existing pool definition from the libvirt configuration the user has to invoke the Undefine API. This call removes the internal state and all private configuration files associated with the pool.

The entropy pool needs to be inactive at the time of undefining it as the support for transient[3] pools will not be implemented in this thesis.

`virEntropyPoolGetXMLDesc`

This API call can be used to retrieve the definition of an entropy pool that was already stored by libvirt. The definition is returned as a string containing the XML document.

`virEntropyPoolCreate`

When a entropy pool is defined it is not yet active. To activate a pool, the user has to call this API. Libvirt will then load the configuration and start the entropy pool.

`virEntropyPoolDestroy`

This function can be used to deactivate an active entropy pool. The destroy call is not graceful by default and will immediately terminate all operations happening on a pool. This behavior can be controlled using the flags argument.

`virEntropyPoolLookupBy*`

To look up a entropy pool object according to one of the unique identifiers the user has to invoke this API call. An `virEntropyPoolPtr` is returned that can be then used to manipulate the pool. The pools can be looked up using either the name or UUID.

6.3.2 `virsh` commands

Each API expansion of libvirt requires implementing the new API functions into the `virsh` virtualization shell. The interface of the entropy pool driver was exposed as commands starting with `entropy_pool` prefix. The commands are implemented in `tools/virsh-entropypool.c`.

The commands implement basic management capabilities for entropy pools and allow to test the implemented API without the need to implement a separate application. Until other management applications implement support for entropy pools this will be the primary way to configure entropy pools.

6.4 `virtentropyd`

The `virtentropyd` daemon represents the distribution layer in this system. The daemon is responsible for opening and managing a entropy source, opening channels to the virtual machines and supplying entropy to them and shaping the flow of entropy in the case a client is consuming more entropy than configured.

[3] Temporary.

```
virsh # help entropy_pool
 entropy pool (help keyword 'entropy_pool'):
    entropy_pool-define            define or update a entropy pool from an XML file
    entropy_pool-destroy           destroy a active entropy pool
    entropy_pool-dumpxml           entropy pool information in XML
    entropy_pool-edit              edit XML configuration for a entropy pool
    entropy_pool-list              list entropy pools
    entropy_pool-start             start a (previously defined) inactive entropy pool
    entropy_pool-undefine          undefine a entropy pool
```

Figure 6.4: virsh help output for entropy pool management commands

6.4.1 Configuration

Initial configuration of the virtentropyd daemon is really simple. The virtentropyd daemon is configured using command line arguments. This allows to start a instance that will be serving requests of a single entropy pool instance. The parameters used to configure virtentropyd are the libvirt connection URI used for the connection and the pool name.

The main configuration of the entropy source, shaping classes and possible outputs that is stored in libvirt's internal structures and as a XML file is then loaded using the `virEntropyPoolGetXML` method directly from the libvirt daemon. This simplifies the interface and avoids having a separate place to store the configuration.

The configuration loaded via the libvirt connection is then parsed into internal structures and the daemon is initialized. Failure to establish the initial libvirt connection is fatal, but after the configuration is loaded the connection may break subsequently.

6.4.2 Communication with libvirt

The entropy distribution daemon is designed to work as a pure libvirt client. Apart from loading the pool configuration XML the libvirt connection is used to receive asynchronous events about virtual machine life-cycle and configuration status.

After an event regarding a guest is received virtentropyd determines if a change of state is needed according to the configuration of the guest. This asynchronous interface is ideal for this type of communication and the pure-client approach avoids the need to create specific RPC protocols for virtentropyd.

6.4.3 Sourcing of entropy bits

As a initial implementation virtentropyd supports only character devices as backends. The main purpose will be to connect to `/dev/random` or `/dev/hwrng` and use this as the source. In the future additional sources may be added according to common usecases of the distribution system.

Each instance of the distribution daemon supports a single source of entropy. This will initially simplify the design and the extension to a multi-source multiplexing architecture may be added later if it will be desired.

6.4.4 Shaping of entropy requests

To control flow rates of entropy to the host, `virtentropyd` uses hierarchical token bucket algorithm. This allows to create hierarchical structures that are used to to specify the peak

and sustained read rates for separate guests or groups of guests.

6.4.5 Distribution to the guest

`virtentropyd` connects to the unix socket created by the QEMU processes running a virtual machine and uses the EGD protocol to communicate with the RNG device backend.

When a request for entropy is received from the guest the handler thread is woken up. The thread looks up the origin of the request and determines the shaping classes in the path to the entropy source. If the limits on the path to the source are enough to cover the request of the guest the entropy is read from the source and written to the socket of the guest. The cycle repeats then from the beginning.

In case the limit for entropy consumption was reached by a host a timer is started that will wake up the handler thread after the correct amount of time that will be needed to refill the quota for a guest.

6.5 Integration of `virtentropyd` into libvirt

The `virtentropyd` source code was integrated into the libvirt source tree and the daemon is being built along the other binaries contained in the project. The source file is located in `src/entropy/entropy_daemon.c`.

To bridge the interaction between `virtentropyd` and libvirt an instance of the entropy pool driver was created. This instance is a stateful driver that starts and manages `virtentropyd` processes when entropy pools are created or destroyed. The driver implements the API introduced by adding the pool support (6.3.1).

The implementation of the entropy pool driver follows the coding guidelines [8] of the libvirt project and is located in `src/entropy/entropy_driver.c`. The driver is built as a loadable module and is automaticaly loaded into the libvirt daemon on startup. No other configuration is needed to start using entropy pools.

6.6 Documentation

The guidelines of submissions to the libvirt project require that documentation is added with each change. The parts of this thesis that already were accepted contain documentation in the upstream repositories. For the code that was not yet accepted for addition to the upstream repository documentation was not created yet. Open source projects with active comunity usualy propose design changes as a part of the submission process which would require changing the documentation every time, thus the documentation is usually created as the final step before merging the new feature into upstream.

Chapter 7

Impact analysis of the system

The entropy distribution system implemented for the purpose of this thesis will now be tested in regarding of improvement of entropy contents in the LRNG kernel pool and the usability of the system.

7.1 Testing approach

The infrastructure and approach used to test the implementation will be similar to the one described in section 4.4. The guest will be started with the VirtIO RNG device enabled and the data will be supplied from a hardware random number generator integrated into the system-on-chip CPU of a Raspberry Pi embedded system. Using a network connection entropy will be transported to the host running the virtualization.

The guest will be running the `rngd` daemon that is used to seed the kernel entropy pool from external sources. This daemon is included in the `rng-tools` software package. The `rngd` daemon will be confiugured to sample the state of the entropy pool each second and seed the pool in case the minimum threshold is underrun.

In addition to the experiment with an idle guest, the entropy levels will also be monitored while periodically starting processes inside the guest to drain the entropy pool. As the `rngd` daemon has a internal polling interval for the state of the entropy pool, it's expected that the entropy levels will fluctuate periodically.

To test the influence of this system during the boot of a virtual machine will be configured to start the `rngd` daemon as a service while booting and the experiment done in section 4.10 will be re-run to verify the results. The state of the entropy pool will be monitored by the `entropy_boot.c` program and graphed for visual representation.

7.2 Results

The results were collected on a host system using the QEMU-1.0.4 hypervisor running under management of the libvirt library with changes done for the purpose of this thesis. The hardware used was a laptop with the Intel Core i7 processor.

7.2.1 Long term performance

The guest operating system is able to maintain high levels of available entropy in the pool when the passthrough device is in use. On an idle system the entropy pool fills up to the maximum level of 4096 bits within two seconds and the level is maintained forever.

In case of active consumption of entropy in the system for example by periodicaly starting processes, entropy from the pool is consumed but the `rngd` daemon is able to steadily refill the pool from the entropy source passed to the guest. The oscillations in graph 7.2.1 are caused by starting 10 processes per second and the `rngd` daemon was checking the contents of the entropy pool once per second.

The overal performance of the system was improved by orders of magnitude thanks to a steady external entropy source that is designed as such and doesn't have to extract entropy from deterministic systems.

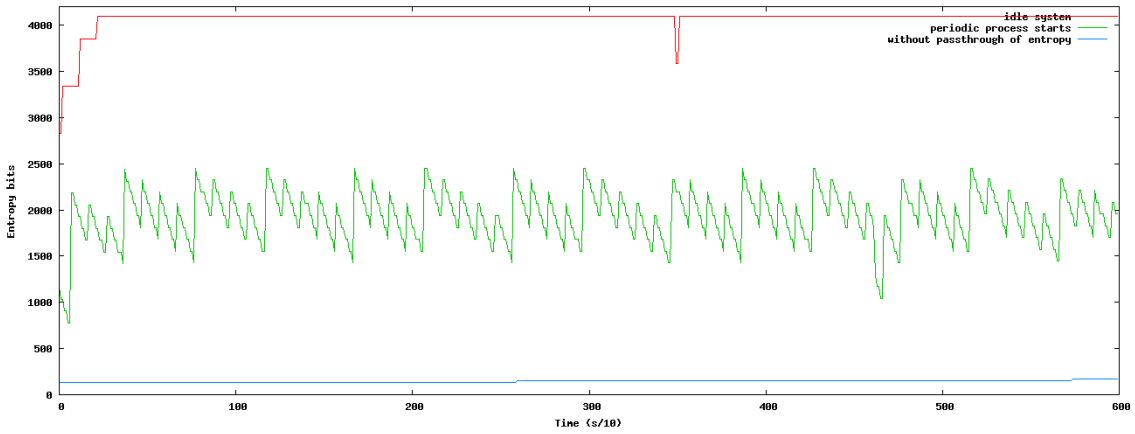


Figure 7.1: Entropy pool levels with entropy distribution in used

7.2.2 Boot of the operating system

When the machine is booting the service used to seed the entropy pool is started late in the boot process. This creates a dead period at the beginning when the RNG pool behaves similarly to the non-improved approach.

After the `rngd` service was started but the guest was still booting the entropy pool contents were sharply changing roughly according to the 1 second polling interval of the pool filling procedure. Between individual filling steps the pool was again drained by the amount of processes started by the init scripts. After the startup of the guest finished the pool was able to quickly fill up to the maximum.

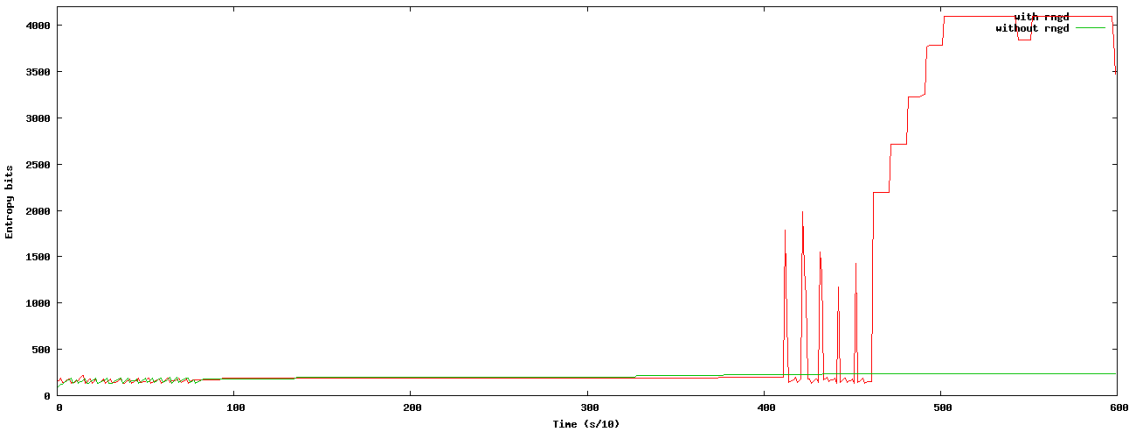


Figure 7.2: Boot of a guest system that uses entropy passthrough

Chapter 8

Conclusion

The analysis of the performance of the LRNG showed that even on physical hardware the performance of the generator is insufficient compared to the demand of entropy in usual systems. Apart from applications needing entropy for cryptographic purposes the biggest consumer of entropy in a Linux system is the kernel while seeding each newly started process with initial entropy used for stack protection and other purposes.

In an virtual environment the LRNG loses sources such as user interface peripherals and accelerated virtual hardware drivers sometimes avoid code paths ususally used to extract entropy and thus degrade the performance of LRNG performance even more. Virtual machines are nowadays deployed more extensively than physical hardware for cost and security separation benefits. Applications running in such environment may suffer from inability to source entropy for cryptographic or simulation purposes.

As a part of this work a system was implemented purpose of which is to introduce more entropy to a guest virtual machine from the host's pool or an hardware random number generator by passthrough and ensure fairness in the distribution. The system consists of a virtual RNG device in the hypervisor and support in the management application to configure it and ensure entropy distribution. The system proposed is now partly included into the libvirt virtualization library upstream repositories.

With use of the system created for purpose of this thesis, kernel entropy pool levels in virtual machines could be improved and it was proved that an integration of this system into libvirt is possible. This will allow to adopt the system in production environments as QEMU and libvirt are commonly used solutions for virtualization and are used in projects as oVirt, Open Stack and even for standalone libvirt users.

The choice of entropy source was summarized but the final decision has to be made by the end user. The user has to determine needs for quality of the entropy source and it's performance and choose one that suits the needs and requirements of the application.

8.1 Future work

The system developed as a part of this thesis will be used as the part of the Red Hat Enterprise Virtualization (RHEV) product. The main focus will be now to test the implementation and find possible problems and provide support for RHEV customers.

8.1.1 Upstream acceptance

Some parts of the code developed for purpose of this thesis were not yet accepted into upstream development repositories. The future plan is to work with the upstream community to finalize the design of the components and reach acceptance into the libvirt project. One of the next goals will be to rise awareness that such functionality exists and is ready to use.

8.1.2 RDRAND emulation support for QEMU

After introduction of the `RDRAND` instruction[3] into the `x86_64` architecture by Intel, QEMU could implement support for emulating this instruction on processors where the instruction is missing. The emulated `RDRAND` instruction would then use the same sources as QEMU already has for VirtIO RNG with the existing infrastructure. This would then allow to use the benefits of the distribution system that was implemented as a part of the thesis with this functionality too.

This would allow to use this instruction in heterogeneous hardware clusters that may contain nodes that don't support RDRAND and would also allow migration of virtual machines between such hosts.

`libvirt` would then add this functionality as a new RNG device model with minimal changes to other code.

8.1.3 Kernel-space entropy pool seeding

Currently the LRNG uses only the approaches described in section 2.2.4 to seed it's contents and provide the entropy. To seed the kernel entropy pool from a different source a separate user-space daemon is required. For a better adaptation of properly seeded RNGs in the operating system the kernel could use internal routines to seed the pool from available external sources such as VirtIO RNG or the RDRAND instruction.

This would require adding code to the kernel that would act similarly to `rngd` in userspace.

Bibliography

[1] Elaine Barker and John Kelsey. Recommendation for random number generation using deterministic random bit generators. http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf, 2012.

[2] Kees Cook. ELF: implement ATRANDOM for glibc PRNG seeding. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f06295b44c296c8fb08823a3118468ae343b60f2, 2009. linux.git commit f06295b44c296c8fb08823a3118468ae343b60f2.

[3] Intel corporation. Intel digital random number generator (drng). http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R_DRNG_Software_Implementation_Guide_final_Aug7.pdf.

[4] Randall J. Easter and Carolyn French. Annex c: Approved random number generators for fips pub 140-2, security requirements for cryptographic modules. http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf, 2012.

[5] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the linux random number generator. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1624027&isnumber=34091, 2006. vol., no., pp.15 pp.-385, 21-24.

[6] M. Tim Jones. Virtio: An i/o virtualization framework for linux: Paravirtualized i/o with kvm and lguest. http://public.dhe.ibm.com/software/dw/linux/l-virtio/l-virtio-pdf.pdf, 2010.

[7] Anthony Liguori. Re: [Qemu-devel] virtio-rng and fd passing. https://lists.gnu.org/archive/html/qemu-devel/2013-03/msg00165.html, 2013.

[8] Libvirt maintainers. Implementing a new api in libvirt. http://libvirt.org/api_extension.html.

[9] A. Theodore Markettos and Simon W. Moore. The frequency injection attack on ring-oscillator-based true random number generators. http://www.cl.cam.ac.uk/~atm26/papers/markettos-ches2009-inject-trng.pdf.

[10] Merriam-Webster dictionary. Entropy. http://www.merriam-webster.com/dictionary/entropy.

[11] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. http://csrc.nist.gov/groups/ST/toolkit/rng/documents/SP800-22rev1a.pdf, 2010. Special Publication 800-22.

[12] Claude E. Shannon. A mathematical theory of communication. http://www.alcatel-lucent.com/bstj/vol27-1948/articles/bstj27-3-379.pdf.

[13] Simtec Electronics. The entropykey: The technical stuff. http://www.entropykey.co.uk/tech/.

[14] The PAX Team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[15] The virt-manager community. Virtual machine manager. http://virt-manager.et.redhat.com/index.html, 2012.

[16] Florian Weimer. [security] [dsa 1571-1] new openssl packages fix predictable random number generator. http://lists.debian.org/debian-security-announce/2008/msg00152.html.

[17] Eric W. Weisstein. Linear congruence method - from mathworld–a wolfram web resource. http://mathworld.wolfram.com/LinearCongruenceMethod.html.

List of Figures

Appendix A

Contents of the attached CD

Directories:

- **docs** - this report including source files
- **data** - data sets used to create graphs in this thesis including scripts
- **libvirt** - libvirt source git repository including code done for this thesis
- **tools** - other source files
- **literature** - copies of publicly available literature

Appendix B

Glossary

- **API** - aplication program interface
- **egd** - entropy gathering daemon
- **ELF** - executable and linkable format, a format of binary executable files
- **guest** - virtual machine running on a virtualization host
- **host** - Host computer, physical device that runs virtualization
- **hypervisor** - software that creates
- **KVM** - kernel based virtual machines
- **libvirt** - C library used as virtualization management abstraction
- **LRNG** - The Linux kernel Random Number Generator
- **PRNG** - Pseudorandom Number Generator
- **RDRAND** - CPU instruction to request entropy
- **rngd** - software used to fill the Linux entropy pool from external source
- **RNG** - Random Number Generator
- **TRNG** - True Random Number Generator
- **UUID** - universally unique identifier
- **VPN** - virtual private network
- **XPath** - language used to access and modify XML documents using the object model