Federal Office
for Information Security

Analysis of Random Number Generation in Virtual Environments

Abstract

The evaluation of the suitability and quality of cryptographic mechanisms is tasked to the Federal Office for Information Security (BSI – Bundesamt für Sicherheit in der Informationstechnik) in Germany. The BSI therefore initiated this study about the generation and collection of entropy in virtual machines and virtual environments. Virtual machines are increasingly used especially in Cloud-based solutions, covering sensitive areas in enterprises as well as in government. Good random numbers require one or more noise sources supplying entropy which implies that these noise sources are a vital requirement for the security of electronically processed data.

Operating systems use various noise sources which may exhibit properties and behaviors which may deviate significantly when used on a bare metal system or within a virtualized environment. This study analyzes the impact of virtual environments on the presence of entropy for noise sources. The goal of this study is to identify measures for using noise sources in virtual environments in such a way that they collect sufficient entropy.

Besides conducting an analysis of the general impact of virtual environments on noise sources, this study discusses the Linux random number generator of /dev/random and /dev/urandom which includes several noise sources. Also, this study evaluates possibilities of receiving entropy from the virtual machine monitor (VMM) as well as noise sources which collect entropy independently from a virtual environment. Again, the goal is to obtain sufficient entropy in virtual environments. The quality of the Linux random number generator is assessed when executing it in the VMMs of KVM, VirtualBox, Microsoft Hyper-V and VMWare ESXi.

As a summary, the major finding of this study is that all assessed VMMs depending on their configuration, allow Linux to obtain sufficient entropy. The different noise sources of the Linux random number generator, however, operate with varying quality which implies that depending on the use case issues may arise. For example, the quality of the generated random numbers after system boot is questionable. With the provided questionnaire, users are able to analyze whether they are affected by such issues and to what extent.

Software-based noise sources which require hardware support for obtaining entropy are most likely to be adversely affected by a VMM operation. Such noise sources should therefore be assessed in detail for its applicability to a virtualized environment. Hardware noise sources are commonly unaffected by a VMM. With an appropriate support mechanism, a VMM may even deliver entropy to guest systems.

The provided analysis starts with the assessment of the architecture of various noise sources. This is followed by a study of the impact of virtualization on the obtained Entropy and applies the findings to the Linux random number generator.

Authors

Stephan Müller, atsec information security GmbH

Gerald Krummeck, atsec information security GmbH

Helmut Kurth, atsec information security GmbH

BSI-Reference

BSI Title (German): Analyse der Zufallszahlenerzeugung in virtualisierten Umgebungen

BSI Project Number: 213

Document History

Version	Date	Author(s)	Change log
1.0	2016-10-21	Müller, Stephan	First release of document

Table of Contents

List of Figures

1 Introduction

1.1 Summary

Random numbers should not be generated with a method chosen at random.

Donald E.Knuth
The Art of Computer Programming

Cryptographic mechanisms are essential for ensuring privacy, integrity and authenticity of electronically processed data. The strength of almost all cryptographic mechanisms rests on high-quality random numbers as they are used for the generation of cryptographic sensitive parameters such as key material. Thus, the random number generation procedure must be analyzed for its suitability for cryptographic use cases.

The evaluation of the suitability and quality of cryptographic mechanisms is tasked to the Federal Office for Information Security (BSI – Bundesamt für Sicherheit in der Informationstechnik) in Germany. The BSI therefore initiated this study about the generation and collection of entropy in virtual machines and virtual environments. Virtual machines are increasingly used especially in Cloud-based solutions, covering sensitive areas in enterprises as well as in government. Good random numbers require one or more noise sources supplying entropy which implies that these noise sources are a vital requirement for the security of electronically processed data.

Operating systems use various noise sources which may exhibit properties and behaviors, which may deviate significantly when used on a bare metal system or within a virtualized environment. This study analyzes the impact of virtual environments on the presence of entropy for noise sources. The goal of this study is to identify measures for using noise sources in virtual environments in such a way that they collect sufficient entropy.

Besides conducting an analysis of the general impact of virtual environments on noise sources, this study discusses the Linux random number generator of /dev/random and /dev/urandom which includes several noise sources. Also, this study evaluates possibilities of receiving entropy from the virtual machine monitor (VMM) as well as noise sources which collect entropy independently from a virtual environment. Again, the goal is to obtain sufficient entropy in virtual environments.

This report was prepared by atsec information security GmbH under contract of the German BSI with the BSI project number of 213.

1.2 Structure of Document

The document is segmented into several parts:

- Chapter 2 explains the scope of the study, including a definition assumptions and constraints applied to the virtual environment. Furthermore, the terminology used in this study is defined.
- Chapter 3 discusses the architecture of noise sources including a discussion of the origin of the entropy collected by these noise sources.
- Chapter 4 analyzes the impact of virtual environments on the noise sources. Once the impact is identified, measures are discussed on how the impact of virtual environments on noise sources can be reduced or even eliminated. This analysis is the basis for the check list provided in Appendix A.
- Using the result of the analysis in chapter 4, chapter 5 discusses the behavior of the Linux random number generator of /dev/random and /dev/urandom in virtual environments. This discussion results in a list of measures that can be taken during the configuration and maintenance of the virtual environment to ensure that sufficient entropy is available. The discussion is supplemented with quantitative measurements of the suggested configurations and modifications to the virtual environments.

- Chapter 6 presents and analyzes a noise source that is unaffected by virtual environments. In addition, ways to provide entropy to guest operating systems in virtual environments by the virtual machine monitor are discussed.

2 Scope of Study

2.1 Basic Assumptions and Constraints

In this study we assume that the virtual machine monitor (VMM) providing the virtual environment is not malicious or has been subverted. A malicious VMM could easily manipulate the guest but also manipulate the behavior of the noise source used to generate the entropy. In many cases access to the noise source may be intercepted by the VMM which allows the VMM to control the values passed to the virtual environment (which assumes they come directly from the noise source). Also, when timing differences are used as an entropy source one has to keep in mind that in most systems that provide support for virtualization a guest's access to the timers can be intercepted by the VMM. This allows the VMM to manipulate the timer values passed to the virtual environment.

Even when the underlying platform provides interfaces (e. g. an instruction) to directly access random numbers generated by the platform, care has be to taken. For example on Intel x86 processors that support the RDRAND and RDSEED instructions a malicious VMM may prohibit the direct use of those instructions in one of the three following ways:

1. Both RDRAND and RDSEED are instructions the VMM can mark as desireable instructions to intercept. Although those instructions are not privileged and also do not allow a virtual environment to detect that it is not directly executing on a physical platform, Intel decided to make those instructions interceptable.
2. A VMM may emulate those instructions on hardware platforms that do not provide them and signal the existence of those instructions to the virtual environment.
3. A VMM may use binary rewrite techniques to intercept the execution of those instructions (or any other) for a virtual environment.

Therefore we have to make the assumptions that the VMM does not attempt to deliberately interfere with the actions taken by the virtual environment to generate random numbers and that the VMM itself is sufficiently protected against any attack that subverts the VMM. Similar we have to make the assumption that the underlying hardware/firmware platform of the VMM itself does not maliciously attempt to interfere with the virtual environment when it generates random numbers.

Even with those assumptions there are significant differences of a virtual environment to a non-virtual environment. The most critical differences for our task are:

- Access to physical resources may be virtualized, and therefore the results obtained may differ significantly from the results that would have been obtained from the real physical resource. The timing of access may especially differ significantly, which has a severe impact if timing characteristics are used for entropy generation.
- Operations that the virtual environment assumes to be atomic may only be atomic from the view of the virtual environment but not be atomic in the real system.
- Assumptions on the behavior (especially timing behavior) of operations that may hold in the real environment may not hold in the virtual environment.
- Other guest systems may be able to observe behavior the virtual environment assumes to be not observable (as they would be in a real environment).

An additional assumption underlying this analysis is that the VMM is not malicious – this assumption is orthogonal to the aforementioned assumption that the VMM does not try to deliberately interfere with the virtual environment. The hypervisor part of the VMM operates with the highest software privileges in the system and has therefore full access to the runtime memory of all guest operating systems. This would allow a hypervisor to perform any actions unrestricted, including the subversion of the guest. Also, the supporting functionality of the VMM, such as the emulation logic for devices must be assumed trustworthy. All functionality of the VMM at least have the ability to interfere with the integrity of the guest. For example, functions that are restricted to supervisor state on bare metal systems must also be accessible to software executing in supervisor state when emulating such hardware.

The VMMs discussed in this section operate with hardware support for virtualization. This implies that the VMM does not need to catch general purpose CPU instructions issued by the guest software covering standard processing.

Furthermore, the VMM is assumed to provide an effective guest isolation by enforcing a proper separation of a guest's resource from other guests. This is supported by a proper separation enforcement in the hardware, if the hardware is required for such task (e.g. SR-IOV).

Although the VMM is assumed to be trustworthy, the guests do not need to trust each other. It is in line with this study if the VMM executes trustworthy and malicious guests in parallel.

2.2 Terminology

The term Virtual Machine Monitor covers the collective software components that are needed to provide an operational environment for guest operating systems. The VMM may consist of one or more software components implementing the following aspects:

- Hypervisor: The hypervisor is the software component executing with the highest software privilege in the system. It controls the execution of virtual machines used for guest operating systems as well as for VMM use.
- Virtual “motherboard”: The virtual motherboard is the software component that implements the emulation hardware as well as the backend part of paravirtualized devices. The virtual motherboard may execute as part of the hypervisor or as a separate software entity in a virtual machine controlled and isolated by the hypervisor.
- Administrative components: The administration of the VMM is made possible with appropriate administrative interfaces and associated handling functionality. Such administrative functionality again may be implemented as part of the hypervisor or in software components executing in a virtual machine controlled and isolated by the hypervisor.

The terms virtual machine guest, guest, guest operating system or virtual machine refer to the same execution environment controlled by the hypervisor. The hypervisor separates and isolates the execution environments of different virtual machines.

3 Architecture of Noise Sources

The analysis of the impact of a virtual machine monitor (VMM) on a noise source first requires a sound architectural model of noise sources.

Using such an architectural model, the components of a noise source affected by a VMM operation can be identified and then further analyzed. All components of a noise source that are not affected by the VMM operation can be left out of scope for a more detailed analysis, which in turn reduces also the effort to assess the impact of a VMM operation on noise sources.

Furthermore, having a sound architectural model and an understanding of how a VMM affects this model supports the reader in applying the results of the analysis to noise source implementations that are not yet covered by this study.

This chapter first develops the architectural model of a noise source and then applies this model to different real-life implementations of noise sources to verify that the model is sound.

3.1 General Architecture of Noise Sources

Noise sources can be found in many different forms, including:

- Physical noise sources designed for the sole purpose of providing entropy bits. Such noise sources can be found on physical devices like smart cards, special circuitry, hardware security modules (HSMs), etc.
- Noise sources that observe the behavior of events of regular hardware. These would include observing timing behavior of human interface devices (e.g. mouse movements or typing on a keyboard), block devices (e.g. spinning hard disks) or interrupts.
- Noise sources utilizing capabilities of the CPU, including timer-based noise sources, CPU instructions like RDRAND on Intel processors, etc.

Irrespective of the source of the noise, figure 1 illustrates the concept which applies to all noise sources. This illustration has close relationships to [SP800-90B] chapter 4. In addition, this figure also relates to the description of a noise source given in [AIS2031] with the difference that the health tests are not as pronounced in figure 1. As health testing is not the prime focus of this study, it is covered to a small degree only. With the description throughout this document it will become clear that the health tests will not be interfered with by the VMM operation.

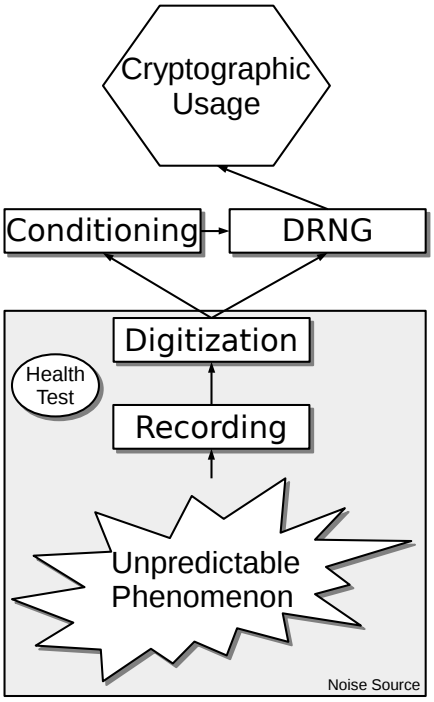

Figure 1: Architecture of Noise Source

Figure 1 shows the entire logic flow for generating random numbers. The origin of any random number is the noise source marked as a gray field in figure 1. The output of a noise source is fed into a deterministic random number generator (DRNG) which generates the output for cryptographic use cases. In some systems, a conditioner is applied to the output of the noise source where the output of the conditioner is then used as input to a deterministic random number generator.

It is possible, and even often seen in real-life environments that multiple deterministic random number generators are chained. Such a chain of deterministic random number generators is fed by the noise source or conditioned noise source data.

The gray box in figure 1 depicts the general concept of a noise source. Chapters 3 and 4 of this study revolve entirely around the discussion of that gray box. The conditioner, the deterministic random number generator(s), or applications consuming the random numbers are out of scope in these chapters. Conditioners or deterministic random number generator do not add any entropy as they only shuffle the existing data. They are used to enhance the entropy per bit by applying a compression.

The architecture of a noise source as shown in figure 1 contains the following major parts

- A phenomenon is measured that exhibits an unpredictable or partially unpredictable pattern to the observer. It is key to understand that the unpredictability always relates to the observer and may vary depending on the type and skills of the observer – i.e. the unpredictability and therefore the resulting entropy is **relative** to the observer. For a lot of noise sources, the observed phenomenon may be completely deterministic if all parameters are known that affect the phenomenon. Such noise sources depend on the fact that one or more of these parameters cannot be predicted by an observer with the required accuracy. This unpredictable phenomenon can either be:
  - a physical phenomenon that is unpredictable in nature, such as thermal noise or shot noise, metastability in bi-stable circuits, or even radioactive decay[1];

1 Albeit radioactive decay is a good example of an unpredictable physical phenomenon with a proven physical theory behind it, the author is well aware that radioactive decay is highly impractical in normal computing environments. Therefore, it shall serve as an example for discussion only.

    - an unpredictable phenomenon triggered by the interaction between the computer hardware and its environment. For example, human interaction, or the receipt of interrupts triggered from external devices recording some externally triggered events would fall into this category.

- A recording logic is required that is capable of measuring the events generated by the unpredictable phenomenon. The recording logic does not necessarily need to store the measured data though.
- Using the recorded events, the digitization logic turns the recorded data into a digital data stream which is then provided to either a post-processing conditioner or directly into a deterministic random number generator. The use of a deterministic random number generator at this stage is not intended to stretch the entropy over a large amount of output, but its purpose is the same as the conditioner discussed in the following. Commonly only one of the mentioned mechanisms is used to post-process the data from a noise source. Albeit it may be possible to use the output of the digitization logic directly as input into cryptographic use cases, such course of action is commonly disregarded. The conditioner as well as the deterministic random number generator perform a whitening[2] of the noise source data that does not reduce the collected entropy[3] and yet transforms the data into white noise. As already mentioned, the key value of those components is to increase the entropy per bit by performing a compression by using XOR. In addition, the conditioner may be used to hide skews in the raw data by applying a Von-Neumann unbias operation or using a linear feedback shift register (LFSR).
- For noise sources, it is commonly suggested – and it is required for noise sources to be accepted by BSI – to employ some form of health check to guard against total breakdown of the event recording or the operation of the measured phenomenon. Naturally, the health check cannot detect changes in the entropy rate delivered by the recording logic, for example, due to aging or negative influences from the environment. However, small statistical tests tailored to the entropy source can detect non-tolerable defects in the stochastic behavior of the noise source in a reasonable time window. An example of such a test is the Chi-squared test.

To make the theoretical discussion more understandable, the following sections illustrate how various real-life noise sources relate to the aforementioned theoretical construct.

3.2 Common Noise Source Designs

This section enumerates various commonly used noise source designs. This section is intended to provide the reader with a description of noise sources without considering a specific implementation.

The noise sources are described with the concepts which disregard details of a particular implementation. Only the components which are important to the noise source and the processing of that noise to obtain entropy are documented.

3.2.1 Hardware Noise Source: Ring Oscillator

In the realm of hardware, one noise source is found very often: a ring oscillator. A mathematical analysis of the noise derived from a ring oscillator is provided in a number of documents, for example [JITTERROSC], [SECROSC], or [ENTROPYSOURCES].

An example of well-known hardware systems which uses that ring oscillators are all Apple iPhone and iPad devices that employ a Secure Enclave – i.e. all devices with an Apple A7 or

2 A conditioner may provide cryptographic or non-cryptographic whitening, whereas a deterministic random number generator is a cryptographic cipher. Thus a deterministic random number generator satisfies more requirements than a cryptographic conditioner. However, both are intended to generate white noise, increase the entropy per bit and to reduce the skew in the raw data.

3 When using cryptographic mechanisms such as hashing, their nature of behaving like a randomly constructed function implies that a small degree of entropy is lost. However, this is considered immaterial to this discussion here.

later CPU. [IOSSEC] explains that multiple ring oscillators are used to feed an [SP800-90A] deterministic random bit generator (DRBG), specifically a CTR DRBG.

Noise sources based on ring oscillators are a common design for many popular hardware devices.

An interesting exception from the commonality of ring oscillators are Intel x86 CPUs as discussed in chapter 3 of [INTELDRNG]. As discussed in [INTELES][4] the noise source is a "circuit that pushes a latch into metastability". This noise source is further analyzed in section 3.3.2.

A ring oscillator is formed by a set of (2n+1) inverters that are connected linearly into a ring. Figure 2 illustrates the basic construct of a ring oscillator. Real-life implementations will be more complex[5], but the common parts relevant to this conceptual discussion are all present.

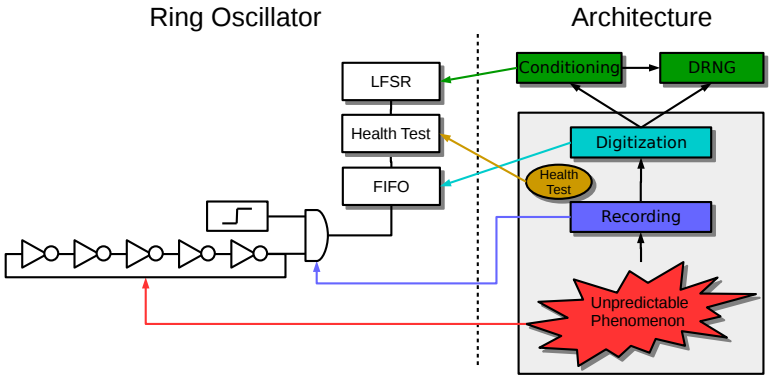

Figure 2: Mapping Ring Oscillator to Noise Source Architecture

Figure 2 illustrates the ring oscillator on the left hand side with the chain of 5 inverters connected into a ring. The number 5 shall serve only as an illustration; it could be any odd number.

The ring of inverters is connected to one of the inputs of an AND gate – in real-life implementations a delay-flip-flop is commonly used here which may also store some state in addition to the AND logic. The other input of the AND gate is connected to a stable sampling frequency, which commonly is significantly lower than the frequency of the ring oscillator[6]. With this sampling frequency, the current state of the ring oscillator is measured in certain intervals. The AND gate returns either a zero – the ring oscillator at the point where the AND gate is connected has a low voltage –, or one – the ring oscillator has high voltage. The output of the AND gate therefore obtains the state of the ring oscillator. The obtaining of the state of the ring oscillator can be mapped to the recording logic in the noise source architecture. In real-life scenarios, usually multiple ring oscillators are used concurrently and measures are taken to avoid synchronization between them. Their recording outputs are connected typically with an XOR cascade before being fed into the next circuitry. Multiple ring oscillators and the associated XOR cascade, however, are immaterial to the topic of discussing the impact of virtual environments on the noise source of ring oscillators. Therefore, multiple ring oscillators are not considered here.

---

4 Albeit this document may look like a questionable source, it is authored by DJ Johnston, a member of the Intel RDRAND architecture team.

5 An example of the Verilog code of a ring oscillator can be found at https://github.com/secworks/rosc_entropy.git.

6 The sampling frequency should be significantly lower than the ring oscillator frequency. Oversampling, i.e. sampling very fast, will result in a low entropy per sampled bit. The correct choice for the sampling rate depends on the random jitter of the ring oscillator. It is important that at the time of sampling the relative jitter of the sampling clock is equally distributed over the period of the ring oscillator. The rule of thumb is that the less random jitter is present in the ring oscillator the lower the sampling frequency must be.

The post processing now may feed the output of the AND gate into a first-in, first-out (FIFO) circuitry for buffering for the next step – again, instead of a FIFO, the aforementioned delay flip-flop may combine the depicted AND gate and FIFO. With the FIFO, the data recorded from the ring oscillator now is available in a digital form for any post processing circuitry. The data in the FIFO may then be subjected to health tests followed by a conditioning operation like a linear feedback shift register (LFSR), as depicted in figure 2, or it may even implement a conditioning operation based on cryptographic primitives like a SHA-256. It is very important to determine the entropy contained in the FIFO data and design the post-processing accordingly. The output of that LFSR now can be used to feed a deterministic random number generator.

With the odd numbers of inverters, powering this circuitry generates an oscillation between two voltage levels. The oscillation starts spontaneously above a threshold voltage. That threshold is dependent on the technology used for the inverters and other factors such as the layout of the circuitry on silicon.

The uncertainty that is the basis for collecting entropy is the timing fluctuations in rising and falling edges of the voltage levels in the ring oscillator. Figure 3 characterizes the phenomenon by showing the voltage level with a solid line. At the rising and falling edges, however, a slight uncertainty regarding the exact time of the rise and fall exists, shown with the dotted lines. Larger time deltas between the actual and the expected rise and fall times are less probable than smaller time deltas. In fact, the distribution of the jitter is often assumed to approximately follow a Gaussian distribution.

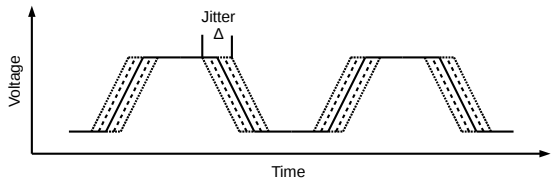

Figure 3: Timing Jitter of Ring Oscillator

With every inverter introducing some jitter (even with some inverters canceling out the jitter of another), every round of the ring adds up the per-round deviation from the ideal oscillation curve. As soon as the deviation is bigger than the switching time of one inverter (λ/2 of the full oscillation), the measurement of the oscillator turns into random value. Once the ring oscillator is observed and the random value is obtained, the oscillator needs to run again until the jitter is big enough to provide a random result for the next observation as otherwise skews or dependencies are evident – this is enforced by choosing an appropriate sampling frequency. If enough inverters are in the oscillator ring and the sampling frequency is low enough, this requirement can be easily met. Further discussions and mathematical analyses of ring oscillators are given in many other documents, including [JITTERROSC], [SECROSC] as well as [ENTROPYSOURCES].

In addition to the illustration of a ring oscillator, figure 2 contains the generic noise source architecture picture discussed in section 3.1. The different architectural components of a noise source specified with figure 1 are marked with different colors in figure 2. Arrows using the respective color map both illustrations as follows:

- The unpredictable phenomenon is the previously discussed timing behavior of the raising and falling edges of the voltage of the ring oscillator. Therefore, the ring oscillator is the technical representation of that phenomenon.
- The sampling circuitry which is depicted with the AND gate supported by the sampling frequency, records the state of the ring oscillator with the intention to collect the timing uncertainty. Therefore, it implements the recording component.
- The output of the AND gate and the FIFO can be considered as the digitization logic as it provides a stream of bits to the later processing logic.
- The health testing circuitry is clearly identifiable.

The LFSR component of the ring oscillator noise source is considered to act as a conditioner for the data.

One important key aspect for the subsequent discussion and the topic of this document is the identification that all aspects of the noise source architecture for ring oscillators are implemented in hardware only. There are no software components involved.

3.2.2 Software Noise Source: Time Stamping of Events

Similar to ring oscillators being a standard noise source in hardware, the time stamping of events is a common approach in collecting entropy for software. The timing of the observed events is believed to exhibit an unpredictable behavior, at least to a certain degree where the least significant bits of a high-resolution time stamp of a given event cannot be predicted. The more least significant bits are demonstrated to be unpredictable, the higher the entropy is for one event observation.

A very common event type are interrupts. Such interrupts may be recorded globally for all handled interrupts, or for dedicated interrupts only. This noise source is used in the following well-known software systems:

- The Linux kernel /dev/random and /dev/urandom devices are fed by interrupts, human interface device (HID) events as well as block device events. Further discussions of this system is given in section 3.3.1.
- The XNU kernel that drives Apple Mac OSX as well as Apple iOS records the time stamp of interrupts using a high-resolution time stamping mechanism. For Mac OSX on x86 platforms, the RDTSC CPU instruction is used to obtain a high-resolution time stamp. XNU maintains an entropy pool that is segmented into 32-bit entries. The low 32 bits of a time stamp of an interrupt is XORed with the 32-bit entry of the entropy pool. After modifying one 32-bit entry of the entropy pool, the logic selects the next 32-bit entry to porcess the next interrupt time stamp. With this algorithm, each of the 32-bit entries are modified in a serial fashion.
- OpenBSD uses HID events, block device events, and receipt of network packets as a source for entropy.

Irrespective of the monitored type of the event, a common approach in implementing such a noise source is evident. That common approach is depicted in figure 4.

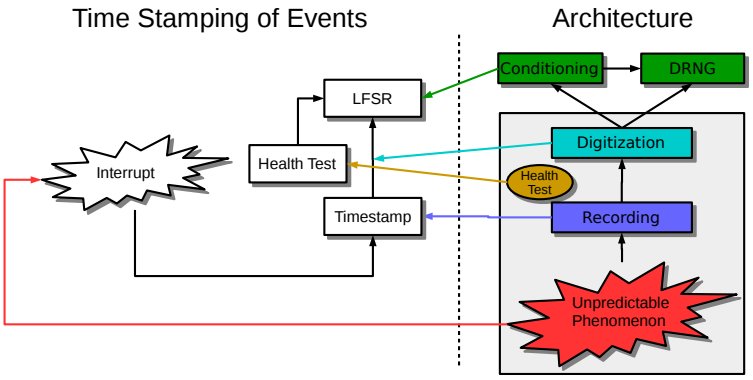

Figure 4: Mapping of Time Stamping of Events to Noise Source Architecture

The basic idea for using a hardware event as a source for entropy is based on recording a high-resolution time stamp every time that event is triggered. The time stamp is then used as a basis to obtain entropy. Figure 4 illustrates this approach with an event triggered by interrupts. When an interrupt is triggered, the interrupt handler records the time stamp of the event. That time stamp is then mixed into an entropy pool maintained with an LFSR or a cryptographic primitive like a SHA-256 hash. To extract entropy, a cryptographic operation like a SHA-256 is calculated for the entropy pool. Health tests may be implemented and commonly act on the data in the entropy pool or on the data that is will be added to the entropy pool.

When mapping such a noise source to the generic noise source architecture discussed in section 3.1, the following equivalence is evident:

- The unpredictable phenomenon is the occurrence of the event, in this case the monitored interrupt.
- The recording of an interrupt within the noise source implementation is implemented by obtaining the high-resolution time stamp from the underlying CPU. Almost all commonly used CPUs have the ability to deliver a high-resolution time stamp. Depending on the implementation, it may be possible that only the least significant bits are processed and the most significant bits are discarded. The least significant bits of a high-resolution time stamp are the fast moving bits. Therefore, only parts of the time stamp must be considered unpredictable. For example, current CPUs return a high-resolution time stamp with 64 bits. If the CPU operates with one GHz and this frequency is used for the high-resolution time stamp, the least significant 32 bits are moved within

  $$\frac{2^{32}}{(1{,}000{,}000{,}000\,Hz)} = \frac{4{,}294{,}967{,}296}{1{,}000{,}000{,}000}\,sec \approx 4\,sec \quad .$$

  The high 32 bits of the time stamp therefore move at a rate slower than 4 seconds. An observer of the operation of the operating system should easily be able to detect and predict an interrupt event with a precision in the millisecond or even microsecond range. This implies that the high 32 bits must be assumed to be always predictable and can therefore be discarded.
- The digitization commonly is implemented by interpreting the time stamp as a byte stream that is mixed into an entropy pool. The logic to mix data into an entropy pool is implementation-dependent where often either an LFSR or a cryptographic hash function is used.
- If implemented, a health testing logic is usually applied to the data before being processed by the LFSR or the cryptographic operation to mix data into the entropy pool.

The entire noise source is commonly a mix of hardware and software components. The unpredictable phenomenon commonly is a hardware device. The recording logic is usually software, but depends on a high-resolution timer that can only be delivered by hardware. The digitization and health tests are implemented in software as they have no specific relationship to any hardware.

3.3 Particular Implementations of Noise Sources

While the preceding section covered generic noise source designs, this section takes particular implementations of noise sources and shows how they apply to noise source architectures specified above.

The selection of particular implementations is driven by the use cases applicable to the environments BSI is responsible for.

3.3.1 Linux /dev/random and /dev/urandom

The Linux Random Number Generator (LRNG) implementing the /dev/random and /dev/urandom device files provides a default seed source on all Linux systems. User space libraries providing cryptographic services as well as in-kernel services draw seed data for random number generators. Also, the data from these devices are often used directly as key material or other cryptographic use cases without processing them with a deterministic random number generator.

The implementation of the LRNG is discussed in detail in [LRNG] and will not be re-iterated here[7]. Before continuing with the following discussion, the reader should be familiar with the design and implementation of the LRNG.

7 The description of the LRNG is based on the implementation found in Linux kernel 3.6 and later. At the time of writing, the Linux kernel 4.4 was current.

Figure 5 provides a mapping of the LRNG with the theoretical discussion about the above mentioned noise sources. Using the mapping, the noise sources can be clearly identified and separated from the remainder of the LRNG processing. Such identification is essential for the subsequent discussion about how a virtual environment can affect the entropy collection of the LRNG.

With figure 5, three areas are illustrated which are separated by a dotted line:

- The upper left part contains the LRNG illustration as covered in [LRNG]. The LRNG is solely a software entity where all parts depicted in the upper left part are implemented in the Linux kernel.
- The LRNG observes and records events from various hardware devices. These hardware devices are illustrated in the lower left part of figure 5. Each of the gray boxes of the LRNG containing “add_*_randomness” map to a device type that is monitored by the LRNG. The LRNG boxes of “add_device_randomness” and “add_hwgenerator_randomness” are not further mapped and discussed, as they either do not deliver any entropy or access highly specialized hardware that is not commonly present in standard systems[8]. To keep the entire discussion concise, these two boxes are therefore disregarded.
- The right part of figure 5 contains the architecture illustration from figure 1. As the discussion is about noise sources, figure 5 does not further show the box about the cryptographic usage of data obtained from the noise source via the deterministic random number generator.

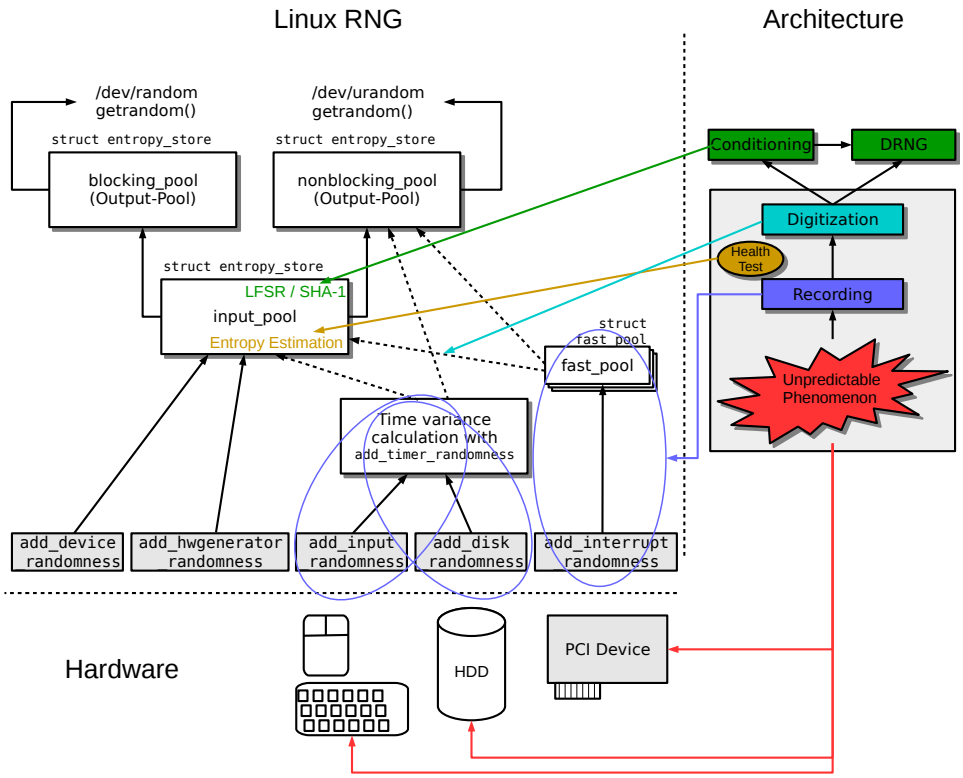

Figure 5: Mapping Linux /dev/random to Noise Source Architecture

With the given description of the LRNG, the LRNG can be considered as a software noise source that is based on time stamping of events as mentioned in section 3.2.2.

Before explaining the mapping and all the colored arrows in figure 5, the LRNG will be dissected to aid the understanding and narrow down the discussion to the vital components. As shown in figure 5, the LRNG consists of three “entropy pools”, the input_pool, the blocking_pool and the nonblocking_pool. As discussed in [LRNG], all three entropy pools are

8 See [LRNG][LRNG] chapter 2 for further discussion of these components.

processed in a completely identical fashion with the only exception of the following differences:

- The source of data fed into these pools is different between the input_pool and the two output pools of blocking_pool and nonblocking_pool. The data source for the input_pool is the output from the noise sources – a complete decomposition of the noise sources is given below. In contrast, the blocking_pool and nonblocking_pool are both fed by the output of the input_pool[9].
- Each entropy pool contains an LFSR where the input_pool has a different polynomial than the two output pools. This difference is due to a difference in size of the input_pool versus the two output pools.

Irrespective of these small differences, each pool is processed completely independently from the other pools. Also the state of each entropy pool is maintained in isolation of the respective other entropy pools. Considering the cryptographic behavior of these three pools and their independent operation, each entropy pool can be considered as an independent random number generator. Therefore, the LRNG contains three random number generators: Each entropy pool is its own deterministic random number generator with an LFSR as state transition function and a SHA-1 operation as output function.

The two output pools are seeded by input_pool and are therefore deterministic random number generators that are chained to another deterministic random number generator, the input_pool. Therefore, both output_pools have no direct interaction with the noise sources and can therefore be disregarded in any discussion of LRNG noise sources. Please note that the nonblocking_pool is directly connected to the noise sources during boot time until a threshold is reached. After reaching that threshold, the kernel severs that link irrevocably until the next boot. For that time frame, the nonblocking_pool exhibits the same characteristics as the input_pool when considering and discussing noise sources. In the following, only the input_pool is discussed together with the noise sources where the reader can apply the assessment to the nonblocking_pool at boot time.

As only the input_pool is connected with the noise sources, the mapping of the LRNG to the architecture of noise sources is limited to the input_pool and the logic handling hardware events as depicted in figure 5.

After providing clarifications around the LRNG, the explanation of how the LRNG maps to the noise source architecture presented in section 3.2 is examined in the following. The right side of figure 5 shows the theoretical noise source concept from figure 1. Figure 5 uses different colors for the different components and uses equally colored arrows that point to the respective components of the LRNG. To be precise:

- The unpredictable phenomenon identified with the red arrows in the LRNG are the events triggered by the monitored hardware components. As discussed in [LRNG] chapter 2, the following hardware events are monitored:
  - Human Interface Devices (HID): The events of pressing or releasing a key on the key board, the movement of the mouse or other pointer device, or the pressing of a mouse button is considered an unpredictable phenomenon. The unpredictability relates to the timing of the occurrence of these events and to a lesser degree to the actual event type itself, such as the actual key that is pressed or the X-Y coordinates of the movement of the mouse. The assumption regarding the unpredictability rests on the human factor: when a human types on the keyboard, variations in the speed of typing occur. Even with an experienced typist, variations in the range of milliseconds are measurable. Similarly for the movement of a mouse, the speed and therefore the timing of different events depends on the human factor whose entropy can be extracted with a high-resolution timer.
  - Block devices: The events of a read or write access to a hard disk are treated as an unpredictable phenomenon. The assumption is based on the complexity and randomness of the interaction between the read-write head and the spinning

9 Figure 5 and [LRNG] chapter 2 explains that during initialization time, the nonblocking_pool is also fed directly by the noise sources. This fact is discussed in the next paragraphs.

platters. To access a given sector on the disk, the read head needs to be re-positioned which requires time depending on the previous location of the head. It is commonly unknown where the read head was located before starting the request and where the sector is stored on the physical disk. Furthermore, for the sector to be access, the spinning disk needs to rotate until the start of the sector is under the read head. The hard disk needs time to spin the angle between the location of the start of the sector when the access request is made and the location of the read head. Both wait times can be detected with a high-resolution timer and serve as the basis for the unpredictability of the precise access time. Furthermore, potential turbulences that are created by the spinning disk may also impact the time until the access request can be satisfied.

- Devices triggering an interrupt: Interrupts are commonly generated by extension devices that are connected to the bus system of the underlying hardware. On x86 systems, the PCI or PCIe bus is the most common bus system. Other devices may use other bus systems, but the type of bus is immaterial for the assessment of the noise source. The key is that a device triggers an interrupt when it wants to notify the operating system that some event has occurred, like the submission or receipt of a network packet by a network card. When using a high-resolution timer, the exact timing when such an event occurs cannot be predicted with complete accuracy due to various reasons. One of these reasons is that the interrupt handler is subjected to the CPU execution time jitter[10]. Also, the time when such interrupts occur depends on environmental constraints that may not be completely predictable by an adversary, like varying network latency.

For the discussions in later sections, it is important to note that the unpredictable phenomenon always originates in a hardware device.

- The recording of the unpredictable phenomenon, i.e. the events and their precise timing triggered by the aforementioned hardware components, is performed by the blue-marked components of the LRNG, namely the add_*_randomness functions. The following list explains the recording operation for the different hardware components:
  - HID: As described in [LRNG], chapter 2, two data sets are recorded. The add_input_randomness function records the event type in form of the numeric number of the pressed or released key, or the coordinates of the mouse movement. In addition, the add_timer_randomness function records the time stamp of these events using a high-resolution timer.
  - Block devices: The function add_disk_randomness records the block device number of the accessed disk. Just like for HID, the add_timer_randomness obtains the high-resolution time stamp of the event time of the disk access.
  - Devices triggering an interrupt: The function of add_interrupt_randomness records the value of the CPU instruction pointer, the high-resolution time stamp and the interrupt number for each interrupt. This information is used to update a per-CPU data structure called fast_pool. This fast_pool is a small-scale LFSR that mixes the recorded interrupt data with previously recorded interrupt data. The maintenance of the fast_pool is still considered to belong to the recording of an event, because it is used to break any possible correlation between the interrupt recording and the events recorded for block devices and HID – note that for block device events and HID events, interrupts are recorded, too. The fast_pool modifies the data collected from add_interrupt_randomness and must be therefore treated as part of the recording logic of the unpredictable phenomenon.

  The recording logic is pure software that uses the interfaces to the hardware for obtaining its information. Furthermore, all collected data originates in hardware devices, including the high-resolution time stamp. Such high-resolution time stamps are commonly provided by the underlying CPU, usually with a CPU instruction[11].

10 The CPU execution time jitter will be discussed in much greater detail in sections 3.3.3 and 6.1 where a random number generator using that phenomenon is introduced.

11 For x86, this is the RDTSC (Read Time Stamp Counter) instruction. STCK (Store Clock) is provided on IBM zArchitecture CPUs. On PowerPC based systems, the MFTB (Move From

- The digitization of the data obtained with the recording components is implemented by injecting the recorded data into the input_pool. Digitization is performed in a very simple fashion in the LRNG as noted in the following:
  - For HID and block devices, the recorded event type and time stamp are stored in a data structure which then is simply treated as a byte-stream that is injected into the input_pool. The interpretation of the recorded data as a byte stream is the digitization of that data.
  - For interrupts, regular snapshots of the fast_pool are injected into the input_pool. Similarly to the HID and block devices, the contents of the fast_pool is treated as a byte-stream when it is mixed into the input_pool. Again, the interpretation of the fast_pool as a byte-stream implements the digitization aspect.

  The digitization logic is implemented solely in software and has no dependency on the hardware other than that the CPU must implement standard CPU instructions to operate on memory.
- When considering the health test, the LRNG implements one mechanism that serves as a common health test for all data that will be mixed into the input_pool: the entropy estimation heuristic. The entropy estimation calculates the first, second and third derivation[12] of the time stamp of each event. The minimum of these derivation values is used to increase the entropy estimation of the input pool. Now, when this entropy estimation is zero, the input_pool does not provide any data to the callers of the nonblocking_pool or the blocking_pool. Therefore, if an event is received where one of these derivations is zero and hence indicate a pattern that should be considered to have very little or no entropy, the event data is mixed into the input_pool without allowing the input_pool being read by that amount of data. In effect, this implies that the mixed in event data is treated as poor data where the noise source failed to deliver entropy.

  The health test, just like the digitization logic, is implemented in software without depending on particular hardware support. Again, only standard CPU instructions to operate on memory are used.

In addition to the noise source components, figure 5 also identifies the conditioning and deterministic random number generator logic. The management of the LRNG entropy pools is implemented twofold:

- An LFSR takes the digitized data from the noise sources and mixes them into the state of the input_pool. That LFSR therefore acts as a state transition function φ according to the terminology of [AIS2031].
- The calculation of a SHA-1 hash of the entire state of the input_pool serves as the output function ψ according to the terminology of [AIS2031].

Both functions, the LFSR and the SHA-1 calculation together can be interpreted as parts of a deterministic random number generator[13] which can be mapped to the architecture illustration in figure 1. The deterministic random number operation is implemented with pure software logic and has no dependency on hardware, again with the exception of standard CPU instructions. Albeit the Linux kernel supports hardware implementations of SHA-1, the LRNG always uses a software implementation.

On the other hand, picking up the unpredictable events from the hardware devices, the LRNG requires the following hardware accesses:

- HID: The Linux kernel drivers for keyboards and mice must be able to interact with the devices. These HID devices are accessed with different hardware buses like USB, PS/2 or even serial lines. The entire stack of kernel drivers that accesses the bus(es)

Time Base) instruction is available.

12 The first derivation is the time delta between the last event and the current event. The second derivation is the delta of these time deltas. Finally, the third derivation is the delta of the delta of these time deltas.

13 This document does not try to provide an argument or mathematical rationale whether this random number generator complies with AIS 20/31. It simply provides a mapping only.

commonly with memory mapped input/output (MMIO) and interprets the information received from the HID requires unaltered access to these devices. When such devices are absent, like with headless server systems, the LRNG will not pick up any entropic data from HID.

- Block devices: The Linux kernel implements block device drivers that require access to hardware similar to HID. The main difference is that the bus to be accessed is IDE or SATA which usually use a PCI device. Again, commonly MMIO operations together with direct memory access (DMA) operations are performed. Thus, the entire driver stack for all these components is required to interact with block devices and support the noise source operation.
- Interrupts: Interrupts are triggered by the devices connected to the lowest level of hardware bus(es). Most commonly, interrupts are associated with PCI devices. The top half of the interrupt handler (i.e. the code that is immediately accessed when an interrupt is received and which tries to offload the interrupt information and to acknowledge the interrupt) also triggers the noise source operation.

The unpredictability is measured with a high-resolution time stamp which is commonly provided with a CPU instruction. Any interference with that time stamp has a direct impact on the noise source operation as well.

3.3.2 Intel RDRAND and RDSEED

The CPU instructions of RDRAND and RDSEED in the current Intel x86 CPUs are advertised as noise sources delivering entropy and random numbers at a very high rate. Both instructions are in use more and more often which warrants a coverage at this stage to support a further analysis in the subsequent sections.

Albeit current AMD x86 CPUs also provide an RDRAND and RDSEED instruction with the same mnemonic as the Intel CPU, no public design information was found at the time of writing. Therefore, this section disregards the AMD implementation.

The general internal structure of the implementation of RDSEED and RDRAND is provided in chapter 3 of [INTELDRNG]. For gaining an understanding of the operation and design of the noise source, however, this document contains insufficient data. The information gaps can be filled with the details provided in [INTELES]. When combining the information from both documents, the following architecture is evident – the left hand side of figure 6 is copied out of [INTELES] and depicts this architecture:

1. An entropy source (marked as ES in figure 6) collects “a stream of entropic data” as described in [INTELDRNG], chapter 3. The entropy is implemented as a “self timed, digital circuit that pushes a latch into metastability at ~3GHz. The timing of each cycle is determined by the resolution time of the latch. Thermal noise determines the resolution state of each metastable event and this is the fundamental source of entropy in the circuit” (see [INTELES]). According to [INTELES], the entropy source consists of the metastable latch, the extraction logic which uses a clock to measure the timing of events when the latch flips, and an XOR accumulator that is fed with the extracted data.
2. The output of the entropy source is processed by an online health test marked as OHT in figure 6. This health test conducts testing with a “[d]ynamic per sample of the ES outputs against expected statistics”. Furthermore, the health test “[c]ontrols quality and volume of raw entropy going into Conditioning” as documented in [INTELES].
3. After the data is processed by the health test, it is fed into the conditioner. This conditioner is an AES CBC MAC that implements a compression rate of 2, i.e. when 512 bits are obtained from the entropy source, 256 bits are then forwarded to the next stage. As outlined in [INTELDRNG], chapter 3, the output of the conditioner is either forwarded to a DRBG described in the next item, or it is forwarded for output via the RDSEED instruction. [INTELDRNG] mentions that one data set of the conditioner is never forwarded to both destinations.
4. The final stage is a CTR DRBG with AES core which receives the data from the AES CBC MAC conditioning logic.

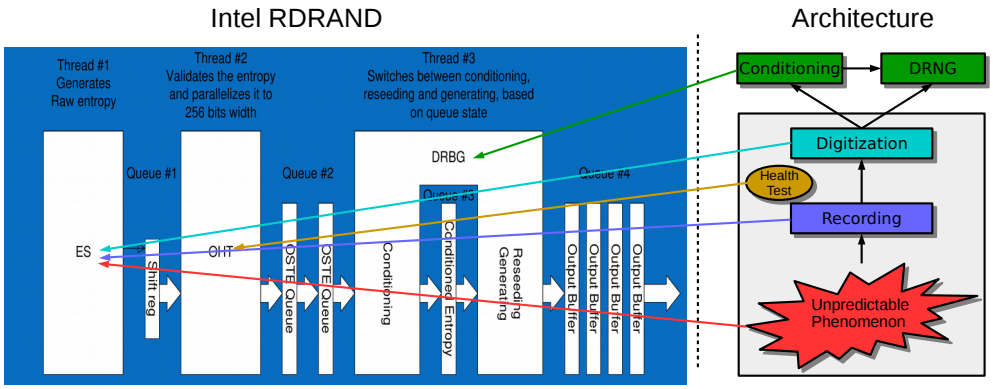

Figure 6: Mapping of RDRAND and Noise Source Architecture

In order to map the RDRAND implementation to the noise source architecture, figure 6 contains the architecture illustration discussed in section 3.1. The following mapping can be established:

- The unpredictable phenomenon is the exact timing of the switching of the latch during its metastable phase.
- The extraction logic mentioned above serves as the recording component. In figure 6, the extraction logic is integrated into the box for the entropy source. [INTELES] contains an illustration that dissects the entropy source showing the extraction logic.
- The digitization is implemented with the XOR accumulator that is also located within the entropy source in figure 6. Again, the digitization logic is depicted in the illustration provided with [INTELES].
- The health test is implemented in the RDRAND logic with a component of the same name as depicted in figure 6.

The AES CBC MAC conditioner as well as the SP800-90A DRBG find their equivalents in the conditioning and deterministic random number component of the noise source architecture.

The description shows that the entire noise source is implemented in the silicon of the Intel CPU. It provides an interface for software to pick up entropic or random data via the CPU instructions of RDRAND and RDSEED.

3.3.3 CPU Execution Time Jitter Random Number Generator

Besides the LRNG discussed in section 3.3.1, the Linux kernel contains a second noise source that is used to seed the in-kernel SP800-90A DRBG: the Jitter RNG. The Jitter RNG is discussed here and mapped to the concept of the noise source architecture since it will be further analyzed and vetted in section 6.1 assessing whether this Jitter RNG is affected by a virtual environment at all.

This discussion and the entire analysis is not intended to assess the entropy provided by the Jitter RNG. The analysis assumes that this Jitter RNG delivers some amount of entropy and analyzes how much this entropy changes when this Jitter RNG is operated in a virtual environment, if at all. A brief architectural description of the Jitter RNG is given in the following[14].

The Jitter RNG is documented with [JENT]. This document provides an architectural description in chapter 3. In particular chapter 3.1 [JENT] illustrates the noise source which is based on the following phenomena that are considered unpredictable:

- The timing of memory accesses using a high-resolution timer shows slight variations. These variations are based on the following: the clock speed of the CPU commonly is

14 Note, the Jitter RNG is developed and maintained by one of the authors of this study. As the current study does not assess the quality of the Jitter RNG to produce entropy, any potentially existing bias of the authors of this study towards this noise source is considered to not affect the analysis or results of this study.

faster than the clock speed of the caches of even the main memory bus. When the CPU wants to fetch or store data in these memory components, the hardware has to synchronize the clock pulse of the CPU with the pulse of the accessed memory. This synchronization potentially adds wait states to the CPU memory fetching operation until the pulses are synchronized for the access request. The wait states naturally differ in length depending on the state of the clock in the CPU versus the clock for the memory. And these varying wait states affect the time for a memory access. For more details, see [JENT] section 6.2.1.

- The timing of the execution of a fixed set of instructions shows slight variations. The analysis of [JENT] section 6.1 shows that the variations depend on the CPU internal state of the branch prediction logic, frequency scaling state, the translation look-aside buffer (TLB) state, and the CPU instruction pipeline. Most likely other CPU-internal state aspects affect the variations in the execution time of instructions as well[15].

Figure 7 is derived from section 3.1 [JENT] illustrating the core of the noise source in the left part. The right part of figure 7 again specifies the general architectural concept of noise sources as presented in section 3.1.

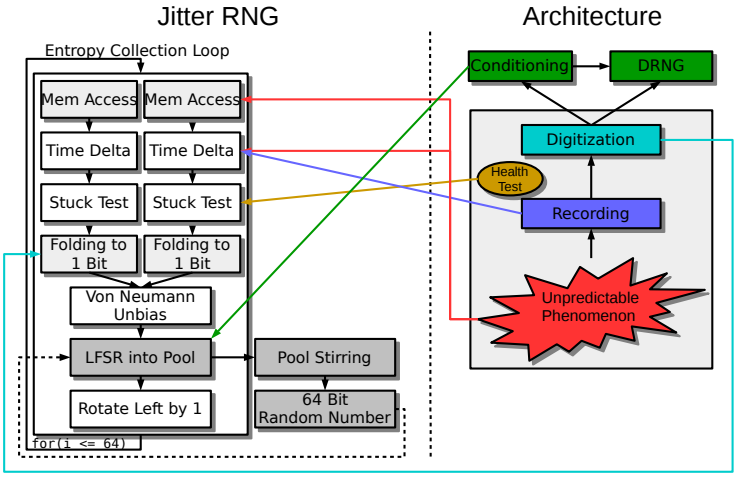

Figure 7: Mapping Jitter RNG to Noise Source Architecture

The Jitter RNG maintains an entropy pool that is 64 bit in size. The concept of the Jitter RNG rests on an entropy collection loop that in its first step performs the memory access. In a second step, the time stamp is obtained from the CPU and the delta of the time stamp of the previous round is calculated. The idea is that the timing variations in the memory access as well as the processing of the Jitter RNG logic in general is picked up and compared to the previous round of the entropy collection loop.

The obtained time delta is now subjected to a health test, the “stuck test”. The result of the health test whether the measurement is considered healthy is enforced in a later step discussed below. The health test at this step only determines if a bit is considered healthy but does not alter the operation depending on that result. The health test considers the nature of how the unpredictability is recorded: variations of the delta of time stamps, i.e. the first derivation of the absolute time stamps, convey the unpredictability. Therefore, the health test calculates the second and third derivation of the time stamps, i.e. the delta of time deltas and the delta of deltas of time deltas. All three derivations must be non-zero at the same time to accept a time delta measurement as appropriate.

15 For multi-core CPUs, even the instruction to read the high-resolution time stamp may require synchronization of all cores to ensure that the time stamp is a monotonically increasing number. The CPU must ensure that when the time stamp is read on core 0 immediately followed by a read on core 1, the time stamp on core 1 will be larger than on core 0 barring a wrap-around of the integer.

The time delta is now collapsed into one bit by taking the XOR sum of the bits in the binary representation of the integer value of the time delta with each other. The analysis of [JENT] shows that the bit derived from one time delta is independent from the next bit. Therefore, the Jitter RNG applies a “Von-Neumann” unbiasing operation to obtain a final bit that is now mixed into the entropy pool. The mix-in operation uses an LFSR to update the right-most bit in the entropy pool – i.e. the polynomial of the LFSR defines the bits of the entropy pool that are XORed together with the received bit to form the new bit in the entropy pool. After the LFSR operation, the entropy pool is rotated left by one bit to allow mixing in the next bit from the next entropy collection loop iteration into the next bit of the entropy pool. Irrespective of the result of the health test, the time delta is always XORed with the right-most bit of the entropy pool. However, if the health test indicates a problematic time delta value, the LFSR operation is not performed and the bit location is not changed. This implies that the next bit from the next entropy collection loop iteration is mixed into the same bit position of the entropy pool. The omission of the LFSR operation in case of a failing health test is self evident now: if one bit is deemed not to pass the health test and the second bit passes, the application of the LFSR for the first bit would be canceled out by the LFSR operation for the second bit due to it being an XOR operation using constants. Therefore, the LFSR operation is only applied for bits that pass the health test and for which the entropy pool is rotated. The entropy collection loop is performed as often as needed to modify all 64 bits of the entropy pool. In the best case, i.e. without any time deltas marked as problematic by the health test, 64 iterations of the entropy collection loop are performed.

As it is evident from the description, this noise source is completely implemented in software. The only dependency on the hardware besides the general processing of data is the presence and use of a high-resolution timer.

To map the noise source architecture concept to the Jitter RNG, the following considerations should be noted:

- The unpredictable phenomenon upon which the Jitter RNG rests is the CPU internal state as well as the wait states the CPU must apply when performing memory accesses. The unpredictability is manifested in tiny variations of the execution time of CPU instructions.
- The recording of the unpredictability is performed by obtaining a high-resolution time stamp before as well as after the execution of the memory accesses and other CPU operations. The resulting time delta is the recorded information that is processed further.
- The digitization of the recorded time delta information is the collapsing of the data into one bit.
- The stuck test described above in figure 7 acts as a health test.

The LFSR to mix the obtained one bit into the entropy pool is already a conditioning logic that supports the noise source operation. The same applies to the Von-Neumann unbias operation.

3.3.4 Apple Mac OS Noise Source

The XNU kernel implements the noise source used for Mac OS and iOS. This noise source provides entropy to in-kernel users as well as to user space. Both operating systems are based on the same kernel, the Apple XNU kernel. Albeit natural differences are present for both kernels due to the use of different hardware architectures, the noise source architecture is common for both. The following description therefore explains the common architecture and highlights differences, if applicable.

Although no reference to a design description can be given at this point, the XNU source code is available at Apple's open source web site. The noise source maintenance and how it is embedded into the remainder of the XNU kernel can be derived from that source code.

Figure 8 illustrates the Apple XNU random number generator architecture on the left hand side of the picture. That random number generator includes the noise source implementation. The figure contains the gray boxes of the HMAC DRBG and the read_frandom and read_erandom functions. These components are implemented in the XNU random number generator but are relevant for specific use cases inside the kernel only. General purpose

cryptographic use cases, however, do not access these components. Therefore, they are not covered in this discussion.

The XNU noise source is implemented solely in software and monitors the interrupt activity of the entire operating system.

The unpredictable phenomenon is the timing of interrupts. Every time the top half of the interrupt handler is triggered by the receipt of an interrupt, the XNU kernel obtains a high-resolution time stamp with the function ml_entropy_collect. With the open source version of XNU, only the x86 architecture specific code paths are present. For x86, XNU uses the RDTSC CPU instruction to obtain a high resolution timer. Albeit the source code for the ARM architecture applicable to iOS is not publicly available, it is assumed that a high resolution time stamp from the ARM CP15 co-processor is used as well. The high-order 32 bits from that time stamp are discarded. Only the low 32 bits are used for the subsequent processing.

The function ml_entropy_collect is only invoked for interrupts received on CPU0 – the master CPU. All interrupts received by other CPUs do not contribute to the entropy collection.

Before discussing the process of how the time stamp of an interrupt is processed, a word about maintaining the data is important. The XNU kernel maintains an entropy pool which is a 64 byte kernel global variable. As only the master CPU executes the function, no locking is necessary. The entropy pool is segmented into 4 byte words. Considering the size of 64 bytes, the entropy pool holds 16 words. In addition to the entropy pool words, an index pointer is maintained. This index pointer is incremented by one before modifying the entropy pool. As that index pointer points to a 4 byte word, incrementing this variable implies that it points to the next word in the entropy pool. The logic for incrementing this pointer also addresses the “wrap-around”, i.e. when the current value of the pointer points to the last word in the entropy pool, an increment will point to the first word again.

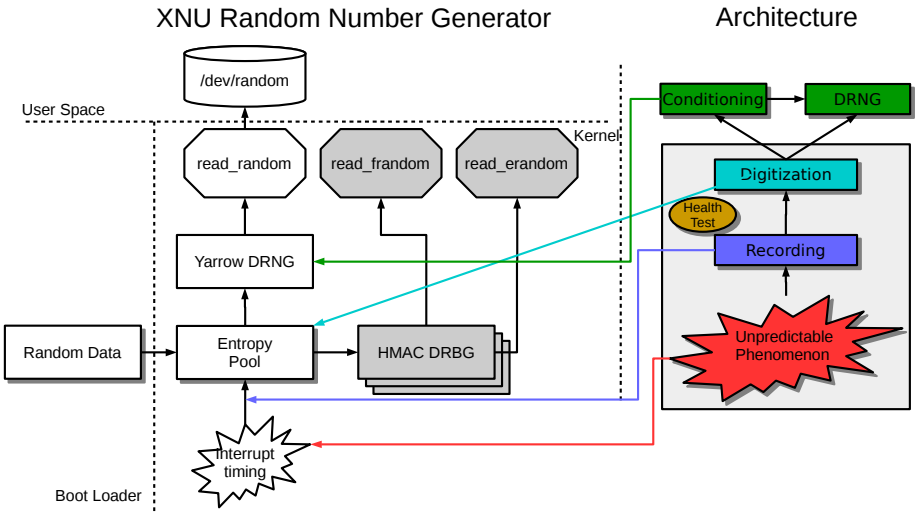

Figure 8: Apple XNU noise source mapping to noise source architecture

Now, when a new time stamp is received the obtained low 32 bits of that time stamp are rolled right[16] by 9 bits followed by XORing the time stamp into the entropy pool word pointed to by the index pointer. After completion, the index pointer is incremented. With this approach, the entropy pool is updated sequentially with each time stamp of a newly arrived interrupt.

To obtain data from that entropy pool, the requested amount of data up to 64 bytes are simply provided to the caller, in this case the Yarrow DRNG. The read out of the entropy pool is followed by a shuffling operation of the entropy pool where all adjacent 4 byte words are

16 The given number of right-most bits are removed, all other bits are shifted right by the given amount and the initially removed right-most bits are concatenated as left-most bits.

XORed. i.e. the first and 16th word are XORed to form the new first word. The original 2nd word is XORed with the new first word to obtain the new 2nd word, and so on.

The noise source data is used to seed and re-seed a Yarrow DRNG which serves the in-kernel read_random function as well as the user space /dev/random device. The Yarrow DRNG is reseeded with data from the noise source after generating 17,597 bytes.

To map the noise source architecture to the XNU random number generator, the right-hand side of figure 8 is relevant:

- The unpredictable phenomenon is the arrival of interrupts and their respective arrival timing.
- The recording operation is implemented with the rolling and XORing of the newly obtained time stamp of one interrupt with the current word of the entropy pool.
- The digitization logic is provided with the entropy pool. The reason for this is that the entropy pool is read and this data is directly used as seed for the Yarrow DRNG.
- Neither an online health test (ensuring that the noise source produces entropic data) nor a start-up health test (ensuring that the noise source logic like its cryptographic operation operates properly) is implemented.

3.4 Conclusion of Design Discussion

The preceding sections list for each noise source implementation the following information that will be used in the next chapter:

- Location of the implementation, i.e. whether the noise source is implemented in hardware, in software or is a hybrid of software and hardware.
- If the noise source is implemented in hardware, a reference to its interface for software is given.
- If the noise source is implemented in software, the rationale provides a list of dedicated hardware support required to operate the noise source.

With the following chapter, the operation of a virtual machine monitor (VMM) and how it affects noise sources is introduced. To understand the influence of a VMM onto a noise source, this information is vital.

4 Virtual Machine Monitor Impact on Noise Sources

VMMs are an abstraction layer between the software of an operating system and the hardware. The VMM has the goal to „virtualize“ parts or all of a hardware to allow a concurrent execution of another operating system. All concurrently executing operating systems must be made to believe that they exclusively act on the hardware resources.

To analyze the impact of VMMs on noise sources, first step is to find the relevant properties of the VMM which are the key to understand their impact on noise sources. In a second step, these properties are connected with the noise source architecture description to explain the theoretical impact of VMMs. Following the theoretical discussion, particular VMMs will be analyzed to identify which practical impact they have on the noise sources discussed in section 3.3.

4.1 VMM Access Mediation to Resources

A VMM's concurrent execution of guest operating system requires that it mediates access requests to resources present on the current hardware system. Such access mediation can be illustrated with figure 9.

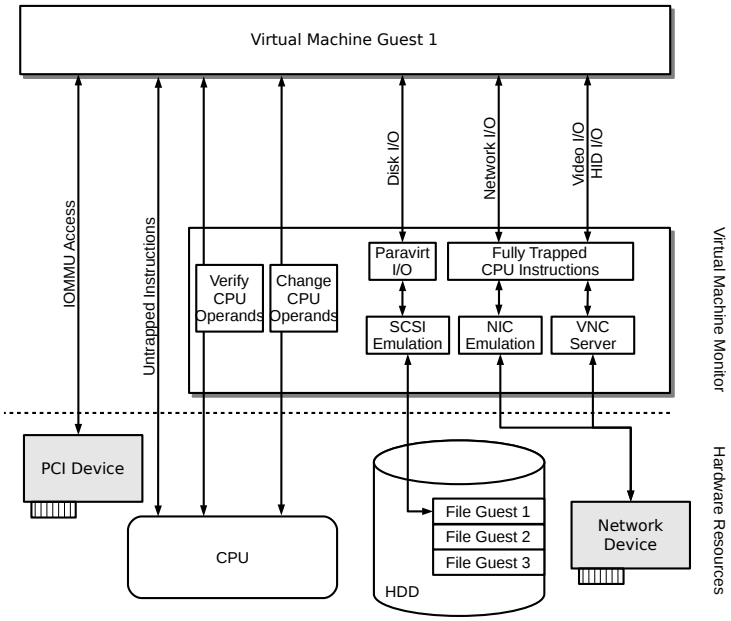

Figure 9: VMM Access Mediation to Resources

Figure 9 shows a guest triggering various operations that are intended to be handled by hardware illustrated by arrows. The analysis of the different types of requests is given in the following section.

4.1.1 VMM Access Mediation to Hardware Resources

To support a concurrent operation of operating systems, which are also called guests or guest operating systems, the VMM must classify all available hardware resources as follows:

- Hardware resources that are not accessible to a guest, but only to the VMM. Since the VMM fully controls these resources, it can make them available to guests as follows: The VMM provides a device to the guest which the guest can use with a device driver. Such a device, however, exists purely in software and is not backed by hardware.

  The VMM instantiates that device for each guest it wants to make that virtual resource available. Now, the guest “sees” a device, which it can access with a regular device driver. When the guest issues an access request to that device, the VMM intercepts it and performs any kind of operation on the resource according to its own policy. One example for such type of resources is virtualized block devices for the guests, which are technically backed by files stored in the file system of the VMM (see figure 9 with a

guest issuing disk I/O). The VMM “translates” access requests to these block devices by the guest into file access requests, following the example given in figure 9. Another example is the support for graphics cards to guests as depicted in figure 9, where the guest issues video I/O. The VMM emulates a graphics card to the guest where any request is translated into a Virtual Network Computing (VNC) server or Remote Desktop (RDP) server operation. This allows an administrator to access the console of a guest by accessing the respective VNC or RDP server endpoint offered by the VMM.

Common approaches of making such resources available to guests in today's VMMs take the following forms which are visible in figure 9:

- Paravirtualization: The VMM exports a device to the guest whose interface is implemented in such a way that the overhead for the VMM to interpret the received data is minimal. The paravirtualized device does not resemble a device that has an equivalent hardware. The guest must use a dedicated device driver that is implemented to get used by the respective VMM. An example is a virtio device as defined in [VIRTIO].
- Full virtualization: The VMM exports a device to the guest which is accessible in a manner that is equivalent to a real physical hardware device. The guest may use a device driver it would normally use to access a physical hardware if it would execute directly on hardware. An example is a VESA graphics card.

- Hardware resources may be exclusively assigned to a guest. A 1:1 mapping between hardware and consumer can be established and must be enforced by the VMM. Examples of such resources are PCI cards where one card, such as a network interface card, is exclusively assigned to a guest as depicted in the left hand side of figure 9. Commonly, a VMM does not intercept any requests made by a guest to these type of devices. A VMM configures the hardware virtualization support such as the Input/Output Memory Management Unit (IOMMU) in such a way that the hardware enforces any virtualization limits of memory access limits.
- Hardware resources may also be shared between guests. The sharing is commonly implemented by serializing the access requests from guests to a hardware resource. The VMM can assign the resource for a certain amount of time to a guest, and this guest can use the resource exclusively during this time slot.

  For example, the VMM can assign one or more CPU cores to a guest. However, the device is still accessed by the guest like it would access a native hardware component. The serialization now implies that at one point in time the VMM stops the guest from being able to access that device. This may be due to a re-scheduling of the guest when the resource of a CPU core is considered.

  The prime example for such types of resources is the CPU itself which is shared by guests as controlled by the VMM. Another example is RAM that is assigned on-demand to guests in a method that is conceptually similar to how operating systems offer memory to applications. To support such hardware sharing, the VMM commonly configures the hardware to exclusively grant access to the resource to one guest. After the expiration of a time window, the VMM halts or intercepts all requests to this resource to reconfigure and reassign it. The time window in which a device may be accessible by a guest may be in the range of microseconds considering scheduling based sharing – such as the CPU itself – up to days, weeks or even months for administrator-based device reassignments. An example of devices with a long time window is a PCI device or a USB device that is re-assigned to another guest by the administrator.

4.1.2 VMM Access Mediation to CPU

Orthogonal to the support of hardware resources including the CPU is the handling of CPU instructions. The VMM must intercept certain types of CPU instructions issued by the guest software to analyze them and take appropriate actions.

When a VMM wants to intercept a CPU instruction, it configures the CPU's virtualization support such that either a white-list or a black-list of CPU instructions to be trapped is defined. When the CPU executes the virtual machine, it looks up each instruction in that list

and traps that instruction if found. A CPU trap translates into an exception that is relayed back to the VMM. The VMM has an exception handler that implements the logic to handle such traps. When trapping an instruction, the CPU does not execute the trapped instruction. Instead, the CPU gives control to the VMM with the information about the trapped instruction to allow the VMM to decide whether to re-execute the instruction or to enforce another approach.

Regarding the trapping CPU instructions, a VMM can implement the following possibilities:

1. Well defined CPU instructions may be issued by guests that are not trapped by the CPU and thus intercepted by the VMM as illustrated in figure 9. Such instructions are deemed to be uncritical to the virtual machine separation and isolation because either their operation has no impact on the VMM goals or the CPU already implements different kinds of virtual machine separation and isolation checks. In the latter case, the VMM must rely on the correct enforcement of the virtual machine boundaries by the CPU configured by the VMM. As the CPU instructions are not intercepted, they execute with native speed that is indistinguishable from the execution of the operating system directly on hardware.
2. A set of pre-defined CPU instructions is intercepted and its operands are verified that the calling guest is allowed to make the request as depicted in figure 9. After the operands are checked that they do not violate the boundaries the VMM has defined for the guest, the instruction is now allowed to proceed, i.e. the instruction is handed to the CPU for execution. If the operands do not comply with the VMM rules, the instruction is not forwarded to the CPU and an exception is returned to the calling guest.

   Such an instruction intercept adds a delay in the execution of the CPU instruction compared to executing the guest directly on hardware. But this processing does not alter the mnemonic or the triggered operation of the CPU instruction.
3. A set of CPU instructions are intercepted and analyzed which is also shown in figure 9. Depending on the type of instructions and operands, the VMM alters the instruction operands and forwards the instruction with the altered operands to the CPU for normal processing.

   This instruction intercept not only adds a delay in the execution of the CPU instruction but also changes the resulting operation. An example for such an alteration of operands are CPU instructions accessing memory where the CPU does not implement a second-level address translation, such as Extended Page Table (Intel) or Nested Page Table (AMD). In this case, the VMM must trap such memory accesses, alter the memory address to add the offset of the memory starting address of the guest memory.
4. The VMM intercepts CPU instructions with the goal to not forward the instruction to the CPU. Instead, the VMM performs a completely different operation which may be a different set of CPU instructions – or no operation at all. The typical example is the implementation of virtual devices whose concept is already introduced in the listing of access mediation to hardware resources, bullet 1 above. With such types of intercepts, software that is not aware of the VMM can neither rely on the timing of the CPU instruction nor that it accessed hardware at all.

The VMM may add one more kind of interception that has not yet been discussed, but cannot be mapped to particular CPU instructions or hardware: interception of the guest operation when the CPU issues exceptions. The most common exception issued by a CPU to be handled by the assessed VMMs is the page fault. Even when a CPU instruction issued by a guest is not planned to be intercepted by the VMM, the CPU may page fault when the VMM did not allocate the requested memory before. This is logically equivalent to memory handling in operating systems. When such a page fault occurs, the issued guest-issued CPU instruction is left untouched, but the VMM updates the memory configuration and re-issues that CPU instruction. Therefore, such page faults extend the duration a CPU instruction takes to process. Commonly, other CPU exceptions such as illegal instruction exceptions are reflected to the offending guest to handle. Yet, the CPU first notifies the VMM about the occurrence of

such exceptions and the VMM must identify the offending guest to relay that exception to. Again, this logic adds delays in the execution time of guest operating systems.

In reality, there are not just these aforementioned concepts, but the VMM has the freedom to implement the continuum of operations from not intercepting a request up to the point where the request is completely changed. However, the discussed types of CPU instruction intercepts must be used by the VMM. For example, the VMM may intercept a CPU instruction, and verify one operand and modify a second operand, combining two types of the aforementioned interception possibilities.

In addition to intercepting CPU instructions for immediate processing, the VMM is also capable of catching any CPU instruction at any time for the purpose of putting the guest operation on the respective CPU to sleep. With the expiry of the execution time allotted to a guest (also called the CPU time slice), the CPU virtualization support will interrupt the processing of the CPU instruction stream originating from a guest to return control of the CPU to the VMM. Technically this is implemented as follows: when the expiry of the time slice is triggered in the CPU, the CPU completes the current CPU instruction performed for the guest. But instead of fetching the next CPU instruction of the guest, it obtains the CPU instruction defined by the VMM – at this time, the VMM logic takes control of the CPU. The VMM now may, depending on its capabilities and configuration, do the following:

- Assign the CPU to another guest or for VMM use with the goal that the guest that was just put to sleep will again obtain control of the CPU after the expiry of the time slice for the other guest. In this case, the guest is temporarily suspended for rescheduling.
- On the other hand, the VMM may also not intend to give control of the CPU back to the guest just to put the guest to sleep. The VMM will save the state as needed and put the state of that guest to a halt. The halting of a guest can be used, for example, for:
  - pausing the guest is akin to a hibernation of an operating system,
  - taking snapshots for backup or for instantiating the guest again, or
  - copying or moving the guest to another environment.

4.2 VMM Impact on Noise Sources

Now, the key understand whether a virtual machine monitor has an impact on a noise source is to answer the following question: Are any of the resource accesses required for the cryptographic strength of a noise source subject to any interference by the virtual machine monitor?

This question can only be answered by analyzing each noise source and identifying all hardware accesses required by this noise source. Before the different noise sources discussed in section 3.3 are analyzed, a discussion of the architectural model of noise sources is of interest.

After illustrating the general VMM operation and how it impacts the guest operation, the following general findings can be derived:

1. If a guest implements a software logic that only operates on the CPU with potential memory fetches, but is not operating on other hardware, the probability that the VMM intercepts CPU instructions of that guest is minimal to non-existent considering the assumption that the VMM acts benignly. If an interference occurs it is only the CPU instruction intercept to verify that the operands do not overstep the boundary of the virtual machine. However, no alteration of any CPU instruction is performed. For example, if the guest implements a cipher like SHA-256 completely in software, the VMM will not intercept.
2. Any operation that is exclusively performed in hardware is equally not interfered with by the VMM. To be more precise: the VMM does not have the ability to interfere with the operation of the hardware gates. Similarly, when an operation is implemented in

firmware which executes on hardware that is inaccessible to the VMM[17], it is unaffected by a VMM operations.

3. The interference of a VMM with the operation of a guest is always at the borderline between hardware resources and software. A guest operating system is always implemented in software. Every time the guest interacts with the hardware where the virtual machine limits are not enforced by hardware[18], a VMM may and most likely will interfere.

In the subsequent subsections, the different impacts of the VMM on noise sources is analyzed. In the first step, the study leaves the realm of the assessment of noise sources a bit by discussing common errors iwhen using VMMs that have a direct impact on noise sources and entropy. This is followed by analyzing the impact of the VMM on the timing of events. In a next step, the impact of a VMM on the noise source architecture given with figure 1 is discussed.

4.2.1 Common Errors in Use of VMMs

When using virtualization environments, it is easy to make common mistakes when considering noise sources. This section enumerates the common errors and explains how to avoid them. The avoidance of these common errors eliminates the respective impact of the VMM onto the noise source operation. As steps for avoiding these common errors are given here, they are not further discussed in the remainder of this document assuming that administrators will apply the mitigation steps.

All usage errors revolve around the handling of the state information of a noise source held in resources accessible and manageable by a VMM. Commonly, software noise sources maintain a state like an entropy pool or simple variables that are needed for implementing software aspects of noise source components specified in figure 1. Naturally, this state information must be protected against unauthorized access as well as duplication. This duplication implies that two instances of a guest have the same state information which is used to maintain entropy. When duplicating such state information, the entropy must be considered gone as the output from one guest now allows to make predictions of the noise source behavior of the other guest[19]. Mathematically speaking, the duplication introduces a dependency of the outputs of the now two noise source states.

4.2.1.1 Noise Source State in Cloud Environments

In Cloud environments, which are a special use case of VMMs, guest operating systems are created and deployed on-demand. Administrators generate system images of guest operating systems by installing such a guest and then taking snapshots as explained in section 4.1.2. When the snapshot is taken of a hibernated guest operating system image, the entire state of the guest operating system is present. If a noise source is completely or even partially implemented in software, that state will therefore be duplicated as well. It is therefore utmost important that any cloning or other duplication of a guest operating system instance shall only happen if its noise source(s) are deactivated. This is usually the case when the guest operating system is properly shut down.

Hence, the administrator of the cloud environment should only perform snapshots and duplication of guest operating systems when they are shut down.

One special case of a particular and important noise source is to be considered: the LRNG commonly saves a file with random data on non-volatile storage, commonly in the /var directory[20]. When duplicating a guest Linux operating system image even when it is shut down, that file is duplicated. When the guest Linux operating system boots, the contents of that file is written to /dev/random. As discussed in [LRNG], when writing data to /dev/random, the data is mixed into the blocking_pool and nonblocking_pool, but not into the input_pool.

17 High-end network interface cards have a small processor of its own that is driven by firmware. This firmware operation cannot be interfered from the VMM.
18 Such as memory or IOMMU controlled devices.
19 This issue applies to other cryptographic relevant states like open cipher handles of cryptographic libraries, the state of a deterministic random number generator and the like.
20 The particular location of the file varies slightly in the different Linux distributions.

Also, the writing of data does not alter the entropy estimator governing the blocking behavior of the blocking_pool and the input_pool. Hence, the LRNG does not assume that the data read from the saved file and written to /dev/random contains any entropy. That data is further used to mix the pool but does not affect whether a pool is believed to contain entropy. Therefore, a duplication of guest Linux operating system images where the saved random number file is duplicated, too, does not have an impact on the LRNG and the maintenance of its entropy. Hence, it is considered immaterial to the noise source behavior or the entropy collection whether that file is left unchanged, is deleted or modified.

4.2.1.2 Noise Source State in Backups

Snapshots of guest operating systems may also be taken for backup purposes. Again, in this case the same finding from above applies here, too. If a backup needs to be resurrected, it must completely reset the state of the noise source which is commonly performed with a full reboot of the guest operating system. Another issue is the aforementioned loss in entropy: To avoid that loss and yet create a snapshot of a running guest, an administrator has toprotect the backup image properly from any untrusted access. For example, the backup image can be stored using encryption and integrity protection where the security strength of the cryptographic mechanisms should match the required level of protection of the random numbers.

4.2.1.3 Noise Source State when Transferring a Guest

Another example of a duplication of a guest operating system is the transferring of a guest from one VMM host to another. Such a transfer is commonly implemented with the following steps:

1. The guest in the originating VMM is halted.
2. Its image is copied over to the receiving VMM.
3. The guest in the receiving VMM is scheduled and therefore resumes operation.
4. The guest in the originating VMM is erased.

With current VMMs, it is even possible that the disk image required by the guest will be shared between the two VMMs using a storage area network (SAN) or similar where a transferal is not required. Thus only the transfer of the RAM state of the guest to the receiving VMM is needed. This transfer is now performed while the originating VMM keeps the guest running. In an iterative process, all memory pages are copied to the receiving VMM and marked as clean in the originating VMM. When the guest performs a write operation, the VMM marks the respective memory page as dirty and in a next round, only the dirty pages are copied again. After a few loops the majority of all pages are now current between both VMMs. At that time the guest is halted, and the still dirty pages are copied over. The goal with this approach is to minimize the down time of the guest, i.e. the time between the halting of the guest on the originating VMM and resuming the guest on the receiving VMM.

Again, the description of the transfer process shows that the state of the guest is shared. Thus, at one point two copies of the guest operating system exist. The VMM implementation as well as the administrator must now be prepared to handle an error in the transfer process when these two copies exist. The error handling must ensure that only one guest instance is resumed and the other is immediately erased. As state information pertaining to the noise source is usually held in RAM, the error handling must ensure a proper clearing of the RAM of the guest instance about to be terminated, e.g. by overwriting it with zeros. Another important aspect is that the link between both VMMs must be protected at a level to maintain the the confidentiality and integrity of the entropy in the state. Similarly to the storing of a backup, when using a cryptographically protected link between both VMMs, the cryptographic mechanisms must be as strong as the entropy that can be extracted from the noise source's state.

4.2.1.4 Noise Source State during Suspend and Resume

Another example where the state of a noise source can be jeopardized is the suspend and resume cycle that may be provided by a guest operating system. To implement suspend, the operating system uses a file system object (such as Windows) or a disk device (such as Linux) to store the entire RAM content onto a persistent storage device.

As mentioned previously, noise sources that have parts or all aspects implemented in software commonly maintain their state in RAM. With the suspend operation, that state data is flushed to disk. Whereas the RAM may be protected by the active VMM, when a guest has stored its state onto a non-volatile memory location, the VMM may also choose to cease to operate without impacting the guest. In this case, the non-volatile storage of the guest is not covered by a VMM protection any more. For example when booting another software instead of the VMM, the storage locations may not be protected against unauthorized accesses. The administrator therefore must now ensure that the guest image is appropriately protected to avoid unauthorized access to this image. One solution is to encrypt the storage location where a guest stores its image. Again as mentioned several times above, the strength of the used cryptographic mechanism must be as strong as the entropy that is expected from the noise source.

Upon resume, an operating system usually does not prune, i.e. securely erase, the non-volatile store of the previous image. Therefore, the administrator must still ensure protection of that image, considering that in the worst case:

1. the guest suspended,
2. the guest is resumed,
3. the guest operating system now shuts down after resuming, and
4. the VMM shuts down shortly thereafter, too.

Albeit no active suspend image is present in the non-volatile store, the image now still contains the image of the previous suspend. If unauthorized access to this image may be possible, an attacker may at least backtrack a potential seed of a deterministic random number generator. Again, for guarding this scenario, an administrator must protect that non-volatile storage area as mentioned before.

4.2.1.5 Noise Source State on Swap Space

The scenario discussed in section 4.2.1.4 for the non-volatile storage area holding suspend data equally applies to any non-volatile store used for the purpose of swap space. All findings in section 4.2.1.4 apply therefore also to the swap space store.

4.2.1.6 Device Reassignments

As depicted in figure 9 and discussed in section 4.1.1, a VMM may assign devices for exclusive access to guests. As explained, such exclusive assignments may be canceled for a particular guest and the respective device may be reassigned to another guest or even the VMM itself.

If a device is used as a hardware resource supporting or providing the noise source, special care must be taken during the device reassignment. Hardware resources may keep a state that is relevant to the entropy collection from that resource. Naturally, if such state information pertaining to the entropy is still present after the device has been reassigned to another guest, the entropy in the originating guest must be considered diminished or even eliminated. The receiving guest now owning the device may be able to reconstruct all or parts of the data that used to contain entropy for the originating guest.

Therefore, it is mandatory that for a device reassignment, the state information found in that guest must be deleted unconditionally during that reassignment period but before the receiving guest is taking control of the device.

Two types of device reassignments must be considered: a device reassignment automatically performed by the VMM and a device reassignment performed by the administrator of the VMM.

- Automated device reassignment: A VMM may automatically reassign devices based on well-defined rules. In such a case, the VMM has full information about the device type, the originating guest and the receiving guest. For this type of device reassignments, it is expected that the VMM implements the appropriate logic to clear the state of that device before it is reassigned. A very common reassignment of hardware resources is the main memory. Most current VMMs allow a guest to request additional memory or release memory. When a guest has released memory and another requests it, a VMM

may assign memory pages to the second guest that used to belong to the first guest. Before such an action is performed, the VMM now must overwrite the memory with zeros to prevent any kind of information leak.

- Administrator-triggered device reassignment: An administrator may exclusively assign a hardware resource, such as a PCI device, to a guest. At a later time he may decide that this very PCI device shall not be owned by the initial guest, but by a second guest. Now, he can trigger a device reassignment where he re-configures the VMM such that it severs the link between the originating guest and the device. This is followed by establishing the link between the receiving guest and the device. Commonly, the VMM has little or no information about the device and how state data is maintained with it. Therefore, it is the task of the administrator to clear out all state information from the device to be reassigned before the receiving guest is able to take control of it. State information pertaining to entropy and noise rarely if ever is maintained in non-volatile store. Thus, an administrator commonly can assume that relevant state information is kept in volatile storage that loses its information during a power cycle. Therefore, unless the administrator has better knowledge of the device, a power cycle of at least the device to be reassigned should be performed by the administrator to ensure that no state related to entropy is present when the receiving guest takes control of the device.

4.2.1.7 Sharing of Hardware Resources

As mentioned before, VMMs may assign a hardware resource directly to one guest. With such assignment, the VMM does not interfere with the operations performed by the guest on that device. Depending on the use cases, it may be conceivable that a hardware resource is assigned to more than one guest instead of to just one. When that mapping is configured by the VMM such that the VMM does not mediate the access to the resource, two guests now have access to a resource where the guests can directly interfere with each other.

A VMM can configure such hardware resource assignment to multiple guests by configuring, for example, the IOMMU such that the DMA destination of a hardware resource is allowed to be accessed by multiple guests. In addition, the VMM updates its memory mapping such that the MMIO registers are mapped into multiple guest address spaces.

If such a hardware resource is used by one or all guests to provide an aspect or a complete noise source, per definition no entropy can be obtained from such resource. Since all guests having access to that resource also have the ability to observe the state and events of that resource equally, any phenomenon that is considered unpredictable is now accessed by multiple guests. Therefore, the resulting event or data is not unpredictable any more as it is known to multiple guests.

Such sharing of hardware resources is rare in real-life because such sharing would violate one of the major goals of a VMM: the isolation and separation of guests. For the purpose of noise sources and entropy collection, an administrator must not assign a hardware resource to multiple guests unless he undoubtedly knows that this resource is not used in noise source implementations of any guest.

4.2.1.8 Sharing of Hardware Resources: SR-IOV

A twist to the sharing of resources discussed in section 4.2.1.7 is the use of hardware resources supporting the Single Root Input/Output Virtualization (SR-IOV) that is found on x86 architectures.

An SR-IOV device is a multi-function device where the hardware device is architected with virtualization support. Each function of an SR-IOV device can be assigned to a different guest using device assignment. Yet, the guests cannot interfere with each other as enforced by the hardware resource.

An example of such SR-IOV devices are high-end network interface cards. Such cards may host multiple physical ports for Ethernet cables. In addition, that card has one processor handling network traffic for each Ethernet port. Finally, a virtualization logic is present that ensures that a combination of one physical port and one processor can be independently operated from the other physical port and processors. Such SR-IOV cards may be perceived

as multiple PCI network interface cards merged into one. Each function of an SR-IOV card is accessible with its own PCI bus number by the software.

The SR-IOV cards are designed to maintain separation between guests. Yet, they have shared resources like common internal buses. Considering that a number of flaws were found in the hardware and firmware implementation of SR-IOV cards, it is not fully clear whether the initial objective of separating guests is implemented with an appropriate architecture. Furthermore, the potential presence of shared components within an SR-IOV card hints at the possibility of side channels. When such devices are used to support noise sources, these aspects are critical.

As a safe baseline, the SR-IOV functionality should be deactivated if that hardware is used to support noise sources. It could be enabled after a careful study of how the architecture and implementation of a particular SR-IOV card impacts a noise source operation and entropy collection.

4.2.2 Side Channels in VMMs

A common security issue with VMMs is the presence of side channels which in almost all cases are timing channels due to the use of shared resources. With such side channels, one guest may partially observe the operation of another guest. For example, by creating a certain structure in the caches, followed by a switch to the victim guest, followed by a switch to the attacking guest, the attacker may now analyze the contents of the cache and deduce operations performed by the victim guest.

Such side channels are a common security issue which could also be used for obtaining information about the noise source operation or entropy collection performed by one guest. As this issue is a general problem with VMMs and not restricted to noise sources or entropy handling, this study will not further examine side channels and how to prevent them.

Ultimately, if an administrator of a guest is really worried about side channels, VMMs should not be used. The operating system in question should execute on its own dedicated hardware that does not execute any other operating system.

4.2.3 Execution Time of VMM

In chapter 3, a common theme for noise source emerges: regardless of the type of noise source, the exact timing of events is in almost all cases the root of the unpredictable data. The VMM operation with the interception of requests from guests adds time delays which which would not be present if the guest operating system was executed natively on hardware.

The VMM execution time can add time delays in the range of microseconds when intercepts are processed, up to weeks or months if the guest is not scheduled any more as discussed in section 4.1.2.

Noise sources whose entropy collection is based on the timing of events where the VMM intercepts a guest's interaction with the noise source are always affected by the VMM operation. Obviously, a VMM has an indirect impact on those noise sources as the VMM will take away processing time from the guest operating system. For example, the scheduler of a VMM may simply interrupt a guest operation, because it is time to schedule the next guest. Or, the VMM may interrupt the guest to perform instruction completion. Now, when the guest waits for an event to happen and wants to use the event's time stamp as a source of entropy, the guest will see that event happen at a later time if the VMM operation re-schedules the guest.

This results in the conclusion that entropy collection based on timing where the VMM can interfere is impacted by the general VMM operation.

One key aspect must be considered to assess the impact on VMM operation: noise sources that depend on the timing of events derive the entropy on the unpredictability of the timing of the event occurrence. It is important to understand that for such noise sources, the entropy is always in the time deltas to the previous event, i.e. when comparing the timing of two or more events. A series of events that occurs at a regular interval where the used timer will always measure the same time delta will obviously have no entropy because at least the events are not independent and most likely predictable. The key for the entropy collection is

that there are unpredictable variations in the timing of those events. This is the root cause of such timer-based entropy sources.

Now, with the VMM operation, the following extreme scenarios are possible, albeit they just define the edges of a continuum of possible scenarios – note, in systems with a high-resolution timer, such extreme scenarios are unlikely and thus define worst cases:

- The VMM operation and re-scheduling occurs at such a regular interval that every time the guest wants to measure the timing of an event, the generation of the time delta simply eliminates the VMM impact. The following formula explains it very clearly where a is the static time added by the VMM operation and the variable t_i is the time stamp of the event that is observed. The result t_{delta} is the time delta where entropy is believed to exist.

  $$t_{delta} = (t_2 + a) - (t_1 + a) = t_2 - t_1$$

  The addition of a constant time onto the time measurement can be interpreted as if the guest is executed on a slower CPU where the CPU's timer has a higher resolution compared to a CPU that is indeed slower and its clock frequency is lower.

- The VMM operation and re-scheduling occurs at intervals that are either non-deterministic or have a deterministic pattern. In this case, the calculated time delta is impacted by the VMM operation, i.e. the time delta that is supposed to measure the occurrence of an event now contains time variations added by the VMM operation. Again, depicting that impact with a formula:

  $$t_{delta} = (t_2 + a_2) - (t_1 + a_1) \neq t_2 - t_1$$

  In this formula, the VMM operations add the time a_1 and a_2 to the time measurements. From the viewpoint of the noise source measurement, it is unknown whether the differences between a_1 and a_2 are due to deterministic factors or non-deterministic factors.

When assuming that the VMM operation is not intended to deliberately sabotage the guest time measurements, i.e. the VMM is assumed to behave in a benign manner, the only question left to be answered before a conclusion is: Is the execution time of the VMM operation independent of the event the guest wants to measure? Using the mathematical symbols just introduced, the question arises whether t_i is statistically independent from a_i .

To obtain hints whether they are independent, the VMM operation must be considered. The VMM operation commonly includes:

- Re-scheduling of guests or virtual CPUs because their allotted time expired.
- Interrupts and similar operations that the VMM needs to handle and to potentially reflect to a guest.
- VMM internal operations triggered by the receipt of a CPU exception, such as a hypercall. Such CPU exceptions are logically equivalent to a re-scheduling operation from the guest to the VMM.

On the other hand, a guest using a noise source depending on the timing of events requires the receipt of an event and obtaining the associated time stamp. Section 4.2.4.2 discusses the time gathering CPU instructions and how a VMM may interfere with them. To keep the discussion in this section concise it is assumed that the VMM does not interfere with the guest time stamp gathering operation.

If the aforementioned variables are dependent, the VMM would need to monitor or trigger the event occurrence in a way that certain VMM operation is deliberately triggered based on this observation. Only this way, a VMM operation would be able to cause a dependency of its operation with the guest's noise source using event timings. Technically, such dependency can be easily achieved. Practically when considering the common VMM tasks listed above, none of these tasks even remotely hint to such dependency. Therefore, the VMM operation

can be identified to be independent of the timing of events as used in noise sources. Thus, the following can be concluded:

- t_i is a series of timing data collected by the noise source which contains entropy at a level assumed by the noise source.
- a_i is a series of VMM-based time offsets whichmay or may not have entropy relatively to each other.
- t_i and a_i are independent relative to each other as discussed above. This means that adding the VMM-based offset of a_i to the entropic value of t_i will not diminish the entropy in t_i .

Considering these general tasks of a VMM, one can generally conclude that the logic implementing the timing of events performed by guest operating systems is independent from the VMM operation. Thus, the following can be concluded: the additional time the VMM adds to the time measurements of a guest is independent from the guest's time measurement.

Leveraging this independence of the VMM operation from the guest timing operation, the following conclusion can be drawn: the VMM operation does not have any impact on the entropy that is derived by the guest from measuring the time deltas of events, which clearly prevents any disadvantageous effect on the entropy estimate. If the timing of the VMM operation varies and is non-deterministic, it even adds entropy to the time delta measurement of the guest, which is a benefit for the noise source's entropy.

4.2.3.1 LRNG Entropy Estimator

The LRNG has a specialty which is rarely found in other random number generators and associated noise sources: it maintains an entropy heuristic by awarding each received event an entropy estimate.

That entropy estimate is calculated by obtaining the first, second and third derivation of the jiffies time stamp of an event compared to the previous event occurrences. The jiffies time stamp is a coarse timer which, depending on the compile time options of the Linux kernel, runs with 1000Hz – i.e. the timer variable is increased by one each millisecond. Please note that with the current implementation, simply switching to a high-resolution timer for the following calculation will increase the entropy estimation, but commonly overestimates entropy significantly.

Using the variables from above, the variable t_i would act as the jiffies time stamp value. Therefore, the following applies:

$$t_{event} = t_i + a_i$$

The calculation of the heuristic entropy value without interference by the VMM can be characterized with:

$$H_{event} = min\left(11, min\left(\left|\left(f'(t)\right)\right|, \left|\left(f''(t)\right)\right|, \left|\left(f'''(t)\right)\right|\right)\right)$$

where discrete derivatives are used. When considering the interference by the VMM during the heuristic entropy calculation

$$H_{event} = min\left(11, min\left(\left|\left(f'(t+a)\right)\right|, \left|\left(f''(t+a)\right)\right|, \left|\left(f'''(t+a)\right)\right|\right)\right)$$

This formula implies that the first, second and third derivatives of the “real” jiffies time stamp added with the timing of the VMM operation are calculated.

The following question now needs to be answered: Is it possible that the VMM operation defined with a_i increases the heuristic entropy calculation?

The clear answer is: Yes, it can. Since the formula above indicates the use of the absolute value of the derivations, which implies that the lower boundary is zero, it is more likely that the calculated H is increased by the VMM operation.

The question aries of that is a problem. As mentioned in section 4.2.3, the VMM operation may add entropy on top of the noise source operation. Such entropy is only added if a_i changes over time. Mathematically, the first, second and third derivative of a_i must not be zero in this case. The calculation of H simply picks up the entropy added by the VMM operation.

Of course it is technically possible that the VMM has a deterministic operation that produces a pattern of a_i which is not picked up with the aforementioned calculation of H. But this very same problem is present for the noise source operation too. The calculation of H is only a heuristic for the estimation of the entropy. Considering [LRNG] which shows that the LRNG entropy heuristic is very conservative, it is therefore equally assumed that the conservative calculation also supports the entropy calculation when executing the LRNG as a guest.

Therefore, the result of this discussion is: The heuristic entropy calculation of the LRNG within a VMM is likely higher compared to executing the LRNG on bare metal. But this higher value mirrors the additional entropy the VMM operation adds to the existing noise sources.

To support this finding, testing in section 5.3 will be conducted to measure the heuristic entropy and compare it with the measured entropy.

4.2.4 VMM Interference with Noise Source Architecture

After discussing ancillary errors that could be made regarding the handling of guests in VMMs that would adversely affect noise sources and concluding that the timing behavior of the VMM in the worst case is immaterial to noise sources, this section now brings the noise source architecture discussion from section 3.1 together with the VMM behavior regarding resource access as explained in section 4.1.

With the entire description delivered in chapter 3, it should now be clear that noise sources are always an external resouce from the point of view of an operating system. At least the basic part, the unpredictable phenomenon, is anchored somewhere in hardware. Therefore the explanation on how VMMs mediate access to resources from 4.1 must now be applied.

Figure 10 provides an illustration of the VMM part from figure 9 on the right-hand side and the noise source architecture illustration from figure 1.

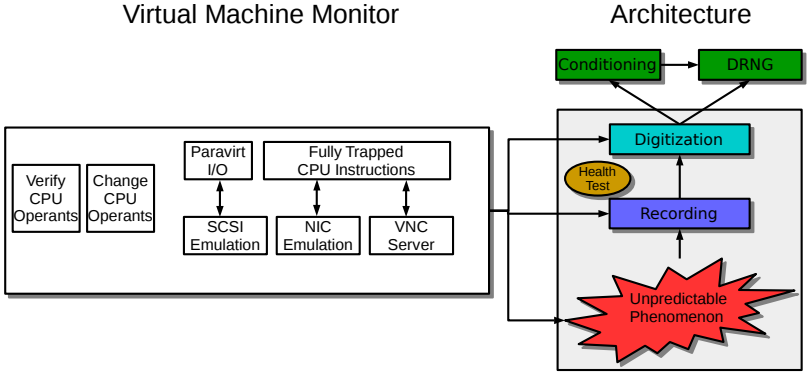

Figure 10: VMM Interference with Noise Sources

With the various types of intercepts, a VMM has the ability to influence every component of a noise source to various degrees. The following subsections explain the potential VMM interference for every noise source component individually.

Albeit the noise source components are discussed separately, depending on the implementation of a noise source, none, one or more of the noise source components may actually be interfered with by a VMM. The discussion how particular implementations of noise sources are affected by the VMM operation will be described in later sections. Nevertheless, all those sections will base the analysis and conclusions on the following subsections.

When the following subsections refer to a VMM's “interference”, it refers to how a guest's view of a component or the data obtained from that component is impacted and altered. The

term interference does not relate to a conclusion whether the entropy that is derived from the noise source and its data is affected. The discussion of how much the entropy is affected can only be answered when particular noise source implementations are discussed and thus will be subject in section 4.3.

4.2.4.1 VMM Interference With Unpredictable Phenomenon Components

As already mentioned, the unpredictable phenomenon is anchored in hardware or is observable at hardware only. At the time of writing, no architecture is known to the authors where the unpredictability is derived solely from a software operation that is independent of any hardware construct.

The first step in identifying whether a VMM can interfere with the unpredictable phenomenon is to identify where it is really found. The following general sources of such a phenomenon are present.

4.2.4.1.1 CPU Instruction

A CPU instruction may deliver an unpredictable behavior. An example would be the Intel RDRAND instruction where the instruction itself delivers an unpredictable behavior[21]. In this case, access to that CPU instruction is required. If the data provided by that CPU instruction is processed by the guest operating system, the VMM can interfere with the instruction as depicted in figure 10[22].

Before iterating through the different types of CPU instructions and VMM interference, one indirect effect VMMs have on CPU instructions should be considered. To support the performance of VMMs and its guests, the underlying hardware together with the VMM may implement different types of additional caches. Such caches may be maintained by the CPU transparent to any software. These caches implicitly have the ability to alter the execution time of a CPU instruction which will therefore have an impact on the not yet intercepted CPU instruction. As the timing of the CPU instruction is affected, the assessment provided with section 4.2.3 therefore applies for this scenario as well. The conclusion from 4.2.3 which therefore applies here as well is that in the worst case, there is no impact on the collected entropy when timing variances are introduced by the VMM operation.

A VMM now may interfere with CPU instructions as follows:

- For a large set of CPU instructions, the VMM does not intercept them at all. In this case, the VMM does not interfere with that CPU instruction.
- The VMM may intercept the CPU instruction to verify the CPU operands. Assuming that both the VMM and the guest operating system behave benignly, the verification operation will be successful and the VMM will then forward the CPU instruction to the CPU for processing. The CPU instruction result will be returned to the caller unprocessed by the VMM. Therefore, the VMM interference implies that the execution time will be enlarged without any additional interference. The conclusion in section 4.2.3 therefore applies.
- Another possibility is that the VMM intercepts the CPU instruction to change operands. A VMM commonly performs this change if memory addresses need to be changed to keep the CPU instructions within the boundaries defined for the calling guest. Assuming that a VMM behaves in a benign fashion and only wants to enforce the guest resource limitation, it must be considered unlikely that the CPU operands are changed in a way that potential unpredictability is adversely affected. Yet, it cannot be ruled out completely – an analysis of the particular noise source implementation at least has

21 The author notes that this entire document does not provide an analysis of the quality of unpredictable phenomena. Thus, this sentence is not to be used to conclude that the author endorses RDRAND. RDRAND shall only serve as an example of a CPU instruction that is intended to deliver unpredictable data.

22 Figure 9 depicts CPU instructions that are not intercepted by the CPU. To keep figure 10 self-explanatory, these non-intercepted CPU instructions are not depicted, but must be considered in this section.

to perform the assessment on how the operands are changed and whether they impact the unpredictability.

Nonetheless, the VMM interference again extends the execution time as mentioned before. Again, the conclusion from section 4.2.3 is applicable.

- Finally, the VMM has the ability to intercept the CPU instruction and to either discard it or to replace it with its own logic. In such a case, the base-line assumption an evaluator must apply is that the unpredictable phenomenon that used to be available with the CPU instruction is now not accessible any more. Hence, the noise source cannot rely on it and the guest is not expected to be able to derive any entropy from it.

  Though, it may be possible that in special circumstances the unpredictable behavior of the intercepted and completely replaced CPU instruction may still be present to some degree. For example, it may be conceivable that on x86 the RDRAND instruction is intercepted and replaced with an invocation of a VMM-provided random number generator. In this case, however, a very careful analysis of the interference of the VMM must be conducted. If such a case is identified for the noise sources and VMMs to be discussed in this document, a complete assessment is provided.

4.2.4.1.2 Hardware Resource

When considering figure 9, hardware resources other than CPU instructions are handled by a VMM by either assigning the device directly to a guest or by mediating the complete access.

When a hardware resource is assigned to a guest, the guest operating system has complete access to the hardware resource. The VMM configured the underlying hardware such that direct access by the guest to the hardware resource is allowed. When a guest performs operations on that directly assigned hardware resource, the VMM does not interfere with these operations. Hence, in such a scenario the VMM does not have an effect on the noise source operation. Nevertheless, as illustrated in section 4.2.4.1.1, the presence of additional caches may impact the performance of access requests to that hardware. As explained in section 4.2.4.1.1, that impact at a worst case has no influence on any collected entropy.

The opposite is the complete interception of access requests by the VMM to a hardware resource as depicted in figure 10. The goal with such interception is to allow a VMM to “translate” the guest access request following one type of “protocol” – i.e. the protocol that would be used with a particular hardware device – into another “protocol” to be used with a real hardware device. The following types of correspondence between the virtualized and the used real device are evident:

- The VMM may export the virtual interface of a VESA VGA graphics card to the guest. The guest now uses a VESA VGA driver to perform graphics related access requests. The VMM intercepts all of those access requests to convert them into VNC server operations that operates as a network server. In such a scenario, the virtual hardware device the guest can access has no relationship with the one or more physical devices that are exercised by the VMM. Hence, the guest cannot make any assumption about the behavior – including whether that behavior exhibits an unpredictability – of that virtualized device. Such devices are therefore unsuitable to support noise sources.
- The VMM exports a virtual device to a guest which is translated into access requests of similar physical devices. For example, the VMM exports a virtual SCSI disk interface whose access requests are converted into file operations by the VMM in case the “backend” of that SCSI disk is a regular file maintained by the VMM[23]. Again, in such cases, the base-line is that, the VMM operation renders such devices unsuitable for use in noise sources. However, when carefully studying particular implementations of noise sources in guests, it may be possible that such virtualized hardware are found to still provide some entropy and can still be used to support noise sources.

Besides a clear-cut handling of hardware by the VMM, i.e. either allowing full access to a hardware resource or intercepting all access requests to such resource, a mix of both is conceivable and currently discussed for x86-based VMMs. For example, directly assigning a

23 Such file “backends” for the block device of a guest is a common use case for KVM, VirtualBox, Xen and other commonly used VMMs.

PCI device via IOMMU to a guest provides a performance that is close to native speed. The drawback is a severe security issue: more complex PCI devices may use a firmware to drive the logic of that device. It is common that such firmware can be replaced by the host computer. Technically, such replacement is commonly implemented by exposing MMIO registers to the host where one or more of those registers allow the replacement of the firmware. In case of using IOMMUs, the entire memory rage of the device including all MMIO registers is mapped to a guest. Hence, a guest has the ability to replace the firmware of that PCI device. On x86, the boot sequence includes the PCI device enumeration which gives PCI devices full hardware access to the entire system during boot. When a guest now has the ability to replace the firmware, that guest implicitly has the ability to influence the entire boot sequence, too, before the VMM is able to take control of the hardware. One mitigation would be to have a per-device white-list of MMIO registers and other MMIO addresses whose use cannot affect the VMM or the host in a way described with the example of replacing the firmware. In this case, the IOMMU would be configured to allow unintercepted access to MMIO ranges considered safe. Access to other MMIO ranges on the other hand are intercepted by the VMM to prevent security issues.

Assigning hardware resources to guest operating systems for exclusive use may pose a general security risk on x86 systems. More complex hardware resources, such as advanced network interface cards or RAID controllers have the ability to allow an operating system to update their firmware. Now, when assigning such devices to a guest, the guest can exercise the relevant interface to install a new firmware. If that guest is assigned a lower trust level than the VMM, the VMM must ensure that this guest cannot interfere with the VMM operation. On x86, however, the BIOS performs a device-enumeration during the boot cycle of the hardware. During that device enumeration, the BIOS executes the firmware of every device it enumerates which in turn has full hardware access to the entire system. If a less trusted guest is able to change that firmware, it can now install code that is executed with full hardware privileges during boot. To prevent such threats, a VMM may intercept some instructions to a device that is otherw ise directly assigned to a guest. The VMM would in this case intercept the use of any device interfaces, such as MMIO or Programmed Input / Output (PIO) registers, that are known to allow the update of the device firmware. If such partial interception is implemented by the VMM, the base-line assumption is that devices with such type of VMM interception are unsuitable for use in noise sources. Careful studies of particular noise sources and the VMM, however, may show that the intercepts apply to access requests that have no relationship to the unpredictable phenomenon[24]. In such cases, the interception by the VMM is considered to have no impact on the entropy collection from that hardware device.

4.2.4.1.3 Conclusion

Naturally a noise source may be formed by either one or more CPU instructions, one or more hardware resources, or a combination of both. Hence, the analysis of a particular noise source implementation must analyze the unpredictable phenomenon in detail to conclude:

- Which CPU instruction(s) deliver the unpredictable behavior, if any?
- Which hardware resource(s) deliver the unpredictable behavior, if any?

After identifying those sources, the VMM must be analyzed to determine which type of impact the VMM exerts on those sources.

4.2.4.2 VMM Interference With Recording Component

Unlike the unpredictable phenomenon, the recording component in a noise source may be supported by hardware resources or special CPU instructions or it may be implemented completely in software. An example of recording operations is the gathering of a time stamp associated with an unpredictable event.

With a complete software implementation, i.e. without dependencies on particular hardware or CPU support, the VMM is assumed not to interfere with the operation as discussed at the beginning of section 4.2. It may be possible that the VMM intercepts some of the CPU

24 In the given example, the VMM would only intercept update requests to the firmware. Such requests are unlikely to be used as unpredictable phenomenon. Other access requests to the hardware exhibit the relevant unpredictability.

instructions to check their operands. In this case, the execution time of a CPU instruction is extended which does not affect the entropy collection even in the worst case as mentioned several times before and analyzed in section 4.2.3. Therefore, the implementation of all or aspects of the recording component in software implies that the VMM has no impact on the recording operation of the underlying unpredictable noise source.

For a large number of noise sources implemented at least partially in software, specific CPU instructions or hardware resources are needed to perform the recording of events from the unpredictable phenomenon. This is commonly the case where high-resolution time stamps are obtained as often mentioned in chapter 3. Almost all CPUs offer a high-resolution timer to software. That timer is to be accessed with various CPU instructions.

The following instructions are available on the different architectures to obtain a high-resolution time stamp:

- x86 architecture: RDTSC (Read Time Stamp Counter) instruction
- POWER architecture: MFTB (Move From Time Base) instruction
- System Z architecture: STCK/STCKE (Store Clock/Store Clock Extended) instruction
- ARM architecture: Access to CP15 CNTVCT[25]. According to ARM Cortex A7 CP15 register set, this register is defined as CRn=0, Op1=1, CRm=c14.

In case the recording component of a particular noise source is identified to explicitly use either one or more CPU instructions and/or one or more hardware resources the entire analysis and all conclusions drawn in section 4.2.4.1 for the unpredictable phenomenon applies to the recording component as well.

4.2.4.3 VMM Interference With Digitization Component

The digitization component transforms the recorded data into a binary data set that can be interpreted by later post-processing logic. Examples of such digitization logic in software are:

- Interpretation of the data from the recording component as a bit stream.
- Applying a deterministic operation on the data from the recording component to obtain one or more bits for post-processing.
- Changing the data from the recording component by entangling it with other data, such as XORing or concatenating a time stamp with additional data.

Compared to the recording component, for this component it is even more likely that its logic is entirely implemented in software where the VMM would only interfere with the execution time and speed of the logic as already analyzed in section 4.2.4.2. Again, the reference to section 4.2.3 explains that such impact on the execution time has no impact on the digitization component or on the entropy processed with it. The digitization logic is provided with an algorithm to transform data where the algorithm is not dependent on the duration how long that algorithm takes to obtain the result.

But again just like with the recording component, if the digitization component's operation depends on dedicated CPU instructions for its operation or on dedicated hardware resources, the discussion from section 4.2.4.1 applies to the digitization component, too.

4.2.4.4 VMM Interference With Communication Between Components

The different components forming the noise source depicted in figure 1 must also communicate with each other as noted with the arrows in that figure. Therefore, these communication links must not be forgotten in the analysis of the noise source. In software, such communication links are function invocations where function parameters transport data. It may also be conceivable to have some form of asynchronous inter-process communication to transport data.

If these communication links are implemented with software support, the same questions raised in the preceding sections must be raised again:

25 The Linux kernel code implementing the access to CP15 is found in the function arch_counter_get_cntvct.

- Are the communication links implemented purely in software without requiring any specific CPU or hardware support? If yes, the VMM again has no impact on the communication other than the execution time impact discussed in section 4.2.3. As long as the communication functionality between components does not rely on its execution speed for conveying entropic data, section 4.2.3 implies that the VMM timing impact does not have an influence of the operation of the communication functionality. The authors of this study are unaware of communication mechanisms that rely on a precise timing for their operation.
- If parts or the entire communication link implemented in hardware and yet controlled by software, the VMM can interfere. The interference discussion given in section 4.2.4.1 applies therefore to its fullest extent to the communication link implementation, too.

4.3 VMM Impact on Particular Noise Sources

After identifying how a VMM impacts the operation of a noise source in general, particular noise source implementations discussed already in section 3.3 will be assessed. The following subsections enumerate those particular noise source implementations and discuss where a VMM can interfere. To support the discussion, the theoretical analysis about where VMMs can interfere with the noise source is supplemented with an analysis how the following VMMs impact the respective noise source:

- KVM / QEMU when used with the libvirt management framework as provided with Ubuntu 15.10 and Linux kernel 4.2.0-25
- Oracle VirtualBox 5.0.14
- Microsoft Hyper-V as provided with Windows Server 2012 R2
- VMWare ESXi 6.0

For the assessment of the closed source VMMs of Microsoft Hyper-V and VMWare ESXi only limited information provided with the guidance is available. Therefore, conclusions and resulting suggestions for appropriate configurations may be limited or based on assumptions. If this is the case for a particular statement, it is marked appropriately.

With the analysis of the following subsections, questions for the different noise sources are phrased which must be answered to understand whether a VMM has an impact on the noise source.

All the following subsections assume that the reader is already familiar with the discussion in section 3.3.

4.3.1 Linux Random Number Generator

The presentation of the LRNG architecture in section 3.3.1 indicates that it rests on the following noise sources:

- Block devices
- Human Interface Devices
- Interrupts

As discussed, the additional source of add_device_randomness is implemented such that it is not assumed it delivers any entropy. Therefore, this source can be disregarded from the following assessment.

The source of add_hwgenerator_randomness is an in-kernel version of the rngd daemon logic, which is able to obtain data from /dev/hw_random and inject it into the input_pool of the LRNG via an IOCTL on /dev/random. The goal is to use hardware random number generators as noise sources to feed the LRNG. The assessment whether a VMM has an impact on such hardware random number generators must be performed for each of those hardware devices independently. Such assessment, however, does not apply to the software behind the LRNG interface of add_hwgenerator_randomness. Thus, this interface, which serves as a noise source for the LRNG, does not need to be assessed here.

4.3.1.1 Block Device Noise Source

The LRNG with the block device entropy collection rests on the assumption that time variances of disk accesses are based on physical phenomena, like the position of disk read heads or spinning speed of the disk. That means, every time the Linux kernel issues an I/O operation, the kernel assumes that these physical phenomena are triggered. If that phenomenon is not present such as for SSDs, the entropy estimator of the LRNG is assumed to ensure that this I/O operation is awarded zero bits of entropy. In modern Linux kernels, all block devices that are identified for not using spinning disks, including Device Mapper targets are excluded from the noise source collection.

As the kernel feeds every block device I/O operation as entropy source into LRNG entropy collection, the kernel implicitly assumes that the physical phenomenon is present for every access.

When considering figure 5 to identify where the VMM can intervene, the following guiding question can be applied: Where is the hardware / software boundary in that picture? Exactly at this boundary a VMM can interfere. With figure 5 the following components are to be considered:

- The unpredictable phenomena of a block device mentioned above are completely confined to the block device and therefore unaffected by a VMM. A VMM cannot alter the operation of a hard disk with its spinning platters.
- The digitization and conditioning is implemented in software using standard computing instructions. This includes the implementation of the used SHA-1 algorithm, where the C implementation available in the kernel is used. Such standard computing operation is assumed to be unaffected by the VMM operation as discussed in section 4.2. As illustrated in this section, a VMM operation may only affect the duration of the processing, but not its result.
- The recording operation implemented with the add_disk_randomness, however, can be interfered with since it is the software that takes its input from the hardware, i.e. the block devices. The VMM interference affects the guest Linux kernel's ability to access the block device hardware. In addition, add_timer_randomness uses two time stamps for the LRNG processing:
  - the high-resolution time stamp mentioned in section 4.2.4.2, and
  - the coarse time stamp provided with the Linux kernel jiffies variable. Please note that besides mixing the Jiffies into the entropy pool, the LRNG uses the jiffies to implement the heuristic entropy calculation. The first, second and third derivation of the jiffies time is used to calculate the entropy estimate the Linux kernel assumes for the given event.

  The Linux kernel jiffies variable is derived from the time stamp delivered by the Linux kernel's clocksource framework. The Linux kernel implements multiple clocksources, but only one source is active at one point in time. The available clocksources are listed in the file /sys/devices/system/clocksource/clocksource0/available_clocksource whereas the current clocksource is specified in /sys/devices/system/clocksource/clocksource0/current_clocksource. Per default, RDTSC is used as a clocksource on x86 systems. By writing one entry from the available_clocksource file into the current_clocksource file, an administrator can change the clocksource.

The discussion shows that the LRNG block device noise source only applies to block devices with spinning disks. As discussed at various places above, a VMM always can intercept guest block device I/O operations and “translate” them into operations that are not related to block devices or use block devices that do not have spinning platters. For example, a VMM may perform network requests to satisfy the guest block I/O operation. Or the VMM may serve the requested data from a ramdisk. All cases where the VMM translates the guest block I/O operation into a request that is not covered by a block device with spinning disks, the block device visible in the guest must be considered unsuitable for noise source collection. On the other hand, if the VMM uses a disk with spinning disks to serve the guest block I/O requests, the following paragraphs apply.

The VMM may intercept a disk access issued by the guest Linux kernel. Thus, when the guest Linux kernel issues an I/O operation and thus feeds that operation to the LRNG, the VMM may intervene such that there is no real disk access. The intervention is done using the VMM's buffer cache. If a page that the guest Linux kernel wants to read from the physical disk is present in the VMM's buffer cache, the VMM prevents the disk access and just performs a memory read. Thus, the I/O access operation of the guest does not issue a real disk operation. This means that the I/O access operation information given to the guest LRNG is not based on the above mentioned physical phenomenon.

The following list of steps may help understand the problem:

1. Some disk access is triggered.
2. guest kernel block layer checks its buffer cache. If buffer cache can fulfill access, the kernel returns the data – in this case, the guest LRNG block device noise source is not triggered.
3. Otherwise, the block layer decided that disk is to be accessed which triggers an I/O operation. The completion time of the I/O operation depending on the following steps is fed into the LRNG.
4. VMM block layer checks its buffer cache. If buffer cache can fulfill access, the kernel returns the data to the guest.
5. Otherwise the VMM block layer decided that disk is to be accessed which triggers an I/O operation causing the disk to spin.

The first three steps are performed by the guest OS whereas the latter two steps are performed by the VMM.

Note, the check and operation of the VMM in step 4 is the interception of the block device I/O operation of the guest Linux kernel. This interception now implies that the block device is not used to complete the guest Linux kernel's operation.

On the other hand, step 5 can also be subject to a VMM intervention: The VMM may store the data for the guest on a backing store device that is not using spinning platters. For example, the VMM may store the data on a solid state disk (SDD) that looks like a hard disk with spinning platters to the guest. Furthermore, the VMM may even store the data in a ramdisk which implies that I/O operations remain pure memory operations.

With these findings, the following questions must be raised to assess whether a particular VMM implementation interferes with the block device noise source of the LRNG:

- **Does the VMM directly assign all block devices used by the guest to that guest?**

  If yes, then no interference from the VMM is applicable to the LRNG access to block devices.

  If no for at least one block device, then the following questions apply for those block devices:

  - **Does the VMM implement a form of buffer cache which is used to satisfy block device I/O operations from the guest?**
  - **Does the VMM uses a backing store for the block device I/O other than a hard disk with spinning platters?**

  Only if both questions are answered with “no”, then the VMM is expected to not interfere with the LRNG access to block devices.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the LRNG.

If yes, then the VMM interferes with the high-resolution time stamp gathering of the LRNG and thus affects the noise source operation.

- **Does the VMM intercept the hardware access to the guest Linux kernel's current clocksource specified in /sys/devices/system/clocksource/clocksource0/current_clocksource?**

  If no, then the VMM is expected to not interfere with the clocksource time gathering of the LRNG.

  If yes, then the VMM interferes with the clocksource time gathering of the LRNG and thus affects the noise source operation.

Only if all answers confirm that the LRNG block device noise source operation is not affected, the VMM implementation can be relied upon to not affect the noise source.

After identifying the mechanisms the LRNG block device noise source rests on, they need to be assessed how they are affected by different VMM implementations.

4.3.1.1.1 LRNG Block Device Noise Source – Detour

Albeit the following behavior is not related to the use of VMMs, it is significant to obtain a full picture of the block device noise source behavior.

The LRNG block device entropy collection callback of `add_disk_randomness` is only triggered if the underlying block device contains the flag `QUEUE_FLAG_ADD_RANDOM`. This flag is cleared (i.e. removed) when the Linux kernel detects:

- Use of SCSI, SATA or IDE block devices without spinning disks (such as SSDs as well as the virtio block device driver),
- Network block devices,
- Block devices known to be not backed by rotational disks, such as memory-based devices, or
- Use of the Device Mapper to manage the block devices.

A user may obtain the information whether a block device contributes to the LRNG entropy by reading the file `/sys/devices/virtual/block/<DEVICE>/queue/add_random` where <DEVICE> is to be replaced with the device name of the block device to be assessed.

The removal of the Device Mapper devices from the devices delivering entropy is due to preventing double accounting of a disk access. If a physical disk is used to store data which provides entropy to the LRNG, it will also invoke the LRNG if that physical disk is accessed via the Device Mapper. If that Device Mapper “virtual” disk accessed by user space would also credit entropy for one access attempt, this access attempt would be credited twice for entropy, because the call traverses the Linux block layer twice: once for the Device Mapper and once for the physical disk access. Hence, the Device Mapper targets are all excluded from entropy collection.

4.3.1.1.2 KVM/QEMU

KVM allows a block device to be directly assigned to a guest. In this case, KVM does not interfere with the LRNG block device noise source. Examples of such direct assignments are the following options supplied with the QEMU application – see the qemu(1) man page for details:

- IDE drive:

  ```
  -drive if=ide,index=1
  ```

- SCSI disk with unit ID 6 on the bus #0:

  ```
  -drive file=file,if=scsi,bus=0,unit=6
  ```

However, it is common to use the following mechanism offered by KVM, which will have an effect on the LRNG block device noise source: QEMU exports a virtio, SCSI or IDE device to the guest, but uses either:

- a file in the VMM file system as backing store,

- an iSCSI device,
- a network block device
- an SSH remote disk

With that given list, the reader can already determine whether KVM accesses a disk with spinning platters: this may only be given for a file-based backing store unless SSDs are used. All other disk devices are based on network accesses, which have a different access pattern than spinning platters. Some level of entropy may be provided by such devices, but it is currently unclear how much and whether the LRNG entropy heuristic estimates its entropy correctly. Any testing for such devices must correlate the guest I/O access requests with the externally visible operations to give an estimate of the available entropy. As mentioned in section 4.3.1.1, such devices must be treated to not deliver any entropy. This is done by defining such disks as QEMU virtio block devices where the guest Linux kernel automatically excludes such disks from the entropy collection process.

When QEMU is configured to mediate disk accesses for guests, the question arises whether the VMM buffer cache is used. Per default, QEMU uses the buffer cache of the host Linux kernel (i.e. the kernel supporting the VMM). QEMU can be configured to disable that buffer cache with the option “cache=none” applied to the “-drive” command.

QEMU or the VMM's Linux kernel do not intercept the RDTSC, MFTB, STCK/STCKE instructions. Access to the ARM CP15 is intercepted by KVM. However, KVM implements an alteration or specific operations for a subset of the CP15 register set. The CP15 registers that are subject to alteration are defined in the Linux kernel source code with the tables cp15_regs, and cp15_64_regs. For the CP15 CNTVCT register, the Linux kernel's KVM support does not implement any special handling. Therefore, the CP15 access is trapped but is re-issued to the hardware unchanged by the KVM support. Thus, the Linux kernel only slows down the processing of the CP15 CNTVCT access request, but according to section 4.2.3 this will not affect the entropy collection. Hence, the Linux kernel KVM support can be concluded to not affect the time stamp gathering on ARM CPUs.

On the other hand, for the x86 architecture, KVM implements support for accessing the high-precision event timer (HPET), which can be configured in the guest Linux kernel as clocksource. KVM also allows configuring time shifts with this timer. Hence, KVM will have an effect on the time gathering relevant for the noise source in this case. Also, on x86, the clocksource provided by ACPI is available. Access to ACPI is mediated by KVM which implies that this clocksource is also affected by the KVM operation.

To conclude, only the following use cases ensures that KVM/QEMU will not have an effect on the LRNG block device noise source executing as guest:

- ensuring that the guest Linux kernel uses the applicable CPU instructions of RDTSC, MFTB, STCK/STCKE or CP15 CNTVCT as clocksource, and
- ensuring one of the following block device access types:
  - direct assignment of the block devices to a guest, or
  - using a file backing store on a HDD where the VMM's Linux kernel buffer cache is disabled for the guest by using “cache=none”.

4.3.1.1.3 Oracle VirtualBox

Similarly to KVM/QEMU, VirtualBox uses files that are stored in the VMM file system as backing store that is presented as a block device to guest virtual machines. The guest operating system has access to an emulated hard disk controller of either IDE, SATA (AHCI), SCSI, SAS, or USB mass storage controller as documented in VirtualBox Virtual Storage documentation.

As VirtualBox operates on top of a Linux operating system or other host systems, per default it is affected by the respective operating system's kernel buffer cache operation. According to the VirtualBox Virtual Storage documentation section 5.7, VirtualBox starting with version 3.2 obtained the option to disable the VMM's Linux kernel buffer cache. The buffer cache can be disabled as described in the VirtualBox documentation: '... VirtualBox allows you to configure whether the host I/O cache is used for each I/O controller separately. Either uncheck the "Use

host I/O cache" box in the "Storage" settings for a given virtual storage controller, or use the following VBoxManage command to disable the host I/O cache for a virtual storage controller: `VBoxManage storagectl "VM name" --name <controllername> --hostiocache off'`.

Unlike KVM/QEMU, VirtualBox implements an intercept of RDTSC – as other hardware architectures are not covered by the virtualization support of VirtualBox, other CPU instructions do not need to be considered. The intercept is defined for AMD CPUs in the table g_apszAmdVExitReasons and for Intel CPUs in the table g_apszVTxExitReasons; both are defined in the source code file VMM/VMMR3/HM.cpp. The handler for the RDTSC instruction implements an emulation of that instruction.

VirtualBox implements a HPET emulation as provided with the source code of Devices/PC/DevHPET.cpp. In addition, VirtualBox implements an intercept of the Intel i8254 PIT support, which includes the HPET timer hardware. Thus, it must be concluded that VirtualBox interferes with the HPET operation, which in turn affects the LRNG block device operation. Also in VirtualBox, access to ACPI is mediated by the VMM, which implies that this clocksource is affected by the VMM operation. With such impact on the timer, testing must be conducted to measure the extent of the impact.

As a conclusion, there is no configuration available that ensures that VirtualBox does not interfere with the LRNG block device noise source executing as part of a guest. To limit the effect, at least the following options can be set:

- ensure that the backing store for the block device rests on a HDD, and
- ensure that the buffer cache of the VMM's Linux kernel is disabled.

4.3.1.1.4 Microsoft Hyper-V

The assessment of how Hyper-V handles block devices for virtual machines rests completely on the review of the Microsoft UI and the use of Hyper-V. Therefore, the following statements must either be confirmed by testing or by having the developer answering the aforementioned questions. Without access to the source code, those questions cannot be conclusively answered.

The common use case for Hyper-V is to maintain a file-based backend for the guest block devices. Hyper-V emulates either a SCSI or an IDE controller to allow the guest to access those block devices. In addition, Hyper-V allows a direct assignment of hard disks to virtual machines.

Albeit it cannot be verified in the guidance documentation, the authors assume that file-based backends are subject to the buffer cache of the VMM implemented by the Windows Server. That Windows Server hosts the virtual machine worker processes, which implement the virtualization logic for the SCSI and the IDE controller as well as the translation to the backend file accesses.

On the other hand, when assigning a hard disk to a guest, the authors assume that no buffer cache implemented in VMM is present to interfere with the guest's block device accesses. If this were to be confirmed, such a configuration of assigning a full hard disk would be considered to not interfere with the noise source operation of the LRNG block device.

However, the question regarding the intercepting of the high-resolution time stamp gathering cannot be answered using the guidance or by observing the Hyper-V configuration frontend. This means that no statement about whether the VMM impacts the guest operation can be made.

4.3.1.1.5 VMWare ESXi

Just like with Hyper-V, the discussion in this section is based on using the administrative UI for managing ESXi supported by guidance documents. Due to VMWare ESXi being proprietary, source code access was unavailable. To conclude whether the statements are accurate, either the developer must answer the given questions or testing must be conducted. Testing of the LRNG on VMWare ESXi is performed in chapter 5.

ESXi uses files as backends for virtual machines stored in the VMFS partition assigned to ESXi. These files are provided to guests as either IDE or SCSI hard disks. In addition, ESXi supports a para-virtualized interface. Neither the documentation nor the administrative UI could

conclusively answer the question whether a buffer cache is applied to mediate guest access to the backend files by the VMM.

In addition, the question of the high-resolution time stamp interference cannot be answered either with the available information.

Therefore, based on the available documentation and with the administrative UI, the question whether ESXi interferes with the LRNG block device noise source cannot be answered.

4.3.1.2 Human Interface Device Noise Source

The LRNG HID noise source uses the uncertainty pertaining to the occurrence of an event from a human interface device like a mouse or keyboard. In a lot of use cases for virtual machines, guest virtual machines are “headless”, i.e. operate without keyboard and mouse (and without a monitor interface). In such configurations, the LRNG HID noise source is unavailable.

Other configurations for VMMs may establish emulated keyboards and mice for guest operating systems. Those emulated devices are not connected to real keyboards and mice, but to VNC or RDP server protocols. Such server protocols allow attaching of a remote or local network client that transports keyboard events or mouse events to the guest operating system. The VNC and RDP protocols typically do not alter the HID event type, i.e. the pressed key number is transported via VNC/RDP unaltered at the time the user pressed the key. The VMM commonly forwards the received key number to the guest for interpretation as soon as possible. Similarly, any mouse movement forwards the coordinates either unchanged or processed with a scaling factor. But again, the direction of the movement or the timing of the movement is forwarded to the guest.

When the VNC/RDP protocol is used locally, i.e. a user access the console of the hardware that hosts a virtual machine, the protocol is assumed to be communicated via localhost. Commonly VMMs do not allow guests to access such network communication over localhost. This means that neither local nor remote unauthorized entities can sniff or interfere with that data stream. In addition, the time delay added by the VMM operation does not have an impact on the entropy the LRNG assumes with such HID events as discussed in section 4.2.3. In such a case, the HID noise source of the LRNG may be considered an appropriate noise source. The author of this study, however, reminds on the common use case of VMMs: they act as server type systems in Cloud environments that are configured headless or whose console is used in rare cases only. Hence the more common case is discussed as follows.

Albeit the events are triggered by a human just like on a local console, and the VMM only forwards the HID events to the guest, the following should be considered when assessing the impact of that use case to the noise source: The network protocols (VNC, RDP) that are commonly used when transporting HID events issue one network packet per HID event. For example, one network packet per pressed and released key or one packet per mouse movement is sent. According to [LRNG], the HID noise source derives the majority of entropy from the time stamp of the occurrence of an event. Now, if an adversary is able to detect such HID events with a high precision, then the quality of the HID noise source must be considered degraded, i.e. its entropy must be considered reduced. In case of network protocols where each event is sent over the network in individual packets – e.g. as described in RFC6143 chapter 4 –, the adversary has the ability to detect such network packets. If the adversary is close to the system hosting the guest, he can determine the timing of events with good precision. This applies even if the VNC/RDP links are protected by cryptographic mechanisms like TLS (or SSH as used for KVM/QEMU)[26]. As for noise sources a worst case assessment is warranted, any HID events transmitted over the network must be considered to have degraded entropy. This in turn implies that although the VMM operation per-se does not alter the transmitted HID events, the usage of the network as transmission channel renders these events unsuitable[27] for the LRNG HID noise source.

26 An analysis of the conceptual problem that each key stroke turns into a network packet is provided with the SSH Traffic Analysis.

27 Testing may show that an adversary cannot predict the time stamp of the event used by the LRNG with a precision of less than 11 bits (the cap of the LRNG entropy estimation heuristic for one event). But that testing must be performed. The author performed a similar test by snooping the X11 input events – similar to the command “xinput test” –

It is technically conceivable that a keyboard and mouse attached to a host system are directly assigned to a guest. For example, a USB interface to which a keyboard or a mouse is connected to is directly assigned to a virtual machine. In this case, the VMM does not intercept the flow of events between the HID and the guest allowing that these devices can be used to support the LRNG.

The LRNG requires the high-resolution time stamp as well as the jiffies time stamp discussed in section 4.3.1.1 to process the HID data.

Again, based on this analysis, the following questions must be answered for a VMM implementation in order to identify whether that implementation affects the LRNG HID noise source:

- **Does the VMM directly assign all HID used by the guest to that guest?**

  If yes, then no interference from the VMM is applicable for the LRNG HID noise source.

  If no, then the VMM affects the LRNG HID noise source.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the LRNG.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the LRNG and thus affects the LRNG HID noise source operation.

- **Does the VMM intercept the hardware access to the guest Linux kernel's current clocksource specified in /sys/devices/system/clocksource/clocksource0/current_clocksource?**

  If no, then the VMM is expected to not interfere with the clocksource time gathering of the LRNG.

  If yes, then the VMM interferes with the clocksource time gathering of the LRNG and thus affects the LRNG HID noise source operation.

When answering all questions such that the VMM will not interfere with the different aspects of the LRNG HID noise source operation, then the VMM can be considered to not affect the entire LRNG HID noise source.

With these questions in mind, the following subsections will try to give an answer for the analyzed VMM implementations.

4.3.1.2.1 KVM/QEMU

As mentioned before, the common use case for virtual machines is to use a VNC server or a headless system – uses cases which are common for KVM/QEMU as well. When such a virtual machine is accessed remotely, the LRNG HID noise source must be considered not trustworthy.

However, when accessing the VNC server locally using the viewer offered by the Virt-Manager tool that accesses the KVM/QEMU libvirt managment framework, i.e. the virtual machine is executed on the same Linux host system and hardware as the VNC viewer, the connection is established via a Unix domain socket. That Unix domain socket is access-protected such that no unprivileged user can access it. Therefore, no adversary can sniff the communication sent over it. Hence, when using Virt-Manager[28] and this VNC viewer resides locally to the accessed virtual machine, the communication link is protected such that the LRNG HID noise source can be considered not interfered with.

---

and compared the time stamps with the LRNG time stamps. The testing showed that the time stamp measurements vary by 500,000,000 ticks on a current hardware, far more than the mentioned 11 bits. Thus, such snooping does not seem to provide an attack vector. Yet, this testing has to be performed for network traffic as well.

28 Other VNC clients may be usable too, but specific configurations must be applied to those tools whereas Virt-Manageruses this approach automatically .

In addition, QEMU allows other possibilities on how HID are connected to the guest:

- When not using any specific options around HID, QEMU per default uses the SDL graphics library to render the console. The SDL library implies that the QEMU console is rendered as “just another” X11 window. In this case, the X11 window obtains the keyboard and mouse events just like any other X11 window. Therefore, when this SDL windows has the X11 input focus, VMM translates the HID events from the host's X11 representation to the device driver, adding a constant processing time. As this addition of a constant time is immaterial to the entropy collection as discussed in section 4.2.3, the use of the SDL interface must be considered to not interfere with the HID event processing in the guest Linux kernel.
- QEMU allows the assignment of USB interfaces to guest operating systems. When HID devices are connected to USB and those USB ports are assigned to a guest, this guest receives HID events that are not interfered with by the VMM.

The Linux kernel processing of the HID noise source also requires the same time stamps as needed for processing the block device events. Thus, the discussion about the time stamp interference by KVM/QEMU given in section 4.3.1.1.2 applies here, too.

Thus, the following settings can be used with KVM/QEMU to ensure that the HID events are not interfered with by the VMM:

- ensuring that the guest Linux kernel uses the applicable CPU instructions of RDTSC, MFTB, STCK/STCKE or CP15 CNTVCT as clocksource, and
- ensure that the HID events of locally attached devices are received by the guest using one of the following configurations:
  - using SDL to draw the guest console and obtain HID events from the X11 layer, or
  - directly assigning the USB devices to which HID are connected to.

However, it is clear that for enterprise use cases, a virtual machine is neither started with SDL support nor with directly assigned USB HID, nor with a console where the human user sits right in front of the hardware hosting the virtual machine. Therefore, for common use cases, KVM/QEMU will interfere with the HID noise source. As headless virtual machines are found often, the HID noise source will rarely be triggered in a guest Linux kernel.

4.3.1.2.2 Oracle VirtualBox

Similarly to KVM/QEMU, VirtualBox supports the connection to the console via RDP. In addition, VirtualBox can be configured to provide a headless guest system. Again, in case of accessing the console remotely, the LRNG HID noise source is considered to be unusable.

Similarly to the KVM/QEMU approach, when accessing the console of a virtual machine executing on the same Linux host system as the VirtualBox viewer, the connection between the viewer and the virtual machine's RDP server is established via a Unix domain socket that is accessible only by the host Linux user executing the virtual machine. Therefore, this Unix domain socket is protected against access from unauthorized users. This again implies that such unauthorized users cannot sniff the communication and derive information about HID events communicated over that link. Thus, for such local access of the virtual machine's console, the Linux HID noise source must be considered unaffected.

As documented for VirtualBox, an SDL interface is available, but not considered applicable for regular use. Therefore, this is configuration is not considered further.

The LRNG HID noise source also requires the hardware to provide time stamps. The interference with the time stamping by VirtualBox is already discussed in section 4.3.1.1.3 and applicable here as well.

With those considerations, no configuration of virtual machines in VirtualBox can be derived which will not interfere with the Linux HID noise source.

4.3.1.2.3 Microsoft Hyper-V

Microsoft Hyper-V can be remotely administered as documented with the Hyper-V guidance. With such remote administration, the console of a virtual machine can be accessed as well.

Therefore, the consideration around the network protocols and how they allow adversaries to detect HID events must be considered applicable to the Hyper-V remote management as well. This conclusion is supported by the assumption that the virtual machine console is made available via some form of RDP communication, which is logically equivalent to VNC.

As already mentioned in section 4.3.1.1.4, information about the handling of the high-resolution time stamp operation cannot be obtained.

With the mentioned constraints, a safe assumption is that Microsoft Hyper-V interferes with the operation of the LRNG HID noise source.

4.3.1.2.4 VMWare ESXi

VMWare employs a remote management console executing on a Windows system to access the ESXi virtual machine monitor. Using this remote management system, access to the console of a guest can be obtained.

As it is unknown to the authors which precise protocol is used to display the console, it must be assumed to be similar to VNC or RDP. Hence, the issues around HID events transported over such protocols mentioned above should be assumed to be present for ESXi as well.

Again, with the absence of information around the high-resolution time stamp interference and the remote console access, a safe assumption is that VMWare ESXi interferes with the LRNG HID noise source.

4.3.1.3 Interrupt Noise Source

Each interrupt received by the Linux kernel is processed by the LRNG[29]. Interrupts are naturally generated by hardware, but in virtual environments, interrupts are also generated by the VMM. Why are interrupts generated by the VMM? The answer to that question is clear when considering how the devices seen by a guest are technically implemented: most if not all of those devices a guest “sees” are emulated devices. For those emulated devices, interrupts must be “injected” into the guest operating system by the VMM to emulate the device operation.

It is of course possible that para-virtualized devices are in use which use a special communication channel between guest and VMM. That communication channel may not follow the principles of hardware communication channels and would allow exchange of data without the need of any interrupts. Such communication channels are seen with the virtio framework used by KVM or the VMBus implementation offered by Hyper-V. Both implement a form of a ring buffer in a memory segment shared between the guest and the VMM that serves as the communication channel. The guest may follow two strategies to wait for data from the VMM:

1. Poll the communication channel in regular intervals to check for the availability of data.
2. Receive an out-of-band notification such as an interrupt or exception from the VMM when the VMM added data to the communication channels.

Albeit option 1 is possible, it would waste CPU cycles and is therefore commonly not seen. Option 2 is in use for the para-virtualized devices offered by the VMMs discussed in this document. When the guest implements para-virtualized device drivers, they are only used as a source of entropy if the entire guest to VMM communication mechanism uses the standard Linux interrupt handling logic.

With this initial consideration alone, the reader can conclude that the VMM operation interferes with the LRNG interrupt noise source.

But unlike for the other noise sources where no additional theoretical considerations were identified, additional thoughts can be applied to assess whether the interference for this particular noise source will diminish the entropy the LRNG obtains from the interrupts. The following discussion only applies if the VMM and its potentially existing para-virtualized device drivers allow interrupts to be processed by the LRNG's interrupt handling code – i.e. it is not applicable for Hyper-V para-virtualized devices.

29 The Linux kernel uses the interrupts as noise source starting with Linux kernel version 3.6.

When the LRNG receives an interrupt, the interrupt number, the CPU instruction pointer and the high-resolution time stamp are used to feed into a per-CPU fast_pool. After 64 received interrupts or after 1 second (whatever comes later), a snapshot is taken from the fast_pool and injected into the input_pool; then, the entropy estimator is increased by one. This implies that the LRNG assumes that at least 64 interrupts together deliver one bit of entropy. The maintenance of the fast_pool is also intended to break any correlation with the HID and block device noise sources considering that HID and block device events are triggered by interrupts received by the Linux kernel. Thus, the approach of maintaining the fast_pool together with the small amount of entropy awarded to interrupts allow the conclusion that this mechanism is assumed to break the mentioned correlation without over-estimating the entropy derived from them.

The VMM now tries to mimic the behavior of the hardware for emulated devices. That means that albeit the processing speed may be different, the occurrence of interrupts is similar to that of the real corresponding hardware – the assumption of a benignly acting VMM is the basis for such a conclusion. As mentioned in section 4.2.3, the execution speed variations introduced by the VMM are immaterial to the entropy, even in the worst case.

This finding is supplemented by interrupts that may be received from directly assigned hardware by the guest. Technically, the interrupt of such directly assigned guests are sent to the VMM by the CPU. But the VMM checks that the interrupt is intended for hardware assigned to a guest and thus forwards that interrupt unaltered to the guest for processing. Again, section 4.2.3 comes to the rescue indicating that the added processing time from the VMM intercept is immaterial to the entropy recorded by the LRNG from the interrupt noise source.

Now, considering the emulated devices act similarly to the corresponding real hardware, the unaltered forwarding of interrupts by directly assigned devices and the architecture of the LRNG interrupt noise source processing together allows the conclusion that no matter how the VMM interferes with the interrupts, it has no effect on the entropy derived out of them by the LRNG. This finding is supplemented with testing in section 5.5 which provides an adequate quantitative proof of this statement.

With the given discussion, it is clear that the VMM impact on the occurrence of interrupts is immaterial to the noise source operation. However, the LRNG still uses a high-resolution time stamp which provides most of the entropy for the interrupt event. Therefore, the following questions are applicable to VMMs:

- **Does the VMM offer para-virtualized devices?**

  If no, para-virtualized device drivers are used, the VMM is expected to not interfere with the LRNG's interrupt collection.

  If yes, do the device drivers in the Linux guest use the standard Linux interrupt handling logic?

  If yes, the VMM is expected to not interfere with the LRNG's interrupt collection.

  If no – i.e. device drivers bypass the standard Linux interrupt processing, the VMM interferes with the LRNG's interrupt collection.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the LRNG.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the LRNG and thus affects the LRNG interrupt noise source operation.

Only if all questions are answered such that the VMM does not interfere with the LRNG interrupt noise source, the VMM can be considered to not affect the interrupt noise source operation.

Again, the following sections discuss how the different VMMs handle that question.

4.3.1.3.1 KVM/QEMU

The discussion about the time stamp interference by KVM/QEMU given in section 4.3.1.1.2 applies here as well. This discussion concludes that KVM/QEMU does not interfere with the high resolution time stamps required for the LRNG interrupt noise source.

KVM employs virtio to implement para-virtualized device drivers. The virtio para-virtualized device drivers use the standard Linux interrupt handling logic as outlined in [VIRTIO].

4.3.1.3.2 Oracle VirtualBox

The interference with the time stamp gathering by VirtualBox is already discussed in section 4.3.1.1.3 and also applicable here. This section identifies that VirtualBox interferes with the high-resolution time stamp gathering.

VirtualBox uses virtio for para-virtualized device driver support. Therefore, the standard Linux interrupt handling logic is used as discussed in [VIRTIO].

4.3.1.3.3 Microsoft Hyper-V

The description in section 4.3.1.1.4 mentions that information about the handling of the high-resolution time stamp operation cannot be obtained. Therefore, no statement about whether the VMM impacts the guest operation can be made.

The para-virtualized device drivers offered with the Linux Integration Services[30] for Hyper-V use VMBus.

The VMBus communication channel is the building block upon which the Hyper-V para-virtualized device drivers rest on. The VMBus implementation registers a special interrupt vector 0xf3 in the Linux interrupt vector table. If the CPU now delivers an interrupt with this vector number, the VMBus implementation is invoked. All Hyper-V para-virtualized device drivers are directly invoked from the VMBus implementation depending on the information the VMBus finds in the ring buffer shared with the VMM. Therefore, for all Hyper-V para-virtualized devices, the standard Linux interrupt handling code is completely bypassed which implies that the LRNG interrupt callback handling function of `add_interrupt_randomness` is never triggered for these devices. Note that the Linux Integration Services provide para-virtualized device drivers for the following devices:

- Network device
- Memory ballooning driver
- Framebuffer device
- HID devices (mouse, keyboard)
- Block device driver
- File copy device driver
- Hyper-V utility driver

Thus, when using those device drivers – which is expected as they provide significant performance boosts compared to emulated devices – the LRNG interrupt processing code is not invoked. Hence, for those devices, the LRNG interrupt noise source will not collect entropy.

4.3.1.3.4 VMWare ESXi

As described in section 4.3.1.1.5, the available documentation does not allow to identify whether the high-resolution time stamp is interfered with. Therefore, no conclusion about whether the VMM impacts the LRNG interrupt noise source can be given.

VMWare allows the use of para-virtualized device drivers. The Linux kernel tree provides several of such VMWare para-virtualized device drivers. All of these device drivers register with the standard Linux kernel device driver framework and thus use the standard Linux kernel interrupt processing logic.

30 The para-virtualized device drivers are part of current Linux kernel source trees where no additional drivers need to be installed.

4.3.2 Intel RDRAND and RDSEED

The entire implementation of RDRAND and RDSEED is encapsulated within the CPU. This fact alone implies that a VMM cannot interfere with the RDRAND and RDSEED operation.

However, the RDRAND and RDSEED logic is intended for software executing on top of the CPU. Both are CPU instructions accessible by software. Here, a VMM can surely interfere with a guest operating system's attempt to access these CPU instructions. There are the following two ways how a VMM may interfere with a guest operating system access request to these noise sources:

- In general, VMMs have the ability to trap every CPU instruction. When such trapping is enabled, a complex instruction decoding and handling must be implemented within the VMM to cover the x86 instruction set. During such decoding, a VMM may decide to handle the RDRAND or RDSEED instruction. In practice, it is now rare that such complete x86 instruction set trapping is implemented. Please note that such trapping and CPU instruction handling requires instruction completion engines which are extensive and require much knowledge about the CPU behavior.
- Unfortunately, Intel decided to design RDRAND and RDSEED in such a way that they may be configured to cause so-called “VM-exits”. The Intel CPU specification [INTELPROC], volume 3, appendix C, table C-1 references the VM-exits that a VMM can configure. This means that a VMM can decide whether a guest-issued RDRAND or RDSEED instruction will be trapped by the CPU and allow the VMM to intercept it. Such architected trap possibilities require very little code in the VMM. The author of this study implemented an RDRAND/RDSEED trap in KVM where these instructions return zero with as little as 10 lines of C code.

  Note, the AMD x86 CPU implements RDRAND as well. As this instruction is not listed in the AMD processor manual [AMD64VOL2] section 15.9, table 15-7, the AMD processor will not cause VM-exit when this instruction is issued by the guest.

With these considerations, a developer of a particular VMM should answer the following question to assess whether that VMM interferes with the RDRAND or RDSEED operation:

- **Does the VMM intercept all instructions of a guest for complete system emulation and implements processing of RDRAND and RDSEED?**

  If yes, then the VMM is already identified to interfere with these instructions and thus with guest accesses of these noise sources.

  If no, the following question should be answered:

- **Does the VMM configure the x86 CPU to cause a VM-exit for the instructions of RDRAND and/or RDSEED?**

  If yes, then the VMM must be closely analyzed how it handles these traps.

If both questions can be answered with no, the VMM is identified to not interfere with the RDRAND and RDSEED CPU instructions and thus with a guest operating system using them as noise sources.

As already exercised with the other noise sources, the following sections iterate through VMM implementations and assess whether they affect RDRAND and RDSEED.

4.3.2.1.1 KVM/QEMU

All KVM versions up to and including Linux the latest kernel version 4.4 do not configure the x86 CPU to cause a VM-exit for RDRAND and RDSEED.

The code configuring the VM-exits for the Intel x86 CPU is found in the Linux kernel code tree with the table kvm_vmx_exit_handlers. For AMD x86 CPUs, this table is provided with svm_exit_handlers.

4.3.2.1.2 Oracle VirtualBox

Unlike KVM/QEMU, VirtualBox does intercept the RDRAND and RDSEED instructions on Intel x86 CPUs. Naturally, on AMD x86 CPUs, VirtualBox cannot and does not intercept the RDRAND instruction.

The x86 CPU trap configuration is defined with the tables g_apszAmdVExitReasons for the AMD x86 CPU and g_apszVTxExitReasons for the Intel x86 CPU in the the source code file of VMM/VMMR3/HM.cpp.

The exit handler for RDRAND, i.e. the function invoked when the guest issued an RDRAND instruction, is implemented with the function hmR0VmxExitRdrand. This function simply disallows the use of RDRAND.

The RDSEED instruction exit handler immediately branches to a function returning an error to the guest. This implies that the VMM disallows the instruction to be used.

4.3.2.1.3 Microsoft Hyper-V

Due to the lack of documentation and source code access, the questions for RDRAND and RDSEED cannot be answered for Hyper-V.

4.3.2.1.4 VMWare ESXi

Based on public VMWare documentation, it is unclear how the RDRAND and RDSEED instructions are processed with VMWare ESXi. According to a VMWare community bulletin, RDRAND “is supported” with the virtual hardware version 9. With another VMWare community bulletin it is described that RDSEED is supported with virtual hardware version 11.

However, it remains unclear whether the VMM intercepts the RDRAND and RDSEED instructions. Due to the unavailability of the source code or appropriate documentation, this issue cannot be solved here.

4.3.3 CPU Execution Time Jitter Random Number Generator

The Jitter RNG uses two noise sources as explained in section 3.3.3:

- timing of memory accesses, which is affected by the CPU caches, and
- timing of the execution duration of a given code fragment.

The operations to access memory[31] as well as the execution duration of a code fragment[32] are considered as standard CPU operations, which are assumed to not be subject to VMM interference other than variations to execution time as discussed in section 4.2. As discussed in section 4.2.3, the timing interference from the VMM due to virtual machine rescheduling has no impact on the entropy, even in the worst case.

Hence, the VMM's interference with the time stamp gathering is the only type of interference a VMM exerts on the Jitter RNG. The Jitter RNG uses the time stamp CPU instructions mentioned in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15). Thus, the following question must be raised for a particular VMM implementation:

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the Jitter RNG.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the Jitter RNG and thus affects the noise source operation.

When the time stamp gathering is intercepted by the VMM, the analysis should focus on how the VMM alters the invocation. The Jitter RNG uses the time stamp to generate a difference to a previously read time stamp. Thus, the impact of the VMM operation on the time delta, i.e. the first derivation of the time stamp, is of interest.

31 The memory access is simply a read and a write of an integer value.
32 The code fragment is implemented as a loop which performs XOR operations.

If the VMM uses the underlying CPU instruction to obtain a value and then simply adds an offset to the value, the offset is eliminated when the first derivation is calculated – the mathematical behavior is described in section 4.2.3.

Also, when the offset is varied by the VMM, e.g. when the VMM tries to hide its execution time, the offset will monotonically increase. In this case the VMM acts like it slows down the guest execution and enlarges the time delta. A slowdown of the execution speed, however, is immaterial to the Jitter RNG entropy gathering as explained in [JENT] section 5.1.1 with the discussion of the frequency scaling impact.

In both cases, however, the VMM does not alter the precision of the time stamp, i.e. that the time stamp has a nanosecond resolution. The precision of the time stamp is the key for the Jitter RNG as it tries to detect tiny variations.

Based on this assessment, the following clarifying questions can be raised if the VMM intercepts the time stamp gathering:

- **When handling the intercept of the time stamp gathering, does the VMM issue the same time stamp operation as triggered by the guest, but adds an offset to that value to disguise the VMM's or other guest's execution time?**

  If yes, the VMM intercept can be considered to have no effect on the Jitter RNG noise source operation after all.

  If the answer is no, e.g. the VMM changes the precision of the time stamp by masking some low bits out or by using a completely different time source, the VMM intercept will have an effect on the Jitter RNG noise source operation.

Thus, if a VMM intercepts the time stamp gathering as just described, it can be concluded to not have an impact.

Just like for the other noise sources, the following subsections discuss the impact of the different VMMs on the Jitter RNG.

4.3.3.1.1 KVM/QEMU

The discussion about the time stamp interference by KVM/QEMU given in section 4.3.1.1.2 applies here as well. There, the assessment concludes that the time stamp CPU instructions are not intercepted. Hence, KVM/QEMU does not interfere with the Jitter RNG.

4.3.3.1.2 Oracle VirtualBox

The interference with the time stamp gathering by VirtualBox is already discussed in section 4.3.1.1.3 and also applicable here. The RDTSC exit handler of hmR0VmxExitRdtsc. implements two methods to handle a RDTSC instruction which are mutually exclusive:

1. Emulation of the TSC with VMM internal mechanisms.
2. Use of the CPU RDTSC (or the RDTSCP instruction, which is equivalent to RDTSC) that is added an offset.

Per default, if no specific configuration is set, the emulation logic is used[33]. When the configuration option of “TSCMode” is set to “RealTSCOffset”, the second option is used. Any other configuration chooses the first option. During runtime, the RealTSCOffset is also set if a virtual machine is instantiated with KVM acceleration[34].

When VirtualBox enforces the second option, the time stamp operation is intercepted by VirtualBox such that the Jitter RNG noise source operation is not affected. When using the first option for the time stamp intercept, the Jitter RNG noise source is impacted.

33 This behavior can be seen in the log file for a started virtual machine when searching for “enmTSCMode” – any mode not equal to 2 implies the use of the emulation logic.

34 This setting can be configured per virtual machine in Settings → System → Acceleration → Paravirtualization Interface.

4.3.3.1.3 Microsoft Hyper-V

With section 4.3.1.1.4 this document already explains that information about the handling of the high-resolution time stamp operation cannot be obtained. The VMM impact on the noise source operation can therefore not be assessed without testing.

4.3.3.1.4 VMWare ESXi

Section 4.3.1.1.5 describes that the available documentation for ESXi does not allow to verify whether the high-resolution time stamp is interfered with. Therefore, a conclusion about whether the VMM impacts the Jitter RNG noise source not be obtained.

4.3.4 Apple Mac OS Noise Source

With the description of the Apple Mac OS and iOS noise source implemented in the XNU kernel given in section 3.3.4, it is readily clear that the only specific hardware mechanism in use is the time stamping of received interrupts. The additional processing of maintaining the entropy pool and generating an output for the caller is standard CPU processing which is not considered to be interfered with by the VMM as discussed in section 4.2.

Considering the nature of VMMs where the VMM provides emulated hardware or mediates access to hardware, the visibility of interrupts in virtual machines are directly affected. Section 4.3.1.3 already discusses the use of interrupts as noise sources. All statements and findings apply to this use case of interrupts as noise source as well. The conclusion drawn in this section is that the VMM impact on the occurrence of interrupts is immaterial to the noise source operation.

The Apple Mac OS and iOS noise source, however, use a second hardware mechanism relevant for maintaining the noise source operation: gathering the time stamp. Thus, the same question raised for other time-based noise sources must be raised here as well:

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the XNU kernel.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the XNU kernel and thus affects the noise source operation.

At the time of writing, the following VMMs are documented to support Apple Mac OS virtualization:

- Parallels Desktop
- VMWare Fusion
- Oracle VirtualBox

Given the listing of VMMs to be assessed in this study in section 4.3, only VirtualBox is assessed in the following.

4.3.4.1.1 Oracle VirtualBox

VirtualBox interferes with the time stamp gathering as already discussed in section 4.3.1.1.3. This section explains that the interference is such that it affects the noise source operation.

4.4 Impact of VMM on Entropy

With the discussion about how a VMM influences the operation of a noise source, the question around how the entropy is affected naturally arises. This question can only be answered for a particular noise source after the effect of the VMM on that noise source is already carefully studied and understood.

Therefore, every time the aforementioned sections identify that a noise source is impacted by a VMM operation or it is unclear whether a VMM impacts a noise source operation, a careful study of the noise source's entropy collection in light of the VMM interference must be conducted.

Such analysis must start with the study of how the VMM interference affects the theoretical foundation of that noise source. In some cases, the entire foundation of the noise source vanished with the execution within a virtual machine. In such case, the noise source can be all but deactivated.

If the theoretical foundation of a noise source is still sound with the VMM interference, the next step is to conduct measurements whether the statistical properties of the noise source change compared to the execution on bare-metal. If the statistical properties are equivalent when operating the noise source within a guest environment and on bare-metal, the noise source can be considered to be unaffected by the VMM and can remain operational. When changes in the statistical results are evident, a careful analysis is required whether the changes in the statistical properties imply that the entropy level changed.

With chapter 5, such complete analysis is performed for the LRNG. This analysis may provide answers for noise sources that are similar in nature to the LRNG noise sources.

5 Linux Random Number Generator Assessment

Using the conclusions derived in the preceding chapters, the Linux Random Number Generator (LRNG) providing the user space interfaces of /dev/random, /dev/urandom, and the `getrandom` system call in addition to the in-kernel API call of `get_random_bytes` is assessed in this chapter.

The goal of this chapter is to show the impact of the VMM operations on the LRNG and its noise sources with quantitative tests. In addition, these tests shall serve as a demonstration about how to test a noise source implementation and how to identify a VMM interference. Furthermore, the discussion provides guidance on VMM configurations that limit the VMM impact on the LRNG.

The discussion of the LRNG in the following assumes that the reader is familiar with the LRNG architecture discussed in [LRNG] or explained in other resources.

In a first step, the general test approach is discussed. This is followed by the actual testing of the different LRNG noise sources on the different VMMs referenced in section 4.3. Each noise source testing is explained, the results are discussed and a conclusion is drawn.

5.1 General Test Approach

To obtain quantitative results from the noise sources and the general LRNG processing, its runtime operation must be supervised. Such supervision can only be conducted by instrumenting the Linux kernel to be able to read out the relevant parameters and values from the kernel-internal memory. Though, such instrumentation shall not or only insignificantly affect the measurements.

Current Linux kernels implement a number of different tracing mechanisms which can be applied during runtime. Currently, the following tracing mechanisms are available:

- SystemTap
- Ftrace
- KGDB
- Manual instrumentation of the source code using `printk`
- `ptrace` system call to analyze the system call invocations

For conducting the tests presented in this chapter, the authors chose SystemTap. SystemTap allows to read out arbitrary variables and memory locations at arbitrary code locations. Furthermore, the creation of test cases is similar to writing a Shell script. It therefore provides a great amount of flexibility at a reasonable programming effort.

Note that, the SystemTap approach has already been successfully used in the analysis of the LRNG as documented in [LRNG].

5.1.1 SystemTap Testing

SystemTap provides an infrastructure in the Linux kernel which allows collection of data from the kernel. Contrary to legacy tools and methods to analyze the Linux kernel, SystemTap does not need a modification of the Linux kernel source code and a resulting recompilation of the kernel. The kernel binaries that are provided by Linux distributions can be analyzed with SystemTap out of the box without any modification of the system. To perform testing with SystemTap, the SystemTap “scripts” are compiled and loaded at runtime. These tests are unloaded once they are complete. There is neither a reboot necessary nor does the system environment need to be changed. The intended work load can directly be executed on the Linux kernel with the SystemTap script active.

In user space, SystemTap provides a command line tool, which compiles the SystemTap script and loads it into the kernel. Furthermore, the user space tool establishes the kernel to user space interface to extract data from the kernel environment.

To understand how SystemTap operates, a small peek behind the curtain is warranted at this point. Though, this document does not provide a complete description of the technical

aspects of SystemTap – the SystemTap project website provides much more details. SystemTap provides a compiler that translates the SystemTap script code into a Linux kernel module C code. After this first translation, the C code is compiled into a loadable Linux kernel module, which is then loaded into the Linux kernel by the SystemTap user space tool. To extract data from kernel space, DebugFS hooks are added to the newly compiled SystemTap kernel module. These hooks create DebugFS files which relay the generated data from kernel space to user space. The user space SystemTap tool picks up the data in such a way that it looks like the tool itself has generated the data.

SystemTap scripts consist of functions or Linux kernel C code line references which are triggered when the instruction pointer hits the referenced Linux kernel code. In addition, a time-based alarm system allows triggering of SystemTap script operations.

The use of SystemTap for LRNG testing has a major drawback, which must always be considered during the analysis of the test result data: SystemTap scripts are not re-entrant. The Linux kernel is a multiprocessor kernel which allows the execution of a particular code path simultaneously on different CPU cores. The LRNG C code is not test-friendly because different aspects of one logical operation – such as the processing of one noise source event – is distributed over multiple functions that operate on multiple data structures. In addition, some of those functions which require close examination, namely the `_mix_pool_bytes` and `credit_entropy_bits` are invoked when events from the noise sources are processed, as well as when callers inquire the LRNG for random data. This means that those interesting functions must be assessed with the invocation context – which is a challenging proposition. Yet, the devised SystemTap scripts and the test approach discussed in the following sections take that issue into consideration and try to minimize resulting measurement errors. In addition, based on experience of testing the LRNG with SystemTap scripts, the authors note that measurement errors are commonly easy to spot as they create outliers in the test result dataset.

5.1.2 SystemTap Prerequisites

For executing SystemTap scripts the following prerequisites must be fulfilled:

- The kernel source code must be accessible by SystemTap to compile kernel modules. Linux distributions offer Linux kernel “development” packages[35]:
  - Ubuntu: linux-headers-generic
  - Red Hat / SUSE: kernel-devel
  - For self-compiled kernels, the source code must be accessible at `/lib/module/$(uname -r)/build`
- The debug symbols of the current kernel must be available. Again, Linux distributions offer packages with these symbols:
  - Ubuntu: linux-image-$(uname -r)-dbgsym
  - Red Hat: kernel-debuginfo
  - SUSE: kernel-default-debuginfo
  - For self-compiled kernels, the kernel configuration option CONFIG_DEBUG_INFO must be set. This option is found with the kernel configuration UI at “Kernel Hacking” → “Compile the kernel with debug info (DEBUG_INFO) [N/y/?]” → “Kernel debugging (DEBUG_KERNEL) [Y/n/?]”.

  Please note that the debug symbols must fit **exactly** to the used kernel binary. Any recompilation, even with the same kernel configuration, requires the use of new kernel debug symbols.
- The binaries and libraries forming the SystemTap framework must be installed. They are either to be loaded from the SystemTap project website or the following Linux distribution packages need to be installed:

35 Albeit these development packages only provide access to kernel header files, they are sufficient for SystemTap.

- Ubuntu: systemtap
- Red Hat: systemtap (on older systems: stap)
- SUSE: systemtap

5.1.3 SystemTap Impact on Test Results

The LRNG implementation derives its entropy from:

- high-resolution and low-resolution time stamps, and
- event-specific information

In addition, the entropy estimation heuristic rests on the low resolution time stamp.

All SystemTap scripts devised in the following will neither change the execution logic of the LRNG nor change any values processed by the LRNG. The scripts will only read data and values. Therefore, the used SystemTap scripts will not affect the event-specific information processed by the LRNG.

On the other hand, the SystemTap test logic naturally adds additional code to the processing of the CPU, when parts of the LRNG code are triggered. This additional processing has an impact on the time stamp gathering of the high- and low-resolution time stamps. However, as always the same code is executed when triggering the SystemTap logic, the gathering of the time stamps is always delayed by the same amount of (execution) time, disregarding the CPU execution jitter. Therefore, the delays act like the LRNG is executed on a slower CPU with faster high-resolution timer. The design of the LRNG is intended to be appropriate for a wide variety of CPUs and different CPU execution speeds. The LRNG is intended to operate equally well on all types of CPUs. Therefore, the impact of SystemTap scripts on the time stamp gathering is considered to minimally affect entropy collection and heuristic entropy calculation without invalidating the applicability of the obtained test results to the LRNG in general.

To obtain interpretable test results for the time stamps, the authors consider how entropy is represented with time stamps. Time stamps are monotonically increasing counters when disregarding the wrap-around that may happen when using integer variables that are unable to hold the theoretical number space of that timer. Monotonically increasing counters contain the entropy, but it is not visible by looking at the counter values. When printing a histogram of recorded time stamps, the resulting distribution should be an equal distribution, as each particular time stamp value should have an equal chance of being recorded. However, entropy manifests itself with the variations of the deltas of the time stamp values. When printing the data plot and the histogram of the time deltas, the resulting distribution shows the (hopefully unpredictable) variations. Statistical values can now be obtained with the time delta data set and distribution. For example, when applying the Shannon Entropy formula or the Minimum Entropy calculation on the time deltas, the entropy content of the time stamps can be determined. Thus, the statistical analyses in the following sections use the time deltas as a basis when assessing measured time stamps as does the LRNG.

5.1.4 Test System

The test system uses an Intel x86 Ivy Bridge Core i7 CPU with the following specifications:

- VMMs with the versions specified in section 4.3,
- Lenovo Thinkpad T530, CPU: i7 2.9GHz: 8GB RAM, 1TB hard disk with spinning disks,
- Guest system is an Ubuntu 15.10 with Linux kernel version 4.2.0.-25.

5.2 Mathematical Background

The analyses in the following section refer to several statistical values. These statistical values are briefly presented in this section.

5.2.1 Shannon Entropy

The Shannon Entropy formula (23) as defined in [AIS2031] section 2.3.2 is applied.

5.2.2 Minimum Entropy

The Minimum Entropy formula (19) as defined in [AIS2031] section 2.3.2 is applied.

5.2.3 SP800-90B Minimum Entropy

In addition to the minimum entropy calculation provided in section 5.2.2, more sophisticated approaches to the calculation of the minimum entropy are available as documented in SP800-90B ([SP800-90B]). The document specifies the following types of minimum entropy:

- Bins Test
- Collection Test
- Compression Test
- Correlation Test
- Markov Test

All listed types of minimum entropy are applied in this study.

The formulas for all minimum entropy values are given in [SP800-90B] and are not re-iterated here.

A prerequisite to perform the SP800-90B tests is the test array to determine whether the given data set is independent and identically distributed (IID). The specification of the tests for the IID property is given in [SP800-90B] section 9.1.2. These tests are performed to understand whether the data set is IID or not.

5.3 LRNG Block Device Noise Source Testing

The LRNG registers the callback function `add_disk_randomness` with the block device layer. This function is the entry point for its block device noise source. The LRNG callback function is only invoked by the block layer for a given block device, when the following condition is met: The block device must be identified as a “rotational” block device, i.e. a block device with spinning platters.

For example, when using the VirtIO para-virtualized device driver[36] in the Linux guest, the Linux kernel identifies that this device is not “rotational” and therefore excludes it from the entropy collection.

Such behavior of not using a block device as noise source is immediately visible with the test cases described in the following subsections: the test cases simply do not return any results when configuring such VirtIO para-virtualized devices. The use of a block device as noise source can conclusively be answered by reading the content of the file /sys/devices/virtual/block/<DEVICE>/queue/add_random or /sys/devices/<DEVICE>/queue/add_random where <DEVICE> is the name or the PCI address of the considered block device. If that file contains a 0, the respective device is not used for entropy collection.

5.3.1 Test Approach

The goal of the testing is to identify the behavior of the LRNG block device noise source in a virtual environment. Now, what does “behavior” mean here?

The block device noise source obtains its entropy from the following data, collected every time a block device operation is triggered:

- the block device numerical identifier as a 32 bit integer
- a low-resolution time stamp of the event as a 64 bit integer
- a high-resolution time stamp of the event as a 32 bit integer

36 The use of the VirtIO para-virtualized device driver is easily identified with the `lsblk` command which lists the used block devices as `/dev/vdX`.

These variables are defined with the `struct sample` in the function `add_timer_randomness`.

Therefore, the test must record this data for each block device I/O event. The SystemTap script raw_entropy_disk.stp obtains the data for each event and prints a result file where each of the mentioned values is printed for each block device I/O event.

A second test is devised that obtains the heuristic entropy estimation applied by the LRNG for one particular block device I/O operation. The SystemTap script `entropy_per_event_blk.stp` records the entropy estimate for each event.

To trigger block device I/O operations, the following command is invoked on the Linux guest to simulate I/O load on the block device:

```
while [ 1 ]; do
       dd if=/dev/sda of=file count=10000 oflag=direct
done
```

The command to cause I/O load reads out the first 5,120,000 bytes from the /dev/sda device and writes them into the file “file”. The option of “oflag=direct” implies that the file system objects of /dev/sda and file are opened with the O_DIRECT flag discussed in the open(2) man page. That flag implies that the buffer cache of the guest Linux kernel is not used and the guest Linux kernel performs a real block device I/O operation for each read request. The command is executed in an endless loop to simulate a worst case scenario where the block device is strained with read and write operations.

The following subsections enumerate the different VMMs on which the test is executed. They also contain a conclusion in the form of configuration requirements to ensure that the VMM impact on the noise source is reduced to the extent this noise source operates appropriately or to ensure that the noise source operation is deactivated if the VMM impact renders the noise source as unsuitable.

5.3.2 KVM/QEMU

The Linux kernel starting with version 2.4 implements the O_DIRECT flag for the “open” system call. Files opened with this flag do not use the buffer cache for disk accesses. This flag was intended for processes implementing their own caching logic.

When the QEMU process in the KVM host OS opens the backend disk devices with O_DIRECT, the disk access is not cached by the host Linux kernel buffer cache.

The O_DIRECT flag can be configured with QEMU when using the `-device` command line flag and the flag “cache=none”. The enforcement of the O_DIRECT flag is visible when checking the contents of the fdinfo /proc file of the file descriptor for the disk file. For example, the file descriptor 10 is the disk backend file:

```
# ls -la /proc/30427/fd
insgesamt 0

...
lrwx------ 1 libvirt-qemu kvm 64 Mär  8 10:53 10 →
/var/lib/libvirt/images/Fedora17-18.img
...

# cat /proc/30427/fdinfo/10
pos:    16106127360
flags:  02140002
```

Please consider the flags field and compare it to the value of O_DIRECT defined in the kernel source code in fcntl.h:

```
#define O_DIRECT 00040000
```

When comparing the definition of O_DIRECT in the code and the flags output from the used QEMU disk file – which is a bit mask – it is clear that the O_DIRECT flag is set.

As discussed in section 4.3.1.1.2, the KVM host allows the configuration of two possible use cases relevant to the block device operation:

- enabling the buffer cache in the VMM (i.e. the host Linux kernel that executes the QEMU application encapsulating the virtual machine), or
- disabling the buffer cache in the VMM.

As noted in the discussion around the LRNG block device noise source, its behavior is modeled to pick up variations of hard disk accesses. Therefore, the following tests only configure block devices for the tested virtual machine whose backend storage is maintained on a hard disk with spinning platters, as well: a file-based backend is used which is stored on the VMM's hard disk. The use of network file systems as well as the use of SSDs is ignored.

In the following subsections, different use cases and virtual machine configurations are tested.

5.3.2.1 Use of VirtIO for Block Device Access

The virtual machine which was used for testing, was configured to access the block device via the VirtIO bus. After starting the virtual machine, the SystemTap test script was executed on the guest to record the block device events obtained by the LRNG.

No test results were recorded. This means that the block device callback function of the LRNG `add_disk_randomness` was not invoked and the LRNG was unable to collect entropy from that block device. This is readily clear when consulting the add_random file:

```
cat /sys/devices/pci0000:00/0000:00:07.0/virtio2/block/vda/queue/add_random
0
```

The file contents shows a zero which implies that the VirtIO disk will not contribute to the LRNG block device noise source.

5.3.2.2 VMM Buffer Cache Disabled

The next test is now conducted with the following guest configuration:

- Use of the SATA bus emulation offered by KVM.
- Disabling the VMM buffer cache by using the QEMU command line option of “cache=none” for the instantiation of the guest.

After the Linux guest is started, the SystemTap script is invoked. To stimulate block device I/O operations, the dd command listed above is used.

The first results of interest are derived directly from the dd command behavior:

- The HDD LED of the test system was lit solid for the entire time of the dd command execution. That indicates that the buffer cache of the guest is unused (as intended with command's option of “oflags=direct”). Furthermore, the buffer cache of the host is not in use either, as intended with the VMM configuration mentioned above.
- The read/write speed of the dd command is about 1.3 MBytes per second. This value is similar to the speed obtained directly in the host environment. It shows that all buffer caches along the way of the I/O request from the guest to the physical hard disk are disabled.

Using the output of the SystemTap scripts, the following results can be observed.

5.3.2.2.1 Block Device Numerical Identifier

The block device numerical identifier is always the same number. This is expected as the command to stimulate the block device always accesses the same device. Therefore, no entropy is derived from that block device number.

5.3.2.2.2 Jiffies Delta of Block Device I/O Events

The Jiffies coarse time stamp which is recorded during the testing returned data as illustrated in figure 11, providing a data plot of the Jiffies delta values. The data plot shows the following properties:

- The abscissa depicts the 100,000 samples taken during testing.
- The ordinate specifies the Jiffies delta.
- The vast majority of deltas are either zero or one – which is shown with the black bar at the bottom (the diagram shows a line graph – if the diagram would be zoomed in, a flicker between zero and one could be seen).
- Very few larger time deltas are recorded.

The legend in the diagram shows a number of statistical values where the following are of interest here:

- The Shannon Entropy is defined with 0.43, which means that the Jiffies delta hardly contain any entropy.
- The median of 0 and a mean of 0.09 show that the vast majority of the recorded Jiffies deltas are zero. Note that, albeit the black bar would indicate that a large number of ones are recorded, the number of ones are comparatively small.

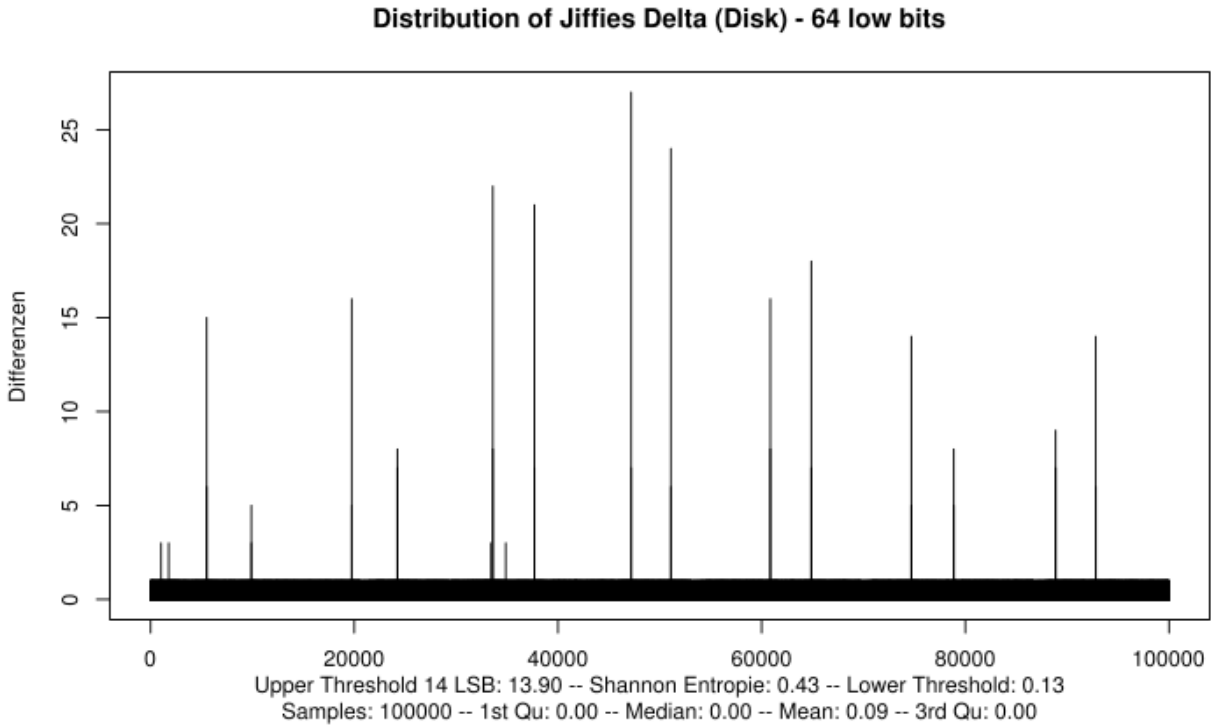

Figure 11: Distribution of Jiffies Delta for Block Device – KVM Without Buffer Cache

The conclusions drawn from figure 11 are supported with figure 12 which provides the histogram of the Jiffies delta. The histogram shows that the vast majority of Jiffies deltas is zero.

The diagram in figure 12 contains the following additional information:

- The two green vertical bars indicate the first and third quartile of the data set.
- The mean of the data set is indicated with a red vertical bar.
- The median of the data set is indicated with a blue vertical bar.

- The red dotted line is a normal distribution with the mean and standard derivation of the data set. This line should help to put the distribution of the data set in perspective with a normal distribution.

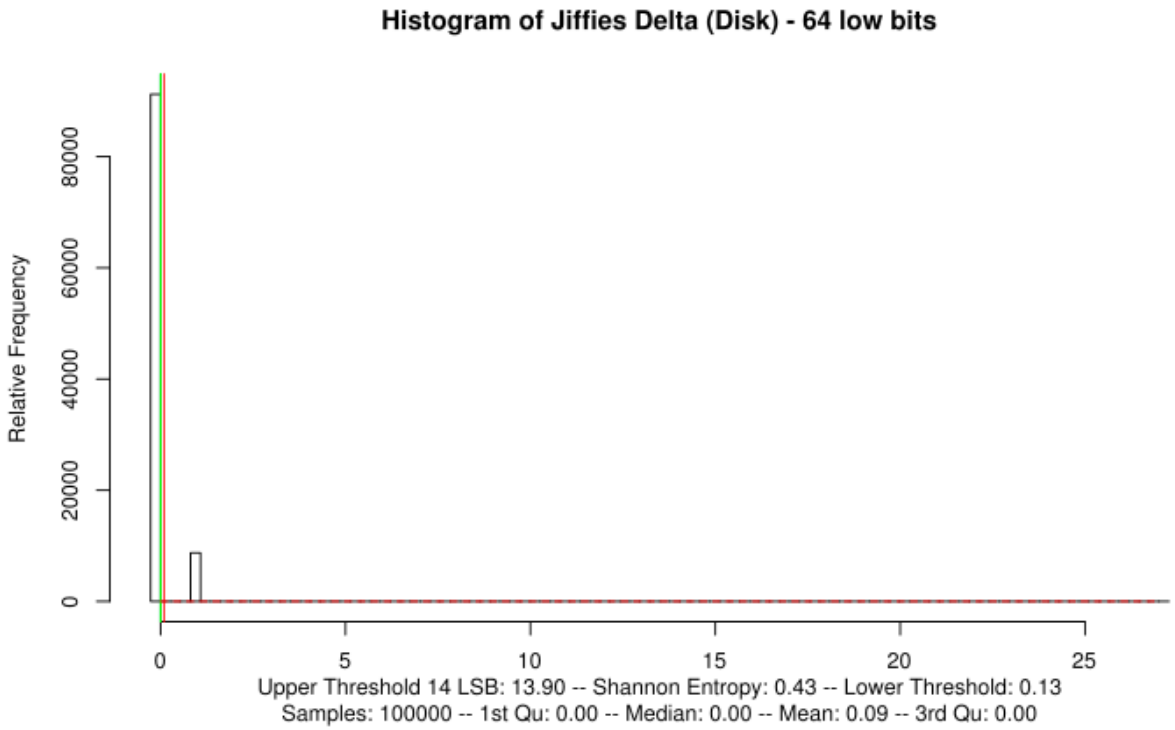

Figure 12: Histogram of Jiffies Delta for Block Device – KVM Without Buffer Cache

Both figures allow the final conclusion, that very little entropy is collected with the Jiffies time stamp for block devices with a worst case scenario.

5.3.2.2.3 High-Resolution Time Delta of Block Device I/O Events

The same measurements as received for the Jiffies delta, are obtained for the high-resolution time delta.

Figure 13 contains the plot of the high-resolution time delta. A few spikes are visible which are attributed to rescheduling events by the VMM. Therefore, these spikes are expected and do not illustrate any anomaly. The rest of the information is found in the graph between those spikes. To analyze what happens between these re-scheduling spikes, the data set is “zoomed-in” by discarding all high bits above 22. The value 22 is an artificial threshold which was chosen as it provides the significant information without loosing too much precision.

Figure 14 contains the “zoomed-in” plot of the high-resolution timer that only use the low 22 bits. With this plot, hardly any information is lost as demonstrated by comparing the Shannon Entropy value in the legend of figures 13 and 14: both values are identical to the second places after the decimal point.

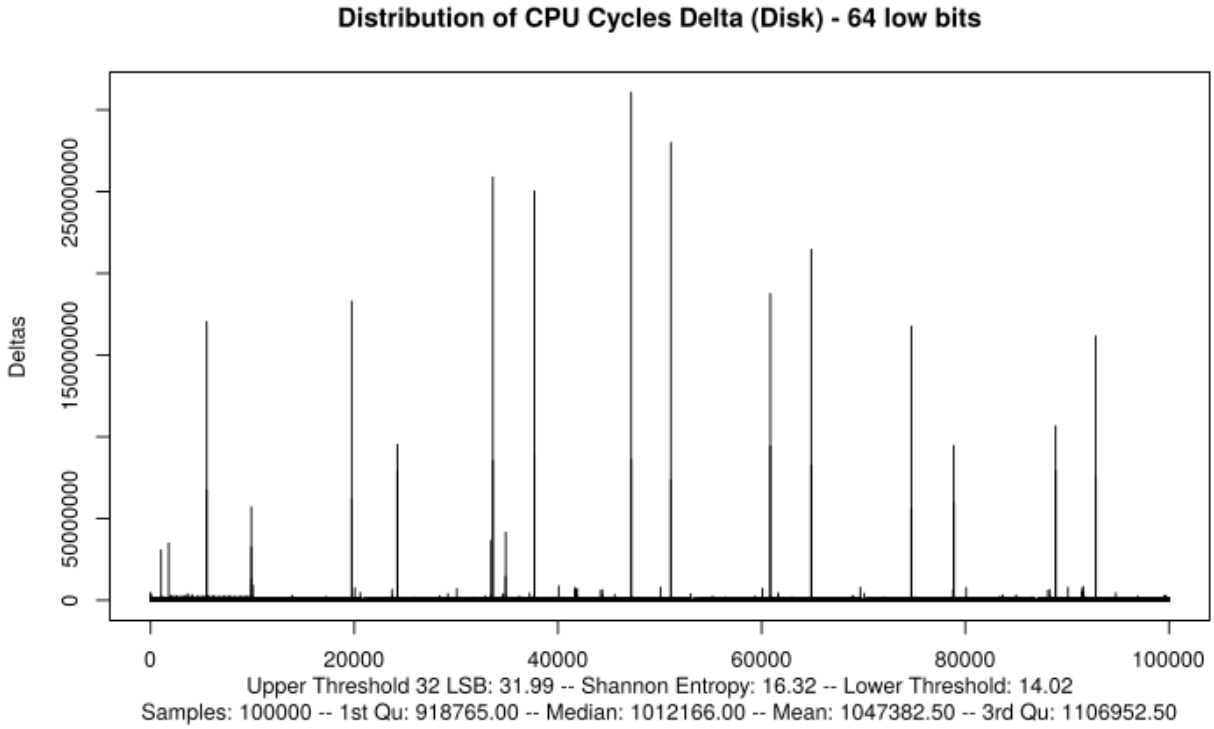

Figure 13: Distribution of high-resolution time delta for block devices – 64 bit – KVM Without Buffer Cache

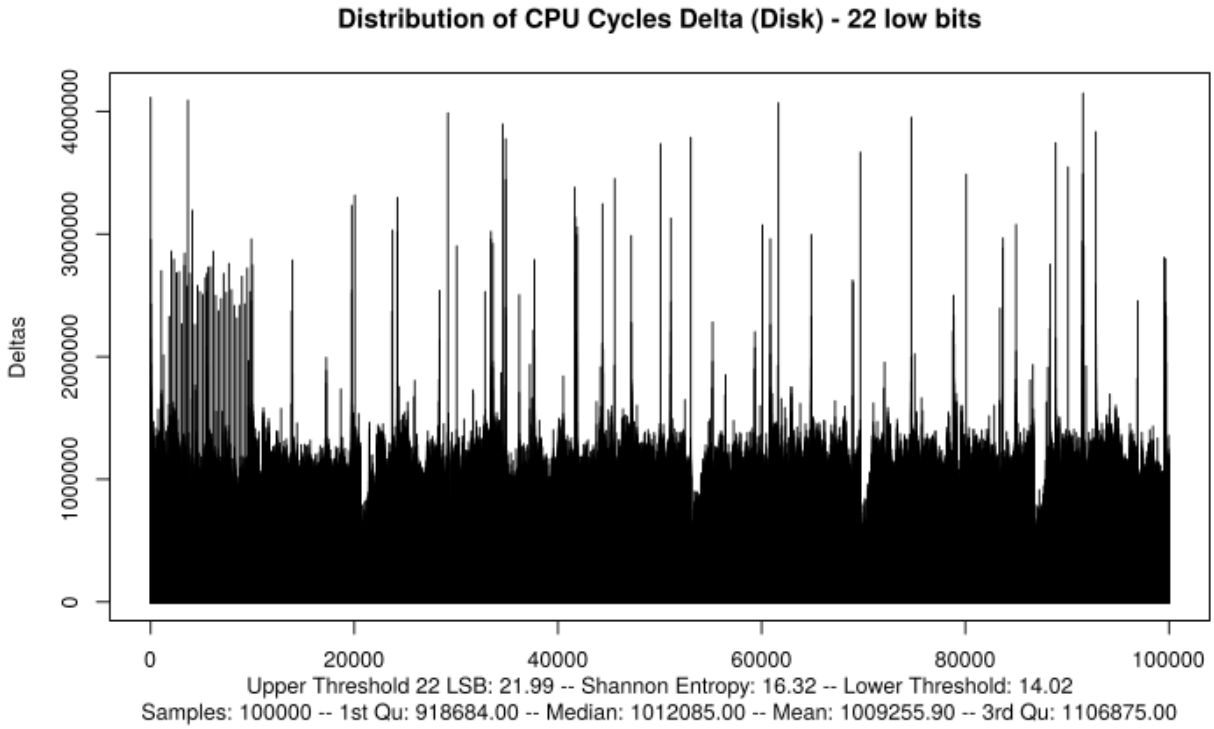

Figure 14: Distribution of high-resolution time delta for block devices – 22 bit – KVM Without Buffer Cache

The plot of the 22 low bits of the high-resolution time stamp shows a graph where the majority of time deltas is in the range from 1 million to 1.5 million ticks. A number of spikes are shown which can be interpreted as re-scheduling events of the guest by the underlying VMM. Yet, the variations of the time deltas visible with the Shannon Entropy value shows more than 16 bits.

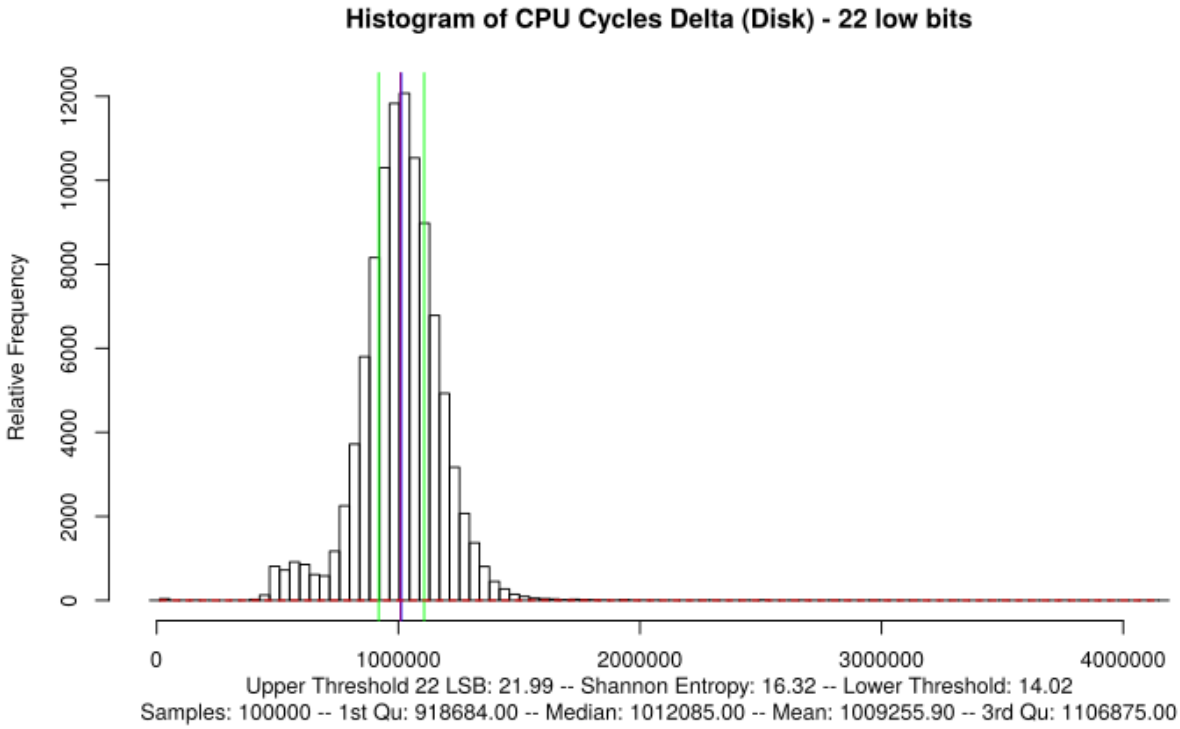

Figure 15: Histogram of High-Resolution Time Deltas for Block Devices – 22 low bits – KVM Without Buffer Cache

The histogram for the data plot is given with figure 15.

With this value, it is clear that the high-resolution time stamps deliver the majority of the entropy for the LRNG's block device noise source.

Furthermore, the Shannon Entropy value is significantly above the maximum heuristic entropy value of 11 bits that can be awarded by the LRNG to a particular block device I/O operation. For details about the threshold of 11 bits of maximum heuristic entropy and how the heuristic entropy is calculated, please see [LRNG].

In conjunction with the assessment of the entropy content in the high-resolution time stamp, the test of the entropy estimate applied by the LRNG entropy heuristic is conducted. The result of the test is depicted in figure 16. This histogram shows the 12 bit values from zero bits to 11 bits that can be awarded to an event by the LRNG entropy heuristic.

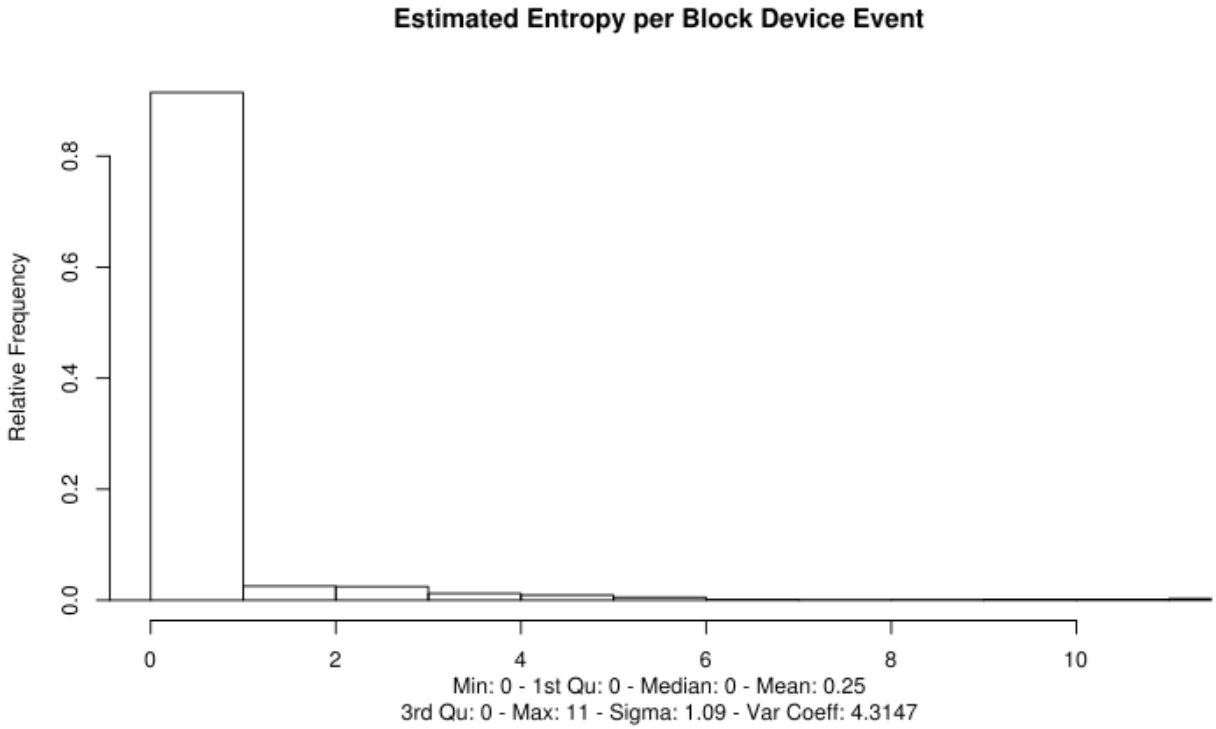

Figure 16: Estimated Entropy per Block Device Event – KVM Without Buffer Cache

Figure 16 allows the following conclusions to be drawn:

- The LRNG heuristic entropy estimation still works in the virtual environment.
- It still awards zero bits of entropy to the majority of events.
- Considering figure 14 with its Shannon Entropy value, it is clear that the entropy heuristic still massively underestimates the entropy found in block device I/O operations.

In addition to the graphs, the various types of Minimum Entropy defined in [SP800-90B] are calculated from the high-resolution time deltas. Before performing the calculations of the different Minimum Entropy values, [SP800-90B] asks for the determination whether the data is IID (independent and identically distributed). Various tests are defined which must be applied to the input data set. The data set to be analyzed for being IID is the absolute time stamp data set. The reason for using the absolute time stamp at this point is that it is clear that if this value is IID, the time deltas must be IID as well. However, this is not true in the opposite direction: if the time delta is considered to be IID, the absolute time stamp does not need to be IID. When processing the time stamp, it is clear that the most significant bits depend on each other. Considering that the LRNG entropy estimator is capped at 11 bits, only the 11 least significant bits of the time deltas are analyzed for the IID property. When calculating the IID tests, the result shows that 50ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The different types of Minimum Entropy can commonly only be calculated for a bit stream where the data chunks are in the single digit range. To comply with the prerequisites to the Minimum Entropy calculations, the time deltas are processed as follows:

- 4 bit chunks:
  - the four least significant bits (LSB) from each of the time delta value are used
  - two four bit values from successive time deltas are concatenated to form one byte – the first four bits are used for the high nibble of one byte and the second four bits are used for the lower nibble of a byte

    - the bytes formed from two time deltas are concatenated to form a byte stream
- 8 bit chunks:
    - the eight least significant bits (LSB) from each of the time delta value are used and are individually treated as one byte
    - the bytes formed from the time deltas are concatenated to form a byte stream

The processing of the time delta results in a set of byte streams which are now input into the Minimum Entropy formulas discussed in [SP800-90B]. The results of the calculation are given in table 1.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.529987	N/A[37]
Bins	3.867346	6.500445
Collection	2.903738	5.829157
Compression	2.961081	5.728355
Collision	2.952748	5.400626

Table 1: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG with a mean value given in in figure 16. Hence, the entropy estimator significantly underestimates the available entropy.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

Therefore, the conclusion can be reached that when the VMM buffer cache is disabled, KVM does not significantly interfere with the block device noise source of the LRNG.

5.3.2.3 VMM Buffer Cache Enabled

The testing conducted in section 5.3.2.2 is repeated. However, this time the guest configuration is changed such that the buffer cache of the VMM is enabled. This is achieved by omitting the “cache” command line option. The default of QEMU is documented in the `qemu(1)` man page as “writeback” which uses the buffer cache. All other configurations of the guest are identical to section 5.3.2.2.

Again, after starting the guest and invoking the dd command to stimulate disk use, the first results are readily visible:

- The hard disk LED of the hardware system marginally flickers during the execution of the dd command. This indicates that the guest block device I/O operations only rarely reach the physical disk.
- The read/write throughput of the dd command is around 5 MBytes per second and is significantly higher compared with a configuration without the buffer cache in the VMM as documented in section 5.3.2.2.

Again, the following sections now discuss the different test results from the SystemTap scripts.

5.3.2.3.1 Block Device Numerical Identifier

The result from section 5.3.2.2.1 is unchanged: only one block device identifier is recorded which implies that no entropy is derived from this identifier.

37 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

5.3.2.3.2 Jiffies Delta of Block Device I/O Events

The Jiffies time delta recorded for the LRNG on a VMM with buffer cache are shown in figure 17. This figure shows the same plot as in section 5.3.2.2.2. When comparing this diagram with the diagram when the VMM buffer cache is disabled in section 5.3.2.2.2, it is evident that almost all Jiffies delta values are zero. The spike for the first recording relates to the initialization of the test variable and can therefore be ignored. Also, when comparing the Shannon Entropy value between the test, with and without VMM buffer cache, significant differences are visible: the test results with VMM buffer cache has a Shannon Entropy value that is only half of the value on the VMM without buffer cache.

This result already demonstrates the significance of the VMM buffer cache.

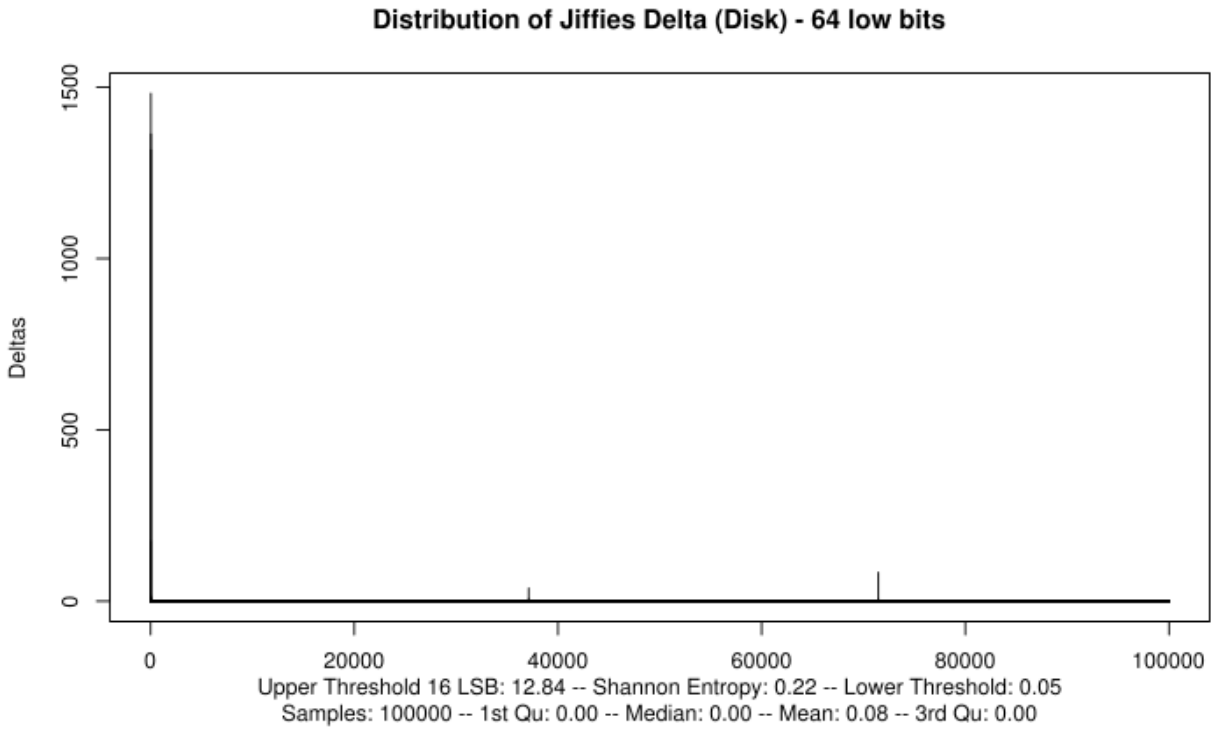

Figure 17: Distribution of Jiffies Delta for Block Devices – KVM With Buffer Cache

Again, supporting to the plot is the histogram of the Jiffies delta given with figure 18. This histogram shows that almost all Jiffies deltas are zero values.

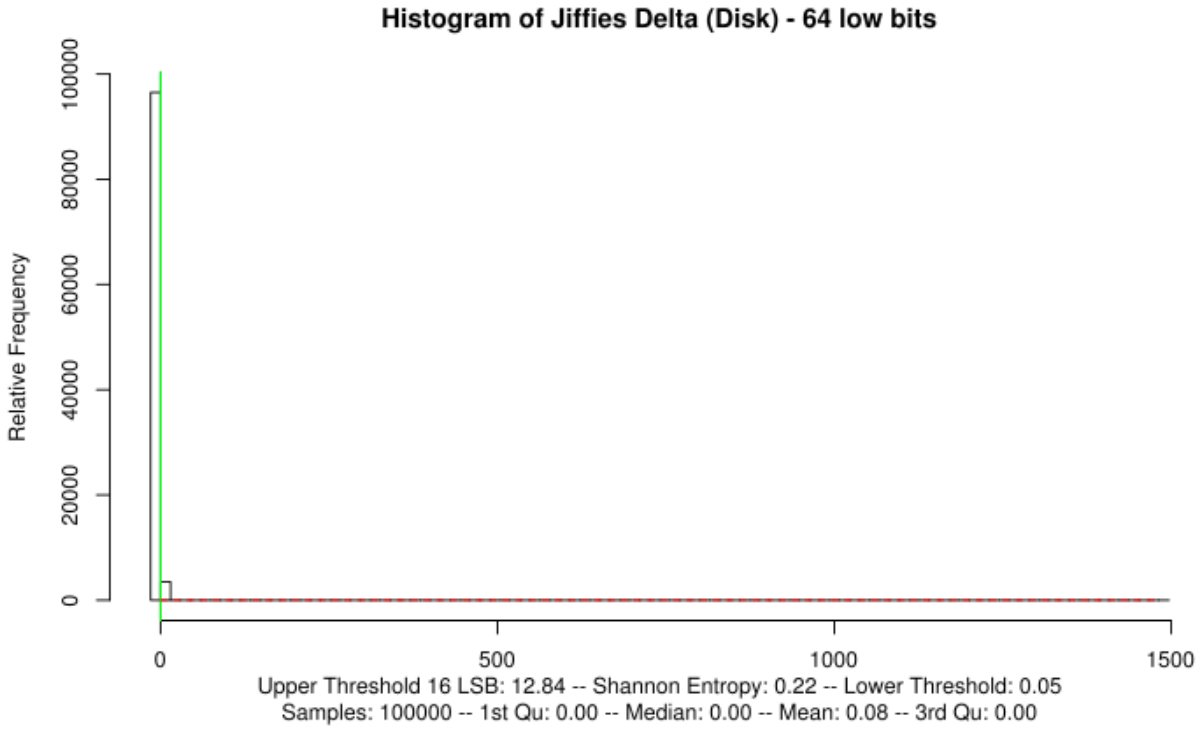

Figure 18: Histogram of Jiffies Delta for Block Devices – KVM With Buffer Cache

Although significant differences are evident when comparing the distribution of the Jiffies delta between a VMM with and without buffer cache, the conclusion is the same: the Jiffies only provide minimal entropy to the LRNG block device noise source. The level is so minimal for the VMM with buffer cache that it can effectively be disregarded.

5.3.2.3.3 High-Resolution Time Delta of Block Device I/O Events

Similarly to the assessment in section 5.3.2.2.3, this section analyzes the high-resolution time delta of I/O events.

The discussion in section 5.3.2.2.3 explained that a “zoom-in” is helpful to visualize the results in a way to allow interpretation. Figures 19 and 20 present the test results “zoomed-in” to the 22 low bits to allow a comparison with the figures in section 5.3.2.2.3.

The plot given in figure 19 shows that the high-resolution time deltas oscillate between 300.000 and 500.000 clock ticks and have a few spikes. The time deltas are significantly lower compared to the time deltas obtained with disabled VMM buffer cache. This demonstrates that the majority of the block device I/O requests are satisfied from the VMM buffer cache rather than from the physical hard disk.

The spikes in the plot are attributed to a re-scheduling event of the guest by the VMM. The number of these spikes is lower compared to the number of spikes observed from the testing on KVM with VMM buffer cache disabled. This can be explained with the fact that the test with buffer cache enabled executed much faster. Hence the VMM had less opportunity to re-schedule the guest.

A very interesting result, however, is the value for the Shannon Entropy: it is only one bit less compared to the results obtained with the VMM without buffer cache. The value of the Shannon Entropy is still significantly over the threshold of 11 bits applied by the entropy estimation heuristic of the LRNG. Considering the estimation heuristic result for block device I/O events depicted with figure 16 showing that on average a block device I/O event is awarded 0.25 bits of entropy, the conclusion can be drawn that the block device I/O events provide sufficient variations to satisfy the LRNG entropy estimation.

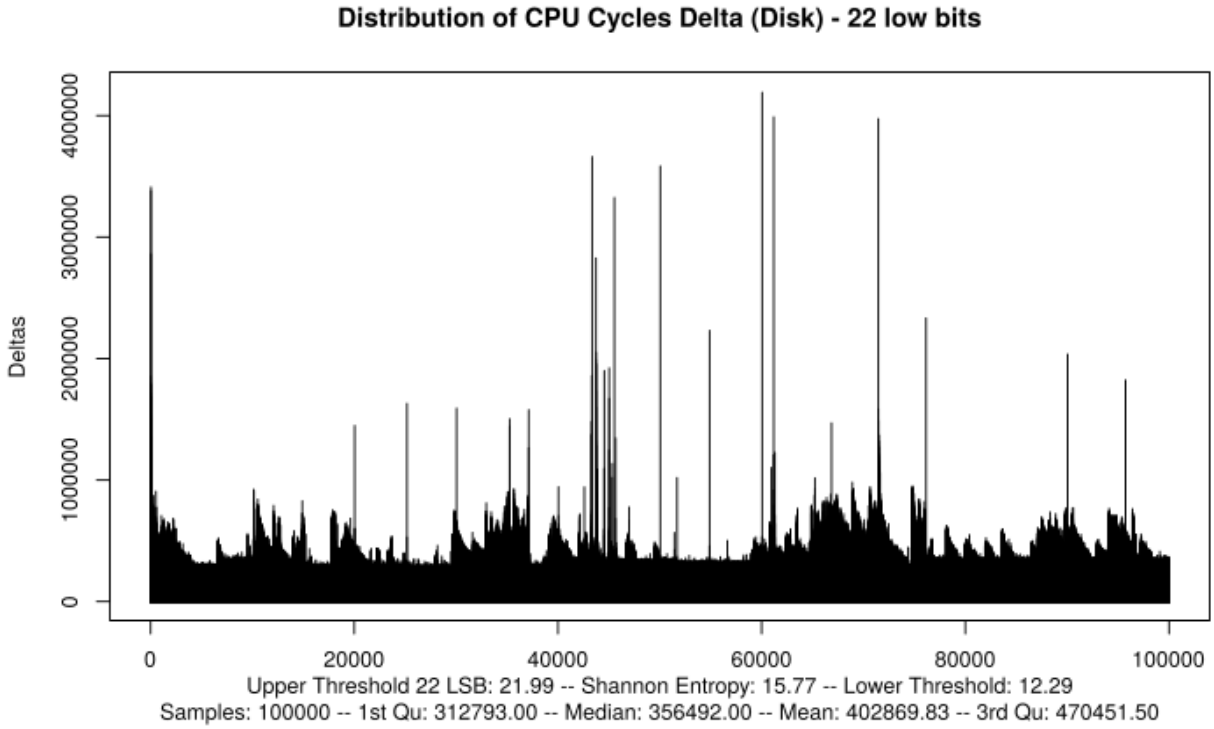

Figure 19: Distribution of High-Resolution Time Delta for Block Devices – 22 low bits – KVM With Buffer Cache

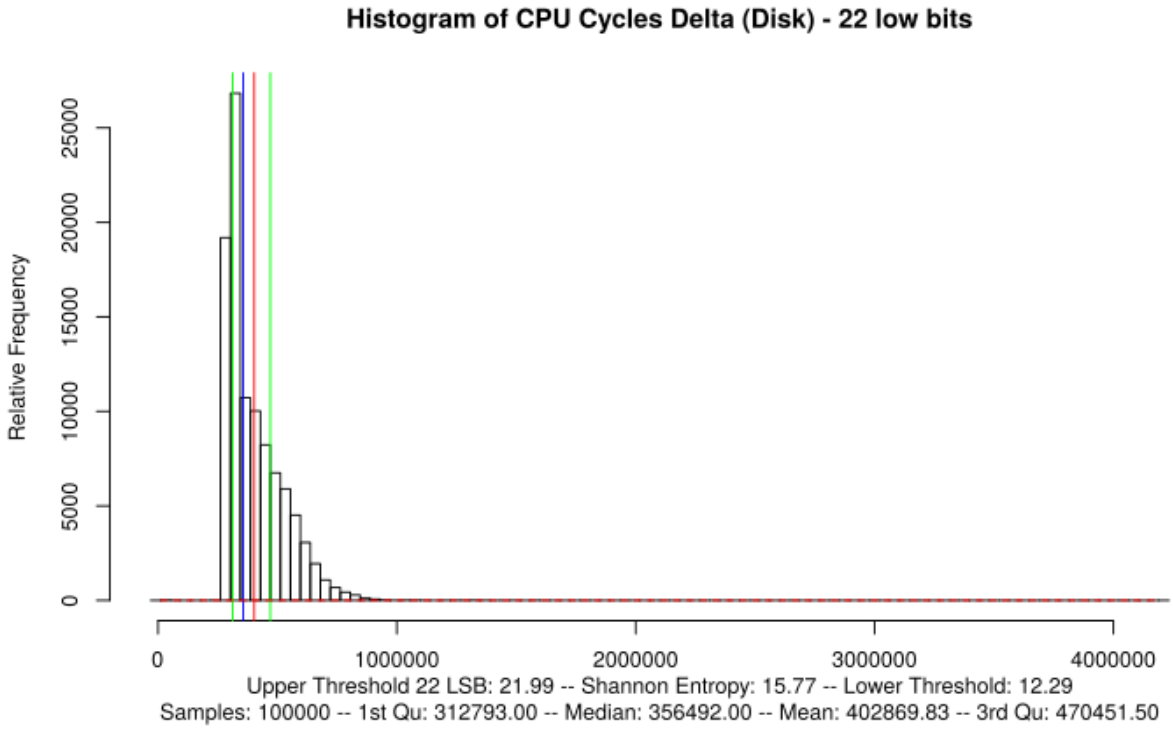

Figure 20: Histogram of High-Resolution Time Deltas for Block Devices – 22 low bits – KVM With Buffer Cache

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 49 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 2.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.467964	N/A[38]
Bins	3.852682	6.509618
Collection	2.622315	5.259129
Compression	2.612185	5.178301
Collision	2.701991	5.115189

Table 2: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

Albeit the variations of the block device I/O events are sufficient to cover the needs of the LRNG, it should not be forgotten that in case of using the VMM buffer cache for satisfying the majority of the I/O requests, the underlying unpredictable phenomenon for the block device noise source is replaced. Whereas the block device noise source without the VMM buffer cache rested on the uncertainty of the disk accesses, the block device noise source with VMM buffer cache rests on the CPU execution jitter that is analyzed to a large extent in [JENT].The measurement shows that the replacement of the unpredictable phenomenon, however, does not affect the suitability of the noise source! Therefore, it is concluded that the use of the VMM buffer cache does not affect the LRNG operation in a way to make it unsuitable.

The assessments and large scale tests provided in [JENT] are directly applicable to the test results obtained here. The tests in [JENT] showed that for a given code snippet, variations in the execution time of that code exist. These variations are found in all analyzed CPU architectures to a similar degree, including Intel x86, ARM, MIPS, IBM System z, IBM POWER, and Sparc. Furthermore, tests were executed on a bare metal environment, i.e. without an operating system and without interrupts enabled to demonstrate that the variations do not result from the operating system mechanisms. Thus, the analysis showed that the CPU execution time jitter is an intrinsic phenomenon in the CPU. However, the absolute root-cause of the phenomenon is not yet fully understood even though the author of [JENT] discussed the matter with different major chip vendors. To support the analysis, one vendor agreed to perform a chip-internal analysis to track down the root-cause for the CPU execution time jitter.

There is one significant distinction though between the VMM with and without buffer cache: the LRNG entropy heuristic is based on calculating the first, second and third derivation of the Jiffies values. The measurements above have shown that the block device I/O requests are significantly faster on a VMM with buffer cache rather than without buffer cache, therefore it is clear that the chances of the LRNG entropy heuristic being zero is much higher when using a buffer cache. That means that the LRNG awards less entropy to block device events when using the VMM buffer cache. Thus, even though sufficient entropy is present on systems even with the host buffer cache enabled, the LRNG credits only a very small amount of entropy.

38 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

5.3.2.4 Conclusion: Configuration Requirement

Considering the test results and the discussions of these tests, the following configuration constraints must be considered to enable the block device noise source:

- Do not use the VirtIO bus for the block device when configuring a KVM virtual machine.
- With the enabling of the host buffer cache, a complete different unpredictable phenomenon is the basis of the noise source. This phenomenon is well tested and all tests indicate that the phenomenon is an intrinsic property of modern CPUs, the root cause is not yet fully understood. Therefore, if the reader has concerns about the change of the unpredictable phenomenon underlying the noise source, the host buffer cache should be deactivated.

In addition, the following configuration constraint is suggested to gain more entropy from the block device noise source:

- The backend of the block device offered to a virtual machine must be stored on the hard disk with spinning disks (i.e. no SSDs) accessed by the VMM. This is done by either using a file as a storage backend hosted on a hard disk with spinning disks for the guest block device or by assigning a partition on a block device with spinning disks to the guest.

5.3.3 Oracle VirtualBox

The description in section 5.3.2 explains that the Linux kernel implements the O_DIRECT flag. Since VirtualBox uses the Linux kernel as part of the VMM, the question arises whether that flag is also in use by VirtualBox.

When starting a guest with VirtualBox, the host environment executes the application `VirtualBox`, which is logically equivalent to QEMU for KVM. Therefore, the same assessment as done for QEMU in section 5.3.2 can be applied here, too.

Without any specific configuration, the following can be obtained at runtime of a `VirtualBox` application instance:

```
# ls -la /proc/19781/fd
...
l-wx------ 1 root   root   64 Apr  8 09:29 42 ->
/home/user/vb/testguest/testguest.vdi
...
#  cat /proc/19761/fdinfo/42
pos:    0
flags:    02140000
```

The listing shows the file descriptor, the `VirtualBox` process uses for the block device backend file. When analyzing the options used with this file descriptor in the “flags” field, it is visible that per default VirtualBox uses the O_DIRECT flag as the flags value includes the bit value 0x40000.

When re-configuring the guest using the VirtualBox UI and enabling the use of the host cache in the configuration mask for the block device bus type, and starting the guest again, the following can be observed:

```
# ls -la /proc/19944/fd
...
l-wx------ 1 root   root   64 Apr  8 09:29 42 ->
/home/user/vb/testguest/testguest.vdi
...
#  cat /proc/19944/fdinfo/42
pos:    0
flags:    02100000
```

The listing of the flags field shows that the bit 0x40000 is not set, which implies that the O_DIRECT flag is not set for the block device backend file.

Therefore, just like QEMU/KVM, the VirtualBox VMM allows the configuration of:

- enabling the buffer cache in the VMM as a default configuration, or
- disabling the buffer cache in the VMM by enabling the configuration option of the host cache.

Again, just like for KVM/QEMU, the subsequent tests only configure block devices for the tested virtual machine whose backend storage is maintained on a hard disk with spinning disks as well – a file-based backend is used which is stored on the VMM's hard disk. The use of network file systems as well as the use of SSDs is disregarded.

The following subsections test different use cases and virtual machine configurations.

5.3.3.1 VirtIO for Block Devices and VirtualBox

VirtualBox does not support the use of VirtIO for block devices. The emulated block device bus is SATA. The block device layer of the Linux guest kernel therefore enables this block device for entropy collection. The following command executed in the Linux guest demonstrates this:

```
# cat
/sys/devices/pci0000:00/0000:00:0d.0/ata3/host2/target2:0:0/2:0:0:0/block/sda/queu
e/add_random
1
# lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0    20G  0 disk
├─sda1   8:1    0   9,9G  0 part /
└─sda5   8:5    0   1,5G  0 part [SWAP]
```

The command listing shows that the `add_random` boolean is set to one, which implies that the sda disk is enabled for entropy collection.

In addition, the `lsblk` listing demonstrates that the sda block device is used for the partitions supporting the operating system.

5.3.3.2 VMM Buffer Cache Disabled

The following test is performed using the following guest configuration:

- The configuration option of enabling the “Host I/O Cache” is not set for the block device bus configuration.

After starting the Linux guest, the SystemTap script is executed. Block device I/O operations are caused by using the `dd` command listed above.

Again, the first results are derived from the `dd` command:

- The HDD LED of the test system flickered heavily for the entire time of the `dd` command execution. This indicates that the buffer cache of both, the guest and host are unused. However, the LED is not lit solid as for KVM, which is unexpected, as it is an indication that not all block device I/O requests are sent to the physical hard disk.
- The read/write speed of the `dd` command is about 6 MBytes per second. This read speed is about the same as for KVM **with** enabled VMM buffer cache. This is a totally unexpected result and cannot be explained at this point as it would imply that the VMM buffer cache **is** used. Note, comparing that result with the `dd` speed when the VMM buffer cache is enabled for VirtualBox as listed in section 5.3.3.3 shows that both speeds are very similar indicating somehow a cache **is still in use**.

Using the output of the SystemTap scripts, the following results can be observed.

5.3.3.2.1 Block Device Numerical Identifier

As expected from the KVM testing, the same result is obtained as discussed in section 5.3.2.2.1: only one block device identifier is recorded which implies that no entropy is derived from this identifier.

5.3.3.2.2 Jiffies Delta of Block Device I/O Events

The distribution of the Jiffies delta with the VMM buffer cache disabled is presented in figure 21. The diagram has strong similarities to the graph in section 5.3.2.3.2 showing the KVM result when the VMM buffer cache is enabled. The Shannon Entropy of the VirtualBox results with disabled VMM buffer cache is even lower than the Shannon Entropy value for KVM with VMM buffer cache enabled.

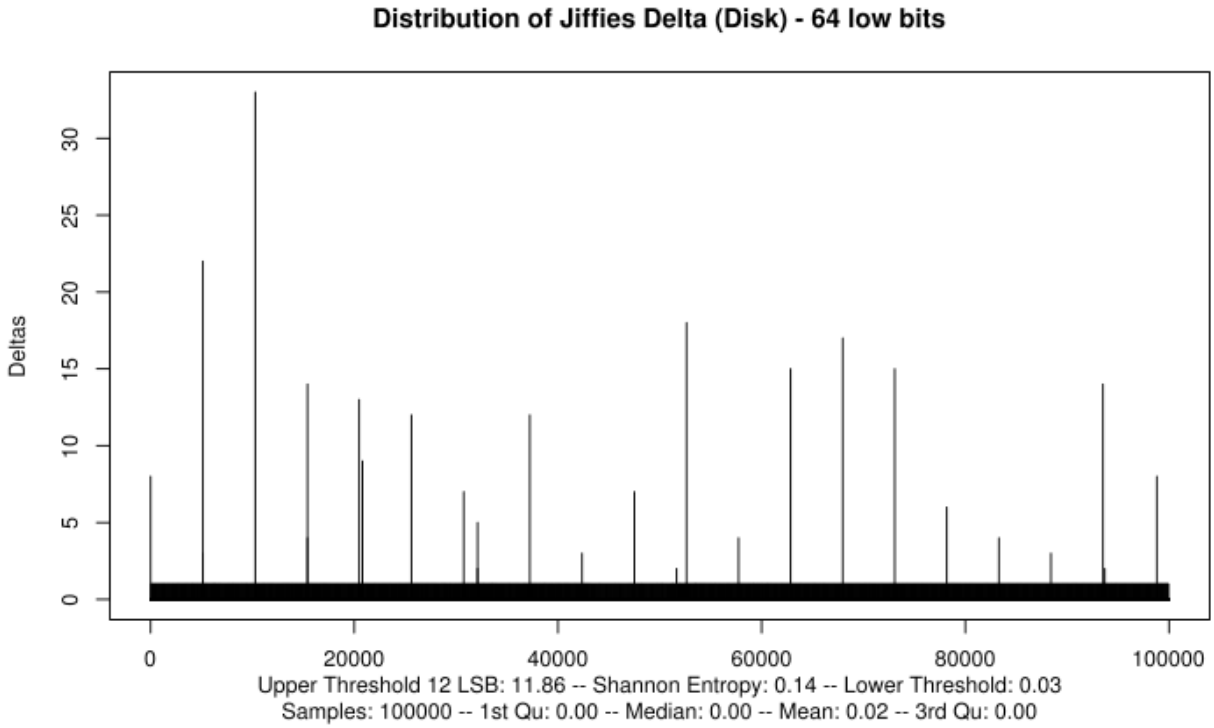

Figure 21: Distribution of Jiffies Delta for Block Device – VirtualBox Without Buffer Cache

The histogram of the Jiffies delta given in figure 22 supports the distribution result by indicating that almost all deltas are zero.

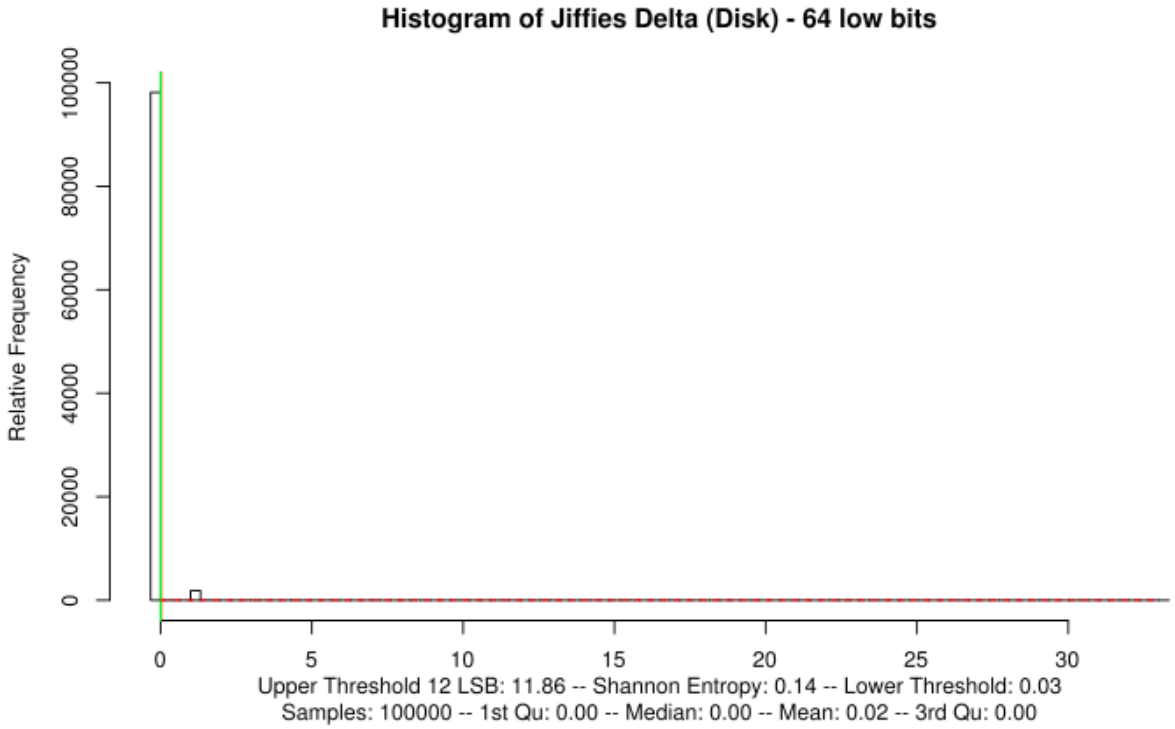

Figure 22: Histogram of Jiffies Delta for Block Device – VirtualBox Without Buffer Cache

The conclusion from the Jiffies delta results can be summarized as follows:

- The Jiffies deliver an insignificant amount of entropy to the block device noise sources.
- The test results from a Linux guest executing when the VMM has disabled the buffer cache have more similarities with the KVM results when the buffer cache is **enabled**. This supports the surprising test result of the dd command from section 5.3.3.2 – it is an additional strong hint that the VirtualBox VMM employs some additional buffer cache in addition to the Linux kernel buffer cache.

5.3.3.2.3 High-Resolution Time Delta of Block Device I/O Events

The high-resolution time stamps obtained from the block device I/O events are analyzed in this section.

The discussion in section 5.3.2.2.3 showed that “zooming-in” supports the visualization of the results such that they can be interpreted. The figures 23 and 24 depict the test results by only showing the 22 low bits to support a comparison with the figures in other sections of this document.

With the data plot given in figure 23 it is evident that the high-resolution time deltas oscillate between 300.000 and 500.000 clock ticks with a few spikes in-between. The time deltas are very similar to the KVM time deltas obtained with VMM buffer cache enabled. Again, this result is a strong hint to the VirtualBox cache handling given above.

The spikes in the plot are attributed to a re-scheduling event of the guest by the VMM. The number of these spikes is in line with the spikes observed for the testing on KVM with VMM buffer cache enabled.

The Shannon Entropy value is similar to the Shannon Entropy obtained for the testing on KVM with VMM buffer cache. Again, the value of the Shannon Entropy is significantly higher than the threshold of 11 bits used as a cap by the entropy estimation heuristic of the LRNG.

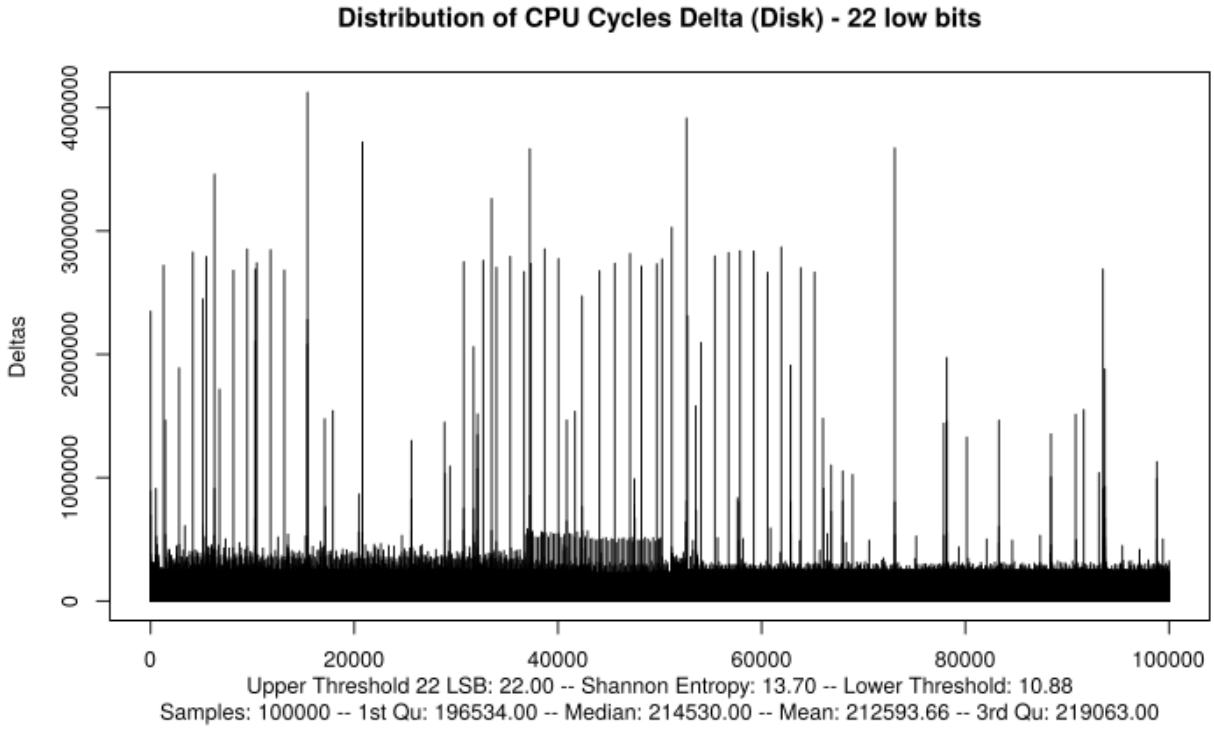

Figure 23: Distribution of High-Resolution Time Deltas for Block Device – 22 low bits – VirtualBox Without Buffer Cache

The histogram given in figure 24 supports the results obtained from the data plot. The reader, however, should consider that the bars in the histogram include a range of values. Therefore, the two large bars in figure 24 cover a number of different high-resolution time deltas. The Shannon Entropy value given in the legend of figure 24 indicates this.

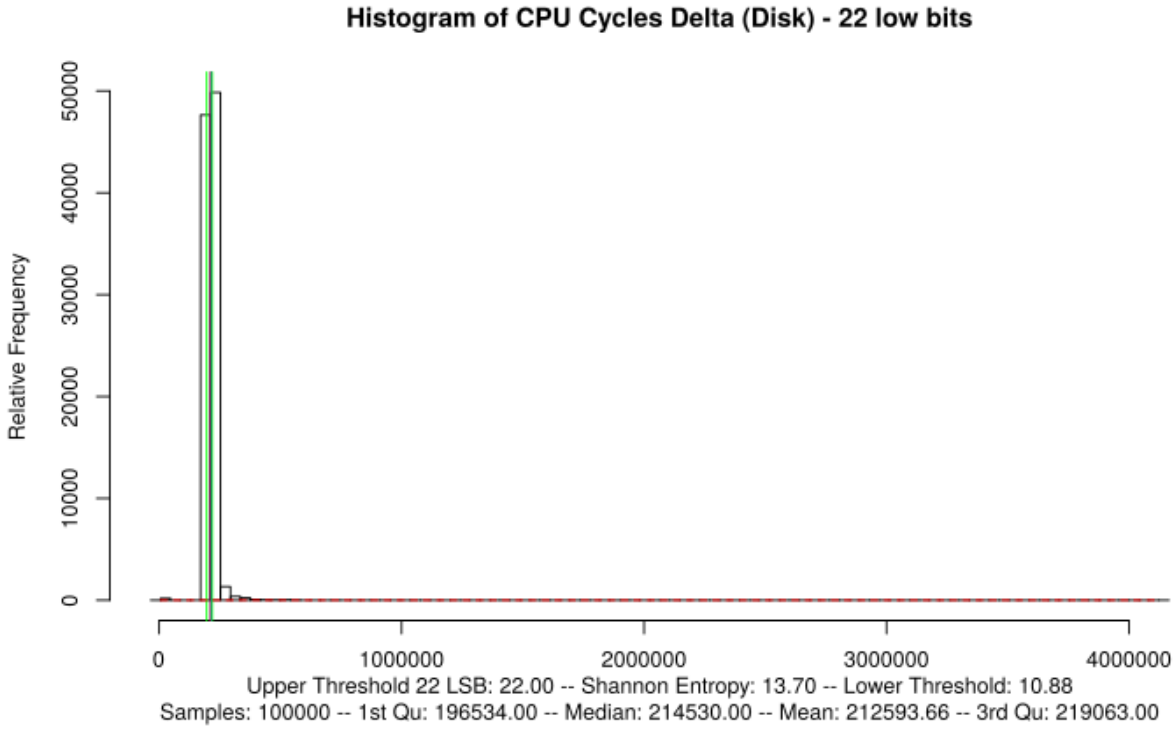

Figure 24: Histogram of High-Resolution Time Deltas for Block Device – 22 low bits – VirtualBox Without Buffer Cache

In addition to the distribution of the high-resolution time delta, the result of the entropy heuristic applied by the LRNG is shown in figure 25. This figure lists the 12 entropy values which can be calculated by the LRNG entropy heuristic and shows how often the LRNG awards which entropy value to block device I/O events.

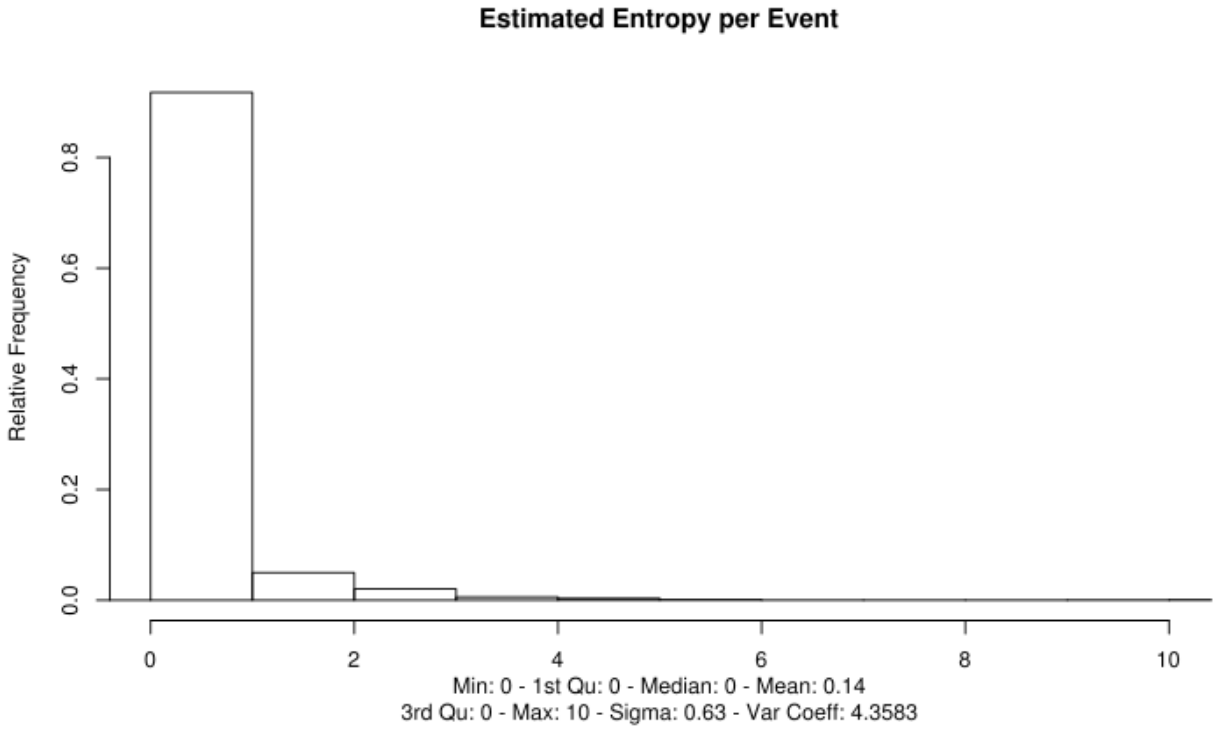

Figure 25: Heuristic Entropy per Block Device Event – VirtualBox Without Buffer Cache

Figure 25 shows that the majority of 90% of block device I/O events are awarded with zero bits of entropy. This finding is supported by the mean value of the awarded entropy of 0.14 bits. Comparing this value with the Shannon Entropy of the high-resolution time delta derived from the block device I/O events, it is clear that the entropy heuristic of the LRNG significantly underestimates the available entropy. This can be interpreted as a large safety margin being present and that the VirtualBox VMM operation does not adversely affecting the usability of the block device noise sources.

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 50 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 3.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.634109	N/A[39]
Bins	3.841678	6.521497
Collection	3.775395	6.659401
Compression	3.573211	6.464542
Collision	3.605001	6.395537

Table 3: Minimum Entropy according to SP800-90B

The results in the table show that:

39 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

Another result from figure 25 is that the VirtualBox VMM does not affect the LRNG heuristic to a degree that it becomes unsuitable for the LRNG operation.

5.3.3.3 VMM Buffer Cache Enabled

The testing conducted for the disabled VMM buffer cache is repeated with the following virtual machine configuration:

- Enabling of the option to use the host cache for block devices.

The dd command as well as the resulting hard disk LED behavior is identical to the testing with the VMM buffer cache disabled as documented in section 5.3.3.2.

5.3.3.3.1 Block Device Numerical Identifier

The test result for the block device identifier is unchanged compared to section 5.3.3.2.1 so the same conclusion can be drawn: the identifier does not deliver entropy.

5.3.3.3.2 Jiffies Delta of Block Device I/O Events

The plot of the Jiffies delta for block device I/O events is depicted in figure 26. The diagram shows a very similar behavior to the KVM result when the VMM buffer cache is enabled given in section 5.3.2.3.2. The Shannon Entropy is significantly lower than for the same testing on KVM. Yet it is so low that the conclusion from KVM still holds: the Jiffies time stamp obtained for a VirtualBox guest with VMM buffer cache enabled hardly provides any entropy for the block device noise source.

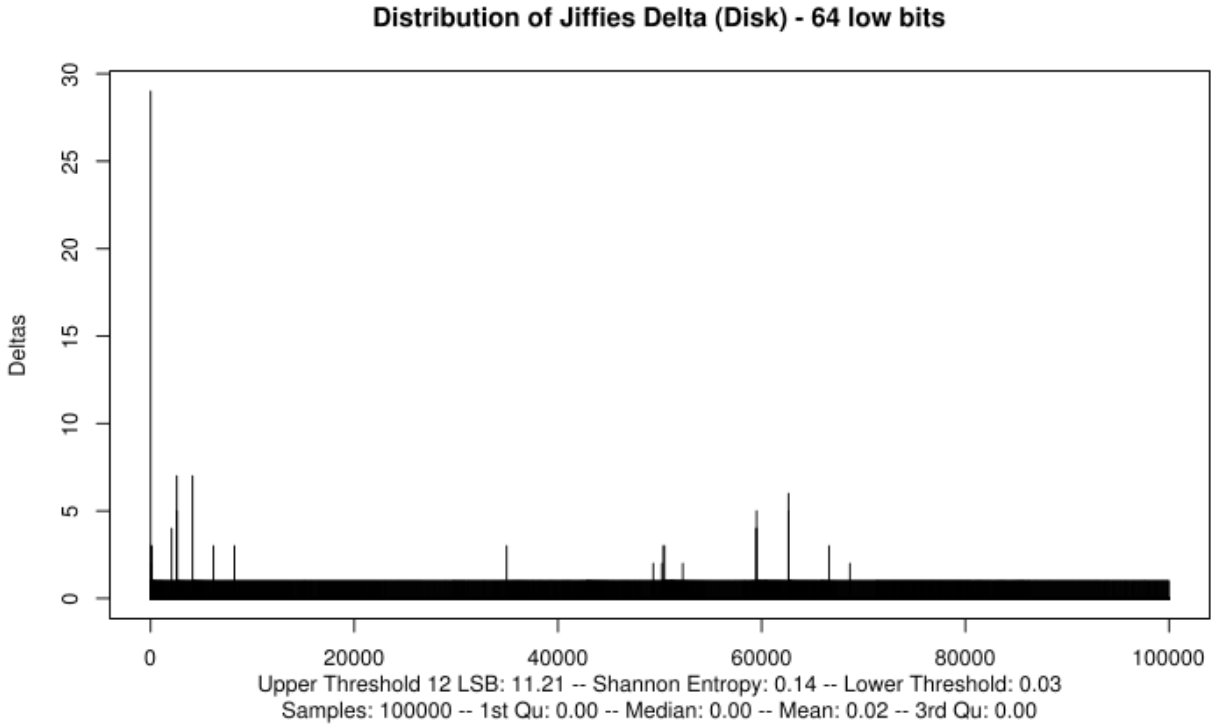

Figure 26: Distribution of Jiffies Delta for Block Devices – VirtualBox With Buffer Cache

The conclusion about the Jiffies delta is supported by figure 27 showing the histogram of the delta values. The histogram shows that almost all delta values are zero.

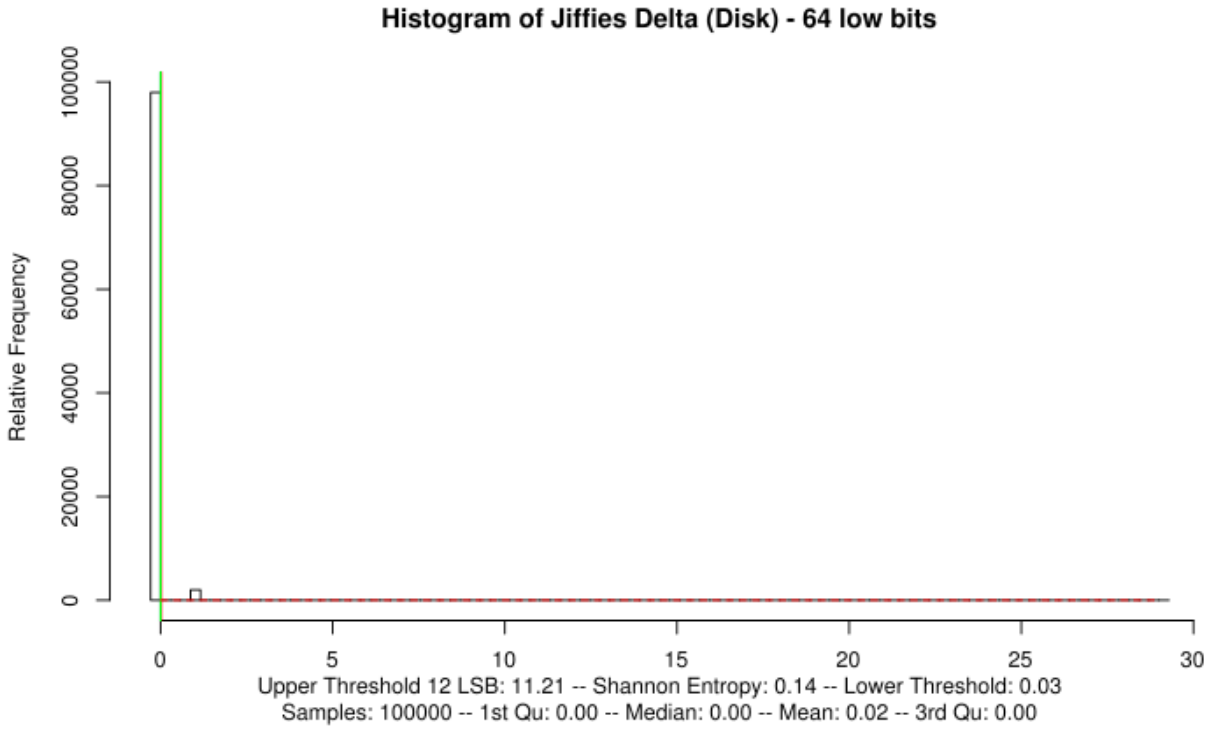

Figure 27: Histogram of Jiffies Delta for Block Device – VirtualBox With Buffer Cache

5.3.3.3.3 High-Resolution Time Delta of Block Device I/O Events

Just like in the previous test rounds, the last analysis applies to the high-resolution timer for block device I/O events. Figures 28 and 29 provide the data plot and the histogram of the test results using the 22 low bits as used in the aforementioned high-resolution time delta analyses.

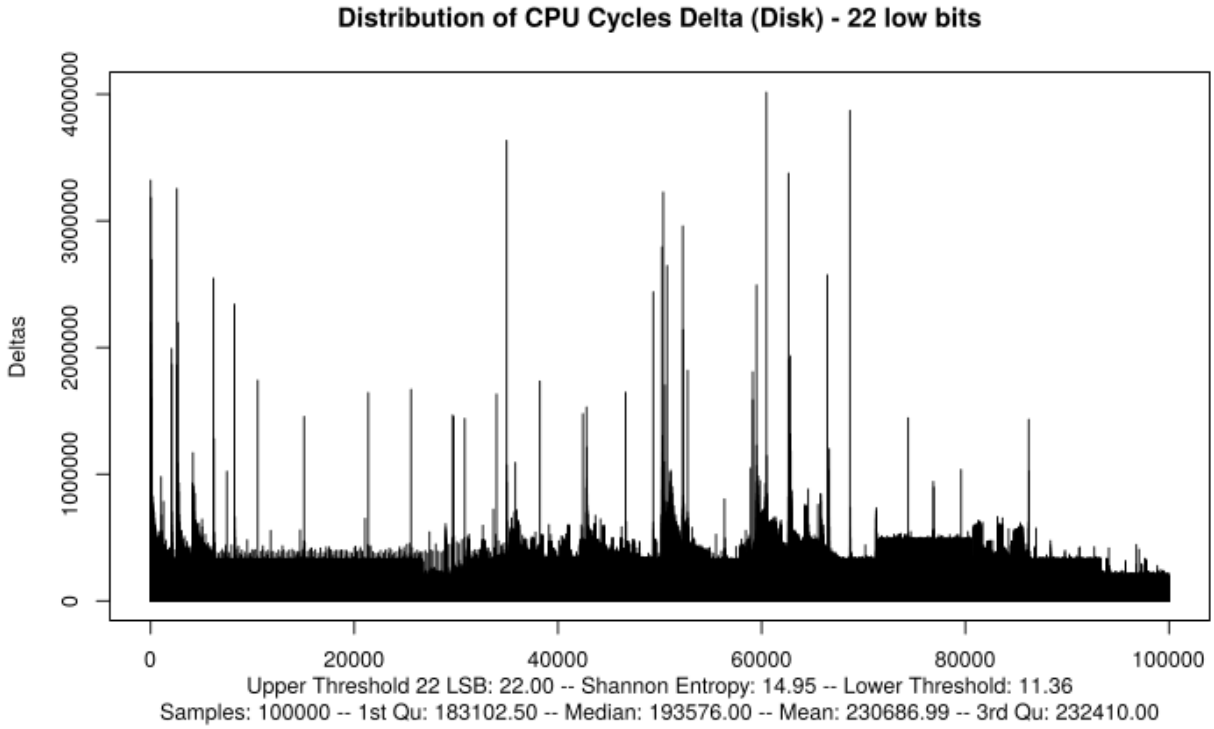

Figure 28: Distribution of High-Resolution Time Deltas for Block Device – 22 low bits – VirtualBox With Buffer Cache

Both figures are very similar to the high-resolution time deltas obtained for the VirtualBox VMM without buffer cache given in section 5.3.3.2.3. The oscillation of the time delta is in the same range of 300,000 to 500,000 clock ticks. The number of spikes are a bit lower which may result from less re-scheduling events by the VMM. Due to the less number of spikes, the Shannon Entropy value is slightly lower than in section 5.3.3.2.3.

In general, however, the measurements for both, VirtualBox with and without buffer cache are strikingly similar. This conclusion is yet another hint similar to the one stated above that VirtualBox must have another cache for handling block device I/O.

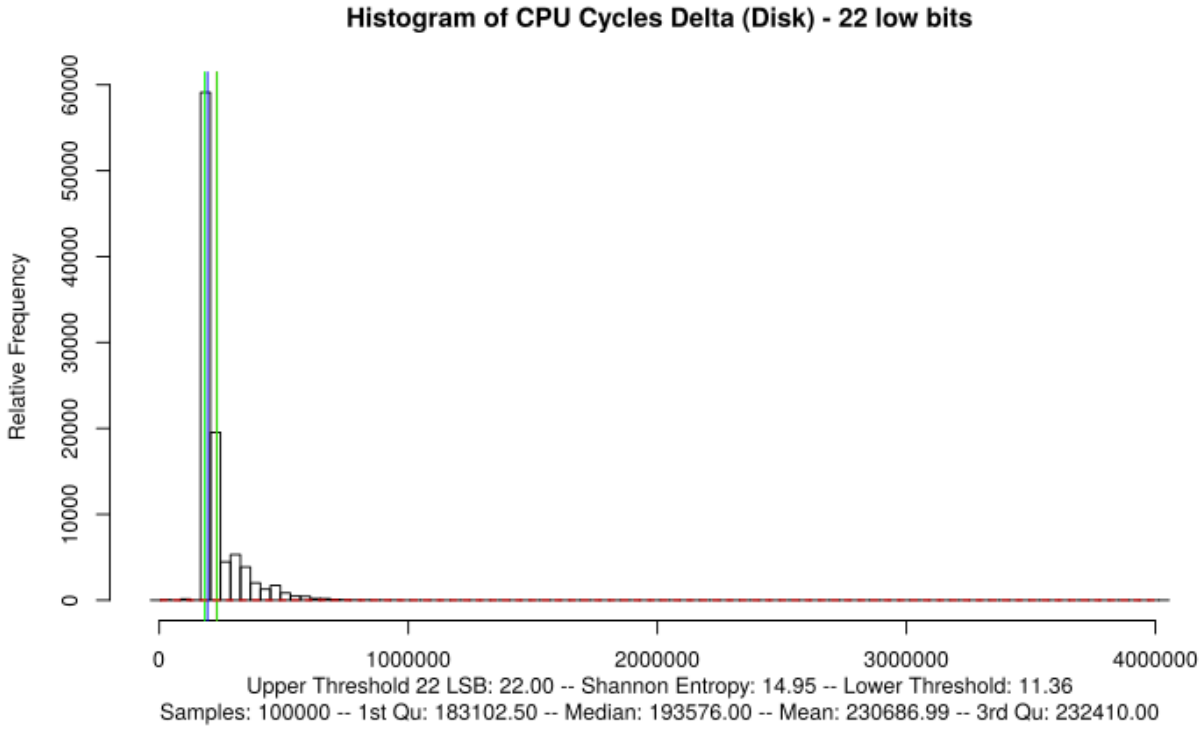

Figure 29: Histogram of High-Resolution Time Deltas for Block Device – 22 low bits – VirtualBox With Buffer Cache

Yet, the variations shown with the Shannon Entropy value and considering the LRNG entropy heuristic which applies zero bits of entropy to the majority of the block device I/O events indicate that the LRNG block device noise source operation is still effective when executing as a guest under VirtualBox.

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 50 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 4.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.588522	N/A[40]
Bins	3.842298	6.512249
Collection	3.004145	5.752238
Compression	3.045804	5.655808
Collision	3.101399	5.605563

Table 4: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.

40 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

5.3.3.4 Conclusion: Configuration Requirement

To ensure sufficient entropy it is suggested that the backend of the block device offered to a virtual machine must be stored on the hard disk with spinning disks (i.e. no SSDs) accessed by the VMM. This is done by either use a file as a storage backend residing on a HDD with spinning disks for the guest block device or by assigning a partition on a block device with spinning disks to the guest.

5.3.4 Microsoft Hyper-V

Microsoft Hyper-V does not provide configuration options to tune the block device behavior of the guest. The test virtual machine operates with the default block device support.

The block device is emulated as a VMBus device for the virtual machine. The following listing shows that the block device is used as noise source by the LRNG.

```
cat
/sys/devices/LNXSYSTM:00/LNXSYBUS:00/PNP0A03:00/device:07/VMBUS:01/vmbus_1/host2/t
arget2:0:0/2:0:0:0/block/sda/queue/add_random
1
```

When applying the dd command during the execution of the SystemTap testing, the following results are recorded:

- The hard disk LED is lit for the entire time of the execution of the dd command.
- The read/write speed of the dd command is a bit less than 2 MBytes per second.

Both observations are very similar to the behavior seen on KVM with VMM buffer cache disabled. As no documentation on Hyper-V handling of the block devices for guests is found, the dd test results provide a strong indication that the Hyper-V VMM does not use a buffer cache.

5.3.4.1 Block Device I/O Event Testing

5.3.4.1.1 Block Device Numerical Identifier

As expected, the test results show that the block device identifier is always identical for all I/O operations. This allows to conclude that the identifier does not deliver entropy.

5.3.4.1.2 Jiffies Delta of Block Device I/O Events

The execution of the SystemTap testing returned the results for the Jiffies delta as depicted in figure 30 for the 100,000 measurements made. The solid bar at the bottom of the data plot is in fact an irregular oscillation between zero and one – the majority of the values are zero. Very few spikes are present which indicate delays due to scheduling events or because the VMM itself requested some block device I/O operations.

The Shannon Entropy value is very low indicating that the Jiffies deltas hardly contribute to the entropy harvested by the LRNG block device noise source.

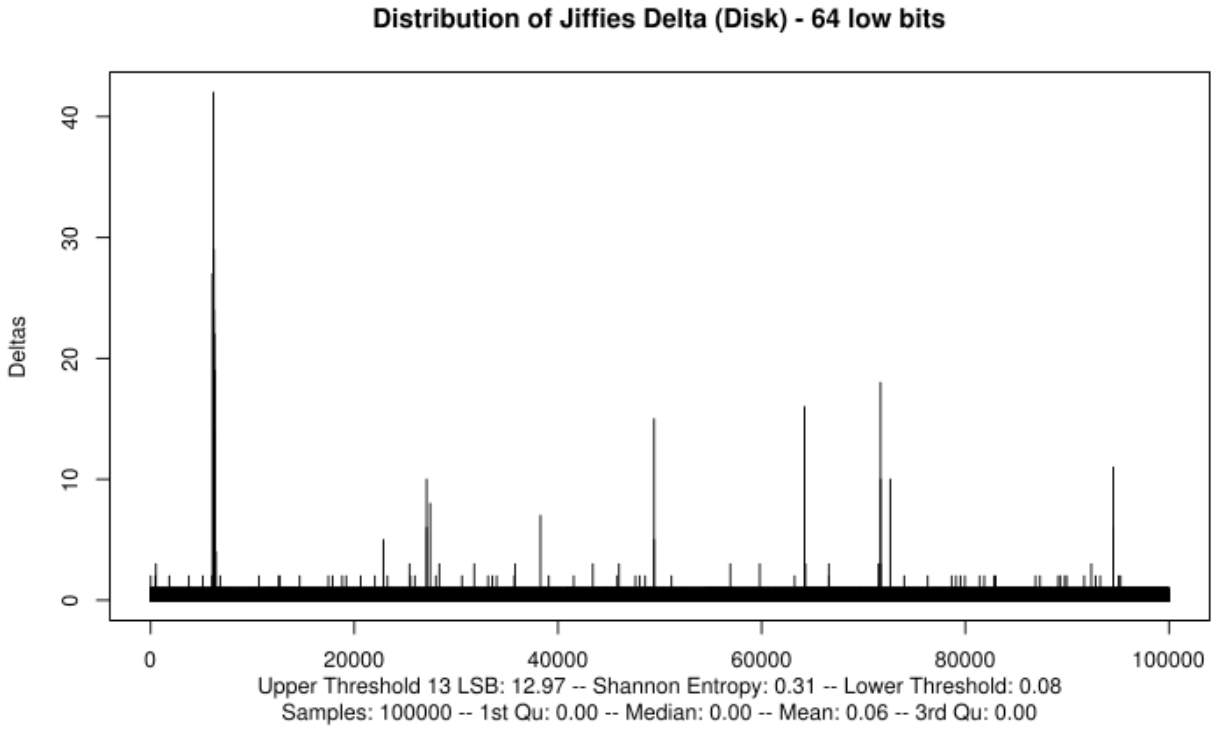

Figure 30: Distribution of Jiffies Delta for Block Devices – Hyper-V

The observations from the data plot are supported by the histogram given in figure 31. This histogram shows that the vast majority of Jiffies delta is zero.

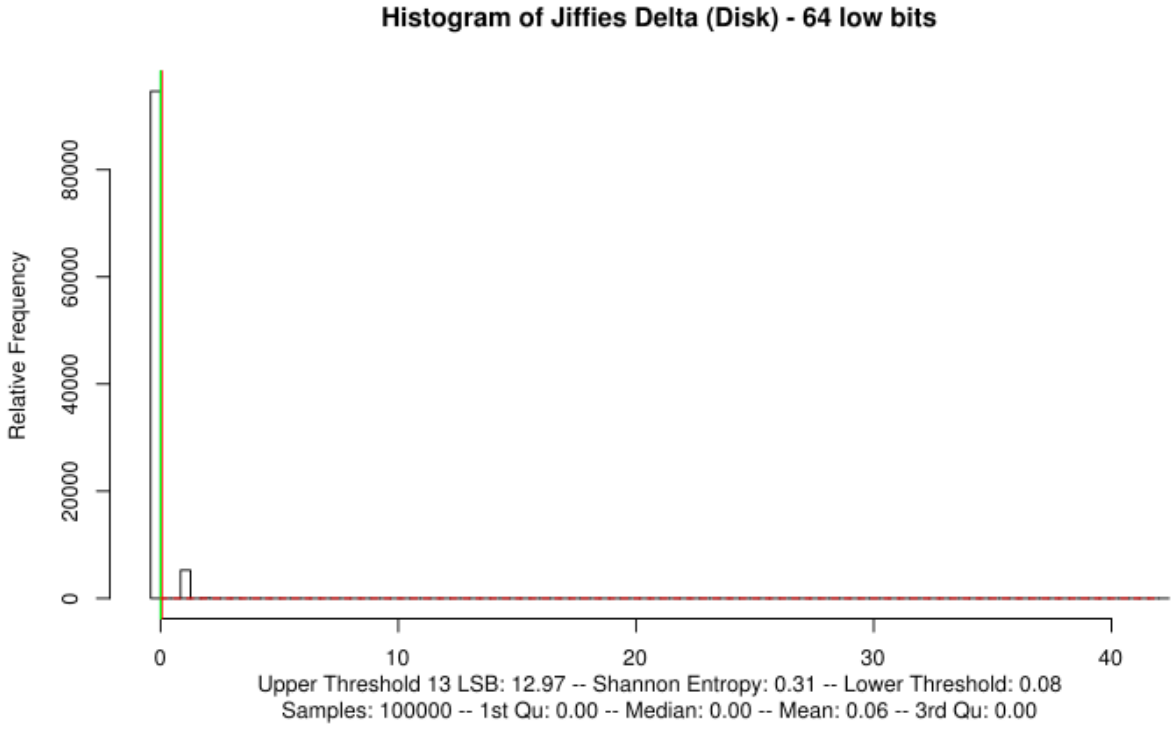

Figure 31: Histogram of Jiffies Delta for Block Device – Hyper-V

5.3.4.1.3 High-Resolution Time Delta of Block Device I/O Events

The recorded high-resolution time stamps for block device I/O events are analyzed in this section. The obtained high-resolution time deltas show very high delta spikes in regular intervals. As outlined for KVM and VirtualBox, those spikes indicate the wrap-around of the high-resolution time stamp and are, therefore, an indication that the testing worked properly.

To present graphs that can be analyzed, the high-order bits are removed. Just like for the previous high-resolution time delta analyses, only the low 22 bits are used to generate the graphs in figures 32 and 33.

The data plot in figure 32 shows that the majority of high-resolution time deltas oscillate between 500,000 and 1.5 million clock ticks. The calculated Shannon Entropy value of more than 15 bits indicates a fluctuation of the time deltas, which is significantly higher than the cap of 11 bits applied with the heuristic entropy calculation of the LRNG.

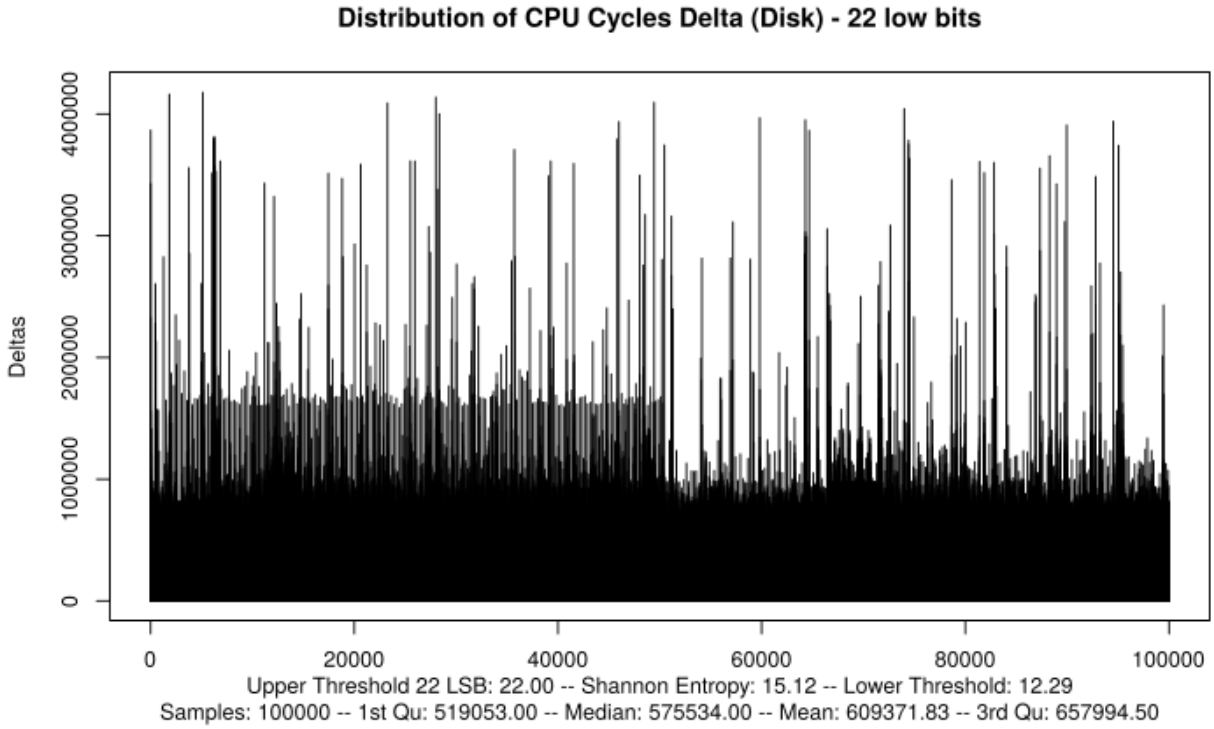

Figure 32: Distribution of High-Resolution Time Delta for Block Devices – 22 low bits – Hyper-V

The histogram in figure 33 confirms the conclusions drawn for the data plot.

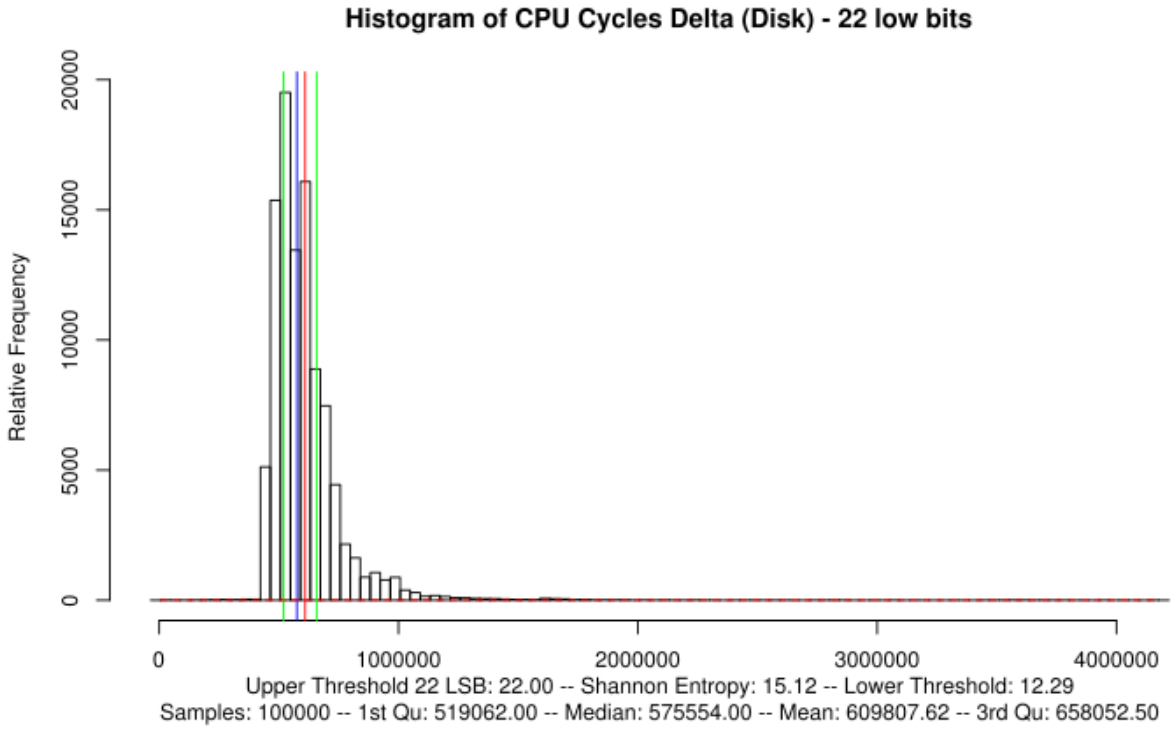

Figure 33: Histogram of High-Resolution Time Deltas for Block Devices – 22 low bits – Hyper-V

In an additional testing, the LRNG entropy heuristic is measured to obtain the entropy values applied to the different block device I/O events. Figure 34 provides the histogram of the 12 entropy values that can be awarded to an event – 0 bits through 11 bits. The graph illustrates that more than 80% of all events are processed to contain zero bits of entropy. The average entropy applied to the block device events is 0.17 bits of entropy. When comparing this value with the measured Shannon Entropy value of more than 15 bits for the high-resolution time stamp block device events, it becomes evident that the entropy is significantly underestimated.

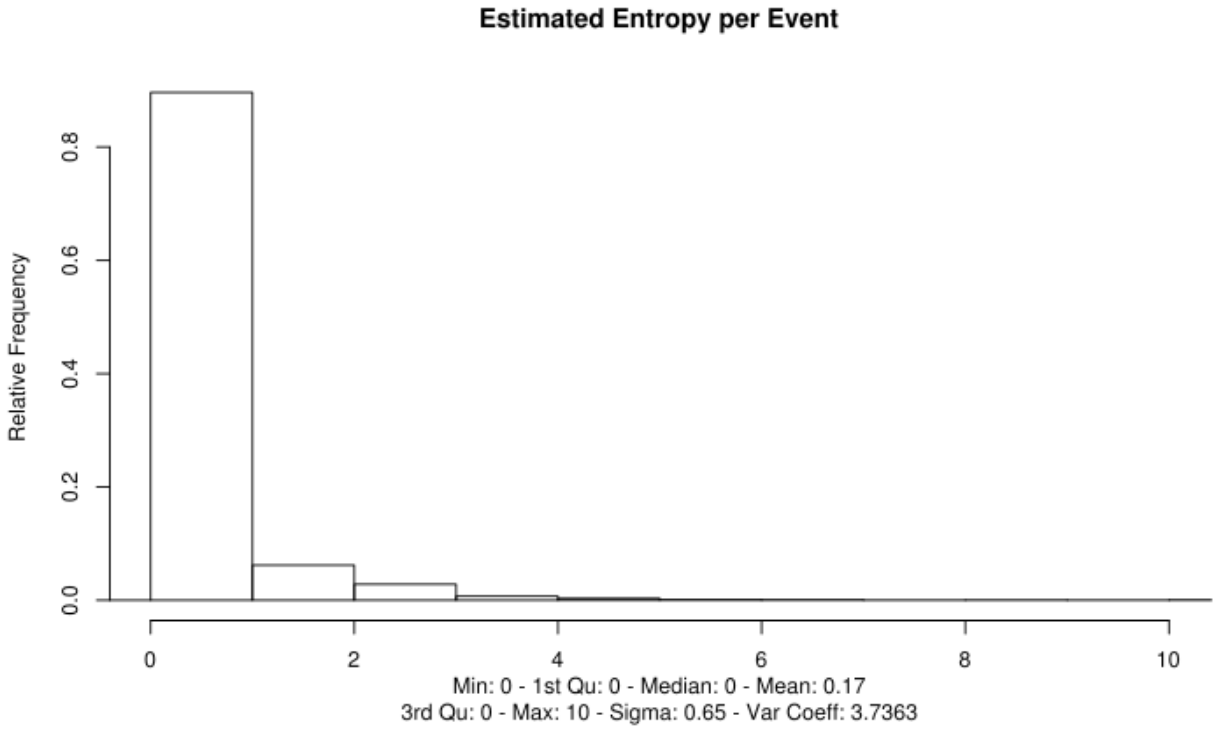

Figure 34: LRNG Entropy Estimation for Block Device Events – Hyper-V

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 50 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 5.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.626499	N/A[41]
Bins	3.845406	6.503060
Collection	3.617106	6.810880
Compression	3.423988	6.681457
Collision	3.700506	6.638223

Table 5: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

This result allows the conclusion that the block device noise source of the LRNG operates appropriately within a Hyper-V guest.

41 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

5.3.4.2 Conclusion: Configuration Requirement

With the test results obtained for Hyper-V, the same configuration constraints listed in section 5.3.3.4 are applicable.

5.3.5 VMWare ESXi

Similarly to Microsoft's Hyper-V, VMWare ESXi does not provide configuration options to influence the guest block device handling. Therefore, the test virtual machine again operates with the default block device support.

The block device visible by the guest is an emulated SATA disk as visible with the following listing. That listing also indicates that the block device is used as a noise source by the guest LRNG.

```
cat
/sys/devices/pci0000:00/0000:00:10.0/host2/target2:0:0/2:0:0:0/block/sda/queue/add
_random
1
```

The use of the dd command during testing shows the following results:

- The hard disk LED is lit for the entire execution time of the dd command.
- The read / write speed of the dd command is about 2.5 MBytes per second.

Both results are similar to the behavior seen on KVM with VMM buffer cache disabled or Hyper-V. Without documentation around the VMWare handling of the block devices for guests, the authors conclude that the dd test results provide a strong indication that the ESXi VMM does not use a buffer cache.

5.3.5.1 Block Device I/O Event Testing

5.3.5.1.1 Block Device Numerical Identifier

Again as expected, the test results show that the block device identifier is always identical for all I/O operations. This allows to conclude that the identifier does not deliver entropy.

5.3.5.1.2 Jiffies Delta of Block Device I/O Events

The measurements of the data recorded by the LRNG for block device I/O events include the data for Jiffies delta, a plot of which is given in figure 35. The plot is remarkably different to the other VMMs: there is hardly any spike. It is assumed that the ESXi VMM tries to access the block device itself much less often than the host systems of KVM, VirtualBox or Hyper-V. For those three host systems, the reader should note that the host environments implement full general purpose computing environments whereas VMWare ESXi is a VMM that is solely dedicated to act as a hypervisor.

Nonetheless, the plot shows the solid bar which in fact is an irregular oscillation between zero and one when magnifying the graph – the majority of the values are zero. The Shannon Entropy value indicates that only very little entropy is present in the Jiffies delta values which allows the conclusion that again Jiffies do not provide a significant entropy contribution to the LRNG block device noise source.

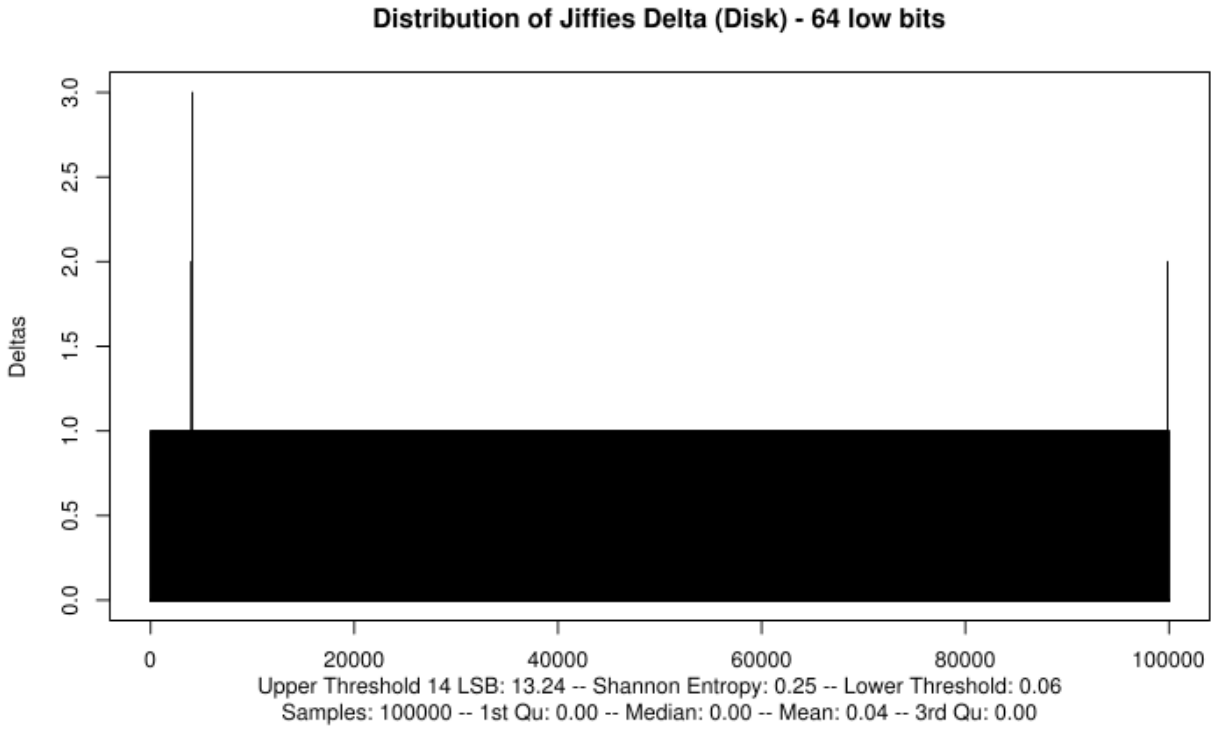

Figure 35: Distribution of Jiffies Delta for Block Device – ESXi

The support for the conclusion is presented with the histogram of the Jiffies delta given in figure 36. This graph shows, as expected, that more than 80% of all Jiffies delta values are zero.

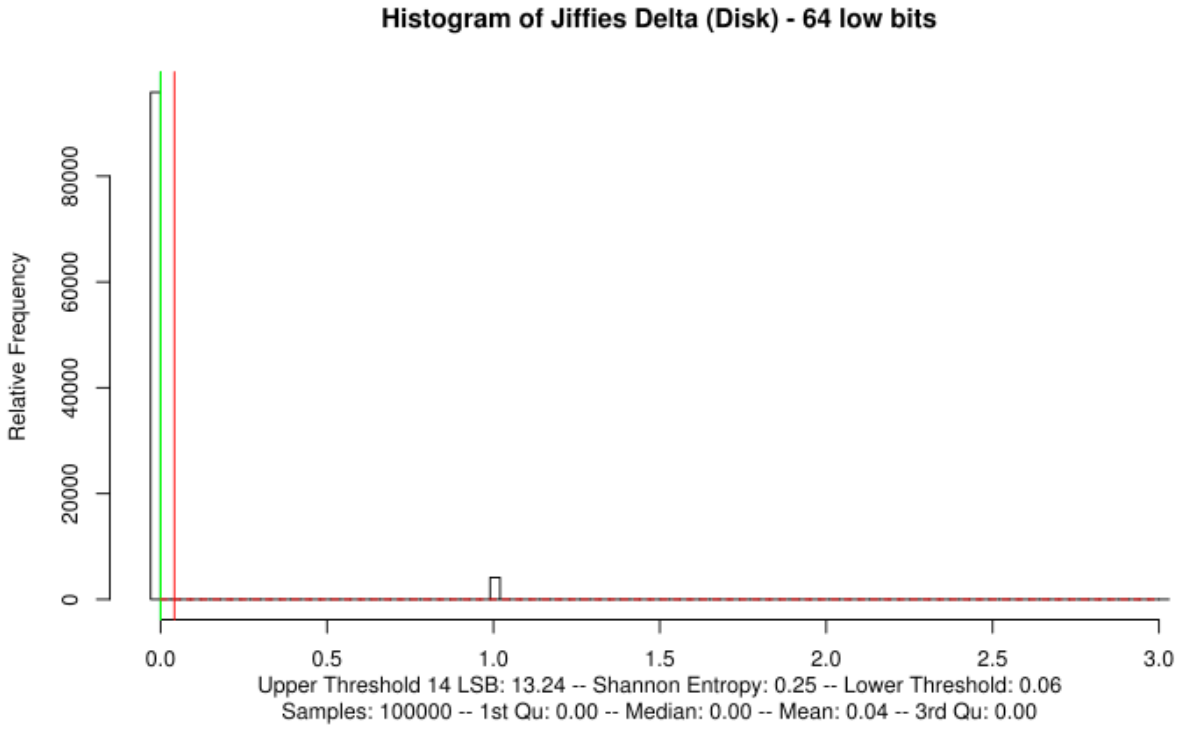

Figure 36: Histogram of Jiffies Delta for Block Device – ESXi

5.3.5.1.3 High-Resolution Time Delta of Block Device I/O Events

The high-resolution time deltas obtained during testing when block device I/O events are processed by the LRNG are depicted in figures 37 and 38. With the same argument as already outlined for the other VMM result discussions, the high-resolution time stamp analysis focuses on the 22 low bits.

The data plot shows a fluctuation of the high-resolution time stamp between 400,000 and 700,000 clock cycles. A number of spikes are visible which are assumed to originate in re-scheduling events.

The high-resolution time delta data set has a calculated Shannon Entropy more than 13.6 bits.

The data set is presented as a histogram in figure 38, which confirms the conclusions drawn from the data plot.

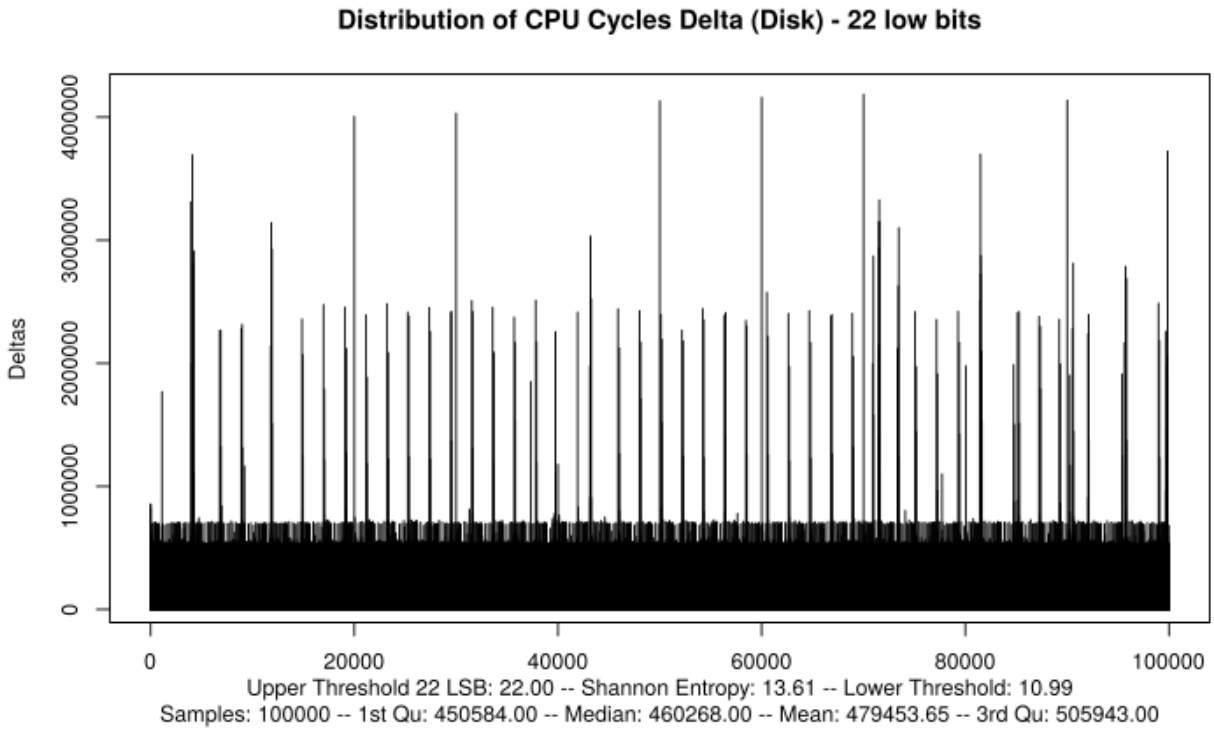

Figure 37: Distribution of High-Resolution Time Deltas for Block Device – 22 low bits – ESXi

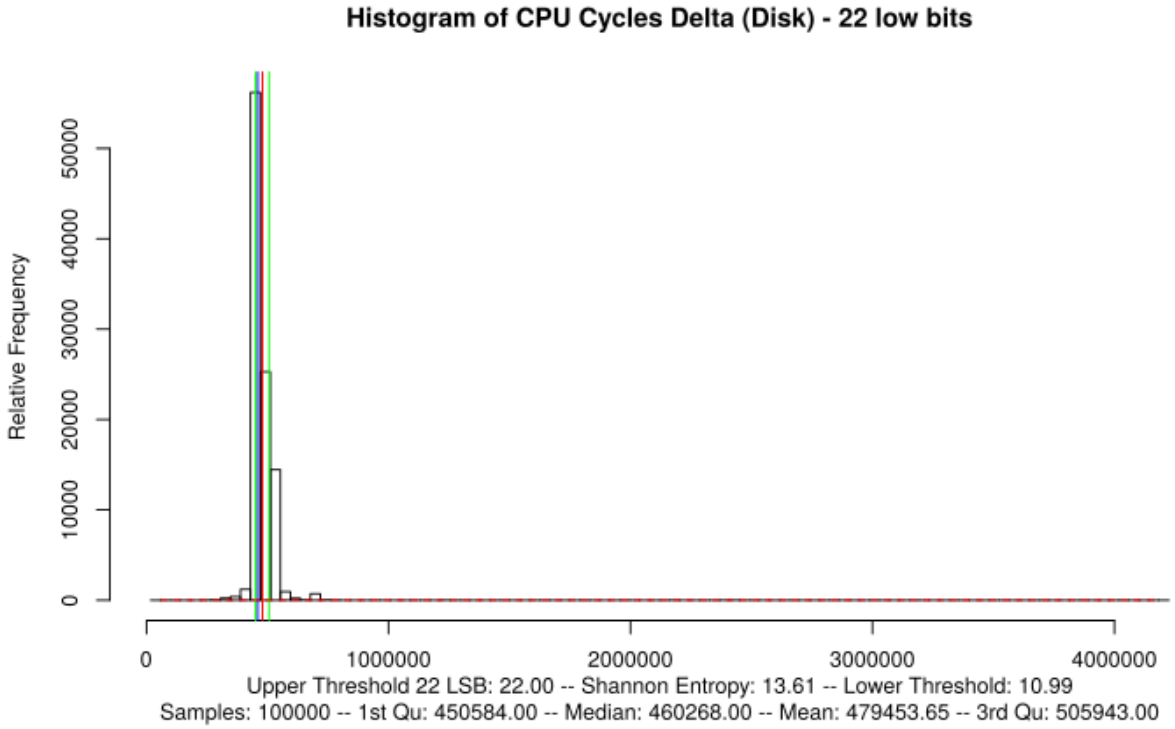

Figure 38: Histogram of High-Resolution Time Deltas for Block Device – 22 low bits – ESXi

Just like for the other VMMs, the LRNG's heuristic entropy estimation per block device I/O event is recorded. Figure 39 presents the histogram of the estimation. More than 95% of all block device I/O events are processed to have zero bits of entropy. When comparing this estimate with the measured variations of the high-resolution time delta and its Shannon Entropy value, the conclusion can be drawn that again, the LRNG significantly underestimates the available entropy.

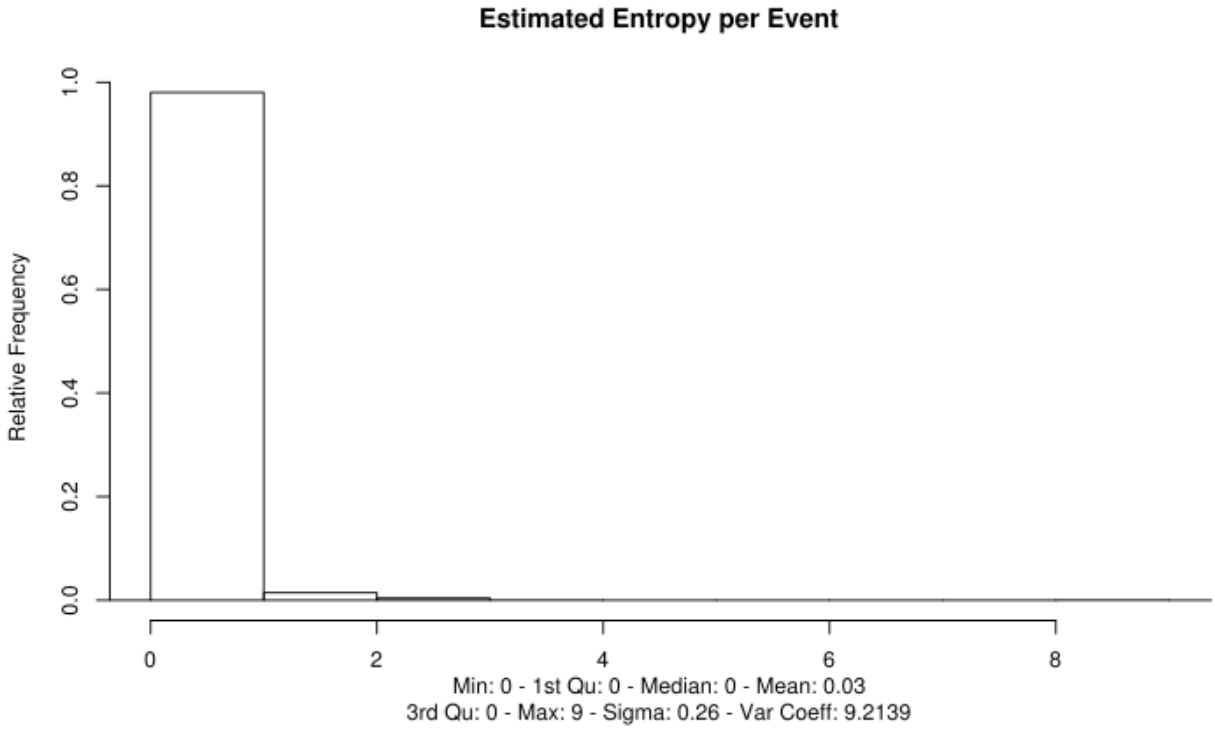

Figure 39: LRNG Heuristic Entropy Estimation for Block Device – ESXi

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 44 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 6.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.597241	N/A[42]
Bins	3.858948	6.500445
Collection	3.683436	6.442920
Compression	3.489236	6.242428
Collision	3.549820	5.744722

Table 6: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

In addition, a separate conclusion can be reached that the LRNG entropy heuristic is not adversely affected by the VMM operation.

42 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

Considering both results, the final conclusion can be reached that the VMWare ESXi hypervisor does not adversely affect the LRNG block device noise source.

5.3.5.2 Conclusion: Configuration Requirement

The testing and its results obtained for VMWare ESXi, the configuration constraints specified in section 5.3.3.4 are applicable as well.

5.4 LRNG HID Noise Source Testing

For receiving HID events, the LRNG registers the callback function `add_input_randomness` with the input event layer of the Linux kernel. This layer commonly triggered with all different kinds of input devices, such as mice or keyboards. Interestingly, testing showed that keyboards or mice attached to the system via the USB bus are not picked up by the LRNG HID callback function.

The LRNG HID callback function marks the entry point for its HID noise source. This callback function is always invoked for each HID event processed by this layer. In terms of virtual machines, human interface devices are the keyboard and mouse identified by the guest. Such HID are only provided with the console support provided by the VMM. These virtual machine consoles are commonly externalized via RDP or VNC network protocols but are also directly accessible on the same host when using KVM/QEMU, VirtualBox or Hyper-V.

Though, the reader should consider the big picture of virtual machine usage. Virtual machines are commonly used as server systems whose services are made available via network applications. It is uncommon that administrators of VMMs give access to the console of a guest, just like it is uncommon that administrators give access to the console of bare metal servers. When keeping this finding in mind, the LRNG HID noise source will hardly be used in common use cases of virtual machines. This may be different, of course, in case the VMM environment is used to secure a desktop system and untrusted code shall be separated from trusted code. In such use cases, HID are usually attached to the system and accessible by the guest.

The following tests are all performed by accessing the console and generating HID events. Therefore, these test results should only be considered by the reader when the discussed VMM environments also offers general access to the virtual machine consoles. Otherwise, the reader may disregard this section and assume that the LRNG will not obtain entropy from HID events.

5.4.1 Test Approach

The LRNG HID noise source records the same information for an event as for block devices:

- the HID numerical identifier that triggered the event as a 32 bit integer – this identifier is the key number when the HID event was a key press/release or the movement directions of a mouse.
- a low-resolution time stamp of the event as a 64 bit integer
- a high-resolution time stamp of the event as a 32 bit integer

These variables are defined with the `struct sample` in the function `add_timer_randomness`.

The SystemTap test case is programmed to record this data for each HID event. The SystemTap script raw_entropy_hid.stp implements this test and generates a result file where each of the mentioned value is printed for each HID event. A sample size of 100,000 events is recorded.

Just like for block devices, a second test is implemented that obtains the heuristic entropy estimation applied by the LRNG for one particular HID event. That SystemTap script is called `entropy_per_event_hid.stp` and records the heuristic entropy estimation for each HID event.

To trigger HID operations, the mouse is moved as well as keys on the keyboard are pressed.

In the following subsections the different VMMs are listed on which the tests are executed. Unlike the block device tests, no configuration suggestions are provided. Either the HID

offered by the virtual console are used or not. If the virtual console is not in use, no HID events will be recorded.

5.4.2 KVM/QEMU

The guest managed by KVM operates without the graphical UI provided with X11. Nonetheless, the Linux kernel has the driver with mouse activated which means that the console of the KVM guest picks up keyboard activity as well as mouse activity.

After starting the test script with the guest, the keyboard as well as the mouse are used to obtain the requested sample size of HID events.

5.4.2.1 HID Event Testing

5.4.2.1.1 HID Numerical Identifier

The HID numerical identifier is recorded from the key number of the pressed key as well as the direction identifier of the mouse movement.

For conducting the testing, the mouse is moved to stimulate the event generation, but without any real use case. Keys are also pressed, but again without any particular text to write. 1850 different event IDs were recorded which all have similar probabilities. This shows that the HID numerical identifier recording works in a virtual environment.

However, further conclusions are not derived from that test measurement because the testing cannot be considered to represent a normal use of human interface devices. For example, when considering that the user writes an English text, the used key strokes are not uniformly distributed as some characters are more often used for a local language than others. This property of the used language is not reflected in the test results.

5.4.2.1.2 Jiffies Delta of HID Events

The recording of the Jiffies delta is presented in an equal manner as shown for the block devices above.

The graph in figure 40 shows the data plot of the sampled Jiffies deltas. It shows a high spike which is due to leaving the test system quiet for some time.

The Shannon Entropy value recorded with the data set is given in figure 40 and reaches more than 2.7 bits for one event. This value implies that the Jiffies contribute to the entropy of the HID noise source but its contribution is yet way below the threshold of 11 bits of entropy applied as a cap by the heuristic of the LRNG.

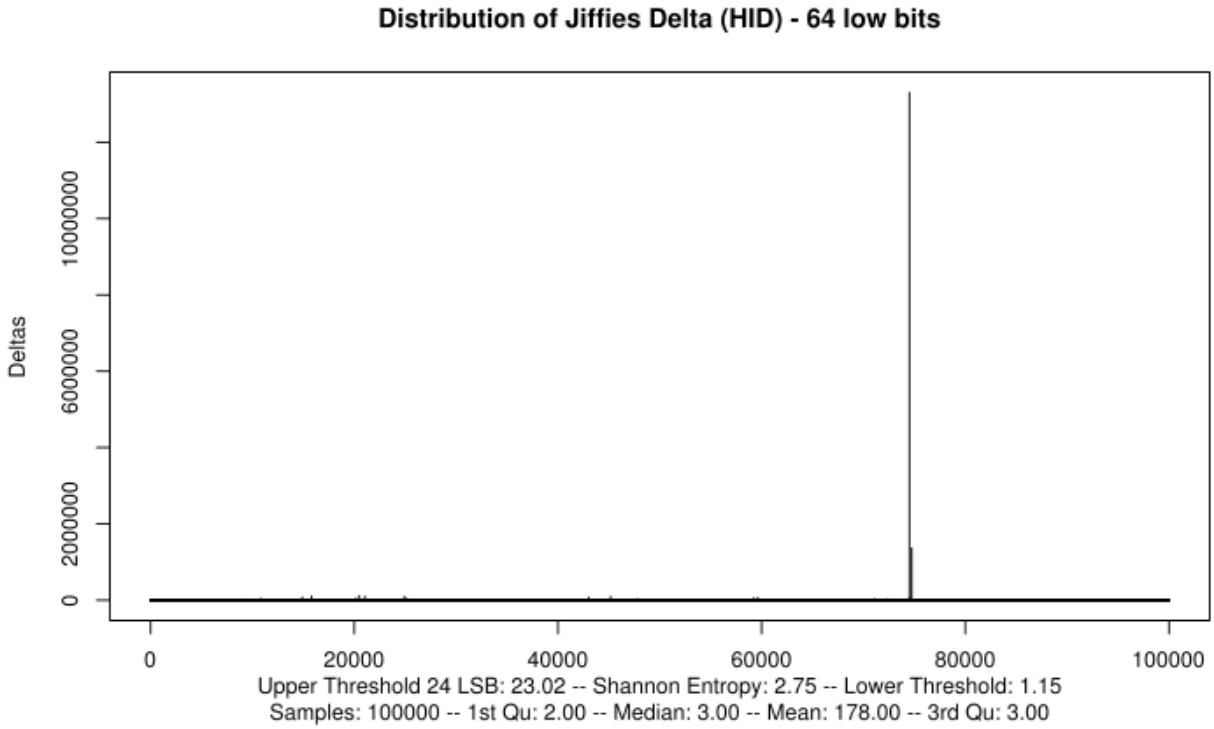

Figure 40: Distribution of Jiffies Delta for HID – 64 low bits – KVM

To allow the reader to gain more insight into the distribution, all bits above 19 bits are removed from the Jiffies delta values. The value of 19 bits is chosen arbitrarily to limit the effect of the spike and yet do not change the resulting data plot significantly. The resulting graph is given with figure 41.

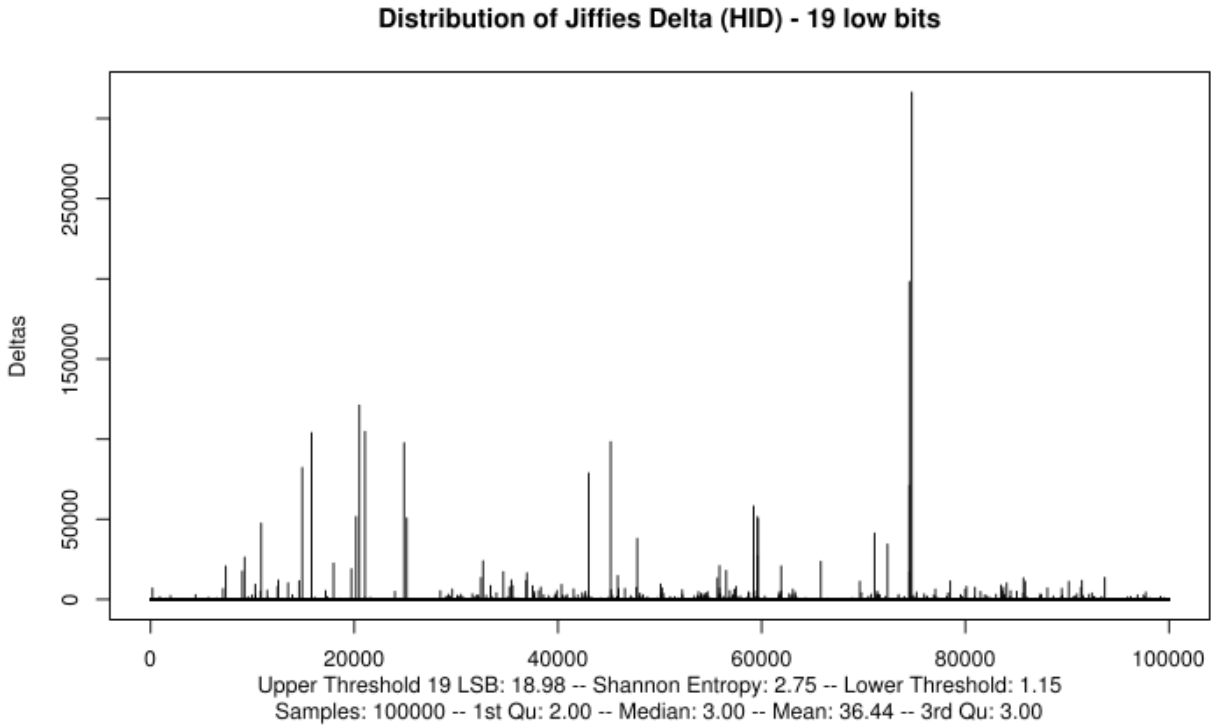

Figure 41: Distribution of Jiffies Delta for HID – 19 low bits – KVM

In addition to the data plots, the histogram of the Jiffies deltas is provided with figure 42. This graph shows that the vast majority of Jiffies deltas is small as also can be seen from the median and mean values.

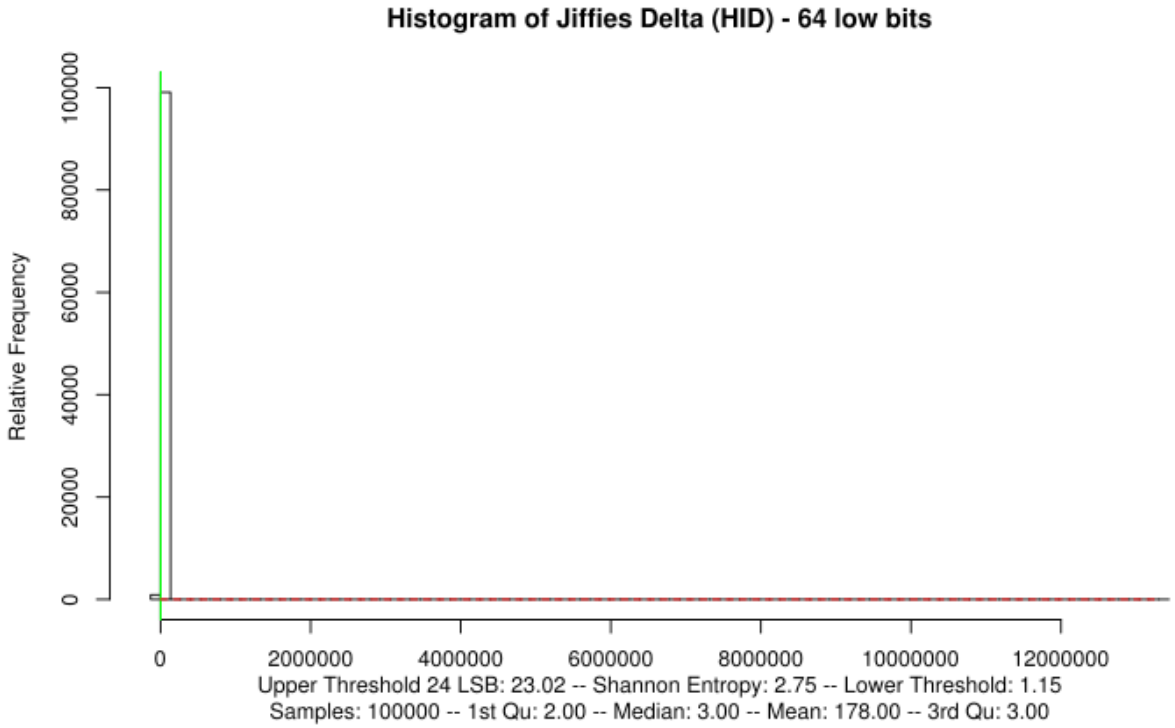

Figure 42: Histogram of Jiffies Delta for HID – KVM

The recorded data shows that in case human interface devices are used, the LRNG records the events to feed its noise source.

5.4.2.1.3 High-Resolution Time Delta of HID Events

In addition to the Jiffies delta, the high-resolution time deltas are obtained. Figure 43 shows the data plot of the high-resolution time deltas demonstrating that the deltas are spread over the possible value range between 0 and 2^{32}.

The Shannon Entropy value of 16.4 bits is close to the theoretically achievable value of $\log_2(100{,}000)$ for the recorded 100,000 events. This result demonstrates that the high-resolution time stamp contributes significantly to the entropy of the LRNG HID noise source.

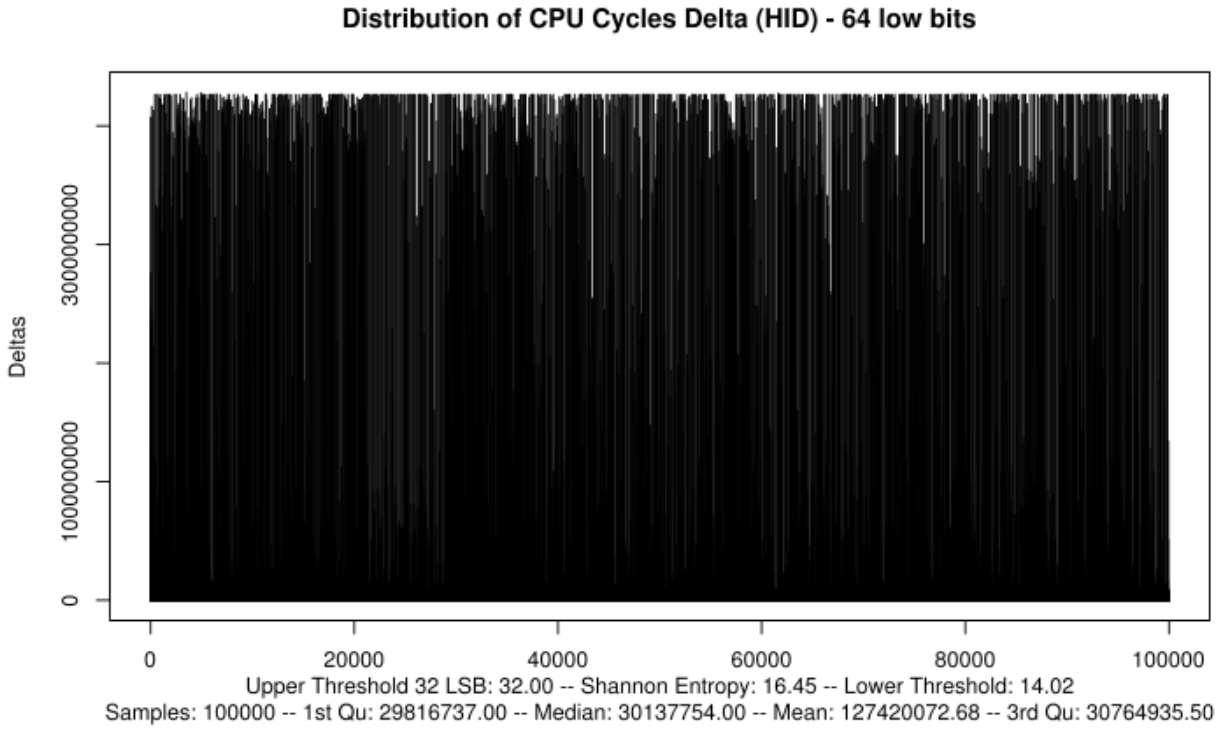

Figure 43: Distribution of High-Resolution Time Delta for HID – KVM

When analyzing the histogram of high-resolution time deltas with figure 44, however, it becomes evident that the majority of time deltas are concentrated in the lower range of the possible values. When “zooming in” to the lower range by discarding the high order bits above 19, the graph given in figure 45 becomes visible. Figure 45 indicates that the time deltas are spread significantly in the lower range of the possible number space.

Nonetheless, the calculated Shannon Entropy of the high-resolution time deltas is significantly above the 11 bits threshold applied by the LRNG entropy heuristic.

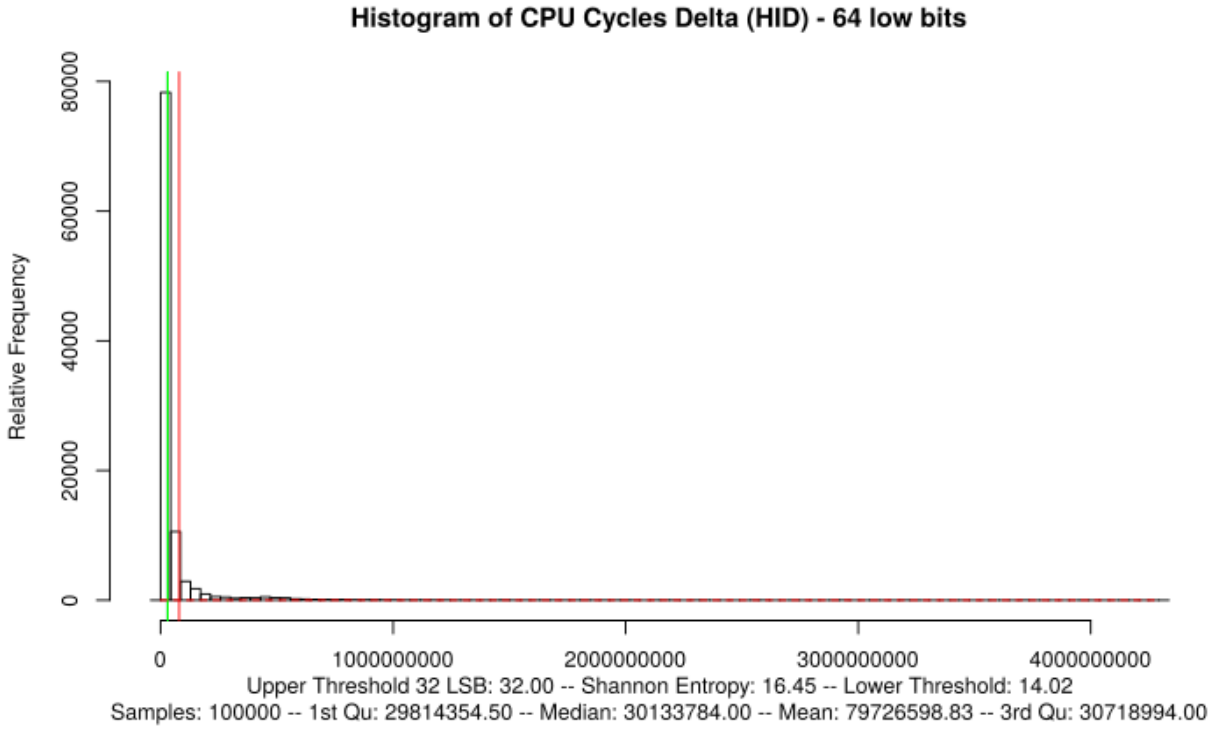

Figure 44: Histogram of high-resolution time deltas for HID – 64 low bits – KVM

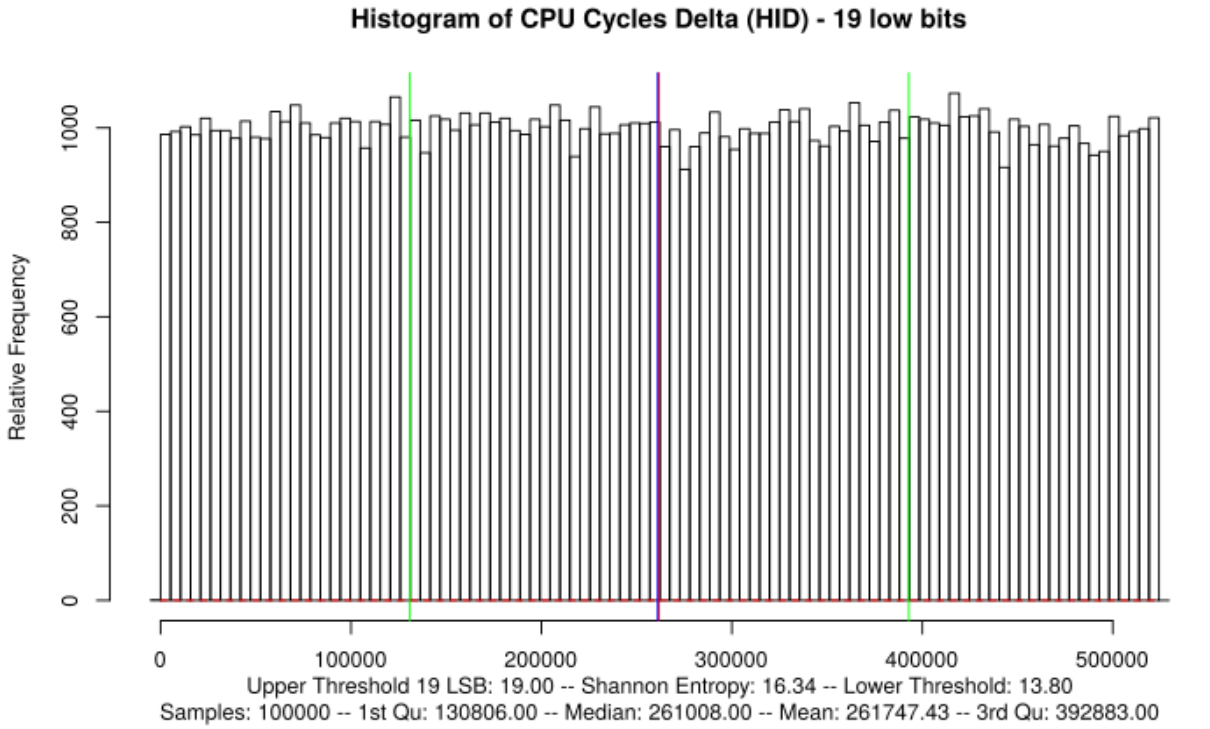

Figure 45: Histogram of High-Resolution Time Deltas for HID -- 19 low bits – KVM

In addition to the measurement of the time deltas, the result of the LRNG entropy heuristic is depicted with figure 46. This graph shows that more than 70% of all HID events are awarded

zero bits of entropy by the heuristic. Only a very limited number of events are processed with an entropy content of four bits or higher.

Comparing this heuristic entropy calculation with the Shannon Entropy value of the high-resolution time stamp plus from the Jiffies, it becomes clear that the LRNG heuristic underestimates the entropy content to a large degree.

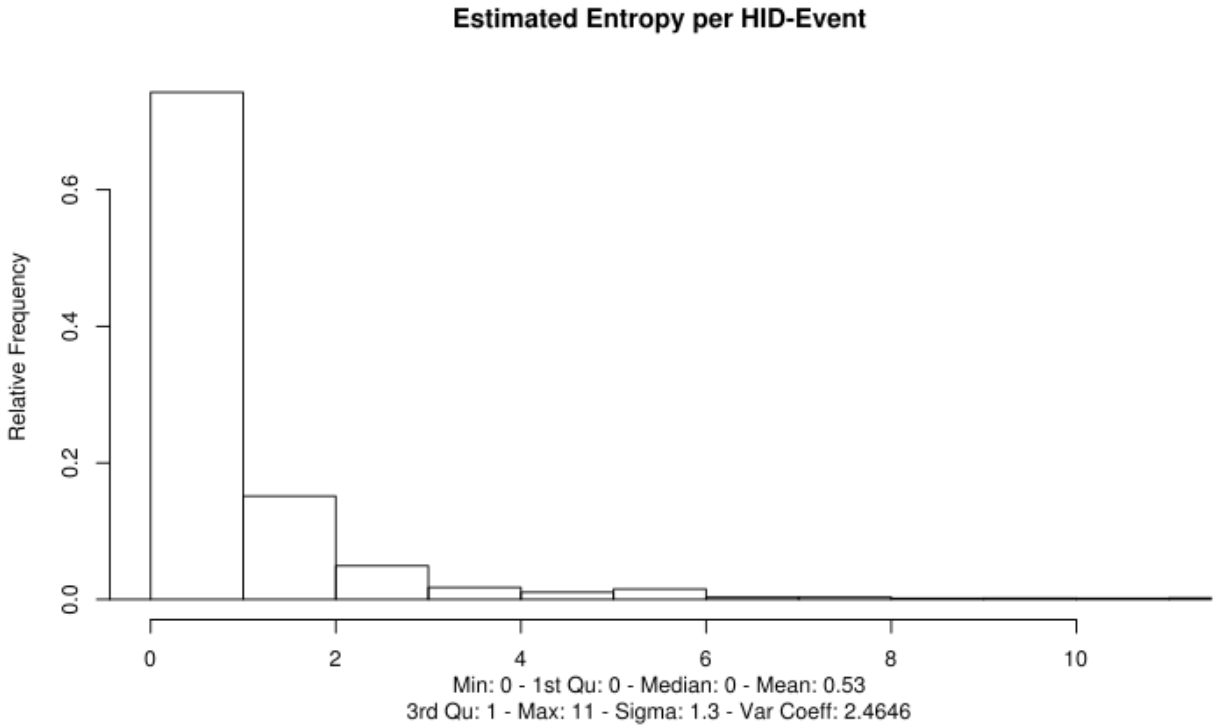

Figure 46: Heuristic Entropy Estimation per HID Event – KVM

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 39 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 7.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.597284	N/A[43]
Bins	3.838785	6.510933
Collection	3.267144	5.884432
Compression	3.213322	5.794970
Collision	3.344693	5.574929

Table 7: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.

43 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

With the underestimation of the available entropy, the conclusion can be drawn that as long as the human interface devices are used with KVM, the VMM impact on this noise source is insignificant.

5.4.2.2 Conclusion: Configuration Requirement

Apart from the requirement that the human interface devices must be used to obtain entropy for the LRNG HID noise source, no special configuration needs to be applied.

5.4.3 Oracle VirtualBox

Just like the KVM virtual machine, the guest operating under VirtualBox is configured without the graphical UI and without X11. However, the Linux kernel used the driver for the HID of mice and keyboards which implies that the VirtualBox console for the guest picks up keyboard activity as well as mouse activity.

5.4.3.1 HID Event Testing

5.4.3.1.1 HID Numerical Identifier

During the HID testing of the guest on VirtualBox, 515 different HID numerical identifiers are recorded. This recording shows that VirtualBox emits HID events when using the guest console. However, as described in section 5.4.2.1.1, these values are not further analyzed as they are not considered to be representative.

5.4.3.1.2 Jiffies Delta of HID Events

With the rationale provided in section 5.4.2.1.2, the data plot of the Jiffies delta for HID events in VirtualBox guests given with figure 47 is limited to the low 19 bits. A few larger deltas are seen which are due to some time of inactivity.

The Shannon Entropy of 2 bits indicates the slight contribution of the Jiffies to the HID noise source operation. Yet, as discussed for the KVM HID event Jiffies in section 5.4.2.1.2, the Jiffies are not sufficient to cover the 11 bit heuristic threshold applied by the LRNG.

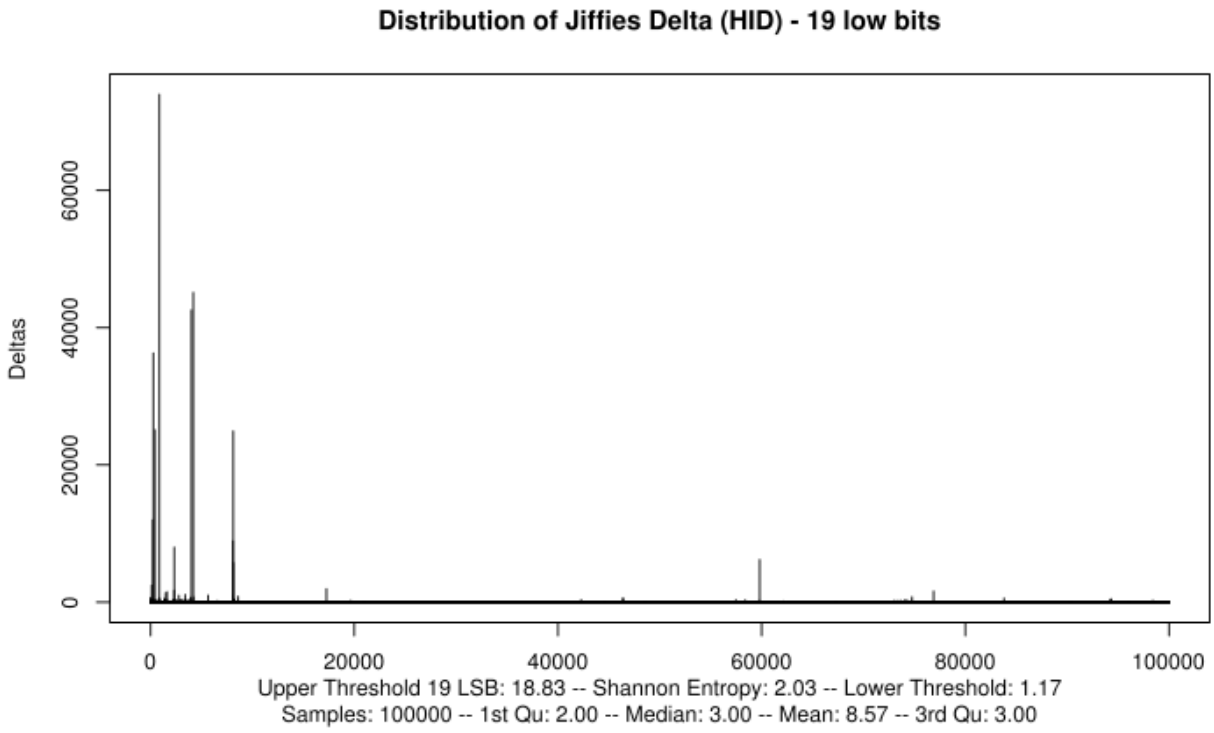

Figure 47: Distribution of Jiffies Delta for HID – 19 low bits – VirtualBox

The histogram of the Jiffies delta given with figure 48 indicates that the vast majority of the Jiffies deltas again are very small.

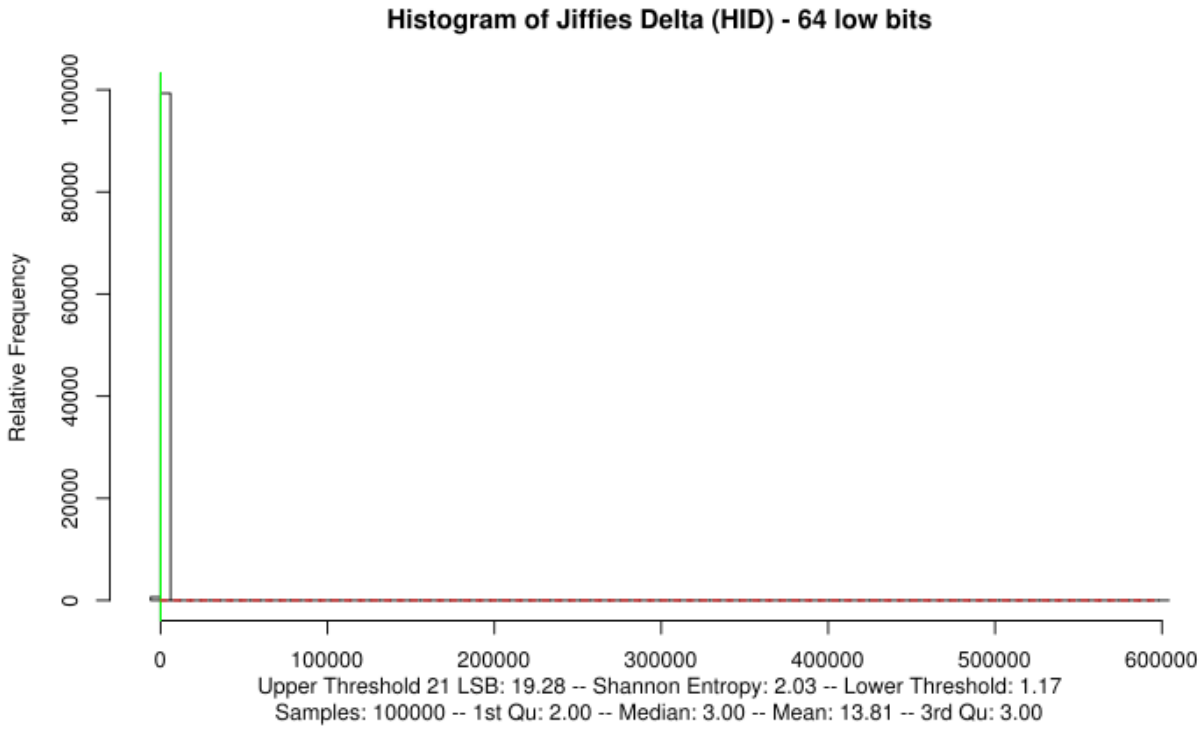

Figure 48: Histogram of Jiffies Delta for HID – VirtualBox

5.4.3.1.3 High-Resolution Time Delta of HID Events

Reviewing the high-resolution time deltas for HID events in addition to the Jiffies, the same picture emerges as identified for KVM already: The high-resolution timer provides more than sufficient uncertainty to keep the LRNG HID noise source operational.

The graph in figure 49 shows the data plot of the high-resolution time deltas of the HID events occurred with the VirtualBox guest. The time delta values fluctuate over the entire possible value space between zero and 2^{32}.

The Shannon Entropy of 16,4 bits is close of the theoretical maximum for 100,000 events and is significantly over the threshold of 11 bits applied by the LRNG entropy heuristic.

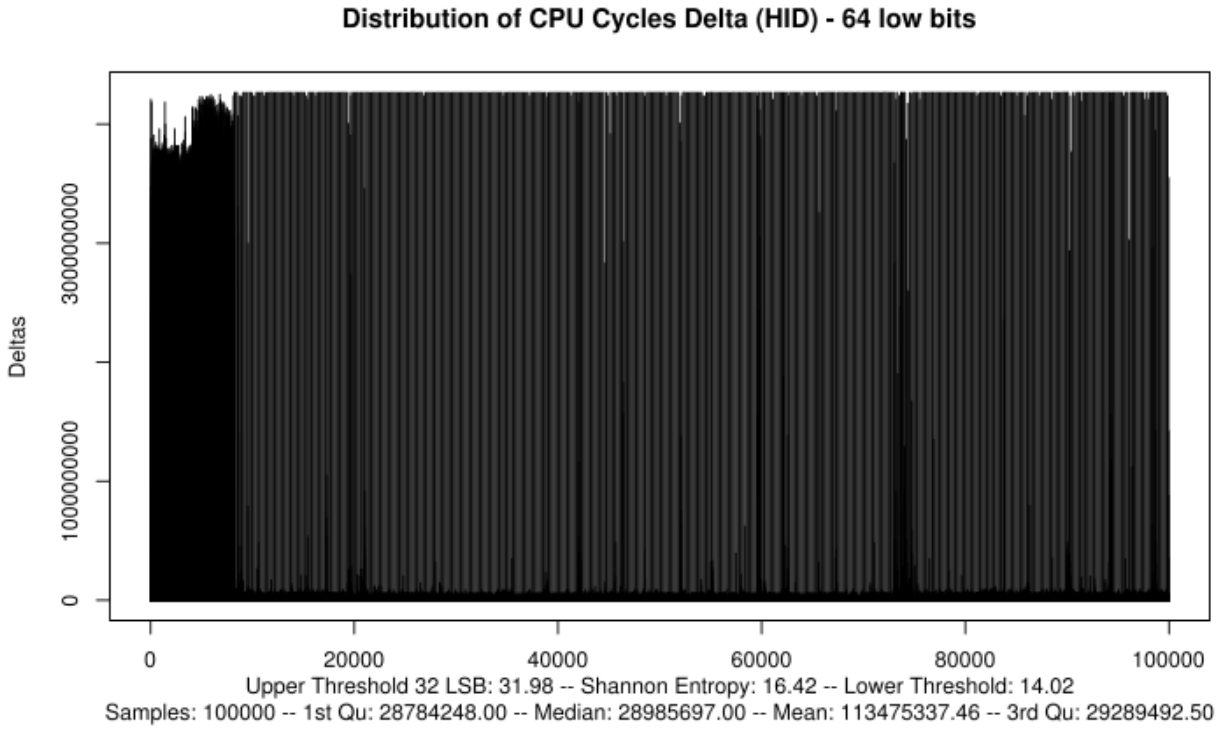

Figure 49: High-Resolution Time Deltas for HID – VirtualBox

The histogram given with figure 50, however, shows that the majority of time deltas are concentrated in the lower range of the possible values. When zooming into this lower value range by discarding all bits higher than 19, the graph given with figure 51 emerges.

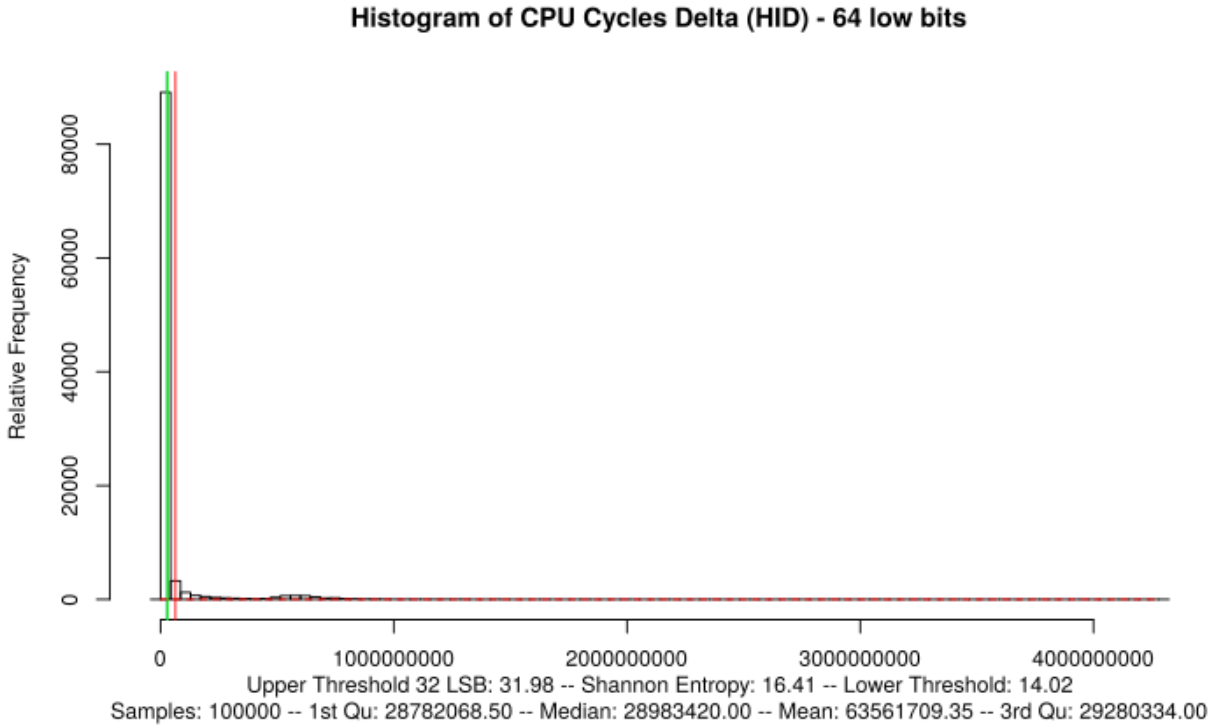

Figure 50: Histogram of High-Resolution Time Deltas for HID – 64 low bits – VirtualBox

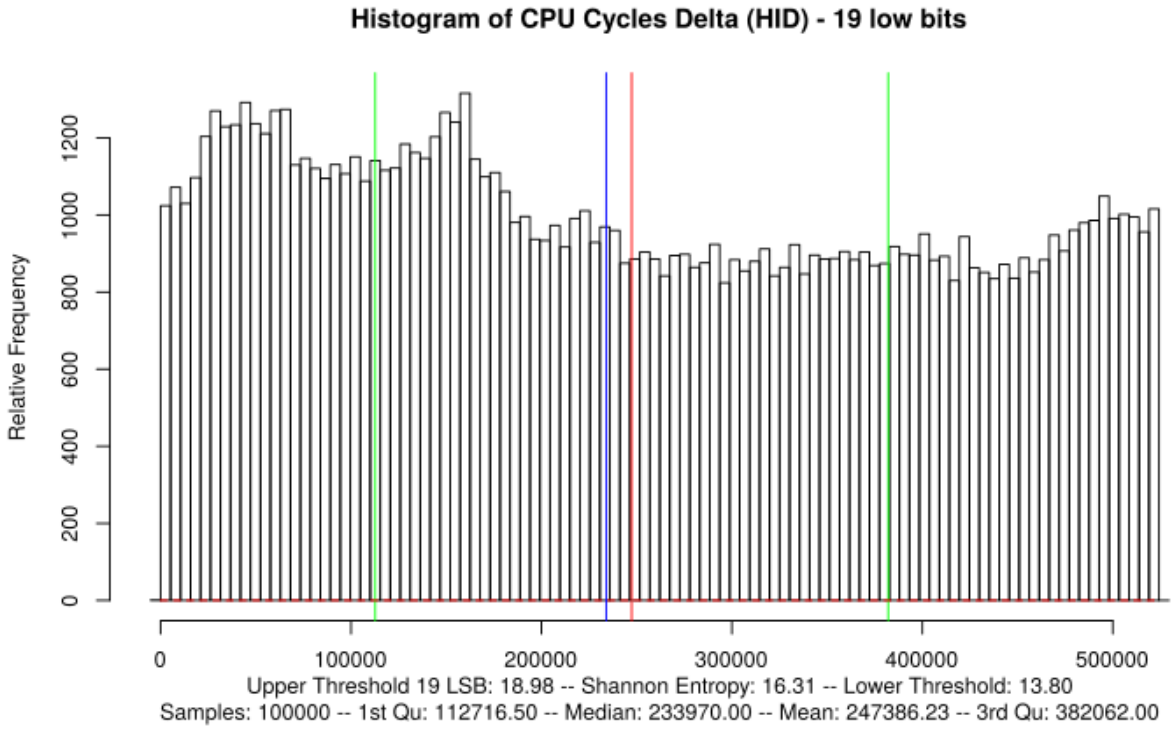

Figure 51: Histogram of High-Resolution Time Deltas for HID – 19 low bits – VirtualBox

Comparing the Shannon Entropy of the data set measured with the LRNG's heuristically calculated entropy estimate depicted in figure 52 allows the conclusion that the LRNG entropy heuristic again underestimates the available entropy to a large extent.

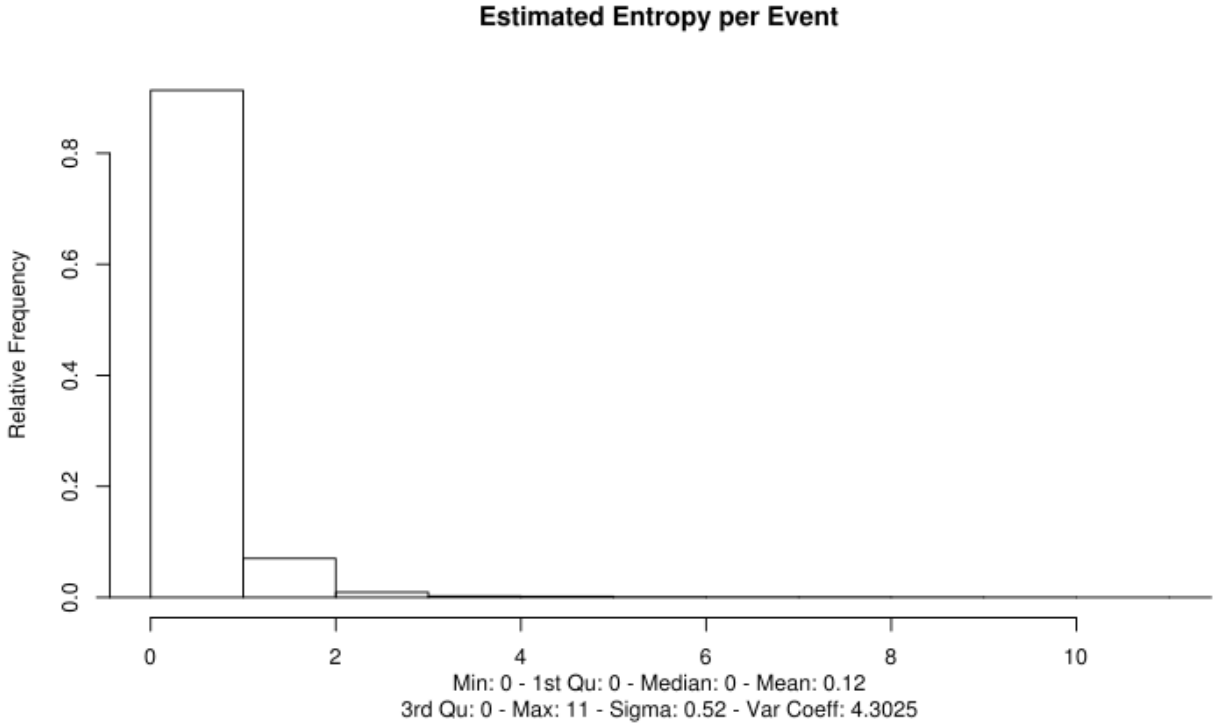

Figure 52: Heuristic Entropy Estimation per HID Event – VirtualBox

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 44 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 8.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.476400	N/A[44]
Bins	3.846235	6.500445
Collection	2.847398	5.882005
Compression	2.789176	5.673606
Collision	2.665133	5.501569

Table 8: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

In addition, the measurements allow the conclusion that the LRNG HID noise source is operational and the VMM effects on this noise source are limited.

5.4.3.2 Conclusion: Configuration Requirement

Assuming that the human interface devices are used, the LRNG HID noise source is operational irrespective of the available VMM configuration.

5.4.4 Microsoft Hyper-V

Like the aforementioned tests, the Hyper-V virtual machine operates without the graphical UI and without X11. The Linux kernel, however picked up the keyboard as well as mouse activity.

5.4.4.1 HID Event Testing

5.4.4.1.1 HID Numerical Identifier

While conducting the HID testing of the guest on Hyper-V, 1918 different HID numerical identifiers are recorded. This result allows do conclude that Hyper-V emits HID events when using the guest console.

As described in section 5.4.2.1.1, these values are not further analyzed as they are not considered to be representative.

5.4.4.1.2 Jiffies Delta of HID Events

The Jiffies delta data plot of the recorded data set is given with figure 53. As explained for KVM and VirtualBox, the graph from figure 53 only depicts the low 19 bits of the Jiffies delta.

Apart from a few spikes which are attributed to some idle time during testing, the measured Shannon Entropy value of 1.7 bits indicates that – just like for KVM and VirtualBox – the Jiffies add some entropy to the HID noise source under Hyper-V.

44 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

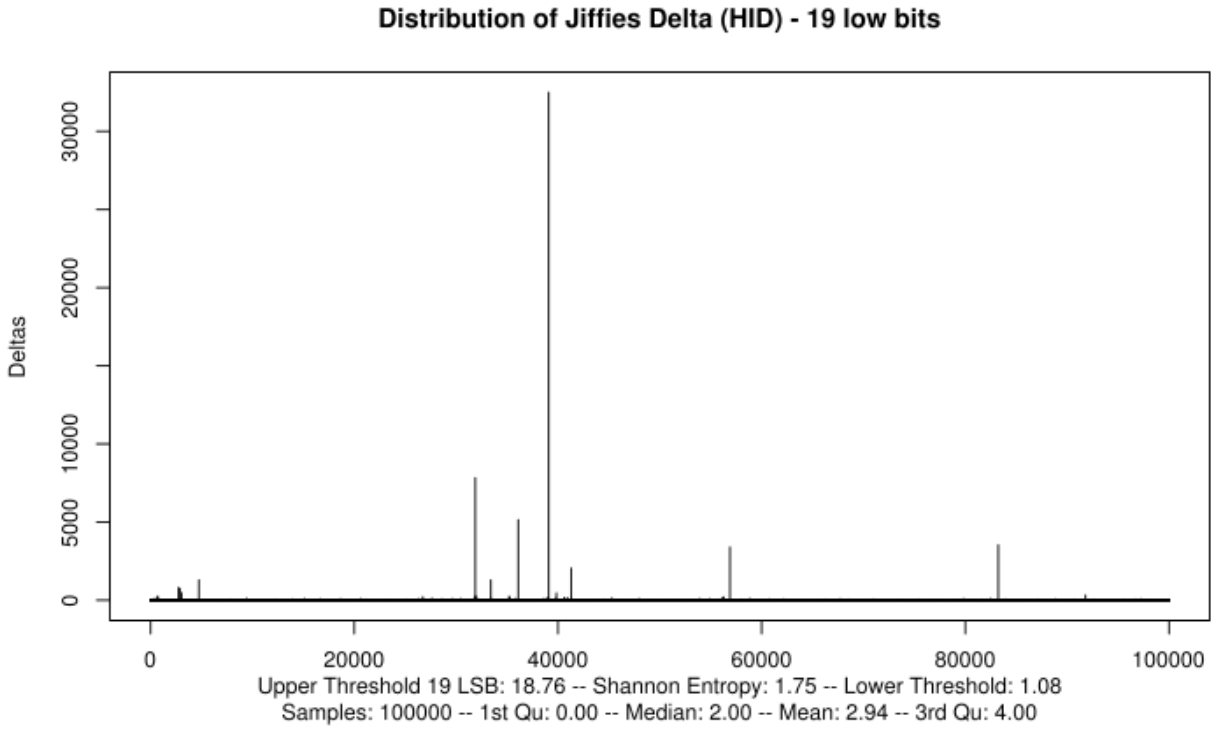

Figure 53: Distribution of Jiffies Delta for HID – 19 low bits – Hyper-V

The histogram given with figure 54 indicates that the large majority of the Jiffies delta is on the low end of the possible values.

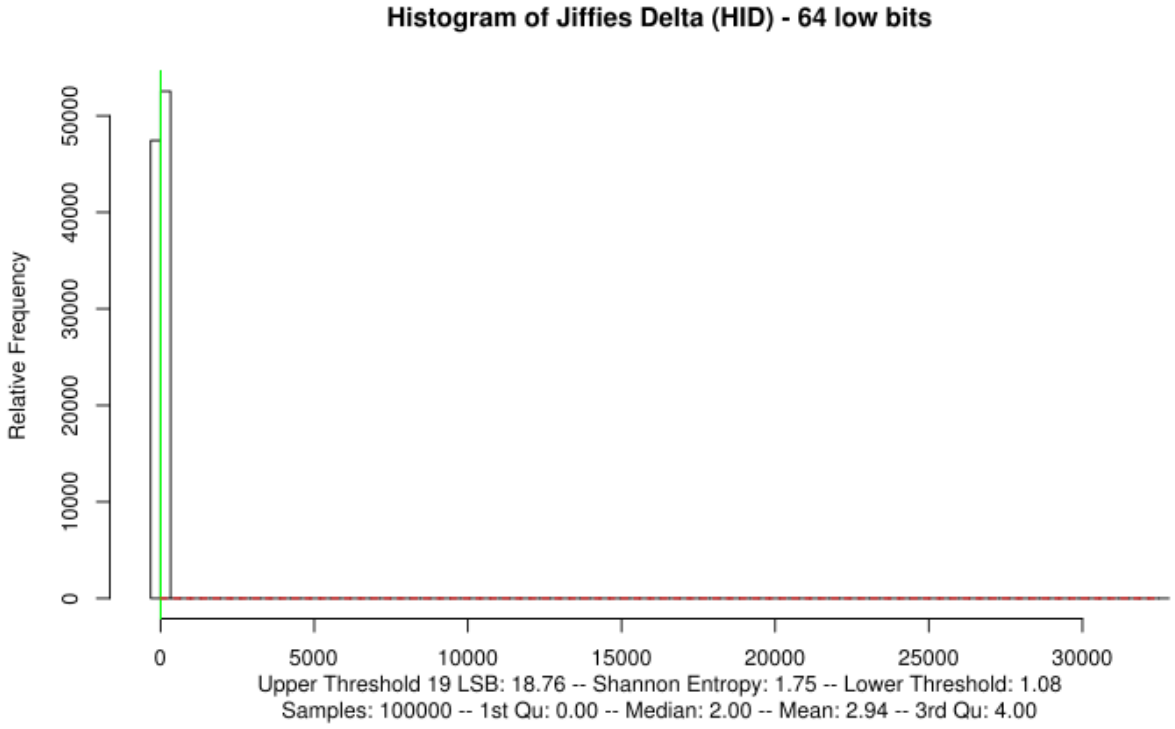

Figure 54: Histogram of Jiffies Delta for HID – Hyper-V

5.4.4.1.3 High-Resolution Time Delta of HID Events

The data plot for the high-resolution time deltas of the HID events collected with a Hyper-V guest are shown in figure 55. Again, the entire possible range of time deltas of 0 to 2^{32} is visible with section 55.

The Shannon Entropy value of 15.4 bits again is significantly larger than the 11 bits threshold implemented by the LRNG entropy heuristic. Therefore, the available entropy in the measured data set is more than sufficient to cover the requirements of the LRNG HID noise source.

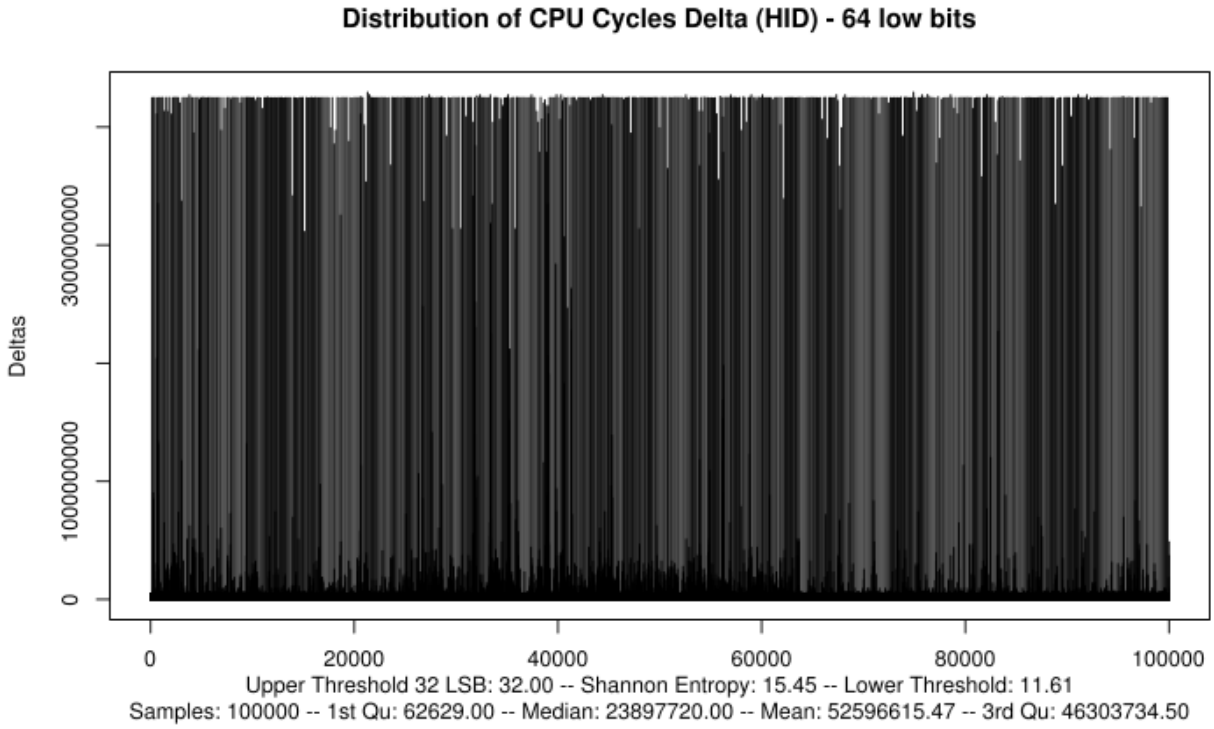

Figure 55: Distribution of High-Resolution Time Delta for HID – Hyper-V

The histogram in figure 56 indicates that the majority of the high-resolution time deltas exhibit a low delta.

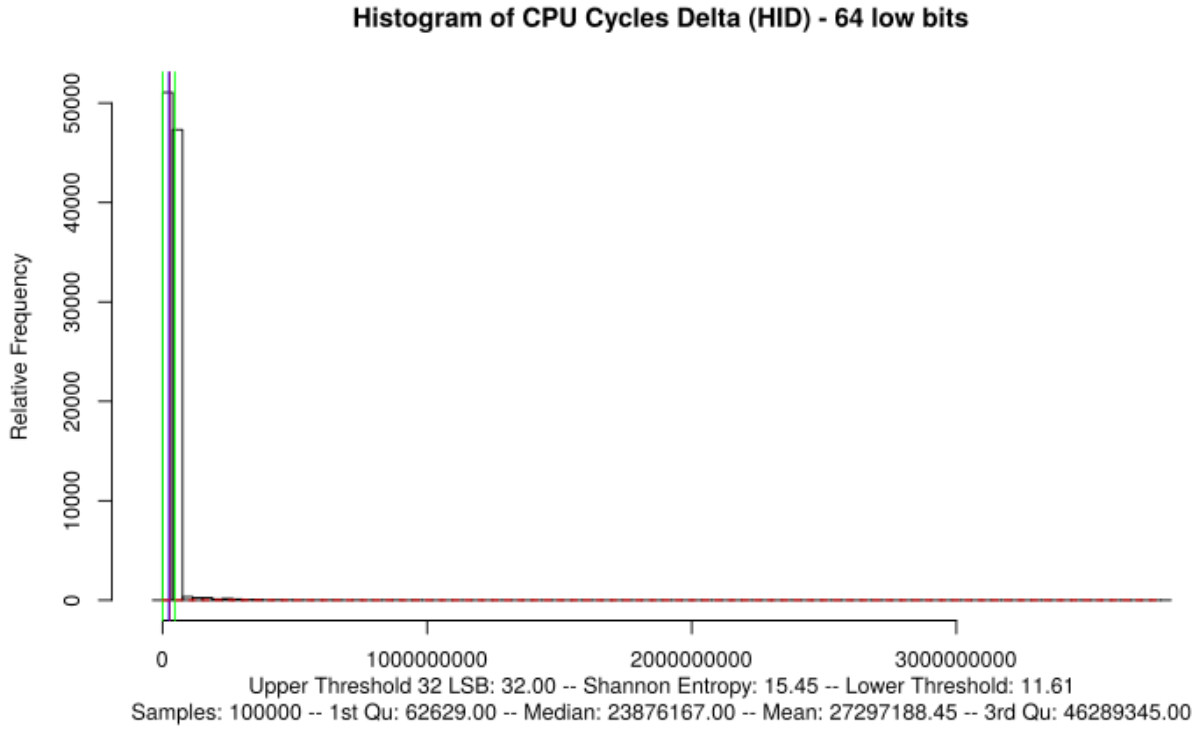

Figure 56: Histogram of High-Resolution Time Delta for HID -- 64 low bits – Hyper-V

Zooming the histogram into 19 bits, figure 57 emerges.

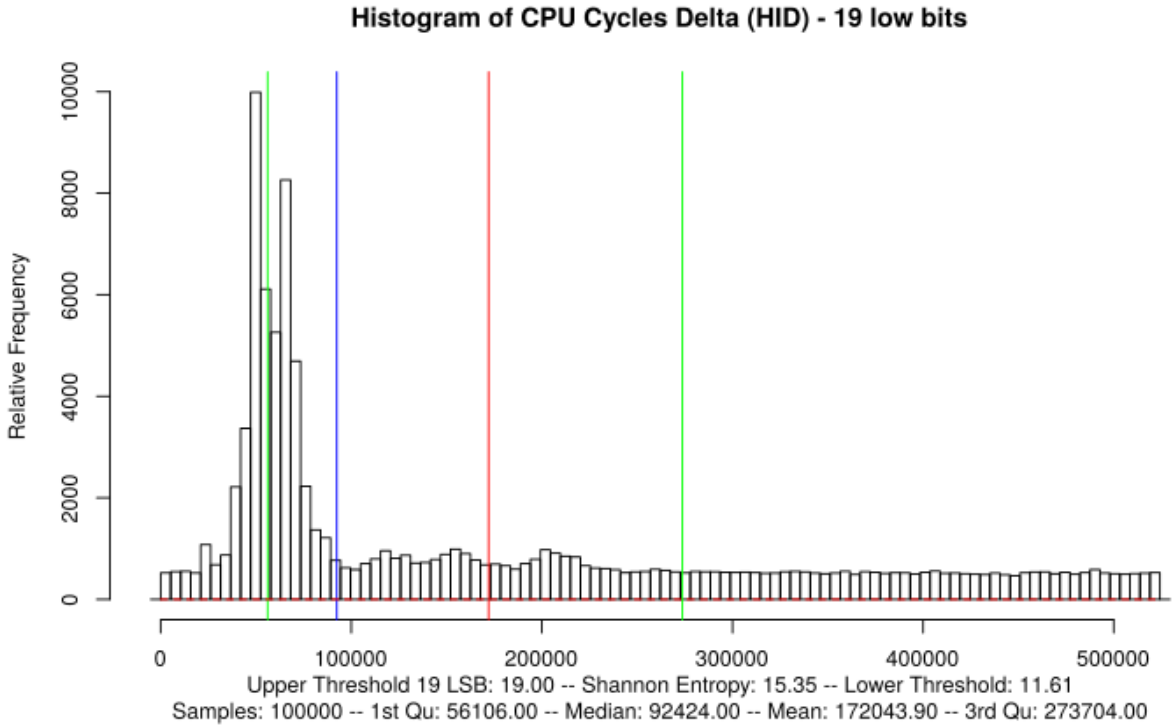

Figure 57: Histogram of High-Resolution Time Delta for HID -- 19 low bits – Hyper-V

Figure 58 allows the comparison of the measured entropy value for the Shannon Entropy with the heuristic entropy estimate applied by the LRNG. With the mean value of about 1 bit of entropy, the heuristic entropy measurement underestimates the available entropy to a large degree.

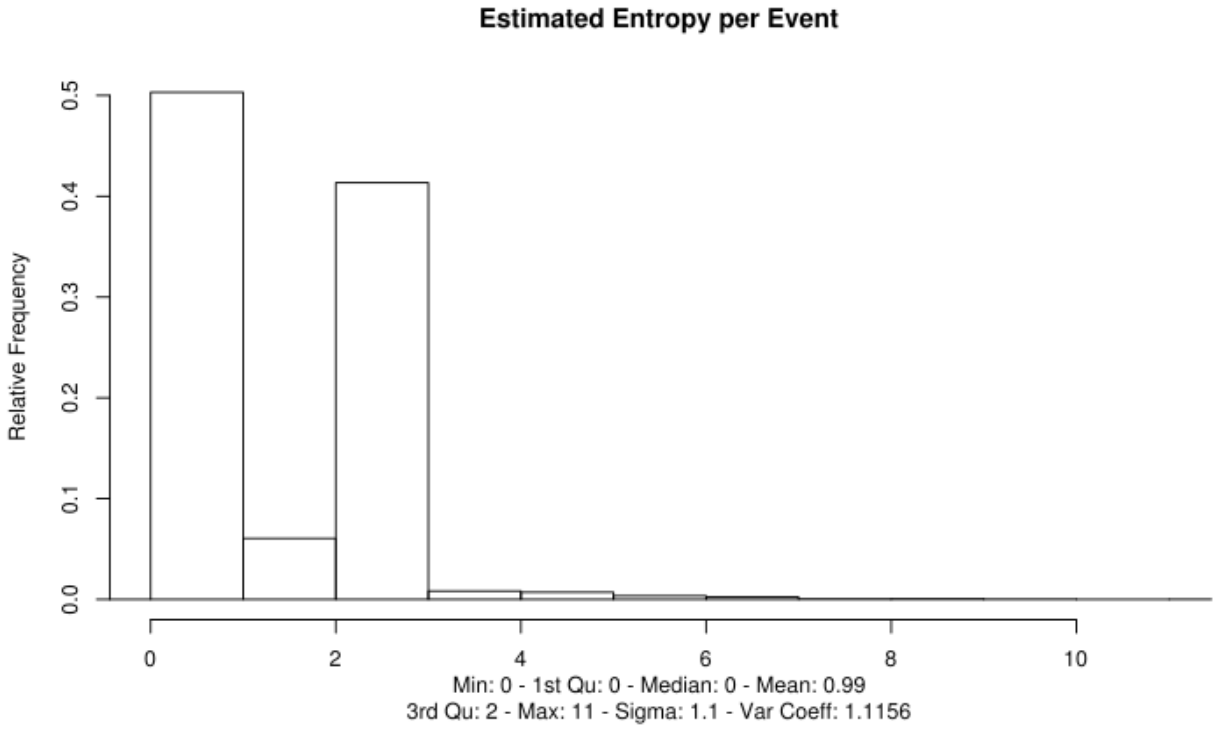

Figure 58: Heuristic Entropy Estimate per HID Event – Hyper-V

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 41 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 9.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.622438	N/A[45]
Bins	3.853516	6.505680
Collection	3.337999	6.144166
Compression	3.434660	6.403868
Collision	3.290378	6.849554

Table 9: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

45 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

The high-resolution time stamp test results show that the HID noise source of the LRNG is operational on Hyper-V where the VMM only has limited impact.

5.4.4.2 Conclusion: Configuration Requirement

When using the human interface devices, the LRNG HID noise source is operational irrespective of the available VMM configuration.

5.4.5 VMWare ESXi

Like the previous tests, the virtual machine hosted by VMWare ESXi executes without a graphical UI and without X11. The Linux kernel includes a driver, which picks up the keyboard as well as mouse activity.

5.4.5.1 HID Event Testing

5.4.5.1.1 HID Numerical Identifier

The execution of the HID testing with the guest on ESXi showed 1122 different HID numerical identifiers. With this result the conclusion can be drawn that ESXi emits HID events when using the guest console.

As described in section 5.4.2.1.1, these values are not further analyzed as they are not considered to be representative.

5.4.5.1.2 Jiffies Delta of HID Events

The data plot of the Jiffies delta is given with figure 59, which only depicts the 19 low bits of the data set. The only spike visible in the data set is due to a quiet time without any HID activity.

The Shannon Entropy of 2.5 bits implies that Jiffies contribute slightly to the HID noise source.

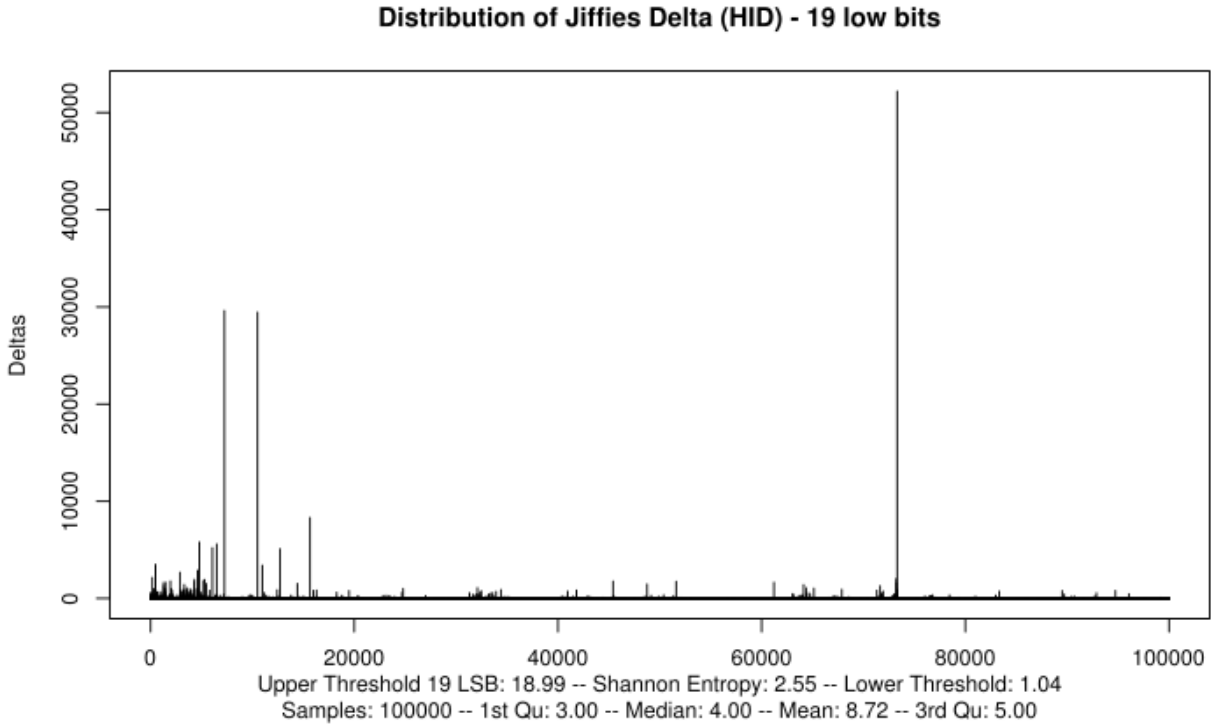

Figure 59: Distribution of Jiffies Delta for HID -- 19 low bits – ESXi

As seen for the other VMMs, the majority of the Jiffies deltas are very close to zero shown in the histogram with figure 60.

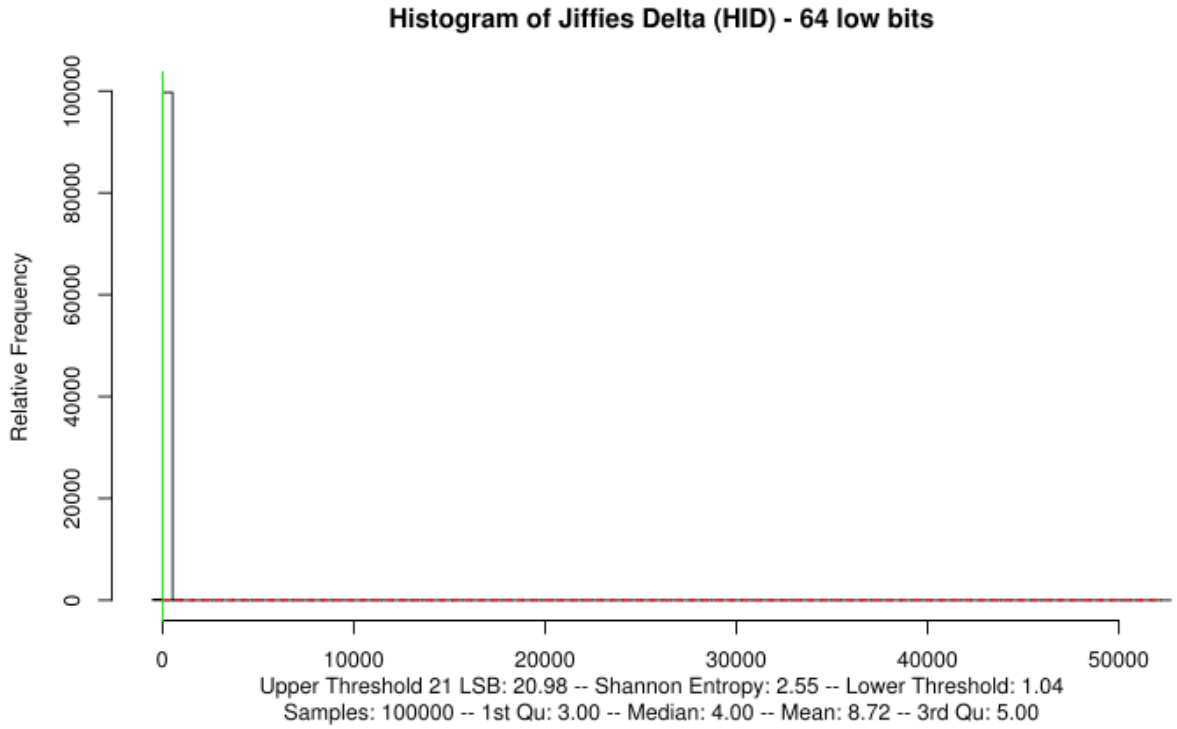

Figure 60: Histogram of Jiffies Delta for HID – ESXi

5.4.5.1.3 High-Resolution Time Delta of HID Events

With figure 61 the data plot of the high-resolution time delta for HID events on ESXi are given. The graph shows that the deltas span the entire theoretically possible range from zero to 2^{32}.

The Shannon Entropy value of 16.6 bits given in the legend of figure 61 again indicates that the high-resolution time stamp provides sufficient entropy for the LRNG HID noise source considering the heuristic entropy estimation which is capped at 11 bits per event.

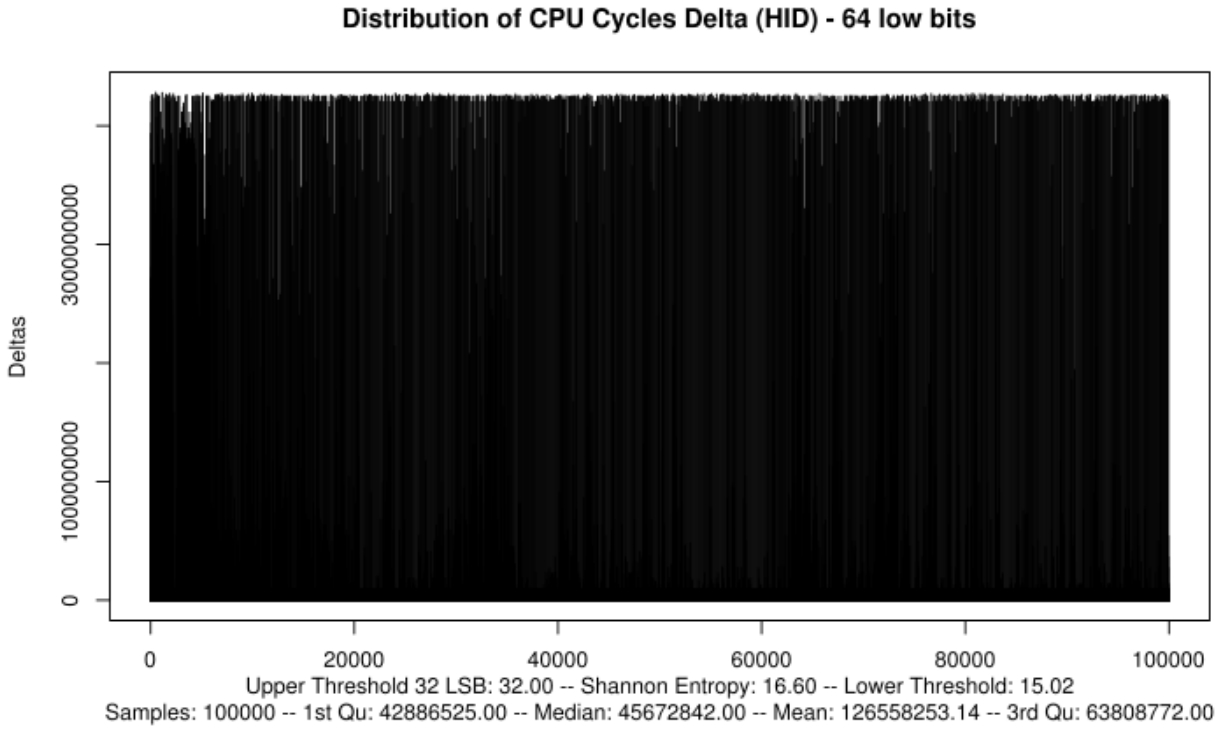

Figure 61: Distribution for High-Resolution Time Delta for HID – ESXi

The figures 62 and 63 provide the histogram for the high-resolution test results, one depicted with all bits and in a zoomed-in version, and the other only the low 19 bits are shown.

Both figures show that the majority of high-resolution time deltas are on the low side of the possible range of values. Nonetheless, the variations of time deltas in that low range is sufficient to deliver a Shannon Entropy of more than 16 bits.

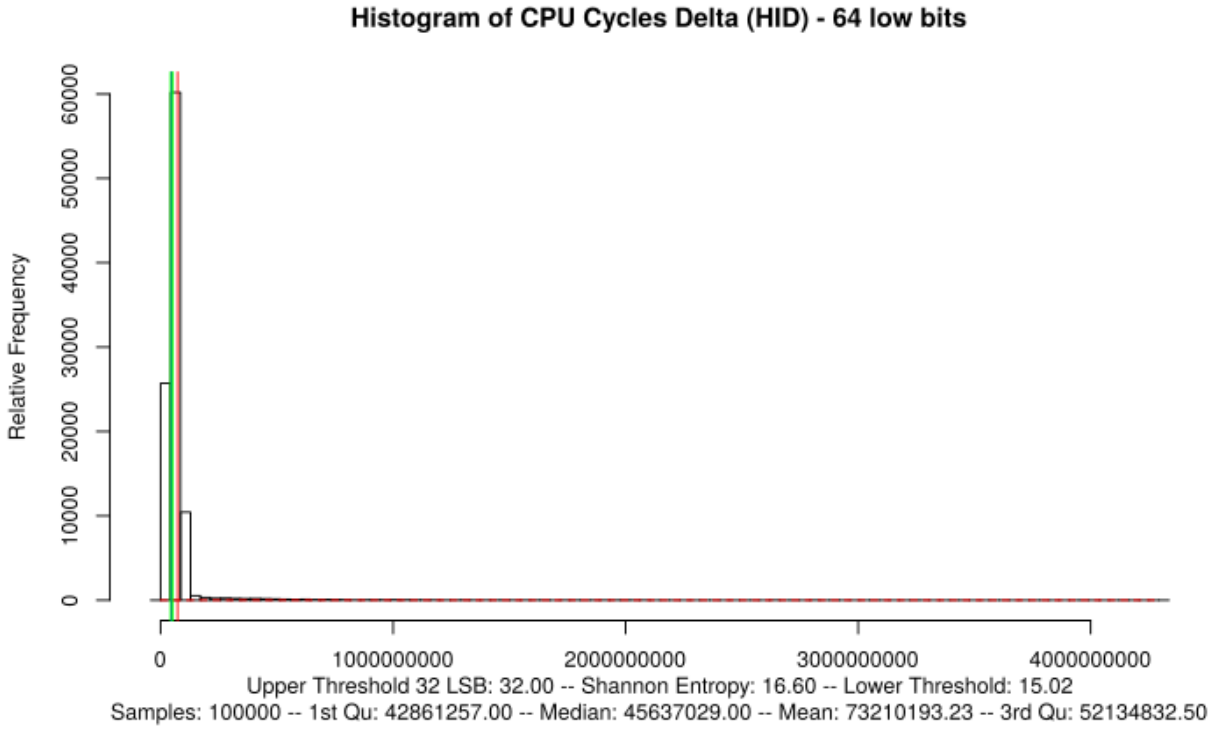

Figure 62: Histogram of High-Resolution Time Delta for HID – 64 low bits – ESXi

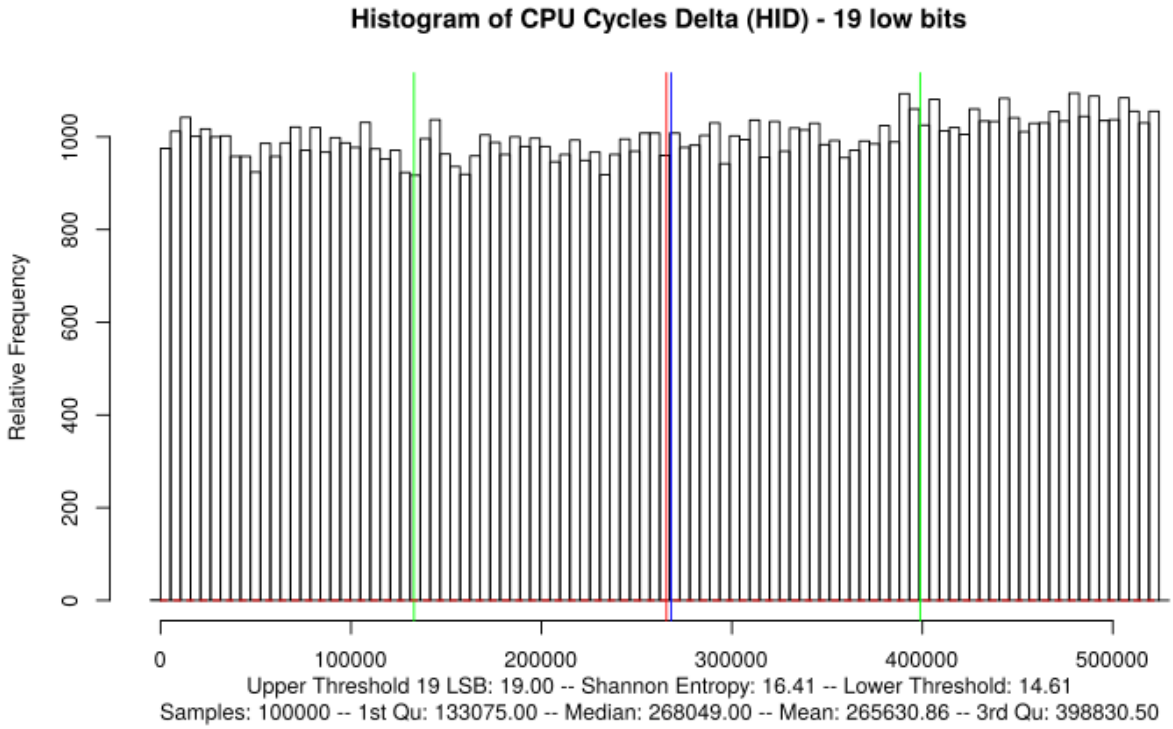

Figure 63: Histogram of High-Resolution Time Delta for HID – 19 low bits – ESXi

Comparing the measured time delta distribution with its Shannon Entropy value with the heuristic entropy estimation applied by the LRNG as given with figure 64, again shows that the LRNG underestimates the available entropy by a large degree.

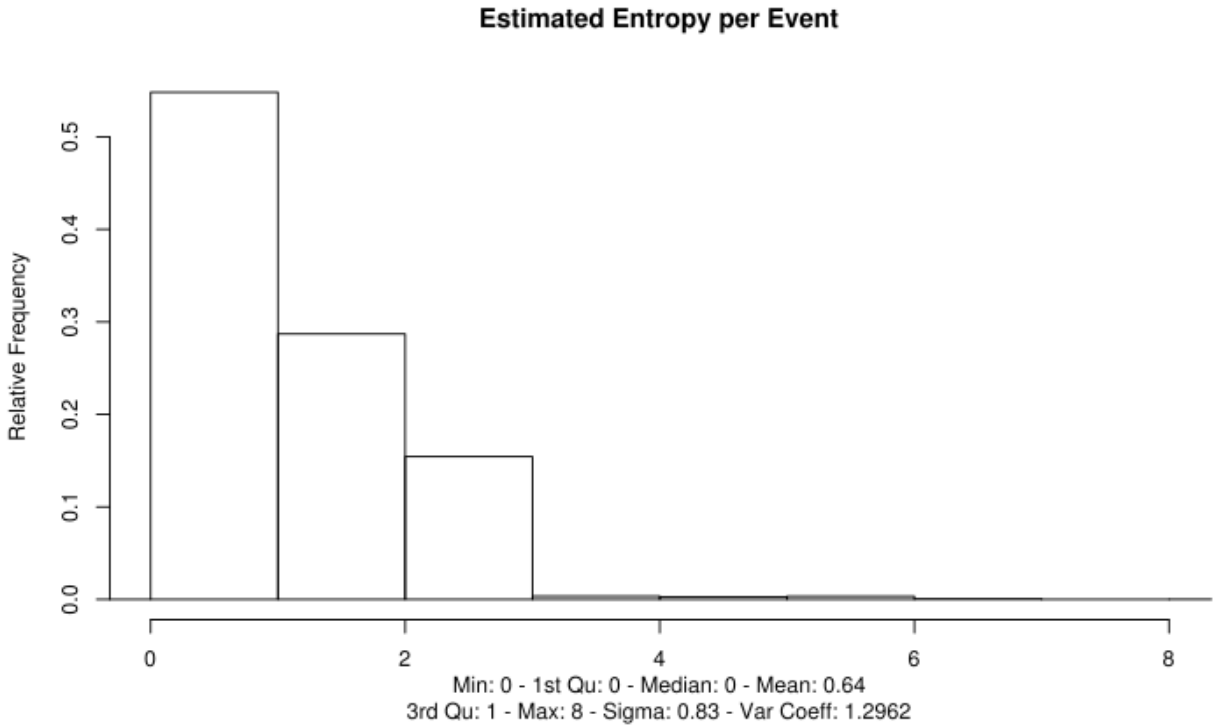

Figure 64: Heuristic Entropy Estimation per HID Event – ESXi

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 44 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 10.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.575396	N/A[46]
Bins	3.854351	6.512249
Collection	3.598987	6.703284
Compression	3.589019	6.262489
Collision	3.609025	5.946973

Table 10: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

46 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

The assessment of the test results shows that the LRNG HID noise source is operational in VMWare ESXi. Furthermore, ESXi affects the noise source operation to a degree that has no impact to its overall appropriateness.

5.4.5.2 Conclusion: Configuration Requirement

Under the assumption that the human interface devices are used, the LRNG HID noise source operates unaffected by the VMM operation.

5.5 LRNG Interrupt Noise Source Testing

The interrupt noise source of the LRNG operates differently compared to the two previously discussed noise sources covering block devices and HID. The LRNG mixes data obtained during the occurrence of interrupts into a per-CPU `fast_pool` data structure. After receiving 64 interrupts, the `fast_pool` is mixed into the `input_pool`[47]. The mixed-in data is awarded one bit of entropy.

The event values recorded for one interrupt are mixed into the `fast_pool` using an LFSR. Therefore, the analysis in this section does not read out the `fast_pool`, because it would assess the quality of the LFSR. Instead, the following test is applied: A SystemTap script is devised which is triggered every time an interrupt is received by the LRNG. With that trigger point, the SystemTap script records the high-resolution time stamp of the event occurrence considering that such time stamp is obtained by the LRNG as well. The additional information obtained by the LRNG interrupt noise source handler, such as the instruction pointer, are disregarded in testing with the following reason. The LRNG assumes only one bit of entropy for the data obtained from 64 interrupts. The following test results will show that the high-resolution time stamp delivers much more entropy than the required 1/64th bit per interrupt. With such a result, the analysis of additional sources of entropy becomes irrelevant.

Note that, this section does not provide an analysis on whether the dependencies between the interrupt noise source and the block device and HID noise sources are reduced. Any results obtained in this section for the different VMMs therefore only compare the measured variations with the entropy content assumed by the LRNG.

To obtain test results for a worst case, interrupts must be generated at a high volume which are controlled by an external entity. A simple way of generating interrupts is by generating network traffic. Each network packet which is received by the Linux kernel triggers an interrupt. To generate a large number of interrupts, a ping flood is sent to the virtual machine under test. The worst case scenario is applied with the following test approach:

- For KVM and VirtualBox with a Linux host system, the ping flood is initiated in the host system. With this approach, network devices that may introduce latencies and variations like switches or routers are discarded. As no tool is available in the standard Windows environment which is capable of generating a ping flood, this approach is not viable for Hyper-V.
- For Hyper-V as well as for ESXi, an Ethernet-Crossover cable links the test system with a second Linux system. This Linux system now sends a ping flood to the virtual machine under test. Again, no physical network devices are present that may introduce latencies.

The graphs in the following subsections include the value for the Shannon Entropy as well as the Minimum Entropy for the given data set.

47 If the last mix-in of data from the current fast_pool is less than one second ago, the LRNG will collect more than 64 interrupts until the one second threshold is reached. For the sake of discussion, this detail will be disregarded. When ignoring it, a worst-case analysis is performed because only the absolute minimum of 64 interrupts are analyzed.

5.5.1 KVM/QEMU

The QEMU configuration of the virtual machine covered the para-virtualized VirtIO network device. The VirtIO network device is assumed to be the default due to its performance compared to other options. Furthermore, other options are an emulation of hardware which will add more variations than measured with VirtIO at least due to the CPU execution time jitter.

5.5.1.1 Interrupt Event Testing

With the received ping flood and the generated interrupts, the graph with the histogram of the high-resolution time delta given in figure 65 can be drawn. The graph's legend with the Shannon and Minimum Entropy indicates that the time stamp generated for each interrupt contains an entropy content of more than 11 bits.

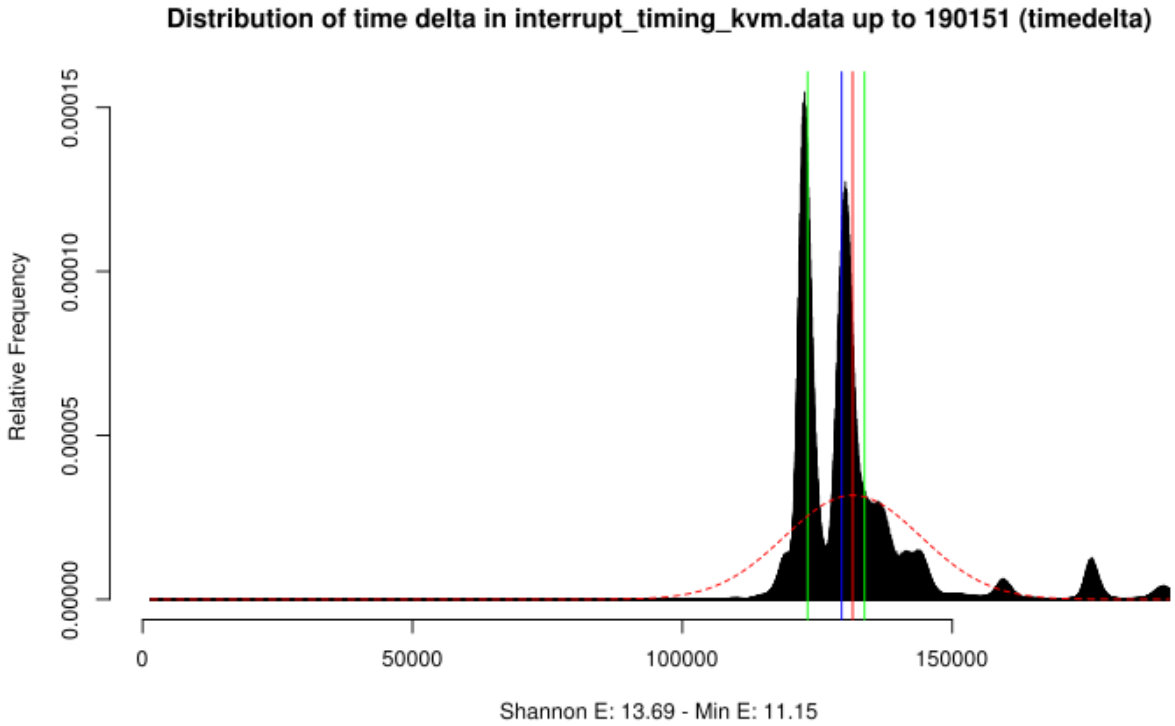

Figure 65: Histogram of High-Resolution Time Delta for Interrupts – KVM

The graph allows the following conclusions to be drawn:

- The variations visible in the high-resolution time deltas of interrupt events is vastly larger than the LRNG's assumed entropy content.
- The VMM injects interrupts into the guest in such a way that the guest Linux kernel interrupt handler is invoked. Thus, the VMM's interference with the interrupts is such that it is negligible.

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 49 ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID. The author notes, however, that the IID calculation was performed using the first 10,000 time stamps only. Using all 5,000,000 time stamps would have caused a very long calculation time. To support the obtained result, another two sets of 10,000 time stamps starting with the 1,000,000th and 2,000,000th time stamp show the same results. As the used subset already indicates a non-IID, the author assume that the remaining time stamps will not change the result.

The time delta byte data stream shows the Minimum Entropy values shown in table 11.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.938358	N/A[48]
Bins	3.987679	7.809735
Collection	3.615081	7.136812
Compression	3.511658	6.936858
Collision	3.373492	6.338972

Table 11: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

5.5.1.2 Conclusion: Configuration Requirement

No special configurations neither for the VMM nor for the guest are required to allow the LRNG operating in the guest to receive interrupt events.

5.5.2 Oracle VirtualBox

The network device is emulated by the VMM as an Intel Gigabit Ethernet Controller. The guest uses the device driver of e1000.

```
lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: InnoTek Systemberatung GmbH VirtualBox
Graphics Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet
Controller (rev 02)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio
Controller (rev 01)
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA
Controller [AHCI mode] (rev 02)
```

Code 1: Device Listing

5.5.2.1 Interrupt Event Testing

As visible with figure 66, the guest LRNG receives the interrupts triggered by the ping flood. The high-resolution time delta is significantly larger than the entropy content assumed by the LRNG. Yet again, figure 66 demonstrates that the interrupts injected into the guest by the VMM are received by the guest LRNG. This implies that the VirtualBox VMM operation does not affect the appropriateness of the LRNG interrupt noise source.

48 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

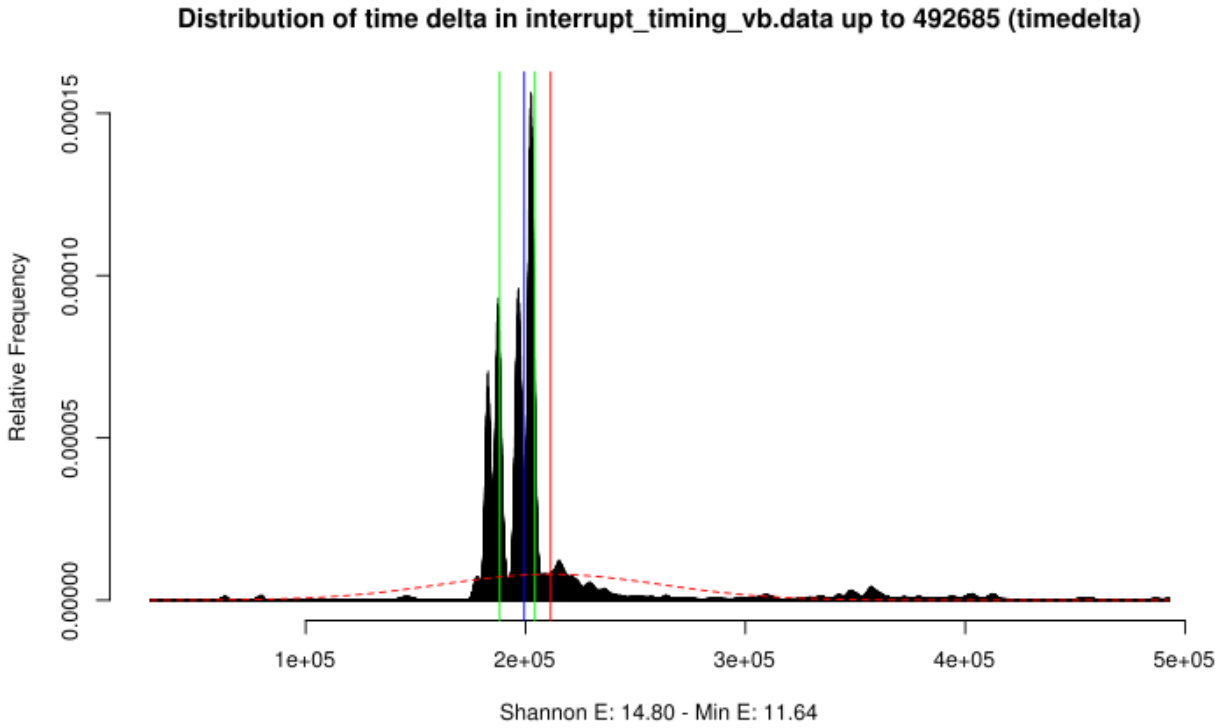

Figure 66: Histogram of High-Resolution Time Deltas of Interrupt Events – VirtualBox

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 40 (first 10,000 time stamps), 49 (10,000 time stamps starting with the 1,000,000th time stamp), and 39 (10,000 time stamps starting with the 2,000,000th time stamp) ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 12.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.933145	N/A[49]
Bins	3.987988	7.810372
Collection	3.565372	7.174327
Compression	3.474479	7.095742
Collision	3.268734	6.798623

Table 12: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

49 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

5.5.2.2 Conclusion: Configuration Requirement

The devices emulated by the VirtualBox VMM trigger the interrupt handler of the guest similarly to regular hardware. Therefore, no special configuration of the VMM or the guest is necessary.

5.5.3 Microsoft Hyper-V

Microsoft Hyper-V Linux Integration Services implement with the VMBus, a mechanism that bypasses the entire Linux interrupt handling code. That is visible with the following configuration: When configuring a virtual machine with the default configuration values, the interrupt testing with the ping flood will **not** return any results. Hence, when using the guest as a network server, interrupts are not recorded by the guest as long as only the para-virtualized devices using the VMBus receive events.

Interrupt events are received, however, for devices that are not attached to the VMBus. For example, when configuring the virtual machine to use the legacy network device, the guest is emulated a DEC network device:

```
lspci
00:00.0 Host bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX Host bridge
(AGP disabled) (rev 03)
00:07.0 ISA bridge: Intel Corporation 82371AB/EB/MB PIIX4 ISA (rev 01)
00:07.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:07.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 02)
00:08.0 VGA compatible controller: Microsoft Corporation Hyper-V virtual VGA
00:0a.0 Ethernet controller: Digital Equipment Corporation DECchip 21140
[FasterNet] (rev 20)
```

For accessing the DEC Ethernet device, the guest Linux kernel uses the “tulip” device driver.

5.5.3.1 Standard Linux IRQ Handler: Interrupt Event Testing

With the legacy network device configured, the guest LRNG receives interrupt events with the histogram of the high-resolution time deltas shown with figure 67. The variations of the high-resolution time delta again are so large that the LRNG's assumption of its entropy content is significantly underestimated. Furthermore, figure 67 demonstrates that the standard interrupt handling logic of the Linux kernel is exercised for the legacy network device.

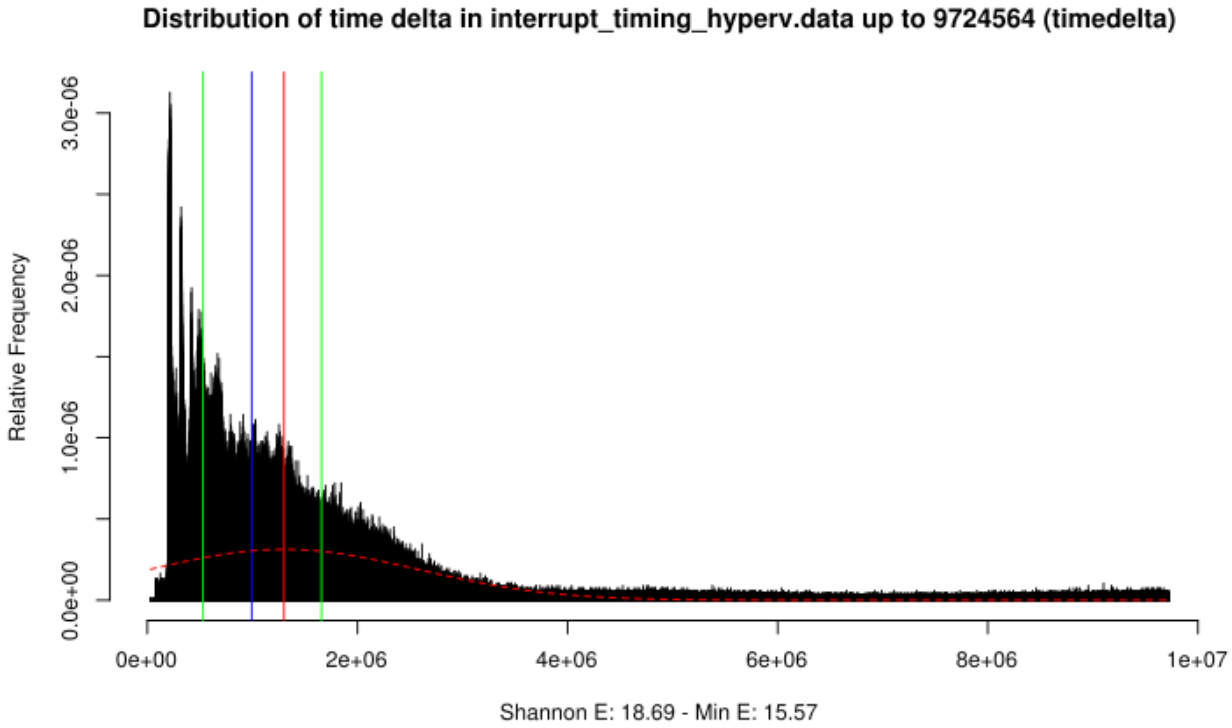

Figure 67: Histogram of High-Resolution Time Deltas for Interrupt Events – Hyper-V and Standard Linux IRQ Handler

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 50 (first 10,000 time stamps), 48 (10,000 time stamps starting with the 1,000,000th time stamp), and 48 (10,000 time stamps starting with the 2,000,000th time stamp)ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 13.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.950860	N/A[50]
Bins	3.981952	7.713445
Collection	3.812029	7.326153
Compression	3.730873	7.195062
Collision	3.713442	8.000000

Table 13: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

50 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

5.5.3.2 VMBus IRQ Handler: Interrupt Event Testing

The testing of the high-resolution time deltas of interrupt events is repeated for interrupts handled by the VMBus. The SystemTap script was slightly modified to be triggered when the VMBus interrupt handler is invoked.

The resulting high-resolution time deltas are depicted with figure 68. The entropy present in the VMBus interrupts is only slightly smaller than the interrupts handled by the Linux interrupt handler shown with figure 67. Yet, they are significantly larger than required by the LRNG entropy heuristic.

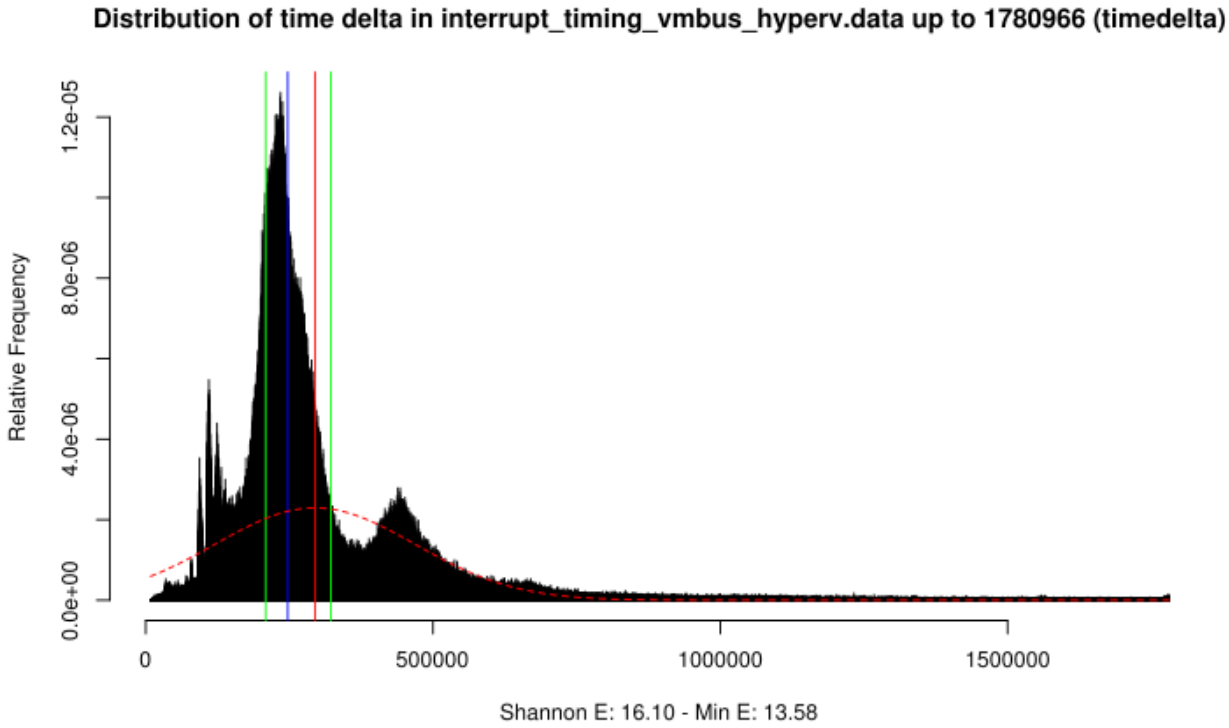

Figure 68: Histogram of High-Resolution Time Deltas for Interrupt Events – Hyper-V and VMBus IRQ Handler

Please note that the VMBus interrupts are **not** used as noise sources by the LRNG. Hence, the analysis in this section is educational only[51].

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 48 (first 10,000 time stamps), 50 (10,000 time stamps starting with the 1,000,000th time stamp), and 48 (10,000 time stamps starting with the 2,000,000th time stamp) ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 14.

51 At the time of writing, patches are being discussed among kernel developers to use the VMBus interrupt handler as noise source which would imply that this measurement will become relevant if the patches are applied.

Min Entropy	4 Bit	8 Bit
Markov	3.936930	N/A[52]
Bins	3.977387	7.670464
Collection	3.812029	7.124521
Compression	3.821335	7.048533
Collision	3.670764	6.390010

Table 14: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

5.5.3.3 Conclusion: Configuration Requirement

The administrator of the Hyper-V VMM and its guests should understand that all para-virtualized devices using the VMBus communication channel between the guest and the VMM will not trigger the LRNG interrupt noise source.

When the LRNG interrupt noise source shall be invoked for a given device, a legacy device configuration must be selected, if available. For example, the network device configuration offers the choice of a legacy device.

The use of a legacy device, however, implies a drop in performance for the given device as the para-virtualized devices are usually significantly faster.

5.5.4 VMWare ESXi

The guest hosted by VMWare ESXi is configured with the default values offered by the configuration frontend. The devices visible in the guest are shown with the following listing.

```
lspci
00:00.0 Host bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX Host bridge
(rev 01)
00:01.0 PCI bridge: Intel Corporation 440BX/ZX/DX - 82443BX/ZX/DX AGP bridge (rev
01)
00:07.0 ISA bridge: Intel Corporation 82371AB/EB/MB PIIX4 ISA (rev 08)
00:07.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:07.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:07.7 System peripheral: VMware Virtual Machine Communication Interface (rev 10)
00:0f.0 VGA compatible controller: VMware SVGA II Adapter
00:10.0 SCSI storage controller: LSI Logic / Symbios Logic 53c1030 PCI-X Fusion-
MPT Dual Ultra320 SCSI (rev 01)
00:11.0 PCI bridge: VMware PCI bridge (rev 02)
00:15.0 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.1 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.2 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.3 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.4 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.5 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.6 PCI bridge: VMware PCI Express Root Port (rev 01)
00:15.7 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.0 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.1 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.2 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.3 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.4 PCI bridge: VMware PCI Express Root Port (rev 01)
```

52 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

```
00:16.5 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.6 PCI bridge: VMware PCI Express Root Port (rev 01)
00:16.7 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.0 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.1 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.2 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.3 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.4 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.5 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.6 PCI bridge: VMware PCI Express Root Port (rev 01)
00:17.7 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.0 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.1 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.2 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.3 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.4 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.5 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.6 PCI bridge: VMware PCI Express Root Port (rev 01)
00:18.7 PCI bridge: VMware PCI Express Root Port (rev 01)
02:01.0 USB controller: VMware USB1.1 UHCI Controller
02:02.0 USB controller: VMware USB2 EHCI Controller
03:00.0 Ethernet controller: VMware VMXNET3 Ethernet Controller (rev 01)
```

The last entry in the device listing references the para-virtualized Ethernet device. The Linux device driver is provided with vmxnet3.

5.5.4.1 Interrupt Event Testing

Figure 69 provides the histogram of the high-resolution time deltas obtained from the interrupt events triggered by the ping flood. Again, the variations in these time deltas are so large that the LRNG's assumed entropy content massively underestimates the available entropy.

Also, the test result depicted in figure 69 shows that the interrupt handler of the LRNG is triggered when receiving interrupts from para-virtualized device drivers.

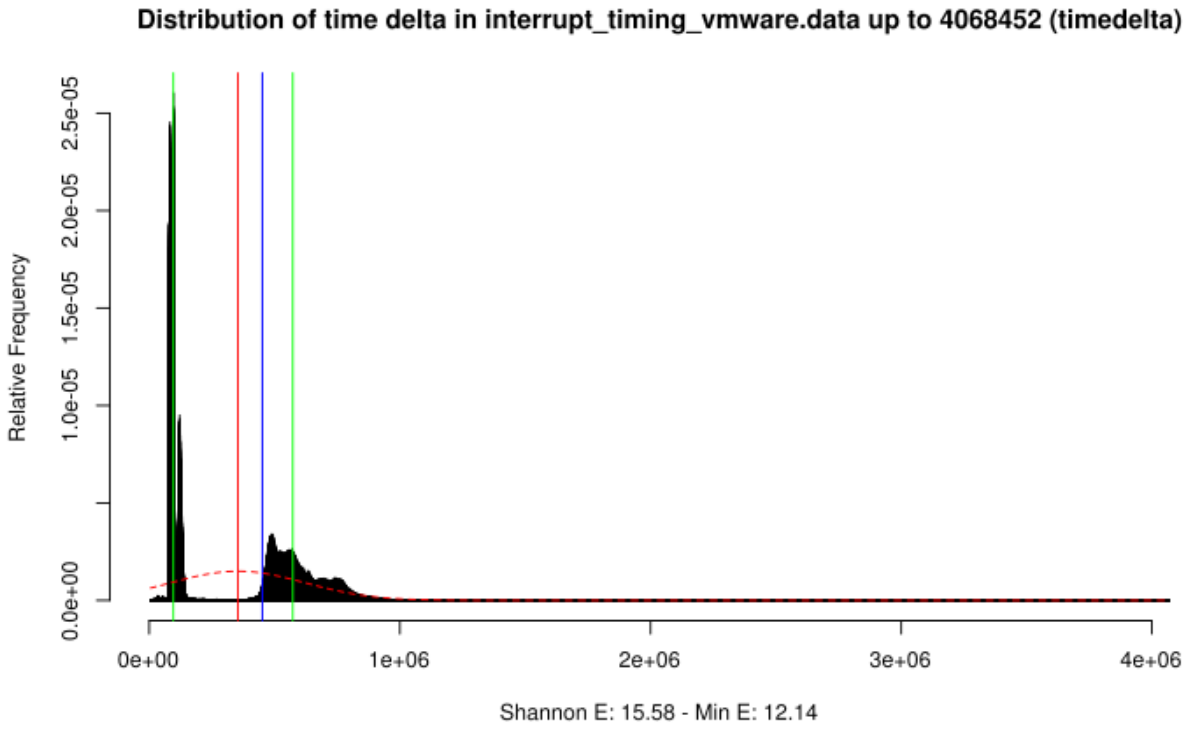

Figure 69: Histogram of High-Resolution Time Deltas of Interrupt Events – ESXi

Using the same approach as discussed in section 5.3.3.2.3 for calculating the SP800-90B Minimum Entropy values, the following result can be obtained.

The IID calculation shows that 49 (first 10,000 time stamps), 49 (10,000 time stamps starting with the 1,000,000th time stamp), and 50 (10,000 time stamps starting with the 2,000,000th time stamp) ranks are below 5% or above 95% which implies that according to section 9.1.2 [SP800-90B] the noise source is considered to be non-IID.

The time delta byte data stream shows the Minimum Entropy values shown in table 15.

Min Entropy	**4 Bit**	**8 Bit**
Markov	3.957971	N/A[53]
Bins	3.985992	7.773209
Collection	3.849618	7.208783
Compression	3.766380	7.301729
Collision	3.670764	7.119938

Table 15: Minimum Entropy according to SP800-90B

The results in the table show that:

- Considering the 8 bit column, much more entropy is available than the average of the entropy heuristic of the LRNG.
- Comparing the 4 and 8 bit columns, the entropy value increases almost linearly which implies that the entropy is evenly distributed in the least significant bits.

5.5.4.2 Conclusion: Configuration Requirement

The LRNG interrupt handler is triggered by devices emulated by the ESXi VMM similarly to real hardware. Therefore, no special configuration of the VMM or the guest is considered necessary.

5.6 Seeding of /dev/urandom

The Linux kernel prints a kernel log entry when the nonblocking_pool behind /dev/urandom is initially seeded with 128 bits of entropy. In addition, it prints a log when it is first accessed by a user space process.

When executing Linux as a guest in a KVM virtual environment, the following kernel log was obtained:

```
[    0.575029] random: systemd urandom read with 3 bits of entropy available
[   33.206682] random: nonblocking pool is initialized
```

The number at the beginning is the time in seconds since the kernel came to life during boot. The boot process of the operating system completes within 2 seconds. This boot process includes cryptographic daemons like the OpenSSH daemon that requires seed from /dev/urandom for its DRNG.

The test was repeated multiple times and showed varying time stamps for the second entry, i.e. the entry indicating that the nonblocking_pool is seeded with 128 bits of entropy. The time stamps were all between 25 seconds and 90 seconds.

These log entries need to be compared to a Linux system executed directly on the hardware without a VMM:

```
[    0.702178] random: systemd urandom read with 8 bits of entropy available
```

53 The calculation of the Markov Minimum Entropy cannot be performed for chunks larger than 6 bits.

```
[    6.324100] random: nonblocking pool is initialized
```

The differences in the time stamps are significant and allow the conclusion that /dev/urandom is seeded with significantly less entropy at the same time during the boot process compared to a system executing directly on the hardware.

5.7 Final Conclusions for VMMs

The test execution and the test results in the preceding subsections allow to draw various conclusions.

In general, for all tested VMMs there are VMM-specific configurations that do not impact the appropriateness of the different LRNG noise sources. Such configurations are suggested to receive entropic random data. Nonetheless, there are VMM configurations, where one or more of the LRNG noise sources are non-operational. The following subsections summarize the configurations which impact the LRNG operation to the extent that one or more noise sources are unavailable.

As a first step, conclusions applicable to all VMMs are discussed. This is followed by a discussion of conclusions applicable to a given VMM.

5.7.1 General Conclusions Applicable to All VMMs

During the conduction of the testing, configuration and usage aspects are identified which affect the entropy collection of the LRNG. In some cases, the identified issues are not solely attributed to the use of VMMs and their effect on the LRNG noise sources, but are relevant to the discussion nonetheless.

5.7.1.1 Human Interface Devices

Virtual machines are commonly used as server type systems. With such systems, it is rare or uncommon to use the console of the server system. This implies that human interface devices are hardly used or are completely unused.

Therefore, in common use cases, the HID noise source will not, or only very rarely collect any entropy.

5.7.1.2 KVM/QEMU

The KVM/QEMU environment includes configurations that deactivate LRNG noise sources.

5.7.1.3 VirtIO Block Devices

When providing a block device to the guest using the VirtIO bus, the guest Linux kernel will deactivate that block device as a source for entropy. Hence, these block devices will not trigger events in the LRNG block device noise source.

5.7.2 Oracle VirtualBox

Currently, no VirtualBox specific configurations are identified that would affect the LRNG noise source operations.

5.7.3 Microsoft Hyper-V

A Linux guest operating under a Microsoft Hyper-V may be affected by the VMM operation. With this operation a dangerous and yet common configuration can be conceived which implies that the LRNG will not collect any entropy – i.e. all noise sources are deactivated.

5.7.3.1 VMBus

Para-virtualized devices are provided with Microsoft's Linux Integration Services. These devices use a Hyper-V-specific communication channel called VMBus. The Linux driver for VMBus is implemented such that interrupts generated by Hyper-V for informing the Linux kernel about the presence of new data are processed in code paths which are special to VMBus. This means the standard interrupt handling code of the Linux kernel is not invoked when Hyper-V issues interrupts pertaining the VMBus communication channel.

This implies that all para-virtualized devices using the VMBus will not contribute to the LRNG interrupt noise source. The important device drivers for block devices, HID and networking can be provided using VMBus. Assuming the absence of auxiliary devices in the guest, the LRNG interrupt noise source may not collect any entropy.

Note that, the use of para-virtualized device drivers with VMBus support is considered as default, because the default guest configuration applied by Hyper-V enables those. Also note that, current Linux kernels and distributions already offer drivers using VMBus. No additional software must be installed.

5.7.4 VMWare ESXi

Currently, no VMWare ESXi specific configurations are identified which would affect the LRNG noise source operations.

6 Alternatives and Supplements to LRNG in VMMs

The discussions in chapters 4 and 5 show that VMMs commonly have a significant impact on software noise sources. This can be seen in chapter 5 which explains that the Linux Random Number Generator requires a number of special considerations and use cases if it shall work similarly as on bare-metal systems.

The following sections are intended to illustrate alternatives or supplements to the LRNG, when operating in VMMs which:

- are unaffected by the VMM, or
- are noise sources provided by the VMM.

These sections list particular implementations of noise source and should be considered to help the reader find additional noise sources or develop noise sources which are based on unpredictable phenomena unaffected by VMMs. This list is by no means considered to be complete but shall give the reader ideas how to identify noise sources that operate unaffected by the VMM.

For collecting entropy, it is always wise to use as many noise sources as possible, provided they operate independent of each other. Therefore, the discussed noise sources in the following sections should supplement the existing noise sources of the LRNG. The LRNG provides two general ways how additional noise sources can be added:

1. Using appropriate IOCTLs on /dev/random allows the injection of random data into the entropy pools of the LRNG and to update the entropy estimator.
2. Implementing a device driver that hooks into the hardware random number device framework in the kernel allows the LRNG to draw from this noise source.

6.1 Noise Source Unaffected by VMM: Jitter RNG

The description of the operation of the Jitter RNG is given in section 3.3.3. As explained there, this document has no intention to assess the quality of the RNG and its entropy collection operation to avoid any conflict of interest. The goal of this document is to analyze the noise source regarding how it is affected by the VMM.

The Jitter RNG is a pure software algorithm which derives its entropy from the uncertainty of the execution time of a set of CPU instructions. With this statement, the hardware components required by the Jitter RNG software part are already defined:

- ability to execute a set of CPU instructions directly on the CPU, and
- the ability to use a high-resolution timer to measure the execution time of those instructions.

The discussion of chapter 4 shows that the VMM only interferes when a software algorithm using regular CPU instructions requires the use of hardware or the CPU instructions try to access restricted resources such as memory-mapped registers of emulated devices.

The Jitter RNG uses the behavior of the CPU to derive its entropy from. To stimulate the unpredictable behavior such that it can be recorded, the Jitter RNG executes regular CPU instructions such as branching, addition and XOR. None of these instructions would warrant a VMM intercept – the instructions used by the Jitter RNG do not have the potential to violate the virtual machine isolation or the protection of the VMM from the guest. When the guest executing the Jitter RNG is scheduled, the VMM allows the execution of the CPU instructions directly on the CPU. Therefore, the first constraint of the Jitter RNG is met without VMM interference: direct execution of the Jitter RNG code stimulating and recording the unpredictable phenomenon on the CPU.

The second aspect required by the Jitter RNG is access to a high-resolution time stamp. As discussed in section 4.2.4.2, the high-resolution time stamp is provided with a CPU instruction. The VMM would impact the Jitter RNG operation if the VMM would intercept the instruction to obtain the high-resolution time stamp. The analysis in section 4.3 outlines which VMMs intercept the CPU instruction for gathering the high-resolution time stamp. Although at least Oracle VirtualBox can intercept the CPU instruction, the testing in sections

5.3ff indicates that the intercept still provides a high-resolution time stamp to the guest. i.e. the VMM will at most add or subtract an offset to the time stamp. Therefore, the Jitter RNG' s high-resolution time stamp gathering is not considered to be affected by the VMM operation.

This allows the conclusion that the Jitter RNG in the worst case is not affected by the operation of the VMM assessed in this document. Quite to the contrary: the Jitter RNG measures the timing behavior of a set of instructions. As the VMM may reschedule within the Jitter RNG operation, the duration of the execution of a given set of instructions will be significantly longer. Thus the scheduling of the VMM adds to the unpredictability of the timing measurements. This implies that the entropy collected by the Jitter RNG within a VMM may even be higher than on a bare metal system.

6.1.1 Muen Separation Kernel

Although the Jitter RNG works fine with the VMMs discussed here, hypervisors can indeed interfere with the operation of this RNG. A real-world example is provided here as a brief digression to demonstrate that non-interference by the VMM must not be taken for granted, but needs to be analyzed for new VMMs.

In the course of writing this document, the author was contacted by a developer of the Muen Separation Kernel (SK) with the notification that a current Linux kernel produces kernel logs, indicating that the Jitter RNG present in the Linux kernel crypto API does not work correctly. The logs showed that the runtime test verifying that the received random data indicates a healthy noise source warned, that the source was degraded.

The Muen SK is used to provide a separation hypervisor, allowing the execution of multiple native (bare-metal) and VM components. The SK claims strong isolation properties and supports Linux as guest VM.

To counter side channel attacks, untrusted Linux guests on Muen have no direct access to the fine-grained CPU Time Stamp Counter (TSC). Instead, a paravirtualized TSC driver is provided, which only exports TSC values with maximum granularity in the microsecond range.

The demo system in question even used millisecond granularity at first. The Jitter RNG, however, requires a time stamp that is more fine grained for operation. Therefore, the runtime test performed by the Jitter RNG correctly treated the noise source as degraded.

After a few rounds of discussion and testing with the Muen developer, a setting in the Muen scheduling policy was found that provided a time stamp to the guest with a resolution sufficiently fine grained for the Jitter RNG to operate and yet cover the needs of the Muen developers.

6.2 Noise Source Provided by VMM: KVM virtio-rng

The VMM may provide specific support for random number generation. Such support can be used by the guest to obtain entropy from noise sources which would otherwise be either inaccessible or heavily affected by a VMM operation.

This section discusses the example of such VMM support to provide entropy to guest operating systems by discussing the KVM virtio-rng mechanism.

Conceptually, virtio-rng establishes a pipe between the VMM /dev/random or /dev/urandom device and the guest. The guest uses its endpoint of that pipe as a noise source for its Linux Random Number Generator.

The operation of virtio-rng with its associated information flow is shown in figure 70. This figure is separated into three main logical compartments:

- One part of the VMM is represented by the host kernel space which hosts the Linux kernel acting as hypervisor. Considering the CPU state, the Linux kernel operates in host context, i.e. unrestricted by the CPU virtualization logic.
- The second component of the VMM is the QEMU process part that implements the emulation of devices and the management of the virtual machine. This part of QEMU operates in host context and is equally unrestricted by the CPU virtualization logic.

- The guest operating system again has a user space and kernel space which both operate in guest context and are therefore constrained by the CPU virtualization enforcement. In figure 70 the guest is assumed to be a Linux operating system as well.

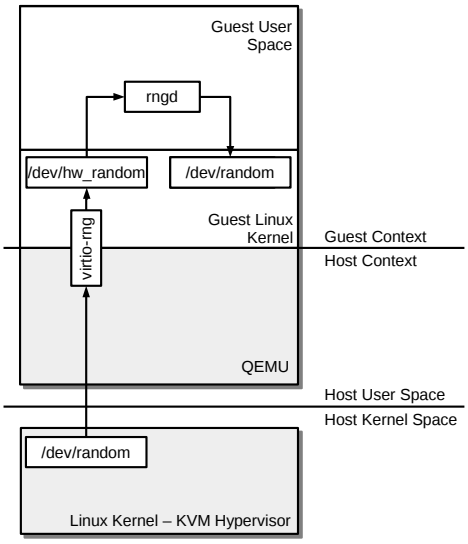

Figure 70: Virtio-rng: information flow between host and guest

The host Linux kernel has full unrestricted access to the hardware and can therefore operate the LRNG in its intended environment.

When virtio-rng is enabled by the administrator of the VMM for the respective guest, QEMU opens /dev/random[54] for reading. In addition, QEMU implements a para-virtualized device, the backend handler of virtio-rng, which offers a virtio interface to the guest as defined in [VIRTIO]. This interface only supports reading by the guest. If the guest tries to read data from that virtio device, QEMU will read the requested amount of data from /dev/random and place it into the virtio ring buffer for the guest to pick up.

The guest Linux kernel implements a virtio-rng device driver, which connects the virtio device with the Linux kernel HW RNG framework. This HW RNG framework exports a device file /dev/hw_random to the guest user space. When the guest user space requests data from that device file, a read request is sent to the virtio device. At this point, the guest user space already is provided access to random numbers from the host.

However, to make the available random numbers usable when pulling from the guest's /dev/random device, the rngd[55] daemon can be used. That daemon is intended to connect /dev/hw_random with /dev/random as follows: The rngd daemon installs a callback at /dev/random which triggers the rngd if /dev/random has insufficient entropy. When the rngd receives the callback, it reads data from /dev/hw_random and injects that data into /dev/random via the RNDADDENTR0PY IOCTL. This IOCTL injects the data into the `input_pool` of the guest LRNG and increases the entropy estimator by the injected amount of data.

Considering now the entire data path, insufficient entropy in the guest LRNG triggers a chain of events which will start a read operation from the host Linux kernel's /dev/random. This data will end up in the `input_pool` of the guest LRNG and increase the entropy estimate in the guest LRNG.

54 Or /dev/urandom, depending on the configuration.

55 In Linux kernels starting with 3.16, the use of the rngd is not needed any more as kernel developers implemented a kernel-internal link between the guest virtio driver and the guest's `input_pool`.

7 Summary of Findings

During the assessment the following major findings were obtained:

- Noise source that are completely implemented in hardware or firmware which operate logically below a virtual machine monitor (VMM) are not affected by the VMM operation. Nonetheless, the interfaces of those noise sources offered to software can be influenced by a VMM and thus impact the random data read by the software. Further assessments are given in section 4.2 supported by the findings of section 4.1.
- Noise sources which are implemented completely in software and do not require specific hardware mechanisms like a high-resolution time stamp are commonly affected in their timing behavior by a VMM at most. Further assessments are given in section 4.2 supported by the findings of section 4.1.
- Software noise sources that require the support from hardware are commonly significantly affected by a VMM operation. Detailed analyses must be prepared for such noise sources to assess whether they still deliver sufficient entropy in virtual environments. As part of the analysis an assessment is required whether the unpredictable phenomenon supporting the noise source is available and usable. The assessments are provided in sections 4.2.3, followed by a discussion of that topic. In addition, section 4.3 provides assessments for various noise sources already. With appendix A a set of questions are provided whose answers can be used as a guide to assess the impact of a particular VMM on a noise source.
- Noise sources allow broad configurations for aspects pertaining to the guest environment as well as to the VMM. With these possibilities, settings can be achieved that are dangerous to the operation of a noise source. A list of common errors and their impact on noise sources is given with chapter 4.2.1.
- Noise sources that derive entropy out of the unpredictability of the exact time of the occurrence of an event – a common approach in most operating system internal noise sources – are affected in their timing behavior by the VMM. The changed timing behavior – disregarding all other types of VMM influence on noise sources – does not alter the obtained entropy in the worst case. In contrast, the entropy will increase due to the VMM operation in normal circumstances. This aspect is further assessed in section 4.2.3.
- As part of this document the different noise sources feeding the Linux /dev/random and /dev/urandom devices are assessed qualitatively and quantitatively when operating as guest in a KVM, VirtualBox, Microsoft Hyper-V and VMWare ESXi environment. During the assessment, no pathological configurations of the used VMMs were identified, i.e. all configurations imply that at least one noise source is active. In addition, the following general results have been obtained:
  - Commonly one or two of the three available noise sources will fail to operate within a virtual environment. /dev/random is affected to the extent that the duration in which processes reading random data are blocked is increased significantly. On the other hand, the cryptographic strength of /dev/urandom is reduced significantly as follows. During the boot time of Linux /dev/urandom receives a cryptographically strong seed much later when operated in a virtual environment. Measurements showed that a sufficiently strong seed was received after more than a minute after boot. This implies that all programs invoked during the boot process that read random data from /dev/urandom are supplied with cryptographically significantly weaker random numbers. More details are given in section 5.6.
  - The unpredictable phenomenon of the block device noise source (spinning hard disks with their physical variations) is replaced with a completely different unpredictable phenomenon (CPU execution time jitter) by the VMM in case the VMM employs a buffer cache. See sections 5.3.2.3 and 5.3.2.4 for details.
  - The heuristic for the entropy estimation applied by the Linux random number generator is almost unaffected by the VMM operation. The entropy is still significantly underestimated. See the test results in sections 5.3 and following.

- The VMM can support the guest for the task of the entropy generation by supplying entropy to the guest. An implementation of the concept with KVM/QEMU for Linux is presented in chapter 6.2.
- General hints on the impact of either KVM/QEMU, VirtualBox, Microsoft Hyper-V, or VMWare ESXi on the Linux random number generator are given in section 5.7. These hints cover configuration aspects and implications when using the respective VMM.

Appendix A. Checklist for Assessment of Virtualized RNGs

The following sections enumerate various noise sources and provide a questionnaire that is to be answered by a VMM developer. The goal of these questions is to identify whether a VMM has an impact on a noise source operation.

All of the given questions are closed questions which only need to be answered with a yes or no.

The questions have associated descriptions indicating that depending on the answer the VMM is likely to have an impact on the noise source or not. When at least one question for a noise source are answered such that the answer indicates an interference of the noise source by the VMM operation, the VMM operation and interference must be analyzed in detail.

A.1 Linux Random Number Generator

The Linux Random Number Generator (LRNG) contains different noise sources that are used to collect entropy. These noise sources potentially are impacted by virtual machine monitors (VMM) and their operation. To assess the impact on the LRNG noise sources, the following questions need to be answered for any given VMM.

The following list of questions do not require any prior knowledge of the LRNG and should be answered by architects of a given VMM.

The following sections enumerate the different noise sources feeding the LRNG.

A.1.1 Linux Random Number Generator Block Device Noise Source

The VMM interference with the LRNG block device noise source is discussed in detail in section 4.3.1.1.

- **Does the VMM directly assigns all block devices used by the guest to that guest?**

  If yes, then no interference from the VMM is applicable to the LRNG access to block devices.

  If no for at least one block device, then the following questions apply for those block devices:

  Does the VMM implement a form of buffer cache which is used to satisfy block device I/O operations from the guest?

  Does the VMM uses a backing store for the block device I/O other than a hard disk with spinning platters?

  Only if both questions are answered with “no”, then the VMM is expected to not interfere with the LRNG access to block devices.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the LRNG.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the LRNG and thus affects the noise source operation.

- **Does the VMM intercept the hardware access to the guest Linux kernel's current clocksource specified in /sys/devices/system/clocksource/clocksource0/current_clocksource?**

  If no, then the VMM is expected to not interfere with the clocksource time gathering of the LRNG.

If yes, then the VMM interferes with the clocksource time gathering of the LRNG and thus affects the noise source operation.

Only if all questions are answered to indicate that the LRNG block device noise source operation is not affected, the VMM implementation can be concluded to not affect the noise source.

A.1.2 Linux Random Number Generator HID Noise Source

The VMM interference with the LRNG HID noise source is discussed in detail in section 4.3.1.2.

- **Does the VMM directly assigns all HID used by the guest to that guest?**

If yes, then no interference from the VMM is applicable for the LRNG HID noise source.

If no, then the VMM affects the LRNG HID noise source.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the LRNG.

If yes, then the VMM interferes with the high-resolution time stamp gathering of the LRNG and thus affects the LRNG HID noise source operation.

- **Does the VMM intercept the hardware access to the guest Linux kernel's current clocksource specified in /sys/devices/system/clocksource/clocksource0/current_clocksource?**

If no, then the VMM is expected to not interfere with the clocksource time gathering of the LRNG.

If yes, then the VMM interferes with the clocksource time gathering of the LRNG and thus affects the LRNG HID noise source operation.

When answering all questions such that the VMM will not interfere with the different aspects of the LRNG HID noise source operation, then the VMM can be considered to not affect the entire LRNG HID noise source.

A.1.3 Linux Random Number Generator Interrupt Noise Source

The VMM interference with the LRNG interrupt noise source is discussed in detail in section 4.3.1.3.

- **Does the VMM offer para-virtualized devices?**

If no, para-virtualized device drivers are used, the VMM is expected to not interfere with the LRNG's interrupt collection.

If yes, do the device drivers in the Linux guest use the standard Linux interrupt handling logic?

If yes, the VMM is expected to not interfere with the LRNG's interrupt collection.

If no – i.e. device drivers bypass the standard Linux interrupt processing, the VMM interferes with the LRNG's interrupt collection.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the LRNG.

If yes, then the VMM interferes with the high-resolution time stamp gathering of the LRNG and thus affects the LRNG interrupt noise source operation.

A.2 Intel RDRAND and RDSEED

Although Intel CPU instructions of RDRAND and RDSEED cannot be affected by the VMM operation, its use by an operating system can surely be affected. The following questions relate to the possible usage of the instructions.

The set of questions are discussed in detail in section 4.3.2.

- **Does the VMM intercept all instructions of a guest for complete system emulation and implements processing of RDRAND and RDSEED?**

  If yes, then VMM is already identified to interfere with these instructions and thus with guest accesses of these noise sources.

  If no, the following question should be answered:

- **Does the VMM configure the x86 CPU to cause a VM-exit for the instructions of RDRAND and/or RDSEED?**

  If yes, then the VMM must be closely analyzed how it handles these traps.

If both questions can be answered with no, the VMM is identified to not interfere with the RDRAND and RDSEED CPU instructions and thus with a guest operating system using them as noise sources.

A.3 CPU Execution Time Jitter Random Number Generator

The full description about how the Jitter RNG is affected by the VMM operation is given in section 4.3.3. The following questions are derived from this section.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, MFTB, STCK/STCKE, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the Jitter RNG.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the Jitter RNG and thus affects the noise source operation. In this case, the following question should be answered to conclude the impact assessment:

- **When handling the intercept of the time stamp gathering, does the VMM issue the same time stamp operation as triggered by the guest, but adds an offset to that value to disguise the VMM's or other guest's execution time?**

  If yes, the VMM intercept can be considered to have no effect on the Jitter RNG noise source operation after all.

  If the answer is no, e.g. the VMM changes the precision of the time stamp by masking some low bits out or by using a completely different time source, the VMM intercept will have an effect on the Jitter RNG noise source operation.

A.4 Apple Mac OS Noise Source

The VMM impact on the XNU's noise source used in the Apple Mac OS and iOS environment is detailed in section 4.3.4 and covered with the following question.

- **Does the VMM intercept the CPU instruction for obtaining a high-resolution time stamp listed in section 4.2.4.2 (e.g. RDTSC, access to CP15)?**

  If no, then the VMM is expected to not interfere with the high-resolution time stamp gathering of the XNU kernel.

  If yes, then the VMM interferes with the high-resolution time stamp gathering of the XNU kernel and thus affects the noise source operation.

Appendix B. Checklist For Avoiding Common VMM Usage Errors

When using VMMs, common usage errors are seen. The following list enumerates suggestions on how to handle VMMs and guests such that a secure state is retained. For more details, please see section 4.2.1.

- The administrator of the cloud environment should only perform snapshots and duplication of guest operating systems when they are shut down.
- When a guest is to be transferred at runtime to another VMM, the administrator must ensure proper error handling to prevent that the state remains duplicated after the termination of the transfer operation.
- The storage location for guest swap space as well for suspend state must be protected at a similar level as the runtime environment of the respective guest.
- When reassigning a device from one guest to another, the state information found in that guest must be deleted unconditionally during that reassignment period but before the receiving guest is taking control of the device.
- If a hardware resource is used by more than one or all guests to provide an aspect or a complete noise source, per definition no entropy can be obtained from such resource. The administrator must be aware of this fact when configuring shared resources.
- As a safe baseline, devices with SR-IOV functionality should not be used as noise sources. Only after a careful study of how the architecture and implementation of a particular SR-IOV card impacts a noise source operation and entropy collection, it may be enabled for use as noise source.

Appendix C. Abbreviations and Glossary

DM	Device Mapper
Guest	Operating system executing within a virtual machine.
HPET	High-Precision Event Timer
HSM	Hardware Security Module
Hypervisor	The Hypervisor is the separation kernel enforcing the separation of virtual machines.
I/O	Input / Output
IID	Independent and identically distributed – terminology used by SP800-90B
IOMMU	Input / Output Memory Management Unit
LFSR	Linear Feedback Shift Register
LRNG	Linux Random Number Generator, which is the entire kernel software component that implements the devices of /dev/random and /dev/urandom on Linux.
LSB	Least Significant Bit: This is the right-most bit.
LVM	Logical Volume Manager
MSB	Most Significant Bit: This is the left-most bit.
Virtual Machine Monitor	The Virtual Machine Monitor is the combination of the hypervisor and the support software implementing full-virtualized or para-virtualized devices.
VMM	See Virtual Machine Monitor
White Noise	From Wikipedia: White Noise is a random signal with a constant power spectral density. In discrete time, White Noise is a discrete signal whose samples are regarded as a sequence of serially uncorrelated random values with zero mean and finite variance.

Appendix D. Literature

- AIS2031: Wolfgang Killmann, Werner Schindler, A proposal for: Functionality classes for random number generators, 2011
- AMD64VOL2: AMD, AMD64 Architecture Programmer's Manual Volume2: System Programming, 2015
- ENTROPYSOURCES: Sonu Shankar, David McGrew, Entropy Sources – Practical Designs And Validation Challenges,
- INTELDRNG: Intel, Intel® Digital Random Number Generator (DRNG), Software Implementation Guide, 2014
- INTELES: George Cox, Charles Dike, and DJ Johnston, Intel’s Digital Random Number Generator (DRNG), 2011
- INTELPROC: , Intel 64 and IA-32 Architectures Software Developer's Manual, 2014
- IOSSEC: Apple, iOS Security, iOS 9.0 or later, September 2015
- JENT: Stephan Müller, CPU Time Jitter Based Non-Physical True Random Number Generator, 2014
- JITTERROSC: Ali Hajimiri, Sotirios Limotyrakis, and Thomas H. Lee, Jitter and Phase Noise in Ring Oscillators,
- LRNG: Stephan Müller, Gerald Krummeck, Mario Romsy, Dokumentation und Analysedes Linux-Pseudozufallszahlengenerators, 2015
- SECROSC: Mathieu Baudet, David Lubicz , Julien Micolod, and André Tassiaux, On the security of oscillator-based random number generators,
- SP800-90A: Elaine Barker and John Kelsey, NIST Special Publication 800-90A (A Revision of SP 800-90) Recommendation for Random Number Generation Using Deterministic Random Bit Generators, 2012
- SP800-90B: Elaine Barker, John Kelsey, NIST DRAFT Special Publication 800-90B Recommendation for the Entropy Sources Used for Random Bit Generation, 2012
- VIRTIO: Rusty Russell, Virtio PCI Card Specification, 2012