Randomness in Virtual Machines

Diogo A. B. Fernandes, Liliana F. B. Soares, Mário M. Freire and Pedro R. M. Inácio
Instituto de Telecomunicações, Department of Computer Science, University of Beira Interior
Rua Marquês d’Ávila e Bolama, 6201-001 Covilhã, Portugal
Emails: {dfernandes,lssoares}@penhas.di.ubi.pt, {mario.inacio}@di.ubi.pt

Abstract—Virtualization technology provided cloud computing with the means to rapidly disseminate throughout the industry and achieve the utility computing long-envisioned era. Efforts on this research area have been focused on assuring isolation between co-resident virtual machines to avoid escaping the sandbox, but less attention has been given to the implications virtualization may pose to the efficiency and quality of random number generation on guests. On Linux distributions, the good provisioning of entropy gathered by the kernel is crucial for the functioning of its random number generator. However, entropy sources may be scarce on virtual machines due to the abstraction implied by the virtualization layer. As a consequence, both the generation speed and the quality of random numbers might drop when compared to hosts. This paper looks into this issue and analyzes the outputs of the /dev/random interface of the Linux kernel on virtual machines. With a well-known statistical library it is shown that the outputs are of high quality and are independently generated, even though they are produced on a slower basis.

Keywords—clouds; randomness; security; virtualization

I. INTRODUCTION

Cloud computing has been on the rise for the last handful of years and it is expected to see it play a bigger role in the future of computing, by streamlining the industry diversity surrounding the technology. Cloud computing shifts computing perceptions by offloading Information Technologies (IT) to outsourced clouds managed by cloud providers through a pay-per-use business model, lowering costs for and increasing business productivity for customers. Clouds can be deployed on a public, private, or hybrid setting, mainly delivering Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), or Software-as-a-Service (SaaS) solutions. These establish the foundations for offering anything-as-a-service. However, clouds encompass numerous security issues [1]. IaaS clouds stand out for their reliance on virtualization that is an enabler for cloud computing to thrive ahead. IaaS wraps up clustered physical servers with hypervisors on top supplying on-demand emulated hardware to Virtual Machines (VMs). VMs can encapsulate entire Operating Systems (OSes) (also referred to as guests) and abstract them from the underlying hardware. This allows to migrate VMs image files easily and scale well on a network of co-resident VMs dwelling over hypervisors.

Virtualization boomed off in mid-2000s when Central Processing Unit (CPU) vendors added full virtualization support to their chips. From a security viewpoint, virtualization raised several challenges, notably related with cross-VM channels [2]. Moreover, guests cannot access underlying hardware in the same manner host systems can. For the specific case of true random number generation on Linux OSes, this might comprise an additional problem. The Linux kernel Random Number Generator (RNG) relies on the good provisioning of entropic events for generating quality random jargon out of the /dev/random interface. However, the Linux RNG may be deprived of an otherwise more diverse and large set of events because of the abstraction layer, and thus output weaker random material on a slower basis.

The good generation of random numbers is critical for a number of areas, notably cryptography, and for applications that assume OSes RNGs to be secure. Because of the significance cloud computing has today and may have tomorrow, it is important to assess if the Linux RNG is undermined on VMs. In this paper, the /dev/random interface of the Linux RNG is empirically studied for its outputs in terms of efficiency and quality on both guests and hosts under various cloud scenarios. The contributions are twofold. On one hand, it quantifies the speed of /dev/random outputs by means of several metrics. On the other, it assesses if random numbers generated on VMs are of good quality and if they are correlated with the ones of co-resident VMs or of the hosts, by means of a stringent statistical library.

The remainder of this paper is structured as follows. Section II gives insight into the problem and discusses related work. Section III describes the approach on the problem, while Section IV gives insight into the problem and discusses the results of the tests. Afterwards, Section V sheds light on workarounds and solutions for the problem addressed in this paper. Finally, conclusions and future work are wrapped up in Section VI.

II. BACKGROUND AND RELATED WORK

The Linux RNG was first scrutinized by Gutterman et al. [3] and later by Lacharme et al. [4]. In a nutshell, the Linux RNG relies on kernel events triggered by noise sources to fill and stir the 512 bytes main entropy pool. Two secondary entropy pools of 128 bytes each rely on the main pool. The blocking pool, from which /dev/random is tied to, and the non-blocking pool, used by /dev/urandom. Among others, typical kernel events include interrupts,
mouse movements, keystroke timings, and disk head seeks. Nonetheless, such events may not be as heterogeneous and frequent for guests as they would normally be for host OSes due to the virtualization layer. As a consequence, the entropy pools may starve. Additionally, since virtualization is sometimes provided over a single hardware system, one may wonder if the randomness generated in co-resident VMs is of lower quality or correlated, since the physical substrate is the same. Low levels of entropy and poor randomness can yield catastrophic consequences on cryptosystems [5], [6]. The /dev/random interface completely uses the randomness provided by its pool to generate random material, and thus it hangs until recharges are dully received. The /dev/urandom does not block because it maintains a Pseudo-Random Number Generator (PRNG) state.

Emulated hardware is created when VM instances are first booted up. This virtualization barrier separates the guests plane from the underlying devices. For example, OpenStack heavily caches block input and output operations [7]. That, combined with Solid-State Drives (SSDs), diskless devices, or servers devoid of arbitrary human interaction via mouse or keyboard, may restrict guests to network interrupts only [8], [9]. This may considerably affect how entropy pools are refilled and the inherent quality. The former can lead to entropy starvation, while the latter may lower the randomness characterizing the material outputted by /dev/random, when compared to normal host circumstances.

VMs may be fired up for short periods to serve specific purposes. As such, there might not be a large enough time-window to develop a sufficiently unique entropy pool [8]. Moreover, snapshotting is useful for backing up VMs states at some point. But restoring snapshots loads the entire OS, including the states of entropy pools and PRNGs, like those of /dev/urandom. Using the same snapshot repeatedly may result with correlated random numbers. Because of that, Transport Layer Security (TLS) sessions and Data Signature Algorithm (DSA) keys might be compromised [5]. Kerrigan and Chen [9] found the cycle variable (part of kernel interrupts) to be highly correlated among Xen guests. The entropy starvation problem is illustrated by discussions throughout the Internet on Amazon Elastic Compute Cloud (EC2) [10], VMware and VirtualBox [11], [12].

This work distinguishes from previous research for its study on the outputs originating from the Linux RNG /dev/random interface under various cloud setups. The pertinence to analyze the /dev/random interface is due to its full reliance on fresh interrupts and events to generate randomness and for being reserved for user space applications. The /dev/urandom interface is not analyzed due to its embedded PRNG and kernel interface get_random_bytes().

III. METHOD AND TESTBEDS

A. General Picture

To address the problem outlined in previous sections, a method was adopted to harvest and process samples of random material originating from /dev/random. A Python script was written to read four byte chunks from /dev/random at a time and write them as unsigned integers, along with a timestamp representing the amount of seconds since the Unix Epoch. Henceforth, a timestamp is referred to as an instant. The resulting sample files are analyzed in two manners. The first is focused on obtaining a set of metrics to quantify the efficiency, while the second converts the 32-bit numbers into floats in the [0, 1) range, so as to submit them to TestU01. TestU01 is a well-known stringent statistical testing library written in C, designed for searching potential statistical weaknesses. These analyzes are repeated for all tests (and scenarios) identified below.

The purpose of TestU01 is to empirically test RNGs under the null hypothesis that, for each integer \( t \geq 0 \), the vector \((u_0, \ldots, u_{t-1})\) of \( t \) successive outputs from a RNG is uniformly distributed over the \( t \)-dimensional hypercube \((0, 1)^t\). This means that the empirical values should imitate independent random variables from the uniform distribution over the interval \([0, 1]\) (i.i.d. \([0, 1]\)), or over the two-element set \(\{0, 1\}\) in case of binary testing. TestU01 provides several statistical batteries that output \(p\)-values, a measure that provides an idea of how close or how far the results are to the null hypothesis. Tests fail decisively for \(p\)-values outside the confidence interval \([10^{-10}, 1 - 10^{-10}]\). TestU01 may be downloaded in [13].

B. Cloud Setups and Configurations

The tests were executed on several cloud-oriented setups, using several technologies and hypervisors, for the sake of impartiality. Locally, three setups were created. A single laptop running Arch Linux with VirtualBox 4.2.12 and Ubuntu 12.04 as guest; a single desktop running Ubuntu 13.04 with VirtualBox 4.2.10 and various Ubuntu 13.10 as guests; and a Xen Cloud Platform (XCP) (version 1.6) using two servers and another desktop. One of the servers provided a Network File System (NFS) to store VM images, while the other server and desktop ran Ubuntu 12.04 guests. Further tests were remotely executed on Amazon EC2 Micro instances running Ubuntu Server 13.04. These setups ran baseline tests by sampling hosts and guests independently and concurrently, co-resident guests, and snapshotted guests. Their goal is to find evidences of undermined /dev/random outputs.

None of the computers had SSD disks, VT-x technology was enabled and usage was kept to a minimum. This, combined with the multi-core CPUs, gives good chances of avoiding CPU time races and thus schedule the hypervisors free of constraints. Except for the memory and disk sizes, the remaining parameters were kept to the default values.
when instancing VMs, including virtual memory size, virtual CPUs, CPU execution cap, and so on. Choosing VirtualBox and Xen allows to analyze the problem on hosted and bare-metal hypervisors, respectively.

C. TestU01 Configurations

Stringent batteries of random numbers usually require a very large amount of numbers. This is the case of TestU01, which we needed to customize due to the Linux RNG slowness, in spite of the lengthy experiments conducted. Specifically, the following tests of the SmallCrush battery had to be manually configured: BirthdaySpacings, Collision, SimPoker, CouponCollector, MaxOft, WeightDistrib, MatrixRank, HammingIndep, and RandomWalk. However, figuring out optimal configurations for each one is not straightforward because each has its own parameters and restrictions regarding their combination. Most adjustments were made in accordance with the notes of the authors of TestU01 for smaller datasets, but some parameters were kept equal across all tests, under a worst case scenario, to make it fair. Such tests remain valid because subsequences of random sequences should be as random as larger sequences. It is fair. Such tests remain valid because subsequences of random sequences should be as random as larger sequences. Each test ran independently of the others, meaning that the same dataset was reused for the different statistics.

IV. RESULTS

A. Datasets

Table I characterizes each dataset obtained from the conducted tests. In the table, \( s \) and \( i \) refer to samples (four byte-sized random numbers) and instants, respectively. The total sampling time and the total number of samples are given by \( \Delta t \) and \( \#_s \), respectively. \( P_i \) denotes the percentage of instants under the total number of samples, while \( P_{\text{comm}(i)} \) denotes the percentage of instants in which samples were collected at the same time in guests and hosts or in two guests. Finally, \( \overline{\Delta t} (\sigma) \) denotes the average value of the differences between instants, while \( \overline{s_i} (\sigma) \) gives out the average number of samples per instant, with the standard deviation of the obtained values represented between brackets. The rate of samples per second is expressed by \( s/\text{second} \). The average values of all the results are depicted in the last two rows of the table. Regarding the \( P_{\text{comm}(i)} \) column, two averages are presented for the guests row. The first (top) one refers to the average value for tests number 3, 5 and 6 (host vs. guest), while the second (bottom) one concerns tests number 4 and 8 (guest vs. guest).

B. Analysis of the Instants

The results included in Table I confirm the expectations regarding the slowness of the Linux RNG when run on VMs. Entropy recharges take longer on VMs than on hosts, and thus \( \overline{\Delta t} \) depicts a fairly high average value of 32.22 seconds, while hosts take about 2.71 seconds to unblock /dev/random. Because hosts are sampled more often, there is the likelihood of reading /dev/random simultaneously with guests, and hence the higher value for \( P_{\text{comm}(i)} \) (48.60%). As such, the total number of samples and instants, as well as the rate of samples, are greater on hosts. An interesting finding was the pattern that is seen throughout all guests regarding \( \overline{s_i} (\sigma) \), while hosts show different values. Notwithstanding, the tests depicted some diversity. The values for \( \overline{\Delta t} \) on VirtualBox guests is higher than on Xen, except for EC2, which turned out to be really slow gathering entropy. Although test number 2 depicted an average value of 2.02 for \( \overline{\Delta t} \), the standard deviation value of 1.75 seems out of place. This points to an unexpected variability in the number of samples per instant, despite the remaining results falling within the aforementioned patterns. Moreover, the laptop host also behave differently, producing fewer numbers than the desktop host with slightly higher
Table II: Results of the \( p \)-values outputted by the customized TestU01 battery with regard to tests number 1 through 12.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hosts (10)</td>
<td>0.41</td>
<td>0.53</td>
<td>0.32</td>
<td>0.29</td>
<td>0.29</td>
<td>0.86</td>
<td>0.48</td>
<td>0.39</td>
<td>0.39</td>
<td>0.66</td>
<td>0.24</td>
</tr>
<tr>
<td>Guests (11)</td>
<td>0.32</td>
<td>0.62</td>
<td>0.76</td>
<td>0.44</td>
<td>0.64</td>
<td>0.97</td>
<td>0.85</td>
<td>0.99</td>
<td>0.03</td>
<td>0.46</td>
<td>0.48</td>
</tr>
<tr>
<td>Hosts and Guests (12)</td>
<td>0.64</td>
<td>0.94</td>
<td>0.29</td>
<td>0.29</td>
<td>0.29</td>
<td>0.86</td>
<td>0.92</td>
<td>0.94</td>
<td>0.16</td>
<td>0.16</td>
<td>0.04</td>
</tr>
</tbody>
</table>

values for \( \overline{\Delta t} \). This may be related with the exceptional usage, mostly during daytime.

To illustrate the discussed facts, Figure 1 contains histograms of \( \overline{\Delta t} \) for tests number 3 (a) and 6 (b). Because \texttt{/dev/random} blocks for some time on guests, an entropy refill only provides sufficient amounts to generate two chunks of random data, while a residual value of 1.09 seconds for \( \overline{\Delta t} \) and a higher 5.72 value is seen on the host for test number 3. Interestingly, the host produces an even number of samples per instant because entropy extraction from the main pool is done ten bytes at a time. Regarding test number 6, the behavior of the guests is maintained, while the host reaches four samples per instant occasionally with usage, including the period between 50 and 60 hours of sampling which comprised normal usage. These results also show that no apparent relation is witnessed between host and guests, because the generation of material on guests does not change for busy and inactive periods.

C. Lack of Entropy — Case Study

The impact of slow entropy generation is herein empirically measured by timing the \texttt{gpg} suite shipped with most Linux distributions. Its \texttt{--gen-key} option creates cryptographic keys uniquely based on \texttt{/dev/random}. The measurements were conducted on a VirtualBox guest and desktop, each running Ubuntu 13.04, and on an EC2 Ubuntu Server guest as well. A 2048-bit RSA key was generated 30 times in order to obtain the mean and standard deviation values of the time elapsed. The EC2 guest took 3052.18 (531.36) seconds, over three times more than 858.99 (257.52), which were the results for the VirtualBox guest. This is expected because the ratio of the \texttt{s/second} between desktop guests and EC2 is almost three. The results for the host are 21.32 (6.49) seconds. Clearly, these experiments illustrate the implications of entropy starvation on the Linux RNG on VMs. Taking so much time to generate cryptographic keys is not feasible in many cases, and thus the Linux RNG may not provision enough material for more keen applications on both hosted and native hypervisors.

D. Quality of the Random Numbers

The datasets were first submitted to less-stringent batteries included in TestU01. The results for those were positive, meaning that the datasets passed the tests. The results for the customized TestU01 are shown in Table II which contains the \( p \)-values outputted by each statistic for each dataset. Each dataset in the table was merged with respect to each dataset of the respective test number in Table I (e.g., the two datasets of test number 3 were merged into a new file), except for the last three that are the results for the merging of all the hosts.
(10), guests (11), and both of these combined (12). Some values for the Gap test were not included because it requires a minimum number of inputs, which were sometimes not available. RandomWalk encompasses five sub-statistics, each outputting a p-value every two steps for a given walk. For example, a total of 30 p-values (five per sub-statistic) are outputted in a ten-sized walk. In the table, they are averaged accordingly to each sub-statistic.

The analysis of the results in Table II suggest that the outputs of /dev/random are of high quality and, most importantly, uncorrelated with the ones generated on co-resident guests or underlying hosts. The two tests marked with an asterisk outputted suspect p-values, possibly due to the small size of the respective datasets. If otherwise, the tests covering all the merged datasets would systematically fail too, which was not the case. Though the datasets from guests represent only 4% to 5% of the resulting merger, it should be noted that such quantity would be enough to trigger failures, if the datasets were correlated.

V. Workarounds and Solutions

One can simply inject /dev/urandom outputs (less preferred) or from specialized protocols (e.g., Entropy Gathering Daemon (EGD)) into /dev/random to overcome entropy starvation. On clouds, hypervisors could address this problem by provisioning seeds and entropy to guests. OpenStack has considered the first [7] solution, while the QEMU virtualizer provides the VirtIO RNG driver to wire up host entropy sources to guests via /dev/hwrng. Regarding the generation of random numbers, Intel implemented a digital RNG on their Ivy Bridge CPUs. The RNG has a raw throughput of around 3 Gb/s by sampling thermal noise. A PRNG is seeded with that to create secure random numbers. This in-built randomness source may be seamlessly integrated across computers, dismissing the need to invest on more expensive and hard-to-manage hardware-based entropy generators like Entropy Key. Intel Secure Key, as it is named, can be exposed through the hypervisor to guests. Denial-of-service would hardly be an option given its efficiency and it would be secure for its natural unpredictability.

VI. Conclusions and Future Work

In this paper, the Linux RNG was studied for its efficiency and quality on various cloud setups. Empirical evidences obtained from several datasets suggest for now that the outputs of the /dev/random interface are not statistically weaker nor correlated with the hosts counterparts or other co-resident VMs. However, the Linux kernel is very slow collecting entropy on VMs, and consequently /dev/random blocks quite often. We are collecting larger samples in order to improve the confidence of the TestU01 results depicted in this paper. It is also planned to devise a means to improve the efficiency of random number generation in VMs without jeopardizing the quality of the generated sequences.

References


