A Study of Entropy Sources in Cloud Computers: Random Number Generation on Cloud Hosts

Brendan Kerrigan and Yu Chen

Dept. of Electrical and Computer Engineering, SUNY - Binghamton

Abstract. Cloud computing hosts require a good source of cryptographically strong random numbers. Most of the standard security practices are based on assumptions that hold true for physical machines, but don't translate immediately into the domain of virtualized machines. It is imperative to reconsider the well accepted security practices that were built around physical machines, and whether blind application of such practices results in the possibility of a data breach, machine control, or other vulnerabilities. Because of Cloud computers reliance on virtualization, access to the hardware based random number generator is restricted, and virtualization can have unforeseen effects on the operating system based random number generator. In this paper, the entropy pool poisoning attack is introduced and studied and a Cloud Entropy Management System is proposed. Extensive experimental study verified that there are measurable problems with entropy in Cloud instances, and the management system effectively solves them.

1 Introduction

A rapidly growing trend is the offloading of computing resources from internally owned and operated infrastructure to the Cloud. Outsourced computing is often cheaper and incurs less business overhead than maintaining private computing infrastructure. Businesses may also build private Clouds, which often reduce the amount of infrastructure necessary by increasing the utilization of computing resources. Cloud computing also brings new challenges, and the aggressive adoption of Clouds could lead to a large population of highly concentrated machines that are vulnerable to different attack vectors.

The combination of multiple users on a single physical machine raises many security concerns. Many of these issues are related to issues that operating systems had to address when they went from single user to multiple user systems. Of main concern is keeping one user's data, activity, programs, and resources separate from all others. In the Cloud arena, this means keeping virtual machine information and state separate from other virtual machines, and insofar as it's possible to keep the virtual machine activity and state separate from the controller that schedules operating systems.

Creating such secure partitions while maintaining the advantages and flexibility of Cloud services is a difficult task. Other issues include the migration of data across networks (sometimes instances are moved due to heavy local machine load), inheritance of security vulnerabilities from platforms used to construct Clouds, trust of the Cloud operators (an unavoidable concession in public Clouds), and new vulnerabilities that emerge from a Cloud's architecture and implementation. Creation of fully homomorphic encryption systems is an essential key tosolving the trust problem of public Cloud operators. Such a system would allow users to encrypt their data, send it to the Cloud for some computational purpose, and the Cloud would perform that computation on the encrypted message, returning an encrypted result. This protects the data because it is never decrypted on the shared system. Inherently the robustness of such a system relies on truly physical random sources. It is critical to identify, demonstrate, and provide a solution to securing the possible vulnerabilities.

The main focus of this work is to explore evidence of weaknesses in the generation of random numbers in Cloud hosts, and provide tools for mitigation of these weaknesses. A thorough background on the generation of random numbers is presented, along with the reasoning why currently accepted methods for random number generation don't translate immediately into the Cloud environment. A series of experiments has been conducted to reveal statistical weaknesses in the way random samples are processed by Cloud guests and the controlling operating system. To address these weaknesses, a customized Cloud Entropy Management System is designed and implemented.

2 Random Number Generation

This section provides a brief explanation on the generation of random numbers on computer systems, focusing on the difference between a true random number generator (TRNG) and a psuedo-random number generator (PRNG) including the PRNG used in the Linux kernel.

Currently computers generally rely on two sources to gather randomness, one which measures hardware noise (hardware RNG), and another that conditions user-input to derive a certain amount of randomness which is used to seed a pseudo-random number generator. An interesting contrast with most computer functionality, generation of random numbers is actually very slow.

Corresponding to the two sources, there are two commonly recognized types of random number generators in computers. The first, a TRNG, measures some physically random features to generate bits. These are bits of entropy; true randomness, especially following some conditioning for the elimination of biases. The second, a PRNG, is a deterministic algorithm which produces streams of random appearing numbers based on some random input 'seed.'

True Random Number Generators (TRNG). TRNGs are somewhat of a burden to semiconductor designers because the standard design requires analog components, at a large power consumption price, and also considerable redesign costs because the analog designs don't scale down quite as neatly as the digital

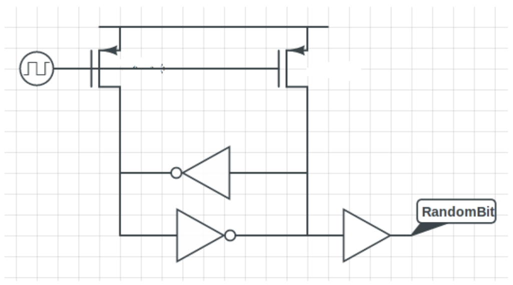

Fig. 1. Schematic of random source used in Intel's Bull Mountain Digital Hardware RNG

counterparts when a new process technology is transitioned to (e.g. 45nm to 28nm) [1].

The basic theory of operation behind a TRNG is amplifying some noise in the system, generally the thermal noise in a resistor, and sampling that signal. This very basic design has some shortcomings such the rate of bit generation and power consumption, and Intel made improvements to it [1]. Instead of simply sampling the noise, the amplified noise was used to control a voltage controlled oscillator. Another oscillator, at 100x the frequency, is then used to create a two bit signal. To remove possible biases, a "von Neumann corrector" is used. The output of the corrector is then run through a secure hash algorithm before it is accessible from its application interface. It could produce random bits at 300Kbit/sec.

Intel has again looked to create a better random number generator recently, this time side stepping the problems with TRNGs to the present by creating an all digital generating circuit [2]. This alleviates the power consumption of analog amplifiers, and simplifies the moving of a design between fabrication technology. Their solution, code named Bull Mountain, violates common digital design practices used to ensure stability in the name of creating randomness. The circuit used as a random source is shown in Figure 1. Note that each input node of the inverter has two drivers, the output of the other inverter, and the drain of a pFET with a clocked gate. When the clock is low, both inverters have their input forced to high. This is where the circuit is in a metastable state, and the final stable output will be detemined by thermal noise that exists because of the resistive losses in the transistors. Each time the inputs are forced into that metastable state, one bit of randomness is produced. Again these raw random measurements are not used directly, but go through some conditioning like the von Neumann corrector. Once finished conditioning, the output should then be used to seed a quality PRNG.

Pseudo-Random Number Generators (PRNG). Psuedo-random number generation is the use of deterministic algorithms to produce a seemingly random sequence of numbers. By itself, a PRNG is a poor source of randomness, as knowledge of the seed value is all that is needed to reproduce the output. This makes the secrecy of the seed paramount. Other attacks are also described that PRNG algorithms can be vulnerable to [6].

There are also a number of different PRNG algorithms, and parameters for the algorithms can have an enormous effect on the statistical quality of the output. Being the goal is a uniform distribution, a good PRNG will generate each number in the entire range, regardless of seed. A chronological background of selected popular generators follows.

For many years, PRNGs were created based on Prime Modulus Multiplicative Linear Congruential Generators (PMMLGC) [13]. A generator, g, is an element of a group, G, which when raised to integer powers, generates a cyclic subgroup belonging to G [15]. In this case, the group is the integers (without 0) over a prime modulus. This relies on some number and group theory results.

A very efficient PRNG can be constructed from shift registers with feedback. These generators rely on bitwise shifts, and usually the XOR operation, and are in general called General Feed-back Shift Registers (GFSRs). However, they do perform poorly on some statistical tests for randomness, such as initialization sensitivity and partitioning problems [3].

An interesting solution to the problem of apparent parallel hyperplanes in these generators is to carefully combine the output of two generators, creating what is a called a combined PRNG [14]. The resultant sequence of pseudo-random numbers, after combination, doesn't exhibit the apparent parallel biasing.

Two less traditional approaches have been proposed in [4] and [5]. In [4] a cryptographically secure pseudo-random number generator, named Yarrow, is created. The main idea in Yarrow is to keep a more conservative estimate of the real entropy gathered from the system, and only keep an entropy accumulator, as opposed to a giant pool with the samples mixed in. The accumulator is then used in combination with one-way hash function to provide random numbers. The width of the accumulator is a major drawback to high performance applications.

The designers of Yarrow relooked at the problem a few years later, with a focus on removing need for highly precise entropy estimation. Their new scheme, named Fortuna [5], relies on keeping 32 pools of entropy, and spreading the entropy across the pools in an even manner.

The Linux RNG (Twisted GFSRs). The Linux kernel provides two character devices which output random numbers, /dev/random (blocking) and /dev/urandom (non-blocking). Cryptographic services are recommended to avoid the use of the non-blocking random numbers, as they could be open to state compromise extension attacks [6]. The quality of the outputs of the random number generator relies heavily on the entropy provided by the input.

The Linux RNG collects entropy by measuring inter-interrupt timing from various sources, mainly the keyboard, mouse, disk read and writes, and network interrupts [7–9]. Cloud instances are starved of the first three, and are largely dependent upon network interrupts for entropy. This has significant ramifications for the security, because these interrupts not only contribute to the virtual machines entropy pool, but also to the scheduling operating system's entropy pool.

The inter-interrupt timings are used as input, along with the current pool contents, to a twisted General Feedback Shift Register. The output is fed back into the pool. Finally if a read of the pool is requested, the pool is used as the input to the Secure Hash Algorithm (SHA). This step is to provide further security to the secret state of the entropy pool, as SHA is considered to be irreversible. While the SHA is currently considered good in that respect, there is a possibility of the algorithm being analyzed where attack is possible. It would be desirable to run SHA on the entropy samples. However, it would have serious overhead when being called so frequently.

3 Distributed vs. Shared Entropy Distribution

In this section, scheduling in the Xen hypervisor is discussed to provide the necessary understanding of how virtualization scheduling leads to hypothetical weaknesses in random number generation in virtual machines, and the virtual machine monitor itself. When it comes to entropy sources on cloud systems, each level of service may offer access to a different type of source. For Infrastructure as a Service (IaaS) systems, entropy sources are distributed among the instances; each instance generates its own random numbers. For Software as a Service (SaaS) and Platform as a service (PaaS), it is possible that all services share a common pool of random numbers.

There are advantages and weaknesses to both approaches. The advantage of a distributed entropy generation is the control in selecting how random numbers are generated, and the separation from possible adversaries. It does however open up the possibility of attacks on the privileged operating system that schedules instances. In shared entropy systems, it is possible for a malicious user to drain entropy from the entropy pool faster than it can be filled, leading to performance degradation of other users and possible denial of service. If a shared entropy system is unable to create random numbers at a rapid enough rate, it would stand to limit access to the pool, or even market the access. In cloud computers that leverage the Xen virtualization hypervisor, high entropy sources are scarce. This scarcity will be explained in the next section.

3.1 Xen Scheduling

The Xen system has a multitude of different schedulers, which use various mechanisms to decide which operating system gets scheduled next. The scheduling of operating systems is analogous to the scheduling of processes within an operating system. The main three schedulers that Xen uses are the Borrowed-Virtual-Time (BVT) scheduler, the Earliest Deadline First Scheduler (sEDF), and the Credit scheduler [12]. There is also a round-robin scheduler, which will be used to simplify the presentation of attacks. The BVT scheduler allows latency sensitive applications to jump the scheduling queue, at the cost of owing that CPU time at a later point [10].

In the sEDF scheduler, anytime a process is scheduled, the priority queue of processes is searched for the one with the most imminent deadline [11]. Both of these schedulers are to be deprecated in future Xen versions. The credit scheduler is the most recommended scheduler by the Xen maintainers, and tasks are scheduled based on a currency based system where Virtualized CPUs (VCPUs) are scheduled in a queue, and 'pay' for real CPU time. Each VCPU receives an allowance from an accounting system.

In Section 6, we use a round robin approximation, assuming an attacker saves his VCPU credits (used as a score in normal Xen scheduler) for a few rounds of scheduling, and hence has a reliably long window to execute the attack.

4 Attacks on Cloud Entropy Sources

Two possible attacks on cloud entropy were conceptualized. One is the random number pool depletion attack that focuses on a shared entropy distribution architecture like that of a PaaS cloud might have, and another one is the entropy pool poisoning attack that focuses on a distributed entropy distribution architecture like an IaaS cloud would likely have. The latter is the focus of this paper.

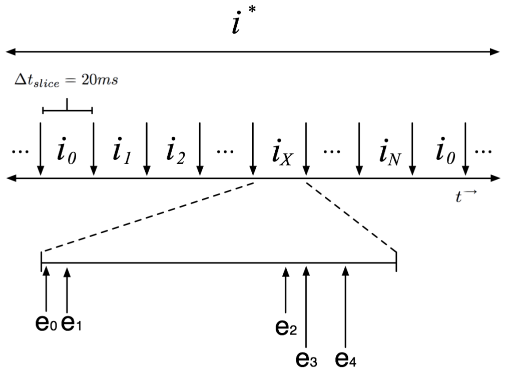
i^*
$\Delta t_{slice} = 20ms$
... i_0 i_1 i_2 ... i_X ... i_N i_0 ...
$t^{\rightarrow}$
e_0 e_1
e_2
e_3 e_4

Fig. 2. Overview of Scheduled Pool Poisoning Attack

Pool Poisoning Attack The entropy pool poisoning attack is a theoretical attack with dire consequences for victims, from attackers being able to decrypt secured traffic (exposing for instance, credit card numbers) up to arbitrary code execution through remote procedure calls (RPCs). From a high level, the attacker tries to make contributions to another user's entropy pool, by generating interrupts with known delays between them. Those delays are run through the Linux RNG as input, and provide some knowledge of another user's entropy pool contents.

$$\begin{aligned} I &= \{i_0, i_1, i_2, \ldots, i_x, i^*, \ldots, i_{n-1}, i_n\} \\ E &= \{e_0, e_1, e_2, e_3, e_4\} \\ \tau &= (\varDelta t_0, \varDelta t_1, \varDelta t_2, \ldots, \varDelta t_m) \end{aligned}$$

Where:

- I: set of n instances on the cloud host
- i_x: attacker instance
- i^*: dom0 operating system
- E: set of attack events (enumerated in detail below)
- τ: Sequence of inter-interrupt timings generated by i_x during e_1

Note that with respect to interrupts, i^* "sees" all the interrupts that the currently scheduled instance "sees". Below is an enumeration of the events in E:

- e_0 Flush the entropy pool (deplete until empty, then stop)
- e_1 Begin sending TCP packets.
- e_2 Stop sending TCP packets.
- e_3 Read the entire entropy pool.
- e_4 Transmit or store before time slice ends.

For clarity, note that in e_1 every TCP packet that is sent is followed by the next $\varDelta t_i$ over the entire sequence τ. Following these steps, the entropy pool of i^* should be full of the contributions matched in the copy of the entropy pool that is stored or sent in e_4.

5 Cloud Entropy Management System

In this paper, a Cloud Entropy Management System is proposed to address the weaknesses of entropy generation on Cloud instances. The Cloud Entropy Management System was created to operate on the cloud hosts to help provide guests with more refined estimates of entropy in their pool. It also provides a mechanism for getting rescue entropy in the case where entropy production from the cloud hosts is insufficient to provide for a workload that requires a large quantity of entropy. This design uses the philosophy of least intervention, allowing the existing kernel PRNG to operate normally, but giving it a more realistic estimate of the entropy in its samples. This allows the kernel PRNG to operate under the assumptions the PRNG creators outlined in the source file (namely that secrecy of entropy contributions is maintained and accurately estimated.) The amount of true entropy bits in a physical cloud host is not very straightforward. In general estimating entropy is difficult, however, it is important to have a reasonable estimate, as this is used as a parameter in the number of samples mixed into the pool. The Linux kernel provides easy access to the estimate in the proc filesystem. The entropy management system looks to provide a more reasonable entropy estimation to overcome accumulation of errors in each sample. Figure 3 shows the operation of the system. Each cloud host runs a server which provides service to the instances that run on it. If a cloud instance on the server is running the Entropy Manager, it will connect on start up to the server. The server provides a count of the number of instances on the cloud, which is used as a divisor for the entropy estimate on the guest. This is a consequence of the principle earlier mentioned that the sum of the entropy of the virtual machines cannot exceed the entropy of the physical host. This divisor is broadcast to clients.

Consequently, the Linux PRNG will use this updated estimate to provide appropriate feedback to internal thresholds used to control generation. Finally, in order to provide for flexibility, the client can request entropy packages from the server. For testing, one source of entropy is a HTTP GET request to random.org for 2KB of digitally sampled atmospheric noise.

There are some security concerns (namely packet snooping between the cloud host and random.org) with this, but it is only used as a proof of concept. For instance, if an attacker was able to intercept the HTTP response from random.org, and knew the hash function used, the system would provide no extra security. There are methods that could be used to overcome this fairly easily, such as employing encryption between the two, or even adding some local entropy to the sample before the hash function.

The server program is designed to allow flexible source selection. This was done in anticipation of new sources becoming available, such as the Bull Mountain source [2]. Emergency entropy is added to the entropy estimate, then the total is checked again to ensure that the estimate hasn't changed in the mean time. Following this, a read is allowed to return.

6 Experimental Study and Results

Building an Experimental Cloud. Initial work was done on the Amazon EC2 cloud, however flexibility required for experimentation required a private Xen Cloud Platform to be setup. The test cloud host was small, however it is sufficient for the testing required. The private cloud host consisted of a Dell PowerEdge 2950 server with the following specifications: 2x Quad-core Xeon E5335 Processors @ 2GHz, 8GB @ 667MHz, and 500GB of Storage.

6.1 Pool Poisoning Attack

The pool poisoning experiment revealed the source of entropy weakness in cloud guests. The entropy contributions used in the Linux RNG consist of the *jiffie, num, and cycle* variables. Each interrupt causes a read on these variables. *Jiffie*

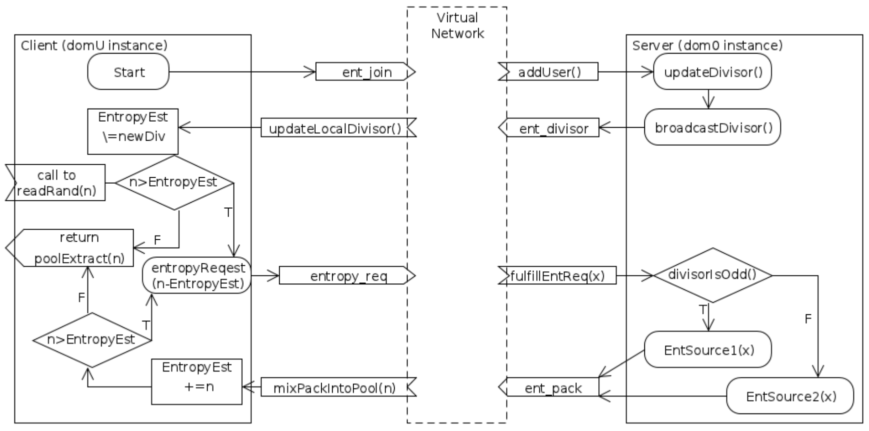

Fig. 3. Activity diagram of the Entropy Management System

is a counter of the number of timer interrupts generated (every 4ms in Linux). *Num* indicates the type of interrupt, such as keyboard, or network. *Cycle* is a free-running counter register available in x86 CPUs that runs at the clock frequency.

Invariability of jiffie and num. In Figure 4, the invariability of both the jiffie and num variables with 4 guest instances are demonstrated. After the first 300 seconds, the jiffies quantity is invariant for both domain 0, and the guest instance. This problem is likely filtered by the hypervisor, so no guests are affected by it. The num variable however, has significant change in all cases for domain 0, but is completely static for all guest instances. The same invariability is observed in the experiment over one and two guest instances.

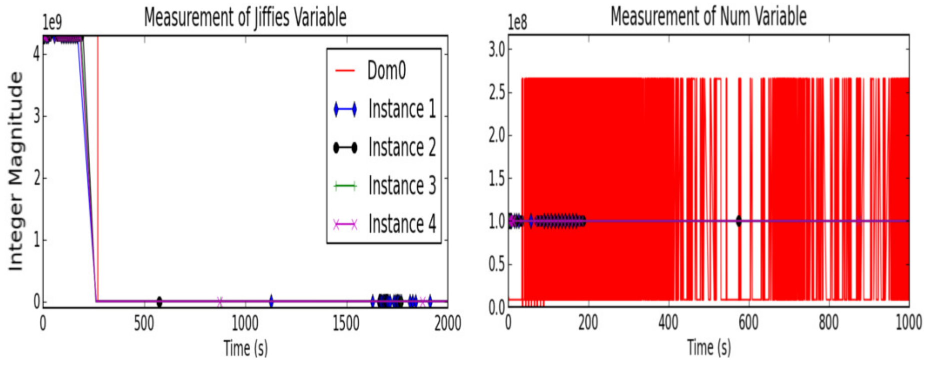

Fig. 4. Jiffie and num variables with 4 instances

The cycle variable. The cycle variable has a much more interesting, and entropy rich behavior across all instances, both guests and domain 0, and are shown in Figure 5. The guest instances show tight coupling between the cycle variable and each other, however the domain 0 instance is largely independent of any guest instance. In Table 1, the correlation and covariance of the cycle variable between the guest instance and the dom0 OS is rather low. The covariance for each pair is normalized to the first listed instance's covariance with itself. The correlation across guests is much higher than that of the cases with the domain 0 OS, and is shown in Table 2.

Table 1. Guest Instance Cycle contributions correlated with Dom 0 Cycle Contributions (4 guests)

Instance (with Dom 0)	Correlation	Covariance
DomU1	0.177657311	0.2019764114
DomU2	0.3175715104	0.2947513962
DomU3	0.4681160782	0.5268961915
DomU4	0.3753263986	0.3768116959

Table 2. Correlation and Covariance of Entropy Contributions Across Guest Instances

Instance Pair	Correlation	Covariance
2 Instances		
DomU1-2	0.7698154055	0.7775670479
4 Instances		
DomU1-2	0.6424939995	1.6548446143
DomU1-3	0.7283067151	0.7936474751
DomU1-4	0.893786794	2.2162510283
DomU2-3	0.8870338305	1.0030611841
DomU2-4	0.9862760451	0.9635573195
DomU3-4	0.8700133244	1.0424442764

6.2 Analysis of Results

The assumed ability to contribute to the dom0 entropy pool by generating interrupts on the guest instances turned out to be largely incorrect. While this makes the pool poisoning attack unlikely, the correlation between the guest instances was remarkably high over large runs of the samples. This violates the assumption made by the designers of the Linux RNG that the contributions are uniquely

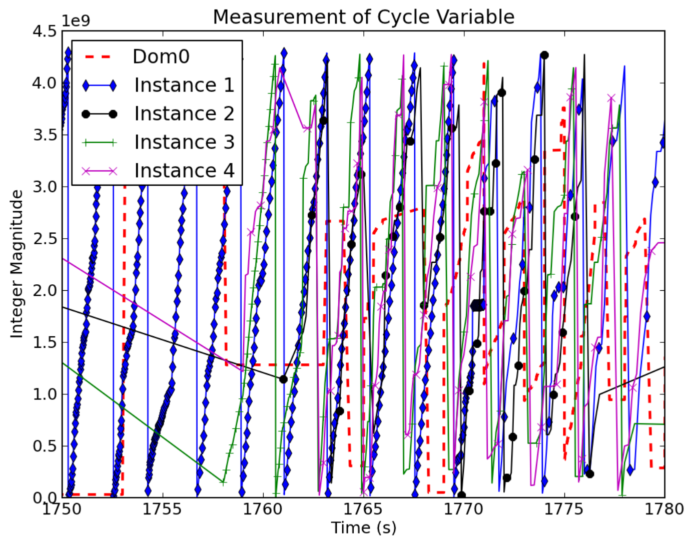

Fig. 5. Cycle variable with 4 instances

random; unknowable to anything but the RNG. The cycle variable is measured by the assembly instruction rdtsc. It is zeroed on a reset of the processor, and increases every clock cycle. The correlation is likely the result of the free running counter register being synchronized among cores, which is typical for multicore processors, though not guaranteed. So highly correlated guests were scheduled on the same CPU but a different core.

The general invariance of the rest of the two-thirds of each sample does negatively affect the entropy estimate of contributions to the pool. The num variable was invariant due to its source, the type of interrupt generating the event. For instance, a keyboard event would pass the keyboard scancode to the num variable. Being that guests receive all their I/O from the network, there is no variation. The invariance of the jiffie variable is most likely due from Xen trying to handle the jiffie clock for guests. Entropy generation rates are negatively affected by this. Therefore, the proposed Entropy Management System does serve a purpose to provide an emergency source of entropy.

7 Discussions

Overall there aren't any glaring practical security vulnerabilities that are demonstrated in the Xen Cloud Platform, however the rate of recovery for guest instances is likely overestimated, which leads to the conclusion that the estimates were inflated throughout the tests. While this makes a theoretical cryptanalytic attack hypothetically possible, it is not much different from other hosts which suffer from overestimation. This is the reasoning behind the use of a dividing mechanism in the Entropy Management System. It's far better to underestimate the entropy than it is to overestimate it. If a guest instance is to be cloned, it ought to have its pool drained before cloning. Upon start up of the cloned instances, the pool should be again drained, and finally any keys should be regenerated. While the step of draining the pools after cloning was performed in tests, SSH keys were not regenerated, and as a result the RSA fingerprint for all instances was identical. Mechanisms for dealing with the shared state of cloned instances, namely breaking the common state for things that require uniqueness, would be a fruitful endeavor in securing cloud instances. Carelessness of users cannot be underestimated, and tools to automate these chores would be very useful. Another area which is hypothetically weak in respect to Cloud entropy is the loadbalancing mechanisms and cloning mechanisms. There are times when the instances may be loadbalanced over a public network. If an instance is cloned or loadbalanced across a public network, it would also be prudent to drain the pool, and regenerate keys.

8 Conclusions

In this work, we explored the potential weaknesses in the generation of random numbers in Cloud hosts, and provided tools to mitigate these weaknesses. First, the evidence of entropy coupling between domain U instances in Xen Cloud Platform hosts is revealed. There is a possibility of prediction of the variable given enough instances are under the control of an attacker.

Second, our experimental results show that virtualization affects entropy sample collection. The num variable in particular did not change once in any of the experiments, while the jiffie variable did exhibit one change throughout all the experiments, apparently most changes being filtered by the virtualization layer. Finally, the high correlation of the cycle variable is concerning, and makes a case for entropy gathering on Xen guests to be managed differently in the kernel.The use of the Cloud Entropy Management System sidesteps the problems with the correlation of samples by providing bailout entropy that is uncoupled. The implementation is low overhead, and consists of two daemons written in Python. It provides a reasonable way to ensure entropy estimates and entropy pools in Cloud guests aren't susceptible to exploitation.

References

1. B. Jun and P. Kocher, "The Intel Random Number Generator," Cryptography Research Inc., white paper prepared for Inter Corp., Apr. 1999, `http://www.cryptography.com/resources/whitepapers/IntelRNG.pdf`.
2. G. Taylor and G. Cox, "Digital randomness," *IEEE Spectrum 48*, Sept. 2011.
3. G. Lian, "Testing Primitive Polynomials for Generalized Feedback Shift Register Random Number Generators". `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.318&rep=rep1&type=pdf`

4. J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator". Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999. `http://www.schneier.com/paper-yarrow.ps.gz`.
5. N. Ferguson and B. Schneier, "Practical Cryptography". John Wiley & Sons, pp. 161-182, 2003.
6. J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic attacks on pseudorandom number generators", Fast Software Encryption, Fifth International Proceedings, pp. 168-188, Springer-Verlag, 1988.
7. Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the linux random number generator. In Proceedings of the 2006 IEEE Symposium on Security and Privacy. IEEE Computer Society, 2006.
8. M. Mackall, Linux Kernel Source 2.6.32.8 Random Character Driver, (/linux2.6.32.8/drivers/char/random.c in kernel source tree)
9. T. Beige, "Analysis of a strong Pseudo Random Number Generator by anatomizing Linux Random Number Device". November 2006. `http://www.suse.de/~thomas/papers/random-analysis.pdf`.
10. K. Duda and D. Cheriton. "Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler." In Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99), Dec. 1999.
11. "Earliest deadline first scheduling" Internet: `http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling`, 4 December 2010 [April 26 2011].
12. J. Mathai "Scheduling - Xen Wiki" Internet: `http://wiki.xensource.com/xenwiki/Scheduling`, 09 June, 2007 [7 May 2011].
13. S. Park and K. Miller, "Random Number Generators: Good Ones Are Hard to Find", *Communications of ACM*, vol. 21, no. 10, Oct. 1988.
14. P. L'Ecuyer, "Efficient and Portable Combined Random Number Generators," *Communications of the ACM*, vol. 31, no. 6, pp. 742-774. 1988.
15. C. Carstensen, B. Fine, and G. Rosenberger, "Abstract Algebra - Applications to Galois Theory, Algebraic Geometry and Cryptography". Heldermann Verlag. 2011.