Randomness Exposed – An Attack on Hosted Virtual Machines

Christopher J. Thompson, Ian J. De Silva, Marie D. Manner,
Michael T. Foley, and Paul E. Baxter
Department of Computer Science
University of Minnesota
{cthomp, desilva, manner}@cs.umn.edu, {fole0103, baxte075}@umn.edu

Abstract

The wide-spread adoption of virtualization technologies by the industry has enabled new innovations, but has also broken existing assumptions about the security of the system. While virtualization makes multiple operating systems, shared hardware, and cheap storage possible, it brings with it new and unforeseen attacks and potential security holes. In this research paper, we explore a possible attack in which the host operating system records the entropy used by the guest and predicts the guest's generated pseudo-random number. This allows the attacker to break some of the guest's cryptographically secure numbers or communications, such as session keys.

1. Introduction

Hosting services, such as Amazon's Web Services [1], have long been considered trusted entities when it comes to virtualization. These services control the hardware, so logically, they could look at any data in memory or on drives. Encrypted storage helps provide security for the data on the hard disk, while new technologies, such as Virtualization Technology for directed input/output (VT-d) [2], are beginning to provide some ways to secure the guest partition's memory.

We are still left with an assumption that the virtual machine manager (VMM) and/or host partition is uncompromised. What assurances can be provided to a client of a hosting service that 1) an insider (such as a disgruntled employee) has not compromised the hypervisor/host partition, and 2) malware has not been installed on the host partition? This may not have detrimental effects if the client and its data is encrypted, but what if the attacker can record the entropy being fed into the guest operating system (OS)? Could the attacker break any keys generated on that system?

In this paper, we will explore our hypothesis that an attacker can, using a compromised host, record the entropy being passed into the guest's pseudo-random number generator (PRNG) and predict its output. We will use KVM, a popular open-source virtualization system built on the GNU/Linux operating system, for this attack.

We chose KVM because it is in widespread use and, unlike Xen, does not currently provide a mechanism to all users on the host for accessing guest memory [3]. This is particularly relevant because, as we will discuss in section 3, we focus on a weaker adversary than the super user of the host machine.

Section 2 will outline existing work in this area and 3 will expand upon our threat model. We will then look briefly at some of the theory behind our hypothesis (section 4) before detailing the set-up and approach (sections 5 and 6). Finally, we present our results, future areas for research, and concluding remarks (sections 7, 8, and 9).

2. Related Works

We begin with a brief discussion of works on pseudo-random number generators used by operating system kernels, then describe the PRNG used by the Linux kernel. We then present previous research showing that virtual machines are both theoretically and provably cryptographically insecure. Section 4 describes entropy and uses; for our purposes here we use the terms good entropy (new, private, uniform bits) and bad entropy (reused, exposed, predictable, or adversarially-chosen bits) [4].

2.1. Pseudo-Random Number Generators

Pseudo-random number generators (PRNGs), such as that used in OpenSSL [5] and the Linux kernel, are available as source code with varying levels of documentation. Research has analyzed the Linux PRNG (LPRNG) and suggested various other designs, as well as evaluated current PRNGs in use by other systems.

Attacks on PRNGs have been known for over ten years- such attacks include direct cryptanalytic, input-based (the sorts of bad entropy mentioned above), and state compromise extension, which includes backtracking, permanent compromise, interative guessing, and meet-in-the-middle attacks [6].

Cryptographic operations using methods from OpenSSL use the PRNG's $rand(3)$ function. The OpenSSL documentation advises seven items for a good PRNG: a good hashing algorithm, an initial source of random state, a very large state, influential seed data, inability to extract data used to seed the PRNG from the PRNG state, potential for two identical states to change over time, and inability to predict subsequent random numbers given a state [7].

Researchers have devoted much time and effort to designing new PRNGs that resist known attacks or at least make the new PRNG less likely to fail against such an attack. The Yarrow PRNG is built in this way, designed with efficiency, attack-resistance, fast recovery from compromise, and smaller likelihood of failure [8]. The Yarrow PRNG was designed to:

- estimate starting entropy conservatively
- use a computationally expensive reseeding process
- simplify seed file management
- simplify the interface
- be based on an apparently secure block cipher
- avoid data dependent execution paths
- process input via hash function before adding to the entropy pool
- reseed current entropy only after separately collecting new, unguessable entropy
- limit backtracking
- explicitly reseed pool before generating high value keys
- separate activities of collecting entropy, reseeding the pool, generating outputs, and deciding when to reseed
- use 2 pools for fast and slow reseeding
- use commonly available, secure hash functions and block ciphers

There are further restrictions on each component of the generator, which will not be discussed much further here. The components strive to, among other things: limit the possibility of one source contributing most entropy to the pools, carefully time when to refresh and empty the pools, use secure hash functions, and conservatively estimate entropy sources.

2.2. Linux Kernel PRNG

In the Linux kernel's PRNG – located at `/drivers/char/random.c` [9] – we know that timings, among other sources, are mixed into the pool with a CRC-like function. Usually event IDs are gathered from device driver interrupts, keyboard and mouse events, and other sources, then paired with timing information; this information is added to the entropy pool and mixed.

In 2006, Gutterman and Pinkas analyzed the Linux PRNG and found that three pools exist for holding entropy: primary, secondary, and urandom [10]. Entropy is added to the primary pool from various sources; when this pool is full, entropy is added to the secondary pool. Output from the primary pool goes to the secondary and urandom pools, and the Linux PRNG output comes from these two pools. Each pool keeps estimates of its own amount of entropy.

If an application requests entropy from the blocking `/dev/random` interface, the secondary pool will try to provide the requested entropy. If the secondary pool doesn't have enough bits, it will transfer bits from the primary pool. If this fails, the call is *blocked* and will not return entropy until the primary pool receives input.

If an application requests randomness from `/dev/urandom`, bits are taken from the urandom pool, which is refilled by hashing extracted output from both the urandom and secondary pools with SHA-1. Requests from urandom are non-blocking, meaning the call will always return reasonable (but theoretically predictable) entropy. When a sequence of bits is requested from either pool, the result is generated by computing the SHA-1 hash of the pool contents, and lowering the corresponding estimate of the amount of entropy contained in that pool.

Because events may occur in a predictable sequence, users are also advised to include a short script to seed and save the entropy pool upon start up and shut down, respectively. This is intended to make the entropy pool harder to guess when the attacker has no prior knowledge about the system's pool upon last shut down [9].

2.3. Virtual Machine Vulnerability

Virtual machines have unprecedented ability to compromise networks of systems, due to the ability to roll

back a machine to any previous, potentially unpatched and insecure state [11]. Because virtual machines can be saved, deleted, rolled back, or shared as easily as a file, companies are able to leverage them to run multiple versions of software and operating systems on a single machine. While patch updates can easily be applied to a virtual machine's operating system, the system can just as easily be rolled back to a pre-patch and therefore vulnerable state. Such vulnerabilities apply to cryptographical needs – random numbers, nonces, and even one time use keys are no longer guaranteed to be random. Furthermore, the virtual machine manager must log changes so the machine can be rolled back, logging even sensitive information like cryptographic keys or private documents. We see that the ease of use and high utility of virtual machines comes with the cost of potential vulnerabilities.

One possible attack against VMs is to reset the VM to a previous state and see if the pseudo-random numbers or encryption keys generated afterwards are the same as before the reset. This attack was shown to work against VMWare and VirtualBox; certain cryptographic operations, specifically key exchange and signing, used the same entropy after a virtual machine state reset [4]. We refer to these state reset and subsequent broken cryptographic operations as VM reset vulnerabilities or attacks. The VM reset attack, which causes bad entropy, along with oft-documented failure to provide randomness on the part of the PRNG, severely compromises common systems such as public-key encryption, symmetric-key encryption, and digital signatures. While VM reset attacks are shown to work against clients (i.e., web browsers seeking secure http connections) and servers (responsible for generating and sending session keys) there is a fix, hedging, that works reasonably well against bad entropy and therefore VM resets.

Yilek describes the framework for hedging thusly. Use a deterministic *hedging function* which takes arbitrarily sized bit strings R and some number p of data bit strings $(d_1, \ldots, d_p) \in (\{0,1\}^*)^p = d$ [4]. Originally the operation Op performs $Op(i_1, i_2, \ldots, i_k; R)$, where R was supplied by the PRNG and i is the input. Replace this call with $Op(i_1, \ldots, i_k; Hedge(R, \langle d \rangle))$ where $\langle d \rangle = (OpID, i_1, \ldots, i_k)$ and the cryptographic operation Op gets an identifying number $OpID$. Yilek shows HMAC to be a reasonable hedging function for *Hedge*.

Hedging showed at least some improvement for the three systems mentioned above, and even seemed to fix digital signatures completely. Hedged cryptography works regardless of encryption system and PRNG failure, and focuses on improving individual primitives. Because this has not been adopted in the LPRNG, we leave evaluation of the effectiveness of this method against our proposed attack to future work.

3. Threat Model

This research takes the approach of a malicious or compromised hosting service. The service may be working on behalf of a government, may have allowed the hypervisor/host partition to become infected with malware, or may simply have an untrustworthy employee.

Garfinkel et al. proposes that the PRNG should be a resource that the virtual machine manager (VMM) provides and guests pull from [11]. However, this mechanism, while helping to ensure that pseudo-random numbers appear random, is more susceptible to a malicious/compromised hosting service than the current model.

We operate under several assumptions. First, the guest system can detect if it is not getting as many cycles as was purchased. This allows the guest some mechanism to detect if it is being migrated to another system or the state is saved in some other way. Secondly, hardware virtualization enablement technologies, like Intel's VT-d, do not have directed I/O enabled [2]. We use the second assumption because it seems as though many hypervisors do not yet support this technology [12].

If an administrator has access to a management partition (DOM0), but does not have direct access to the hypervisor, he likely will not be able to read all of memory or view the state of the guest's entropy pool. He may still be able to read or affect the entropy that is reported to the guest because, in our model, it is fed into the guest by the host. This model, while somewhat specialized, could be implemented in an environment using SELinux and/or encryption. If the virtual machine itself is protected from the administrator by one of these tools, then the administrator cannot look directly into the virtual machine itself to determine the state of the PRNG.

4. Theory

In this section, we refer to entropy as the unknown bits in the pool of bits available to the system for applications that require random numbers, called the 'entropy pool', from the point of view of the attacker, not the machine. An entropy pool can be defined as having classes of states. The first is *known state*, where the entropy is 0 and all bits are known to the attacker. The next is the opposite of that, or *unknown state*,

where the entropy is N bits in an N-bit pool, or 100%; all bits are unknown to the attacker. Lastly are the states in between, or the *partially unknown states*, where $0 < P < N$ bits are unknown in an N-bit pool.

4.1. Entropy Pool States and Transformations

All seeding into an N-bit entropy pool from any source can be defined in a deterministic system as a transformation from N-bit to N-bit space with a Q-bit input, where Q is the bit size of the seed. For example, a 1024-bit entropy pool may be fed a 64 bit value and transformed into a new 1024-bit pool. This transformation can be expressed as $F : \mathbb{F}_2^N \times \mathbb{F}_2^Q \mapsto \mathbb{F}_2^N$, where F is a known transformation. In the Linux PRNG, the input to the transformation of the entropy pool is a 64 bit time stamp counter, a 32-bit jiffy (a unit of time within the Linux kernel used for timer interrupts - see [13] for more information), and a 32-bit event identifier summing to 128-bits, giving us $Q = 128$ [9].

Given a known state entropy pool N, the next state given after a transformation of Q-bit unknown input will have an effective entropy of Q. That is, though F maps to an N-bit output, we need only iterate over this transformation for the domain of Q in order to find the next state via brute force, provided some method of verification. This will constitute a cryptographic break if $Q < N$, as it reduces an N-bit problem to a Q-bit problem. This transformation can be chained to evaluate the entropy after a series of inputs. When an Q_2-bit domain transformation is applied to the partially unknown state entropy pool given after a Q_1-bit transformation to a known state entropy pool, the resulting state will have a Q_1+Q_2-bit entropy. That is, we could exhaustively iterate over all transformations in domain Q_1, which would then need to be exhaustively mapped and iterated over for all transformations in Q_2. Any series of transformations to N can thus express the entropy pool as $\prod_x (2^{Q_x})$ for x transformations each with bit size Q_x.

For example, a 4-bit, 8-bit, 16-bit and 32-bit domain sized transformations applied in a row would have an effective entropy of $4 + 8 + 16 + 32 = 60$-bits. This would take 2^{60} iterations to resolve. It is trivially evident that once the product of the entropy of all transformations exceeds N bits, the attack is no longer a break; it is faster to exhaustively search over the N-bit domain of the entropy pool. Hence an attack with multiple unknown transformations will multiplicatively degrade and lose its breaking power within a very small series of transformations. For example, in a 1024-bit entropy pool machine, if a single 8 unknown bit transformation is measured in between verifications, the entropy of the pool is only 8 bits, trivial to brute force. But if fifty transformations of 8-bit size occur in between verifications, the effective entropy of the end state is $8 * 50 = 400$ bits, which is beyond the processing power of modern technology.

This rapid degradation is slowed if the input of each transformation is partially known. If P of the bits in a Q-bit transformation are unknown, where $0 \leq P_x \leq Q_x$ then it follows that the entropy of a chain is $\prod_x (2^{P_x})$ for x transformations each with Q_x-bit input of unknown P_x-bit uncertainty.

For example, if a 128-bit seed is provided to the entropy pool from a disk interrupt as is the case with the Linux PRNG [9], but the attacker knows that disk interrupts will fall within a range expressed by 16-bits, then the attacker need only search over a 16-bit size for that transformation. This P-bit portion can be seen as a measure of integrity of the input methods. The LPRNG has predictable bits in its input [4], which may carry a low entropy; a 128-bit seed may have far lower than 128-bits of entropy. On the other hand, a very cryptographically secure machine may use a hardware dongle that grants an effectively random input to the attacker, making $P \approx Q$. But a lax machine in a lower security environment may use somewhat predictable hardware or network inputs to seed its pool, allowing a substantial reduction in the entropy of each transformation. Also, any transformations in which a seed is not known, but is reused from an early unknown seed and the reuse is known to the attacker, has a $P = 0$, i.e., no entropy. Once the first transformation has been determined, the seed is known and further transformations can be reduced to a single state, granting no entropy.

When the attacker has some means to engage in a chosen input attack, the number of unknown bits of each transformation can be lowered or made 0. For example, if an attacker has gained complete control of a seeding input source of the PRNG, it can be assumed that all transformations generated from seeds given from that device have $P = 0$ and are known, taking only a single calculated transformation to reconstitute. Thus all transformations from a compromised device can be reduced or made trivial in entropy. If the attacker has some method of influencing but not controlling the seed of a device, for example by spamming network packets from an immediate node to a machine that uses network timing as a seed, that entropy will have a P value less than Q but not necessarily zero. Thus, if an attacker can control a portion of the inputs to the transformations on an entropy pool, the entropy of the pool formed by reconstituting the transformations can be greatly reduced.

Given this information, we know that if the host is able to monitor injections into the guest's entropy pool, the host OS can drastically reduce the possible number of states of the guest's entropy pool. With perfect monitoring, the host should be able to know the exact state of the guest's entropy pool, given an initial value. Since virtual machines make frequent use of snapshots to duplicate machines or restore a clean state, a malicious host can determine this initial state simply by using a snapshot with a known entropy pool.

4.2. Entropy Verification

Crux to this attack is a need for some form of verification- a method by which we can confirm that the transformation from a known entropy pool is equal to the unknown next state. Verification may be as trivial as generating a random number using this pool, or complicated by the volatility of a pool after each transformation and the need for multiple random numbers to test. As more than one pool may map to the same random number for a given range, a single test may not suffice.

After verifying that the entropy pool from a series of transformations on the known state is equivalent to the unknown state we are testing for, the unknown state becomes known and has an entropy of 0. Thus, it is important *when* we can test an entropy pool. If two $P = 32$-bit transformations are applied to an entropy pool, but we are able to verify the entropy pool in between and after the transformations, then the attack is reduced to a 2^{32} instead of $2^{32+32} = 2^{64}$ entropy system. The output of the first transformation would be known after 2^{32} tests, and then that known state could be tested for the final output after another 2^{32} tests. Hence any series of transformations could express the big O time required to resolve the entropy as only the largest product of unknown bits in the inputs of all transformations in between verifications, written as $O(\prod_x(2^{P_x}))$ for x transformations in between two verifications, for the largest $\prod_x(2^{P_x})$, with input bit size Q_x of P-bit unknown, where $P \leq Q$.

As a consequence, our attack is limited to only the weakest link in our series of transformations. Even if an attacker is able to seize complete control of network timings and disk interrupts of a system and reconstitute the 1024-bit entropy pool after millions of transformations, if just two transformations of 32-bit size occur in tandem in between verifications, the final entropy pool still has an entropy of 64 bits, which may put the attack out of the range of modern technology. Meanwhile, if an attacker is able to pause the OS and arbitrarily verify his entropy pool after every single transformation, the attack can be reduced to only the largest P value for all inputs.

4.3. False Positive Verifications

However, verification introduces a degree of uncertainty because in the Linux PRNG, all bits pulled from the entropy pool are hashed by a SHA-1 algorithm before being returned [9]. As a direct consequence, there is a chance that more than one entropy pool state may map to the same hashed output, resulting in a false positive when attempting to verify the pool. While the output for that same set of bits may be the same, further transformations of the entropy pool will diverge. Assuming a perfectly random hash algorithm, the odds of any random input giving the same hashed output as the state we are verifying is 1 in 2^V for an V bitsize verification. That is, when a 1024 bit entropy pool with 2^{1024} combinations is hashed to one of 2^{1024} possible outputs, the odds of any other input mapping to the same output are 1 in 2^{1024} for a perfectly random hash function. Yet if a 1024 bit entropy pool is mapped to a 32 bit verification, there is a 1 in 2^{32} chance that any given random state will map to the same output as the correctly verified state. While this chance is minute in a small data set, the uncertainty spirals upwards rapidly as the number of rounds between verifications and states that must be iterated over exponentiate. If a single state has a $1 - ((2^V - 1)/(2^V))$ frequency of returning a false positive on verification, than for r total states being verified, assuming a randomly distributed correct state in the iteration of false states, there will be a chance of returning a false positive before identifying the correct state expressed as $1 - ((2^V - 1)/(2^V))^{(r-1)/2}$.

While the odds of hitting a false positive before the correct verification occurs are statistically insignificant when exploring small sets of transformations with large output bitstrings, the chance of a false positive puts a high degree of uncertainty when the output bitstrings are smaller and the set of transformed states to verify is large. For example, in a system with $r = 2^{32}$ for a total of 2^{32} states to iterate over, with only $V = 32$ bits pulled per verification, the odds of returning a false positive before the correct state is 39.35%. This puts a theoretical upper bound limitation on our attack as a break; the uncertainty of a series of s verifications each with r_s states to verify between verifications will have a $1 - ((2^V - 1)/(2^V))\prod_s((r_s-1)/2)$ probability of returning a false positive at some point, rendering the end state no longer valid.

An idealized attack might pull a full 1024 bits from the PRNG's 1024 bit entropy pool and operate on

only a small set of states to iterate over between verifications, but this may not be the case in any practical application. For example, a process constructing a secure cryptographic key may request a random 32 bit integer for a non-secure function which is observable to the attacker, allowing a verification of the entropy pool but only with a high degree of uncertainty.

In the frame of our attack, the odds of running into any false positives on a system running the Linux PRNG are negligible for any verifications of a large bit size. If the entire pool is pulled at 1024 bits, the chances of a false positive do not become significant until long after the amount of iterations that would be viable on modern architecture. However, the false positive problems present a hurdle in real world applications of the attack- any smaller bitsize verifications would be restricted by higher uncertainty; a 32 bit integer may introduce an unacceptable 1.55% false positive rate at 2^{27} states, when an attacker may be able to brute force 2^{56} iterations.

5. Experimental Setup

We use the Linux-based application, QEMU-KVM, to host and attack various guest machines on Linux. We seek to break otherwise cryptographically secure communication by listening in on the guests' system interrupt information. The KVM application is described in section 5.1, and the adversary is described in section 5.2. A hardware and software description of our machine is given in section 5.3.

5.1. KVM

The Linux Kernel-based Virtual Machine (KVM) [14] is the hypervisor built into the Linux kernel. Unlike Xen, it is most frequently used with unmodified guests.

KVM uses the user-space of the host as the management domain (sometimes called the host partition or DOM0). The actual hypervisor code resides within kernel-space. This separation allows policy to be created using standard Linux mechanisms, including SELinux.

We focus on the scenario of a host using KVM to host other Linux guests. We assume that the guests are headless—i.e., they do not receive keyboard or mouse events and are not running GUI environments—which is common for virtual-machine hosted servers accessed remotely via SSH.

Systems do exist to provide more entropy inside guests, such as entropy brokers [15] [16], which can even pull from truly random sources, but would still allow the host to log all entropy input passed into the guests. We leave the security implications of entropy brokers to future work.

5.2. Adversarial Model

In our analysis, the adversary is an administrator on the hosting provider, who does not have access to core dumps of the guest or to the contents of the disk.

This adversary wants to be able to predict or influence the random bits that the guest receives from its entropy pool on reading `/dev/random` or `/dev/urandom`, the block devices for accessing the Linux PRNG.

The victim is most likely a client of the hosting provider, running some sort of cryptographic software that relies on good randomness.

The adversary, in successfully predicting the entropy that the guest receives, can break many of the assumptions used by cryptographic protocols that require good randomness [4] [17].

5.3. Software and Hardware Setup

Our host machine is a 64-bit Linux server with dual Intel Xeon E5440 CPUs. The host runs the 2.6.32-30-generic kernel on Ubuntu Linux 10.04. The guest machine is using a modified version of the current (as of April 2011) long term stable kernel (as of April 2011), 2.6.35-12-generic on 64-bit Ubuntu Linux 10.04. The guest's kernel is modified so that `/drivers/char/random.c` prints debugging entries to the system log. The guest is run using one virtual CPU core.

6. Method

We created a known-input attack against Linux guests of a KVM host. The KVM model allows monitoring and control of interrupts and hard disk events, the two primary inputs into the entropy pool of a headless Linux system.

6.1. Interrupt and Event Monitoring

On a KVM host, each guest runs inside a user-space QEMU process [18], while control of the hypervisor is handled through `/dev/kvm`. When a guest would receive an interrupt, the hypervisor catches it, determines which guest it is for, and injects it into that guest through the QEMU process.

The QEMU application provides an event tracing system [19] which allows us to log each time a disk

request completes, an interrupt is injected, etc.. Further instrumentation of QEMU is possible— for example, modifying the QEMU executable to create new trace points. Individual traces already in QEMU can be enabled through the management console.

Many production environments of KVM use the paravirtualized virtio library [20] and its user-space block device driver for handling guest disks. As with interrupts, this resides in user-space, and is thus susceptible to malicious modification to log all hard disk events that would contribute to the guest's entropy pool. The built in trace event for `virtio_blk_req_complete` logs each time a disk request completes, before QEMU tells KVM to inject the interrupt for that event into the guest. Similarly, the `virtio_irq` trace event can be used to log interrupt injections into the guest.

After enabling these trace events, the adversary can use the logs of the interrupts and hard disk events injected into a particular guest to reconstruct the events that are used to seed that guest's entropy pool. This is effectively a known-input attack on the Linux PRNG of the guest. Given enough known-input into the guest's entropy pool, the adversary effectively decreases the amount of good randomness the guest receives, thus letting the adversary guess the randomness the guest receives faster than by brute-force.

6.2. Description of attack

We compiled a modified version of the 2.6.32-15 Linux kernel to log debug messages from `/driver/char/random.c` to the system log. These can log components of each input to the guest's PRNG: processor cycles, the time stamp counter (TSC), jiffies, and the event number.

This modified kernel was installed in the guest machine. Logging the times recorded by the guest on a disk interrupt, as well as the time recorded by the host on an interrupt, we calculated the difference to find the average time the guest's interrupts differed from the host's interrupt times.

We have some confidence in how much of the pool input we know, because we can log when all inputs to the guest's PRNG will occur. On a headless guest machine the only events that will contribute are interrupts and disk events.

Performing the SHA-1 hash on the entropy pool contents gives a random output, so knowing the entropy pool is enough to know the random output. We can now determine random numbers used in cryptographic applications, such as the random input used as initialization for session keys used in secure communication. Communication packets are also associated with times, so we will know when a random number was requested and what call or communication that corresponded to.

Knowing random numbers and communications, we can potentially take the known inputs, guess the entropy pool, perform offline calculations, and use previously recorded communications to crack session keys.

7. Results

The value of the Time Stamp Counter (TSC), the current jiffies value, and the event number are used to seed the entropy pool in a Linux system. Because we are attacking a headless system, the event number varies little or is constant. In fact, we observed only disk events being added to the pool, which caused the event number to remain constant.

Gathering debug entries from `/drivers/char/random.c` on the guest over the course of 19 hours, we observed that all 65,369 events were disk events, each with the same event number. This is due to the fact that all disk events were from the same logical disk and there are no contributing interrupts in the default kernel options. Over the 19 hour period, the guest received an average of 0.9518 events per second.

Over all of the 65,369 disk events, entropy credits were added to the input pool 15,087 times, averaging 7.38 entropy credits added per event. The maximum entropy added at once was 11 credits. Since each input is 128-bit, we can see that the LPRNG is conservative in its estimation of entropy when adding to its internal count.

We also observed 14,365 attempted reseeds of the non-blocking pool, and of those, 552 were successful at extracting entropy from the input pool and mixing it into the non-blocking pool. The average reseed, when successful, moved 201.2 entropy credits from the input pool to the non-blocking pool.

In measuring the TSC and the jiffies counts on both the host and the guest, we found a linear drift in both clocks. The jiffies value drifted at a rate of 1.209050 per second. The TSC value drifted at a rate of 8.844×10^6 per second.

We predicted the guest's TSC value and jiffies value by subtracting a previously measured difference between the guest's values and the host's. We found that we could consistently predict the jiffies value of the guest with an average error of 5 bits. We could also consistently predict the TSC value of the guest, with an average error of 16 bits.

Because the event number is constant, the average error of our predictions of the input to the guest's

PRNG was 21.1 bits, with a maximum error of 32 bits. Figure 1 shows the bit difference of our predictions versus the actual reported value in the guest.

Even though the host TSC and jiffies values drift considerably from those of the guest over time, we only measured a small positive correlation between the total prediction error and time. This means that an attacker could expect the drift of the host TSC and jiffies values from those of the guest to cause an increase error of the prediction the longer they try to monitor the guest.

In addition, if an attacker was able to characterize the drift of the clocks between the host and the guest (i.e., the rate of drift as previously measured), they could use it to better predict the difference between the host's clocks and those of the guest.

Using the linear model of host jiffies to guest jiffies (see Appendix A), we were able to reduce the average error in our predictions to 2 bits, an improvement by 3 bits. This effectively removed the positive correlation in the jiffies prediction error. However, using the linear model of host TSC to guest TSC did not improve the prediction error. This seems reasonable as our normal prediction error for the guest TSC already did not show any positive correlation.

Based on these results, we can see a reduction in the number of unknown bits within the entropy pools, but, after a few inputs, enough unknown bits will have been added that we cannot predict the output of the PRNG.

Also of note is that for guests that do not run SMP (Symmetric Multi-Processing) enabled kernels, calls to `get_cycles()` in the kernel (the function that reads the TSC on x86-based systems) always return zero. This means that an attacker could predict the input to the guest's entropy pool to an average error of 5 bits; a reduction in error by about 16 bits.

8. Future Work

The first extension of this attack would be to inject interrupts into the guest to affect the entropy pool, described further in section 8.1. Future work should also perform the same attacks against other virtual machines and test the affects of hedging, discussed in section 8.2. Lastly, we consider side channel attacks in section 8.3.

8.1. Interrupt Injection

Because the QEMU process that runs the guest on the host injects real interrupts into the guest, a malicious QEMU process could inject arbitrary (maliciously chosen) interrupts instead of, or in addition to, the real interrupts destined for that guest. This attack combines the known-input attack of logging interrupts injected into a guest with the chosen-input attack of injecting arbitrary interrupts.

8.2. Other VMs and Hedging

A natural extension to this project is to try to perform the same attack against other virtual machines including, but not limited to VirtualBox, VMware and Xen. The differences in implementation and licensing both influence how easy the attack would be; a large part of why we chose to investigate KVM was due to it being open source and the transparency that provided.

Defending against such an attack would also be worth investigation. This is a difficult problem, as it requires a method for the unprivileged, guest user to be able to determine that the host administrator has been tampering with the guest's random number source. If the host has simply been monitoring the input entropy, it becomes even harder for the user to detect. A solution (or part of one) could be determined while trying to perform the attack on additional virtual machines, as different implementations may prove to be stronger or weaker against this form of attack.

We would have liked to inspect and attack the hedged cryptography implemented on the OpenSSL library. Hedging as discussed in section 2.3 may render our attack moot, but it remains to be seen if we are also able to discover the arbitrary bit strings and operation performed.

8.3. Side Channel Attacks

Side channel attacks should also be explored to see if we can gather further information about the virtual machine and its activities. In the past, such attacks have been used to extract private keys stored on a local machine [21], as well as gather and share information between virtual machines executing on the same host [22].

We already make use of the virtual machine's process execution time to make an educated guess as to when the guest receives a given interrupt, though our current method is less effective when the process is moving across cores on a multicore host. The simplest improvement to the attack would be to improve how the timing estimates are taken, resulting in improved accuracy of the host's guess as to when the virtual machine receives a given interrupt.

Other side channel attacks may also be explored; depending on the privileges and access of the attacker, side channels like network traffic, cache timings and

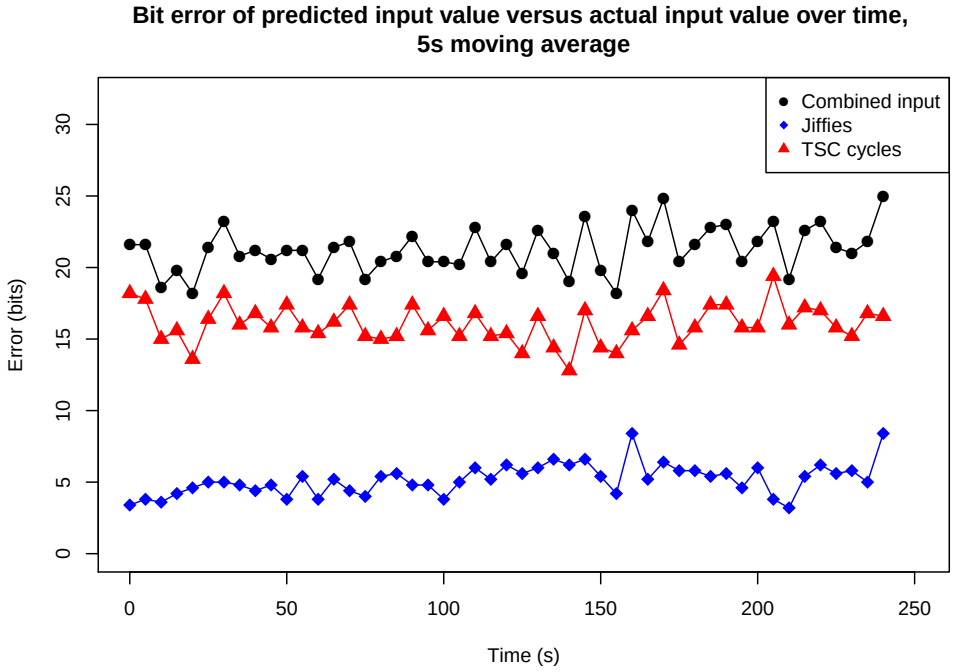

Figure 1: Our total prediction of the input to the guest's PRNG had an average error of 21.12351 bits. We saw a small but significant correlation between our prediction error and time—a Pearson's coefficient of 0.1925 with a p-value of 0.002189. We saw a stronger positive correlation in the prediction error for the guest's jiffies value with time (coefficient of 0.3026409 with p-value 1.030×10^6), and no correlation in the prediction error for the guest's TSC with time (coefficient of 0.03410603 with p-value 0.5907).

other process-related information could be used to make guesses as to when the VM is generating keys, or performing some other task that uses the random number generator.

9. Conclusion

In this paper we discussed an attack on the PRNG of a Linux system run within a virtual machine. We configured KVM to allow an administrator with limited host access to monitor when events on the host were added to the guest's entropy pool.

In doing so, we found that there was uncertainty in recording LPRNG inputs due to the inability of the host to measure when the guest checked the system timers and added the input to the LPRNG entropy pool. We empirically measured the timing differences by instrumenting a guest and making predictions based on the guest's process running time. Based on the error of these predictions and a theoretical analysis of the LPRNG, we found that the number of unknown bits in the input to the LPRNG could be reduced, especially in the case that the virtual machine process was allocated to a single core.

This demonstrates a reduction in the complexity of predicting pseudo-random numbers generated by the guest, but the reduction is not enough to achieve a cryptographic break on SMP systems due to the compounding nature of unknown input. For non-SMP systems, we observed a greater reduction in the complexity of predicting PRNG outputs, indicating that a cryptographic break may be possible. Further work is required to determine this conclusively.

References

[1] "Amazon web services," 2011. [Online]. Available: http://aws.amazon.com/

[2] T. W. Burger, "Intel® virtualization technology for directed I/O (VT-d): enhancing intel platforms for efficient virtualization of I/O devices - intel® software network," Feb. 2009. [Online]. Available: http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices/

[3] N. A. Quynh and Y. Takefuji, "Towards a tamper-resistant kernel rootkit detector," in *Proceedings of the*

2007 ACM symposium on Applied computing, 2007, p. 276–283.

[4] T. Ristenpart and S. Yilek, "When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography," in *Proceedings of the 2010 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2010. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/10/pdf/15.pdf

[5] "OpenSSL: the open source toolkit for SSL/TLS," 2011. [Online]. Available: http://www.openssl.org/

[6] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Cryptanalytic attacks on pseudorandom number generators," in *Fast Software Encryption*, 1998, p. 168–188.

[7] "OpenSSL: documents, rand(3)." [Online]. Available: http://www.openssl.org/docs/crypto/rand.html

[8] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator," in *Selected Areas in Cryptography*, 2000, p. 13–33.

[9] T. Ts'o and M. Mackall, "/dev/random.c," 2005, linux 2.6.35.12.

[10] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the linux random number generator," in *Security and Privacy, IEEE Symposium on*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 371–385.

[11] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: Security challenges in virtual machine based computing environments," in *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*. Santa Fe, NM: USENIX, 2005. [Online]. Available: http://www.usenix.org/events/hotos05/prelim_papers/garfinkel/garfinkel_html/

[12] M. T. Jones, "Linux virtualization and PCI passthrough," Oct. 2009. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-pci-passthrough/

[13] A. Rubini and J. Corbet, "Chapter 6: Flow of time," in *Linux Device Drivers*, 2nd ed. O'Reilly Press, 2001. [Online]. Available: http://www.xml.com/ldd/chapter/book/ch06.html

[14] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, p. 225–230.

[15] "Entropy broker." [Online]. Available: http://www.vanheusden.com/entropybroker/

[16] "Virtio-RNG." [Online]. Available: http://www.mnementh.co.uk/home/projects/collabora/virtio-rng

[17] S. Yilek, "Resettable Public-Key encryption: How to encrypt on a virtual machine," in *Topics in Cryptology - CT-RSA 2010*, ser. Lecture Notes in Computer Science, J. Pieprzyk, Ed. Springer Berlin / Heidelberg, 2010, vol. 5985, pp. 41–56, 10.1007/978-3-642-11925-5_4. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11925-5_4

[18] "QEMU." [Online]. Available: http://wiki.qemu.org/Index.html

[19] S. Hajnoczi, "User documentation: Features/Tracing - QEMU." [Online]. Available: http://wiki.qemu.org/Features/Tracing

[20] R. Russell, "virtio: towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Operating Systems Review*, vol. 42, p. 95–103, Jul. 2008, ACM ID: 1400108. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400108

[21] D. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," *Topics in CryptologyCT-RSA 2006*, p. 1–20, 2006.

[22] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, p. 199–212.

Appendix

Figures 2 and 3 show the behavior of the host and guest clocks over time. Given this data, we were able to additionally predict the offset between the host and the guest over time. As described in Section 7, using the linear model for the guest jiffies and cycles as a function of the host's clocks allowed us to reduce the error in our predicted jiffies value from 5 bits to 2 bits. However, the linear model did not reduce the error of our predicted cycles value.

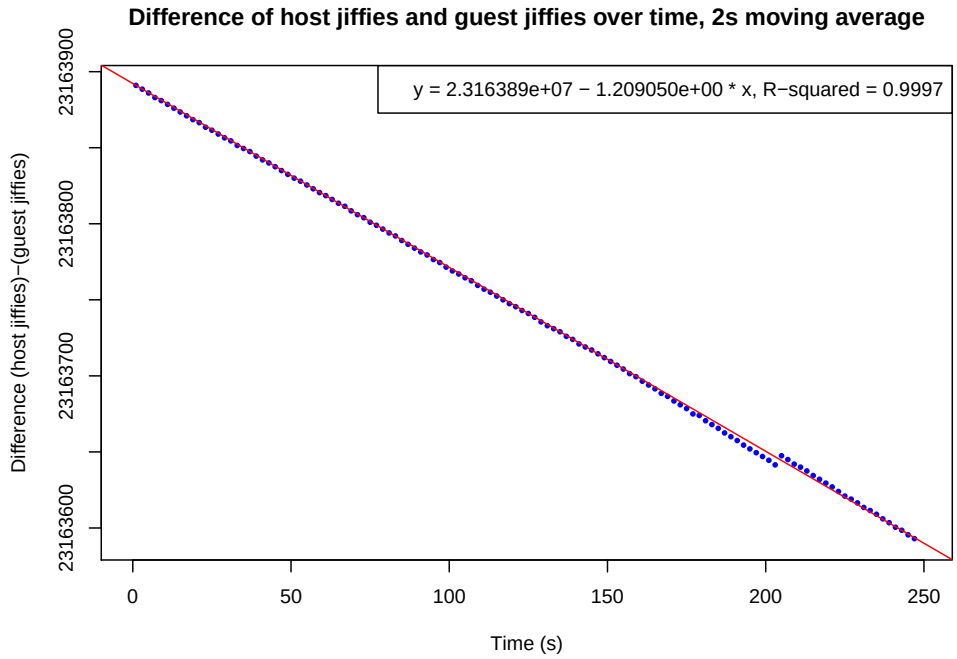

Figure 2: The difference, Δ, between the host's jiffies value and the guest's jiffies value over time can be strongly characterized by the linear model $\Delta = 2.316 \times 10^7 - 1.209 \cdot t$, with $R^2 = 0.9997$.

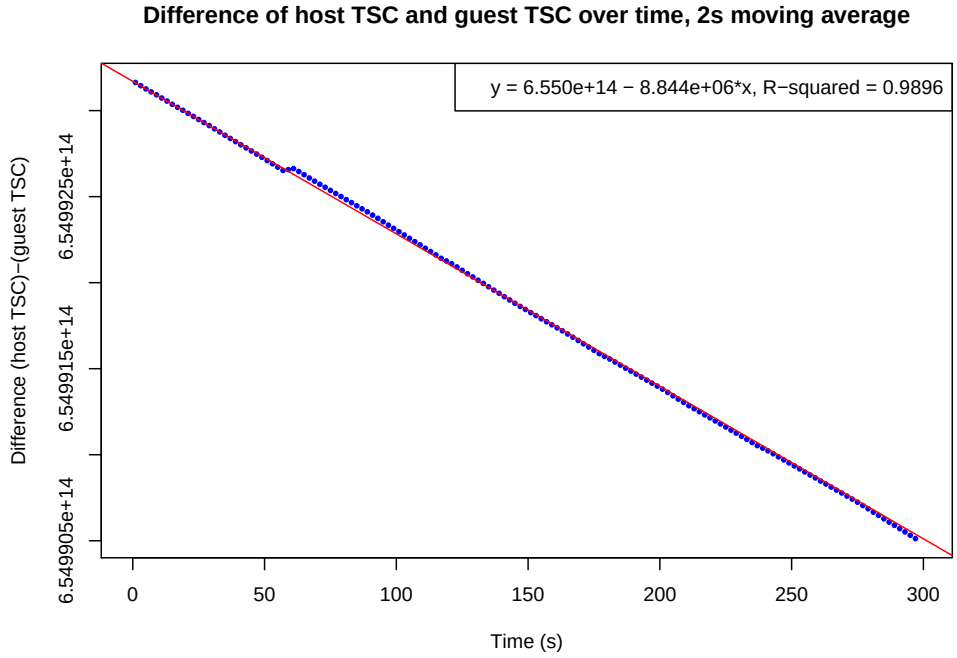

Figure 3: The difference, Δ, between the host's TSC cycle count value and that of the guest over time can be strongly characterized by the linear model $\Delta = 2.316 \times 10^7 - 1.209 \cdot t$, with $R^2 = 0.9997$.