Not-So-Random Numbers in Virtualized Linux and the Whirlwind RNG

Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, Michael Swift
Department of Computer Sciences
University of Wisconsin-Madison
{*ace, yanzhai, jellinek, rist, swift*}*@cs.wisc.edu*

Abstract—**Virtualized environments are widely thought to cause problems for software-based random number generators (RNGs), due to use of virtual machine (VM) snapshots as well as fewer and believed-to-be lower quality entropy sources. Despite this, we are unaware of any published analysis of the security of critical RNGs when running in VMs. We fill this gap, using measurements of Linux's RNG systems (without the aid of hardware RNGs, the most common use case today) on Xen, VMware, and Amazon EC2. Despite CPU cycle counters providing a significant source of entropy, various deficiencies in the design of the Linux RNG makes its first output vulnerable during VM boots and, more critically, makes it suffer from catastrophic reset vulnerabilities. We show cases in which the RNG will output the exact same sequence of bits each time it is resumed from the same snapshot. This can compromise, for example, cryptographic secrets generated after resumption. We explore legacy-compatible countermeasures, as well as a clean-slate solution. The latter is a new RNG called Whirlwind that provides a simpler, more-secure solution for providing system randomness.**

Keywords-**random number generator; virtualization**

I. INTRODUCTION

Linux and other operating systems provide random number generators (RNGs) that attempt to harvest entropy from various sources such as interrupt timings, keyboard and mouse events, and file system activity. From descriptions of events related to these sources, an RNG attempts to extract (by way of cryptographic hashing) bit strings that are indistinguishable from uniform for computationally bounded attackers. While recent system RNGs can make use of hardware RNGs such as Intel's rdrand instruction, security still relies on software sources either exclusively (e.g., on older CPUs) or in part (e.g., because of uncertainty about the efficacy of closed-source hardware RNGs [18]).

There exists significant folklore [14,15,26] that system RNGs such as Linux's may provide poor security in virtualized settings, which are increasing in importance due to adoption of cloud computing services such as Amazon's EC2. Stamos, Becherer, and Wilcox [27] hypothesized that the Linux RNG, when run within the Xen virtualization platform on EC2, outputs predictable values very late in the boot process. Garfinkel and Rosenblum [8] first hypothesized vulnerabilities arising from the reuse of random numbers when using virtual machine snapshots. Ristenpart and Yilek [25] were the first to show evidence of these and called them *reset vulnerabilities*. They demonstrated that

user-level cryptographic processes such as Apache TLS can suffer a catastrophic loss of security when run in a VM that is resumed multiple times from the same snapshot. Left as an open question in that work is whether reset vulnerabilities also affect system RNGs. Finally, common folklore states that software entropy sources are inherently worse on virtualized platforms due to frequent lack of keyboard and mouse, interrupt coalescing by VM managers, and more. Despite all this, to date there have been no published measurement studies evaluating the security of Linux (or another common system RNG) in modern virtualized environments.

Our first contribution is to fill this gap. We analyze a recent version of Linux and its two RNGs, the kernel-only RNG (used for stack canaries and address-space layout randomization) as well as the more well-known RNG underlying the /dev/urandom and /dev/random devices. Via careful instrumentation, we capture all inputs to these RNGs in a variety of virtualized settings, including on local Xen and VMware platforms as well as on Amazon EC2 instances. We then perform various analyses to estimate the security of the RNGs. Our work reveals that:

- Contrary to folklore, we estimate that software entropy sources, in particular (virtualized or non-virtualized) cycle counters provide significant uncertainty from an adversary's perspective during normal operation of the system (i.e., after it has booted).

- However, when booting a VM the first use of the kernel-only RNG as well as the first use of /dev/urandom are both vulnerable. There exists a boot-time entropy hole, where insufficient entropy has been collected before use of the RNGs. Later outputs of the RNG, however, appear intractable to predict, suggesting the concerns of Stamos et al. are unwarranted.

- Finally, the /dev/urandom RNG suffers from catastrophic snapshot reset vulnerabilities, which unfortunately answers the open question of [25] in the positive and obviates a countermeasure suggested for the user-level vulnerabilities previously discovered [25]. We show that resets can lead to exposure of secret keys generated after snapshot resumption.

Our results are qualitatively the same across the different VM management environments, though note that EC2 does not currently support snapshots and therefore does not (yet)

OS	RNG	Reset Security	Boot Security	Tracking Security
Linux	GRI	No	No	No
	/dev/(u)random	No	No	Yes
FreeBSD	/dev/random	No	?	Yes
Windows	rand_s	No	?	?
	CryptGenRandom	No	?	?
	RngCryptoServicesProvider	No	?	?
Linux	Whirlwind	Yes	Yes	Yes

Figure 1. Security comparison of system RNGs. A question mark (?) indicates ""Unknown". Reset security refers to safety upon VM snapshot resumption, boot security means sufficient entropy is generated prior to first use, and tracking security is forward- and backward-security in the face of compromise. See Section II-C for more details.

suffer from reset vulnerabilities.

We also perform limited experiments with FreeBSD and Windows, and specifically demonstrate that reset vulnerabilities affect FreeBSD's /dev/random and Microsoft Windows rand_s as well. This suggests that problems with virtualized deployments are not confined to the Linux RNGs.

We move on to offer a new RNG design and implementation (for Linux), called Whirlwind. It directly addresses the newly uncovered deficiencies, as well as other long-known problems with the Linux RNG. Here we are motivated by, and build off of, a long line of prior work: pointing out the troubling complexity of the /dev/random and /dev/urandom RNG system [6,11,17]; showing theoretical weaknesses in the entropy accumulation process [6]; designing multi-pool RNGs without explicit entropy counters [13,20]; and showcasing the utility of instruction and operation timing to quickly build entropy [1,22,23].

Whirlwind combines a number of previously suggested techniques in a new way, along with several new techniques. It serves as a drop-in replacement for both of the Linux RNGs, and provides better security (see Figure 1). In addition to security, the design focuses on simplicity, performance, theoretical soundness, and virtualization safety (though it will perform well for native settings as well). At its core is a new cryptographic hashing mode, inspired by but different from the recent construction of Dodis et al. [6], plus: a simple two-pool system, simpler interface, stream-lined mostly-CPU-lock-free entropy addition, a method for bootstrapping entropy during boot and VM resumption, direct compatibility with hypervisor-provided randomness, and support for the rdrand instruction when it is available. We emphasize that the security of Whirlwind never relies on any one feature in particular (e.g., using rdrand by itself), and instead uses multiple inputs sources to ensure the highest possible uncertainty even in the face of some entropy sources being compromised.

In terms of performance, Whirlwind matches the current Linux /dev/urandom, and in some cases performs better. We also show experimentally that it suffers from none of the problems for virtualized settings that render the current Linux RNG vulnerable. We do caution that more analysis will be needed before widespread deployment, since the Linux RNGs must work in diverse environments. For

example, future analysis will include low-end embedded systems, another problematic setting [11,12,22]. Towards this, we are in the process of making Whirlwind ready for public, open-source release.

Finally, we explore hypervisor-based countermeasures for legacy guest VMs with the old RNG. In particular, we investigate whether the hypervisor can defend against reset vulnerabilities by injecting entropy into the guest RNG via (artificially generated) interrupts during resumption. We show that the host OS can force Linux /dev/random to refresh itself, but that it takes at least a few seconds and requires a large number of interrupts. Such limitations suggest that legacy-compatible approaches are unsatisfying in the long term, and we instead suggest moving to a new RNG such as Whirlwind.

II. Background

A. The Linux RNGs

The Linux kernel provides three RNG interfaces which are designed to provide cryptographically strong random values: /dev/random, /dev/urandom, and get_random_int (GRI).

The /dev/(u)random RNG. The Linux kernel exposes two pseudo-devices that implement interfaces to what we call the /dev/(u)random RNG. The first, /dev/random, may block until enough entropy is available, while the second, /dev/urandom, is non-blocking. On the systems we examined, applications and the Linux operating system itself use exclusively /dev/urandom and never read from /dev/random. The RNG consists of (1) entropy gathering mechanisms that produce *descriptions* of system events; (2) several entropy *pools* to which these descriptions are mixed with a cryptographically weak generalized feedback shift register; (3) logic for how and when entropy flows between pools (described below); and (4) APIs for consumers to query to obtain randomness. To retrieve random numbers, an application opens one of the device files, performs a read, and (presumed-to-be) random bytes are returned. Additionally, an application may write to either device, in which case the /dev/(u)random RNG mixes the contents of the write buffer into both secondary entropy pools (also described below) but does not update any entropy estimates. For example, during boot a file containing output from /dev/urandom during the preceding shutdown is
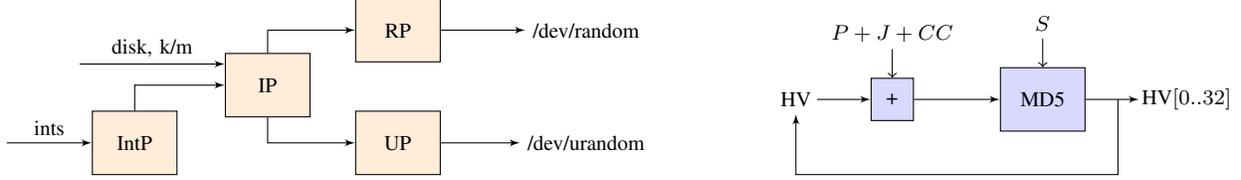
Figure 2. The Linux RNGs. **(Left)** Data flow through the /dev/(u)random RNG and **(Right)** the kernel-only RNG GRI.

written back into /dev/(u)random. Read and write requests are always made in units of bytes. The /dev/urandom RNG also has a kernel-only interface get_random_bytes() that does not use the pseudo-device but is functionally identical to /dev/urandom.

An entropy pool is a fixed-size buffer of random data stored in kernel memory along with associated state variables. These variables include the current mixing location for new inputs and an entropy count measured in bits. There are four pools as shown on Figure 2. In the below, we omit details regarding the non-cryptographic mixing functions. Detailed descriptions appear in [6,17].

Interrupt pool (IntP): The kernel IRQ handler adds a description of each interrupt to a 128-bit interrupt pool (called a "fast pool" in the source code). There is one IntP per CPU to eliminate contention. Each interrupt delivery takes a description (cycle counter xor'd with kernel timer, IRQ number, and the instruction pointer at the time the interrupt is received) and mixes it into the pool using a cryptographically weak function. The entire contents of each IntP are mixed into the input pool IP using another (more complex generalized feedback register) mixing function every 64 interrupts or if a second has passed since the last mixing into IP. At the same time, the input pool entropy count denoted IP.ec is incremented (credited) by one (bit), which represents a conservative estimate .

Input pool (IP): The 4096-bit input pool has the interrupt pool mixed into it as just mentioned, and as well has device-specific event descriptions (kernel timer value, cycle counter, device-specific information) of keyboard, mouse, and disk events mixed in using the cryptographically weak mixing function. We will only consider settings with no keyboard or mouse (e.g., servers), and so only disk events are relevant. (Network interrupts go to IntP.)

Non-blocking pool (UP): A 1024-bit pool is used for the non-blocking /dev/urandom interface. Upon a request for $8n$ bits of randomness, let $\alpha_u = \min(\min(\max(n, 8), 128), \lfloor IP.ec/8 \rfloor - 16)$. If UP.ec $< 8n$ and $8 \leq \alpha_u$ the RNG transfers data from the input pool IP to UP. Put another way, a transfer occurs only if UP.ec $< 8n$ and IP.ec ≥ 192. If a transfer is needed, the RNG extracts α_u bytes from IP and mixing the result into UP, decrementing IP.ec by $8\alpha_u$, and incrementing UP.ec by $8\alpha_u$. If a transfer is not needed or not possible (by the restrictions above), then UP is left alone. In the end, the RNG decrements UP.ec by $8n$, extracts $8n$ bits from UP, and return those bits to the calling process.

Blocking pool (RP): A 1024-bit pool is used for the blocking /dev/random interface. Upon a request for $8n$ bits of randomness, let $\alpha_r = \min(\min(\max(n, 8), 128), \lfloor IP.ec/8 \rfloor)$. If RP.ec $\geq 8n$ then it immediately extracts $8n$ bits from RP, decrements RP.ec by $8n$, and returns the extracted bits. Otherwise it checks if $\alpha_r \geq 8$ and, if so, transfers α_r bytes from IP to RP, incrementing and decrementing entropy counters appropriately by $8\alpha_r$. It then immediately extracts $\lfloor RP.ec/8 \rfloor$ bytes from RP, decrements RP.ec appropriately, and returns the extracted bits to the calling process. If on the other hand $\alpha_r < 8$, then it blocks until $\alpha_r \geq 8$.

Figure 3 summarizes the conditions required for transferring data from one pool to the next. The design of /dev/(u)random intimately relies on ad-hoc entropy estimates, which may be poor. We will also see, looking ahead, that the entropy counters cause trouble due to their use in deciding when to add entropy to the secondary pools. For example, we observe that there exists a simple *entropy starvation* attack against /dev/urandom by a malicious user process that continuously consumes from /dev/random (e.g., using the command dd if=/dev/random). In this case, reads from /dev/urandom will never trigger a transfer from IP.

Output extraction: We give an overview of the cryptographic extraction routine for IP, UP, and RP. To begin, the contents of the pool are first hashed with SHA-1 and then mixed into the same pool using the non-cryptographic mixing function. Then 64 bytes of the pool are hashed with a modified SHA-1 algorithm that uses the first 5 bytes of the previous hash as the IV. This 20-byte digest d is then reduced to a 10-byte output block $d' = d[0..3] \oplus d[12..15] \mathbin{||} d[4..7] \oplus d[16..19] \mathbin{||} d[8..9] \oplus d[10..11]$ where $||$ is concatenation. This entire routine is repeated until the ouput is at least as long as the requested number of bytes. The final output block is truncated as needed. The entropy count for the pool is decremented by the number of output bits that were generated (excluding any truncated bits). When extracting from IP, the output value is mixed into either UP or RP (whichever generated the request). When extracting from UP or RP the output value becomes the output of /dev/urandom or /dev/random respectively.

3

Transfer	When	Condition
IntP → IP	Interrupt arrival	64 interrupts or 1 second
IP → UP	n bytes requested from /dev/urandom	UP.ec $< 8n$ IP.ec ≥ 192
IP → RP	n bytes requested from /dev/random	RP.ec $\leq 8n$ IP.ec ≥ 64

Figure 3. Conditions for transfers between entropy pools.

get_random_int: the kernel-only RNG. GRI is a simple RNG that provides 32-bit values exclusively to callers inside the kernel. GRI is primarily used for Address Space Layout Randomization (ASLR) and StackProtector "canary" values used to thwart stack-smashing attacks. The GRI RNG is designed to be very fast and does not consume entropy from the pools in /dev/(u)random.

The GRI RNG uses two values stored in kernel memory: a per-CPU 512-bit hash value HV and a global 512-bit secret value S, which is initially set to all zeros. During the `late_init` phase of boot, the kernel sets the secret value S to 512-bits obtained from /dev/urandom.

Each time it is called, GRI adds the process ID (PID) P of the current process, the current kernel timer value J (called *jiffies*), and the lower-32 bits of the timestamp cycle counter CC into the first 32-bit word of the hash value HV, and then sets HV to the MD5 hash of HV and the secret value S. That is, it computes $HV = H(\ (HV[1..32] + P + J + CC)\ ||\ HV[33..512]\ ||\ S)$ where "+" is integer addition modulo 2^{32}, and $H(\cdot)$ is the MD5 hash. The first 32 bits of HV are returned to the caller, and the new HV value becomes the stored hash value for the next call.

Use of hardware RNGs. If available, the /dev/(u)random RNG uses architecture-specific hardware RNGs during initialization and output generation. During boot, /dev/(u)random reads enough bytes from the hardware RNG to fill each pool and uses the weak mixing function to mix in these values. This is done for the input, nonblocking, and blocking pools, but not for the interrupt pool. During output generation, /dev/(u)random XORs 10-bytes from the hardward RNG into each 10-byte block of output that is produced. GRI returns a 32-bit value from the hardware RNG in place of the software implementation described above.

B. Virtualization

In this work, we focus on the efficacy of the Linux RNGs when operating in virtualized environments without the aid of a hardware RNG. In a virtualized environment, one or more guest virtual machines (VMs) run on a single physical host, and the hypervisor mediates access to some hardware components (e.g., the network interface, disks, etc.). There is a management component for starting, stopping, and configuring virtual machines. In Xen, this is called Dom0,

while in hosted virtual machines (e.g., VMware Workstation) this is the host operating system.

A VM can be started in one of three ways. First, it can execute like a normal physical system by booting from a virtual disk. As it executes, it can update the state on the disk, and its next boot reflects those changes. Second, a VM can be repeatedly executed from a fixed *image*, which is a file that contains the persistent state of the guest OS. In this case, changes made to the OS state are discarded when the VM shuts down, so the OS always boots from the same state. This is the default case, for example, in infrastructure-as-a-service cloud computing systems including Amazon EC2. Third, a VM can start from a *snapshot*, which is a file that contains the entire state of a running VM at some point in its execution. This includes not only the file system but also memory contents and CPU registers. Both Xen and VMware support pausing a running VM at an arbitrary point in its execution and generating a snapshot. The VM can be resumed from that snapshot, which means it will continue executing at the next instruction after being paused. If a VM continues running after the snapshot, restarting from a snapshot effectively rolls back execution to the time when the snapshot was taken.

It has long been the subject of folklore that RNGs, and in particular, /dev/(u)random, may not perform as well when run within a VM [14,15,26,27]. First, hypervisors often *coalesce* interrupts into batches before forwarding them to a given guest domain to improve performance. Second, memory pages are typically zeroed (set to all zeroes to erase any "dirty" data) by the hypervisor when new physical memory pages are allocated to a guest VM. Zeroing memory pages is required to ensure that dirty memory does not leak information between different guests on the same host machine. Third, several system events used for entropy by /dev/(u)random are not relevant in popular uses of virtualization, in particular keyboard and mouse events do not occur in virtualized servers.

C. RNG Threat Models

The Linux RNGs are used by a variety of security-critical applications, including cryptographic algorithms and for system security mechanisms. Should RNG values be predictable to an adversary or the same (unknown) value repeatedly used, the RNG-using applications become vulnerable to attack. As just a few examples, /dev/urandom is used to seed initial TCP/IP sequence numbers and by cryptographic libraries such as OpenSSL to generate secret keys, while GRI is used as mentioned above for ASLR and stack canaries.

RNGs are therefore designed to face a variety of threats from attackers both off-system and (unprivileged) local attackers. We assume that the attacker always knows the software and hardware stack in use (i.e., kernel versions,

distribution, and underlying hypervisor). The threats to RNG systems are:

(1) *State predictability*: Should the entropy sources used by the RNG not be sufficiently unpredictable from the point of view of the attacker, then the RNG state (and so its output) may be predictable. For example, a low-granularity time stamp (e.g., seconds since the epoch) is a bad entropy source because it is easily guessed [10].

(2) *State compromise*: The attacker gets access to the internal state of the RNG at some point in time and uses it to learn future states or prior states (forward-tracking and back-tracking attacks respectively). Forward-tracking attacks may use RNG outputs somehow obtained by the attacker as *checkpoints*, which can help narrow a search by allowing the attacker to check if guessed internal states of the RNG are correct. VM snapshots available to an attacker, for example, represent a state compromise.

(3) *State reuse*: With full-memory VM snapshots, the same RNG state may be reused multiple times and produce identical RNG outputs. Since the security of a random number is its unpredictability, this can eliminate the security of the operation using a repeated RNG output.

(4) *Denial-of-service*: One process attempts to block another process from using the RNG properly.

Our focus will be on the design of the RNGs, and so we will not attempt to exploit cryptanalytic weaknesses in the underlying cryptographic primitives MD5 and SHA-1.

III. MEASUREMENT STUDY OVERVIEW

In the following sections we report on measurements in order to answer several questions about the security of the Linux RNGs when used on virtual platforms. In particular:

- When booting from a VM image, how quickly is the RNG state rendered unpredictable? (Section IV)

- Does VM snapshot reuse lead to reset vulnerabilities? (Section V)

Along the way we build a methodology for estimating uncertainty about the RNG state, and, as a result, assessing the suitability of various sources of entropy. Of course, one cannot hope to fully characterize software entropy sources in complex, modern systems, and instead we will use empirical estimates as also done by prior RNG analyses [9,22]. When estimating complexity of attacking an RNG, we will be conservative whenever possible (letting the adversary know more than realism would dictate). Where vulnerabilities appear to arise, however, we will evidence the issues with real attacks.

To accomplish this, we perform detailed measurements of the Linux RNGs when rebooting a virtual machine and when resuming from a snapshot. We produced an instrumented version of the Linux kernel v3.2.35, which we refer to as the *instrumented kernel*. The instrumentation records all inputs submitted to the RNGs, all calls made to the RNGs to produce outputs, changes to the entropy counts for each of /dev/(u)random's pools, and any transfers of bits between entropy pools. To avoid significant overheads, the resulting logs are stored in a static buffer in memory, and are written to disk at the end of an experiment. Our changes are restricted to the file: /drivers/char/random.c.

There were surprisingly non-trivial engineering challenges in instrumenting the RNGs, as the breadth of entropy sources, inherent non-determinism (e.g., event races), and the potential for instrumentation to modify timing (recall that time stamps are used as entropy sources) make instrumentation delicate. For brevity we omit the details. However, we did validate the correctness of our instrumentation by building a user-level simulator of the RNGs. It accepts as input log files as produced by the instrumented kernel, and uses these to step through the evolution of the state of the RNGs. This allowed us to verify that we had correctly accounted for all sources of non-determinism in the RNG system, and, looking ahead, we use this simulator as a tool for mounting attacks against the RNGs. For any computationally tractable attacks, we also verify their efficacy in an unmodified Linux kernel.

We will publicly release open-source versions of the instrumented kernel as well as simulator so others can reproduce our results and/or perform their own analyses in other settings. Links to open-source code for this project can be found on the author's website.

We use the following experimental platforms. For local experiments, we use a 4-core Intel Xeon E5430 2.67 GHz CPU (64-bit ISA) with 13 GB of main memory. We use Ubuntu Linux v12.10 in the *Native setup*, and we use the same OS for host and guest VMs. The *Xen setup* uses Xen v4.2.1, and the single Xen guest (domU) is configured with a single CPU and 1 GB of main memory. The cycle counter is not virtualized on Xen experiments (the default setting). The *VMware setup* uses VMware Workstation 9.0.0 with guest given a single CPU and 2 GB of main memory. On VMware the cycle counter *is* virtualized (the default).

Although we performed experiments with Ubuntu, our results should apply when other Linux distributions are used in either the host and/or guest. Finally in our *EC2 setup*, we built an Amazon Machine Image (AMI) with our instrumented kernel running on Ubuntu Linux v12.04 (64-bit ISA). All experiments launched the same AMI on a fresh EBS-backed m1.small instance in the US East region. In our experimental setups, there exist no keyboard or mouse inputs, which is consistent with VM deployments in data centers.

IV. BOOT-TIME RNG SECURITY

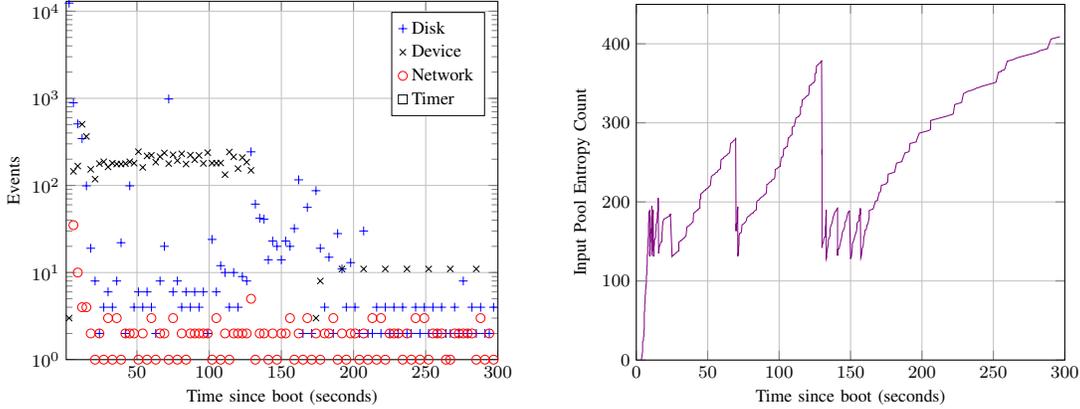We examine the behavior of the two Linux RNGs (GRI and /dev/(u)random) during boot, in particular seeking to

Figure 4. **(Left)** The number of event descriptions input to /dev/(u)random RNG by type of system event for the first 5 minutes of boot captured on VMware. The y-axis contains number of events (logscale) that occurred during each 3-second bin. **(Right)** The value of the input pool's entropy counter IP.ec during boot.

understand the extent to which there exist boot-time entropy holes (insufficient entropy collection before the first uses of the RNGs). As mentioned, in the past concerns have been raised that the Linux RNGs, when running on Amazon EC2, are so entropy starved that cryptographic key generation towards the end of boot could be compromised [27]. Our results refute this, showing that uncertainty in the RNGs is collected rather rapidly during boot across a variety of settings. We do, however, expose a boot-time entropy hole for the very first uses of both GRI and /dev/(u)random. In both cases the result is that stack canaries generated early in the boot process do not provide the targeted uncertainty: they are limited to 27 bits of uncertainty (instead of 64 bits) due to weak RNG outputs.

We perform analyses using the instrumented kernel in the Native, Xen, VMware, and Amazon EC2 setups (described in Section III). We perform 200 boots in each environment, and analyze the resulting log files to assess the security of the RNGs. After boot, the VM is left idle. We break down our discussion by RNG, starting with /dev/(u)random.

A. /dev/(u)random boot-time analysis

The left graph in Figure 4 displays the quantity and types of inputs to the RNG for a single boot in the VMware setup (the other VMware traces are similar). The x-axis is divided into 100 equal-sized buckets (3 seconds each) and the y-axis represents the number of occurrences of each input to the RNG state observed during a single time bucket (on a logarithmic scale). The majority of RNG inputs during boot are from disk events and other device interrupts while timer events are rare. The other platforms (Native, Xen, and EC2) were qualitatively similar.

The right chart in Figure 4 plots the entropy counter IP.ec for the input pool over time. This is the RNG's estimate of how much entropy that pool has gathered. Sharp drops indicate a transfer of data from the input pool to one of the secondary pools. We observe that the blocking /dev/random interface is never used in these systems, all calls are to /dev/(u)random. The estimates of entropy by the RNG may or may not be accurate, and so we seek to use the data from these runs to assess the actual unpredictability of the RNG state (and, hence, its outputs) from an attacker's point of view.

Estimating attack complexity. In order to estimate the security of /dev/(u)random *outputs*, we seek a lower bound on the complexity of predicting the state of the RNG by examining its *inputs*. Given that we target only lower bounds, we are conservative and assume the attacker has a significant amount of information about inputs and outputs to the RNG. When these conservative estimates show a possible vulnerability, we check for attacks by a more realistic attacker.

To establish a lower bound, we define the following conservative attack model. The attacker is assumed to know the initial state of the RNG (this is trivially true when booting VM images, due to zeroed memory) and the absolute cycle counter at boot time (the exact value is not typically known). To estimate the security of output i of /dev/(u)random, we assume the attacker has access to *all* preceding RNG outputs and the exact cycle counter for each output generation, including the ith output. This means we are assessing a kind of checkpointing or tracking attack in which the attacker can utilize knowledge of previous RNG outputs generated by typical requests to reduce her search space.

We will additionally assume that the exact sequence of RNG input types and the values of all event descriptions except the cycle counter are known to the attacker. This makes the cycle counter the only source of unpredictability for the attacker. The reason we do this is that, in fact, the other inputs included in event descriptions such as IRQ appear to provide relatively little entropy. (Of course a real

attacker would need to predict these values as well, but again we will be generous to the attacker.)

Input and output events. For a given platform (Xen, VMware, EC2, or native) we analyze traces of inputs and outputs of /dev/(u)random collected using our instrumented kernel. We call a single boot of the instrumented kernel a *trial*. Each trial produces a *trace*, a sequence of recorded input and output events of the form $\mathsf{Tr}_k = (y_1, ctr_1, CC_1), (y_2, ctr_2, CC_2), (y_3, ctr_3, CC_3), \ldots$ where $0 \leq k < 200$ is the index for a given trial, $y \in \{\mathsf{keyboard}, \mathsf{mouse}, \mathsf{disk}, \mathsf{IRQ}\text{-}0, \ldots, \mathsf{IRQ}\text{-}n, \mathsf{output}\}$ is the type of event recorded, ctr is the counter for events of this type that appear in trace Tr_k, and CC is the 64-bit cycle counter at the time the event is recorded. The output type represents any output from /dev/(u)random and the remaining values indicate different sources of input events. In the context of our analysis, we often use the terms trial and trace interchangeably.

Across all 200 traces, we group input events with matching type y and counter ctr into a vector $\vec{p}$. For example, for $y = \mathsf{IRQ}\text{-}16$ and $ctr = 20$, we take the cycle counter from the 20th occurrence of an interrupt on IRQ 16 from each trace and produce a vector $\vec{p}$. Let $\vec{p}[k]$ designate the kth entry in this vector, and so $\vec{p}[k]$ is the cycle counter associated with an input from trial k.

Similarly, we identify /dev/(u)random output events by their position in the list of output events in a given trace. To analyze the security of output i in trial k we need to determine which input events occurred after the previous output, $i - 1$, but before output i. We define this set of events as $S_i = \{\vec{p} \mid CC_{i-1} < \vec{p}[k] < CC_i\}$ where CC_i is the cycle counter associated with output i in trial k. Let $\ell_i = |S_i|$ be the length of S_i in a given trial.

Grouping input events into these sets is critical to the analysis: an attacker must correctly predict all inputs in S_i in order to to guess the internal state of the RNG when output i is generated. The complexity of this attack then grows exponentially with the length of S_i assuming each input presents some uncertainty to the attacker.

We define $\alpha \geq 0$ as the number of lower bits of a group of input cycle counters $\vec{p}$ that appear to be uniformly distributed. For any $\vec{p}$, it is likely the case that some number of upper bits are biased (not uniformly distributed) but still provide some amount of uncertainty to the adversary. For simplicity we ignore these and focus only on the lowest α bits for any $\vec{p}$.

Statistical test for uniformity. To determine how many low bits appear to provide uncertainty, we use the Kolmogorov-Smirnov (KS) 1-sample test [28]. The KS test examines the maximum difference between the cumulative distribution function of a set of samples compared to a reference distribution (in our case, the uniform distribution). When the maximum difference is above some threshold for

---

Algorithm 1 findAlpha($\vec{p}$)

$max_\alpha \leftarrow 0$
for $\alpha = 1$ **to** 64 **do**
    if ksUniformTest($\vec{p}, \alpha$) **then**
        $max_\alpha \leftarrow \max(max_\alpha, \alpha)$
    end if
end for
return max_α

Computes the maximum number of lower bits α that pass the KS test for uniformity. The input $\vec{p}$ is a vector of 64-bit cycle counter values.

Algorithm 2 minAlpha($\vec{p}, E$)

$min_\alpha \leftarrow$ uniformAlpha($\vec{p}$)
for $\vec{e} \in E$ **do**
    $\vec{\delta} \leftarrow \vec{p} - \vec{e}$
    $\alpha \leftarrow$ uniformAlpha($\vec{\delta}$)
    $min_\alpha \leftarrow \min(min_\alpha, \alpha)$
end for
return min_α

Computes the minimum number of lower bits α that appear uniformly distributed for any given input event considering all possible offsets with previous input and output events. Here E is the set of input and output vectors that strictly precede $\vec{p}$.

---

a given significance level, the KS test rejects this set of samples. For a candidate value α and a given $\vec{p}$, we begin by masking the upper $(64 - \alpha)$ bits of each cycle counter in $\vec{p}$. We then compare this set of t values to the uniform distribution over $[0, 2^\alpha - 1]$ using the KS test. We find the largest α that passes the KS test. See Algorithm 1.

Typical significance levels for the KS test are 0.1, 0.05, 0.025, and 0.001 [28]; we chose 0.1 which is most conservative (it favors smaller values for α). For this use of the KS test, the significance level can be thought of as the false-negative rate of the test; that is, how likely is the KS test to reject some collection of values that are randomly sampled from a uniform distribution. So our choice of a significance level of 0.1 indicates that there is a 10% chance that a given α value is rejected even if the lower α bits are sampled from a uniform distribution. This is the reason that a significance level of 0.1 produces the most conservative results compared to smaller values.

Any given input may be highly correlated with some previous input or output event and an attacker can use this correlation to her advantage when guessing cycle counters. To account for this, we also apply the above tests to relative cycle counter values. That is, for any vector of input cycle counters $\vec{p}$, we collect the set E of all vectors of cycle counters for input and output events that strictly precede $\vec{p}$. For any $\vec{e} \in E$, we compute the relative cycle counters between each component: $\vec{\delta} = \vec{p} - \vec{e}$, where $-$ is component-wise (vector) subtraction. Then we compute the maximum α that passes the KS test for uniformity among the relative cycle counter values in $\vec{\delta}$. We repeat this for every vector in E and keep only the minimum α value computed. This is shown in Algorithm 2.

Note that Algorithm 1 considers all candidates $0 \leq \alpha \leq 64$. In limited trials, we always observed $\alpha \leq 24$, so for efficiency of the analysis, we use this limit on the maximum α tested.

We also experimented with using a χ^2 test in place of the KS test in the procedures just described. The results were nearly identical for the same significance levels, with the KS test being slightly more conservative (it chose smaller α values). We therefore use the KS test and only report on it.

Computing complexity. The number of inputs preceding any given output varies from trial to trial on the same platform due to slight timing differences during the boot sequence. Recall that ℓ_i is the number of inputs that precede output i in a given trial. This value is a key component to the complexity of predicting the RNG's state since each input increases the attacker's search space by a multiplicative factor (assuming each input carries some uncertainty). So we compute the complexity of predicting a given set of inputs individually for each trial.

We compute the complexity of predicting the cycle counters in the input set S_i for trial k as: $s_i = \sum_{\vec{p} \in S_i} \mathsf{minAlpha}(\vec{p})$. This is the same as computing the logarithm of the size of the search tree for cycle counters when the attacker must only predict the lower $\alpha_{\vec{p}}$ bits for each $\vec{p}[k] \in S_i$ and these bits are each chosen independently and uniformly at random.

To determine the complexity of predicting output i in trial k we compute $\kappa_i = \max\{s_1, s_2, \ldots, s_i\}$. We use maximum instead of a sum for simplicity, since in general the cumulative complexity is dominated by the most complex input set. Again, we compute κ_i individually for each trial k and then examine the minimum value across all trials.

To summarize, 2^{κ_i} represents a lower bound on an adversary's ability to predict the ith output of the RNG during boot assuming that the low $\alpha_{\vec{p}}$ bits of the cycle counter for each input $\vec{p}$ are set uniformly at random. Unless specified otherwise, this will be the standard method for computing lower-bounds on attack complexity, and although it is a heuristic, we believe it to be a good one.

Figure 5 shows the complexities for the platforms we tested during the first few seconds of boot. These values were computed using $t = 200$ boots on each platform using our instrumented kernel. In all cases the first output is vulnerable; see discussion below. Beyond that, our analysis shows that the lower-bounds on attacks increase very rapidly, with Xen and the native platform exhibiting the smallest complexity for the second output, an attack complexity of at least 2^{129}. The native platform reaches a $\min \kappa_i = 1024$ (the minimum κ_i taken over all trials) at output $i = 140$ which is generated 4.0 seconds after boot (not shown). After 5 seconds on all platforms the attack complexity reaches the maximal value for this RNG: 1024 bits. Note that the times of outputs reported in this table are relative to the time the

Linux kernel is initialized, which does not include the time in the VM manager's startup and guest boot loader (e.g., this is about 3.5 seconds in VMware).

We observe that very large input sets dominate the cumulative attack complexity κ_i, which is not surprising. In all trials on all platforms, we observed $\max \ell_i \geq 395$ in the first 5 seconds after boot, that is, all boots have at least one set of 395 or more inputs. This means that each input cycle counter needs to carry only 2.6 bits of uncertainty on average for κ_i to reach its maximum value of 1024. On a platform with a 1.8 GHz clock (the slowest platform we tested, EC2), this represents an average jitter for input events of 2.9 ns.

Note that this analysis assumes that the cycle counters of input events are not under the control of an attacker and that cycle counter values are not leaked to an attacker through a side channel. Although such attacks may be possible, they require an attacker to control or influence nearly all inputs to the RNG or gain knowledge of nearly all bits of each of the tens of thousands of inputs that occur during boot.

First output entropy hole. Note that Figure 5 shows that the complexity of predicting the first output is zero. The first output of /dev/(u)random *always* occurs before any inputs are added to the RNG. Because the VM is supplied with zeroed memory pages, the Linux RNGs always start in a predictable state and so this output is deterministic. We observe this behavior on both VMware and Xen. The first output, always the 64-bit hexadecimal value `0x22DAE2A8 862AAA4E`, is used by boot_init_stack_protector. The current cycle counter is added to this RNG output (the canary equals $CC + (CC \ll 32)$ where CC is cycle counter) to initialize the stack canary of the init process. Fortunately, the cycle counter adds some unpredictability, but our analysis of the first GRI output (see Section IV-C) indicates that cycle counters early in boot carry about 27 bits of uncertainty, which is significantly weaker than the ideal security of 64-bits for a uniformly random stack canary.

B. Minimum Entropy Analysis

We also perform analysis similar to the method used in [22] where the authors examine inputs to /dev/(u)random during boot on embedded platforms. In that method, the authors compute Pearson correlation coefficients ($-1 \leq \rho \leq 1$) between all pairs of inputs, exclude any input with $|\rho| > 0.4$, and then compute a lower bound estimate of the security of /dev/(u)random by summing the empirical minimum entropy (H_∞) for each input during boot. The authors ignore security of individual outputs and instead produce a final lower bound for the RNG once boot has completed.

We perform a similar analysis on our datasets. We group our inputs according to input type and construct input sets as we do the in the KS analysis. We then mask the cycle counters using the minimum α value computed using the KS method and compute Pearson correlation coefficient ρ

	Native				Xen				VMware				EC2				VMware GRI			
i	T_i	ℓ_i	s_i	κ_i	T_i	ℓ_i	s_i	κ_i	T_i	ℓ_i	s_i	κ_i	T_i	ℓ_i	s_i	κ_i	i	T_i	κ_i	τ_i
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0.3	22	27
2	0.9	48	129	129	0.1	9	129	129	1.0	66	794	784	1.1	15	216	134	2	0.3	33	44
5	1.0	0	0	129	0.2	0	0	700	1.0	0	0	784	1.1	0	0	785	5	0.4	76	94
10	1.2	0	0	129	2.1	3	24	1024	5.2	75	1024	784	1.4	0	0	1024	10	0.4	109	171
15	3.6	0	0	129	2.1	1	0	1024	5.2	0	0	1024	2.6	1	0	1024	15	0.5	172	248

Figure 5. **(Left)** Complexity estimates for predicting the first /dev/(u)random outputs generated during boot. T_i is the maximum time in seconds (relative to boot start) that output i was requested; ℓ_i is median number of inputs preceding output i; s_i is the median complexity for predicting inputs preceding output i; and κ_i is the minimum cumulative complexity across all trials on a given platform. **(Right)** Complexity estimates for GRI outputs. Here τ_i is the actual attack complexity (in bits) of the attack we implemented.

for every pair of inputs and exclude any with $|\rho| > 0.4$ (no such pairs were found). We use 0.4 as our threshold simply because this is the same value used in prior work [22]. This is a somewhat arbitrary threshold, but generally speaking $|\rho| \leq 0.4$ implies little or no correlation among the values being compared.

We compute the empirical min-entropy H_∞ for each input by counting the number of unique values that appear after masking away all but the lower α bits of each cycle counter and taking the $\log_2$. We then compute a lower bound on the security of each output i by summing the empirical min-entropy of each input in the set. Again, we compute two values, the first is the complexity of predicting all the inputs in a given set for output i. Recall that for a given trial k, S_i is the set of inputs that precede output i (but occur after output $i-1$). We define the lower-bound for the complexity of predicting this set as $\sigma_i = \sum_{\vec{p} \in S_i} H_\infty(\vec{p})$. For a given trial, the cumulative complexity λ_i for output i is again the maximum input set complexity $\lambda_i = \max\{\sigma_1, \ldots, \sigma_i\}$. Since the size of input sets vary from trial to trial, we compute σ and λ individually for each trial and then report on the median σ values and minimum λ values among all trials in our dataset.

Figure 6 shows the results of this analysis. In all cases these results are more conservative than the method using just the KS test, but we note that with $t = 200$ trials, each input is limited to contributing at most $\log_2 200 = 7.6$ bits of entropy whereas the KS test often estimates $\alpha > 7.6$. We note that many inputs show $H_\infty(\vec{p}) = 7.6$, and so the estimate appears restricted by the number of trials in our dataset. Still, after masking for only the lower α bits, we find no pairs of inputs with correlations larger than 0.4 on any platform which provides further evidence that these lower bits are uniformly distributed and thus difficult for an attacker to predict.

C. GRI boot-time analysis

To predict the 32-bit output of the GRI RNG, an attacker needs to know the state of the GRI before the call (HV and S, 128-bits and 512-bits, respectively) as well as the inputs used (J, CC, and P). When booting on a VM, the initial state of HV and S are all zeroes. S remains zero until it is initialized from /dev/(u)random (after approximately 100 calls to GRI in our observations). If the correlation between

	Native		Xen		VMware		EC2	
i	σ_i	λ_i	σ_i	λ_i	σ_i	λ_i	σ_i	λ_i
1	0	0	0	0	0	0	0	0
2	77	77	68	68	452	445	114	69
5	0	77	0	485	0	452	0	390
10	0	77	0	809	515	445	0	668
15	0	77	0	809	0	515	0	668

Figure 6. Complexity lower bounds for predicting inputs during boot using the min-entropy analysis method after masking away all but the lower α bits of each cycle counter. σ_i is the median input complexity (among all trials) for output i and λ_i is the minimum cumulative complexity (among all trials).

the jiffies counter J and the cycle counter CC is known (they are based on the same underlying clock), then the only unknown to an attacker is CC at the time each output is generated. The worst case scenario occurs on VMware where the cycle counter is virtualized by default, and so begins at 0 each time a VM is booted. In our experiments with VMware, we observed only 2 unique values of J at the time the first call to GRI is made. So if an attacker correctly guesses the CC and its associated J value, then future values of J can be computed using the ratio of cycles to timer ticks. We therefore focus only on the cycle counter.

We use a complexity estimate similar to that in the last section, except that when bounding the complexity for output i of the GRI RNG we do not assume the attacker knows the prior outputs. If we did, then each output would only have as much uncertainty as a single cycle counter carries — the GRI RNG does not attempt to deal with checkpointing attacks and never adds entropy except during output generation. For GRI, we define s_i to be the minimum number of lower bits α that appear uniformly distributed across the cycle counters used when output i is generated across all t trials. We use the same algorithm for computing α as we use for /dev/(u)random. Our computation of κ_i for GRI differs, we define κ_i as the sum of all preceding s_j values: $\kappa_i = \sum_{j \in [i]} s_j$. Again, this is because we are excluding checkpointing attacks.

Figure 5 (right table) shows the resulting complexity estimates κ_i for the first few outputs i of GRI from 200 boots on VMware (results on Xen and EC2 were similar). If we exclude the secret value S from GRI, which is a known value at the start of boot, then GRI has a maximal security state of 128-bits (the size of its hash chaining variable). GRI

reaches this state after 10 calls, well before the secret value S is initialized at approximately the 100th call. For the second output and beyond, predicting the internal state by guessing inputs is no easier than guessing any single 32-bit output value. The first value, however, shows less than ideal security for κ_1. We explore this next.

Predicting early GRI outputs. To confirm that, in fact, there is a vulnerability, we build an attack that works as follows. First, we collect a dataset of multiple boots using our instrumented kernel. From each of t traces, we group all the cycle counters from the first call to GRI, all the cycle counters from the second, and so on as we did with previous complexity estimates. Now, however, we select a *range* of cycle counters at each depth to include in the attack. To make the attack more efficient, we search the smallest contiguous range that covers a fraction (we use 80%) of the observed cycle counters in the dataset. This excludes some outliers and provides a moderate speedup of the attack time. We let τ_i denote the logarithm of the search space resulting from this process. Figure 5 shows the values of τ_i for the first few outputs using our dataset of 200 boots on VMware with the instrumented kernel. Again, only the first output is weaker than the desired 32-bits of security.

To evaluate this interpolated attack model we analyzed the first call to GRI experimentally. We constructed a kernel with minimal instrumentation to capture only the calls to GRI and performed 100 boots on VMware. We remove one trace from this dataset (the victim) and train an attack on the remaining traces to identify a range of possible values for the cycle counter CC. The remaining values (HV, J, P and S) are trivially known for the first call on this platform. We use a GRI simulator and iterate over the identified range of values for CC. The attack is successful and we verify that we can produce the full internal state HV, not just the output value. This is useful for validation since collisions are very likely when one tests up to 2^{27} guesses for a 32-bit number; the probability of a collision is 1 in 32.

A successful attack indicates that security is less than it should be for the first output. However, we note that taking advantage of this would require the ability to test which of the 2^{27} values are correct. This value is the stack canary for the `kthreadd` (kernel thread daemon) process. It is not clear that this weakness can be exploited, but this represents a failure of the RNG.

V. Snapshot Resumption RNG Security

Modern VM managers allow users to pause a running VM, make a copy of the entire state (called a snapshot) of the VM including CPU registers, memory, and disk, and later use that copy to restart the VM in the exact state at which it was paused. Both Xen and VMware support this, though Amazon EC2 does not, nor do any other clouds to our knowledge. Nevertheless, snapshots are often used

Situation	Snapshot state	Repeats until	# bits
(1) Cached entropy	UP.ec $\in [8, 56]$	UP.ec $= 0$	UP.ec
(2) Racing fast pool	Any	IntP overflow	∞
(3) Transfer threshold	IP.ec < 192	IP.ec ≥ 192	∞

Figure 7. Three situations leading to reset vulnerabilities with /dev/urandom. The symbol ∞ represents no limit on the number of repeated output bits before the condition in the third column is met.

in other settings such as backups, security against browser compromise, and elsewhere [8,25].

We consider two threats related to VM snapshots. First, we consider VM reset vulnerabilities [8,25], where resuming from a snapshot multiple times may lead to the RNG outputting the same values over and over again. Second, we consider an attacker that obtains a copy of the VM snapshot (e.g., if it is distributed publicly), meaning the attacker has effectively compromised the state of the RNG at the time of the snapshot. Here the question is whether the attacker can predict outputs from the RNG or if, instead, the RNG is able to build up sufficient fresh entropy to recover from the compromise.

A. Reset vulnerabilities (/dev/(u)random)

We show that /dev/(u)random suffers from VM reset vulnerabilities: the RNG will return the same output in two different VM resumptions from the same snapshot. There are several different situations that give rise to this vulnerability, all related to the values of the relative entropy counters and other state at the time the snapshot is taken. Figure 7 summarizes three situations that we have observed lead to reset vulnerabilities with regards to /dev/urandom. Note that these situations are not mutually exclusive, though we will exercise them individually in our experiments. As discussed below, these situations can also cause repeated outputs from /dev/random.

We use the following method to exhibit reset vulnerabilities. A guest VM boots under its default, unmodified kernel and runs for 5 minutes to reach an idle state and starts a userland measurement process designed to: detect a VM reset, capture the input pool entropy count (using /proc/fs) upon resumption, and perform a series of 512-bit reads from /dev/urandom every $500\,\mu s$ until the experiment completes. To perform VM reset detection, the userland measurement process runs a loop that samples the (non-virtualized) cycle counter using the `rdtsc` instruction and then sleeps briefly ($100\,\mu s$). When a sufficiently large discrepancy between subsequent cycle counters is detected (we use 6.6 billion cycles, which is about 2 seconds), the detection process exits the loop and begins reading values from /dev/urandom. Thus we begin capturing outputs from /dev/urandom immediately after snapshot resumption. For each experiment, we captured 10 snapshots while the system is idle, performed 10 resets from each snapshot and examined the resulting RNG outputs.

We performed experiments on both Xen and VMware. However, we experienced occasional errors when resuming from a snapshot on Xen: the guest would occasionally declare the filesystem readonly (presumably because of an error upon resumption) and so below we only report on resumptions that succeed. We experienced no such errors using VMware.

For each 512-bit output produced by /dev/urandom, we declare an output a *repeat* if a full match of all 512 bits occurs in any output from a different reset of the same snapshot. Note that at 512 bits, a repeat can only occur if the same RNG state was used (otherwise SHA-1 collisions would have occurred).

(1) Cached entropy. Recall that if the entropy estimate of a secondary pool (UP or RP) has an entropy count greater or equal to the number of output bits requested, then the output is generated directly from the secondary pool without pulling fresh bits from the input pool IP. We also note that no cycle counter (or other time stamp value) is added into the hash at this point in time, which means that the output of such calls after a reset are fully determined by the state of the secondary pool at the time of the snapshot.

If the /dev/urandom entropy count has a value of $UP.ec = 8n$ for $n > 0$ at the time of snapshot, then the bits in the non-blocking UP pool will be used to satisfy any request of size $\leq 8n$ bits without transferring bits from the input pool. Since the output generation algorithm is deterministic, this results in repeated output of size $\leq 8n$ bits under these conditions. $UP.ec$ has a maximum value of 56 bits because of the internal mechanics of the RNG and so the maximum repeated output length is n bytes where $n \leq UP.ec \leq 7$. The conditions are the same for /dev/random.

(2) Racing the fast pool. Even if a transfer from the input pool occurs after reset, this alone is does not prevent repeat outputs. To generate unique outputs, the RNG requires at least one new input to the input pool *and* a transfer from the input pool to the secondary pool (UP or RP). After a reset, the most likely addition to the input pool is from the function add_interrupt_randomness() as these account for an overwhelming majority of /dev/(u)random inputs. As described earlier, these inputs are buffered in the interrupt pool (also called the fast pool) until an overflow event occurs and the contents of the interrupt pool are mixed into the input pool. This creates a race condition between interrupt pool overflow events and reads from /dev/(u)random. An overflow event occurs every 64 interrupts or if 1 second has passed since the last overflow when an interrupt input is received. During this window, reads to /dev/urandom of arbitrary size will produce repeated outputs.

For /dev/random, repeated outputs will occur during the same window until /dev/random blocks for new entropy. Thus the maximum number of repeated bits from /dev/random is 4088. To exercise this situation for /dev/urandom

we used the experimental procedure above. Because we are comparing 512-bit output values, we can rule out repeats caused by situation (1), discussed above. To exclude situation (3) discussed below (which doesn't involve the input or fast pool), we want the input pool entropy count to be much higher than 192. We achieve this by downloading a large file (1GB) prior to capturing the snapshot. The inbound packets from the download drive interrupts in the guest kernel which increases the input pool entropy count. All resumption had an initial input pool entropy count of at least 1,283 on both Xen and VMware.

On Xen, one snapshot produced no repeated outputs (we didn't win the race), and the remaining 7 snapshots exhibited at least one repeated 512-bit output (the first output requested) after resumption. Of these the maximum duration for repeats was 1.7s after resumption. This demonstrates that the RNG does a poor job of updating its state after resumption, due to the (overly) complicated pool structure and pool-transfer rules.

On VMware, in 20 snapshots with 10 resets from each snapshot, we were not able to exhibit this vulnerability using an unmodified kernel.

(3) Input pool entropy count below threshold. The input pool entropy count $IP.ec$ must reach the transfer threshold of 192 bits before fresh inputs are transferred from the input pool to the non-blocking pool UP. While the RNG is in this state, an unlimited quantity of repeatable output values can be generated from /dev/urandom. For /dev/random of course, this is not true, as repeat values will only be provided until the entropy estimate for the blocking RP pool is exhausted (as per situation (1) above).

To arrange this situation, immediately before capturing the snapshot, we execute a 10 second read from /dev/random to reduce the input pool entropy count below 192 and trigger this condition.

On both VMware and Xen, the maximum value for $IP.ec$ upon resumption was 48 — sufficient to put the RNG into situation (3). On both VMware and Xen, we observed that *all snapshots* produced repeat outputs for the duration of the experiment (30 seconds). This indicates that if $IP.ec$ is very low when a snapshot is captured, it may take more than 30 seconds for the /dev/random RNG to reach a secure state.

Entropy starvation attack for situation (3). In Section II we observed that there exists a simple entropy starvation attack against /dev/urandom, where a (malicious) user-level process simply performs continuous reads from /dev/random. The internal logic of the RNG is such that in this case the input pool will always transfer to the blocking RP pool, and never the UP pool. This can be used to extend the amount of time that /dev/urandom produces repeated outputs in situation (3) where the input pool entropy count is below the threshold to transfer bits from IP to UP. An adversary with the ability to run an unprivileged

process on the system can easily engage this condition by reading from /dev/random. If a remote attacker makes (legitimate) requests to a public interface that triggers large or frequent reads from /dev/random, then the same effect may be possible without requiring a local account.

The experimental procedure above was used with the following deviations. We execute a continuous read from /dev/random (`dd if=/dev/random`) for the duration of the experiment. After reset, the measurement process performs 512-bit reads from /dev/urandom every 1 second for a duration of 120 seconds. Upon resumption, *all* snapshots exhibited repeated 512-bit outputs for the duration of the experiment on both VMware and Xen.

Impact on OpenSSL. The experiments above show that reset vulnerabilities exist in /dev/(u)random, and give applications stale random values after resumption. We now briefly investigate the potential for this to lead to exploitable vulnerabilities against applications relying on /dev/urandom for randomness after a VM resumption. We focus on OpenSSL v1.0.1e and RSA key generation. When calling `openssl genrsa` from the command line, OpenSSL seeds its internal RNG with 32 bytes read from /dev/urandom as well as the current system time, process ID, and dirty memory buffers. We instrument this version of OpenSSL in order to observe internal values of the key generation process. We then set up a VM running an unmodified Linux kernel on VMware that will, immediately after being reset, execute the command `openssl genrsa` from the shell. We observe that just connecting to the VM via SSH to prepare it for a snapshot typically drives the input pool entropy count below 192 before we take a snapshot. This is caused because a number of processes are created during login and each new process consumes many bytes from /dev/urandom to initialize stack canaries and perform ASLR.

We captured 27 snapshots, performed 2 resets from each snapshot and then analyzed the resulting outputs from the OpenSSL instrumentation and OpenSSL's normal output. A single snapshot produced an *identical* prime p in the private key in both resets, but other values in the private key differed. Presumably, after the prime p was generated, differing dirty memory buffers caused the OpenSSL RNGs to diverge. (Knowing one prime value of a private key is sufficient to derive the other and destroys the security of an RSA private key.) Of the remaining 26 snapshots, many had identical /dev/urandom output, but typically the dirty memory buffers differed early enough in execution to produce unique outputs. These dirty memory buffers are likely different between resets because Address Space Layout Randomization (ASLR) (determined in part by GRI) shifts around the OpenSSL memory layout.

To validate this hypothesis, we then disabled ASLR on the guest VM prior to taking a snapshot by executing `echo 0 > /proc/sys/kernel/randomize_va_space`

as root and repeat our experiment for 30 snapshots with 2 resets from each snapshot. Of these, 23 snapshots produced repeated output from /dev/urandom and *identical RSA private keys*. The other 7 snapshots had input at least 1 differing value into the OpenSSL RNG after reset — variously this differing value was one of /dev/urandom output, PID, or system time.

We note that unlike prior reset vulnerabilities [25], these are the first to be shown in which the system RNG is invoked after VM resumption. In [25], the authors ask whether consuming fresh random bytes from the system RNG after a reset is sufficient to eliminate reset vulnerabilities in applications. These results answer that question in the negative, and highlights clear problems with the /dev/(u)random design for settings where snapshots are employed.

B. Reset vulnerabilities (GRI)

As described in Section II, the output of the GRI RNG depends only on the state values HV and secret S and the inputs cycle counter, jiffies and PID (CC, J, P). Across multiple resets from the same snapshot, it's very plausible for the same process (with same PID P) to be the first to request an output. So the only new information after a snapshot resumption is the cycle counter value. For a virtualized cycle counter, in which the cycle counter value will always start from the same value (stored in the snapshot), we might expect reset vulnerabilities. In fact we observe no repeated values output by GRI across any of its invocations in any of the 50 resets on VMware that we performed. This can likely be attributed to small variations in timing between snapshot resumption and the first call to GRI. For 10,000 Xen resets, with the non-virtualized RDTSC, we did not see any repeats as well.

C. Reset vulnerabilities (other RNGs)

FreeBSD. To see if reset vulnerabilities affect other RNG designs, we also perform a limited set of experiments with snapshot resumptions using an (uninstrumented) version of FreeBSD within VMware. A description of FreeBSD's design is given in [13]. We used a similar userland tool to detect resets and sample from FreeBSD's /dev/random interface (same as /dev/urandom on FreeBSD). Repeat outputs on reset were common on FreeBSD but the duration of repeats depended on the internal state of the RNG and parameters of each experiment. The maximum length of time that we observed repeats in our experiments was 100 seconds (taking 1 byte samples every 1 second) and the maximum number of repeated output bytes we observed was 7.5 KB (taking 512-bit samples every 1 ms for 120 ms).

Windows 7. We perform similar experiments on Microsoft Windows 7 running in VMware using multiple different random number generator interfaces. We tested: the rand_s

interface (for C-programs using stdlib); the CryptGenRandom interface (for Win32 applications); and the RngCryptoServiceProvider interface (for .NET applications).

In experiments with rand_s, *all* resets produced repeated outputs when reset from the same snapshot more than once, though the quantity of repeats varied. In one experiment, we performed 10 resets from the same snapshot and collected 32-bit outputs every 1 second for a total of 2000 samples (collected over more than 30 minutes). We found more than 500 (25%) repeated outputs shared between each pair of resets, and some pairs have 1000 (50%) repeated outputs. We also observe that all 2000 outputs generated in the first reset are found in some combination of the following nine resets.

Experiments with the CryptGenRandom and RngCryptoServiceProvider interfaces produced repeats as well. In all experiments we observed identical output sequences from all resets of the same snapshot. In particular, we collected 256-bit outputs every 30 seconds for a duration of 30 minutes. In our experiments, there is no evidence of a time limit after which the Windows RNGs stop producing repeated outputs (unlike the Linux and FreeBSD implementations). These security vulnerabilities have been reported to Microsoft.

These experiments on FreeBSD and Windows demonstrate that RNG reset vulnerabilities are not limited to the Linux RNGs.

D. Snapshot Compromise Vulnerabilities

If a snapshot is disclosed to an attacker, then one must assume that all of the memory contents are available to them. Not only is there likely to be data in memory of immediate damage to an unwitting future user of the snapshot (e.g., secret keys cached in memory), but the RNG state is exposed. While we can't hope to prevent cached secret keys from leaking, we might hope that the RNG recovers from this state compromise when later run from the snapshot. As we saw above, predicting future /dev/(u)random in various situations is trivial since the attacker can often just run the snapshot (on similar hardware). When not in these situations, however, and for GRI, we would like to estimate the complexity of using the compromised state to predict outputs generated after a snapshot resumption.

We use the same methodology as used above with Xen, with the workload that reads from /dev/urandom repeatedly after snapshot resumption. We then use our methodology from Section IV to give lower-bound estimates on the complexity of predicting the very first few outputs to /dev/urandom or GRI.

Figure 8 shows our estimated attack complexity after reset. The complexity estimates for the /dev/urandom outputs are much smaller than for their boot time counterparts (Figure 5 in Section IV). The security of the GRI outputs is similar to boot because GRI security under our model is driven only by the cumulative uncertainty of the cycle

	/dev/(u)random			GRI	
i	T_i	ℓ_i	κ_i	T_i	κ_i
1	0.7 ms	2	0	21 s	22
2	1.4 ms	2	20	21 s	33
5	4.1 ms	2	27	21 s	66
10	7.1 ms	2	27	21 s	105

Figure 8. The minimum estimated complexity κ_i to predict the first few outputs of /dev/(u)random and GRI after a Xen guest is reset from a snapshot. T_i is the minimum time that output i is generated (relative to resumption); ℓ_i is the median sequence length.

counters from each output request. However, /dev/urandom outputs have security dominated by the input sequence length ℓ_i. There are far fewer inputs during a resumption than at boot. This suggests possible vulnerability to prediction attacks, but for brevity we do not pursue them further having already shown above that repeats give rise to predictable outputs.

VI. THE WHIRLWIND RNG

In this section we detail the Whirlwind RNG, which provides a simpler, faster, and more secure randomness service. While our measurement study focused primarily on virtual environments, the design of Whirlwind seeks to provide security for a variety of settings and in general be a drop-in replacement for both /dev/(u)random and GRI. As such, we must handle a variety of goals:

- *Simplicity*: The current /dev/(u)random design is complex, requiring significant effort to understand and audit its design and implementation (with 1041 lines of code) [11]. In contrast, Whirlwind targets simplicity and requires 676 lines of code.

- *Virtualization security*: Unlike all prior RNG designs we are aware of, Whirlwind is explicitly designed to provide security even in virtualized environments that might entail VM snapshot and image reuse.

- *Fast entropy addition*: Whirlwind uses a simple entropy gathering function designed to be fast, usually it requires only 0.5 μs on our 2.67 GHz platform, though for 1/8 of the invocations it applies the SHA-512 compression function [24]. Despite using a slower hash function (SHA-512), we show it to be about as fast as entropy addition in the current /dev/(u)random. Whirlwind uses per-CPU input buffers to reduce lock contention and permit the amount of buffered inputs to scale with the number of CPUs.

- *Cryptographically sound*: We propose a new design for the cryptographic core of Whirlwind, inspired by the recent work of [6]. Whirlwind dispenses with the linear feedback shift registers of Linux /dev/(u)random, and achieves the robustness security goal detailed in [6].

- *Immediately deployable*: The basic Whirlwind design is a drop-in replacement for Linux /dev/(u)random, and requires no hypervisor support.
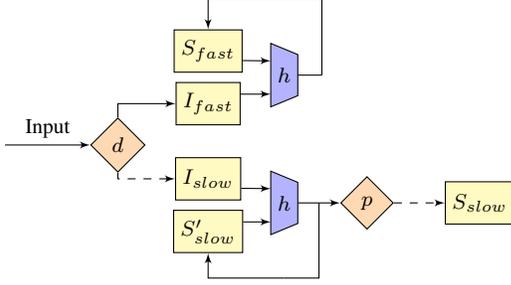
Figure 9. Block diagram of the Whirlwind RNG. Every d^{th} input is directed to the slow pool, and after p updates it is for use in output generation. Here h is the SHA-512 compression function.

A. Whirlwind design

Figure 9 depicts the main components of Whirlwind. It uses two entropy pools, a fast pool and a slow pool, as done in FreeBSD's Yarrow RNG [13]. The fast pool consists of a per-CPU input buffer I_{fast} and a single (global) seed value S_{fast} for the fast pool. The slow pool consists of a per-CPU input buffer I_{slow}, a private (internal) seed S'_{slow}, and a public seed S_{slow}. In our implementation all input buffers are 1024 bits in size which corresponds to one full message block for SHA-512. All three seeds in our implementation are 512 bits, which represents a chaining value for SHA-512. We denote the SHA-512 compression function by h and the SHA-512 hash function by H. Let n be the number of bits of output for both h and H. We initialize the seeds values as: $S_{fast} \leftarrow h(IV, 1)$ and $S_{fast} \leftarrow h(IV, 2)$ where IV is the SHA-512 initialization vector and 1 and 2 are encoded in some unambiguous manner.

Inputs are written to the fast pool I_{fast} by default and every d^{th} input is diverted to the slow pool I_{slow}. In our implementation $d = 10$ which ensures that the fast pool receives the majority of inputs and thus changes rapidly even in low-entropy conditions. Each input is 128-bits and consists of the input source's unique identifier (created by the GCC macro __COUNTER__ and encoded using 32 bits), the lower 32 bits of the cycle counter (or jiffies on platforms without a valid cycle counter), and 64 bits of optional, source-provided information. Input buffers are per-CPU, obviating the need for locking to process most inputs. We denote the macro used for adding inputs by ww_add_input().

When an input pool is full (after 8 inputs are written to a pool), a SHA-512 compression function application is performed, with the chaining variable equal to the pool seed value S_{fast} or S'_{slow} and the message block equal to the input pool. The result becomes the new seed for that pool. Locks are used to ensure that the compression function is computed atomically. Thus, Whirlwind is computing a hash over the sequence of inputs in an online fashion. This ensures the robustness security property introduced by Dodis et al. [6] and which they showed Linux's /dev/(u)random fails to achieve. Robustness requires (informally speaking)

Algorithm 3 ww_generate_bytes(b)

$s_1 \leftarrow S_{fast}$
$s_2 \leftarrow S_{slow}$
$t \leftarrow \lceil 8b/n \rceil$
$ctr \leftarrow \text{atomic\_inc}(Ctr, t) - t$
$hw \leftarrow \text{read\_hw\_random}()$
ww_add_input()
for $i = 0$ **to** t **do**
    $CC \leftarrow \text{get\_cycle\_counter}()$
    output[i] $\leftarrow H(3 \,||\, s_1 \,||\, s_2 \,||\, (ctr + i) \,||\, CC \,||\, P \,||\, hw)$
end for
ww_add_input()
$S_{fast} \leftarrow h(S_{fast}, 0^{1024})$
return first b bytes of output

Routine for generating b bytes of output from the Whirlwind RNG. The variable Ctr is a global output counter.

---

that no matter where entropy resides in the sequence of inputs to the RNG, the RNG outputs always benefit from the added entropy.

In the case of the slow pool, the internal seed S'_{slow} is used as the hash chaining value and upon every p^{th} hash the internal seed S'_{slow} is copied to the public seed S_{slow}. This ensures that the slow pool represents a multiple of p times as many inputs as the fast pool. In our implementation $p = 50$, which, combined with $d = 10$, means the public slow seed is updated every 500 inputs.

Consumers within the kernel request random values from Whirlwind using get_random_int() or get_random_bytes(). From user mode, processes read random values via the existing /dev/random or /dev/urandom read interfaces. Whirlwind handles *all* such requests in the same manner and, in particular, we have completely removed the GRI RNG and we do not differentiate between /dev/random and /dev/urandom. The current implementation does not support writing to the RNG from user-level processes, though it would be easy to add.

Algorithm 3 describes output generation in pseudocode. When Whirlwind receives an output request for b bytes, the RNG first copies the slow and fast pool seeds from static (global) memory into local memory on the stack. Whirlwind then prepares a response by computing a SHA-512 hash over the concatenation of: (1) the local copy of the slow pool seed; (2) the local copy of the fast pool seed; (3) a 64-bit request counter Ctr; (4) the current cycle counter CC; and (5) 64-bits read from a CPU hardware RNG (e.g., RDRAND), if available. The request counter Ctr is atomically pre-incremented for the number of blocks requested (to reserve counter values for output generation) and is incremented locally for each block of output. This ensures that even if concurrent requests have identical values (seeds, P, CC) the outputs are guaranteed to be unique. Two inputs are fed back into the RNG for each output requested. Finally, a single application of h is used to ensure forward security.

14

Algorithm 4   ww_bootstrap()

---
for $i \leftarrow 0$ **to** ℓ **do**
  $CC \leftarrow$ get_cycle_counter()
  ww_add_input()
  $k \leftarrow CC \bmod \ell_{max}$
  for $j \leftarrow 0$ **to** k **do**
    $a \leftarrow (j/(CC + 1)) - (a * i)$
  end for
end for

The Whirlwind entropy bootstrapping mechanism used during boot and snapshot resumption. The values ℓ and ℓ_{max} are configured parameters (default 100, 1024).

Initializing Whirlwind. We also include one special mechanism for quickly initializing (or refreshing) the entropy of Whirlwind, which is needed to prevent a boot-time entropy hole (like the ones in the legacy RNG, see Section IV) and to recover from a VM reset. For boot time, we would have liked to use the recent suggestion of Mowery et al. [22] to quickly generate entropy in the initial stages of boot via timing of functions in the kernel init function. Unfortunately, this is not fast enough for us, since we observe reads to the RNG early in init. We therefore use an approach based on timing of instructions that may take a variable number of cycles, which has been suggested and used previously [1,23]. This provides nondeterminism (by way of contention and races within the CPU state), as shown in prior studies [19]. Pseudocode is shown in Algorithm 4. In our implementation we have $\ell = 100$ and $\ell_{max} = 1024$.

Whirlwind calls this entropy timing loop before the first use of the RNG during boot, and at the start of resumption from a snapshot. The latter takes advantage of Xen's *resume callback*, which is a virtual interrupt delivered to the guest OS when it first starts following a snapshot resumption. Similar facilities exist in other hypervisors.

Entropy sources. It is easy to add entropy sources to Whirlwind, by simply inserting ww_add_input() in appropriate places. This requires no understanding of RNG internals (such as the role of entropy estimates), unlike in the existing Linux /dev/(u)random. In terms of performance, submitting an input to the RNG is fast, but may still require a single SHA-512 compression function call on the critical path. While we expect that, in deployment, Whirlwind might use a wider set of entropy sources, for comparison purposes, we restrict our experiments here to use only the same set of entropy sources as used by the current /dev/(u)random implementation in Linux as well as those called in ww_bootstrap() and ww_generate_bytes().

Hypervisor-provided entropy. As we show below, the already-mentioned software-based sources are already sufficient to provide security during boots and resets. Some users may nevertheless desire (for defense-in-depth) support for the Xen management Dom0 VM (also running Whirlwind)

to provide an additional entropy source for a guest VM's Whirlwind RNG. In current practice, host-to-guest entropy injection is facilitated via virtual hardware RNGs, that then are fed into the Linux /dev/(u)random by way of a user-level daemon (rngd). Unlike these systems, we will ensure host-provided entropy is inserted into Whirlwind immediately after a VM resumption, before any outputs are generated.

To do so, we pass additional entropy with the *Xenstore* facility in Xen, which uses shared memory pages between Dom0 (the management VM) and the guest VM to provide a hierarchical key-value store. We modified Dom0 to read 128 bytes from /dev/urandom and write the value to Xenstore. During a resume callback, Whirlwind detects that a reset occurred, reads the value from Xenstore and adds the value to the RNG via repeated input events. All this requires less than 30 lines of modification to Xen's operation library `libxl`. The entire operation requires 75 ms on average, and the rareness of the operation (once per resumption) makes this tolerable.

Other instantiations. For concreteness, we chose several suggested values of (sometimes implicit) parameters, but it is easy to modify the Whirlwind implementation to support different choices. For instance, instead of letting h be the SHA-512 compression function, one could use the full SHA-512 (or some other secure hash, such as SHA-3), which leads to the RNG computing a hash chain. The approach detailed is faster because it reduces the number of compression function calls. One might also use SHA-256, smaller or larger seed values (to trigger hashing more or less frequently), and the like. Additionally, we choose the output generation hash H as the full SHA-512. Again, this can be replaced with any suitable hash function or even AES in a one-way mode such as Davies-Meyer mode [30].

B. Security evaluation

We evaluate the boot-time and reset security of Whirlwind. We perform 50 reboots in Xen using an instrumented version of the Linux kernel using Whirlwind. We also perform 50 resets from a single Xen snapshot captured while idling (5 minutes after boot); a user level process requests 512-bit outputs from the RNG every 500 µs after resumption. As before, the instrumentation records all inputs and outputs to the Whirlwind RNG. We then perform complexity analysis as done for the legacy /dev/(u)random (see Section IV), which again ignores all input sources except the cycle counter. This provides a conservative estimate of unpredictability from the attacker's perspective. As intended, the adversarial uncertainty regarding the Whirlwind internal state hits 1024 (the maximal amount) before the first use of the RNG either during boot or after a reset. An immediate implication is that reset vulnerabilities are avoided: the probability of repeated output arising from reuse of the same snapshot is negligible.

Throughput (/dev/urandom)		
Block size	Whirlwind (MB/s)	Legacy (MB/s)
4 bytes	0.6	1.6
16 bytes	2.3	4.9
64 bytes	9.3	9.0
256 bytes	21.8	12.0

(Larger is better)

Latency (/dev/urandom)		
Block Size	Whirlwind (μs)	Legacy (μs)
4 bytes	6.9	2.5
16 bytes	6.9	3.3
64 bytes	6.9	7.0
256 bytes	11.7	21.3

(Smaller is better)

Figure 10. Comparing performance of the Whirlwind and legacy /dev/urandom implementations using dd to read 10,000 blocks of various sizes. The latency values are derived from the throughput measurements.

We have not yet evaluated Whirlwind's entropy accumulation on low-end systems, such as embedded systems [12,22]. In particular, here the cycle timing loop may provide less uncertainty because embedded system CPUs themselves have less non-determinism. In these settings, however, we do not expect to be using VM snapshots (making this use moot) and for generating entropy at boot we can use the techniques of [22].

C. Performance evaluation

We turn to evaluating the performance of Whirlwind, particularly in comparison to the existing /dev/(u)random and GRI RNGs. Our Whirlwind implementation uses SHA-512 as opposed to SHA-1 (resp. MD5 for GRI), so we expect to see a performance penalty from the use of stronger cryptography. To compare, we evaluated the throughput of reading from /dev/urandom and GRI for both Whirlwind and the legacy RNGs. While the system is otherwise idle, we execute reads of 10,000 blocks on the /dev/urandom interface for various block sizes using dd. We repeat this 100 times for each block size and report the average throughput in Figure 10. As expected, the legacy RNG performs slightly better at smaller block sizes ($\leq$ 16 byte), but is outperformed by Whirlwind at 64 and 256 byte block sizes.

We also compare measured performance of adding inputs to the new and legacy RNGs. We add minimal instrumentation to time these operations and measure performance during VM boots, resets, and during idle time. We also use these runs to measure performance of reading from GRI. The resulting performance data (shown in Figure 11) indicate the various functions were timed more than 100,000 times for each RNG. The results are that while input processing for /dev/(u)random is as fast in Whirlwind as in the legacy RNG, the GRI output interface requires 10.3 μs (one standard deviation is ± 1.8) for Whirlwind but the legacy RNG requires only 1.0 μs (± 0.5). The standard deviation is higher for Whirlwind, as this implementation more frequently performs

Execution Time	Whirlwind (μs)	Legacy (μs)
GRI	10.3 (± 1.8)	1.0 (± 0.5)
/dev/(u)random Input	0.5 (± 0.4)	0.5 (± 0.1)

Figure 11. Performance results for the kernel-only RNG GRI and input processing for /dev/(u)random inputs over 100,000 invocations of each operation. One standard deviation is shown for each in parenthesis.

hash operations than the legacy RNG.

Note that GRI is only used during process creation. In order to understand whether the GRI slowdown will cause problems in applications, we run the fork benchmark from LMbench [21] 100 times. The average latency of fork is 414 μs (with standard deviation $\pm 5 \mu$s) for the legacy RNG, and 418 μs ($\pm 5 \mu$s) for kernel with Whirlwind. Thus Whirlwind incurs only 1% overhead in this (worst-case) benchmark, and so we believe this is not a problem for practical use.

Lastly, we evaluate the overhead of ww_bootstrap() (Algorithm 4) used at boot and snapshot resumption. The time to execute the timing loops has a mean of 0.7 ms over 50 runs. Boot and snapshot resumption are rare operations, suggesting this level of overhead will not impact deployments.

Overall we conclude that Whirlwind has performance closely matching the existing RNGs, and in some cases even better despite using more expensive (and more secure!) cryptographic primitives. For this, we get a significantly simplified design and improved security.

VII. PREVENTING RESET VULNERABILITIES IN LEGACY GUESTS

While Whirlwind prevents VM snapshot reset vulnerabilities, it requires updating (at least) the kernel. We therefore consider in this section how one might try to prevent, particularly, reset vulnerabilities in legacy guests. We first consider a setting in which we can add user-level daemons to the guest, but cannot modify the kernel. We then consider a setting in which we cannot modify the guest VM at all.

Legacy hypervisor. We first consider a setting with a legacy hypervisor and management VM, but where we are able to install a user level daemon or kernel module into the guest VM. The goal of the daemon is to heuristically detect when a snapshot occurs. For VMs with non-virtualized time, the cycle counter (rdtsc instruction) returns the cycles since physical system boot. Following a resumption, the cycle counter is likely to show a large discontinuity, either a positive or negative jump. (A negative jump is possible when the machine has been rebooted or when a snapshot is reset on a different machine.) Thus, a reset can be detected by periodically polling the cycle counter, detecting large jumps (forward or backward), and assuming the cause is a reset. The overhead of such an approach is small. If the daemon sleeps 1 ms between two detection attempts, then the overhead for such daemon process uses less than 0.5% of

the total CPU time. A kernel implementation that only polls when the CPU is active can avoid some potential adverse affects, e.g., preventing a CPU from going into power-saving mode.

When a reset occurs, we use the cycle timing loop discussed in the last section to generate entropy. These cycles can then be directly written to the (legacy) /dev/urandom interface to inject entropy. As we have evaluated the uncertainty generated by cycle timing (Section VI), for brevity we omit measurements here. Even so, there exists a tension between performance (how long the daemon sleeps for) and the window of opportunity for reset vulnerabilities.

Legacy guest VM. The prior approach requires installing code on the guest VM, which may not always be possible (e.g., for already-deployed VM images). We therefore investigate whether the hypervisor or a management Dom0 can be used to inject additional entropy into a guest following a snapshot resumption. The goal is to ensure that the state of the guest's legacy /dev/(u)random RNG becomes sufficiently refreshed so as to prevent reset vulnerabilities. Unfortunately, this is a very constrained setting, since we must work with the existing deficiencies of the /dev/(u)random RNG.

Our observation is that because the /dev/(u)random RNG uses interrupts as the primary source of entropy, the management Dom0 (or possibly even a remote host) can intentionally inject a flood of random interrupts after a reset by way of network packets. As described in Section II, interrupts go first into the interrupt pool IntP, eventually flow to the input pool IP, and when the input pool entropy counter IP.ec exceeds 192 bits, these inputs will be pulled into the /dev/urandom pool UP where they affect the /dev/urandom outputs. (We focus on /dev/urandom since we observe no consumers of /dev/random in the default installations we tested.) Furthermore, interrupts only flow from IntP to IP at most every 1 second or after 64 interrupts, and then only add 1 count to the input pool entropy counter IP.ec. This means, in the worst case, we need approximately 10,000 interrupts delivered to the guest in order to *guarantee* /dev/urandom will produce no repeated outputs (assuming, of course, that no reads to /dev/random occur).

To test this, we capture a snapshot of the legacy system while it idles. Upon resumption, a user process in Dom0 sends packets at various frequencies to the guest. We repeat VM resets 50 times for the frequencies shown in Figure 12. This table shows that with no intervention, it takes on average more than 50 seconds for the /dev/urandom pool UP to receive a single transfer from the input pool, and injecting interrupts from Dom0 significantly speed this up. However, we are unable to refresh the input pool in less than 3.6 s, due in part to the fact that during resumption there is a time window during which interrupts are delivered slowly (presumably because the hypervisor is busy doing resumption-related work).

Inter-send time (μs)	Transition Time
100	3.6s
250	6.8s
500	11.2s
1,000	16.6s
2,000	24.6s
10,000	38.0s
no interrupts	> 50s

Figure 12. Average time after VM resumption for /dev/urandom entropy pool to be refreshed when injecting interrupts from Dom0 at various frequencies (shown as inter-send time in microseconds).

Discussion. While both of our legacy-compatible countermeasures in this section provide some protections against the reset vulnerabilities in /dev/(u)random, we feel that they are at best stop-gap measures. The difficulty of dealing with this from Dom0 underscores the need to move to an improved RNG (namely, Whirlwind).

VIII. RELATED WORK

Many high profile RNG failures have been reported over the years, including ones leading to: attacks against the Secure Socket Layer (SSL) implementation of an early Netscape web browser [10]; the ability to cheat at online poker [2]; insecure random values in Microsoft Windows [7]; predictable host keys in Debian OpenSSL [31]; jailbreaks against the Sony's PlayStation 3 [4]; factorizable RSA private keys generated on embedded systems [12]; predictable outputs in the OpenSSL RNG on Android [16]; and factoring RSA private keys that protect digital IDs on government-issued smart cards in Taiwan [5].

Several previous papers analyzed the Linux /dev/(u)random RNG. Gutterman et al. [11] provided the first: they reverse-engineer the design of the RNG from the source code (attesting to its complexity!); highlight problems in the hashing steps (that were subsequently fixed and the version we analyze includes these fixes); and point out that in some constrained environments such as embedded systems or network routers there might be insufficient entropy provided to the RNG.

Heninger et al. [12] show that embedded Linux systems suffer from a boot-time entropy hole which leads to exposure of cryptographic secret keys generated on affected devices. Mowery et al. [22] look to fill this boot-time entropy hole by way of timing functions in kernel initialization. Vuillemin et al. [9] perform an in-depth, empirical analysis of entropy transfers in Linux /dev/(u)random, and show that most consumers are in the kernel.

Dodis et al. [6], building off earlier work by Barak and Halevi [3], suggest that the cryptographic extraction component of RNGs should be robust, meaning an RNG should guarantee entropy is collected no matter the rate of entropy in the input stream. They show that /dev/(u)random is not robust, but do not show attacks that would affect

practice. Part of the Whirlwind design is inspired by their online hashing based extractor, though they use universal hash functions and we use cryptographic ones.

None of the above consider RNG performance in modern virtualized environments. We also do not know of any analyses of the GRI RNG before our work.

Turning to virtualized settings, Garfinkel and Rosenblum [8] hypothesized that VM reset vulnerabilities may exist when reusing VM snapshots, and analyses by Ristenpart and Yilek [25], uncovered actual vulnerabilities in user-level processes such as Apache mod_ssl that cache randomness in memory before use. In these settings, the user-level process never invoked /dev/urandom (or /dev/random) after VM resumption, and in particular they left as an open question whether system RNGs suffer from reset vulnerabilities as well. We answer this question, unfortunately, in the positive, suggesting that using /dev/urandom right before randomness use is not a valid countermeasure with the existing design, though it will be with Whirlwind.

In [27], the authors hypothesize that booting multiple times from the same VM image in an infrastructure-as-a-service (IaaS) setting such as Amazon's EC2 may enable attackers to predict /dev/(u)random RNG outputs that can lead to SSH host key compromises. Our analyses suggest that such an attack is infeasible for all uses of the Linux RNGs beyond the first during boot.

Thompson et al. [29] point out the potential for a malicious hypervisor to snoop on the entropy pools of a guest VM. Kerrigan et al. [14] investigate entropy pool poisoning attacks, where one guest VM in a cloud setting attempts to interfere with another's entropy pool by (say) sending interrupts at a known frequency to the guest. Theirs is a negative result, with their experiments showing that the attack fails. Our measurements corroborate this: even just a few bits of uncertainty about cycle counters leads to an unpredictable RNG state even in the current /dev/(u)random implementation. We also investigate using such interrupt injection as a defense.

Finally, we use CPU timing jitter as an entropy source as used in other systems, such as the haveged entropy daemon and the CPU jitter RNG [1,23].

IX. Conclusions

In this work, we performed the first analysis of the security of the Linux system random number generators (RNGs) when operating in virtualized environments including Xen, VMware, and Amazon EC2. While our empirical measurements estimate that cycle counters in these settings (whether virtualized or not) provide a ready source of uncertainty from an attacker's point of view, deficiencies in the design of the /dev/(u)random RNG make it vulnerable to VM reset vulnerabilities which cause catastrophic reuse of internal state values when generating supposedly "random" outputs.

Both the /dev/(u)random and kernel-only GRI RNGs also suffer from a small boot-time entropy hole in which the very first output from either is more predictable than it should be.

Our second main contribution is a new design for system RNGs called Whirlwind. It rectifies the problems of the existing Linux RNGs, while being simpler, faster, and using a sound cryptographic extraction process. We have implemented and tested Whirlwind in virtualized environments. Our results showed that Whirlwind enjoys performance equal (and sometimes even better) than the previous RNG.

REFERENCES

[1] Haveged entropy gatherer. http://www.issihosts.com/haveged/.

[2] B. Arkin, F. Hill, S. Marks, M. Schmid, T. J. Walls, and G. Mc-Graw. How we learned to cheat at online poker: A study in software security. The developer.com Journal, 1999.

[3] B. Barak and S. Halevi. A model and architecture for pseudo-random generation with applications to /dev/random. In *Computer and Communications Security — CCS*, pages 203–212. ACM, 2005.

[4] M. Bendel. Hackers describe PS3 security as epic fail, gain unrestricted access. http://www.exophase.com/20540/hackers-describe-ps3-security-as-epic-fail-gain-unrestricted-access/.

[5] D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. van Someren. Factoring rsa keys from certified smart cards: Coppersmith in the wild. In *Advances in Cryptology — ASIACRYPT 2013*, pages 341–360. Springer, 2013.

[6] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, and D. Wichs. Security analysis of pseudo-random number generators with input: /dev/random is not robust. In *Computer and Communications Security — CCS*, pages 647–658. ACM, 2013.

[7] L. Dorrendorf, Z. Gutterman, and B. Pinkas. Cryptanalysis of the random number generator of the Windows operating system. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):10, 2009.

[8] T. Garfinkel and M. Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *Workshop on Hot Topics in Operating Systems — HotOS-X*, May 2005.

[9] F. Goichon, C. Lauradoux, G. Salagnac, and T. Vuillemin. Entropy transfers in the Linux random number generator. Research Report RR-8060, INRIA, Sept. 2012.

[10] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr Dobb's Journal*, pages 66–71, 1996.

[11] Z. Gutterman, B. Pinkas, and T. Reinman. Analysis of the Linux random number generator. In *IEEE Symposium on Security and Privacy*, pages 371–385. IEEE, 2006.

[12] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security*, pages 205–220. USENIX, 2012.

[13] 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*, pages 13–33. Springer, 2000.

[14] B. Kerrigan and Y. Chen. A study of entropy sources in cloud computers: Random number generation on cloud hosts. In *Computer Network Security*, pages 286–298. Springer, 2012.

[15] M. Kerrisk. LCE: Don't play dice with random numbers, 2012. https://lwn.net/Articles/525459/.

[16] S. H. Kim, D. Han, and D. H. Lee. Predictability of Android OpenSSL's pseudo random number generator. In *Computer and Communications Security — CCS*, pages 659–668. ACM, 2013.

[17] P. Lacharme, A. Rck, V. Strubel, and M. Videau. The Linux pseudorandom number generator revisited. Cryptology ePrint Archive, Report 2012/251, 2012. http://eprint.iacr.org/.

[18] E. Leitl. Intel in bed with NSA. Cryptome mail list, 2013. http://cryptome.org/2013/07/intel-bed-nsa.htm.

[19] N. Mc Guire, P. O. Okech, and Q. Zhou. Analysis of inherent randomness of the Linux kernel. In *Real Time Linux Workshop*, 2009.

[20] R. McEvoy, J. Curran, P. Cotter, and C. Murphy. Fortuna: Cryptographically secure pseudo-random number generation in software and hardware. In *Irish Signals and Systems Conference*, pages 457–462. IET, 2006.

[21] L. W. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conference*, pages 279–294. San Diego, CA, USA, 1996.

[22] K. Mowery, M. Wei, D. Kohlbrenner, H. Shacham, and S. Swanson. Welcome to the Entropics: Boot-time entropy in embedded devices. pages 589–603. IEEE, 2013.

[23] S. Müller. CPU time jitter based non-physical true random number generator, 2013.

[24] National Institute of Standards and Technology. Federal information processing standards publication 180-2: Secure hash standard, 2002. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

[25] T. Ristenpart and S. Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Network and Distributed Systems Security — NDSS*. ISOC, 2010.

[26] A. Shah. About random numbers and virtual machines, 2013. http://log.amitshah.net/2013/01/about-random-numbers-and-virtual-machines/.

[27] A. Stamos, A. Becherer, and N. Wilcox. Cloud computing models and vulnerabilities: Raining on the trendy new parade. *BlackHat USA*, 2009.

[28] M. A. Stephens. Use of the Kolmogorov-Smirnov, Cramér-Von Mises and related statistics without extensive tables. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 115–122, 1970.

[29] C. J. Thompson, I. J. De Silva, M. D. Manner, M. T. Foley, and P. E. Baxter. Randomness exposed — an attack on hosted virtual machines, 2011.

[30] R. S. Winternitz. A secure one-way hash function built from DES. In *IEEE Symposium on Security and Privacy*, pages 88–90, 1984.

[31] S. Yilek, E. Rescorla, H. Shacham, B. Enright, and S. Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *SIGCOMM Conference on Internet Measurement*, pages 15–27. ACM, 2009.