UNIVERSITY OF BEIRA INTERIOR
Engineering

Auditing the Quality of Cryptographic Material in Virtual Machines

Diogo Alexandre Baptista Fernandes

Dissertation Submitted in Partial Fulfillment of the Requirement for the Degree of Master of Science in
Computer Science and Engineering
(2nd Cycle Studies)

Supervised by Prof. Dr. Pedro Ricardo Morais Inácio

Covilhã, October 2013

To my beloved family.

Lumen Ad Viem.

Pro Scientia.

Acknowledgments

A dissertation for obtaining the degree of Master of Science represents a long and demanding path. The support of very particular persons has encouraged me to continue my work even while having a full day job to attend to. I hereby thank everyone who directly or indirectly contributed constructively to this work throughout the last year.

I would like to start by thanking my closest family, for doing their best to provide for my education and future, and also for shedding light over my understanding of the world we live in. Having that said, thank you mother, Maria, thank you father, Luís, and thank you big brother, Ricardo.

I am also especially thankful to Liliana. Along my academic route, her charisma indulged cheerful, unforgettable moments of both joy and work that will echo for eternity, *memento*. I grew to saw her become a talented coworker and, above all that, a truly companion.

I express my gratitude to the Multimedia Signal Processing – Covilhã Group at the *Instituto de Telecomunicações*. The group hosted the final project of my 3-year Bachelor of Science and this dissertation research by providing adequate workstations and a continuous welcoming environment. Part of that gratitude goes to Pedro Correia, who kindly provided valuable discussions on virtualization and virtual machines to work on. It is a pleasure to work with the team and I only regret not having spent more time in the laboratory than I actually did because of the professional career.

Last, but not least, I would like to acknowledge that this dissertation would not have been possible without the guidance of my supervisor, Prof. Dr. Pedro Ricardo Morais Inácio. I would like to express my appreciation to him for his tremendous commitment in helping students to attain their goals during his professor exercise, for becoming a mentor to me, and for all his pleasant and modest words in the morning coffees and evening snacks. He has the profoundest mind I have ever seen, along with an incredible scientific knowledge, wisdom and vision.

Diogo Alexandre Baptista Fernandes

“Pedras no caminho?
Guardo todas, um dia vou construir um castelo...”

Fernando Pessoa (1888-1935)

“In theory, there is no difference between theory and practice.
But, in practice, there is.”

Jan L. A. van de Snepscheut (1953-1994)

“Any one who considers arithmetical methods of
producing random digits is, of course, in a state of sin.”

John von Neumann (1903-1957)

Resumo

A computação na nuvem é, hoje em dia, uma tecnologia já bastante integrada na indústria. As vantagens associadas a este modelo pavimentaram o caminho para a tecnologia crescer a um ritmo rápido, entretanto atraindo atenções dos mundos empresariais e académicos. Este modelo de computação migra as tecnologias de informação e os dados localizadas nas instalações dos clientes para servidores de terceiros hospedados por algum fornecedor de computação na nuvem ou de serviços. Tais conjuntos de servidores formam a nuvem, e fornecem um conjunto de capacidades úteis em forma de rede, processamento, e armazenamento, através da Internet para o utilizador final e clientes empresariais. O derradeiro propósito da computação na nuvem é de fornecer qualquer coisa como um serviço de forma interoperável, elástica e escalável, capaz de o fazer em tempo-real e autonomamente. Estes serviços são auto-provisionados e seguem o modelo de negócio de pagamento por utilização. Na sua essência, a computação na nuvem baixa os custos gerais and acelera o desenvolvimento de serviços, abstraindo os clientes dos detalhes subjacentes, mas concedendo-lhes a capacidade de aumentar a produtividade de negócio.

Os fornecedores de computação na nuvem oferecem uma grande variedade de serviços baseados em três modelos principais de implementação de serviços: Infraestrutura-como-um-Serviço (IaaS), Plataforma-como-um-Serviço (PaaS), e *Software*-como-um-Serviço (SaaS). Estes são o resultado da combinação de tecnologias inovadoras de virtualização com o paradigma bem conhecido cliente-servidor. PaaS fornece plataformas para desenvolver aplicações de computação na nuvem, enquanto que SaaS fornece pacotes de *software* já desenvolvidos. Por outro lado, IaaS fornece desde servidores simples virtuais a complexos e grandes centros de dados virtuais capazes de correr sistemas operativos (ou hóspedes) e sub-redes virtuais por cima de hipervisores. Os hipervisores gerem as máquinas virtuais, monitorizam-nas e mediam a sua criação, edição, destruição, migração, cópia, e restauro. Os hipervisores também oferecem às máquinas virtuais hardware emulado e virtualizado, graças ao suporte para virtualização das unidades de processamento central, resultando em hóspedes co-residentes com outros hóspedes, mas fazendo com que eles sejam processados como se tivessem instalados sobre a máquina anfitriã. No entanto, esta tecnologia nova de computação traz vários problemas de segurança, devido aos quais a computação na nuvem é várias vezes discutida e questionada.

Máquinas virtuais que pertençam a clientes diferentes podem ser executadas lado-a-lado no mesmo servidor físico, e esta configuração pode ser explorada via ataques como canais-paralelos e canais-escondidos. A camada de virtualização abstrai o *hardware* subjacente, e então o *kernel* de sistemas operativos hospedados pode não funcionar como funcionariam normalmente devido às suas assunções de desenvolvimento sobre o *hardware*. Isto é particularmente importante para a geração de material aleatório em sistemas operativos Linux. O gerador de números

aleatórios do Linux depende de fontes de ruído para produzir material aleatório de qualidade através das interfaces `/dev/random` e `/dev/urandom`. Este último mantém um estado de gerador de números pseudo-aleatórios, mas o primeiro depende completamente das entradas entrópicas reunidas a partir de eventos do *kernel* e que são despoletadas por interrupções do *kernel*, por movimentos do rato, pelos momentos do pressionamento das teclas, pela procura em discos rígidos através das suas cabeças, ou de leituras e escritas em discos. Como consequência da virtualização, estes eventos poderão não ser tão heterogéneos e frequentes como seriam para sistemas operativos sobre os anfitriões. Os hóspedes podem ser privados de um conjunto maior e mais diversificado de entropia por causa do número total de eventos distintos disponíveis. Isto pode resultar em níveis baixos na piscina de entropia do Linux, consequentemente propagando a escassez para a interface `/dev/random`, que poderá produzir material aleatório mais fraco e menos rapidamente. Adicionalmente, por causa das máquinas virtuais correrem co-residentemente enquanto partilham o mesmo *hardware*, é possível considerar a hipótese de que o material aleatório gerado independentemente em cada máquinas virtual poderá estar correlacionado. Ainda existe também a possibilidade de ver esta correlação em hóspedes copiados e restaurados mais tarde.

Esta dissertação foca-se no gerador de números aleatórios do Linux, investigando as saídas da interface `/dev/random` enquanto o sistema operativo é executado dentro de uma máquinas virtual. Como mencionado em cima, essas saídas poderão ser em menor quantidade e de baixa qualidade em ambientes de computação na nuvem IaaS, quando comparados para circunstâncias normais de anfitriões. Para investigar isto, o trabalho apresentado nesta dissertação estruturou-se em duas partes. Primeiro, um estudo aprofundado dos conceitos relacionados com a computação na nuvem e a sua segurança é apresentado. O fluxo de discussão passa depois para o tópico da aleatoriedade, descrevendo as abordagens utilizadas para gerar números aleatórios, de forma a introduzir o gerador de números aleatórios do Linux. Um método particular é adotado para testar esta interface em vários ambientes de teste de computação na nuvem. Este método foi seguido para levar a cabo todos os testes de análise à eficiência do gerador e à qualidade dos números aleatórios produzidos. Empiricamente é demostrado que o gerador de números aleatórios do Linux é razoavelmente lento ao disponibilizar entropia à interface `/dev/random` em múltiplos cenários de computação na nuvem. Nenhuma correlação é encontrada entre os carimbos de tempo da geração de números aleatórios entre hóspedes e anfitriões, e também entre hóspedes co-residentes. É apresentado o caso de estudo do *GNU Privacy Guard* (GPG) de forma a mostrar o impacto da lentidão do gerador, mostrando o tempo que leva para gerar chaves criptográficas quando é utilizada a interface `/dev/random` em máquinas virtuais. No entanto, a aleatoriedade que carateriza os números aleatórios produzidos pela interface são de alta qualidade e independentemente gerados em cada hóspede. Isto significa que o material criptográfico gerado em sistemas operativos encapsulados por máquinas virtuais deverão ser de alta qualidade também, embora sejam gerados mais lentamente, quando a

fonte de aleatoriedade é a interface `/dev/random` do gerador de números aleatórios do Linux.

Palavras-chave

Aleatoriedade, Computação na Nuvem, Entropia, Geradores de Números Aleatórios, Linux, Máquinas Virtuais, Segurança, Testes Estatísticos, Virtualização

Extended Abstract in Portuguese

Introdução

Este capítulo, escrito em língua Portuguesa, vem resumir sintetizadamente, mas de um modo mais alargado que o resumo, o corpo desta dissertação. Começa por apontar o enquadramento da dissertação, fluindo a discussão para a descrição do problema e enumeração dos objetivos. As contribuições principais resultantes do trabalho mostrado nesta dissertação são posteriormente enumeradas. São, então, descritos conceitos relevantes para a boa compreensão do que resta do capítulo, juntamente com o trabalho relacionado na área. O método adotado para abordar o problema é descrito na secção seguinte. Na penúltima secção são apresentados os testes levados a cabo e os resultados obtidos destes, pelo que são incluídos alguns comentários tendo em conta o problema. O capítulo termina com as principais conclusões levantadas do trabalho aqui apresentado, bem como algumas direções para trabalho futuro.

Nos capítulos escritos em língua Portuguesa, os acrónimos que são utilizados aparecem em Português nas suas formas longas, apesar das suas formas curtas serem mantidas com o respetivo acrónimo em língua Inglesa para efeitos de consistência. Não obstante, os acrónimos cuja tradução não é diretamente possível, visto a se referirem a serviços ou aplicações, aparecem em itálico como estrangeirismos.

Enquadramento, Descrição do Problema e Objetivos

A Internet é indispensável para o dia-a-dia de utilizadores e empresas. Funciona como um meio de comunicação para diversos propósitos. Ao longo dos anos tem vindo a sofrer mudanças dramáticas, nomeadamente a nível da segurança inerente à sua utilização, pelo que emancipa um clima de ciberguerra. Contudo, os anos também têm mostrado uma tendência para abstrair utilizadores da dependência de *hardware*. Este é o caso da computação na nuvem, que permite subscrever serviços prestados por terceiros, como fornecedores de nuvens para computação. A computação na nuvem adere ao princípio de computação como uma utilidade, pelo que fornece serviços autonomamente com capacidades elásticas e escaláveis. No entanto, a computação na nuvem é uma tecnologia relativamente recente, pelo que levanta bastantes problemas de segurança, particularmente devido à virtualização e no que toca à geração de números aleatórios em máquinas virtuais. A virtualização é um dos grandes proporcionadores da computação na nuvem, permitindo executar sistemas operativos inteiros dentro de máquinas virtuais, chamados de hóspedes. Isto cria uma camada de virtualização entre o *hardware* e o hóspede, criando um problema para o gerador de números aleatórios de sistemas operativos Linux.

O gerador de números aleatórios do Linux foi analisado detalhadamente em dois estudos [GPR06, LRSV12]. Este depende de eventos entrópicos providenciados pelo *kernel* do Linux para manter conjuntos de entropia que são chave para gerar material aleatório de qualidade

através de uma das suas interfaces, `/dev/random`. Os eventos entrópicos são principalmente originados a partir do *hardware*. No entanto, a virtualização abstrai os sistemas operativos hóspedes do ambiente físico em que corre. Assim, existe a possibilidade do gerador sofrer problemas de eficiência e de qualidade do seu material gerado quando corre dentro de uma máquina virtual [KC12,SBW09,Kir12]. Como os hóspedes podem ser executados concorrentemente sobre a mesma máquina física, existe também a possibilidade do material gerado independentemente em cada um estar correlacionado [Tom09], ou mesmo com as máquinas anfitriãs. Por exemplo, foi estudado [KC12] que a variável `cycle`, que faz parte dos eventos entrópicos, está correlacionada entre vários hóspedes numa nuvem Xen. Ainda existe a potencialidade de material aleatório estar relacionado quando se restaura repetidamente uma mesma *snapshot* [RY10], já que esta guarda os conjuntos de entropia anteriormente falados.

O trabalho apresentado nesta dissertação está confinado à interseção dos campos de *computação em nuvem* e de *segurança*, focando-se particularmente nos tópicos de *virtualização* e de *geração de números aleatórios*. O trabalho incide sobre a descrição de problemas de segurança em ambientes de computação na nuvem e na geração de números aleatórios em máquinas virtuais. Já que a qualidade de números aleatórios é fundamental para gerar material criptográfico de qualidade, é importante determinar o potencial impacto da virtualização nestes processos. O âmbito do problema abordado nesta dissertação recai sobre uma auditoria ao gerador de números aleatórios do Linux, particularmente à interface `/dev/random`, dando relevo ao potencial prejuízo da virtualização para a eficiência deste e para a qualidade dos números aleatórios produzidos, tendo em mente o potencial impacto criptográfico. Um conjunto de métricas devem ser produzidas de forma a quantificar a magnitude do problema em várias disposições de nuvens para computação com base em tecnologias de virtualização. Para atingir os objetivos predispostos, as seguintes cinco tarefas foram estabelecidas para dividir o trabalho deste programa de mestrado:

1. A primeira tarefa consiste em rever esta área de conhecimento, de forma a perceber os conceitos relacionados com o problema abordado nesta dissertação e, portanto, prepara para a seguintes tarefas. Deverá começar por explorar o paradigma de computação na nuvem e o campo da aleatoriedade e da geração de números aleatórios no contexto computacional.

2. A segunda tarefa foca-se na construção de uma perspetiva geral do panorama de segurança em ambientes de computação na nuvem. Esta subsequente tarefa deverá levar em conta os conceitos aprendidos na tarefa anterior para identificar pontos críticos de segurança na computação na nuvem e como se relacionam com o problema aqui abordado.

3. A terceira tarefa consista em estudar a tecnologia de virtualização e geradores de números aleatórios com mais detalhe, de forma ser possível estruturar o próximo passo adequadamente. Esta tarefa admite identificar a fonte do problema sobre análise e discutir,

antecipadamente, as suas potenciais implicações.

4. A quarta tarefa deverá elaborar no conhecimento obtido nas fases anteriores para elaborar um método consistente para quantificar o problema sujeito a análise. A tarefa também deverá incluir a identificação das ferramentas que serão necessárias para realizar a auditoria e justificar as escolhas tomadas.

5. A quinta tarefa é caraterizada pela aplicação do método anteriormente definido para obter resultados acerca da qualidade do material criptográfico gerado dentro de máquinas virtuais em ambientes de computação na nuvem. Esta tarefa deverá também identificar os cenários em quais irá ser aplicado o método, e a análise e discussão dos resultados. No final da tarefa, dependendo nos resultados obtidos, deverão ser apontadas algumas recomendações para colmatar possíveis falhas.

Contribuições Principais

Esta secção descreve brevemente as principais contribuições científicas que resultaram do trabalho de investigação apresentado nesta dissertação. As contribuições podem ser resumidas da seguinte forma:

1. A primeira contribuição principal do trabalho apresentado nesta dissertação consiste num estudo compreensivo dos conceitos básicos associados à computação na nuvem e do estado de segurança de tais ambientes de computação. No âmbito desta dissertação, a contribuição é particularmente focada no tópico da virtualização, apontando os problemas que podem advir da virtualização para a tarefa de geração de números aleatórios quando a fonte de aleatoriedade é o *hardware*. Tal estudo consistiu em rever a literatura, descrevendo o assunto e o propósito de cada trabalho encontrado ao longo do caminho. Perto do final, foi proposta uma taxonomia para classificar problemas de segurança em ambientes de computação na nuvem. Este trabalho de investigação materializou-se em duas publicações: um artigo aceite para publicação numa edição especial da *International Journal of Information Security* entitulada de *Security in Cloud Computing* [FSG+13b], publicada pela Springer, a qual tem um fator de impacto de 0.480 segundo o *Journal Citation Reports* 2012, publicado pela Thomson Reuters; e um capítulo aceite para publicação no livro titulado de *Security, Privacy and Trust in Cloud Systems* [SFG+14], publicado pela Springer.

2. A segunda contribuição principal corresponde à definição do método, bem como a sua execução, para a auditoria do gerador de números aleatórios do Linux. Esta contribuição é, em parte, o núcleo técnico desta dissertação. A auditoria é concretizada para duas vertentes. A primeira diz respeito à eficiência do gerador, enquanto que a segunda foca-se na qualidade dos números aleatórios produzidos pelo gerador. Este estudo está em parte

descrito num artigo [FSFI13] aceite para publicação nas Atas da 6th IEEE/ACM *International Conference on Utility and Cloud Computing* (UCC), a qual vai decorrer em Dresden, Alemanha, entre 9 e 12 de dezembro, 2013, e cujas atas irão ser publicadas pela IEEE Computer Society.

Uma terceira contribuição científica, resultante de investigação paralela, mas relacionada com o tema desta dissertação, envolve uma discussão do estado atual em cibersegurança. É o resultado de uma compilação diária de fontes de informação relacionadas com o tópico, levada a cabo durante o primeiro semestre de 2013. Este estudo foi o assunto de um capítulo aceite para publicação no livro entitulado *Emerging Trends in Information and Communications Technologies Security* [FSG+13a], publicado pela Elsevier (Morgan Kaufman).

Conceitos Relacionados e a Aleatoriedade em Máquinas Virtuais

O núcleo do trabalho apresentado nesta dissertação pressupõe um enquadramento inicial em vários tópicos relacionados com o tema principal. Este enquadramento consiste na descrição de conceitos base e de outros relevantes no contexto do tema, bem como no contexto do problema que é apontado após este estudo inicial, descrito no capítulo 2. Essencialmente, o capítulo descreve o modelo de computação na nuvem com ênfase na segurança que este apresenta e também retrata as abordagens existentes para a geração de números aleatórios, incluindo o gerador de números aleatórios do Linux. É dito que, apesar do modelo de computação na nuvem ser recente, apresenta já características desejáveis e múltiplas vantagens. No entanto, o seu estado da segurança não é o ideal dada a importância que esta tecnologia tem para a computação. Vários problemas [AZB13, FSG+13b] incidem ainda sobre os seus modelos de fornecimento de serviços e sobre os seus modelos de implementação, tornando a computação na nuvem arriscada a nível de segurança.

A virtualização é uma das tecnologias por detrás do sucesso da computação na nuvem. As máquinas virtuais são compostas por *hardware* virtualizado criado por hipervisores, que estão encarregues de toda a gestão desses recursos. Só por si, a virtualização também apresenta problemas do ponto de vista da segurança como, por exemplo, canais-paralelos e canais-escondidos [RTSS09], exploração da técnica de deduplicação de memória [SIYA11a], ou extração de informação sensível a partir de despejos da memória [RAC11]. Para a geração de números aleatórios, a virtualização poderá levantar ainda um problema adicional.

Os geradores de números aleatórios podem ser classificados por geradores baseados em *software* ou por geradores baseados em *hardware*. Os primeiros elaboram em fórmulas matemática para simular um comportamento aleatório sob uma determinada distribuição de probabilidade, enquanto que os segundos têm fontes de aleatoriedade verdadeira. Este é o caso do gerador do Linux, que depende de eventos entrópicos provenientes de periféricos ligados às máquinas. Porém, devido à virtualização, tanto o tamanho do conjunto de eventos como a sua hetero-

geneidade podem ser menores [KC12, SBW09, Kir12]. Como consequência, o material aleatório produzido pelo gerador em máquinas virtuais poderá ser mais fraco a nível da qualidade de aleatoriedade enquanto que é gerado mais lentamente. Com a exceção do *Kernel-based Virtual Machine* (KVM), que tem uma *driver* para fornecer fontes de entropia às máquinas virtuais, os restantes hipervisores não tratam este problema. Vários incidentes de falta da entropia são discutidos [Ama11, VMw06, Pat10, Car12] na Internet, enquanto que outros estudados cientificamente mostram efeitos devastadores [RY10, HDWH12] para a criptografia. Destes problemas surge a motivação principal para o trabalho apresentado nesta dissertação, pelo que o seu alvo de estudo convergiu para o dispositivo `/dev/random`, devido à sua dependência total na entropia colecionada pelo *kernel* do Linux.

Método para Colecionar e Analisar as Saídas de `/dev/random`

O método seguido para estudar o problema em mãos tem por base recolher amostras da interface `/dev/random` de forma a analisar a eficiência, ao guardar o carimbo do tempo de quando as amostras são geradas, e a analisar a qualidade da aleatoriedade, ao submeter os números aleatórios a vários testes estatísticos rigorosos. Estes procedimentos estão detalhados no capítulo 3. Essencialmente, a recolha de amostras consiste em guardar continuamente material aleatório produzido pela interface `/dev/random`, durante um período de tempo indeterminado. O material aleatório é guardado em forma de números inteiros de quatro bytes. Juntamente é também guardado um carimbo temporal que identifica quando a interface desbloqueou para gerar quatro bytes. Este carimbo corresponde ao número de segundos desde o *Unix Epoch*. Todos os valores são guardados para um ficheiro de texto.

A análise da eficiência incide sobre o cálculo de métricas que caraterizem os ficheiros das amostras. Por exemplo, é calculada a diferença média entre os carimbos temporais para perceber o quão frequentemente é que o dispositivo `/dev/random` fica disponível para gerar material aleatório, em função do tempo total de amostragem. Relativamente à análise da qualidade da aleatoriedade, recorreu-se à ferramenta conhecida por TestU01 [LS07, Sima], uma biblioteca de análise estatística bastante reputada nesta área. A TestU01 foi construída especificamente para testar geradores de números aleatórios sob a hipótese nula que afirma que sequências de números aleatórios devem seguir uma distribuição uniforme, e o mesmo para qualquer uma das suas subsequências. Devido à rigorosidade da TestU01, são necessários no mínimo 2^{27} números aleatórios para utilizar uma das suas baterias. Contudo, o gerador de números aleatórios do Linux provou ser lento na interface `/dev/random`, pelo que foi construída uma bateria de testes personalizada a partir da TestU01 para testar as amostras disponíveis. Os testes estatísticos incluídos nesta bateria calculam valores conhecidos, na gíria da especialidade, por p-*values*, que servem para estimar o quão próximo ou o quão longe os números estão do caso ótimo, a hipótese nula. O método aqui descrito envolve a construção de vários programas em ANSI C e *scripts* em Python 3.

Testes e Resultados

O capítulo 4 descreve pormenorizadamente cada um dos testes executados nas variadas plataformas de ensaio e cenários escolhidos. Devido aos potenciais problemas levantados pela virtualização para a geração de números aleatórios em sistemas operativos Linux, foram montadas várias plataformas de computação na nuvem para averiguar o potencial impacto sentido à superfície do gerador. Foram utilizados dois servidores de produção, dois computadores de secretária idênticos, um portátil e, adicionalmente, recorreu-se à Amazon *Elastic Compute Cloud* (EC2), uma nuvem bem considerada na vertente de fornecedor de infraestruturas como um serviço. Ambos os servidores e um dos computadores de secretária serviram para montar uma nuvem utilizando o *Xen Cloud Platform* (XCP). Nesta nuvem foram instanciadas máquinas virtuais co-residentes e não co-residentes. Outro computador de secretária foi utilizado para testes variados, enquanto que o portátil correu uma máquina virtual. Por sua vez, na EC2 foi instanciado um conjunto sensivelmente grande de máquinas virtuais.

O método anteriormente descrito foi executado em todos os cenários enumerados em cima, pelo que foram obtidos bastantes valores de métricas e de p-*values*. Essencialmente, os resultados empíricos mostram que o gerador de números aleatórios do Linux poderá não satisfazer requisitos impostos por aplicações do espaço do utilizador. Quando comparado com as condições típicas de uma máquina anfitriã, a interface `/dev/random` em máquinas virtuais perde eficiência, bloqueando frequentemente enquanto espera, em média, 59.97 segundos por entropia suficiente para gerar quatro bytes de material aleatório. Nos anfitriões foi determinado que este valor é de 2.39 segundos. Nos testes realizados, a interface gerou em média e aproximadamente dois números aleatórios por cada segundo em máquinas virtuais, enquanto que nos anfitriões esta métrica é de 4.18. Foi ainda testado este impacto em aplicações, nomeadamente no *GNU Privacy Guard* (GPG), que permite gerar chaves criptográficas recorrendo ao material aleatório da interface `/dev/random`. Os resultados foram claros, a interface `/dev/random` é lenta em máquinas virtuais comparativamente às máquinas anfitriãs. Adicionalmente, não foi encontrada uma correlação entre a geração de números aleatórios nestes dois pontos de teste.

Os achados da análise da qualidade da aleatoriedade dos números aleatórios produzidos pela interface `/dev/random` são muito positivos. Apesar do conjunto entrópico do gerador números aleatórios do Linux ser potencialmente menos heterogéneo, os resultados são positivos no sentido de que não é notada uma perda da qualidade. Mais ainda, os números são gerados independentemente dos outros gerados em máquinas virtuais co-residentes e nas máquinas anfitriãs onde o hipervisor está instalado. A análise dos p-*values* resultantes dos vários testes não mostra nenhuma falha crítica na estrutura estatística dos números aleatórios, nem quando estes são fundidos uns com os outros. Contudo, alguns dos p-*values* são considerados p-*values* suspeitos por caírem no intervalo $(10^{-10},\ 10^{-4}] \cup [1-10^{-4},\ 1-10^{-10})$. Estes podem ser justificados devido ao facto dos tamanhos dos ficheiros amostrais serem pequenos.

Conclusões e Trabalho Futuro

As conclusões e trabalho futuro são enumeradas no capítulo 5. Um secção inicial tece uns comentários com respeito à colmatação do problema apresentado nesta dissertação e à geração de números aleatórios. Em primeiro lugar, os hipervisores poderiam endereçar este problema ao nível da virtualização. Uma possível direção é a de permitir contacto mais direto entre as máquinas virtuais e o *hardware*. Por outro lado, o horizonte para a geração de números aleatórios poderá ser prolifero caso os computadores quânticos se materializem numa realidade próxima.

O modelo de computação na nuvem com certeza traz vantagens bastante significativas, esculpindo assim uma marca nas perceções de computação modernas. Porém, o seu estado de segurança está ainda na sua infância, pelo que ambos os mundos empresariais e de investigação se dedicam, para já, à mitigação de problemas de forma a construir sistemas de computação na nuvem mais seguros. O impacto da virtualização no gerador de números aleatórios do Linux é claro. A sua eficiência é prejudicada, pelo que as aplicações dependentes da interface `/dev/random` ficam em risco de ficar penduradas no fluxo de execução. Contudo, o material aleatório é, acima de tudo, de alta qualidade e não correlacionado entre máquinas virtuais co-residentes e entre máquinas virtuais e anfitriãs.

Olhando para as linhas de trabalho futuro, propõe-se investigar mais aprofundadamente como o *kernel* do Linux se comporta para o gerador de números aleatórios sob múltiplos cenários de computação na nuvem. Isto é, descobrir que tipo de eventos o gerador utiliza para a entropia em máquinas virtuais seria interessante para alargar a fronteira do conhecimento neste tópico. Para este propósito, seria necessário alterar o código do *kernel* para efeitos de monitorização do gerador de números aleatórios. As lições aprendidas a partir destes pontos de estudo poderão ajudar no desenho e implementação de um mecanismo que ultrapasse o problema da escassez de entropia em máquinas virtuais. Neste caso, deverá ser dada atenção às fontes de aleatoriedade para produzir números aleatórios de alta qualidade.

Abstract

Cloud computing is, nowadays, a mainstream technology spiraling across the industry. Its clear advantages propelled this model to grow at a fast pace, attracting attentions from both the enterprise and academic worlds along the way. This computing model offloads on-premises Information Technologies (IT) and data to outsourced servers housed on data centers and hosted by some cloud or service provider. Those sets of servers form the *cloud*, and deliver, through the Internet, a broad vertical set of capabilities for end users and enterprise customers in the form of networking, processing or storage. Ultimately, the purpose of cloud computing is to provide anything-as-a-service in an interoperable, elastic and scalable, and on-demand manner, as a completely autonomous and self-provisioned pay-as-you-go measured service. In essence, cloud computing lowers the overall costs and speeds up the deployment of services, allowing costumers to be abstracted from the underlying details, but granting them the ability to focus on increasing business productivity.

Cloud providers offer a large variety of services based on three main service delivery models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). These are the result of the combination of innovative software and virtualization technologies with the well-known client-server paradigm. IaaS hands out platforms to develop cloud applications, while SaaS provides costumers with pre-built software packages. IaaS offerings, on the other hand, range from simple virtual servers to complex distributed virtual data centers, capable of running complete operating systems (or guests) and virtual subnets on top of hypervisors. Hypervisors manage the Virtual Machines (VMs), monitoring and mediating their creation, edition, deletion, migration, snapshotting, and restoration. Hypervisors also provide overlying VMs with emulated and virtualized hardware, thanks to the Central Processing Units (CPUs) support for virtualization, making co-resident guests running as if installed on real hardware. However, this new computing technology poses several security issues, over which cloud computing is quite often discussed and questioned.

VMs belonging to different customers may run side-by-side on the same physical server, and this setup may be exploited via cross-VM attacks like side-channels and covert-channels. Because the virtualized layer abstracts the underlying hardware, operating systems kernels of guests may not behave as they would normally do because of their development assumptions, which normally state that the system will be installed on real hardware. This is particularly important for the generation of random material in Linux operating systems. The Linux Random Number Generator (RNG) relies on noise sources to output quality random material through the `/dev/random` or the `/dev/urandom` devices. The latter maintains a Pseudo-Random Number Generator (PRNG) state, but the former relies completely on entropic inputs gathered from kernel events, which are triggered by noise sources like kernel interrupts, mouse movements,

keystroke timings, disk head seeks, or disk reads and writes. As a consequence of the virtualization, those kernel events may not be as heterogeneous and frequent for guests as they would normally be for host operating systems. Guests may be deprived of an otherwise more diverse set of entropic inputs due to the lesser number of distinct and available events. This can result in low levels on the entropy pool of the Linux RNG, consequently propagating the scarceness to the `/dev/random` device, that might output weaker random material on a slower basis. Additionally, because VMs can run concurrently while sharing the same underlying hardware, it is possible to consider the hypothesis of material generated independently on each VM to be correlated. There is also the possibility for this correlation to be seen on snapshotted and restored guests.

This dissertation is focused on the Linux RNG, investigating the outputs of the `/dev/random` device while the operating system is running inside a VM. As mentioned above, those outputs can be fewer and of less quality on IaaS cloud environments then on normal host circumstances. To investigate this subject, the work presented in this dissertation was structured in two main parts. First, an in-depth study of the concepts related with cloud computing and its security is presented. Second, the discussion then moves into the topic of randomness, describing the approaches used for generating random numbers, so as to introduce the Linux RNG later on, and the problem virtualization poses to it. A particular method was adopted for testing the `/dev/random` device over several cloud computing testbeds. All tests that were performed followed this method to examine the throughput efficiency of the generator and the quality of its outputs in terms of randomness. It is empirically shown that the Linux RNG is reasonably slow in making entropy available for `/dev/random` on multiple cloud computing scenarios. On the other hand, no correlation is found between the timings of the generation of random numbers in guests and hosts, and between co-resident guests. To show the impact of the slowness of the generator, a case study of the GNU Privacy Guard (GPG) is presented, showing that it takes a perhaps great amount of time to generate cryptographic keys when using the `/dev/random` device on VMs. Nonetheless, the randomness characterizing the random numbers outputted by the device are of high quality and independently generated on each guest. This means that cryptographic material generated on operating systems encapsulated by VMs should be of high quality as well, though generated more slowly, when the inherent source for generating random numbers is the `/dev/random` device of the Linux RNG.

Keywords

Cloud Computing, Entropy, Linux, Random Number Generators, Randomness, Security, Statistical Tests, Virtual Machines, Virtualization

Contents

1 Introduction **1**

1.1 Focus and Scope . . . 1
1.2 Problem Statement and Objectives . . . 3
1.3 Adopted Approach for Solving the Problem . . . 5
1.4 Main Contributions . . . 6
1.5 Dissertation Overview . . . 7

2 Related Concepts and Randomness in Virtual Machines **9**

2.1 Introduction . . . 9
2.2 Cloud Computing . . . 10
2.2.1 Service Delivery Models . . . 10
2.2.2 Cloud Deployment Models . . . 12
2.2.3 Data Center Constructions and Security . . . 14
2.3 Virtualization and Cloud Security . . . 15
2.3.1 Virtualization Techniques . . . 16
2.3.2 Security in Cloud Environments . . . 17
2.4 Randomness in Computer Science . . . 20
2.4.1 Random Number Generators . . . 20
2.4.2 True Random Number Generators . . . 21
2.4.3 Pseudo-Random Number Generators . . . 22
2.4.4 The Linux Random Number Generator . . . 25
2.5 Randomness in Virtual Machines . . . 29
2.6 Conclusions . . . 30

3 Method for Collecting and Analyzing `/dev/random` Outputs **33**

3.1 Introduction . . . 33
3.2 Method Overview . . . 33
3.3 TestU01 . . . 34
3.3.1 Installation, Building and Running . . . 36
3.3.2 Example in ANSI C . . . 36
3.4 Collecting `/dev/random` Outputs . . . 37
3.4.1 Format of the Samples Files . . . 38
3.4.2 Post-Processing the Sample Files . . . 38
3.5 Analysis of `/dev/random` Throughput Efficiency . . . 38
3.5.1 Python Script . . . 39
3.6 Analysis of `/dev/random` Outputs Quality . . . 39
3.6.1 Overview . . . 39

3.6.2 TestU01 Statistical Tests 40
3.6.3 Main Analysis Program in C Programming Language 44
3.7 Conclusions 44

4 Tests and Results **47**
4.1 Introduction 47
4.2 Testbeds 47
4.2.1 Setups and Specifications of the Hosts and Guests 48
4.2.2 Virtualization Configurations and the `RdRand` Instruction 49
4.3 Tests and Datasets 49
4.3.1 Details of the Tests 50
4.3.2 Metrics Notation 51
4.4 Throughput Efficiency of `/dev/random` Outputs 52
4.4.1 Analysis of the Instants 52
4.4.2 Histograms 53
4.5 Case Study — Impact of Low Entropy on Applications 54
4.6 Quality of `/dev/random` Outputs 55
4.6.1 Review of the Statistics and Configurations 56
4.6.2 Less Stringent Batteries 57
4.6.3 Analysis of the p-values 59
4.6.4 Biasing the Tests 61
4.7 Conclusions 61

5 Conclusions and Future Work **63**
5.1 Final Remarks 63
5.2 Main Conclusions 65
5.3 Directions for Future Work 67

References **69**

List of Figures

2.1 Cloud computing service delivery models and encapsulating components (taken from the second scientific contribution [FSG+13b] of this dissertation). 11
2.2 Schematics of the construction of the Linux RNG as perceived in [GPR06, LRSV12]. 26
3.1 Illustration of the method adopted to study the /dev/random on guests and hosts. 34
4.1 Schematics of the various testbeds used for conducting the tests. 48
4.2 Histogram of the number of samples per instant for test number 3. 53
4.3 Histogram of the number of samples per instant for test number 7. 54

List of Tables

2.1 Summary of the cloud deployment models with regard to ownership (Organization (O), Third-Party (TP), or Both (B)), management (O, TP, or B), location (Off-site, On-site, or B), cost (Low, Medium, or High), and security (Low, Medium, or High). This table was taken from the second scientific contribution [FSG+13b] of this dissertation. 14
4.1 Datasets collected in the several scenarios used in the scope of this research work along with the results obtained for the metrics defined to evaluate them. 50
4.2 Results of the gpg tests. 55
4.3 Configuration for the fixed parameters used on the customized TestU01 battery for all the statistical tests. 57
4.4 Configuration for parameters n and d used on the customized TestU01 battery with regard to tests number 1 through 16. Each pair of values in the cells is of the form (n, d), with d not being applicable to some of the statistics. 58
4.5 Results of the p-values outputted by the customized TestU01 battery with regard to tests number 1 through 16. 60

List of Listings

3.1 Example of a TestU01 C program using some functions of the TestU01 library. . . 37
3.2 Example of a sample file resultant from the execution of the Python sampling script. 38

Acronyms

ANERD	Asynchronous Network Exchanged Randomness Daemon
API	Application Programming Interface
APT	Advanced Persistent Threat
BYOD	Bring Your Own Device
CPU	Central Processing Unit
CSA	Cloud Security Alliance
CSIRT	Computer Security Incident Response Team
CSPRNG	Cryptographically Secure Pseudo-Random Number Generator
CSRT	Cloud Security Readiness Tool
DaaS	Data-as-a-Service
DDoS	Distributed Denial of Service
DHCP	Dynamic Host Configuration Protocol
DMZ	DeMilitarized Zone
DSA	Data Signature Algorithm
EC2	Elastic Compute Cloud
EGD	Entropy Gathering Daemon
GAE	Google App Engine
GCC	GNU Compiler Collection
GOF	Goodness-of-Fit
GPG	GNU Privacy Guard
HAVEGE	HArdware Volatile Entropy Gathering and Expansion
HTTPS	HyperText Transport Protocol Secure
IaaS	Infrastructure-as-a-Service
IDS	Intrusion Detection System
IP	Internet Protocol
IPS	Intrusion Prevention System
ISP	Internet Service Provider
IT	Information Technologies
KVM	Kernel-based Virtual Machine
LCG	Linear Congruential Generator
LFG	Lagged Fibonacci Generator
LFSR	Linear Feedback Shift Register
MitB	Man-in-the-Browser
MitM	Man-in-the-Middle
MSR	Multiple Recursive Generator
NAT	Network Address Translation

NFS	Network File System
NIC	Network Interface Card
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OCCI	Open Cloud Computing Interface
OS	Operating System
P2P	Peer-to-Peer
PaaS	Platform-as-a-Service
PRNG	Pseudo-Random Number Generator
PUE	Power Usage Effectiveness
RaaS	Routing-as-a-Service
RAM	Random Access Memory
RDP	Remote Desktop Protocol
RFC	Request for Comments
RNG	Random Number Generator
RSA	Rivest, Shamir, Adleman
S3	Simple Storage Service
SaaS	Software-as-a-Service
SDK	Software Development Kit
SHA-1	Secure Hash Algorithm-1
SIEM	Security Information and Event Management
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SOC	Security Operations Center
SSD	Solid-State Drive
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TGFSR	Twisted Generalized Feedback Shift Register
TLS	Transport Layer Security
TRNG	True Random Number Generator
USB	Universal Serial Bus
VCPU	Virtual Central Processing Unit
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMM	Virtual Machine Monitor
VNC	Virtual Network Computing
VNIC	Virtual Network Interface Card
VPC	Virtual Private Cloud

VPN	Virtual Private Network
VPS	Virtual Private Server
XaaS	Anything-as-a-Service
XCP	Xen Cloud Platform
ZKP	Zero-Knowledge Protocol

Abbreviations

Please consider the following abbreviations with the respective meaning when later invoked in the text:

e.g.	originates from the Latin expression *exempli gratia* which means “for example.”
i.e.	originates form the Latin expression *id est* which means “that is” or “in other words.”
i.i.d.	relates to probability distributions, meaning “independent and identically distributed.”

Chapter 1

Introduction

This dissertation describes a research work on the subject of generating random material from the Random Number Generator (RNG) of the Linux kernel in virtualized environments. The focus and scope of the dissertation are firstly described in this chapter, followed by the problem statement and objectives. The adopted approach to solve the problem, the main contributions, and the dissertation overview are discussed in the three last sections.

1.1 Focus and Scope

The Internet of today is an essential resource for the daily work of users and enterprises, being used for entertainment and businesses, including gaming, e-commerce, online banking, amongst others. Since its first steps, several perceptions like cultural, legal and moral have changed throughout the time regarding its correct utilization. Security awareness has also shifted, mainly due to some security events that, from time to time, come as wake-up calls to us let know something is not right. Currently, the Internet cyberspace is used for cyberwar, comprising a new war domain after land, sea, air and space. The evolution of the Internet has been fueled by the development of several technologies at a fast pace. Initially, the supporting infrastructure was composed by telephone lines, while today it is using fiber-optics that reach businesses and residential homes. Furthermore, history has showed that there has always been an inclination towards the disassociation of users from computer hardware needs, even when the Peer-to-Peer (P2P) paradigm seemed to gain friction. The subsistence of the client-server paradigm is a long-term proof of that fact. Actually, in the fifties and sixties, the mainframes were already accessed using time-sharing approaches from terminals with no processing or storage capabilities. In recent years, the long-envisioned era of computing as an utility has arrived in the form of *cloud computing*.

The *cloud* buzzword is inspired by the symbol used to represent the Internet in diagrams, which conveys a certain level of abstraction of the details of the components within it and how they work. Widespread principles state Information Technologies (IT) should be on-premises so as to be within a trusted and self-managed network environment. Cloud computing, however, moves IT responsibilities to outsourced servers wrapped in an on-demand and self-provisioned pay-as-go service model, autonomously instantiated for each customer with elastic and scalable capabilities regarding networking, storage and processing. Some data center houses the cloud, which is managed by some cloud or service provider. Subscribed services are accessed

via the Internet using standard technology. Services can be deployed on public, off-premises clouds, on private, on-premises clouds, or on hybrid clouds, a mixture of the previous two for added security and trust. Essentially, it all boils down to hardware and IT abstraction for lowering overall costs and boosting productivity, by allowing customers to focus on promoting businesses. For that purpose, cloud computing may take the form of one of the following main service delivery models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). These models constitute the basis to deliver a portfolio of Anything-as-a-Service (XaaS) solutions. While SaaS dispatches pre-built software packages, PaaS provides platforms to write cloud applications that can be constructed according to the requirements of the customer. Behind the scenes, IaaS clouds use virtualization technology to deliver a wide spectrum of solutions, ranging from sandboxes, passing through Virtual Private Servers (VPSes), to entire virtual data centers that can be distributed across various clouds and even data centers.

Cloud computing is believed to continue to gain momentum in the next few year, and it is expected to play an even more distinct role in computing in the future. Virtualization is perhaps the main propeller for cloud computing to grow at a significant speed, but it is also one of the technologies responsible for new security issues specific to clouds. Security issues spanning all of the service delivery models are currently worrying some prospective costumers, while adopters show satisfaction towards the technology. For example, storing sensitive data on multi-tenant clouds is risky because cloud (and big data) workloads mixes both data-at-rest and data-in-motion. Worse, clouds can be accessed by several customers simultaneously, some of which potentially comprising badly intentioned users. Intel and AMD added virtualization support for their Central Processing Units (CPUs) in the mid-2000s and, since then, virtualization has been used for all kinds of purposes, namely dynamic and automated malware forensic analysis. Under general terms, the virtualization layer consists of Virtual Machine Monitors (VMMs), or hypervisors, which programs that are in charge of creating, deleting, editing, migrating, snapshotting and restoring one or more Virtual Machines (VMs). VMs run entire and independent Operating Systems (OSes), known as guests in this case, and are stored on image files when suspended. These files save both storage and memory contents. VMs can be seamlessly copied and migrated to run over other hypervisors installed on other machines. When instantiated, VMs are dynamically allocated with emulated and virtual hardware resources, provisioned by the hypervisors with little overhead. Such resources include Virtual Central Processing Units (VCPUs) and Virtual Network Interface Cards (VNICs). Hypervisors further create virtual network devices like Dynamic Host Configuration Protocol (DHCP) servers and switches for connecting VMs on virtual networks to simulate real networks. The virtualization layer abstracts applications, services, and OSes from the hardware on which they run, and thus OSes kernels might not behave as expected due to their inherent hardware construction assumptions. For example, this is important for the RNG embedded within the kernel of most Linux distributions and, consequently, for

the cryptographic material that is generated with basis on this RNG.

The Linux RNG has been thoroughly scrutinized in [GPR06, LRSV12]. It relies on a variety of entropic inputs gathered from a multitude of noise sources that are gathered by the kernel in form of events, to generate quality random bytes at acceptable speeds. Entropic inputs can be regarded as spurring elements that are blended into an entropy pool for adding randomness to it, and thus they are important in the process of generating random material. This type of RNG construction is well regarded throughout the field of random number generation, since *entropy* is a key ingredient for the randomness soup. In turn, random numbers are crucial components to many areas of knowledge, including computer-based simulation, cryptography, and gambling systems. But, the abstraction imposed by the virtualization layer separates the guests plane from the underlying hardware, without providing explicit contact with erratic randomness sources without first passing through the virtualization layer. In this regard, such kernel events may not be provided in full to the Linux RNG, particularly to its `/dev/random` interface that completely depends on those entropic inputs to generate random material of high randomness quality. Therefore, IaaS clouds introduce this new challenge regarding the generation of random material on, at least, Linux guests, which may be deprived of an otherwise more diverse set of entropic inputs due to the lesser number of distinct and available entropic inputs. Since VMs can run side-by-side, another issue emerges in the case of sharing the same randomness sources across different guests, which can result in correlated random numbers.

The scope of this dissertation is confined to the intersection of the fields of *cloud computing* and *security*, being particularly focused on the topics of *virtualization* and *random number generation*. The research work presented herein settles on the study of the security issues raised by the cloud computing model of the impact that virtualization has on the normal functioning of OSes and applications, namely of the Linux RNG. The quality of cryptographic material is dependent of the quality of outputs of a *true* RNG and, as such, it is of critical importance to assure its satisfying provisioning on new computing environments. This provided the motivation for auditing the Linux RNG on VMs, notably in terms of output efficiency and randomness quality. Under the 1998 version of the ACM Computing Classification System, the *de facto* standard for computer science, the scope of this dissertation falls within the D.4.6 [**OPERATING SYSTEMS**]: Security and Protection–*Security kernels* and G.3 [**PROBABILITY AND STATISTICS**]: Random number generation categories and descriptors.

1.2 Problem Statement and Objectives

The main problem addressed in this dissertation concerns the possible lack of quality of the cryptographic material generated and stored on VMs of clouds. As it will be shown afterward, this quality is often directly dependent of the true and uncorrelated nature of the random bits generated by the OS kernel RNG. As such, the problem at hands is better described in terms

of the probable negative impact that virtualization may have on the efficiency and quality of hardware-based RNGs, more specifically on the Linux RNG, on which this research work is focused on. This subject has been given little attention in the past with just a few studies addressing the problem, though complaints about this can be found on Internet forums and a few studies address other parts of related issues.

Motivated by the importance that the correct generation of strong random numbers has to many disciplines, the field of random number generation has actively proposed several Pseudo-Random Number Generators (PRNGs). With basis on rigorous and robust mathematical theory, software-based RNGs can run on any platform under normal circumstances and keep up performance and output quality, for some applications. However, the Linux true RNG takes a different approach to generate randomness, basically consisting of gathering events for producing entropic inputs for an entropy pool of random bits. In other words, the kernel harnesses unpredictable events provoked by hardware, thereby piling up randomness in the form of bits that is dropped onto the entropy pool. Such hardware sources include keystroke timings, mouse movements, and disk head seeks. However, a virtualization layer creates an abstraction gap between overlying guests and underlying hardware, potentially narrowing down the amount and type of events used to provide entropic inputs to the Linux RNG. The lesser types of events can lead to a less heterogeneous set of entropic inputs, while the smaller amount can affect how fast the Linux RNG responds to calls for generating random material. This problem on the RNG might have significant ramifications with respect to security. Cloud instances are often booted up and immediately go off to create some cryptographic keys, without giving a large enough time-window to develop consistent entropy pools [SBW09]. There is also the possibility of sharing the same environmental noise since VMs can run co-resident over the same physical machine [Tom09]. Kerrigan and Chen [KC12] found the `cycle` variable that is part of kernel events timings to be highly correlated among Xen guests. In terms of the generation of random material, sharing the same source of randomness can result in outputs independently generated in guests and hosts to possibly be correlated. Finally, restoring snapshots previously taken restores entropy pools and PRNGs states. In this case, material produced after taking the snapshot and after restoring it may be correlated as well [RY10].

Foremost, the issues outlined above can contribute to deficient cryptographic material. In fact, low levels of entropy or poor randomness can yield severe consequences for cryptosystems [RY10, HDWH12], not to mention the other areas for which random number generation is also important. Since cloud computing is rising quickly, and given its importance nowadays, it is decisive to assess these problems in the short-term, and address them conveniently. This dissertation is a minor step toward that end.

Given the groundwork defined above, the first main objective of this dissertation is to initially study virtualization and identify where it fits in the cloud panorama, so as to point out the

implications it poses to this computing model. Subsequently, another main objective should be fulfilled by selecting what should be audited in the Linux RNG for assessing the problem while having in mind the underlying cryptographic impact. At this point, and in order to be consistent with the problem definition, such an analysis should be carried out on the efficiency and randomness quality of the `/dev/random` device outputs. Another main objective concerns creating a set of metrics for quantifying the magnitude of the problem, while conveying a one-to-one comparison of the results on various IaaS cloud scenarios.

1.3 Adopted Approach for Solving the Problem

In order to achieve the aforementioned objectives, the research work of this masters program was divided into the following five tasks:

1. The first task consists in reviewing this area of knowledge, so as to understand the basic concepts related with the problem at hands and, therefore, prepare for the subsequent tasks. It should start by exploring the cloud computing paradigm and the randomness field in the computational context.

2. The second task is focused on constructing a general perspective of the security landscape of cloud environments. This follow-up task should take into account the concepts learned from the previous task to identify crucial security points of cloud computing and how they are related to the problem addressed herein.

3. The third task consists on studying virtualization technology and RNGs with more detail, so as to be able to structure the following step accordingly. This task comprises identifying the source of the problem under analysis and discuss beforehand its possible implications.

4. The fourth task should elaborate on the knowledge obtained in the previous phases to devise a consistent method to quantify the problem subject to analysis. It should also include identifying the tools that will be necessary to perform such auditing and justifying the choices that were made.

5. The fifth task is characterized by the application of the previously defined method to obtain results concerning the quality of the cryptographic material generated within VMs in cloud environments. This task should also include the identification of the scenarios in which the method is to be applied, and the analysis and discussion of the results. At the end of this task, depending on the obtained results, some guidelines for improving possible flaws should be handed out on this part.

1.4 Main Contributions

This section describes briefly the scientific contributions resulting from the research work included in this dissertation. Summarily, these contributions can be described as follows:

1. The first main contribution of the work presented in this dissertation comprises a comprehensive study of the basic concepts related with cloud computing and on the security state of such computing environments. Within the scope of this dissertation, the contribution particularly focuses on the virtualization topic, while emphasizing the caveats it may pose for the discipline of random number generation when the underlying source of randomness is entropy. Such a study consisted in reviewing the literature thoroughly, describing the subject and aim of each work compiled along the way. Towards the end, it was proposed a taxonomy[1] for classifying security issues in cloud environments. This research work was the subject of two publications: an article accepted for publication in a special issue of the *International Journal of Information Security* entitled *Security in Cloud Computing* [FSG+13b], published by Springer, which has an impact factor of 0.480 according to the *Journal Citation Reports* 2012, published by Thomson Reuters; and a book chapter [SFG+14] accepted for publication in the book titled *Security, Privacy and Trust in Cloud Systems*, published by Springer.
The second publication is actually a preliminary survey giving insight into cloud security issues. Basic concepts are first explained, and then a review of security issues of several topics is presented. The discussions therein included underpin security along the text. The first publication is an enhanced study of the second publication. It improves the contents discussed therein while also enlarging the scope radius by explaining more concepts and security issues. A taxonomy for security issues is then proposed. In addition, open challenges are pointed out and recommendations for achieving securer cloud environments are given afterward.

2. The second main contribution comprises the definition of a method, and its subsequent execution, to audit of the Linux `/dev/random` device in terms of its output efficiency and randomness strength when running on VMs. The speed is quantified by means of a set of metrics against the same metrics found on hosts, while various samples of random numbers outputted by the device are submitted to stringent statistical tests and are compared with the results of the hosts also. Both the efficiency and the randomness quality are assessed on multiple cloud computing scenarios using different types of hypervisors and inherent technology. This study represents the core of the technical work of this dissertation and it is the subject of the paper entitled *Randomness in Virtual Machines* [FSFI13], accepted for publication in the Proceedings of the 6th IEEE/ACM *International Conference on Utility and Cloud Computing* (UCC) which will be held in Dresden, Germany, between

[1]This taxonomy is only included in the referred journal article.

the 9th and the 12th of December, 2013, which will be published by the IEEE Computer Society.

A secondary contribution involves a theoretical discussion on the current state in cybersecurity, giving out a quick perspective of cybercrime and cyberwarfare trends while citing real-world events so as to better illustrate each idea. This is the result of a daily effort performed throughout the first half of 2013, on which the infosec community was scanned on a daily basis for interesting feeds on the subject. This study was the subject of a book chapter accepted for publication in the book [FSG$^+$13a] named *Emerging Trends in Information and Communications Technologies Security*, published by Elsevier (Morgan Kaufman). Parts of this work are on the basis of several sections of chapter 2 of this dissertation.

1.5 Dissertation Overview

This dissertation is organized in five main chapters. The **body** of the dissertation is constituted by three chapters, preceded and succeeded by the Introduction and the Conclusions and Future Work chapters, respectively. The contents and organization of the main chapters of this dissertation can be summarized as follows:

- **Chapter 1** contextualizes the problem addressed in this dissertation by introducing the topic, and the focus and scope of the research work prior to discussing the problem statement enumerating its objectives. The adopted approach for solving the problem is also outlined in this chapter, along with main contributions of the underlying research work. The organization and structure of the dissertation is included in last.
- **Chapter 2** first explains the basics of cloud computing, so as to better understand the remaining part of the dissertation. It also provides a detailed perspective over security on clouds, focusing on the virtualization technology in the intermediate part of the chapter, providing the substract to understand the source of the problem studied herein. The discussion then evolves to the review of the randomness problem on computers. The Linux RNG is described with detail and with focus on the potential problems posed to it by virtualization. The related work is also reviewed along this chapter.
- **Chapter 3** outlines the method taken to address the problem described in the previous chapter. The discussion follows a top-down approach, by first depicting and discussing an overview of the method, and then providing the details and fundamenting the choices taken along the way. Subsequently, details on how random numbers are harvested from `/dev/random` are given also, followed by sections explaining how its efficiency and the quality of its outputs are tested, respectively.
- **Chapter 4** starts from the presentation of the testbeds used for performing the several tests devised in the scope of this work. The datasets resultant from those tests are then

characterized according to a set of metrics useful for quantifying the efficiency of the Linux RNG. Subsequently, the impact of low entropy levels is illustrated by generating cryptographic keys which depend on the good provisioning of entropy. The quality of the random numbers acquired from the sample datasets is finally demonstrated by empirical means.

- **Chapter 5** includes some final remarks regarding the problem addressed in this dissertation, and then presents the main conclusions of this dissertation, together with directions for future work.

In order to maintain consistency along the dissertation, the long form of an acronym is repeated in the initial chapters, namely Resumo, Extended Abstract in Portuguese, and Abstract, and only once more from the Introduction chapter onward.

Chapter 2

Related Concepts and Randomness in Virtual Machines

This chapter discusses cloud computing and introduces the concepts underlining random number generation processes. When possible the discussion is performed from the security point-of-view. The related work on randomness regarding the problems arising from virtualization is included in here as well.

2.1 Introduction

Despite being relatively new, the field of cloud computing is already mainstream throughout the industry and the academia. The benefits of this paradigm and associated technologies attracted worldwide attentions, with several scientific and industry conferences gathering professionals and researchers from all over the globe, such as the IEEE/ACM International Conference on Utility and Cloud Computing (UCC) and VMworld, respectively. This field has been quite active in terms of scientific publications and white papers. A great part of the research effort is being placed into the security panorama of cloud environments that encompass a great deal of security issues. Within this line of thought, Section 2.2 introduces cloud computing and its security state. One of its key technologies, virtualization, is discussed with more detail in Section 2.3, because it is of critical importance within the context of this dissertation, followed by a focused perspective of the security on cloud environments.

In contrast to cloud computing, the field of *true* and *pseudo* random number generation in computers is characterized by contributions that started in the late forties. Its inherent dilemma turns this field into one of the greatest struggle between man and machine in the search for replicated true randomness. The quality of random numbers is of utmost importance to several areas of computer science, namely cryptography. Most cryptographic operations and mechanisms elaborate on assumptions of unpredictability for upcoming events, and the quality of random numbers is crucial for generating cryptographic material (*e.g.*, encryption keys) to subsist in the long-term. Section 2.4 discusses the generality of random number generation, introducing concepts and illustrating how randomness is carried out using deterministic algorithms in computers. Afterward, in Section 2.5, the problem of randomness in VMs is motivated, since cloud computing in the form of IaaS introduced new challenges in this regard.

2.2 Cloud Computing

Cloud computing enabled several new computing potentialities. It delivers remote pools of resources in the form of networking, storage and processing, replacing on-premises IT infrastructures with those remotely located on a data center that is in charge of some cloud provider. Cloud environments are elastic, thereby providing scalable resources, and their services are self-provisioned while applying a measured pay-per-use business model. This means customers pay for only what they explicitly require. In this sense, their costs are lowered because they do not need to manage entire on-premises IT infrastructures (to some extent), but rather choose what best fits their requirements in a granular manner. Cloud computing concepts actually date back to the fifties, when mainframes were installed on server rooms so as to be used by multiple users in a time-sharing manner. A couple of decades later, in the seventies, the International Business Machines (IBM) released an OS called VM that allowed to have various virtual systems living on the same physical environment [Ste13]. This is the case of IaaS clouds, later discussed in the text.

In 2011, the National Institute of Standards and Technology (NIST) released a document [MG11] containing a definition for cloud computing, in an effort to obtain a consensus regarding this concept. This definition is the one widely accepted nowadays, even for such a rapidly evolving and disseminated technology. Clouds are mainly defined in terms of the service delivery models and deployment models. While the former relates to the types of cloud services, the latter dictates how and where clouds are deployed. These are next detailed with regard to their security. And since cloud infrastructures are mostly put on data centers, it makes sense to review some good security practices regarding those facilities so as to contain insider threats and avoid natural catastrophes. Most of the ideas discussed in this section are also described in the second scientific contribution [FSG+13b] of this dissertation.

2.2.1 Service Delivery Models

Three main cloud service delivery models are usually described on the literature [SK11, XX13]. These are the IaaS, the PaaS and the SaaS, sorted upwardly, and give service providers the possibility to adhere the XaaS and thus provide a wide spectrum of solutions, ranging from small and particular resources to large and complex products. Examples include Data-as-a-Service (DaaS) [VPT+12] and Routing-as-a-Service (RaaS) [CYG+11]. The main service delivery models are represented in Figure 2.1, together with the main components supporting the models at the backend and at the frontend. Inspired by its operating and business models, cloud-specific infrastructure orchestration is where the core of cloud innovation comes in, conveniently supported by cloud OSes. One of the main obstacles cloud adopters normally find is to have to trust outsourced third-parties with their sensitive company data and, as such, trust is a subjective component stretching throughout all of the cloud stack. As it is today, the Internet is already a non-safe, non-trustable place, hence the gap in the *Trust* line in the right side of the figure for

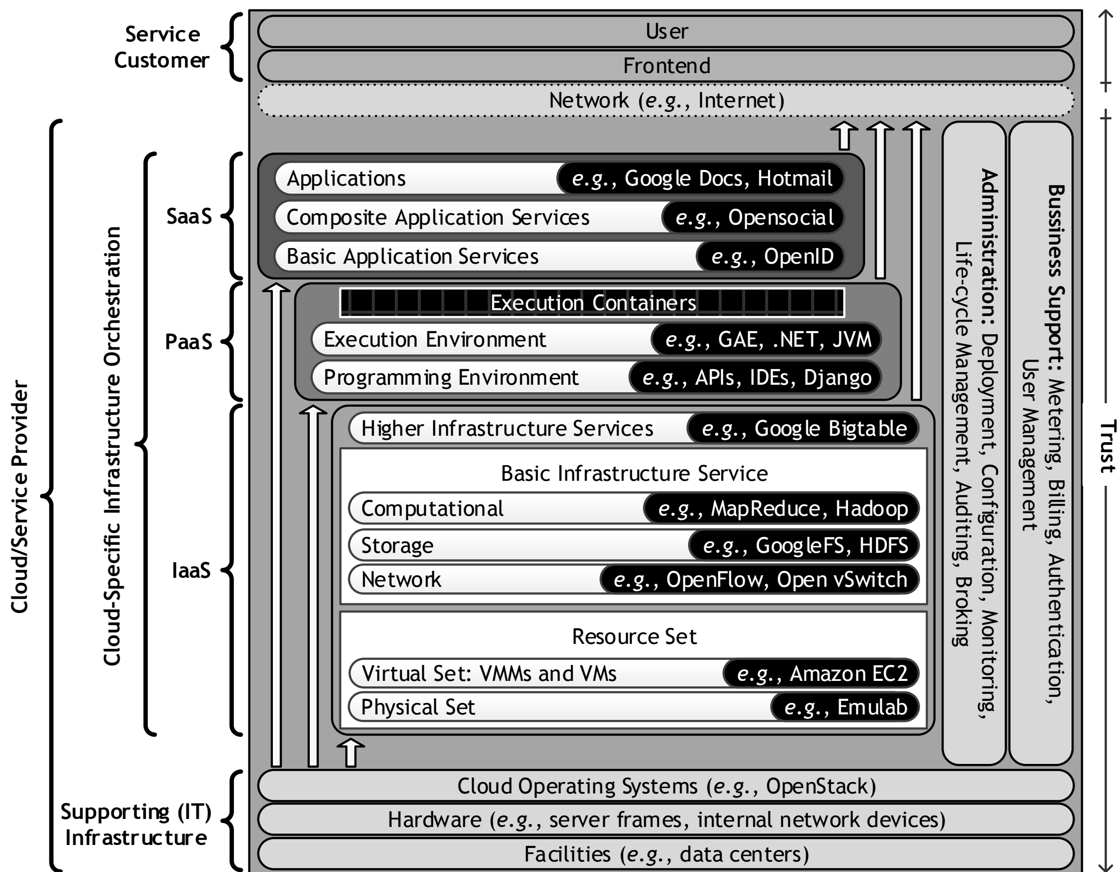

Figure 2.1: Cloud computing service delivery models and encapsulating components (taken from the second scientific contribution [FSG+13b] of this dissertation).

the space referring to the Internet. The service delivery models are described as follows:

- **Infrastructure-as-a-Service**: IaaS clouds revolutionize how IT infrastructures are deployed by replacing entire hardware systems with the virtualized counterparts. In this model, the user is disengaged from hardware needs. Virtualization technologies support IaaS clouds by providing emulated hardware that can be easily and seamlessly setup to instantiate new VMs. IaaS businesses are commonly packaged in the form of VPSes and virtual desktops. VPSes come bundled with features similar to traditional physical servers, but are on VMs and therefore can benefit from the advantages of virtualization. Virtual desktop solutions provide remote connections to the OSes via some remote connection protocol like the Remote Desktop Protocol (RDP) or the Virtual Network Computing (VNC). In IaaS, the traditional view of deploying IT infrastructures is largely overstepped. Whereas installing hardware is usually costly and time-consuming, dynamically allocating and instantiating virtual resources is done on the minute, autonomously, and with minimal effort and resource consumption. In addition, IaaS clouds benefit of other features like autonomic balancing, migration, fault-tolerance and ease of management, adequate for building virtual data centers. The most well-known and scrutinized IaaS cloud provider is perhaps the Amazon Elastic Compute Cloud (EC2) [Amab], a pioneer cloud in this field.

IaaS cloud providers are responsible for the security and management of the underlying virtualization technologies and of the physical substrate, together with the segregation of parallel virtual networks. Both Xen and OpenFlow platform achieve [FMM+11] an adequate flexibility and performance on network virtualization. The IaaS model is subject to study in this dissertation.

- **Platform-as-a-Service**: The PaaS model basically consists in providing customers with developing platforms for programming their own cloud applications. Applications run on isolated containers on a side-by-side manner, thereby creating multi-tenant scenarios, where each tenant accesses its own applications. Containers are managed by the underlying platform. Popular developing platforms in the non-cloud world include Microsoft .NET and Oracle Java, but they can also be used for cloud purposes. Some can share platform pieces like runtime components, libraries and database engines, but others might instead provide pre-packed disk images with the stack required by the customer [RMVC+12]. Cloud applications adhere to the Service-Oriented Architecture (SOA), in this case in the form of web services deployed over the Internet. In terms of security, coding metrics should be put forth so as to quantify the quality of written code in the hope of avoiding applications prone to attacks, especially by other tenants on shared infrastructures. The Google App Engine (GAE) [Goo] is a well-known PaaS cloud with Software Development Kits (SDKs) for Python, Java and Go. The cloud provider is responsible for managing the platforms, including its updates and security features.

- **Software-as-a-Service**: The SaaS model packs pre-built software for customers. The software offered by cloud providers is typically directed at replacing some enterprise applications, like billing and other business management software and office solutions, without the need to develop them and configure them thoroughly. All that is required to access and interact with the such applications is a web browser. SmartCloudPT [Por] is a Portuguese SaaS cloud capable of delivering such applications. Hence, Man-in-the-Browser (MitB) and Man-in-the-Middle (MitM) are the most likely threats to this model. Management and support to the specific applications is a responsibility of the cloud providers.

2.2.2 Cloud Deployment Models

Three main cloud deployment models are usually discussed throughout the literature, namely: public clouds, private clouds, and hybrid clouds. Though deserving normally less attention, two other deployment models named community clouds and Virtual Private Clouds (VPCs) are also discussed. All of these cloud deployment models can be summarized as follows:

- **Public Cloud**: The infrastructure behind a public cloud is, in general, owned by a cloud provider. A public cloud provisions for whoever wants to subscribe available services with the pay-as-you-go approach, therefore being accessed by multiple tenants from anywhere

in the world. The cloud is located at an off-site location accessible through the Internet, thereby being inherently more risky for its openness to the maliciousness coming from the outside. In this case, Service Level Agreements (SLAs) should be extensible and well detailed so as to cover probable atypical scenarios with respect to security.

- **Private Cloud**: For an enterprise environment, a private cloud is bought for placing it within the trusted network perimeter behind security controls. This on-premises cloud deployment has higher costs when compared to public deployments, because it requires highly specialized technical crews to manage the cloud. That includes looking into control, compliance with company policy, and resiliency, in turn giving the organization total control over the cloud that may be in charge of the organization itself or of a third-party company.

- **Hybrid Cloud**: The hybrid cloud deployment model commonly mixes the public and private models in order to overcome the disadvantages each one transpires. This model requires equipment to implement features right onto the networking fabric so as to extend the trusted perimeter to an off-site cloud managed by an assumed untrusted third-party. Such setup typically means that more data and application is transferred between the two endpoints. An hybrid cloud has divided costs and is placed on both on-site and off-site locations.

- **Community Cloud**: The infrastructure underlying a community cloud is shared amongst a well defined set of entities. A community cloud is built to support a common interest between its shareholders, defining detailed specifications for each one in terms of access to resources. It may be managed by the owners committee or a third-party organization and may be placed on-premises or off-premises. The community cloud deployment model eliminates the security risks of public clouds and avoids the big budgets of private clouds.

- **Virtual Private Cloud**: The VPC is less mentioned by sources in the literature, but interestingly its most suitable example is the Amazon VPC [Amac]. A VPC uses Virtual Private Network (VPN) pipe connectivity and isolated resources on the cloud to create virtual private or semi-private clouds. A VPC seats on top of any model previously described, likewise a VPN that is put upon other networks. Because of this, customers do no worry about working in shared or public environments. Implicitly, a VPC is more secure, is placed both on-premises and off-premises, and is managed by the organization and the cloud provider.

From the customer viewpoint, these deployment models are defined by several characteristics that should be well assessed before committing to a specific one. Several factors must be taken into account, namely the purpose of the cloud, available budget, and compliance with company policy. The features above discussed for each model are summarized in Table 2.1. The hybrid

Table 2.1: Summary of the cloud deployment models with regard to ownership (Organization (O), Third-Party (TP), or Both (B)), management (O, TP, or B), location (Off-site, On-site, or B), cost (Low, Medium, or High), and security (Low, Medium, or High). This table was taken from the second scientific contribution [FSG$^+$13b] of this dissertation.

Deployment Model	Ownership	Management	Location	Cost	Security
Public	TP	TP	Off-site	Low	Low
Private	O or TP	O or TP	On-site	High	High
Community	O or TP	O or TP	On-site or Off-site	High	High
Hybrid	B	B	B	Medium	Medium
VPC	B	B	B	Low	High

cloud deployment model is believed by Cisco [Cis13] to be where cloud computing is heading. For now, however, it is still unclear if that opinion is going to stick because the explosion of cloud computing in recent years created a heterogeneous cloud industry. Nebula One [Neb13], for instance, is a sleek private cloud solution that acts much like a single computer, being easily turned on or off. The customer has the ability to choose the number of cores, combined storage and memory of the infrastructure, which provides Application Programming Interfaces (APIs) compatible with OpenStack and both Amazon EC2 and Simple Storage Service (S3). The expectations for this solution are high [dB13]. A side effect from that explosion culminated in the cloud industry to be almost devoid of standards for interoperable clouds. This makes it difficult to build *interclouds*, a place of cloud networking for seamless interoperability and ubiquitous communications between the clouds. Interclouds would overcome the lock-in issue and free data movement among clouds from distinct providers.

2.2.3 Data Center Constructions and Security

The physical substrate of clouds is very similar to that of a tightly-coupled cluster layout. A great number of servers is interconnected and then a virtualization layer is placed above them for delivering elasticity and scalability by means of VMs. In this sense, clouds are held in big IT rooms just as what happens to high performance clusters, and even the mainframes of the past. The facilities housing clouds are nowadays called data centers with the particularity of being built specifically for clouds. Construction of data centers has in mind geological and environmental aspects, such as location, temperate, humidity and earthquakes; and other aspects like political, governmental, and energy-savings related factors. Some data centers enjoy *free cooling* for regulating the temperate within IT rooms by fanning in and out cool and hot air, respectively, so as to obtain good levels of Power Usage Effectiveness (PUE)[1]. Many data centers strive to meet quality body standards by achieving highly reliable and available facilities in terms of uptime and efficiency. Multiple backup generators of electricity with large tanks of

[1]The PUE refers to the ratio between the amount of energy expended to power up a certain processing unit and the amount that entered the procedure. It can be seen as the ratio between the power entering the data center versus the power used to run the infrastructure within it. The powering process is not completely efficient, and some energy is lost for other purposes. The closest this value is to one, the better.

gasoline make data centers achieve, many times, 99.99% of uptime while being fully fault-tolerant, thereby reaching the highest level for data centers quality, the tier 4 certification. The lowest level is tier 1.

Because clouds host many types of sensitive data, it is important to ensure physical protection in order to contain break-ins and insider threats. The security of a data center goes beyond the digital perimeter and, in the first place, falls within the scope of surveillance by means of security cameras in closed circuit. With respect to access management, data center employees should be overseen so as to avoid them to abuse their access to the facility. Different factors of authentication should be spread throughout the different layers or areas of a data center, depending on the security required. For example, unique identity cards assigned to employees can be used to open door locks for accessing restricted areas and palm print scanners may control the access to more confidential areas. In addition, only personnel with higher security clearances and expertise should perform management tasks on cloud servers. Furthermore, servers can be isolated within cages with padlocks for enhanced physical security, for which the opening keys would be kept with the customers. This adheres to the VPC principle. Weighting chambers can also be installed in IT rooms entrances and exits to check for stealing attempts inside the rooms.

Ideally, cloud networks should be monitored on-site by a team specialized in security. Traditionally, large enterprises (*e.g.*, Internet Service Providers (ISPs)) mount monitoring infrastructures composed of security controls (*e.g.*, firewalls or Intrusion Prevention Systems (IPSes), and Intrusion Detection Systems (IDSes)) that log events for analysis and correlation by a large platform capable of segregating events that point to some intrusion or security issue by interpreting the network health status. This is part of a Security Information and Event Management (SIEM) process, which is assigned to a Computer Security Incident Response Team (CSIRT) team placed within a Security Operations Center (SOC) for staying in charge of the digital perimeter security in the general sense.

2.3 Virtualization and Cloud Security

Regarded as one of the main components of cloud computing, virtualization is a novel technology bringing the opportunity of running various OSes on the same physical machine or set of machines without worrying about the underlying hardware. This commodity brought various advantages from several viewpoints, and virtualization is now widely used in both academic and industry sides. But on the other hand, cloud computing is not yet a well matured technology regarding security. Below is explained how virtualization is tightly related with the cloud computing model and which security issues such model has raised.

2.3.1 Virtualization Techniques

IaaS clouds stand out for their reliance on virtualization techniques that made possible cloud computing to rise at a fast pace and thereby bring computer science a step closer to the utility computing era. Research on virtualization boomed off when Intel and AMD added full virtualization support for their CPUs in the mid-2000s. Virtualization is the process of abstracting computer applications, services and OSes from the hardware on which they run. Hardware is provided in an emulated fashion, that is, virtual hardware devices (*e.g.*, VNICs and VCPUs) are created per each VM instance that is booted up. VMs are stored on usually large image files that save volatile and persistent memory storage on-the-fly. OSes running within VMs are commonly called *guests*, and they operate normally as if installed on host machines in direct contact with hardware. In turn, they can have multiple applications running on top. If a machine is suspended, its contents are dully saved. When the VM is resumed, the OS and applications can run or re-run as normally.

VMs cannot run independently and, for managing them, VMMs are used. VMMs, also called *hypervisors*, comprise the software responsible for creating a virtualization layer between guests and the hardware without providing direct access to the bare metal components. The access is strictly mediated by the hypervisors. For example, hypervisors can implement a round-robin algorithm for scheduling VMs access to the physical processor. Hypervisors further create other virtual devices for replicating a real, Ethernet network, by creating virtual switches, DHCP servers, and Network Address Translation (NAT) devices. For example, on popular solutions like Oracle VirtualBox [Oraa], the interfaces `vmnet0`, `vmnet1` and `vmnet8` are used for bridged, host-only and NAT configurations, respectively. Several commercial products, like the ones mentioned below, allow to arrange VMs on different network setups by linking them together using these virtual network devices. Bridged networking hooks up a VM to the underlying physical network, being accessible like any other computer on the local network. The Internet Protocol (IP) address is received via a physical DHCP server in this case. Host-only creates an isolated and private network between the VMs and the underlying host, while NAT shares the Internet connection of the host by masking the activity of the VM as if it came from the host. NAT networking conceal VMs from the rest of the network. Hypervisors can also allow to link a VNIC to a hardware Network Interface Card (NIC) for further flexibility to integrate with the physical network. Custom-made virtual networks can be configured to group particular VMs, just as like breaking up a network in subnets for different departments of an organization. More robust enterprise solutions allow to build more complex networks (*e.g.*, DeMilitarized Zones (DMZs), and Virtual Local Area Networks (VLANs)).

There are two types of hypervisors one can choose from. *Native hypervisors*, also known as type-1 or bare metal hypervisors, are installed directly on the host machine similarly to an OS. In fact, they can be regarded as specialized OSes dedicated to one single purpose. Paravirtu-

alization is a sort of virtualization that changes the kernel code of guests so as to make them comply with the hypervisor by making system calls to it, instead of executing I/O instructions that the hypervisor simulates. Native hypervisors usually provide paravirtualization since they supply predefined templates for guest OSes. *Hosted hypervisors*, or type-2 hypervisors, are installed upon normal OSes and function as an application. Guests, in turn, run on top of those hypervisors. Popular free solutions include the VMware Player [VMw], Oracle VirtualBox [Oraa], RedHat-maintained Kernel-based Virtual Machine (KVM) [Red], Microsoft Hyper-V [Mic], and Xen [Theb], an open-source project of The Linux Foundation. Well-known paid products include VMware Workstation and vShpere [VMw], Oracle VM Server [Orab], Parallels Desktop and Virtuozzo [Par], and Citrix XenServer [Cit] (the paid version of Xen). Examples of native hypervisors are Xen (uses paravirtualization) and KVM, while of hosted hypervisors are VMware Player and VirtualBox.

Since VMs are simply image files, they can be easily copied. This enables hypervisors to save particular states of the guests by snapshotting the file, that is, by simply backing up the files. At a later time, if a VM becomes compromised by means of malware or if it suffers an outage, it may be desirable to rewind VMs to those past states. This is in part similar to the Windows restoration points, but all data since the snapshot is lost. With this feature clouds can move or migrate VMs in real-time to other locations seamlessly while maintaining integrity of their contents. Thanks to virtualization, VMs can be easily instantiated on the minute while spending little resources and time, and thus they provide elastic and scalable services on-demand.

2.3.2 Security in Cloud Environments

The field of cloud computing security spans a large set of topics that vertically affect the cloud stack, each in its own way. A great amount of security issues incur in cloud environments, with several surveys reviewing the literature on this area [AZB13, FSG+13b]. As previously mentioned, public cloud deployment models are outsourced to some third-party cloud provider. Therefore, those clouds inherit some of the security issues already prevalent in the Internet cyberspace of today. That includes malware threats, Distributed Denial of Service (DDoS) attacks (a nostalgic trend) like the Spamhaus [Pri13] case in early 2013, in which a 300 Gb/s flood on a tier 1 ISP was registered, or Advanced Persistent Threats (APTs), which are recent players in the cyberwar domain. Beyond that, customers must deposit a fair amount of trust on the contracted infrastructures managed by the cloud provider. As illustrated in Figure 2.1, trust is a subjective component orthogonal to the cloud business model, ranging from the bare metal to the service frontend interfaces. The loss of control is therefore implicit in this model. For regaining part of that trust and control, the hybrid deployment model connects two endpoints securely right in the networking fabric. Speaking of subjectivity, there are also concerns in the legal domain. Cloud providers may provide data redundancy usually by copying data to

another data center. The 3-2-1 rule[2] is a well accepted principle in this regard. However, large providers can have several data centers spread throughout the world. In the case of having data flows crossing borders, the laws and jurisdictional bounds therein applied change. It is also not clear how a cloud provider complies with customers requests when under a subpoena, nor if there exists cloud subcontractors. These confuse customers regarding the who and the where data lawful interception may fall under with. Moreover, the Internet is also being wrongly tapped by governments for surveillance purposes, by capturing data in transit from fiber-optic links (by using *fiber-optic splitters* [EFF13]) or from ISPs, and by putting backdoors on major service providers, or even secretly weakening well-known cryptographic body standards, some speculate [Gal13, Gre13]. This is the case of the National Security Agency (NSA) programs recently leaked by its ex-employee Edward Snowden, who created lots of commotions and discussions regarding its correctness in breaching end-user privacy, beyond degrading trust on service providers. Bruce Schneier, a highly acclaimed expert in the security field, believes [Sch13] that whistleblowers are required in order to protect people from the abuse of power. To cloud adopters, this is a warning, for their data may be tampered with. The Cloud Security Alliance (CSA) has recently released a document [CSA13] outlining privacy level agreements for cloud services that attempt to allay some of these issues, at least in the European Union.

Besides losing control over configurations, management and security to some extent, customers are not sure where exactly their data resides within the cloud. Because of the cloud autonomic balancing and elasticity, VMs can be moved around, and thus pinpointing their exact physical location is nearly impossible. There is also a high possibility of running VMs co-resident with some other VMs belonging to other customers. This shared virtualized scenario bears new security challenges that can be exploited via popular side- and covert-channel alike attacks [RTSS09, BNP+11, ZJRR12], or via memory deduplication techniques [SIYA11a, SIYA11b] and memory dumps [RAC11]. Hypervisors comprise a large and complex piece of software and, because of that, they should be handled with forethought so as to avoid critical vulnerabilities, as argued by Perez-Botero *et al.* [PBSL13]. Research on this line of work [MLQ+10, HS12] aims at minimizing the hypervisor so as to avoid complexity and possible vulnerabilities, hardening [LSLS09] the hypervisor with additional code, and discarding [SKLR11] the hypervisor so as to provide VMs with more direct access to hardware. In addition, cybercriminals follow trends and, corroborating this rule, they have recently placed some effort in the development of malware to cloud environments by adding anti-debug, anti-sandbox and anti-VM capabilities. By detecting such a presence, the malware behavior flow can change rapidly so as to avoid automated malware analysis by means of sandbox techniques (*e.g.*, Cuckoo sandbox). Efforts [SLC+11, ZMZ12] are being carried out to counterattack this trend, namely via the development of tools to detect malware with such characteristics, including incorporating new analysis techniques into

[2] In the backup context, the 3-2-1 rule dictates to have three copies of the same data in two different formats with one of the copies off-premises.

hypervisors [OGC+12]. Interestingly, this concept can be mimicked to backfire malware so as to deter their propagation by eluding malicious processes into thinking that underlying systems should be avoided rather than to be infected. For this, tools and methods have been proposed also [CAM+08, SG13]. Regarding the SIEM process, virtualization technologies must adapt accordingly to provide the big picture of threat intelligence visibility in virtual infrastructures. Up until now, monitoring VMs was done by placing taps at hardware egress points. But, for the first time, VMware and Rivest, Shamir, Adleman (RSA) introduced [Mit13] a monitoring technology that goes inside the VMs and is capable of capturing packets, logs, and events that can be sent to the SIEM platforms. VMware VMotion can isolate a VM in a sandbox for conducting forensics if a threat is detected by a SIEM processor aware of virtual infrastructures. Such is the case of RSA Security Analytics.

On PaaS clouds, development platforms may not provide the required secure isolation for running multi-tenant applications. It is also imperative to correctly dispose of unused objects within each application memory context so as to avoid memory leaks to co-resident tenants, and ensure thread execution in a fail-safe manner. But neither .NET nor Java platforms can fulfill these requisites [RMVC+12]. IaaS clouds normally offer an interface for customers to have total control over their VM infrastructure, depicting functionalities for creating, editing, snapshotting, restoring, or deleting VMs, as well as for configuring firewalls and security policies. This is the case of Amazon EC2. As such, they are exposed to a myriad of security issues the Internet already suffers from, including the web-based accessed that entails these interfaces to be prone to phishing, for example. They comprise attractive attack points because of these functionalities and an attacker with access to a compromised account can do lots of damage to a customer by accessing sensitive data.

For enterprises, both networking and computing perceptions are suffering deep changes. The Bring Your Own Device (BYOD) paradigm agitates further the internal network perimeter. Since the nineties, the openness of networks has gradually increased, opening doors for applications (*e.g.*, email and websites) and for customers, while the perimeter trust kept narrowing down, creating an attack spectrum spanning several vectors [Amo13]. Mobile devices can roam from telecommunications networks to wireless networks easily and can carry malware, and thus they further hammered down such a perimeter trust. The problem is that they can access sensitive backend enterprise applications hosted on (private or hybrid) clouds while surfing on the Internet on potentially unsafe sites. Additionally, the BYOD paradigm creates a problem from both engineering and security point of views. Conventional networks are usually segmented into several VLANs, including the open wireless network and corporate services, namely printer services. Because popular mobile devices are designed for home networks, they cannot discover other services via layer two, since the devices are connected to distinct VLAN segments.

Due to aforementioned issues, cloud customers should thoroughly consider the advantages and

disadvantages of all cloud deployment and service delivery models before deciding on a particular one. Not only that, but the lack of cloud standards also makes it difficult to build interoperable intercloud networks for a seamless cloud experience. In the meantime, the security state of cloud environments is uneasy, and for improving that aspect, each cloud provider should strive to meet body standards under development (*e.g.*, Open Cloud Computing Interface (OCCI) [OCC]) and adhere to open-source projects (*e.g.*, OpenNebula [Ope] and CloudStack [Apa]). Cloud providers should additionally, and perhaps mainly, quantify how risky it is for moving to the cloud. For this task, the NIST released a document [NIS13] with various steps for formally defining a security reference architecture through a risk-based approach. Microsoft also released the Cloud Security Readiness Tool (CSRT) [Mic13] for helping enterprises assess what they could expect if they replaced their IT systems for cloud solutions.

2.4 Randomness in Computer Science

According to the Oxford English Dictionary, *random* is defined as *"without method or conscious decision."* The same dictionary further defines random as *"statistics governed by or involving equal chances for each item."* Concerning science, both sentences suit well the concept in several areas of knowledge, particularly computer science and mathematics. Randomness is a property characterizing some random process and, beyond computer science and mathematics, random processes are also important for several other fields, namely physics and finances. In fact, randomness may be a property whose existence may be questioned, because what *appears* random to an observer might not *appear* random to another. As such, randomness is as much philosophical as physical or mathematical. In the domains of computer science and mathematics, randomness is key for cryptography and information theory, and for statistics and probability, respectively. In the context of the information age we live in today, randomness is ultimately one of the foundations for the security and privacy of Internet communications and of data storage.

2.4.1 Random Number Generators

Generally speaking, the scientific community extensively and widely utilizes RNGs for generating sequences of random numbers when there is the need to access some source of randomness. In cryptography, the need for random numbers is commonly associated with the purposes of generating cryptographic keys, challenges, or one time pads or passwords. In other areas, such as networking, they may be used for creating values in certain protocols (*e.g.*, Transmission Control Protocol (TCP) sequence numbering). In computer-based simulations, sequences of uniformly distributed random numbers are the basis for Monte Carlo methods or for generating values according to some probability distribution. Further discussions on randomness will be carried while emphasizing the requirements of good properties for cryptography. In its true sense, randomness should refer to an unpredictable behavior, being impossible to guess what is to be generated or expected in the future, even for adversaries having unlimited computing re-

sources and full knowledge of past observations and of underlying software and hardware. This is the fundamental criterion for designing RNGs and, at the same time, the toughest challenge when devising RNGs atop computers. For cryptography, one needs random values which cannot be guessed any more easily than by trying all possibilities [Ell], that is, by brute-forcing the entire output space.

There are several ways to generate random sequences, but it is hard to prove that they are fool-proof in most situations. Therefore, it is a matter of art and assumptions to select good sources of randomness as indicated in the Request for Comments (RFC) 4086 [rSC05], or better yet, to select RNGs that are cryptographically strong and whose outputs endure against a number of stringent statistical tests. RNGs can be classified as either *trully*-random or *pseudo*-random, being known as True Random Number Generators (TRNGs) and PRNGs, respectively. The latter are algorithms which imitate the former. All numbers generated from a RNG fall within a probability distribution. There are several probability distributions, but in security terms, each one has distinct properties that makes some more desirable than others. On the perfect case, RNGs generate numbers under the uniform distribution (*e.g.*, over the interval $[0,1]$, i.i.d. $U[0,1]$), in which each number has an equal probability of being generated in an independent manner. These are called *uniform deviates* or simply *uniforms*.

2.4.2 True Random Number Generators

Scientists seem to agree that, nothing, outside quantum physics, is truly random. However, TRNGs are normally constructed by capturing and transforming noisy signals, like thermal noise, using specialized hardware. For all purposes, they are unconditionally considered unguessable and, therefore, are considered secure as well. TRNGs can be devised virtually on any hardware device. The criterion in this case is that of fetching randomness from a source that is not stable and that it shows erratic, or chaotic, behavior, usually referred to as *entropy* collection (entropy is formally defined in sub-subsection 2.4.4.1). This fuss applies well to the RANDOM.ORG [RAN], that produces random bits out of atmospheric noise. By its very nature, weather forecasts may be a pitfall for this generator. RANDOM.ORG is a service available through the Internet to the Internet community. For security purposes, it is not usually regarded as a good source of material, because the results of the service are transmitted through the Internet, tough over the HyperText Transport Protocol Secure (HTTPS). In addition, it may be slow. In fact, producing true random bits efficiently is one of the common challenges of TRNGs. For several years, another online service did the same job of RANDOM.ORG. Now terminated, the service was named Lavarand and took pictures of the blobs churning away inside lava lamps to generate truly random numbers.

As Connolly [Con07] argues, radioactive decay entropy sources are the best ones for using in hardware-based RNGs. Radioactive sources emit a constant stream of electrons that can be read

by a Geiger counter and then transformed to simple electric pulses. Each electron is literally counted, and then each electric pulse is converted to a digital bit. James Clewett [Cle13] showed how to perform this process with Strontium-90, a radioactive material that emits such a stream of electrons. He found out that, for a certain period of time, the number of electrons going into the Geiger counter follows a Gaussian distribution, or normal distribution, whose notation is $\mathcal{N}(\mu, \sigma^2)$, which may then be transformed into the uniform distribution for usage on computer systems.

For common applications and computers, the previous TRNGs may not be appropriate for usage in applications with tight speed requisites. Additionally, mounting such an apparatus as the one of Clewett is not easy and does not scale up well. To address these needs, Intel, the microchip manufacturer, has developed [TC11] what is now called the Intel Secure Key (previously known as Bull Mountain). Intel Secure Key is a digital RNG that marks the transition from analog circuitry-based RNGs, which consume lots of power, to efficient digitized circuitry. Thermal noise is captured to generate raw streams of random bits at three gigabits per second. That stream is remastered by a conditioner to improve randomness, and then the outcome is inputted as a seed to a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) that produces 128-bit secure random numbers. These numbers can be fetched through the `RdRand` CPU instruction, designation by which the technology is better known by the community. `RdRand` is available on CPUs with the recent Ivy Bridge microarchitecture. It is compliant with cryptographic standards such as NIST SP800-90, FIPS 140-2, and ANSI X9.82 [Int12]. This in-built RNG approach can be seamlessly integrated across computers and thus dismisses the need for external protocols. Benchmarking shows [Kin12] highly scalable provisioning and quick throughput generation, while Dieharder (a statistical testing library) tests indicate good randomness quality. Nevertheless, a recent publication [BRPB13] demonstrated how to tamper CPUs with a logic gate by changing the doping of one transistor. This is a sort of sabotage that can be very hard to detect since it does not require changing circuitry. Worst of all is that it is possible to decrease the aforementioned 128-bit random numbers output to 32-bit random numbers, without triggering built-in self-tests and without failing randomness tests.

2.4.3 Pseudo-Random Number Generators

As said earlier, the true meaning of random is somewhat fuzzy. When discussing PRNGs, the meaning gets blurred even further and, sometimes, can be misleading. Some people assume the PRNGs embedded within OSes, programming frameworks, or compilers, to be secure. These, however, should be avoided for cryptographic purposes. PRNGs are no less and no more than algorithmic generators based on some mathematical theory that output random numbers with little overhead, and thus are also known as software-based RNGs. Prime numbers are a key ingredient for the mathematical soup to work well upon modular arithmetic or elliptic curves. Nevertheless, the problem with this type of generators is inherent to the nature of computers

and of mathematical designs. Humans build computers for them to work and behave correctly, as expected, and not in an inconsistent manner. They are designed to process input data, outputting, if all works well, the correct results. This is better known as *determinism*. That is, computers are deterministic machines because it is known beforehand what the results should be for certain input machine instructions, not to mention that programs are devised according to the machine logic. Well, this holds true without counting the Intel Pentium flaw back in the nineties that rendered inaccurate floating-point calculations.

2.4.3.1 Pseudo-Random Number Generator Constructions

Inventing PRNGs has always been one of the greatest struggles between man and machine that lasts for over a handful of decades. The difficulty is in conceiving PRNGs that show good properties with regard to randomness, namely the *period* and *state*. PRNGs are *finite state machines*, and usually require some sort of spark to initiate the algorithm in order to mingle the internal state as much as possible. That spark is known as the *seed*, and it should be large enough to thwart brute force attacks. Perhaps one of the most known PRNGs is the one shipped with most C compilers which is named `rand()`. The `srand()` function takes as input the seed for the generator, and an accepted way for doing this is by providing the current time with `time(NULL)`.

Along the years, a bonanza of generators of this type has been proposed [LS07] in the literature, each depicting stronger or weaker randomness quality and properties. The C PRNG belongs to the Linear Congruential Generators (LCGs) class which is one of the oldest PRNG constructions. LCGs produce random numbers with basis on the expression (2.1):

$$X_{n+1} = X_n \times a + c \mod m,\ 0 \leq \{X_0, a, c\} \leq m \wedge a \neq 0 \wedge m \in \mathbb{N}, \quad (2.1)$$

where X_0 is the seed, a is the multiplier, c is the increment, m is the modulus, and X_{n+1} is the pseudo random value produced after and with basis on X_n. This PRNG construction immediately implies that all sequences generated from the same initial seed are equal, and that the period of the generator is, at most, m. The mathematical properties and relations between a, c and m are important to achieve good LCGs with long periods. In the LCG implemented by the C programming language, these take the values 1103515245, 12345, and 2^{31}, respectively. Nonetheless, numbers generated from LCGs fall mainly within a lattice structure established by George Marsaglia [Mar68] in 1968, and thus they are considered unsafe for cryptographic purposes. Most of the PRNGs also fail some stringent statistical tests [LS07], including the popular Mersenne Twister that has a very long period of $2^{19937} - 1$. Long periods do not guarantee higher quality, but they certainly help, as the shorter ones can lead to repetitions. Other constructions of PRNGs elaborate on Multiple Recursive Generators (MSRs), that follow the recurrence

principle of LCGs, dismissing the increment and adding more multiplier terms, Linear Feedback Shift Register (LFSR) approaches, that provide feedback through a linear function to a bit register, so as to shift the bits, Lagged Fibonacci Generators (LFGs), among others. LFGs are a generalization of the Fibonacci sequence of the form:

$$S_n = S_{n-j} \star z S_{n-k} \mod m,\ 0 < j < k, \quad (2.2)$$

where $\star$ denotes a general binary operation (*e.g.*, addition or multiplication), and S_{n-j} and S_{n-k} are any two previously generated values. Securer PRNGs are based on block ciphers (*e.g.*, Fortuna) or, more generally, on secure random permutation functions.

2.4.3.2 Cryptographically Secure Pseudo-Random Number Generators

PRNGs can be infinitely combined with one another by seeding one with the outputs of the other or even by mixing the internal functions, but that may not result in a linear increase of the security. Carelessness in this field can result in bad PRNG constructions and, as such, designing these algorithms should be done with care and resorting to a solid theoretical substrate. This is perhaps the main reason for the number of publications in this area which are not abundant in the past few years, but that received nonetheless a fair amount of contributions along the years.

As apposed to weaker PRNGs like `rand()` from the C programming language, an adversary knowing the algorithm of a CSPRNG, along with k bits of a pseudo-random sequence cannot predict, in polynomial time, the $k+1$th bit with probability higher than 0.5. CSPRNGs should also withstand in the revelation of the internal state, hence providing *forward security*, and resist against the prediction of future states in the case of some entropy input is compromised. As with any argument on computational complexity, these proofs are based on assumptions, like the one of the determination of whether all NP-problems are actually P-problems that can be solved in polynomial time, which is not yet solved and divides opinions [Woe13]. Examples of CSPRNGs include the Windows `CryptGenRandom` RNG, the Yarrow [KSF99] algorithm, which is implemented in Mac OS X (cryptographer Bruce Schneier is one of the authors), and Fortuna. Moreover, PRNGs should always be judged with a slight abuse of language when regarding the true meaning of randomness. Groundbreaking advances on cryptography, and subsequently on the construction of RNGs, are natively dependent on the mathematical breakthroughs which only come once in a while.

2.4.3.3 Incidents with Pseudo-Random Number Generators

It is important for this area of knowledge to keep evolving because, from time to time, a discovery may cripple previously used algorithms. Predicting supposedly random numbers to break encryption is more than mere speculation, and may lead to compromising the entire security of a protocol and eventually lead to loss of money or intellectual property. A good example is the case of the Netscape browser. In 1996, two researchers discovered [GW96] a problem with the PRNG Netscape used to create encryption keys for Secure Sockets Layer (SSL) connections. Basically, an adversary could find out possible *seeds* for the PRNG through the time in seconds since the Unix Epoch and the process identifier of the browser running on a victim computer. At that time, this information was trivial to guess. In a matter of minutes, the encryption key for a certain SSL session could be broken. Something similar was possible during the time frame between 2006 to 2008 on Debian-based OSes. Code lines were commented out to avoid warnings of other tools. However, those specific lines restricted the OpenSSL PRNG to be seeded only from the current process identifier, which is only 15 bits long, defining a (small) space of a 2^{15} possible seeds.

More recently, a flaw [Duc12] in the Java-based PRNG used by the Android mobile OS rendered all Bitcoin wallets vulnerable to theft, for the second time. Bitcoin is a growing digital cryptocurrency dismissing the need for a central authority overseeing transactions by adopting the P2P model. Bitcoin resorts to public-key cryptography to sign transactions, particularly by using the Data Signature Algorithm (DSA) over elliptic curves, which requires generating a random number for each signature. However, the mathematics behind the algorithm state that, by gaining access to two messages signed with the same private key and the same random number, it is possible to go backwards and extract the private key. This was the case, and the guilty party was the `SecureRandom` Java class that occasionally repeats pseudo-random sequences on Android devices, being thus possible to find collisions. Furthermore, some Zero-Knowledge Protocols (ZKPs) and other authentication schemes, for example, rely on the freshness of random material to create unique, random nonces. If the same nonce is used multiple times, the security of the methods drops. These incidents and flaws can be seen as alerts, as a heads-up, to keep amending prior mistakes or reinforce current methods. The security state regarding RNGs should always be prioritized by carefully implementing algorithms and choosing good sources of randomness.

2.4.4 The Linux Random Number Generator

The Linux RNG, first written in 1994, is implemented on the `random.c` file that is under the `linux/drivers/char/` kernel source code tree path. Because it is embedded in most Linux distributions, it can be said it is one of the most widely used generators, since Linux is extensively used throughout the academia and the industry for its open-source approach. Several

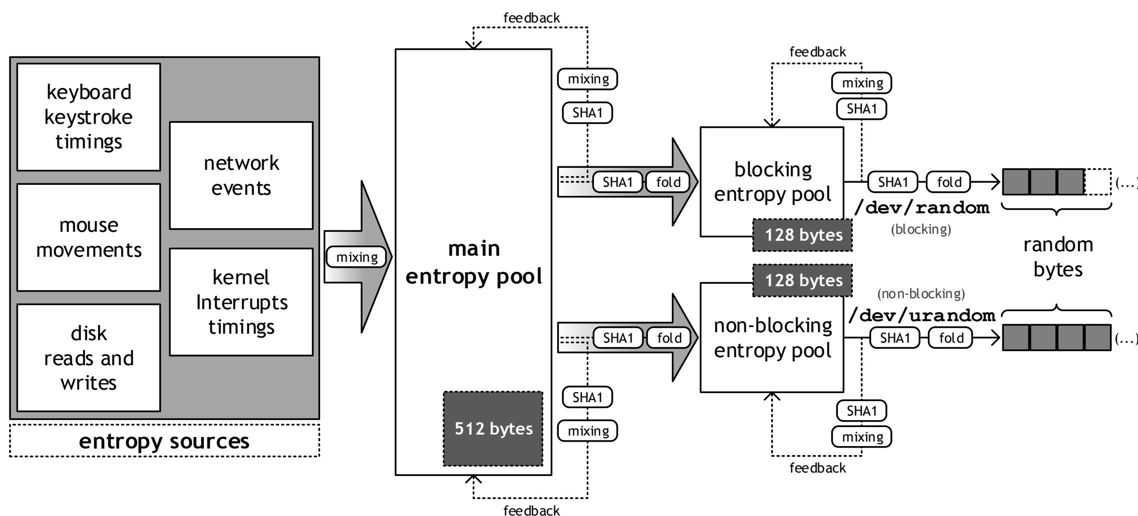

Figure 2.2: Schematics of the construction of the Linux RNG as perceived in [GPR06, LRSV12].

years later, in 2006, Gutterman *et al.* [GPR06] have undertaken the work necessary to reverse engineer and comprehend it to the point of finding security weaknesses on the *forward secrecy* property. An attack with a complexity of $O(2^{64})$ operations was then devised to break it. This study has since served as a baseline for the research community to understand the generator. In 2012, Lacharme *et al.* [LRSV12] revisited the Linux RNG, describing the changes that in the meantime occurred and the implications to the generator. Some changes aimed at patching security caveats, namely the attack devised by Gutterman *et al.*, whose complexity increased to $O(2^{160})$. Since cryptographic material lays its strength on the quality of random numbers, in this dissertation, the Linux RNG is the subject of analysis to assess that quality. Essentially, the Linux RNG fetches randomness from erratic events taking place within the core of the Linux kernel that, in turn, come from the underlying hardware. Such events are regarded as entropic inputs because they stir an entropy pool and, therefore, add randomness in the form of bits. This main entropy pool is responsible for filling two other pools to which the `/dev/random` and `/dev/urandom` devices are tied to. These devices produce random blobs of data, each with its own particularities. This construction design of the devices is based on cascading (*cascade construction*) in the sense that an entropy pool filled with entropic inputs provides randomness to the RNG kernel module that, in turn, return random material via the `/dev/random` and `/dev/urandom` devices. While the former merely transforms and smooths the material provided by the entropy pool, the latter seeds a PRNG and produces arbitrarily long sequences of random numbers. As such, `/dev/random` blocks when there is no entropy available, while `/dev/urandom` does not. In the remaining part of this dissertation, the expression Linux RNG is used to typically refer to the procedures involved in entropy gathering and output transformation via the `/dev/random` device. What was previously described can be seen in the schematics of Figure 2.2, where the behavior of Linux RNG is illustrated. Further details of the Linux RNG are next handed out.

2.4.4.1 Entropy

When applied to information theory, entropy refers to chaos, disorder, or uncertainty associated to a given random variable. Its definition within this area of knowledge is due to Shannon, who introduced it in his 1948 seminal paper [Sha48]. Basically, the higher the entropy, the higher the uncertainty associated to the random variable, and vice-versa. On the other hand, information is not entropy nor uncertainty [Sch11], in the sense it carries data of meaningful material. Shannon defined the entropy of a random variable x as the function $H(x)$ given by

$$H(x) = -\sum_{i=1}^{n} p(x_i) \ln p(x_i), \quad (2.3)$$

where n is the number of possible occurrences of x, and $p(x_i)$ is the probability of the particular occurrence of $x_i \in \Omega$. For a finite number $n \in \mathbb{N}$, the maximum entropy for a particular random variable is given by

$$H(x) = \ln n, \quad (2.4)$$

The amount of entropy is easily calculated in a finite set of observations (*e.g.*, a window of network packets lengths for classifying network traffic [GIP$^+$13]), but that may not hold true for a continuous stream of bits. This is the case of the entropy pools of the Linux RNG. As such, one estimates the amount of entropy via some compression algorithm[3] or statistical tests [Ell]. The Linux RNG estimates entropy with basis on the timing information of the monitored events, which is included in three 32-bit variables: (i) `num`, which is specific to the type of event, (ii) the `jiffies` count that corresponds to the counter of timer interrupts since the last kernel boot, and (iii) the CPU `cycle` count. Because these values provide less than 32 bits of entropy [GPR06], the RNG tries to estimate the entropy in a pessimistic way so as not to overestimate the amount it collects. This is important to avoid `/dev/random` output predictable data. The main entropy pool is 512 bytes long, while the other two are of 128 bytes each. These pools request entropy from the main entropy pool whenever necessary and by respecting certain conditions [LRSV12]. Transferring bits from one pool to another involves hashing the extracted bits with the Secure Hash Algorithm-1 (SHA-1), modifying the pool state, decrement the entropy estimation, and add a new amount of entropy to the target pool. Modifying the state of any of the pools is done so in an LFSR manner by using a mixing function. The mixing function is specifically designed not to lose any entropy already within the pools, and thus diffuse accordingly the entropy that is newly added or fed back through a linear shift register. After the pool is updated, its contents are hashed again with SHA-1 and its output is folded in

[3]Lossless compression algorithms are inherently related with the information theory of Shannon [Sha48] for discovering the minimum number of bits, or the entropy limit, required to encode pieces of information.

half to compute ten bytes to be extracted.

Entropic inputs are gathered by the kernel from a number of sources. Such sources include the user interaction (such as keystroke timings and mouse movements), disk head seeks timings, and interrupt timings. Since kernel version 3.2.0, the Intel Secure Key technology was added to the Linux RNG, according to the Lacharme *et al.* [LRSV12]. This will be an important aspect to consider later in the tests undertaken in this dissertation. Disk accesses can be predictable while mouse-based entropy may be downgraded in networked applications such as web browsers. A mouse sequence and location is published in the traffic generated between a client and a server, knowing which button or interface element was pressed. In this case, the only private noise left is the position of the mouse cursor within such element, rendering only a few bits of variability per event [Dav96]. Because some mouse device drivers and OSes can come with a snap-to capability that instantly points to the center of popped up dialog boxes, this caveat can be further carved [Gal97]. Despite these drawbacks, the Linux RNG is well regarded and is mostly considered secure, mainly the `/dev/random` device.

2.4.4.2 Devices

The random material generated by the Linux kernel is accessible through the `/dev/random` and `/dev/urandom` devices. These devices can be read as any other file. For example, on the terminal it is possible to read four bytes from any of the devices with the following command: `$ head -c 4 /dev/random | openssl enc -base64`.
The first command is instructed to read the leading four bytes of whatever is printed to the standard output, which is then piped to the second command that encodes them into Base64 using OpenSSL. The devices `/dev/random` and `/dev/urandom` differ in two aspects. The first is the level of security of the outputs in terms of true randomness, while the second is the efficiency. The device mentioned first yields extremely secure bits, while the second produces less secure bits because of its embedded PRNG state. The `/dev/random` device is very slow because it is only available when enough entropy is gathered; if the pool is depleted, it blocks and the process reading it hangs. In contrast, `/dev/urandom` never blocks because the PRNG can keep generating pseudo-random numbers indefinitely. This device is also attainable through a direct kernel interface: the `get_random_bytes()` function. As such, any process can use it easily and without concerns regarding speed, while `/dev/random` is reserved for user-space applications [LRSV12].

As said earlier, the main entropy pool of the Linux RNG distributes its entropy bits to anoter two, smaller pools. Because of the above properties, they can be called the *blocking pool* and the *non-blocking pool* with respect to `/dev/random` and `/dev/urandom`, respectively. Each of those smaller pools also has a counter for entropy estimation, independent of the one for the main

pool. Because of the PRNG state, `/dev/urandom` is considered inappropriate for cryptographic purposes by the skeptical or the most paranoid.

2.5 Randomness in Virtual Machines

It was previously said that virtualization abstracts the guests managed by the hypervisor from the underlying hardware devices. VMs encapsulate the OSes and applications in a manner they cannot access the physical hardware consistently, but are rather mediated through virtual hardware devices created by the hypervisor. In turn, the hypervisor schedules appropriately each VMs for CPU cycles, access to the physical network in the case of bridged networking, and others. For the particular case of the Linux RNG, a problem starts to take up form concerning the provisioning of entropic inputs when the Linux kernel runs within a VM. Whereas cloud instances are deprived of real hardware, it is possible to say that the kernel may not behave normally in terms of events, as it would over host circumstances, in which case access to hardware would be provided in full. In practical terms, what this means is that the collection of entropic events may be hindered by the separation of the guests plane from the physical hardware in two ways. First, the set of events monitored by the kernel may not be as heterogeneous for guests as they would normally be for host OSes. Second, their occurrence may be less frequent. In other words, guests can be deprived of an otherwise more diverse set of entropic inputs due to the lesser number of distinct available events. This can result in low levels on the main entropy pool, in turn propagating the scarceness mainly to the `/dev/random` device, for it always requires fresh entropy to generate random blobs of data. As such, `/dev/random` may output weaker random material on a slower basis, when compared to hosts.

The problem enlightened above can be more complex than it may seen at a first glance if the scenario is extrapolated from a single VM to IaaS clouds. IaaS clouds can blend great amounts of resources elastically and seamlessly, while supplying them to a great amount of VMs running on top. This means that there is a high likelihood of two or more VMs being placed over the same physical server, running co-resident with each other. As such, it is entirely possible to consider correlated cryptographic material independently generated on co-resident guests, since the underlying hardware is the same. Furthermore, snapshotting is extremely useful for backing up the state of an OS on-the-fly with little overhead by simply copying the image files. However, saving the entire state of OS means that entropy pools and PRNG states are also saved, and thus restoring the snapshot sometime later also reinstates those elements. By using the same snapshot repeatedly, the outputs of the Linux RNG on every run may be correlated as well. Because of this, Ristenpart and Yilek [RY10] were able to compromise Transport Layer Security (TLS) sessions and extract DSA ephemeral keys[4] from servers. They implemented *hedged cryptography* to protect against this type of repeated randomness failure.

[4]In the cryptography realm, cryptographic keys are sometimes called ephemeral keys when they are generated and used for a single signature.

Yilek [Yil10] has proven the security of cryptographic protocols is called into question when under VM revert and reset conditions due the assumed (incorrect) freshness of random numbers. Heninger *et al.* [HDWH12] further found alarming problems regarding the entropy scarceness and bad randomness issues. They scanned 12,828,613 TLS and 10,216,363 Secure Shell (SSH) hosts on the Internet so as to save their certificates and public keys. Due to entropy problems, 0.75% of TLS certificates shared keys, while the public keys of 0.50% of TLS and 0.03% SSH hosts were factorized, therein obtaining the respective private keys. They also obtained the DSA private keys of 1.03% of SSH hosts. Moreover, Kerrigan and Chen [KC12] discovered that the `cycle` count (part of the Linux kernel events timings) to be highly correlated among Xen guests. This work corroborates the problem so far outlined. In 2009, Stamos *et al.* [SBW09] added an interesting detail at the Black Hat conference. They argued that some VMs are often fired up for short periods of time to serve specific purposes, and thus there might not be a large enough time-window to develop a sufficiently unique entropy pool. In turn, the Linux RNG can produce weaker random material. Based on that, a theoretical attack on Amazon EC2 instances was proposed by brute-forcing SSH keys with PRNG seeds taken from other VMs.

Most popular cloud solutions, namely VMware vSphere and VirtualBox, have no documentation regarding entropy gathering or random number generation, except for KVM, which has which has the `HW_RANDOM_VIRTIO` driver [KVM] that allows to wire external randomness sources through the hypervisor and up to guests. Nevertheless, several discussions on Internet forums and blogs cover the topic of the depletion of entropy pools and consequent starvation of `/dev/random`. Amazon EC2 is included [Ama11], and thus Xen falls to the pit along with VMware and VirtualBox [VMw06, Pat10], and another undisclosed hypervisor [Car12]. On OpenStack, storage blocks are virtualized and their reads and writes are heavily cached by the hypervisor [Kir12]. That, combined with state-of-the-art Solid-State Drives (SSDs), diskless devices, and servers devoid of arbitrary human interaction via mouse and keyboard, may limit guests to network events only [SBW09, KC12]. This might considerably affect how entropy is gathered on the Linux RNG and, consequently, the quality of `/dev/random` outputs. For all the reasons discussed in this section, the most suitable device for study with regard to randomness on VMs is `/dev/random`. Its full reliance on fresh entropy bits to output quality random data is the main reason for the work presented in this dissertation.

2.6 Conclusions

This chapter introduced various concepts related with the cloud computing model and the randomness property. The concepts associated with cloud computing were first explained in order to better contextualize the remaining work of this dissertation. Such explanations were carried out while underlining the security state of this new computing model because, besides being largely recognized for its benefits, it is also widely known for the puzzling security issues it poses. As such, the research on this field is booming, mainly aiming at patching the known

problems in order to achieve securer cloud environments throughout the cloud deployment models. This endeavor is being pushed by both the academia and the industry. This is a good sign for upcoming cloud developments so as to experience cloud subscriptions in a fail-safe manner while enjoying the appealing features cloud computing yields on all service delivery models. It was said that trust, legality, data storage, openness to the outside, surveillance, the BYOD paradigm, and standardization are yet issues to be surpassed. Even though virtualization is perhaps the main element allowing cloud computing to prosper, it is also the source for a new emerging class of unexplored security issues.

The last part of the chapter is dedicated to explaining the problem of generating randomness on VMs, by first focusing on the differences between true and pseudo RNGs and then by defining entropy and the way the Linux kernel typically generates random values. It is said that, in virtualized environments, the Linux kernel may suffer from entropy starvation, mostly due to the abstraction layer that separates guests from the underlying hardware. The problem requires to be studied from two different perspectives. On the one hand, it is necessary to quantify and better study the generation speed of the Linux RNG on VMs; on the other, it is necessary to assess if the material generated on VMs has the same quality of the one generated on the bare metal and, more importantly, if they do not exhibit any correlations due to being run on top of the same hardware. Assuming that cryptographic algorithms are well implemented and used, the source of randomness is the one dictating the strength of the cryptographic material. This masters program was then focused on analyzing the Linux RNG according to the method described in the next chapter.

Chapter 3

Method for Collecting and Analyzing /dev/random Outputs

This chapter details the method that was used to collect outputs of the Linux RNG and to execute several tests to its quality and efficiency on cloud computing scenarios. The description of the analysis of the results is also included herein.

3.1 Introduction

To tackle the problem described at the end of the previous chapter, a method was devised and adopted to execute several tests on distinct cloud computing scenarios. An overview of the method is included in Section 3.2. This method includes describing how the collection of data was performed and how it will be analyzed while having in mind the underlying virtualization layer. The collection of data, described in Section 3.4, focuses on sampling the /dev/random device of the Linux RNG. The analysis of the data is done with two purposes. The first purpose is that of analyzing the throughput efficiency of /dev/random outputs, for which part of the method taken to achieve this is described in Section 4.4. The second purpose is to analyze the quality of random numbers outputted by /dev/random. The description of this part of the method is in Section 4.6, while the explanation of the testing library used for that objective is in Section 3.3. The execution of the method led to the development of various scripts in Python 3 (some of the scripts might not work in Python 2) and of other programs in ANSI C. Python is an easy-to-use dynamic and modular language oriented for both functional and object programming. It is a robust replacement to shell programming for developing quick scripts on-the-fly. For simplification and organization purposes, a custom-made library with useful and common functions was developed for each of the programming frameworks. Each script has its own purpose and, whenever convenient, they are described or referenced in a proper place throughout this chapter.

3.2 Method Overview

The general workflow of the method adopted in this dissertation for studying /dev/random is graphically represented in Figure 3.1. The sampling phase consists in collecting random bytes outputted by the /dev/random device under various IaaS cloud setups. During this phase, four-byte chunks are read at a time from /dev/random, which are stored on the memory using a 32-bit unsigned integer type, rendering numbers in the range $[0, 2^{32} - 1]$. Since most RNGs

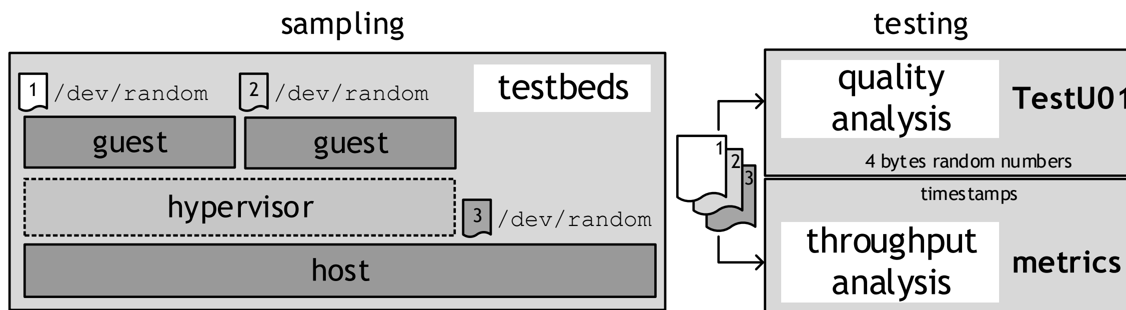

Figure 3.1: Illustration of the method adopted to study the `/dev/random` on guests and hosts.

have an accuracy of 32 bits (31 bits of accuracy is also frequent), it made sense to follow this approach. Some statistical tests are also designed for that accuracy, like the DIEHARD [Mar] statistical testing suite, developed by George Marsaglia, that requires exactly 32-bit integers. Additionally, the timestamp of the read operation is also collected. While the timestamp is mostly useful for the throughput analysis, the random numbers are destined for the statistical analysis of their quality. All values collected in the meantime are saved into sample files that are then post-processed for separating and preparing the values for the respective analysis in the testing phase. For assessing the quality of random numbers, the TestU01 statistical library was used. Figure 3.1 schematizes the use of an hosted hypervisor and two guests running on top. Nevertheless, that is merely representative, because the tests undertaken during the masters program involved more testbeds. These testbeds are described in Section 4.2.

3.3 TestU01

TestU01 is a comprehensive statistical testing library developed by L'Ecuyer and Simard at the Université de Montréal. It was presented to the community in 2007 [LS07] and it is available for download at [Sima]. The download package includes a long guide describing the library. That guide was on the basis of the summary provided herein. The library is written in ANSI C and provides an extensive API. TestU01 is a very stringent library for assessing the quality of RNG because it analyzes thoroughly a generator by submitting a huge amount of random numbers to several stringent statistical tests. These statistical tests try to find statistical weaknesses, each in its own way, but with a common goal. The purpose of the library is to empirically test if the null hypothesis $\mathcal{H}_0$ holds for a given RNG. $\mathcal{H}_0$ states that, for each integer $t > 0$, the vector $(u_0, \ldots, u_{t-1})$ of t successive outputs from a RNG is uniformly distributed over the t-dimensional hypercube $(0, 1)^t$. This means that vector should imitate independent random variables from the uniform distribution over the interval $[0, 1]$ (i.i.d. $U[0, 1]$), or over the two-element set $\{0, 1\}$ in case of binary testing. The hypothesis $\mathcal{H}_0$ denotes the perfect behavior of a random sequence.

The library provides an extensive list of incorporated PRNGs (*e.g.*, Mersenne Twister) and well-know statistical tests. Beyond that, perhaps the best thing of TestU01 for automated analysis

are the predefined stringent batteries of tests. These batteries request huge amounts of random numbers from a RNG and may need to run for several hours on modern desktop computers until they output the verdict. The library defines these batteries by levels of stringency. The less stringent one is named *SmallCrush* and it should run fast as it only executes ten tests. *Crush*, the intermediary battery, may run for periods of time spanning from several minutes to a few hours (depending on the CPU power) and uses approximately 2^{35} random numbers throughout 96 tests. The most stringent battery is called *BigCrush* and can take several hours to complete. It uses close to 2^{38} random numbers spread throughout 106 tests. For convenience, the library includes other less-stringent libraries and tests, namely DIEHARD and FIPS 140-2 (statistical tests defined by the NIST), respectively.

Each statistical test outputs one or more p-values that give an idea of how close or how far the results are to the optimal case, the null hypothesis. The p-value is like a measure of uniformity. A p-value close to one means excess of uniformity, while a p-value close to zero means the opposite. These values are the result of Goodness-of-Fit (GOF) tests, namely the chi-square, denoted by χ^2, the Kolmogorov-Smirnov, denoted by D, and the Anderson-Darling, denoted by A, which compare theoretical distributions with empirical distributions estimated from the input. In TestU01, the confidence interval for the p-values is $[10^{-10},\ 1-10^{-10}]$. Tests fail decisively for p-values outside that interval. This means that if a certain statistic outputs a p-value of say, 10^{-11}, then the input sequence fails that particular test. Systematic clear failures throughout several tests are indicative for a bad random sequence. If a sequence simply fails one test, it may be out of pure chance (bad luck). Increasing the sample size may outwit such cases. The bounds of the confidence interval are somewhat subjective to define, but the one mentioned serves the purpose of testing the Linux RNG. Naturally, smaller intervals (*e.g.*, $[10^{-8},\ 1-10^{-8}]$) will make the library more sensitive to high or low p-values, and vice-versa.

In this dissertation, TestU01 is used for assessing the quality of the outputs of `/dev/random`. It was chosen for being the most stringent library on the field and for its flexibility. Unlike other libraries, namely Dieharder [Bro] (an improved version of DIEHARD), TestU01 allows to use any of the already implemented PRNGs or implement new PRNGs complying with the API. It also allows using external RNGs or reading files of datasets and supports the modification of each one of the tests via adjustment of a few input parameters. These facts are important for the work described in this dissertation, because it was required to read previously saved external files containining random number and adapt the tests for smaller datasets. This is further explained in subsection 3.6.2. The method adopted to save sample files felt like the best approach because it would be impractical to wire up `/dev/random` directly to TestU01 on various VMs and hosts at the same time and analyze the results conveniently afterward.

3.3.1 Installation, Building and Running

TestU01 version 1.2.3 is available in two main flavors: as pre-compiled binaries for Windows OSes or as source code. During this masters program, TestU01 was used in Linux OSes. The library uses configure for standard installation on Linux-based OSes. After downloading the source code, the following commands are enough for carrying out the installation seamlessly:

`$ cd TestU01-1.2.3`,

`$ ./configure -prefix=$path`,

`$ make; make install`,

where `$path` represents the installation path of the library. This path is required for setting some environment variables in order for the C compiler to find where TestU01 is installed when building programs using the library. The environment variables can be set up with:

`$ LD_LIBRARY_PATH=$path/lib;LIBRARY_PATH=$path/lib;C_INCLUDE_PATH=$path/include`,

`$ export LD_LIBRARY_PATH; export LIBRARY_PATH; export C_INCLUDE_PATH`.

For building a C program using the library, it is required to specify some flags to the compiler. Any standard C compiler should do the job. During this research work, the popular GNU Compiler Collection (GCC) was used for that purpose. For a C source file named `program.c`, the following command should compile it through:

`$ gcc -Wall -o output program.c -ltestu01 -lprobdist -lmylib -lm`.

Afterward, running the binary is as simple as:

`$ ./output`.

3.3.2 Example in ANSI C

A simple C program exemplifying how the API of TestU01 can be used is presented in this subsection. Listing 3.1 contains the C code for what is, in fact, a minimal working excerpt of the main program developed in the scope of this dissertation, detailed in subsection 3.6.3. TestU01 provides several header files specific to RNGs, statistics, or other miscellaneous functions. In this case, the headers required are simply `unif01.h` and `bbattery.h`, respectively for using basic functions associated with the definition and manipulation of generators, and for utilizing the predefined test batteries. `unif01_Gen` is a structure representing an arbitrary RNG. Every RNG intrinsic to TestU01 is of that type. In the listing, the generator being created by means of `unif01_CreateExternGen01` must be a generator whose outputs must be within the range $[0,\ 1)$. The function takes as input a character array for the name of the generator and a handler for a function implementing the generator. Such function is represented by `rng` and can be a PRNG that generates random numbers in real-time or can be a function reading from a file or other input source. `bbattery_SmallCrush` applies the *SmallCrush* battery to that particular generator. This means that TestU01 will call upon the `rng` function each time a random number is required for all the statistical tests *SmallCrush* implements. In the end, the generator can be

disposed of using `unif01_DeleteExternGen01`.

Listing 3.1: Example of a TestU01 C program using some functions of the TestU01 library.

```c
#include <stdio.h>
#include <stdlib.h>
#include "unif01.h"
#include "bbattery.h"

double rng (void) {
        /*
         * rng can be a PRNG or a function reading from a file
         * must return a double within the output range [0, 1)
         */
         return 0.0;
}

int main (void) {
        unif01_Gen *gen = unif01_CreateExternGen01("example", rng);
        printf (" name: %s\n", gen->name);
        bbattery_SmallCrush (gen);
        unif01_DeleteExternGen01 (gen);
        return EXIT_SUCCESS;
}
```

3.4 Collecting `/dev/random` Outputs

For collecting the outputs of `/dev/random` and the timestamps of each read on the device, a script in Python was written. When the script is run, it indefinitely collects data until it is purposely and explicitly stopped with a `SIGINT` or `SIGTERM` signal or when the maximum number of samples specified by an input argument has been reached. While these conditions are not met, four random bytes are read from `/dev/random` at a time. If `/dev/random` is waiting for entropy refills, the reading operation hangs until the device makes available at least four bytes of random data, henceforth called samples. Right after, the timestamp is retrieved and both values are written to a sample file. Throughout this dissertation, a timestamp refers to the amount of seconds that have passed since the Unix Epoch. The precision of the timings used is therefore a second. This seemed enough for the purpose of showing the throughput efficiency of `/dev/random` when running on VMs. Assuming the `time` module is imported, this timestamp is obtained via `int(time.mktime(time.gmtime()))`. Let `file` be an object representing an I/O stream with a file, obtained from `open ("/dev/random", 'rb')`, then reading four bytes from the stream is achieved with `bytes = file.read(4)`. In Python, converting format types is

called *unpacking*, and such feature is in the `struct` module. This is the case of converting raw bytes into an unsigned integer, herein attained with `uint = struct.unpack ("I", bytes)[0]`.

3.4.1 Format of the Samples Files

The number of lines of the sample file resultant from the execution of the script is the same as the number of samples collected. Samples files have two columns, and each line represents a read operation on `/dev/random`. The first column contains the timestamps while the second column packs the random bytes in unsigned integer form. Listing 3.2 contains a ten-line snippet of the dataset number 1 collected from one of the tests which is characterized in Table 4.1. As perceivable from the timestamps, the snippet includes samples from a period of two seconds, represented by the timestamps 1373705412 and 1373705413, spanning through the lines number 991 to 1000.

Listing 3.2: Example of a sample file resultant from the execution of the Python sampling script.

```
991  1373705412 3279000729
992  1373705412 1561260504
993  1373705412 2012794694
994  1373705412 2963770328
995  1373705412 2622604791
996  1373705412 3109778389
997  1373705412 3984215824
998  1373705413 206987852
999  1373705413 1457621366
1000 1373705413 3536428877
```

3.4.2 Post-Processing the Sample Files

Post-processing the sample files for each of the analysis means to segregate each of the columns into separate files for analyzing them singularly. This could have been done within the script while collecting data, but that would have not been the best approach since it would only cause unnecessary and unwanted overhead. The command line `cut` utility can do this by:

`$ cut -d ' ' -f field input > output`,

where `field` is an integer list delimited by commas specifying the wanted field numbers, `input` is the input sample file, and is `output` the resulting file with the columns selected.

3.5 Analysis of `/dev/random` Throughput Efficiency

One of the objectives of this work was to compare the efficiency, in terms of throughput, of the `/dev/random` when running in host and over VMs. This was done by using the timestamps collected in the sample files and by calculating a set of metrics that characterizes each of the sample files obtained from VMs and hosts. In scenarios with co-resident guests or one guest

over the host, the analysis also aims at searching for potential correlations in the timestamps. Because the underlying hardware is shared due to virtualization, it is possible for some of the samples to be read at some equal instant in time on co-resident guests and on the host. There may be the case of simple coincidence as well. But if not, the analysis should reveal a systematic behavior, otherwise those false positives should be easily detected.

3.5.1 Python Script

Again, a Python script was written to find the characteristics of individual sample files and of new sample files (mergers) created from tests which involved a VM and the host, two VMs, or several datasets taken from various tests (this is further explained in section 4.6). The script takes as input one or two files. It creates a dictionary per file with respect to the timestamps and counts the amount of times a certain timestamp is repeated. In programmatic terms, the timestamps are referred to as the keys while the counters are the values of the dictionaries. Basically, the dictionary represents a histogram for the timestamps. Analyzing it gives an idea of how often `/dev/random` unblocks for generating random numbers to user-space applications like the script, and how many bytes it outputs in each second. The script also provides information about how spread out or how condensed reads on `/dev/random` are. For two files, the script further calculates how many timestamps they have in common.

3.6 Analysis of `/dev/random` Outputs Quality

Testing the quality of random numbers is not straightforward. As previously said, what may look random to an observer might not appear to another. It is a matter of perspective. Even though testing the quality of random numbers by means of stringent statistical tests cannot completely demonstrate if an RNG construction is foolproof nor if is suitable for all kinds of computer based simulation, they can certainly help to identify statistical weaknesses and strengths. As such, statistical tests may improve or diminish the confidence one has in certain RNGs. But then one can rightly argue that it is impossible for an RNG to pass every conceivable statistical test. As such, it is said that bad RNGs fail simple tests whereas good RNGs fail only the complicated ones [LS07].

3.6.1 Overview

The main purpose of this analysis is to look for weaknesses in randomness due to the entropy collection problems. It also aims at searching for potential correlations between random numbers independently generated on guests and hosts or on co-resident guests. A first task simply submits the files, from all the tests with the random numbers resulting from the post-processing, to TestU01. A second task merges these files and submits them to TestU01 afterward, so as to test the aforementioned correlation. Either way, both analysis assess the quality of the outputs of `/dev/random`, more precisely the quality of the randomness property associated with the several sequences of random numbers.

3.6.2 TestU01 Statistical Tests

As the next chapter discusses, the Linux RNG is reasonably crippled in terms of throughput on VMs. It outputs random numbers on a very slow basis, when compared to hosts. This encompassed an obstacle for analyzing the sample files with TestU01. The *SmallCrush* battery requires close to 2^{28} random numbers, while *Crush* and *BigCrush* call for over 2^{35} random numbers. However, reaching such an amount of random numbers in feasible time proved to be nearly impossible in the time frame of the masters program when the Linux RNG is run within VMs. Hence, the predefined batteries of TestU01 were not used in the analysis of the quality of random numbers. Instead, a custom-made test battery was developed. Fortunately, TestU01 allows to singularly use each of the statistics it implements. Better, it permits configuring them according to some input parameters. The combination of these parameters influences greatly how the statistical tests behave, allowing to tune up or down the stringency of the tests. A common parameter among most statistics enables defining the number of random numbers requested by statistic. Another parameter allows to narrow down or widen the space to which random numbers are thrown to, making it smaller or bigger. In other words, the hypercube $(0, 1)^t$ is cut down into smaller hypercubes for any dimension t, and it is possible to adjust the total number of those smaller hypercubes, in some of the statistics, so as to accordingly preserve stringency.

As such, a customized battery was put in place by adjusting each of the statistics *SmallCrush* uses. These are *BirthdaySpacings*, *Collision*, *Gap*, *SimPoker*, *CouponCollector*, *MaxOft*, *WeightDistrib*, *MatrixRank*, *HammingIndep*, and *RandomWalk1*. While *SmallCrush* applies its statistical tests on one unbroken stream of successive numbers, the customized battery rewinds the file to the beginning before each test. This means that every statistic will test for the same sequence of random numbers. TestU01 provides the `bbattery_SmallCrushFile` interface for doing exactly this, but it does not allow to manually configure each statistic. It applies *SmallCrush* with the difference that random numbers are read from a file.

Although the statistics execute on less numbers, the tests remain valid in this case, because the subsequences of random sequences should be as random as the original one or, at least, they should not fail the duly customized tests that the full sequence does not fail either. This is the same as saying that any pre-specified sequence of independent bits should be composed of smaller sequences of independent bits as well. RNGs are normally expected to produce values that seem occurrences of independent points following the uniform distribution. As such, weakness pointers are given if the tests fail on such subsequences, meaning that the numbers are not uniformly distributed or independent. To make it fair, some of the parameters in common among the statistics were maintained across all the tests. Such parameters include r, s, and t, as depicted in Table 4.3 and as described in subsection 4.6.1.

3.6.2.1 The Case of the *BirthdaySpacings* Statistical Test

The *BirthdaySpacings* statistical test allows to configure the input parameters so as to reduce the testing space. In TestU01, this test is defined by `smarsa_BirthdaySpacings(gen, NULL,` N, n, r, d, t, p), where `gen` is the generator and N is the replication number for the test (typically is one). The *BirthdaySpacings* test throws n points into $k = d^t$ cells. For each integer $t > 0$, the vector ($y_0, \ldots, y_{t-1}$) of t successive uniforms corresponds to a point in {0, . . . , $d-1$}, while p sets the polynomial to use for calculating the cell number for a certain point. This test then sorts in increasing order each cell and counts the number Y of the differences between each cell. Under $\mathcal{H}_0$, Y approximates a Poisson distribution with mean $\lambda = n^3/4k$. At most, n cells are created because the generated points are *converted* into cells. Moreover, the test requires $n \times t$ four-byte random numbers. The relevant restrictions regarding the combination of these input parameters are $k \leq$ `smarsa_Maxk`, and $8N\lambda \leq k^{1/4}$ or $4n \leq k^{5/12}/N^{1/3}$, where `smarsa_Maxk` is an upper-bound constant not relevant for the tests undertaken since k was pursued to be a small value, rather than a very large one.

The aim is to tell the statistical tests to ask for less numbers than they would normally do with the standard configurations included in the TestU01 predefined batteries. In this particular case of the *BirthdaySpacings* test, such is achievable by reducing the aforementioned restrictions for finding the optimal values of the parameters. For $t = 2$ and a given number of samples sam_{size} available, n is a known parameter of the form $n = sam_{size}/t$, which enables one to obtain the minimum value possible. In the restrictions depicted above, the reduction of the inequality should be in order to k, since it is the missing parameter. Since $N = 1$ and $\lambda = n^3/4k$,

$$\begin{aligned} 8N\lambda \leq k^{1/4} &\iff k^{1/4} \geq (8n^3)/(4k) \\ &\iff k^{5/4} \geq 2n^3 \\ &\iff \sqrt[4]{k^5} \geq 2n^3 \\ &\iff k^5 \geq 2^4 n^{12} \\ &\iff k \geq 2^{4/5} n^{12/5}. \end{aligned} \tag{3.1}$$

Using the same assumptions, the last restriction is reduced as follows:

$$\begin{aligned} 4n \leq k^{5/12}/N^{1/3} &\iff \sqrt[12]{k^5} \geq 4n \\ &\iff k^5 \geq (4n)^{12} \\ &\iff k \geq (4n)^{12/5} \\ &\iff k \geq 16 \times 2^{4/5} n^{12/5}. \end{aligned} \tag{3.2}$$

This reasoning shows that, for any given sam_{size}, k may be adjusted as desired automatically.

3.6.2.2 The Remaining Statistical Tests

The parameters for the statistics discussed below are introduced objectively, in a rigorous manner, paying attention to the underlying mathematical restrictions for finding the minimal values possible to assign for the parameters. However, this process is not straightforward since some statistical tests might not behave *as is*, for it may be possible to call for additional numbers in case some statistical property is not satisfied, thereby comprising an obstacle. As such, some of the parameter values utilized in the tests undertaken were found by means of human discernment, left to subjective analysis and repeated executions of the statistics for the same datasets in order to use as many as possible from the available number of samples without surpassing that threshold. The explanation of some of the parameters and inherent configurations are included in subsection 4.6.1. The remaining statistical tests, without counting *BirthdaySpacings*, can be described as follows:

- ***Collision***: The *Collision* test is based on *Serial* test, and is of the form `sknuth_Collision(gen, NULL,` N, n, r, d, t`)`. The *Collision* test divides the unit hypercube (0, 1)^t in $k = d^t$ small hypercubes, generates n points (t-dimensional vectors) in $[0, 1]^t$ using n non-overlapping t successive outputs from the uniform generator. It then compares the number of collisions (the number of times a point hits a cell already occupied) with the expected values. As such, for $t = 2$ the smallest n value is sam_{size}/t. The optimal value for parameter k is found by means of subjective analysis.

- ***Gap***: *Gap* generates n values in $[0,\ 1)$ and, for $s =$ 0, 1, 2, . . . , counts the number of times a subsequence of length s falls within the interval $[\alpha, \beta]$, where α and β are small constants with $0 \leq \alpha < \beta \leq 1$. It then compares with pre-computed expected values for s. *Gap* takes the form of `Gap(gen, NULL,` N, n, r, α, β`)`. The amount of numbers requested by this statistic is variable, since it depends on the maximum value of s that is automatically defined internally by the statistic. As such, n was manually configured for each of the tests.

- ***SimPoker***: The *SimPoker* statistical test computes the number s of distinct integers for each of the generated n groups of k integers in {0, . . . , d−1}, and then compares each observed s with the expected values. *SimPoker* is defined by `sknuth_SimPoker(gen, NULL,` N, n, r, d, k) with the restrictions $d < 128$ and $k < 128$. In this case, finding n assumes the form of sam_{size}/k.

- ***CouponCollector***: Similarly to *SimPoker*, the *CouponCollector* test generates a sequence of random integers in {0, . . . , d−1} and counts how many must be generated before each of the possible d values appears at least once, and repeats this process n times, to finally count how many times exactly s integers were needed. Each s is compared with the expected values. *CouponCollector* is of the form `sknuth_CouponCollector(gen, NULL,`

N, n, r, d). The only restriction is $1 < d < 62$. As in *Gap*, this test executes various times for distinct values of s not controlled by the user. As such, n was manually set.

- ***MaxOft***: This test generates n groups of length t each and calculates the maximum X for each group. As such, the optimal value of n is sam_{size}/t. The form of this test is `sknuth_MaxOft(gen, NULL,` N, n, r, d, t). The restriction of *MaxOft* is $n/d \geq$ `gofs_MinExpected`.

- ***WeightDistrib***: The *WeightDistrib* test generates k uniforms and counts how many fall within the interval $[\alpha, \beta]$. It repeats this process n times, thereby requiring nk numbers as input, and finally compares with the expected values. This test is defined by `svaria_WeightDistrib(gen, NULL,` N, n, r, k, α, β).

- ***MatrixRank***: This test creates an $L \times k$ matrix with each row being filled with s bits from k/s uniforms. The goal of the test is to compute the rank of the matrix or, in other words, the number of linearly independent rows, for which it is needed to compute n matrices and count how many there are of each rank. *MatrixRank* is of the form `smarsa_MatrixRank(gen, NULL,` N, n, r, s, L, k), and it is recommended for $L = k$ or $|L - k|$ to be small if L and k differ.

- ***HammingIndep***: The *HammingIndep* test applies two sub-tests. The first initially builds $2n$ blocks of L bits from the bits provided in the input numbers. Then, it computes the Hamming weights[1] X_j for each of the blocks, with $1 < j < 2n$. Each X_j can take one of $L + 1$ possible combinations. This first sub-test counts the number of each $L + 1$ possibilities among non-overlapping pairs {(X_{2j-1}, X_{2j}), $1 \leq j \leq n$)} with $(L+1) \times (L+1)$ possible values, and compares that with the expected values. The restriction for the first sub-test is $n \geq 2 \times$ `gofs_MinExpected`. A second sub-test takes that counting into an $(L + 1) \times (L + 1)$ matrix, divides it in four equal sub-matrices of the corners of the main matrix, and calculates $Y_3 = n - Y_1 - Y_2$, where Y_1 and Y_2 are the sum of the counters of the lower left and upper right sub-matrices and of the lower right and upper left sub-matrices, respectively. Y_1, Y_2, and Y_3 are tested against the expected values. The restrictions for the second sub-test are $d \leq$ `sstring_MAXD` and $d \leq (L + 1)/2$. Since there are not way to derive n, it was also needed in this test to manually configure it. This test is attainable through `smarsa_MatrixRank(gen, NULL,` N, n, r, s, L, d).

- ***RandomWalk1***: The *RandomWalk1* test applies various random walks over the set of integers $\mathbb{Z}$. The walk starts at zero and, from there onward, it moves to the left or right with probability of 0.5 to either way. It repeats walks of length in the interval $[L_0, L_1]$ by iterating with two steps. For example, the first walk is of length L_0, the second walk is of length $L_0 + 2$, the third one is of length $L_0 + 4$, and so on until L_1 is reached. The

[1] The Hamming weight is a measure of a string that defines the number of symbols different from the zero-symbol of the respective alphabet. In binary computing, the Hamming weight refers to the number of bits equal to one (different from zero).

test takes s bits from each uniform to generate the random walk process, for which it is required at most L_1/s numbers for a walk of length L_1, for instance. *RandomWalk1* contains sub-statistics that make the test generate n random walks. This test is of the form `swalk_RandomWalk1(gen, NULL,` N, n, r, s, L_0, L_1`)`. L_0 and L_1 must be even and $L_0 \leq L_1$.

3.6.3 Main Analysis Program in C Programming Language

TestU01 was written in C and provides an API in that programming language. Hence, the main program for testing the quality of the random numbers was developed in C. For each test undertaken, each sample file is read in the following form. The first 100,000 random numbers are read onto an `unsigned int` array. Whenever a statistic asks for a random number, the next 32-bit number stored on the array is converted to a floating-point number and returned to the statistic. The file is rewinded whenever the request count surpasses the number of samples available on it. Otherwise, if the request count overflows the size of the array, the next 100,000 random numbers replace the array with new numbers, and so on until there are no numbers left, at which point the file is rewound to the beginning also.

Each of the four-byte unsigned integers are converted to floating-point numbers within the range $[0, 1)$. This is done by multiplying each number with $2.328\,306\,437\,080\,797\,4 \times 10^{-10}$. This constant is used for obtaining high floating-point precision for that interval, and was retrieved from the Mersenne Twister C implementation available on the website [Nis] of one of the authors of the algorithm. The results of the transformations to floating-point numbers are stored on the `double` format with a mantissa significand precision of 53 bits, which is enough for storing 32 bits of data. The `float` type would not be enough since its mantissa fraction has a significand precision of only 23 bits according to the IEEE 754 standard for floating-point arithmetic. Such an execution flow is implemented on a function whose purpose is the same as the `rng` handler illustrated in listing 3.1. The rest of the example on that listing pretty much sums up the main program, with the difference that the customized test battery is invoked instead of `bbattery_SmallCrush`. Additional TestU01 headers are nonetheless required to include in the program for using each statistic individually.

3.7 Conclusions

This chapter has detailed the method adopted for auditing the `/dev/random` device of the Linux RNG. It is divided in three ways over two main phases that were overviewed in the beginning of the chapter. The sampling phase consists in collecting data from `/dev/random` for post-analysis in terms of throughput efficiency and quality of the random numbers. For that end, various Python scripts and C programs along with a few command line utilities were used as means during the process. A set of metrics characterizing each dataset was put in place for assessing the throughput efficiency of `/dev/random`, while the well-known TestU01 library was chosen for

analyzing the randomness quality throughout various IaaS cloud scenarios. On the one hand, the metrics will help to quantify how bad the slowness problem addressed in this dissertation is by exhibiting properties inherent to each test run. On the other, various TestU01 statistics manually configured will help to study the part of the problem related with randomness quality.

The method for collecting and analyzing `/dev/random` allowed pursuing the main goal of the masters program. Employing the method to each of the cloud setups discussed afterward enabled the analysis of the resulting datasets in terms of efficiency and quality. Along this part of the work, it was concluded that the virtualization poses limitations in terms of the number of random numbers that can be collected in time to conduct the analysis. The size of the datasets is therefore dependent of that fact, which required the adjustment of the statistical tests of TestU01 to ensure meaningful results. As a follow up to this chapter, the next one discusses the scenarios on which the several datasets were obtained and presents the results of their analysis.

Chapter 4

Tests and Results

This chapter describes the experiments performed to test `/dev/random`, which resulted from the natural follow up of the method explained in the previous chapter. These experiments included collecting and processing several datasets. The analysis of the datasets is presented in a timely manner first and the results obtained are discussed under the context of the problem addressed by this dissertation in a subsequent part of the chapter.

4.1 Introduction

The previous chapter shed light upon the method devised for auditing `/dev/random` thoroughly. Such an assessment is performed herein from two different perspectives. The first concerns the efficiency of the throughput, and it is performed by analyzing the timestamps on which the device was unblocked for reading, and also by paying attention to the potential timestamp overlapping between guests and hosts and between guests. The second concerns the quality of the produced random numbers, looking for signs of lower randomness strength in the hypothetical case that virtualization results in an undermined Linux RNG. To perform these analyzes, several testbeds, described with detail in Section 4.2, were devised while having in mind different cloud setups and using various distinct technologies and hardware. This was done so as to look at the problem from a macro perspective and thus obtain a big picture of the problem. The datasets obtained from the execution of the method on such testbeds are afterward characterized in Section 4.3. The discussion of the datasets with focus on the efficiency of the Linux RNG, particularly of `/dev/random`, is included subsequently in Section 4.4. A case study for quantifying the slowness found on the RNG due to entropy scarceness is presented in Section 4.5, which covers its impact of the aforementioned scarcity in user-space cryptographic applications, namely the GNU Privacy Guard (GPG) command line utility. The analysis of the randomness characterizing the `/dev/random` outputs is finally included in Section 4.6.

4.2 Testbeds

The testbeds utilized for executing each of the experimental tests aggregate a multitude of different computers and technologies so as to investigate if distinct factory hardware and hypervisors impacts entropy gathering on the Linux RNG. With that purpose in mind, two production servers, along with two modern desktops and one laptop were used. Amazon EC2 was also utilized for running various experiments. The setup mentioned in last provides the tests with more

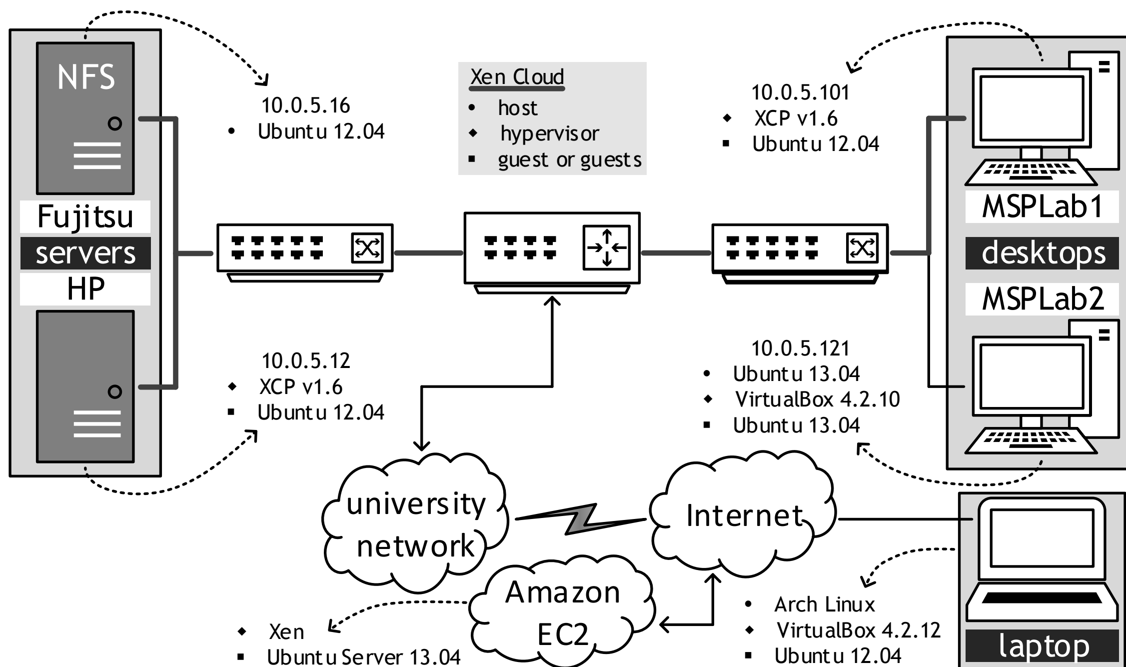

Figure 4.1: Schematics of the various testbeds used for conducting the tests.

realistic data, since EC2 is a popular IaaS cloud used for all kinds of purposes, namely researching, testing, and also for businesses. A layout illustration of the arrangement of the servers and desktops is shown in Figure 4.1. As may be concluded from the observation of the figure, the two servers and one of the desktops, herein called *MSPLab1*, were set aside for creating a dedicated Xen cloud network. The other desktop, named *MSPLab2*, was used independently of the others, analogously to the laptop.

The method presented in the previous chapter was divided into three main phases. The collecting phase consisted of gathering random numbers from `/dev/random`. For collecting a more or less sufficient amount of data in each dataset to submit it to TestU01, it was needed to instantiate the VMs and leave them on for certain periods of time, spanning from several hours to a few days or weeks. This approach allowed gathering as many random numbers as possible, even while the Linux RNG was outputting random material very slowly.

4.2.1 Setups and Specifications of the Hosts and Guests

The servers used in the experiments are Fujitsu Siemens Primergy TX150 S5 with 1 GB of Random Access Memory (RAM) and an Intel Pentium D, and HP ProLiant ML350 G5 with 8 GB of RAM and an Intel Xeon E5405 CPU. The desktops are similar, being characterized by the 6 GB of RAM and the Intel Core i7 processors. Finally, the laptop was a Toshiba with 8 GB of RAM and an Intel Core i5. The Fujitsu Siemens server was used for setting up a Network File System (NFS) for the Xen cloud. It kept template images for instancing VMs rapidly, functioning like a centralized storage point for the VMs disks and files as well. This is useful for disaster recovery, load balancing, and VM migration while easing management tasks. If a VM goes down on some physical node,

it can be quickly booted on another machine with little overhead and without losing data. The Xen cloud was mounted locally within the university campus (in the lab network) using the Xen Cloud Platform (XCP) version 1.6 base installer [Thea]. The HP server, along with *MSPLab1*, were used to instantiate various VMs. The HP server ran a 32-bit Ubuntu 12.04 guest while *MSPlab1* ran two co-resident guests of the same image template. *MSPLab2* ran various tests with various 64-bit VMs during this part of the work. VirtualBox 4.2.10 was installed in this computer within the 64-bit Ubuntu 13.04 OS. The laptop ran Arch Linux with VirtualBox 4.2.12 and a 32-bit Ubuntu 12.04 guest. Amazon EC2 was used as means to fire up a myriad of guests running 64-bit Ubuntu Server 13.04 over Micro instances[1].

4.2.2 Virtualization Configurations and the `RdRand` Instruction

Regarding the local testbeds, none of the computers had SSD disks, VT-x was enabled for hardware virtualization acceleration, and usage was kept to a minimum, with a few periodic SSH sessions for checking the status at most. This, combined with multi-core CPUs, assures that CPU time races did not probably occur and, therefore, the hypervisors were scheduling the VMs free of constraints. Except for the memory and disk sizes, the remaining VM parameters were kept to the default values when creating new VMs. Such default settings include virtual memory size, VCPUs (multi-threading technology for performance optimization), VNICs, CPU execution cap (amount of time a host CPU spends emulating a VCPU), and so forth. All VNICs were networked through bridges to the physical networks. It is noteworthy to say that the CPUs of the machines used to mount the testbeds did not support the `RdRand` instruction of Intel Secure Key technology for generating true random numbers. Since version 12.04, Ubuntu ships with the kernel version 3.2.0 or higher. Since that kernel version, the Linux RNG uses `RdRand` as a randomness source, if the CPU offers support for such instruction (this technology is relatively new and is included in the latest generation CPUs only). This is not the case for the tests herein executed, meaning that `RdRand` was not available. CPUs can be tested for this instruction through a library and API provided by Intel from their website for Windows [Mec] and both Linux and Mac OS [Int].

4.3 Tests and Datasets

The outcome of the tests is discriminated in Table 4.1. The table is organized according to the tests that were performed, grouping them by coloring with gray or white the background of the rows in an interleaved manner. Each test in the table is labeled and sequentially numbered, and includes the hosts or guests that participated in the test. The following subsection details the tests, while the subsequent one describes the notation used to represent the metrics obtained

[1] On Amazon EC2, Micro instances [Amaa] are a low-cost instance option to which few CPU resources are allocated to. It may get additional computing cycles in short bursts whenever they are available. Micro instances are appropriate for low-throughput applications, and can be used freely for periods of no longer than 750 hours combined. For example, two Micro instances can only run 375 hours each, otherwise one gets billed.

Table 4.1: Datasets collected in the several scenarios used in the scope of this research work along with the results obtained for the metrics defined to evaluate them.

		Datasets	Δt (hours)	$\#_s$	P_i	$P_{comm(i)}$	$\overline{\Delta i}$ (σ) (seconds)	$\overline{si}$ (σ)	$s/second$
Desktop	1	Host	80	1465209	16.36%	–	1.13 (0.43)	5.72 (2.16)	5.05
	2	VBox Guest	59	16552	49.58%	–	26.17 (5.50)	2.02 (1.75)	0.08
	3	Host	72	1318162	18.03%	4.43%	1.09 (0.35)	5.55 (1.98)	5.08
		VBox Guest		23076	49.79%	91.66%	22.20 (6.85)	2.01 (0.12)	0.09
	4	Host	787	14038095	18.49%	2.42%	1.09 (0.36)	5.41 (1.85)	4.95
		VBox Guest		212044	49.91%	59.34%	26.33 (4.72)	2.00 (0.33)	0.08
	5	VBox Guest (snapshot)	48	13380	49.90%	4.78%	26.04 (5.95)	2.00 (0.09)	0.08
		VBox Guest (reset)		12927	49.93%	4.94%	26.29 (6.08)	2.00 (0.09)	0.08
Laptop	6	Host	39	114177	45.85%	2.25%	2.68 (2.46)	2.18 (0.57)	0.81
		VBox Guest		6357	49.72%	37.27%	44.46 (27.40)	2.01 (0.14)	0.05
	7	Host	135	166759	49.04%	2.24%	5.95 (6.81)	2.04 (0.27)	0.34
		VBox Guest		21798	49.77%	16.89%	44.89 (25.66)	2.01 (0.17)	0.05
Xen	8	XCP Guest	228	93794	49.64%	–	17.67 (3.83)	2.01 (0.12)	0.58
	9	XCP Guest	1028	436762	49.63%	–	17.08 (3.24)	2.02 (0.13)	0.12
	10	XCP Guest #1	118	49561	49.74%	6.93%	17.26 (3.36)	2.01 (0.10)	0.12
		XCP Guest #2		48489	49.66%	7.09%	17.67 (3.48)	2.01 (0.12)	0.11
	11	XCP Guest #1	951	384529	49.59%	5.37%	17.95 (3.62)	2.02 (0.14)	0.11
		XCP Guest #2		345613	49.78%	5.95%	19.90 (3.60)	2.01 (0.11)	0.10
Amazon	12	EC2 Instance #1	238	21632	49.90%	–	79.54 (13.92)	2.00 (0.06)	0.03
	13	EC2 Instance #2	694	61890	49.92%	–	80.94 (11.53)	2.00 (0.06)	0.02
		EC2 Instance #3		37681	49.91%	–	132.92 (61.90)	2.00 (0.06)	0.01
		EC2 Instance #4		64229	49.89%	–	78.02 (16.85)	2.00 (0.07)	0.03
		EC2 Instance #5		62563	49.91%	–	80.05 (13.32)	2.00 (0.06)	0.03
		EC2 Instance #6		61705	49.92%	–	81.16 (10.91)	2.00 (0.06)	0.02
		EC2 Instance #7		62278	49.92%	–	80.42 (12.33)	2.00 (0.06)	0.02
		EC2 Instance #8		63673	49.91%	–	78.67 (15.65)	2.00 (0.07)	0.03
		EC2 Instance #9		61987	49.92%	–	80.78 (11.75)	2.00 (0.06)	0.02
		EC2 Instance #10		61916	49.92%	–	80.88 (11.58)	2.00 (0.06)	0.02
		EC2 Instance #11		62283	49.91%	–	80.41 (12.47)	2.00 (0.06)	0.02
		EC2 Instance #12		63040	49.91%	–	79.46 (14.39)	2.00 (0.06)	0.03
		EC2 Instance #13		61890	49.92%	–	80.92 (11.64)	2.00 (0.06)	0.02
		EC2 Instance #14		62177	49.92%	–	80.55 (12.35)	2.00 (0.06)	0.02
		EC2 Instance #15		62192	49.92%	–	80.52 (12.21)	2.00 (0.06)	0.02
		EC2 Instance #16		61892	49.92%	–	80.91 (11.51)	2.00 (0.06)	0.02
		EC2 Instance #17		64129	49.89%	–	78.14 (16.72)	2.00 (0.07)	0.03
		EC2 Instance #18		61956	49.92%	–	80.83 (11.63)	2.00 (0.06)	0.02
		EC2 Instance #19		62164	49.92%	–	80.55 (12.16)	2.00 (0.06)	0.03
		EC2 Instance #20		63011	49.91%	–	79.49 (14.35)	2.00 (0.07)	0.03
Hosts Average			344	**3420480**	**29.55%**	**2.84%**	**2.39 (2.08)**	**4.18 (1.37)**	**3.25**
Guests Average				**86338**	**49.85%**	**51.29%** **5.84%**	**59.97 (12.50)**	**2.00 (0.14)**	**0.06**

from the datasets.

4.3.1 Details of the Tests

As can be concluded from the observation of Table 4.1, a total of 13 tests using the various testbeds outlined in the previous section were carried out. The desktop was used to singularly execute tests number 1, 2, 3, 4 and 5, while the laptop conducted tests 6 and 7. The two servers and the other desktop were utilized as means to mount a local Xen cloud, on which the tests number 8, 9, 10 and 11 were performed. Lastly, tests number 12 and 13 were done on EC2. Each of the tests may be described as follows:

1. The test number 1 consisted simply in collecting data from `/dev/random` on the desktop host, without running any VM;

2. In turn, test number 2 ran a VirtualBox guest on the desktop without harvesting `/dev/random` from the host machine;

3. Tests number 3 and 4 consisted in reading material from `/dev/random` on both guest and host at the same time;

4. Test number 5 covered the snapshotting process and consequent restore while saving random numbers after the snapshot and after the restore for similar periods of time;

5. The laptop tests number 6 and 7 both consisted in collecting random numbers at the same time on the guest and host;

6. Passing on to the Xen tests, tests 8 and 9 ran single VMs on the HP server at distinct times;

7. Tests 10 and 11 ran co-resident guests on *MSPLab1*;

8. Finally, the test number 12 consisted in instantiating a single VM on EC2;

9. Afterward, it was found convenient to instantiate VMs in bulk. This was the case for test number 13, on which 19 VMs were booted up, all running on the Oregon data center of EC2, located in the west region of the USA.

4.3.2 Metrics Notation

The notation utilized in most of the columns of Table 4.1 refers to the metrics quantifying the efficiency of the throughput of `/dev/random`. These metrics can be further specified as follows. Herein, s and i are used to refer to samples (four byte-sized random numbers) and instants, respectively. The approximate total sampling time in hours and the total number of samples collected are given by Δt and $\#_s$, respectively. P_i expresses the percentage of instants in which data was collected under the total number of samples. It is possible to harvest various samples on the same instant (herein, instants are referring to the timestamps, recall it has a precision of a second), and thus this metric gives the percentage of unique instants by removing repeated ones. $P_{comm(i)}$ denotes the percentage of instants in which samples were collected at the same time in guests and hosts or in two guests. Finally, $\overline{\Delta i}$ (σ) denotes the average value of the differences between instants, while $\overline{si}$ (σ) gives out the average number of samples per instant, with the standard deviation of the obtained values represented between brackets. The rate of samples per second is expressed by $s/second$. The average values of all of the results are depicted in the last two rows of the table. Regarding the $P_{comm(i)}$ column, two averages are presented for the guests row. The first (top) one refers to the average value for tests number 3, 4, 6 and 7 (host versus guest), while the second (bottom) one concerns tests number 5, 10 and 11 (guest versus guest). Most of the values are rounded to the second decimal place.

For example, test number 3 lasted approximately 72 hours, during which 1318162 and 23076 samples were obtained for the host and the guest, respectively. The number of instants on

which the numbers were read on the host is obtained by taking 18.03% of 1318162, which is 237638, approximately, from which 4.43% are in common with the instants seen on the guest. In turn, the guest shows approximately 11489 instants, being 91.66% of them in common with the host. Finally, it is possible to read approximately 5.55 four-byte integers from `/dev/random` at intervals of 1.09 seconds on the host, while the guest only permits a read of 2.01 samples at each 22.20 seconds, on average. These results determine rates of 5.08 samples per second for the host against 0.09 for the guest.

4.4 Throughput Efficiency of `/dev/random` Outputs

This section looks into the throughput efficiency of the Linux RNG by analyzing the values of the metrics obtained for each of the tests and depicted in Table 4.1. The analysis is done while having in mind the underlying cloud setups, aiming at contrasting the results from guests and hosts as well. For this, a textual analysis is first carried out, and then a graphical visualization of two of the tests illustrates the discussed facts.

4.4.1 Analysis of the Instants

The values evidenced in Table 4.1 confirm the expectations of slow entropy gathering on Linux guests, particularly relevant to the generation of random material of the `/dev/random` device. The desktop host is pretty quick generating random numbers, and was the one producing larger datasets with a rate of around five samples per second (the value of metric $s/second$). The values for $\mathrm{P}_{comm(i)}$ seemed to be normal. Since the desktop produced many numbers at a high rate, the likelihood of doing so in the same instant that the guests do so is large, hence the high percentages for $\mathrm{P}_{comm(i)}$. This was not the case for the other tests, including the co-resident guests on the Xen cloud, thereby not showing a clear relation between one another given the underlying shared environment. Regarding the VirtualBox guests on the desktop, the most distinct factor from the other tests on VMs are the values corresponding to $\overline{\Delta i}$ (σ). In fact, $\overline{\Delta i}$ (σ) got different values for each test group with respect to the underlying technology. The most concerning one was obtained for EC2, which turned out to be really slow generating random numbers. All of the 20 instances got values superior to 79 seconds for $\overline{\Delta i}$ (σ), with a standard deviation oscillating around 11 seconds mostly. Moreover, it was recorded a residual variability on the number of samples per instant and its standard deviation (*i.e.*, $\overline{si}$ (σ)). Each of the values for $\#_s$ and P_i were also very similar, even for different sampling times. These results are very consistent, showing a highly and widely spread identical behavior on EC2 instances. Instance number three actually got all the way up to 132.92 seconds, although the associated standard deviation was higher. This particular result of that instance may be considered an outlier, but the data is faultless. The situation might be worrisome in this case.

It was expected to see a higher throughput on the laptop. However, it was observed to behave similarly to guests with respect to $\overline{si}$ (σ). This means the kernel of Arch Linux was only able

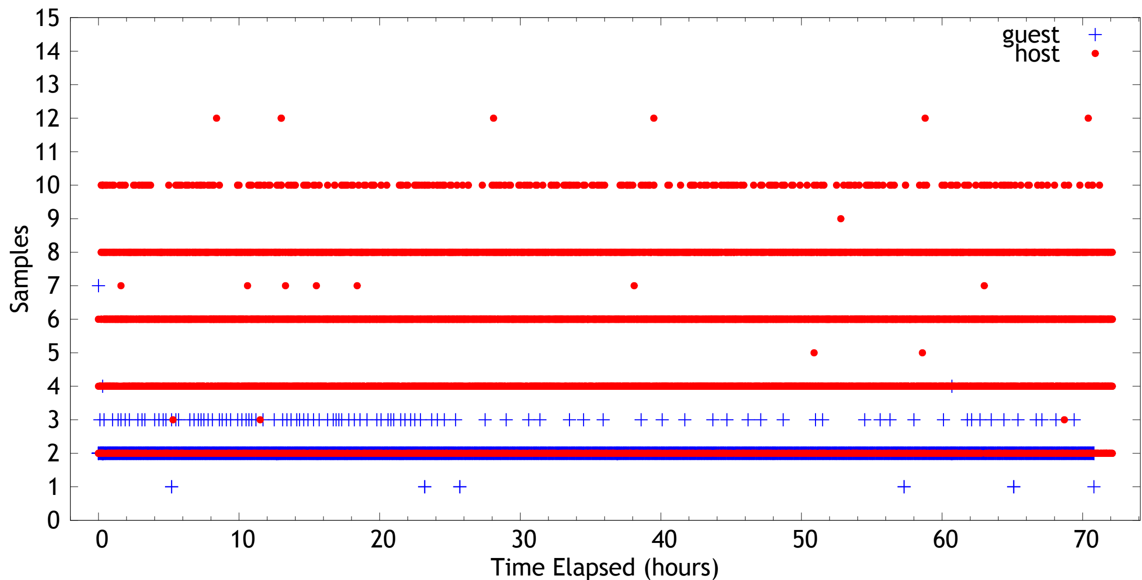

Figure 4.2: Histogram of the number of samples per instant for test number 3.

to harvest enough entropy for four-byte chunks of random data per instant, on average, just like what is seen on the guests. Nonetheless, the VirtualBox guests that ran on the laptop showed a high variability concerning the metric $\overline{\Delta i}$ (σ), taking reasonable high periods to unblock `/dev/random`. Similarly to what is observable in EC2, the results of the Xen cloud also showed to be very consistent for distinct sampling times, and actually similar to the ones of EC2, except for $\overline{\Delta i}$ (σ), which argues that those guests were quicker generating random material. Foremost, the results depicted in Table 4.1 point to a yet unattended, widely spread problem on IaaS cloud computing setups regarding the throughput efficiency of the Linux RNG.

4.4.2 Histograms

As described above, the Linux RNG is slow gathering entropy for the `/dev/random` device when running on VMs. Figure 4.2 and Figure 4.3 depict histograms of the number of samples per instants for tests number 3 and 7, respectively. By averaging out the values of the histograms, one obtains $\overline{si}$ for each of the tests. Because `/dev/random` blocks for quite some time on guests, an entropy refill only provides sufficient amounts to generate two four-byte chunks of random data, while a residual value of 1.09 seconds for $\overline{\Delta i}$ and a higher 5.72 value for $\overline{si}$ is seen on the host for test number 3. This can be observed on Figure 4.2, where the guest barely surpasses the barrier of the number four of samples. Note that the guest outlier marking seven at the beginning of the test is due to the initial entropy reserves that the pool had, which was emptied quickly when the script ran. Interestingly, the host produces an even number of samples per instant because entropy extraction from the main pool is done ten bytes at a time.

Figure 4.3 includes two histograms concerning test number 6. The top one is the histogram for the guest, while the bottom one is for the host. Both are colored according to the usage and non-usage periods carried out on the underlying laptop. The guest maintains the same behavior,

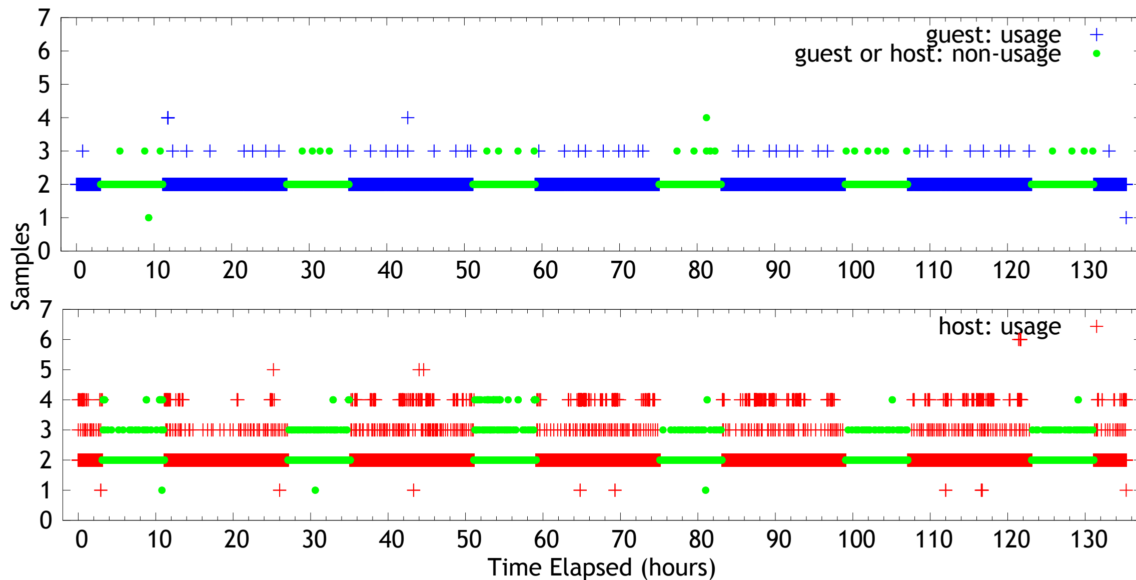

Figure 4.3: Histogram of the number of samples per instant for test number 7.

but the host produces more random material when the laptop is actively processing tasks. It reaches the number of four samples occasionally with usage, including the period between 50 and 60 hours of sampling, which actually comprised normal usage. These results also show that no apparent relation is witnessed between host and guest, with respect to the possibility of generating entropy at some equal instant in time, because the generation of material on guests does not change for busy and inactive periods. The average value of 5.84% regarding $P_{comm(i)}$ also tells that co-resident VMs produce random material at non overlapping instants.

4.5 Case Study — Impact of Low Entropy on Applications

In addition to the previous test on the direct throughput of the `/dev/random` device, a simpler experiment was carried on a cryptographic application that entirely depends on that device. This case study concerns the GPG cryptographic suite that is included in most Linux distributions. It may be used via the `gpg` command on the terminal. Its `--gen-key` option is the one of interest since it instructs `gpg` to generate cryptographically strong, long-lived keys by uniquely requesting random data from `/dev/random`. The number of bytes read from the device depends on the specified key size. For key sizes of 2048 bits, `gpg` needs at least 2048 bits of random material a piece.

It can be verified by means of debugging that `--gen-key` indeed reads `/dev/random` for this purpose. On Linux, the `strace` utility (the equivalent on Mac OS is `dtruss`) comes in handy. It allows to trace the behavior of programs as they run. By executing the following command with root privileges:

```
$ sudo strace -e trace=open,read gpg --gen-key
```
,

one may observe that, somewhere near the beginning of the output of the command, there is

Table 4.2: Results of the gpg tests.

Time Elapsed	EC2 Guest	VBox Guest	Host	Yarrow
$\bar{t}$ (σ) **(seconds)**	3052.18 (531.36)	858.99 (257.52)	21.32 (6.49)	0.38 (0.15)

a line with `open("/dev/random\0", 0x0, 0x0)`. In contrast, `openssl genrsa` and `ssh-keygen` use `/dev/urandom`.

The measurements were conducted by timing `gpg` generating 2048-bit RSA keys using `--gen-key`. This procedure was repeated on 30 runs on an Ubuntu 13.04 desktop host, on a VirtualBox Ubuntu 13.04 guest and on an EC2 Ubuntu Server 13.04 instance. Additionally, the Yarrow algorithm (a PRNG) implemented in Mac OS OSes was included in this case study so as to get a better perspective of the results. The mean value of the time elapsed in seconds on each of the 30 runs along with the standard deviation was retrieved, herein denoted by $\bar{t}$ (σ). For this purpose, the Python wrapper for GPG available at [Kuc] was used for writing a script to generate dummy keys on 30 runs and calculate the mentioned mean and standard deviation. The amount of seconds since the Unix Epoch is again used for timing purposes. Testing `gpg` for different key sizes and iterations would not provide additional meaningful information. In fact, it would be superfluous because the slowness found earlier dictates entropy pools to be replenished at a constant low rate on VMs.

Table 4.2 depicts the results of the tests on the `gpg` utility. The EC2 guest took over three times more than the desktop guest to generate 2048-bit keys. This is not surprising since the ratio of the average rates of samples per second depicted in Table 4.1 for desktop guests and the EC2 infrastructure is almost three. In turn, the guest also takes longer than the host. The fast Yarrow algorithm largely beats `/dev/random` when it runs within VMs or on the host. The results illustrate the practical implications of slow entropy gathering of the Linux RNG on VMs when compared to the hosts counterparts and to PRNGs. It can be said that the Linux kernel does not supply enough entropic inputs to the Linux RNG for the needs of user-space applications. Taking so much time to generate cryptographic keys may not feasible in many cases.

4.6 Quality of `/dev/random` Outputs

Assessing the randomness quality calls upon stringent statistical analyzes aiming at finding flaws in the statistical structure of random sequences or subsequences. This is the case at hands. TestU01 was used to analyze the outputs of the `/dev/random` device for all of the tests described above. Because of the slowness of the Linux RNG, it was necessary to manually configure each of the tests within the *SmallCrush* battery of TestU01. In the quality tests presented in this section, all datasets within a particular test were merged with respect to the instants, except for tests number 14, 15 and 16. For example, both the guest and host datasets of test number 3 were merged into a new file sorted in accordance with the associated timestamp of each sample.

This way, it is possible to focus on the several problems, tested by each cloud scenario, which was actually the motivation for building such setups. For the confidence interval considered throughout these analyzes, which is $[10^{-10},\ 1 - 10^{-10}]$, p-values falling within the disjoint set $(10^{-10},\ 10^{-4}] \cup [1 - 10^{-4},\ 1 - 10^{-10})$ can be considered *suspect p-values* [LS07]. The way to suppress such cases would be to increase the sample size. Since that is not achievable in feasible time using the Linux RNG on VMs, as already explained, they can be baffled by looking at the p-values outputted by the other tests. If a certain random sequence is inherently a statistically bad random sequence, the customized battery of tests will fail systematically for a large part or all of the included tests.

4.6.1 Review of the Statistics and Configurations

Most of the tests in TestU01 have both common and specific parameters that need to be adjusted accordingly to smaller sample sizes, as detailed in subsection 3.6.2. Table 4.3 contains the parameters which were fixed for all the tests discussed in this dissertation, while Table 4.4 depicts the configurations of the variable parameters n and d. The parameter that most influences the amount of random numbers required by a test is n. The dimension of the hypercube (0, 1)^t is expressed by t, and defines the dimension of the testing space (*e.g.*, the size of cell matrices for throwing points at). Parameters d and k have multiple purposes. In the *BirthdaySpacings* and *Collision* tests, d^t gives the number of small hypercubes used for filling with points, while on *SimPoker* and *CouponCollector* d defines an integer range to which the statistics convert input numbers to. On *MaxOft*, it characterizes a proportion given by n/d, while on *HammingIndep* it is used for discarding certain rows and columns from matrices. As for k, in *SimPoker* and *WeightDistrib*, it expresses the size of each group of numbers created by those tests, therein requiring nk numbers to fill the groups. On *MatrixRank*, it is used for defining an $L \times k$ matrix. On the other hand, L denotes the size in bits of $2n$ blocks on the *HammingIndep* statistic. For the *Gap* and *WeightDistrib* tests, α and β define an interval for counting the numbers falling within it. The *Gap* requires a lot of random numbers to carry the statistical processing. As such, this test was not executed to some datasets. A ✖ in Table 4.3 and Table 4.5 signals these cases. Finally, L_0 and L_1 build an interval of walks for the sub-statistics (these are later explained in subsection 4.6.3) within *RandomWalk1*. TestU01 allows to drop r bits from input numbers and then use the following s bits. TestU01 additionally allows to independently replicate each statistic by setting a parameter N. This was not done in the tests, meaning that the replication number was set to one (single testing). Except for *MaxOft*, *HammingIndep* and *RandomWalk1*, all other statistics output a single p-value. *MaxOft* outputs two p-values, while *HammingIndep* calculates three p-values. The amount of p-values processed by *RandomWalk1* depends on the walk length interval specified as arguments to the function.

Tuning up each of the statistics is not straightforward and even though they were configured thoroughly and carefully, only an approximate amount of the input samples were used. That

Table 4.3: Configuration for the fixed parameters used on the customized TestU01 battery for all the statistical tests.

Statistics	k	r	s	t	α	β	L	L_0	L_1
BirthdaySpacings	-	0	-	2	-	-	-	-	-
Collision	-	0	-	2	-	-	-	-	-
Gap	-	0	-	-	0.0	0.00390625	-	-	-
SimPoker	32	0	-	-	-	-	-	-	-
CouponCollector	-	0	-	-	-	-	-	-	-
MaxOft	-	0	-	2	-	-	-	-	-
WeightDistrib	32	0	-	-	0.0	0.125	-	-	-
MatrixRank	32	0	10	-	-	-	32	-	-
HammingIndep	-	0	32	-	-	-	75	-	-
RandomWalk1	-	0	32	-	-	-	-	140	150

means it could be a little less or a little more of the available samples. If it were a little less, the remaining are ignored, otherwise the superfluous ones are reused without impacting the statistical assessment, since it would entail a very small percentage of the total number of samples (*e.g.*, 0.005%). The adjustment of the combination of the statistics is done automatically at runtime (*i.e.*, the developed program does that adjustment). For smaller datasets, such an offset from the available samples would accordingly be a small value, like one hundred random numbers, while for larger datasets the offset could increase up to a few thousands.

4.6.2 Less Stringent Batteries

Although the less stringent batteries included in TestU01 do not grind a generator thoroughly, they can be useful for preliminary analyzes. If random sequences fail such tests, then it is not necessary to further submit them to the more stringent tests. The first battery executed was `bbattery_FIPS_140_2`, which requests less than one thousand numbers. All datasets passed this battery with ease. TestU01 further provides less stringent batteries whose APIs are `bbattery_Rabbit`, `bbattery_Alphabit`, and `bbattery_BlockAlphabit`. The last one applies the middle one with the difference that the `unif01_CreateBitBlockGen` filter is applied. This filter takes 32 32-bit integers at a time and builds a matrix $\mathcal{B}$ with the bits of each one. Then, $\mathcal{B}$ is divided into sub-matrices which are used to create new integers from the rearrangement of the bits in those sub-matrices. The last three batteries mentioned before take as input the number of bits to be used for each of the tests included in each battery. In the tests undertaken, such value was set to the maximum number possible accordingly to the sample size given as input, obtained from $sam_{size} \times 4 \times 8$, where sam_{size} is the sample size. All datasets passed each of these batteries, with just a few suspect p-values falling within $(10^{-5},\ 10^{-4}] \cup [1-10^{-4},\ 1-10^{-5})$. These can be regarded as suspect indicators, whereas the decisive tests of the customized stringent test battery will either confirm or disapprove them.

The DIEHARD statistical testing package was, for many years, the *de facto* approach for testing RNGs. TestU01 implements this battery of tests, being `bbattery_pseudoDIEHARD` its API.

Table 4.4: Configuration for parameters n and d used on the customized TestU01 battery with regard to tests number 1 through 16. Each pair of values in the cells is of the form $(n,\ d)$, with d not being applicable to some of the statistics.

Tests			B.S.	Col.	Gap	S.P.	C.C.	MaxOft	W.D.	M.R.	H.I.	R.W.
Desktop	1	H	732604, 15423723	732604, 15423723	5328	45786, 32	29304, 32	732603, 500	45787	11445	244200, 2	293041
	2	VBox G	8275, 71094	8275, 562	✖	516, 32	330, 16	8275, 500	517	128	2757, 2	3310
	3	H + VBox G	670618, 13876669	670618, 562	4877	41912, 32	26823, 16	670618, 500	41913	10477	223538, 2	268247
	4	H + VBox G	7125068, 236516701	7125068, 3162	51818	445315, 32	285001, 16	7125068, 500	445316	111328	2375022, 2	2850027
	5	VBox G + G (reset)	13152, 123966	13152, 3162	✖	821, 32	525, 16	13152, 500	822	204	4383, 2	5261
Laptop	6	H + VBox G	60266, 770197	60266, 770197	✖	3765, 32	2409, 16	60266, 500	3766	940	20088, 2	24106
	7	H + VBox G	94277, 1317655	94277, 3162	✖	5891, 32	3770, 16	94277, 500	5892	1472	31425, 2	37711
Xen	8	XCP G	46896, 570004	46896, 3162	✖	2930, 32	1874, 16	46896, 500	2931	731	15631, 2	18758
	9	XCP G	218380, 3610525	218380, 3162	✖	13647, 32	8734, 16	218380, 500	13648	3411	72792, 2	87352
	10	XCP G + G	49024, 60118	49024, 601181	✖	3063, 32	1960, 16	49024, 500	3064	765	16340, 2	19610
	11	XCP G + G	365070, 6689079	365070, 3162	✖	22815, 32	14601, 16	365070, 500	22816	5703	121689, 2	146028
Amazon	12	EC2 G	10815, 98027	10815, 3162	✖	675, 32	431, 16	10815, 500	676	168	3604, 2	4326
	13	19 EC2 G	581326, 11690112	581326, 3162	4559	36331, 32	23252, 16	581326, 500	36332	9082	193774, 2	232530
Hosts (14)			8551200, 294406554	8551200, 3162	66288	534449, 32	342047, 16	8551200, 500	534450	133611	2850399, 2	3420480
Guests (15)			1424583, 34272067	1424583, 3162	11043	89035, 32	56982, 16	1424583, 500	89036	22258	474860, 2	569833
Hosts and Guests (16)			9975784, 354202353	9975784, 3162	77331	623485, 32	399030, 16	9975784, 500	623486	155870	3325260, 2	3990314

This battery requests almost 2.0×10^8 random numbers that are distributively used by several individual tests. This battery was applied to the dataset of test number 16 (all the datasets merged), which passed the tests. Although the size of this dataset is not sufficient (almost 2.0×10^7), the samples are not reused since each test did not request a larger amount of numbers for most cases. Some tests reused samples, but even so they passed with p-values falling within the confidence interval. The remaining datasets were not further tested on `bbattery_pseudoDIEHARD` since it would not be complementarily useful and because of the smaller sizes of the respective datasets. These preliminary tests already provide an idea of the results for the more stringent tests analyzed in the next section.

4.6.3 Analysis of the p-values

Table 4.5 depicts all the p-values resulting from the execution of all the statistics included in the customized TestU01 battery, configured accordingly to the parameters in Table 4.3 and Table 4.4, on the mergers of all of the datasets collected. Note that the *RandomWalk1* test withholds five sub-statistics denoted by *H*, *M*, *J*, *R*, and *C*. Each of these outputs a p-value every two steps for a given walk length interval. The walk length interval was set to ten, meaning a total of six runs result in 30 p-values, five per sub-statistic. For the sake of simplification, Table 4.5 contains averaged p-values grouped per each of the sub-statistic. Any test failing or outputting a suspect p-value is marked with an asterisk next to the p-value itself. In the case of the averaged p-values, an asterisk means that particular group includes a suspect p-value or a failing one for a random walk of length $L_0 \geq l \geq L_1$, where $l \mod 2 = 0$ for all L_0 and L_1 even.

A total of three suspect p-values and zero p-values failing decisively some dataset were found. The results on the table are clear. The outputs of the `/dev/random` device are of high quality and are uncorrelated throughout all the datasets, including those of co-resident guests and of a guest and a host. These defeat the preliminary results outputted by the less stringent batteries, portraying quality random numbers on both small and large datasets. Except for the suspect ones, all the p-values observed are considered normal for the underlying confidence interval. Despite the gathering of entropy is done more slowly on any VM of any hypervisor, the entropic input set lumped together by the Linux RNG is apparently sufficiently heterogeneous so that the `/dev/random` device generates random material of high randomness quality. Alternatively, the construction by means of the hashing and mixing functions with feedback of the Linux RNG may be robust and resilient enough to generate quality random material for these cases.

Tests number 2, 3, 6, and 7 empirically prove that numbers are uncorrelated among guests and underlying hosts, regardless of the shared environment. Moreover, test number 5 indicates that a single VM reset does not undermines the random number generation on Linux guests. Tests number 10 and 11 empirically prove that numbers are uncorrelated on co-resident guests. The results of the tests with a single guest also tell the random numbers are of high quality for both

Table 4.5: Results of the p-values outputted by the customized TestU01 battery with regard to tests number 1 through 16.

	Tests		B.S.	Col.	Gap	S.P.	C.C.	MaxOft		W.D.	M.R.	HammingIndep			RandomWalk1				
															H	M	J	R	C
Desktop	1	H	0.08	0.98	0.11	0.89	0.85	0.9936	0.14	0.16	0.93	0.21	0.52	0.95	0.37	0.65	0.19	0.49	0.89
Desktop	2	VBox G	5.1×10^{-3}	0.88	✖	4.7×10^{-3}	0.14	0.78	0.95	0.93	0.28	0.89	0.60	0.61	0.86*	0.78	0.44	0.99	0.42
Desktop	3	H + VBox G	0.18	0.40	0.58	0.18	0.50	0.79	0.83	0.46	0.53	0.98	0.83	0.78	0.79	0.65	0.20	0.47	0.30
Desktop	4	H + VBox G	0.98	0.81	0.80	0.34	0.96	0.96	0.01	0.73	0.37	0.46	0.46	0.34	0.27	0.31	0.72	0.39	0.55
Desktop	5	VBox G + G (reset)	0.28	0.50	✖	0.64	0.23	0.10	0.64	0.43	0.99	0.42	0.15	0.60	0.50	0.15	0.19	0.67	0.94
Laptop	6	H + VBox G	0.10	0.32	✖	0.80	0.10	0.36	0.17	0.05	0.84	0.82	0.04	0.76	0.43	0.06*	0.36	0.94	0.82
Laptop	7	H + VBox G	0.43	0.89	✖	0.79	0.07	0.70	0.34	0.45	0.35	0.71	0.97	0.54	0.48	0.72	0.12	0.53	0.19
Xen	8	XCP G	0.74	0.20	✖	0.11	0.92	0.39	0.85	0.46	0.15	0.64	0.92	0.64	0.18	0.59	0.73	0.52	0.21
Xen	9	XCP G	0.83	0.63	✖	0.12	0.93	0.65	0.97	0.76	0.25	0.87	0.87	0.92	0.69	0.28	0.56	0.42	0.94
Xen	10	XCP G + G	0.67	0.87	✖	0.53	0.22	0.04	0.80	0.49	0.34	0.29	0.44	7.9×10^{-3}	0.51	0.31	0.49	0.65	0.86
Xen	11	XCP G + G	0.41	0.66	✖	0.73	0.61	0.70	0.63	0.82	0.29	0.10	0.40	0.10	0.56	0.71	0.46	0.58	0.94
Amazon	12	EC2 G	0.52	0.24	✖	0.86	0.82	0.94	0.23	0.15	0.04	0.64	0.90	0.05	0.31	0.07	0.20	0.14	0.15
Amazon	13	19 EC2 G	0.04	0.89	0.97	0.20	0.86	0.60	0.69	0.92	0.70	0.59	0.87	0.77	0.35	0.44	0.59	0.76	0.92
Hosts (14)			0.95	0.63	0.48	0.74	0.40	0.96	0.43	0.87	0.80	0.16	0.14	0.29	0.70	0.78	0.69	0.30	0.59
Guests (15)			0.95	0.58	0.98	0.01	0.07	0.32	0.9995*	0.77	0.22	0.93	0.90	0.44	0.60	0.72	0.63	0.05	0.30
Hosts and Guests (16)			5.8×10^{-3}	0.71	0.04	0.83	0.62	0.97	0.94	0.77	0.66	0.94	0.70	0.79	0.52	0.35	0.17	0.82	0.45

native and hosted hypervisors. It can be said that the resiliency of the Linux RNG is put to more rigorous tests when the context is EC2. The slowness found in this cloud is concerning, but no randomness cripple effect is seen. In fact, the p-values fall within the normal set for sequences showing good randomness properties.

4.6.4 Biasing the Tests

In order to find out which portion of correlated numbers would be necessary to trigger lots of failures in TestU01, a program was written in C programming language to purposely inject correlated numbers into the datasets. This experiment allowed to obtain a higher confidence on the results regarding the quality of the outputs in terms of randomness discussed previously. Basically, the program takes as input a percentage p of the numbers desired to be correlated for any given input sample size and then writes to a new file the biased dataset. All biased datasets are again submitted to the customized battery of tests. A given dataset file is looped until all the samples have been processed. For each sample sam of a given dataset, a random number is generated within the interval $[0,\ 1)$, and if it falls within $[0,\ p]$, sam is multiplied by a prime number and is written to the output file together with the result of this operation. In this case, the prime number was set to 13. The Mersenne Twister algorithm (available at [Nis]) was used as means to generate high quality and uniformly distributed pseudo random numbers.

It was found that most datasets started to fail to only approximately 5% of biased samples. The dataset of test number 16 contains a total 19951570 samples, from which 2849158 are all of the aggregated guest samples. This corresponds to approximately 14.28% of the total number of samples. According to the experiment, such an amount would be sufficient for triggering failures, if such would be the case on a hypothetical scenario of having correlated series. The tests with the setups involving co-resident guests, hosts and guests, and the merger on test number 15 would probably fail too. Even thought the size of the samples is not the most appropriate for a higher statistical stringency, the confidence of the results presented in this section regarding the randomness quality is not undermined. This is because TestU01 would find statistical flaws given just a small percentage of correlated numbers.

4.7 Conclusions

This chapter presented the tests performed to achieve some of the most important objectives of this research. The results of those tests were presented herein as well. The discussion flows to the implications that virtualization brings to random number generation. Specifically, such implications are important for the Linux RNG for two different reasons: (i) the speed of the random number generation and (ii) the quality of the outputs, namely regarding the possibility of the sequences produced in two co-resident VMs to be correlated. It was found that the Linux RNG is reasonably slow making entropy available to the `/dev/random` device for outputting quality random jargon. This can have a negative impact on several applications. The case study

of the GPG application as shown in Section 4.5 to demonstrate this issue.

The efficient generation of high quality and completely random numbers is critical for a number of areas, notably cryptography. Creating high quality cryptographic material depends on the entropy source. The `/dev/random` device provides highly secure random data that can serve such purpose. Although the datasets used in this dissertation were small, because of aforementioned reasons, the results obtained from their empirical analyzes show that the outputs of `/dev/random` maintain their quality with respect to their randomness, even when run on OSes within VMs. Since the analyzes did not reveal any apparent flaw on the statistical structure of the random numbers, it can be said they are independently and uniformly distributed. It is therefore possible to confidently say the cryptographic material generated from these random numbers is of high quality as well. It is also important to refer that the fact that the VMs were sharing the same underlying hardware infrastructure did not result in correlated sequences between co-resident VMs. In other words, co-resident VMs generated random material independently of each other. Given this particular conclusion, cloud adopters may rest assure that the cryptographic material of their VMs is not similar to the one of co-resident VMs. The merging of the datasets of the guests and of the hosts also corroborate that those random numbers are uncorrelated. Hypervisors make up an abstraction barrier, separating the VMs from each other and from the hardware, at least in terms of the events that contribute for entropy gathering. Nonetheless, this conclusion may not hold if some kind of provisioning is put in place, like the KVM VirtIO RNG driver that wires host randomness sources to guests.

Chapter 5

Conclusions and Future Work

This final chapter is organized in three sections. The first section contains some final remarks concerning the research work presented in this dissertation, while the second section contains the main conclusions. The third, and last section, suggests research directions that may be addressed in the future.

5.1 Final Remarks

The outcome of the work presented in this dissertation corroborated that the Linux RNG is not effective in gathering entropy on VMs. As a consequence, the `/dev/random` device is negatively impacted and blocks quite often, waiting for the entropy refills to take place into the blocking pool, which only come once in a while. Even before the advent of cloud computing, the community proposed solutions for addressing entropy starvation for more demanding applications. More recent solutions are motivated by the problem on virtualized environments.

Although less preferred, the simplest approach consists in feeding `/dev/random` with something gathered from an external source or by redirecting `/dev/urandom` outputs into `/dev/random`. External sources include the ones based on hardware devices commonly deployed on external appliances such as Universal Serial Bus (USB) sticks (*e.g.*, Entropy Key [Simb]) that may require some third-party driver or application. A single device may be rather expensive as well, and thus deploying these at a large scale may be quite an investment. In addition, their compatibility and portability across multiple OSes might be limited. As such, one would prefer a hardware-based RNG seamlessly integrated within every computing device. In addition, the good thing about entropy is that, when it is gathered locally, it is done in a unique way and secretly, and thus it should only rely on the local system.

The HArdware Volatile Entropy Gathering and Expansion (HAVEGE) [SS03] RNG handles modern CPU constructions for exploiting the internal volatile hardware states as a source of entropy. The algorithm is initialized by the hardware clock cycle counter of the processor to gather some entropy. In average, this source of entropy provides tens of thousands of unpredictable bits per OS system call. It is available as a Linux kernel module. The CPU jitter RNG [M¨] is a non-physical TRNG built with the intent of overcoming the limitations of RNGs running on VMs. It utilizes an entropy source based on CPU execution time jitter that, in part, is also based on the CPU assumptions of HAVEGE. On VMs, it should deliver high-quality entropy on

both user- and kernel-space. It generates 64-bit random numbers at each request and provides perfect backward and forward secrecy. On Linux, access to the CPU high resolution timer is granted to unprivileged processes, and thus each application can deploy its own instance of the CPU jitter RNG. The outputs of HAVEGE and the CPU jitter RNG should be used as seeds for other PRNGs. Furthermore, the Entropy Gathering Daemon (EGD) [War] is a Perl script meant to replace `/dev/random` on systems that do not have it as a source for randomness. It executes as a background process, runs dummy commands to create entropy and collects the randomness of that execution for stirring an entropy pool.

The Asynchronous Network Exchanged Randomness Daemon (ANERD) [Kir] is a network protocol designed to provide entropy to VM clients by means of an external server. When the ANERD server receives an entropy request, it hashes the data received and salts it with the sum of all of the timestamps from data being received since the server started. The client injects the entropy to any specified device. There is a version of this protocol which serves entropy over a TLS connection. There is also the Entropy Broker [vH13] protocol. Entropy Broker functions like an amalgamation of several randomness sources to distributively serve multiple entropy consumers through a centralized mediator and management server. An arbitrary number of randomness sources, like the aforementioned ones, can be utilized to gather entropy. The same author of Entropy Broker has developed daemons for generating entropy, namely the `timer_entropyd`, the `audio_entropyd`, and the `video_entropyd`. The first measures how much longer or shorter a sleep takes, which jitters due to the frequency of the clocks of the timers; the second fetches two images from a video device and calculates the difference between these two; finally, the third de-biases data read from an audio device. All randomness extracted is first analyzed accordingly to the amount of entropy present in those bits, which can be then provided to `/dev/random`. The `audio_entropyd` and the `video_entropyd` may not be suitable for virtualized OSes, but the `timer_entropyd` is usable within VMs.

In terms of cloud scenarios, hypervisors could deploy a feature to provision PRNG seeds and entropy to guests, as suggested by Kirkland [Kir12]. OpenStack has addressed [Mos12] the former. Concerning the latter, the QEMU virtualizer and emulator provides the VirtIO RNG driver for the KVM hypervisor. It allows to wire external entropy sources to the host and then expose randomness to guests via `/dev/hwrng`. Any aforesaid daemon or protocol or even the host RNG can be used as sources in this case. Part of the literature on the security of hypervisors aims at hardening [LSLS09] or minimizing [MLQ+10, HS12] such complex software for more robust features and avoid vulnerabilities, respectively. There is also research on rethinking virtualization by eliminating VM dependency on the hypervisor. This is the case of NoHype [SKLR11]. The system maintains VM concurrency while providing them with more direct contact with the underlying hardware. In terms of collection of kernel events for entropy purposes, it is true that the Linux RNG could generate greater amounts of random material that it does with current

commercial hypervisors, but it would also mean that co-resident VMs would share underlying hardware more directly. This can lead to starvation attacks or correlated material.

On the horizon, a growing paradigm may shift the concept of random number generation under the context of deterministic machines. Things at the microscopic level are different from the reality people are used to. This is yet a thoroughly unexplored world that aims at blowing up the quantumness to larger sizes in a highly controllable way to make up quantum computers. One of the problems is that quantum information has an intrinsic randomness and uncertainty associated with it. It is a highly entropic environment. That may sound painful for the physicist, but for random number generation, it opens up a whole new world of immense possibilities based on harnessing that unpredictable behavior, not to mention the fundamental changes it would pave for computer science regarding computing perceptions. Quantum computing is based on qubits, which are unpredictable states of information, though quantum computers do not maintain a memory state. Quantum computers would allow to solve problems considered unfeasible to solve with classical computers. Nonetheless, they are still far from practical deployment.

5.2 Main Conclusions

The industry is a non-stop engine carving for better technologies that, from time to time, come to revolutionize the computer science world. This is the case of *cloud computing*, a buzzword that is on the hype at the moment, which is one of the topics of interest for the academia and industry. Section 2.2 detailed the cloud computing model, telling how the cloud uptake has revolutionized how the industry conducts businesses when it comes to IT. Enterprises adopting cloud solutions lower their infrastructure costs for the installation, configuration and management. This allows them to focus on promoting the corporation and the business segment by abstracting underlying details of the IT infrastructures. Cloud computing is mainly available in three service delivery flavors over three deployment models. The broad SaaS, PaaS, and IaaS models shape clouds to deliver particular XaaS solutions that can be deployed over private, public, or hybrid settings.

In spite of the aforementioned benefits, the emergence of cloud computing has introduced novel security issues, as the ones pointed out in Section 2.3. The virtualization layer imposed by the hypervisors has brought computer science a step closer to the long-envisioned era of utility computing given by the IaaS clouds. Instancing VMs on the minute with few resources is extremely useful for many purposes. This shift in computing perceptions eliminates the costs and burden of installing physical servers that could take from a few days to weeks in the classical computing model of physical servers and introduces a seamless operation for mounting virtual data centers. But virtualization implies an arbitrary number of VMs to be co-resident, with each being probably owned by different customers on an outsourced cloud setting. This has

introduced the danger of cross-VM attacks, but also the possibility of insider threats, because customers outsource their potentially sensitive data. This dissertation focused particularly on the problem of entropy gathering on Linux OSes when run on VMs, which is magnified by virtualization technology.

As explained throughout Section 2.4, entropy is the most important ingredient in the process of creating high-quality random bytes suitable for cryptographic applications. Random numbers are important for several other fields and areas of knowledge as well, namely for modeling, simulation, physics, and finances. The Linux kernel gathers bits of entropy for mixing a pool of random bits that are then used to generate quality random data with its embedded RNG. This process is performed by the kernel and it is sourced from several physical hardware devices connected to the hardware on which the Linux OS is installed upon. However, virtualization abstracts overlying guests and distances them from the underlying hardware, to which the access is restricted and controlled by the hypervisor. As such, the Linux RNG, particularly the `/dev/random` device, may output weaker random material at a slow pace.

This dissertation has studied the outputs of the `/dev/random` device of the Linux RNG when run within VMs. Since the Linux RNG may be crippled in throughput and output quality, the aim of the work presented in this dissertation was to audit the throughput efficiency and the randomness quality of the random numbers, on both guests and hosts. This assessment allows to extract the most crucial picture of the security implications such potential drawbacks can have to cryptographic material. Chapter 3 introduced a simple method that was adopted to accomplish this work. For quantifying the randomness quality of sequences of random numbers, the most suitable means is to submit the numbers to stringent statistical tests. Since the analysis of the throughput efficiency requires post-processing and since it is unfeasible to test a TRNG directly on statistical tests, it was necessary to first harvest various samples of random numbers generated by the Linux RNG. The collection of samples is composed by files with the timestamps on which the `/dev/random` device unblocked for reading and the associated four-byte unsigned integers. The files were then processed and analyzed from two different perspectives: the timestamps were analyzed to emphasize interesting characteristics of the timing of the generation method by means of various metrics, while the random numbers were analyzed to find statistical weaknesses by means of the TestU01 statistical library. The tests and their results are included in Chapter 4. The tests undertaken were executed on several IaaS cloud setups using different hypervisors and hardware, so as to study the problem more broadly and add confidence to the results. The main empirical findings of this dissertation are twofold. Firstly, the Linux RNG is extremely slow gathering entropy on guests, when compared to the host counterparts. Secondly, the random bytes outputted by the Linux RNG are of high quality, in terms of randomness. Amazon EC2 turned out to be really slow harvesting entropy bits to the Linux RNG. The same pattern was observed among almost all the instances fired up in the cloud,

showing a very similar behavior in terms of the amount of four-byte random numbers generated per instant, the time differences between such generation and, consequently, the rate of the generation. The `/dev/random` device also blocks quite often on other hypervisors. Those metrics were also noticeable on other tests executed on both VirtualBox and Xen guests, regardless of the cloud setup. TestU01 searches for potential undermined random numbers by analyzing the statistical structure of the sequences of random numbers as a whole. In other words, it grinds a generator thoroughly by requesting a very large amount of random numbers and executing the statistical tests over those numbers. All the random number sequences sampled throughout the work in this dissertation are of high quality. These include the numbers obtained from hosts and guests, including the ones that ran singularly and co-resident. Additionally, the statistical tests revealed no correlation between numbers generated on co-resident guests, despite the underlying environment being the same, which is perhaps the most important conclusion of this dissertation.

Both the industry and the academia bespeak a clear interest in addressing cloud security issues. As the field matures, a better understanding of the cloud computing paradigm will eventually come to place. It is expected to see this mainstream technology settle down, converging the current diversity into more streamlined and robust approaches. From a security viewpoint, this path may allow to dissipate doubts on the technology and ultimately leverage its most distinctive traits on secure computing environments.

5.3 Directions for Future Work

There are several lines of research that can be pursued for the main topics addressed in this dissertation, namely the ones related with entropy gathering and with the outputs of `/dev/random`. For a start, it would be interesting to monitor entropy levels at both hosts and guests while requesting random data from the `/dev/random` device, just to search for potential relations between the amount of entropy available and the total size of the outputs. This would also aid in studying the efficiency of the Linux kernel to gathering entropy, and therefore settle more robust foundations justifying the results herein obtained. In addition, it would also be interesting to investigate how the kernel behaves on VMs in terms of the types of events that are taken into consideration. Categorizing the events according to the source would provide deeper insight and enable to point out unattended randomness sources within virtual environments, or even new ones that the hypervisors may use. Performing this work on the host would complementarily allow to distinguish or find replicated events in both of them, if they exist. Since IaaS clouds are most likely to be within some data center, installed over a server infrastructure, it is probable that the OSes are subject to an even more restricted set of entropic inputs due to the lesser noise sources. As such, it would also be interesting to study the Linux RNG on production servers devoid of mouse, keyboard, and other peripherals, but with SSD disks. Investigating which events various hypervisors might cache and how that affects entropy gathering,

and ultimately the generation of random material out of /dev/random, also follows those aforementioned lines of research. In any case, it should be paid attention to the construction of the Linux RNG, in order to determine if a smaller and potentially less heterogeneous entropic set can indeed impact the randomness quality.

Ultimately, the lessons learned by studying the points highlighted above could help devise a means to overcome the entropy starvation problem on VMs. Such a means would be required to source new entropy from nothing else but the own system, and produce a fair amount of entropic bits that would be used by /dev/random to output random numbers of high randomness quality. For carrying out these studies, the source code of the Linux kernel would have to be changed for monitoring purposes, namely the random.c file (under the linux/drivers/char/ kernel source code tree path) that implements the Linux RNG.

Testing the resilience of the Linux RNG in terms of efficiency and quality of the outputs on more cloud scenarios, as the ones described above, would definitely help clarify the problem addressed in this dissertation. In the hypothetical case where statistical flaws are found under diversified cloud setups, then such technical motives would be reasonable to encourage industry players to patch the problem at the virtualization level. Answering the questions on these possible lines of work would essentially help demystify the puzzling problem of generating random material out of the Linux RNG when run on IaaS clouds and, by that, a deeper knowledge of the security on cloud computing environments would be obtained, particularly on the quality of cryptographic material.

References

[Amaa] Amazon. Amazon EC2 Instances. Available in https://aws.amazon.com/ec2/instance-types/. Accessed Sep. 2013. 49

[Amab] Amazon. Amazon EC2 Website. Available in https://aws.amazon.com/ec2/. Accessed Jul. 2013. 11

[Amac] Amazon. Amazon VPC Website. Available in https://aws.amazon.com/vpc/. Accessed Jul. 2013. 13

[Ama11] Amazon Web Services Discussion Forums. Low Entropy on EC2 instances - Problem for anything related to security. Available in https://forums.aws.amazon.com/thread.jspa?messageID=249079, May 2011. Accessed Apr. 2013. xvii, 30

[Amo13] Edward G. Amoroso. From the Enterprise Perimeter to a Mobility-Enabled Secure Cloud. *IEEE Secur. Privacy*, 11(1):23–31, 2013. 19

[Apa] Apache. CloudStack Website. Available in https://cloudstack.apache.org/. Accessed May 2013. 20

[AZB13] Everaldo Aguiar, Yihua Zhang, and Marina Blanton. An Overview of Issues and Recent Developments in Cloud Computing and Storage Security. In *High Performance Semantic Cloud Auditing*, pages 1–31. Springer, 2013. xvi, 17

[BNP+11] Sven Bugiel, Stefan Nürnberger, Thomas Pöppelmann, Ahmad-Reza Sadeghi, and Thomas Schneider. AmazonIA: When Elasticity Snaps Back. In *Proc. of the 18th ACM Conf. on Computer and Communications Security*, pages 389–400, New York, NY, USA, Oct. 2011. ACM. 18

[Bro] Robert G. Brown. Dieharder: A Random Number Test Suite. Available in http://www.phy.duke.edu/~rgb/General/dieharder.php. Accessed Sep. 2013. 35

[BRPB13] Georg T. Becker, Francesco Regazzoni, Christof Paar, and Wayne P. Burleson. Stealthy Dopant-Level Hardware Trojans. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems*, volume 8086 of *Lecture Notes in Computer Science*, pages 197–214. Springer Berlin Heidelberg, 2013. Available from: http://dx.doi.org/10.1007/978-3-642-40349-1_12. 22

[CAM+08] Xu Chen, Jon Andersen, Z. M. Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Proc. of the IEEE Int. Conf. on Dependable Systems and Networks (DNS) With FCTS and DCC*, pages 177–186, Anchorage, AK, USA, Jun. 2008. 19

[Car12] Pedro Carriço. Low entropy on VMs.... Available in http://blog.pedrocarrico.net/post/17026199379/low-entropy-on-vms, Feb. 2012. Accessed May 2013. xvii, 30

[Cis13] Cisco. 2013 Cisco Annual Security Report. Available in http://www.cisco.com/en/US/prod/vpndevc/annual_security_report.html, 2013. Accessed Apr. 2013. 14

[Cit] Citrix. Citrix Products Website. Available in https://www.citrix.com/products.html. Accessed Jun. 2013. 17

[Cle13] James Clewett. Numberphile: Random Numbers. Available in http://www.numberphile.com/videos/random_numbers.html, Apr. 2013. Accessed May 2013. 22

[Con07] Robert Connolly. Entropy and random number generators in Linux. Available in http://www.linuxfromscratch.org/hints/downloads/files/entropy.txt, May 2007. Accessed May 2013. 21

[CSA13] CSA. Privacy Level Agreement Outline for the Sale of Cloud Services in the European Union. Available in https://downloads.cloudsecurityalliance.org/initiatives/pla/Privacy_Level_Agreement_Outline.pdf, Feb. 2013. Accessed Aug. 2013. 18

[CYG+11] Chao-Chih Chen, Lihua Yuan, Albert Greenberg, Chen-Nee Chuah, and P. Mohapatra. Routing-as-a-Service (RaaS): A Framework For Tenant-Directed Route Control in Data Center. In *Proc. of the 30th IEEE Int. Conf. on Computer Communications (INFOCOM)*, pages 1386–1394, Dec. 2011. 10

[Dav96] Don Davis. Need Advice: RNG. Available in http://www.cs.berkeley.edu/~daw/rnd/mouse-pitfalls, Oct. 1996. Accessed May 2013. 28

[dB13] Florence de Borja. Nebula One Seeks To Reinvent Cloud Computing. Available in http://cloudtimes.org/2013/04/18/nebula-one-reinvent-cloud-computing/, Apr. 2013. Accessed Jun. 2013. 14

[Duc12] Paul Ducklin. Android random number flaw implicated in Bitcoin thefts. Available in http://nakedsecurity.sophos.com/2013/08/12/android-random-number-flaw-implicated-in-bitcoin-thefts/, Aug. 2012. Accessed Aug. 2013. 25

[EFF13] EFF. How the NSA's Domestic Spying Program Works. Available in https://www.eff.org/nsa-spying/how-it-works, 2013. Accessed Sep. 2013. 18

[Ell] Carl Ellison. Cryptographic Random Numbers. Available in http://theworld.com/~cme/P1363/ranno.html. Accessed Sep. 2013. 21, 27

[FMM+11] Natalia Castro Fernandes, Marcelo D. D. Moreira, Igor M. Moraes, Lyno Henrique G. Ferraz, Rodrigo S. Couto, Hugo E. T. Carvalho, Miguel Elias M. Campista, Luís Henrique Maciel Kosmalski Costa, and Otto Carlos Muniz Bandeira Duarte. Virtual networks: isolation, performance, and trends. *Annales des Télécommunications*, 66(5-6):339–355, 2011. 12

[FSFI13] Diogo A. B. Fernandes, Liliana F. B. Soares, Mário M. Freire, and Pedro R. M. Inácio. Randomness in Virtual Machines. In *Proc. of the 6th IEEE/ACM Int. Conf. on Utility and Cloud Computing (UCC)*, Dresden, Germany, Dec. 2013. IEEE Computer Society. Accepted for publication. xvi, 6

[FSG+13a] Diogo A. B. Fernandes, Liliana F. B. Soares, João V. Gomes, Mário M. Freire, and Pedro R. M. Inácio. A Quick Perspective on the Current State in Cybersecurity. In Babak Akhgar and Hamid R. Arabnia, editors, *Emerging Trends in Information and Communication Technologias Security*, pages 423-441. Elsevier (Morgan Kaufmann), Burlington, Massachusetts, USA, 2013. In press. xvi, 7

[FSG+13b] Diogo A. B. Fernandes, Liliana F. B. Soares, João V. Gomes, Mário M. Freire, and Pedro R. M. Inácio. Security Issues in Cloud Environments — A Survey. *Int. J. Inf. Secur.: Security in Cloud Computing*, 2013. Available from: http://link.springer.com/article/10.1007%2Fs10207-013-0208-7. xv, xvi, xxvii, 6, 10, 11, 14, 17

[Gal97] James P. Gallegos. Mouse position in button. Available in http://www.cs.berkeley.edu/~daw/rnd/more-mouse-pitfalls, May 1997. Accessed May 2013. 28

[Gal13] Ryan Gallagher. Cryptographers Attack NSA's Secret Effort to Subvert Internet Security. Available in http://www.slate.com/blogs/future_tense/2013/09/16/cryptographers_attack_nsa_s_secret_effort_to_subvert_internet_security.html, Sep. 2013. Accessed Sep. 2013. 18

[GIP+13] João V. Gomes, Pedro R.M. Inácio, M. Pereira, Mário M. Freire, and Paulo P. Monteiro. Identification of Peer-to-Peer VoIP Sessions Using Entropy and Codec Properties. *IEEE Transactions on Parallel and Distributed Systems*, 24(10):2004-2014, 2013. 27

[Goo] Google. Google App Engine Website. Available in https://developers.google.com/appengine/. Accessed Apr. 2013. 12

[GPR06] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux Random Number Generator. In *Proc. of the IEEE Symp. on Security and Privacy*, pages 371-385, Oakland, CA, USA, May 2006. IEEE Computer Society. xiii, xxvii, 3, 26, 27

[Gre13] Matthew Green. The Many Flaws of Dual_EC_DRB. Available in http://blog.cryptographyengineering.com/2013/09/the-many-flaws-of-dualecdrbg.html, Sep. 2013. accessed Sep. 2013. 18

[GW96] Ian Goldberg and David Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal*, Jan. 1996. Accessed Aug. 2013. 25

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Minding Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *Proc.*

of the 21st USENIX Security Symp., pages 205–220, Bellevue, WA, USA, Aug. 2012. USENIX. xvii, 4, 30

[HS12] Jingyu Hua and Kouichi Sakurai. Barrier: A Lightweight Hypervisor For Protecting Kernel Integrity via Memory Isolation. In *Proc. of the 27th Annual ACM Symp. on Applied Computing (SAC)*, pages 1470–1477, Trento, Italy, Mar. 2012. ACM. 18, 64

[Int] Intel. User Manual for the Rdrand Library (Linux* and OS X* Version). Available in hhttp://software.intel.com/en-us/articles/user-manual-for-the-rdrand-library-linux-version. Accessed Sep. 2013. 49

[Int12] Intel. Intel Digital Random Number Generator (DRNG): Software Implementation Guide. Available in http://software.intel.com/sites/default/files/m/d/4/1/d/8/441_Intel_R__DRNG_Software_Implementation_Guide_final_Aug7.pdf, Aug. 2012. Accessed May 2013. 22

[KC12] Brendan Kerrigan and Yu Chen. A Study of Entropy Sources in Cloud Computers: Random Number Generation on Cloud Hosts. In *Proc. of the 6th Int. Conf. on Mathematical Methods, Models and Architectures for Computer Network Security (MM-M-ACNS)*, pages 286–298, St. Petersburg, Russia, Oct. 2012. Springer-Verlag. xiv, xvii, 4, 30

[Kin12] Colin Ian King. Intel rdrand instruction revisited. Available in http://smackerelofopinion.blogspot.co.uk/2012/10/intel-rdrand-instruction-revisited.html, Oct. 2012. accessed May 2013. 22

[Kir] Dustin Kirkland. "anerd" package in Ubuntu. Available in https://launchpad.net/ubuntu/+source/anerd. Accessed Sep. 2013. 64

[Kir12] Dustin Kirkland. Entropy (or rather the lack thereof) in OpenStack instances... and how to improve that. Available in http://www.openstack.org/summit/san-diego-2012/openstack-summit-sessions/presentation/entropy-or-lack-thereof-in-openstack-instances, Oct. 2012. Accessed Jul. 2013. xiv, xvii, 30, 64

[KSF99] John Kelsey, Bruce Schneier, and Niels Ferguson. Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In *Proc. of the 6th Annual Int. Workshop on Selected Areas in Cryptography (SAC)*, pages 13-33, London, UK, UK, 1999. Springer-Verlag. 24

[Kuc] Andrew Kuchling. python-gnupg - A Python wrapper for GnuPG Website. Available in http://pythonhosted.org/python-gnupg/. Accessed Sep. 2013. 55

[KVM] KVM. Kernel for guest with paravirtualization. Available in http://www.linux-kvm.org/page/Tuning_Kernel. Accessed Apr. 2013. 30

[LRSV12] Patrick Lacharme, Andrea Röck, Vincent Strubel, and Marion Videau. The Linux Pseudorandom Number Generator Revisited. *IACR Cryptology ePrint Archive*,

2012:245, 2012. Informal publication. Available from: http://dblp.uni-trier.de/db/journals/iacr/iacr2012.html#LacharmeRSV12. xiii, xxvii, 3, 26, 27, 28

[LS07] Pierre L'Ecuyer and Richard Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *ACM Trans. Math. Softw.*, 33(4), Aug. 2007. xvii, 23, 34, 39, 56

[LSLS09] Fagui Liu, Xiang Su, Wenqian Liu, and Ming Shi. The Design and Application of Xen-based Host System Firewall and its Extension. In *Proc. of the Int. Conf. on Electronic Computer Technology*, pages 392-395, Macau, China, Feb. 2009. 18, 64

[M̈] Stephan Müller. CPU Jitter Random Number Generator Website. Available in http://www.chronox.de/. Accessed Sep. 2013. 63

[Mar] George Marsaglia. The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. Available in http://www.stat.fsu.edu/pub/diehard/. Accessed Sep. 2013. 34

[Mar68] George Marsaglia. RANDOM NUMBERS FALL MAINLY IN THE PLANES. *Proc. of the National Academy of Sciences*, 61(1):25–28, Sep. 1968. 23

[Mec] John Mechalas. User Manual for the Rdrand Library (Windows* Version). Available in http://software.intel.com/en-us/articles/user-manual-for-the-rdrand-library-windows-version. Accessed Sep. 2013. 49

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Available in http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, Sep. 2011. Accessed Sep. 2012. 10

[Mic] Microsoft. Hyper-V Server Website. Available in https://www.microsoft.com/en-us/server-cloud/hyper-v-server/. Accessed Jun. 2013. 17

[Mic13] Microsoft. Small and midsize companies in the cloud reap security, privacy and reliability benefits. Microsoft News Center, Jun. 2013. Accessed Aug. 2013. 20

[Mit13] David Mitchell. VM-to-VM Traffic No Longer a Security Blind Spot. Available in https://blogs.rsa.com/vm-to-vm-traffic-no-longer-a-security-blind-spot/, Aug. 2013. Accessed Aug. 2013. 19

[MLQ+10] Jonathan J. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proc. of the IEEE Symp. on Security and Privacy (SP)*, pages 143-158, Oakland, CA, USA, May 2010. 18, 64

[Mos12] Scott Moser. Change I7d8c1f9b: add 'random_seed' entry to instance metadata. Available in https://review.openstack.org/#/c/14550/, Oct. 2012. accessed May 2013. 64

[Neb13] Nebula. Introducing Nebula One. Available in https://www.nebula.com/nebula-one, Apr. 2013. Accessed Apr. 2013. 14

[Nis] Takuji Nishimura. MT19937: C Program. Available in http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/980409/mt19937-1.c. Accessed Sep. 2013. 44, 61

[NIS13] NIST. NIST Cloud Computing Security Reference Architecture. Available in http://collaborate.nist.gov/twiki-cloud-computing/pub/CloudComputing/CloudSecurity/NIST_Security_Reference_Architecture_2013.05.15_v1.0.pdf, Jun. 2013. Accessed Jul. 2013. 20

[OCC] OCCI. OCCI Website. Available in http://occi-wg.org/. Accessed Apr. 2013. 20

[OGC+12] Yoshihiro Oyama, Tran Truong Duc Giang, Yosuke Chubachi, Takahiro Shinagawa, and Kazuhiko Kato. Detecting Malware Signatures in a Thin Hypervisor. In *Proc. of the 27th Annual ACM Symp. on Applied Computing (SAC)*, pages 1807–1814, Trento, Italy, Mar. 2012. ACM. 19

[Ope] OpenNebula. OpenNebula Website. Available in http://opennebula.org/. Accessed Apr. 2013. 20

[Oraa] Oracle. VirtualBox Website. Available in https://www.virtualbox.org/. Accessed Jun. 2013. 16, 17

[Orab] Oracle. VM Server Website. Available in http://www.oracle.com/us/technologies/virtualization/oraclevm/. Accessed Jun. 2013. 17

[Par] Parallels. Parallels Products Website. Available in http://www.parallels.com/eu/products/. Accessed Jun. 2013. 17

[Pat10] Pratik Patel. Solution: FUTEX_WAIT hangs Java on Linux / Ubuntu in vmware or virtual box. Available in http://www.springone2gx.com/blog/pratik_patel/2010/01/solution_futex_wait_hangs_java_on_linux__ubuntu_in_vmware_or_virtual_box, Jan. 2010. Accessed May 2013. xvii, 30

[PBSL13] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers. In *Proc. of the 2013 Int. Workshop on Security in Cloud Computing (SCC)*, pages 3–10, New York, NY, USA, 2013. ACM. 18

[Por] Portugal Telecom. SmartCloudPT Website. Available in http://www.smartcloudpt.pt/. Accessed Aug. 2013. 12

[Pri13] Matthew Prince. The DDoS That Almost Broke the Internet. Available in http://blog.cloudflare.com/the-ddos-that-almost-broke-the-internet, Mar. 2013. 17

[RAC11] F. Rocha, S. Abreu, and M. Correia. The Final Frontier: Confidentiality and Privacy in the Cloud. *Computer*, 44(9):44-50, Sept. 2011. xvi, 18

[RAN] RANDOM. RANDOM Website. Available in https://www.random.org/. Accessed May 2013. 21

[Red] RedHat. KVM Website. Available in http://www.linux-kvm.org/. Accessed Jun. 2013. 17

[RMVC+12] Luis Rodero-Merino, Luis M. Vaquero, Eddy Caron, Fréderic Desprez, and Aarian Muresan. Building Safe PaaS clouds: a Survey on Security in Multitenant Software Platforms. *Computers & Security*, 31(1):96–108, Jan. 2012. 12, 19

[rSC05] D. Eastlake 3rd, J. Schiller, and S. Crocker. Randomness Requirements for Security. RFC 4086 (Best Current Practice), June 2005. Available from: http://www.ietf.org/rfc/rfc4086.txt. 21

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. of the 16th ACM Conf. on Computer and Communications Security*, pages 199–212, New York, NY, USA, Nov. 2009. ACM. xvi, 18

[RY10] Thomas Ristenpart and Scott Yilek. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *Proc. of the Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, USA, 2010. The Internet Society. xiv, xvii, 4, 29

[SBW09] Alex Stamos, Andrew Becherer, and Nathan Wilcox. Cloud Computing Security: Raining on the Trendy New Parade. Available in https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html, Jul. 2009. xiv, xvii, 4, 30

[Sch11] Thomas D. Schneider. Information Is Not Entropy, Information Is Not Uncertainty! Available in http://schneider.ncifcrf.gov/information.is.not.uncertainty.html, Nov. 2011. Accessed May 2013. 27

[Sch13] Bruce Schneier. What We Don't Know About Spying on Citizens: Scarier Than What We Know. Available in https://www.schneier.com/essay-429.html, Jun. 2013. Accessed Jun. 2013. 18

[SFG+14] Liliana F. B. Soares, Diogo A. B. Fernandes, João V. Gomes, Mário M. Freire, and Pedro R. M. Inácio. Cloud Security: State of the Art. In Surya Nepal and Mukaddim Pathan, editors, *Security, Privacy and Trust in Cloud Systems*. Springer, Berlin Heidelberg, 2014. In press. xv, 6

[SG13] Mark Schloesser and Claudio Guarnieri. Vaccinating systems against VM-aware malware. Available in https://community.rapid7.com/community/infosec/blog/2013/05/13/vaccinating-systems-against-vm-aware-malware, May 2013. Accessed Jul. 2013. 19

[Sha48] Claude E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, Jul. 1948. 27

[Sima] Richard Simard. TestU01 Website. Available in http://www.iro.umontreal.ca/~simardr/testu01/tu01.html. Accessed Jul. 2013. xvii, 34

[Simb] Simtec Electronics. Entropy Key Website. Available in http://www.entropykey.co.uk/. Accessed May 2013. 63

[SIYA11a] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory Deduplication as a Threat to the Guest OS. In *Proc. of the 4th European Workshop on System Security*, pages 1:1–1:6, Salzburg, Austria, Apr. 2011. ACM. xvi, 18

[SIYA11b] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Software Side Channel Attack on Memory Deduplication. In *Proc. of the 23rd ACM Symp. on Operating Systems Principles*, Cascais, Portugal, Oct. 2011. ACM. Poster. 18

[SK11] S. Subashini and V. Kavitha. A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011. 10

[SKLR11] Jakub Szefer, Eric Keller, Ruby B. Lee, and Jennifer Rexford. Eliminating the Hypervisor Attack Surface for a More Secure Cloud. In *Proc. of the 18th ACM Conf. on Computer and Communications Security (CCS)*, pages 401–412, Chicago, IL, USA, Oct. 2011. ACM. 18, 64

[SLC+11] Ming-Kung Sun, Mao-Jie Lin, M. Chang, Chi-Sung Laih, and Hui-Tang Lin. Malware Virtualization-Resistant Behavior Detection. In *Proc. of the IEEE 17th Int. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 912–917, Tainan, Taiwan, Dec. 2011. 18

[SS03] André Seznec and Nicolas Sendrier. HAVEGE: A User-Level Software Heuristic for Generating Empirically Strong Random Numbers. *ACM Trans. Model. Comput. Simul.*, 13(4):334–346, Oct. 2003. 63

[Ste13] James Steddum. A Brief History of Cloud Computing. Available in http://blog.softlayer.com/2013/virtual-magic-the-cloud/, Jul. 2013. Accessed Sep. 2013. 10

[TC11] Greg Taylor and George Cox. Digital randomness. *IEEE Spectrum*, 48(9):32–58, 2011. 22

[Thea] The Linux Foundation. Xen Cloud Platform 1.6 Website. Available in http://www.xenproject.org/downloads/xen-cloud-platform-archives/xen-cloud-platform-16.html. Accessed Sep. 2013. 49

[Theb] The Linux Foundation. Xen Website. Available in http://http://www.xenproject.org/. Accessed Jun. 2013. 17

[Tom09] David Tompkins. Entropy in Cloud Computing Applications. Available in http://blog.dt.org/index.php/2009/08/

entropy-in-cloud-computing-applications/, Aug. 2009. Accessed Apr. 2013. xiv, 4

[vH13] Folkert van Heusden. Folkert van Heusden Website. Available in http://www.vanheusden.com/, 2013. Accessed May 2013. 64

[VMw] VMware. VMware Products Website. Available in https://www.vmware.com/products/. Accessed Jun. 2013. 17

[VMw06] VMware Community Forums. Low /proc/sys/kernel/random/entropy_avail causes exim to stop sending mail. Available in http://communities.vmware.com/message/530909, Aug. 2006. Accessed May 2013. xvii, 30

[VPT+12] Quang Hieu Vu, Tran-Vu Pham, Hong-Linh Truong, S. Dustdar, and R. Asal. DEMODS: A Description Model for Data-as-a-Service. In *Proc. of the IEEE 26th Int. Conf. on Advanced Information Networking and Applications (AINA)*, pages 605–612, Fukuoka, Japan, Mar. 2012. 10

[War] Brian Warner. EGD: The Entropy Gathering Daemon Website. Available in http://egd.sourceforge.net/. Accessed Sep. 2013. 64

[Woe13] Gerhard J. Woeginger. The P-versus-NP page . Available in http://www.win.tue.nl/~gwoegi/P-versus-NP.htm, May. 2013. Accessed Aug. 2013. 24

[XX13] Zhifeng Xiao and Yang Xiao. Security and Privacy in Cloud Computing. *IEEE Commun. Surveys Tuts.*, 15(2):843–859, 2013. 10

[Yil10] Scott Yilek. Resettable Public-Key Encryption: How to Encrypt on a Virtual Machine. In *Proc. of the Int. Conf. on Topics in Cryptology*, CT-RSA'10, pages 41–56, San Francisco, CA, USA, Mar. 2010. Springer-Verlag. 30

[ZJRR12] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proc. of the 19th ACM Conf. on Computer and Communications Security (CCS)*, pages 305–316, Raleigh, NC, USA, Oct. 2012. ACM. 18

[ZMZ12] Muhammad N.A. Zabidi, Mohd A. Maarof, and Anazida Zainal. Malware Analysis with Multiple Features. In *Proc. of the UKSim 14th Int. Conf. on Computer Modelling and Simulation*, pages 231–235, Cambridge, London, Mar. 2012. 18