IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received March 16, 2020, accepted March 26, 2020, date of publication March 30, 2020, date of current version April 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2984414

E-BOOT: Preventing Boot-Time Entropy Starvation in Cloud Systems

FERNANDO VANO-GARCIA, (Graduate Student Member, IEEE), AND HECTOR MARCO-GISBERT, (Senior Member, IEEE)
School of Computing, Engineering and Physical Sciences, University of the West of Scotland, Paisley PA1 2BE, U.K.
Corresponding author: Fernando Vano-Garcia (fernando.vano-garcia@uws.ac.uk)

ABSTRACT Due to the impracticability of generating true randomness by running deterministic algorithms in computers, boot-loaders and operating systems undergo the lack of enough supplies of entropy at boot-time. This problem remains a challenge and affects all computer systems, including virtualization technologies. Unfortunately, this situation leads to undesired side effects, affecting the security of important kernel components and causing large blocking waits in the start-up of userland processes. For example, `SSHD` is delayed up to 4 minutes. In this paper, we analyze the boot-time entropy starvation problem, performing a comprehensive analysis of the Linux kernel boot process revealing that the problem not only affects userland applications but up to 33 kernel functions at boot time. Those functions are weakly fed by random numbers from a non-initialized CSPRNG. To overcome this problem, we propose E-Boot, a novel technique that provides high-quality random numbers to guest virtual machines. E-Boot is the first technique that completely satisfies the entropy demand of virtualized boot-loaders and operating systems at boot time. We have implemented E-Boot in Linux v5.3 and our experiments show that it effectively solves the boot-time entropy starvation problem. Our proposal successfully feeds bootloaders and boot time Linux kernel hardening techniques with high-quality random numbers, reducing also to zero the number of userspace blocks and delays. The total time overhead introduced by E-Boot is around 2 μs and has zero memory overhead, since the memory is freed before the kernel boot ends, which makes E-boot a practical solution for cloud systems.

INDEX TERMS Cloud, virtualization, security, entropy, boot-time, operating systems.

I. INTRODUCTION

Cloud computing has undeniably transformed our society and the way we interact with the world and with people. Over the last decade, it has been gradually established as the *de-facto* model, evolving and offering new possibilities for both academia and industry. It provides unbounded computing resources on-demand, reducing costs due to *economies of scale* [1]. Customers can benefit from vast computing power and storage capabilities of large data centers following a pay-as-you-go model, without the need to possess the necessary hardware resources. Virtualization technologies are one of the fundamental building blocks of cloud computing. Resources utilization can be maximized along with a flexible and efficient on-demand workload multiplexing, yielding to potentially energy efficient infrastructures [2] by aggregating the users demands.

The associate editor coordinating the review of this manuscript and approving it for publication was Yanjiao Chen.

Nowadays, new cloud paradigms and virtualization technologies have emerged due to the compelling tendency to shift to a finer granularity, with the aim of increasing the ability to dynamically scale resource utilization and maximally exploit the infrastructure resources. Serverless computing [3], [4] is an example of cloud computing model that has recently reached a relevant situation in its development. In contrast to *Infrastructure as a Service* (IaaS) [5], where the provider supplies storage, networking and virtualization so that the client has full control over the system from OS layer upwards, the serverless computing model enables dynamic resource allocation, leaving the scaling of infrastructure requirements to the provider. This approach avoids over-provisioning of resources, eventually decreasing effective costs.

Serverless computing allows developers to focus on the application business logic without having to worry about infrastructure management. This leads to the appearance of new computing models and modern technologies

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

offering native high scalability by following the *Function as a Service* (FaaS) [6] paradigm, such as `metacall.io` and `quarkus.io`. Similarly, virtualization technologies are also increasingly leaning towards a finer granularity and more efficient computing. An example of this trend are micro-virtual machines (microVMs), which run custom and tightly fitted kernels that only feature components strictly necessary for a given service, avoiding unnecessary processing and memory usage. One of the representative technologies of microVMs is Firecracker [7], an open source virtual machine monitor (VMM) that uses the Linux Kernel-based Virtual Machine (KVM) to create microVMs. This lightweight approach allows a better fitting in workload balancing and a reduction of the start-up time. In addition, the model is security-aware and reduces significantly the attack surface.

This is important because, given the significance that cloud computing has in people's lives, it is imperative to offer security standards to protect the confidentiality, integrity, and availability in all cloud computing architectures. Cloud providers must ensure the highest possible level of security in order to prevent attacks. Unfortunately, there are scenarios where this is not always possible. An important example is the boot-entropy starvation problem [8]–[11], which appears when a system (e.g., boot-loader or operating system) requires entropy at boot-time but it is unable to generate or collect enough entropy.

The lack of enough entropy at boot-time derives into different problems, including the generation of weak cryptographic keying material [9] and the synchronous blocking of components [12]. As a result, some hardening and protection techniques used by boot-loaders, operating systems and userland applications are weakened or they are blocked until the quality entropy is available. The latter is not an option for operating systems because stopping the boot process could lead to a boot failure or unacceptable delay. This is a significant problem since it fails to meet the security requirements of current cloud computing models. In this paper, we deeply investigate this problem and propose a novel solution for cloud computing.

The main contributions of this paper are the following:

- The problem of boot-time entropy starvation is described, examining the reasons that cause it and the derived consequences.
- We present a comprehensive chronological analysis of the Linux kernel boot process with all the components requesting random data during the boot-time, identifying the cases in which there is insufficient quality entropy available when these requests are made.
- We propose E-Boot, a practical solution to boot-time entropy starvation that provides high-quality entropy to virtual machines in cloud systems at early stages of the boot process.
- We implement the proposed solution in the Linux kernel v5.3 for the x86_64 architecture and evaluate its effectiveness and overhead, showing that the overhead introduced by E-Boot is negligible.

The rest of the paper is organized as follows. Section II provides a detailed background on entropy and the generation of random numbers in modern computers, and an overview of the concrete case of the cryptographically secure random number generator present in the Linux kernel. Sections III, IV and V enumerate and analyze the main entropy sources and entropy collector techniques currently used, followed by a description of entropy consumers, which request random data to them. Section VI describes the boot-time entropy starvation problem, along with the reasons causing it and its consequences. In section VII we propose E-Boot, a solution to solve this problem for virtualization technologies. Section VIII presents an implementation of our design in Linux v5.3 for the x86_64 architecture. Section IX provides an evaluation of the implementation. Finally, section X concludes.

II. BACKGROUND

The importance of entropy and the generation of random numbers are sometimes undervalued. Although the intuitive idea is quite simple, having a good entropy source in deterministic machines is a challenge. This section presents an overview of entropy and generation of random numbers in deterministic machines.

A. ENTROPY

Entropy is an abstract concept with multiple definitions that vary depending on the area of knowledge. According to the literature, there are three types of entropy [13]: thermodynamic entropy, residual entropy and information entropy. This might create confusion, since the same concept is used with slightly different meanings. In this paper, we focus on the latter definition of entropy, related to the information theory field. Thereupon, information entropy is a foundational concept that measures uncertainty or randomness. It is commonly understood as the amount of surprise caused after producing a random value. The higher the entropy, the more uncertainty and, therefore, the more surprise there will be when reading the result of a randomized process. It is an important concept with several applications. For example, it is widely used in many areas of machine learning (e.g., training of Decision Trees [14]), for proving the security of unconditionally secure cryptosystems [15], for detecting incipient damages in a 3D 9-bay truss-type bridge [16] and to derive the mutual information measure in medical image co-registration [17].

Digital computers are deterministic physical machines. By definition, it is not possible to produce true (information-theoretically secure) random numbers from a finite state machine [18]. This is a challenge for operating systems, which try to mitigate this problem by collecting entropy from external non-deterministic and unpredictable events and chaotic sources, typically from hardware including peripherals and dedicated devices to generate randomness. However, when an operating system starts its execution, there is usually little or no entropy available until the kernel is initialized and

able to start collecting entropy from those external sources. In addition, certain systems as embedded devices and virtual machines suffer more constraints to gather entropy for different reasons, such as short boot times and/or the lack of physical hardware providing uncertainty. This reduces the random numbers that virtual machines can produce and incurs a negative impact in security and performance.

B. PSEUDO-RANDOM NUMBER GENERATORS

Without random numbers, many applications would fail and security could not be provided. In cryptography, random numbers are needed to generate private and public keys, initialization vectors, challenges, salts in password hashes, etc. In science, random numbers are used for several different applications, for example for running complex simulations using Monte Carlo methods [20] and randomized controlled trials [21].

As discussed in the section II-A, digital computers are deterministic physical machines and it is not possible to produce true random numbers from a finite state machine [18]. A Pseudo-Random Number Generator (PRNG) is a deterministic algorithm capable of generating large sequences of numbers derived from an initial state (or seed). The output of a PRNG has similar properties to real random numbers, and it will look random for anyone who does not know the initial state. However, it is possible to reconstruct the exact sequence of numbers produced by a PRNG knowing its initial seed or internal state. This is a desired property in some cases. For example, it allows the reproducibility of experiments and simulations by only providing the initial seed value. Contrarily, it can be fatal in the context of cryptography, since it would allow attackers to easily bypass any security provided by secret keys.

For cryptographic functions demanding high quality random numbers to generate secret keying material, regular PRNGs are insufficient. A Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) is a subclass of PRNG with more restrictive properties, including high-complexity for 1) reversing the internal state from a given output, 2) predicting past and future outputs from a given output, and 3) the ability to feed it without compromising its correct operation. To have all those properties, it is fundamental to seed the CSPRNG with high-entropy random data. Otherwise, an attacker knowing the initial state of a particular generator will be able to predict the entire output sequence. A secure seed can be obtained from the output of another CSPRNG, from non-deterministic physical processes or from unpredictable events [22]–[27]. Once it is initialized with a high-entropy seed, it can be used to reliably obtain large sequences of artificial random numbers from a few true random bits. Random numbers generated by a properly seeded CSPRNG are considered computationally secure.

C. LINUX CSPRNG OVERVIEW

Most modern operating systems offer interfaces to one or multiple internal CSPRNGs to allow userspace programs and the kernel itself to obtain computationally secure random numbers for any desired use, such as the Address Space Layout Randomization [28], [29]. In this section, we present the most recent CSPRNG design used by Linux from version v4.8 [30] to 5.5.8, the most recent version at the time of writing this paper.

Linux maintains special memory areas called `entropy pools`, holding true random data derived from different sources of entropy. When raw random data is added to an entropy pool, the input is mixed with the already existing contents using a linear-feedback shift register (LFSR) function. This is how Linux allows updating its entropy pool with any data without compromising the overall unpredictability of the pool contents. Even if a chunk of totally predictable data is mixed into an entropy pool, the uncertainty of its contents does not decrease. Therefore, any pool contents update means either increasing its entropy or keeping the same, but never decreasing it.

Each pool has an associated conservative estimation of available entropy in the corresponding pool. This estimation is decoupled from the contents, implying that not all the input being mixed in the entropy pools is necessarily credited. It is not always possible to accurately measure how much entropy has a certain input. In such cases, the input is mixed with the destination pool anyway, but without incrementing the entropy estimation. This is because, in the worst case, even if the input does not provide any additional entropy (e.g., a totally predictable pattern) the existing unpredictability of the pool contents is not decreased.

When random data is requested from an entropy pool, the raw contents of the pool are never directly exposed. Instead, the contents are hashed using `SHA-1` and the resulting digest is folded by mixing the most significant bits with the least significant bits using the XOR binary operation. The resulting 80-bit value is the one returned as output.

The kernel tries to gather randomness from wherever possible throughout its execution uptime, feeding a primary entropy pool called `input pool`. For example, a typically reliable way to obtain true randomness is from the unpredictable arrival of external events. Additionally, userland programs are also able to provide data to update the primary `input pool` to increase the uncertainty. However, entropy from userland is not always credited and only privileged users can update the estimation accordingly to the data provided.

Linux offers essentially two different sources of pseudo-random data: 1) a secondary entropy pool called `blocking pool` and 2) a CSPRNG based on the ChaCha20 cryptographic function [31]. Both are periodically fed with true entropy from the `input pool`. On the one hand, the `blocking pool` retrieves random data using the operation previously described, using the folded `SHA-1` digest of its contents. This pool is typically smaller than the primary `input pool`, although this can be modified at compile time. An important peculiarity of the `blocking pool` is that random data requests to this pool depend on the available entropy accounted by its associated estimation.

TABLE 1. Overview of interfaces provided by the Linux kernel to obtain random values.

Interface	Description
get_random_bytes() get_random_u32() get_random_u64()	Kernel's internal functions that uses the CSPRNG to return random data. Those functions can only be used from the kernel and no user applications can directly access to them.
/dev/random	Special file that uses the internal blocking pool to return random bytes when read from userspace. If the estimation of available entropy is low, the read operation will block until some more randomness is gathered.
/dev/urandom	Special file that uses the internal CSPRNG to return random bytes when it is read from userspace. In contrast to /dev/random, the read operation never blocks.
getrandom()	System call included in Linux version 3.17 [19] that returns random bytes to userspace. Useful when applications have not available file descriptors and cannot open files (i.e., /dev/random) to obtain random bytes. It is also useful in chrooted environments.

For each request, the entropy estimation decreases and, if the estimation goes below a threshold, successive requests will block until enough entropy is available. On the other hand, the CSPRNG algorithm is based on the ChaCha20 stream cipher to produce as many bytes as requested. At boot-time, its state (primary_crng) is initialized, but for that it requires at least 128 bits of true random data. Afterwards, the CSPRNG is regularly re-seeded. The stream cipher allows the kernel to produce an effectively large sequence of cryptographically secure pseudo-random numbers from a small amount of random data. In contrast with the blocking pool, random data requests to the CSPRNG do not need to block.

With this design, the kernel provides four different relevant interfaces to generate random bits from its CSPRNG and entropy pools, one for internal kernel usage and the rest for userland programs. Table 1 shows an overview of these four interfaces. The file /dev/random uses the blocking pool to obtain random values. The file /dev/urandom uses the CSPRNG instead. The function get_random_bytes() also uses the CSPRNG, but it is not reachable from userspace. Finally, the getrandom() system call returns random data from either /dev/urandom or /dev/random depending on the flag argument of the call.

III. ENTROPY SOURCES

An entropy source is a device or technique from which entropy or randomness can be directly produced. As we highlighted in section II, it remains a challenge for operating systems to generate entropy. This section presents an overview of characteristics and limitations of the main four entropy sources from which a modern computer can obtain true randomness. Later, we present a brief analysis of all of them, synthesizing the relevant features when utilizing them in virtualized operating systems.

A. OS EXTERNAL EVENTS

Unpredictable external events are a suitable method to obtain true entropy in deterministic machines. One of the basic entropy sources from which an operating system can derive true randomness is by sampling events from human interface devices (HID) and other interrupts, including events from rotating disk drives, network adapters, mouse movements, keys pressed, etc. In these cases, the main source of uncertainty is obtained by taking a time-stamp when those HID interrupts arrive with a high-resolution timer, which is typically available in all modern computers. Therefore, some little entropy can be obtained from the unpredictability of the arrival of such events.

However, this approach has some shortcomings when it comes to virtual machines. The virtualization of disk devices and the lack of any type of HID peripherals in these environments substantially reduces the obtained entropy. Since entropy comes from external events, the randomness generation bitrate is slow and the time required to gather enough entropy as to initialize a CSPRNG could be large.

B. CPU JITTER

The high complexity of modern digital computers yields opportunities to take profit from the unpredictability of small fluctuations of different clocks at the nanometric scale to obtain real entropy from deterministic machines. A practical way to obtain true random numbers from modern digital computers is by measuring the small timing variances in the execution of machine instructions and non-modellable noise derived from memory access times, exploiting the complexity of operating systems and the underlying hardware [24]. Listing 1 shows an example of a random bit generator based on the CPU jitter completely implemented in software. It takes a base reference time-stamp, and a loop flips the value of a boolean variable an undefined number of times, where the condition of termination is that the current time-stamp is greater than the reference plus a threshold. This is the general idea followed by some programs such as TrueRand [32] and its derivatives [33], [34]. Another approach is to measure how much time a sleep operation takes [35].

However, the confidence of this method is controversial [36]. Generally, since it relies on a lack of knowledge of complex systems (existing workload, scheduler, state of cache memories, etc), the generated output by using the CPU jitter technique can be predictable to a knowledgeable attacker. Similarly, in simpler systems (e.g., without multiple execution threads and interruptions), the unpredictability is

```
ref = read_tsc()
while(read_tsc() < (ref + THRESHOLD)
    )
        b = !b

return b
```

Listing 1. Example of a random bit generator function.

really challenging. Also, CPU jitter entropy is produced by measuring timings, so it cannot produce large streams of random bits in a short period of time.

C. HARDWARE RNG

Hardware RNG (HRNG) is a specialized hardware circuitry used to obtain randomness from the inherent uncertainty of physical phenomena such as thermal noise [37], [38], quantum events [39], [40], frequency drift in oscillators [41], [42] and analog feedback circuits [8], [43].

There are many manufacturers providing different kind of HRNG devices. For example, a Trusted Platform Module (TPM) is a cryptographic microprocessor which includes a secure TRNG, similar to Intel's 82802 Firmware Hub (FWH) chip [44], [45]. The Unified Extensible Firmware Interface (UEFI) is intended to be a modern replacement of the Basic Input/Output System (BIOS) firmware interface [46], offering several improvements such as faster boot and a new disk architecture to support more and bigger partitions. From version 2.4, the UEFI specification [47] defines an entropy gathering protocol [48], designed for UEFI drivers to provide randomness to upper levels from its internal RNG. In addition, other devices as smart cards [49]–[51] and RNG tokens can be used as a entropy source. RNG tokens are typically attached to the machine via USB port [52], for example, the Altus Metrum ChaosKey [53] and YubiHSM [54]. HotBits [55] is another example that uses a commercial monitor connected to the computer via serial port to derive randomness by detecting radioactive decay events. Similarly, there are peripherals and sensors that can obtain randomness from other physical sources such as audio, video and radio devices [56]–[59].

Another convenient approach to obtain random numbers is by using on-CPU HRNG, which allows the obtaining of entropy from the CPU itself by executing a machine instruction. The CPU has an internal RNG that produces as many entropy as required. Many CPU manufacturers include a RNG in their specification. For example, the VIA C3 Microprocessors family with PadLock engine includes an electrical noise-based RNG accessible through the `xstore` instruction [60]. In similar way, recent x86 architectures provides two assembler instructions, `RDRAND` and `RDSEED`. According to Intel [61], both instructions are compliant to the U.S. National Institute of Standards and Technology (NIST) standards on random number generators (SP 800-90A, B & C) [62]. Although both instructions return random numbers likewise, the implications of their usage are subtly different. The difference is that `RDRAND` gets the entropy from a cryptographically secure pseudo random number generator and `RDSEED` from a true random generator. Similarly, s390 z14 IBM machines have the `CPACF` set of cryptographic instructions [63], [64], and machines with PowerISA 3.0 has the `DARN` instruction [65].

Having a dedicated HRNG, whether it is within a device or integrated in the CPU, allows the programmer to obtain cryptographically secure randomness. In most of the cases, this randomness can be produced at fast bitrates. For example, Intel's `rdseed` and `rdrand` are capable of producing 742.4 Mbit/s and 6.4 Gbit/s of random data, respectively [61], while some USB RNG tokens can offer bitrates of around 327 Kbit/s [66]. However, there are also exceptional cases where a HRNG offers low bitrates. For example, solutions based on radioactive decay are typically limited to 800 bit/s [55].

Unfortunately, not all physical machines are equipped with HRNGs. Even when any HRNG is available, it must be carefully used by virtual machines, because it can cause issues when those virtual machines are migrated to other physical machines lacking HRNGs. In addition, there are some users and developers that do not fully trust in hardware-based RNGs. In recent years, the appearance of news relating to global cyberwar has motivated a public concern about espionage campaigns carried out by intelligence agencies and all kinds of threat actors seeking to introduce stealthy backdoors in hardware to benefit their interests [67]–[72]. Furthermore, in order to use HRNG devices, operating systems require to load a driver in order to communicate with the device. Using them can provide entropy to userland applications and to the kernel itself, but only after the operating system is booted. Therefore, even having a dedicated HRNG device fails to provide entropy to key stages of the operating system boot process where cryptographically secure random numbers are required. An example is the kernel randomization security mechanism present in all modern operating systems, where entropy is required at a very early stage before the kernel starts its execution. In addition, most of the HRNGs do not provide any mechanism to update them in presence of vulnerabilities [73].

D. DRAM RNG

Another source of randomness can be obtained from the Dynamic random access memory (DRAM) chips. The main idea is to misconfigure key low level DRAM memory parameters to cause random alterations to memory cells, and use them as source of entropy.

The main three approaches to obtain randomness from DRAM are:

1) **Violating standard timing parameters**. When the time to refresh memory is reduced below a threshold recommended by the manufacturer, some errors are produced and the value of some random memory cells is changed [26].

TABLE 2. Overview of entropy sources and their characteristics, with regard to cloud environments.

Entropy Sources	Bitrate	VM Migration Friendly	Trustworthy Randomness
OS External Events	○	●	●
CPU Jitter	○	●	●
Hardware RNG	◐	○	◐
Abusing DRAM	●	○	●

2) **Retaining cell charge**. DRAM data is stored in capacitors that leak its charge when its bitline voltage goes below a minimum (Vmin). This approach gets entropy from cells that randomly fails for a given refresh interval [74], [75].
3) **Reading cell contents at power-up**. When a device is turned on, memory cell contents are unpredictable due to interaction between different logic parts, such as precharge, row decoder and column select [76], [77].

The approach that provides good entropy with the lowest latency and highest throughput is ''Violating standard timing parameters'' [26]. It offers more than 5 MiB/s of true random data.

Unfortunately, the commands that govern the behaviour of DRAM chips (i.e., row activation latency, power management, etc.) are performed by the memory controller. Memory controller parameters are typically configured by the firmware in the host machine before loading any operating system. To prevent security issues, virtual machines can only use a subset of physical resources, supervised by the hypervisor. Consequently, the exploitation of these techniques to obtain randomness from DRAM is inflexible, unscalable and prevents to use the DRAM chips being used to generate random data. This results in a non-practical way to obtain random data.

E. BRIEF ENTROPY SOURCES ANALYSIS

Table 2 summarizes the entropy sources discussed in this section considering the random number throughput, virtual machine migration and quality.

Sampling unpredictable external events from the OS is a convenient method to obtain true randomness. Since this is internally done by the operating system, virtual machines can be migrated to other physical machines and use the same approach to collect random numbers. However, in some cases, the virtualization of devices and peripherals normally reduces the obtained entropy and has low generation bitrate. Hence, it requires a long period of time to generate effective amounts of random data.

CPU Jitter is a software-based entropy source that generates random data from the CPU itself, exploiting the high complexity of underlying systems. It is compatible with virtual machine migration in cloud environments because it does not require any specialized hardware to generate random data. Using CPU Jitter to generate random data is slow and a certain amount of time needs to be elapsed to start generating randomness. In addition, unpredictability of data generated by CPU Jitter in some specific simpler environments (e.g., small and limited embedded devices or tiny virtual machines) could be dubious and less than expected.

Hardware-based entropy sources, including HRNG devices and on-CPU HRNGs, can offer true entropy with a normally fast generation bitrate, with a few exceptions (e.g., observing radioactive decay). Unfortunately, those chips are not widely available in all physical machines which introduces challenges to be used by virtual machines with live migration support. In addition, these sources have trust issues because they are typically black boxes, difficult to audit and, sometimes, to update. But the deficiency that probably matters most is that most of them require drivers and therefore the entropy can not be used by the operating systems until drivers are loaded. Unfortunately, this is preventing operating systems to obtain entropy at key boot stages. This affects hardening protection techniques that require entropy at early stages of the boot-time.

Using DRAM chips as source of randomness provides fast and true entropy based on unpredictable memory chip failures. This entropy is available in early stages, including pre-boot environments such as boot-loaders. However, it prevents to use those chips while obtaining the entropy and it also require firmware modifications. This makes the approach unpractical, introduces overheads and reduces the memory available during the entropy generation.

The entropy sources described in this section are the four main techniques used for obtaining true randomness in modern computers. By definition, they are the actual root of entropy. However, these sources are not usually used by entropy consumers (final applications) but by entropy collectors instead.

IV. ENTROPY COLLECTORS

An *entropy collector* is a software program whose main job is to obtain entropy from different *entropy sources*, do some kind of processing and then provide random data to *entropy consumers*. In this section, we present an overview of the main entropy collectors, including different tools and techniques.

A. USERSPACE ENTROPY DAEMONS

Userspace entropy daemons are programs that run in userspace, normally as a background process, and are intended to remedy low-entropy conditions and boost entropy availability in the operating system. There are many available entropy collector daemons at userspace. Following we show the most well-known and widely used in Linux.

Haveged is an entropy daemon that relies on the processor's volatile state [78]. It is based on the HAVEGE (HArdware Volatile Entropy Gathering and Expansion) algorithm to gather entropy using jitter timings from changes in the processor state [79], [80] such as cache memories and branch predictors. `Rng-tools` tries to obtain entropy from a discrete list of sources (e.g., HRNG and CPU Jitter) to collect it

and feed the kernel pool [81]. The daemon does not start if none of its listed sources is available.

Entropy Gathering Daemon (EGD) is a daemon aimed to be used in busy systems that do not feature the any special devices or system call to obtain entropy from the kernel [82]. This entropy daemon is based on unpredictable events such as the output of system administration programs (e.g., lastlog) to obtain randomness. Other programs gather entropy from peripherals using different techniques, for example exploiting audio devices [56], [83], video devices [57] and measuring time of sleep operations [35].

Userspace entropy daemons can be useful to avoid the depletion of operating system's entropy reserves in long-term running systems with high demanding of randomness. However, they are not available at early boot-time, since they cannot be executed until the system is completely booted and running. This prevents operating systems and boot-loaders to use this approach during the boot process where randomness is required.

B. ON-DISK SEED FILES

The fundamental idea of on-disk seed files is to carry entropy across reboots. At system shut-down, a random seed is saved into a file in persistent storage. Next time the system starts up, this seed can be used to feed the entropy reserves of the operating system. This technique takes advantage of the fact that a running system has potentially accumulated more entropy than one just started. In situations where an attacker has knowledge of the environment (i.e., hardware, start-up activities, etc), the seed file makes it difficult to predict the OS internal entropy state, from the second time the system boots onwards.

In Linux systems, on-disk seed files are typically managed by userspace startup scripts or by the init process. Examples of them are `sysVinit-scripts` and `systemd-random-seed`. On the other hand, `Systemd` also uses a similar approach with its optional EFI boot-loader (`systemd-boot`) [84], storing a random seed in the EFI System Partition (ESP). Similarly, `Early-rng-init-tools` is a package of tools for Linux systems that uses different entropy sources and techniques to fetch entropy to earlier phases in the boot process.

On-disk seed files can be useful to increase unpredictability of OS internal entropy state, using entropy from previous executions. However, they must be treated carefully. In cloud environments, its usage in transient virtual machine instances sharing a cloned generic image and live operating systems can be dangerous, since an attacker using the same image has exactly the same seed file. In these cases, the actual entropy of the seed file should be considered zero. In addition, there is no feasible way to measure how much entropy has a certain seed file. Furthermore, on-disk seed file solutions based on EFI are typically discouraged in virtual machines, because the EFI variable space and the disk space can be shared, cancelling the security benefits [84]. Seed files must be handled secretly and their contents must be immediately updated after using them. In some cases, this fact imposes limitations, as the need of synchronous blockage waiting for having enough entropy to generate the seed file replacement.

C. COMPILER-ASSISTED LATENT ENTROPY

Compiler-assisted latent entropy is a technique that consists in applying subtle modifications to an operating system at compile time, without causing any semantic changes, adding logic to obtain randomness from the kernel's runtime state when it is executed.

At the time of writing this paper, there is only one known solution using this technique, which is `Latent Entropy`, a GCC plugin [85] originally designed and written by the PaX Team for grsecurity [86] and later ported to the Linux upstream [87]. It has two parts: 1) *static* random values settled by the compiler and 2) *dynamic* random values computed at runtime. Essentially, different functions are instrumented with additional code that modifies local variables with randomly selected operations, such as addition, shifts and exclusive-or, and other values determined during the build process. Initialization routines, functions called at random times and functions with variable and unpredictable loops are good candidates. At the epilogue of these functions, the computed value of each local variable is mixed into a global variable, which accumulates the entropy. Eventually, the randomness of this global variable contributes to the operating system's internal entropy.

This technique provides per-build and per-boot randomness. On the one hand, static values and selected operations in the modified functions are different per-build. On the other hand, the unpredictability of the operating system's runtime is exploited to obtain dynamic per-boot randomness. For example, the order of functions being called, different branches taken within a function, etc. Interruptions and concurrency can also increase its unpredictability. In addition, contents of some registers such as the stack pointer or the frame pointer can be used to add unpredictability, taking advantage of the randomness provided by other mechanisms as kernel randomization. It is a transparent technique, independent of the hardware, that can be useful to increase the unpredictability of operating system's internal entropy state.

Unfortunately, this technique is still in an experimental state and the actual overhead introduced by the OS modifications is unknown. Current versions of Linux do not credit this entropy into the internal estimation. In addition, it is not feasible to produce large quantity of randomness (e.g. 128 bits) at early stages of the boot process.

D. REMOTE ENTROPY SERVICES

Remote entropy services are a combination of technologies and techniques that allow users to obtain random data from a remote server. Thus, machines with one or more entropy sources can be exploited to provide randomness to remote machines.

An important example of this technique employed in cloud computing is known as Entropy as a Service (EaaS) [88].

This model consists in remote delivery of high-entropy data from a decentralized root of trust over a secure connection where clients can make requests to obtain random numbers. An implementation of this model is the client-server pair `pollinate` [89] and `pollen` [90].

A similar approach is the use of public randomness servers [91]. These servers typically offer randomness from a single high-entropy source or from a combination of different sources. For example, `Random.org` has an `API` to provide random data through `HTTPS`.

Remote entropy services can be useful in environments with internet access but with limited sources of entropy. These services offer a large number of clients the possibility of indirectly using expensive appliances in the server to obtain true randomness by observing real unpredictable physical events.

Unfortunately, in order to use remote entropy services, the network must be ready and a secure channel must be established to request random data. This is to ensure that the random data is transferred securely. Otherwise, attackers can use the random data requested by clients to break all client applications where the remote random data is being used as a unique entropy source. Unfortunately, this is the majority of the situations and the main motivation of using those remote services.

Therefore, since this approach requires the system to be completely booted and it also requires high quality random numbers to establish a secure connection, it cannot be used by boot-loaders, operating systems or any device that requires entropy at early boot stages. Even overcoming this strong requirement, another issue discouraging clients of using a remote service is the fact that the service is externalized. Obtaining random data from remote servers forces clients to extend the trusted computed base (TCB) and exposes them in the presence of a server breach.

For private services such as EaaS, clients are forced to delegate part of their security to remote services. This discourage cloud providers from using remote servers as a trustworthy approach.

E. VIRTUAL HRNG

Virtual HRNG is a technique used to enable virtual machines to access a single physical HRNG. Rather than assign a physical device to a single virtual machine, this technique allows multiple guests to obtain entropy from a real HRNG device.

A popular virtual HRNG solution used in the Linux Kernel-based Virtual Machine (KVM) is `Virtio-RNG`. It consists on virtualizing a HRNG device exposing a virtual HRNG device to each virtual machine [92]–[94]. For example, in Linux each guest will have a new `/dev/hwrng` virtual device. Note that this does not affect the host's internal entropy pool state.

In case of lacking physical HRNG devices, virtio-rng optionally allows to attach the virtual HRNG to other files in the host machine. For example, virtio-rng can be attached to the special file `/dev/urandom` or `/dev/random` providing true random and pseudo random data respectively. Although a virtual random generator is exported from the point of view of virtual machines, reading from `/dev/random` will consume entropy form the host's entropy pool. In this case, a rate-limit can be tweaked to avoid host entropy depletion. The Xen hypervisor also has a similar tool called `xentropyd` [95].

This technique basically offers the same benefits and limitations as hardware RNGs. For example, it allows fast bitrate streams of random data in guests, as long as it is available in the host machine.

The main drawback of using Virtual HRNGs is that the entropy is available too late. Boot-loaders cannot take advantage of this approach when operating systems are loaded at early boot stages. Drivers are necessary in the guest operating system to manage the virtual device and consequently the operating system boot process must be nearly completed. Although adding this driver to the kernel core could be seen a solution, the truth is that having this code in all kernels even in those that the functionality is not required discourages this approach to be implemented. In addition, this does not solve the problem completely, since the boot-loader also requires high quality random numbers (e.g., to load the OS in random memory locations), and this approach would also require code at the boot-loader level.

Although this is possible, to have a virtual HRNG client present in the boot-loader is not a minor change. For instance, the work being done in the Linux kernel boot-loader is relatively simple, so it would be difficult to justify this overhead. As we discus in section VI, this suggests that an appropriate solution is required when high quality random data is required at early boot stages.

F. BRIEF ENTROPY COLLECTORS ANALYSIS

In order to facilitate the comprehension and highlight the importance of entropy collectors, in this section we present a brief analysis of them.

Userspace entropy daemons gather randomness from several entropy sources to enhance the overall operating system's entropy. As userland processes, they do not impose limitations to virtual machine migration and are quite portable. However, userland entropy daemons are not usable when the entropy is required at early boot-time stages. They are generally dependent on the availability of strong entropy sources. Otherwise, the provided randomness of these programs could be of poor quality.

On-disk seed files can be useful to carry entropy across reboots. However, seed files must be treated cautiously for several reasons. There is no feasible and secure way to measure how much entropy they contain. Also there are hidden issues that could prevent this approach for being used. For example, in cloud environments, virtual machine instances may share the same seed file. Additionally, they cannot be used in early stages of the boot process because the filesystem needs to be available.

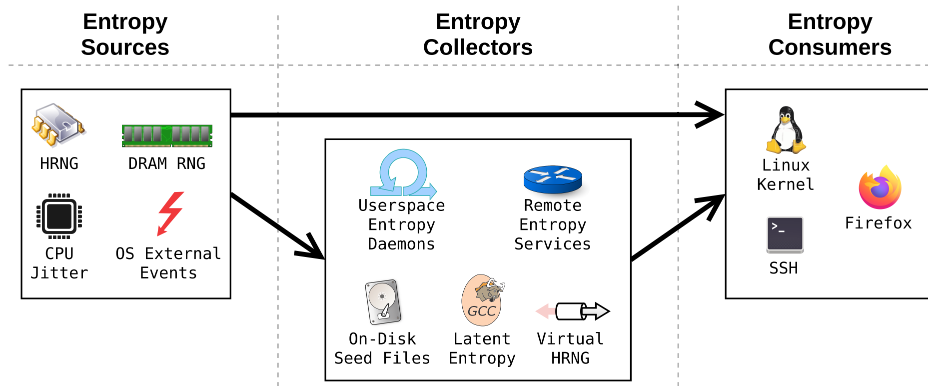

FIGURE 1. Flow of entropy/randomness from sources to collectors and consumers. Sources are the one generating the entropy. Collectors obtain entropy from entropy sources and provide random data to entropy consumers. It is also possible to obtain randomness directly from sources, but this is not recommended for most applications.

Compiler-assisted latent entropy exploits the unpredictability of potential variations in the kernel's runtime state to collect randomness. These variations can be caused by different reasons, for example interruptions upon external events, concurrency and address randomization. Unfortunately, it is not feasible to produce large quantity of randomness at early stages of the boot sequence with this technique. Furthermore, the actual obtained entropy is unknown, and part of their values are settled at compile time, providing zero effective entropy.

Remote entropy services offer the possibility of obtaining randomness through internet services. Although this technique can be useful in some scenarios, it cannot be used at early stages of the boot process. It requires to have the networking system up and working, as well as some high quality random numbers prior to establish a secure connection.

Virtual HRNGs allow host machines to expose physical HRNG devices to guest virtual machines. Guests can benefit from fast streams of true random numbers. However, the use of physical HRNG imposes restrictions to migrate the virtual machines to other hosts lacking the required hardware. In addition, guest operating systems need a driver to communicate with the host backend, which limits its usefulness at early boot-time.

V. ENTROPY CONSUMERS

Entropy consumers, the right square of figure 1, are applications and OS components consuming entropy during its execution time to provide secure properties to the final user. For example, in order to provide a secure channel using `HTTPS` or `SSH` having high quality random numbers is required.

Although entropy consumers are usually userspace applications obtaining random data from entropy collectors, it is also possible to talk directly with entropy sources. However, this would require to allow direct access to those sources and also add this functionality to each application. This approach would probably break applications and virtual machines, since we can not guarantee that all physical hosts will have the same entropy sources. To alleviate this problem, entropy collectors provide this abstraction layer in order to provide high quality randomness to the entropy consumers while deal with the specific entropy sources of the particular host.

The same approach is being used at different levels. For example, an operating system can be at the same time an entropy collector and consumer. In this case the issue is not having an abstraction layer but to use an internal algorithm (entropy collector) fed from some random data obtained from an entropy source. This approach allows operating systems to have an unlimited high quality random numbers for itself and also to export an interface to enable userspace applications (entropy consumers) to obtain randomness.

VI. THE PROBLEM: BOOT-TIME ENTROPY STARVATION

As we have discussed and analyzed in sections III, IV and V, there are many solutions to generate and collect entropy to be consumed by the final user. Unfortunately, after analyzing all the literature, we found that none of those techniques can be employed to solve the boot-time starvation problem [8]–[11].

The issue of lacking enough entropy at boot-time is derived from the fact that it is not feasible to generate real randomness by only running deterministic algorithms in computers, especially in virtual machines and embedded devices. For example, operating systems are not able to gather enough entropy to initialize its internal CSPRNG [36] in early stages of the boot process. This forces to choose between two options. The first would be to block-and-wait random requests until there is enough entropy. This is not something desired or adopted, since the booting process could be stalled and the system could never boot. The second approach would be to return as much entropy as available instead of block-and-wait. Unfortunately, when the returned data is being used in security protection mechanisms such as kernel hardening techniques, those could be seriously weakened.

In the case of Linux, for instance, its internal CSPRNG needs a minimum entropy threshold of quality randomness available to seed the initial state, and it is not ready until this

requirement is fulfilled. In headless systems without peripheral devices generating entropy, such as keyboard and mouse, the time needed to generate the minimum entropy threshold is larger. In cloud virtual machine instances, the missing opportunities to get good entropy from hardware is not only due to the lack of peripherals but also to the virtualization of other components as interrupt requests and block devices. At the time of writing this paper, the availability of high entropy at the early kernel startup is a current concern.

A less obvious issue that could magnify the problem is how some OS improvements affect the entropy generation. For example, as of September 2019, Linux kernel developers had a controversial discussion in the last stages of v5.3 development cycle [12] due to an improvement in the *ext4* filesystem that indirectly caused unexpected boot hangs because the applied changes produced less interrupts early in the boot phase, which produced less entropy. In fact, this blocking problem in Linux systems has been going around for some time, since the incorporation of the `getrandom()` system call by libraries such as OpenSSL [96]–[98]. An example is the Secure Shell (SSH), a widely used service that provides authenticated and encrypted remote access. Its daemon (SSHd) maintains an internal PRNG, which is initially seeded when the program starts. That seed might be requested directly to the kernel or to the OpenSSL library which, in turn, maintains another internal PRNG that also needs to be seeded. Eventually, this entropy originally comes all the way down from the kernel, by means of the `getrandom()` system call. The default behaviour of this system call is to block the caller if the CSPRNG is not ready. This blocking behaviour leads to prohibitive long delays in the boot time of services requiring secure random numbers waiting for the proper CSPRNG initialization. This prevents users and administrators to connect remotely to the machine until the CSPRNG is ready, resulting in prohibitive connection delays. Furthermore, in situations where the blocked services are the only ones capable of providing fresh randomness into the system, it could even cause deadlocks.

The alternative approach to block-and-wait is to return as much entropy as available at the moment of the request. The problem of this approach is that non-blocking requests to an OS CSPRNG made during this low-entropy state produce potentially predictable pseudo-random numbers. This may lead to serious security problems [99], [100] if any of those weak random values are used to eventually generate sensitive data, such as long-term cryptographic keys. For instance, Heninger et. al. [9] performed a large survey of TLS and SSH servers, discovering a widespread presence of vulnerable keys due to insufficient entropy during key generation. Even though not all the requests made to the CSPRNG during the non-initialized state might be intended to be computationally secure, it is conceptually wrong to request random data to a non-initialized CSPRNG. Unexpected security problems can arise by letting this happen [101]. For example, the Linux kernel `ratelimit` feature hides repeating messages if a certain limit is reached, to avoid flooding the message buffer of the kernel. If there are several requests to the uninitialized CSPRNG, the logs will only warn about a small part of them, potentially hiding any dangerous read which actually needed to be cryptographically strong [102].

Summarizing, the boot-time entropy starvation problem leads into two main issues:

1) **Weakened Protection Techniques**: Bootloaders and early stages of operating systems require entropy to securely protect some parts of the OS. For example, in Linux, the boot-loader uses random numbers for kernel randomization, and during the Linux boot process some hardening techniques also require random data. A more detailed list with Linux components can be found in table 3.
2) **Blocking CSPRNG requests**: On systems that are not equipped with special hardware to generate random numbers, even when the kernel has been fully loaded, the CSPRNG could take up to several minutes to be initialized as shown in figure 4. For those systems, this delay is unacceptable since the problem is propagated to any userland application requesting random data to the kernel. As a result, those servers could be unavailable for minutes after a system reboot.

VII. THE SOLUTION: E-BOOT

In this section, we present E-Boot, a novel technique that solves the boot-time entropy starvation problem discussed in section VI by providing high-quality entropy to virtual machines before they start their execution. E-Boot is the first solution that completely satisfies the entropy demand of virtualized boot-loaders and operating systems. Our approach enables boot-loaders and operating systems to access to high-quality random numbers from their very first assembler instructions without requiring any hardware support.

The main idea of E-Boot is to bring high-quality entropy from the hypervisor to guest virtual machines. This enables virtual machines to fulfill all early random data required, as well as to produce their own cryptographically secure random numbers at very early stages of the boot process. The technique can be applied by virtualization technologies (e.g., cloud computing environments) to solve the problem of lacking boot-time entropy where kernel hardening techniques require high-quality entropy to prevent and mitigate attacks.

Although many applications and devices can take advantage of E-Boot, the main benefits that offers for cloud computing are:

1) **Entropy available before VMs are executed**: The hypervisor fills a pre-reserved area in guest memory with random numbers. With this approach, once the virtual machine starts its execution, it can first access this pre-reserved area to retrieve the random numbers and later start its real execution. As a result, kernels and boot-loaders have entropy available from the very first assembler instructions. This is the first step shown in figure 2. The hypervisor fills all pre-reserved areas

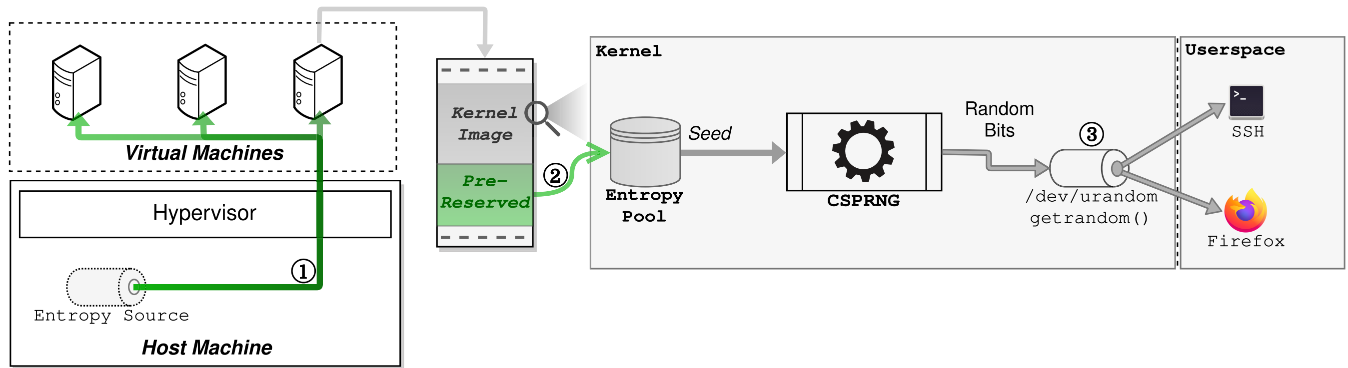

FIGURE 2. Design of E-Boot. Before a virtual machine is started, ① the hypervisor injects entropy into the guest's memory. ② When the guest virtual machine starts its execution, the pre-reserved area filled with random data is already present and reachable from the beginning. All early random data requests can be fulfilled. The OS's internal CSPRNG can be initialized early, allowing the generation of cryptographically secure random numbers. ③ Kernel components and userspace programs are benefited from it, obtaining secure random data and avoiding blocking waits.

of all available and compatible virtual machines with random numbers.

2) **Hardened kernel security**: The entropy from the pre-reserved area can be used to early initialize the internal CSPRNG that all modern operating systems have. This is the second step shown in figure 2. For example, in Linux, there are many security techniques applied during the boot to harden the kernel. Those techniques require random numbers to be effective but, as we discussed in section VI, there is a lack of entropy and those techniques try to do a best effort. Using E-Boot, we can guarantee that the CSPRNG will be fully initialized and all hardening techniques employed during the boot-time will be fully satisfied with all the random numbers required.

3) **Prevents userland blocks**: Unlike what it might be expected, the problem of entropy starvation persists after the kernel has been completely loaded. As we discuss in the section IX, and show in figure 4, the CSPRNG initialization can be delayed up to 4 minutes. This time is required to collect enough entropy to initialize the CSPRNG, and therefore to provide random data to userspace applications. With E-Boot, this time is reduced to zero, and servers and applications are not blocked anymore.

It is important to note that E-Boot does not depend on external hardware, and it is fully compatible with live virtual machine migration including load balancing, resource management and availability. In addition, it does not interfere with the execution of any service running in the guest virtual machines [103]. As with any virtualized environment, guests should trust the hypervisor and therefore, the provided entropy.

Figure 2 shows the design of E-Boot. In step ① , the hypervisor fills the pre-reserved area of all virtual machines with random numbers. The hypervisor is responsible for selecting a proper source of randomness. Once this operation is completed, the hypervisor must indicate to the guest that the job was done. For example, by indicating the quantity of present entropy in the pre-reserved area or by setting a flag. This brings transparency and backwards compatibility with other hypervisors that do not support E-Boot, since guest can ignore the entire pre-reserved area if the pre-reserved area was not filled or has zero entropy.

When the virtual machine starts its execution, the pre-reserved pool filled with random data is already available within its memory, ready to use whenever it is needed. Hence, the guest OS is able to use this entropy from the very first instruction in order to satisfy different random data requests from itself. As shown in step ② of figure 2, E-Boot enables operating systems to add randomness into their internal entropy pools in order to initialize their CSPRNG as soon as possible. This will ensure that all requests from the OS kernel and from userspace (step ③ in figure 2) programs are provided with the required high-quality random data. Note that guests do not require any additional driver running the entire virtual machine lifespan and this process is only being executed once at boot-time. The same principle applies to boot-loaders.

Therefore, E-Boot solves boot-time starvation and all derived problems with an efficient, transparent and lightweight approach, while being compatible with virtual machine live migration. Cloud systems and virtualization technologies can be benefited from it in multiple ways. On the one hand, it potentially accelerates the boot time of services that are otherwise blocked waiting for a CSPRNG initialization for obtaining secure random numbers. On the other hand, it provides strong security for those fundamental components that are necessarily established at an early stage in the boot process and remain untainted throughout the operating system lifespan, such as kernel randomization or memory hardening used by the Linux `SLAB` allocator at boot-time. It also enables the possibility to debug certain workloads that use deterministic PRNGs, providing known seeds to produce controlled and repeatable states across different boots.

This solution can be especially beneficial for virtualization technologies that focus on security while offering

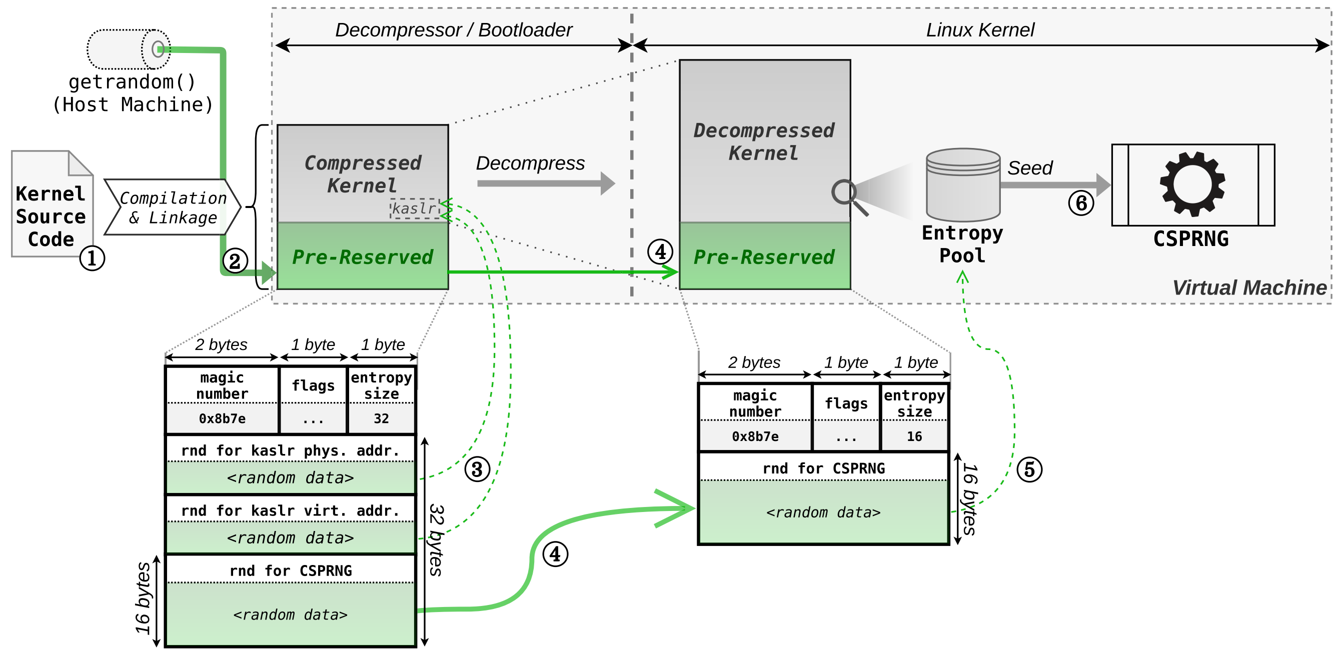

FIGURE 3. **Implementation overview of the proposed E-Boot in the Linux kernel v5.3 for the x86_64 architecture.**

multi-tenant serverless services with a lightweight approach to minimize the attack surface, such as Firecracker [7]. Using E-Boot, micro-virtual machines can benefit from the availability of quality entropy from the beginning of their execution, without needing any additional driver being loaded in the kernel or any channel connected to the host machine.

VIII. E-BOOT IMPLEMENTATION IN LINUX

In this section, we present the implementation of the proposed E-Boot in the Linux kernel v5.3 for the x86_64 architecture. Figure 3 shows an overview of it. Following the design presented in section VII, the first step is to enable the hypervisor to transfer high-quality random numbers to the pre-reserved memory area of all virtual machines.

When the Linux kernel is built from sources, not only the Linux kernel is compiled for a specific architecture but a bootloader capable to load the kernel is also built. On x86_64 and other architectures, by default the kernel is compressed and a single file named `vmlinuz` containing the compressed kernel and the bootloader is created. In cloud computing, this `vmlinuz` is loaded into memory by the hypervisor to later transfer the execution flow to it, more specifically to the Linux bootloader code contained in the `vmlinuz` file.

The Linux bootloader is responsible for loading the Linux kernel into memory and transfer its execution control to it. Since, at this early stage, protection techniques such as the kernel randomization are already underway, E-Boot must be implemented in a way that provides entropy to the Linux bootloader before the kernel is loaded. In addition, after the kernel has been decompressed and loaded but before the execution control is transferred to the Linux kernel, E-Boot must copy random data from the Linux bootloader to the Linux kernel pre-reserved area. Once the entropy has been copied, then the bootloader will transfer the execution flow to the Linux kernel.

Therefore, E-Boot provides to the Linux bootloader entropy for itself but also for the Linux kernel. The memory space to hold the entropy of both must be pre-reserved in the bootloader. We have calculated that 32 bytes of entropy are enough for both, to solve entropy starvation and to offer enough randomness for all early random requests. The bootloader consumes 16 bytes of entropy to assist kernel randomization step ③ in figure 3, and the Linux kernel consumes also 16 bytes of entropy to early start its internal CSPRNG, step ⑤ in figure 3.

To implement E-Boot, we have defined the pre-reserved area which consists of two main parts: 1) a header containing metadata information and 2) a buffer containing the random data, which will be referred to as the payload. The payload buffer is populated by the hypervisor when the virtual machine is loaded into memory. The header contains a 2-byte `magic number` and other two 1-byte fields for `flags` and for the `entropy size`. The purpose of the magic number is to identify the pre-reserved area unambiguously, to avoid unwanted side-effects. The flags field is used for two purposes: 1) to allow the guest to provide a hint to the hypervisor about the entropy preference (i.e., true or pseudo randomness) and 2) to allow the hypervisor to inform the guest that it supports E-Boot. The size field indicates how much random data has been placed in the buffer. We have ① modified two linker scripts [104] of the Linux source code, called `vmlinux.lds.S`, one for the boot-loader and another for the kernel. These scripts are mainly used to describe and control the memory layout of the generated Executable and

Linkable Format (ELF) file, which is the file format used by the Linux kernel. The pre-reserved area is statically allocated within an ELF section of the images (boot-loader and kernel).

In order to provide the entropy to the guest virtual machine, as shown in step ② of figure 3, we have modified the Qemu Virtual Machine Monitor v4.2 [105]. After Qemu has loaded the guest `vmlinuz` image file into memory, the E-Boot parses the file in order to find and fill the pre-reserved area of the bootloader. Before writing the random numbers, a magic number check is done by using the first field of the pre-reserved area header. This is to allow the hypervisor to know that the guest virtual machine is compatible with E-Boot. The second field in the header is one byte reserved for flags. Those flags are to pass information between the hypervisor and the guest virtual machines. For example, the first bit of the flags is set by the hypervisor to indicate that it supports E-Boot. This bit can be later checked by the virtual machine to know that it is running under a hypervisor supporting E-Boot. The second bit is used by the hypervisor to know whether the entropy requested must be from a true random number generator or from a CSPRNG. This bit is set at compile time and cannot be modified unless the `vmlinuz` file is patched. The rest of the bits are unused. The third field of the header is the entropy size contained in the pre-reserved area payload. It is requested by the guest virtual machine at compile time. The hypervisor uses this value to know how many random numbers are being requested. If the provided entropy by the hypervisor is less than the requested, this size field will be updated accordingly, indicating how much actual entropy has been written in the pre-reserved memory area of the guest.

Once the hypervisor has completely updated the guest's pre-reserved area, it passes the execution control to the guest bootloader, which will decompress the kernel into a physical memory. If kernel randomization is enabled [106], [107], the bootloader will calculate the physical and virtual addresses of the final kernel location. As shown in step ③ of figure 3, with E-Boot those two random addresses are calculated using two 8-byte random values from the pre-reserved area. Actually, our implementation performs an *XOR* of the provided random bytes with the little available entropy. This is to ensure that, if the hypervisor does not support E-Boot, the kernel randomization will be as good as before.

After calculating the final virtual and physical addresses, the Linux kernel is decompressed. The boot-loader parses the extracted (kernel) ELF file to move loadable segments to their corresponding address. E-Boot uses the same approach than the one followed in the bootloader. It locates the pre-reserved area, step ④ of figure 3, and it transfers 16 random bytes from the pre-reserved area of the bootloader to the pre-reserved area of the Linux kernel. Then, the bootloader updates the flags and transfers the execution control to the Linux kernel.

At the very first moment the kernel starts its execution, the random numbers are already available in its pre-reserved area and ready to use. As described in section II-C, Linux implements a CSPRNG based on the ChaCha20 [31] cipher, which needs 128 bits of entropy to initialize it. For this reason, as step ⑤ of figure 3 shows, E-Boot provides 16 bytes (128 bits) of random data, to be able to early initialize the Linux CSPRNG. Doing an early CSPRNG initialisation not only solves the problem of entropy starvation but it also satisfies all hardening techniques employed during the kernel boot time. The left-hand column of the table 3 shows a full list of the components and techniques that are benefited from the proposed E-Boot. Note that any future technique, even techniques with high demand on high-quality random numbers at very early stages, will be completely satisfied thanks to the early CSPRNG initialization done by E-Boot, as step ⑥ of figure 3 shows.

To preserve full compatibility with hypervisors not supporting E-Boot, the random data is actually mixed with the kernel's internal entropy pool. This will ensure that the CSPRNG will have the same behaviour when no entropy is provided to guest virtual machines.

To summarize, we define two statically allocated E-Boot sections for both the kernel and the boot-loader. The hypervisor provides the required random data prior to virtual machine start-up. The random data payload size in our implementation is 32 bytes, which are used for kernel randomization and CSPRNG initialization. However, this is completely customisable. In this case, the boot-loader consumes 16 bytes for kernel randomization and transfers another 16 bytes to the kernel.

IX. EVALUATION

In this section, we evaluate how effective E-Boot is for solving the boot-time entropy starvation problem in Linux. Table 3 shows all boot-loader and Linux kernel components that are addressed by the proposed E-Boot. Later, we evaluate the spatial and temporal overhead to demonstrate its feasibility on real systems. At the end of the section, the NIST Statistical Test Suite is used to assess the entropy generation in guests.

To obtain the real impact of E-Boot, we have performed a comprehensive analysis of the Linux kernel v5.3 boot process (for the x86_64 architecture) to ascertain all the kernel components that require random data in early stages, identifying the cases in which there is insufficient quality entropy available when these requests are made. Since Linux requires 128 bits (16 bytes) to initialize its CSPRNG, we will refer to "sufficient entropy" when Linux has collected 128 bits or more of high-quality entropy. Otherwise we will refer as "insufficient entropy", since the CSPRNG will not be able to initialize and all requests to its CSPRNG during the boot time will not be satisfied.

The experiments were carried out running an Arch Linux virtual machine with kernel v5.3, LXDE graphical desktop and full networking (NAT mode provided by Qemu). It executes only basic init-system processes and an SSH server. The physical machine used to run the experiments has an Intel Xeon W-2155 processor (Skylake server microarchitecture)

TABLE 3. Kernel functions called at boot time while there is insufficient quality entropy as to initialize the CSPRNG, comparing Standard, Virtio-RNG and PNAME. The *At* column shows the first occurrence of each kernel function (in milliseconds). A tick (✓) means that there is enough quality entropy available when the corresponding function is reached. † Note that this is the elapsed time since the bootloader started its execution, but the count is reset when the kernel gains control. ‡ Random secret hash seeds are typically used to obtain an uniformly key distribution and to mitigate algorithmic complexity attacks against hash tables.

Kernel Function	Random Data Request		Affected Component	Description	Quality Entropy Available at Boot Time		
	At (ms)	Bootloader/ Kernel/User			Standard	Virtio-RNG	PNAME
kaslr_get_random_long()	*162*†	B	Kernel ASLR	Randomize kernel's physical and virtual addresses.	✗	✗	✓
kaslr_get_random_long()	1	K	Kernel ASLR	Randomize physmap, vmalloc and vmemmap memory regions and the poking address.	✗	✗	✓
add_to_free_area_random()	78	K	Page Allocator	When page allocator's freelist is randomized, this function randomly decides where to insert a new element (head,tail).	✗	✗	✓
cache_random_seq_create() shuffle_freelist()	85	K	Slab Allocator	Randomization of the slab allocator's freelist order to reduce its predictability, protecting against heap overflow exploits.	✗	✗	✓
kmem_cache_open()	88	K	Slab Allocator	Get a random value to harden the slab cache metadata against common freelist exploit methods, such as kernel heap attacks.	✗	✗	✓
init_espfix_random()	88	K	Memory Manager	Randomize the locations of kernel espfix, which protects against leaking high bits of the kernel stack pointer.	✗	✗	✓
boot_init_stack_canary()	89	K	Kernel Stack Protector	Set up a random kernel stackprotector canary value for the idle task (*a.k.a.* swapper).	✗	✗	✓
dup_task_struct()	92	K	Kernel Stack Protector	Set up a random kernel stackprotector canary value for a new process or thread.	✗	✗	✓
shuffle_zone()	92	K	Page Allocator	Page allocator's freelist is shuffled to improve cache utilization and to mitigate attacks exploiting its predictability.	✗	✗	✓
bucket_table_alloc()	93	K	Resizable Hash Tables	Set up a per-bucket random secret hash seed.‡	✗	✗	✓
setup_net()	93	K	Network Namespaces	Set up a random secret hash seed.‡	✗	✗	✓
kcmp_cookies_init()	93	K	Checkpoint / Restore	Get a random value to obfuscate pointers. This is done to prevent information leakage about the location of kernel objects.	✗	✗	✓
neigh_get_hash_rnd()	183	K	Neighbouring Subsystem	Set up an array of random secret hash seeds.‡	✗	✗	✓
rt_genid_init()	183	K	Network Namespaces	Set up a per-device random counter, used for routing purposes.	✗	✗	✓
key_alloc_serial()	308	K	Key Retention Subsystem	Allocate a random serial number for a key. This is used to avoid potential security issues through covert channels.	✗	✗	✓
arch_rnd()	312	K	Userland ASLR	Randomization of a process's memory mapping.	✗	✗	✓
randomize_stack_top()	312	K	Userland ASLR	Randomization of a process's stack top-address.	✗	✗	✓
arch_align_stack()	312	K	ASLR & Performance	Intra-page randomization of a process's stack, to avoid cache evictions by other processes when using Hyper-Threading.	✗	✗	✓
vdso_addr()	312	K	Userland ASLR	Randomization of a process's vdso.	✗	✗	✓
create_elf_tables()	312	K	Userland Stack Protector	Generate the rnd value in the ELF aux. vector (AT_RANDOM), used for stackprotector canary value and pointer mangling.	✗	✗	✓
arch_randomize_brk()	312	K	Userland ASLR	Randomization of a process's heap.	✗	✗	✓
ipv6_regen_rndid()	645	K	Networking (IPv6)	Generate a randomized interface identifier	✗	✓	✓
init_oops_id()	645	K	Log & Debug	Generate a 64-bit random ID for oopses.	✗	✓	✓
prandom_seed_full_state()	645	K	General-Purpose PRNG	Use the CSPRNG to seed the kernel's internal equidistributed combined Tausworthe PRNG (prandom) .	✗	✓	✓
get_module_load_offset()	750	K	Kernel ASLR	Randomize the starting load address of modules. A random offset is settled when the first module is loaded.	✗	✓	✓
urandom_read()	800	U	Several Processes	Non-blocking requests from different userland processes such as graphical manager, systemd and dbus-daemon.	✗	✓	✓
generate_random_uuid()	805	K	Boot-ID	Generate a random Boot-ID. This UUID is exposed in the file /proc/sys/kernel/random/boot_id.	✗	✓	✓
getrandom()	912	U	SSH & Systemd	Blocking requests from systemd-random-seed and SSHd userland processes, waiting until CSPRNG is ready.	✗	✓	✓
addrconf_dad_kick()	1770	K	Networking (IPv6)	Get a random nonce for each IPv6 address for Duplicate Address Detection (DAD).	✗	✓	✓
net_secret_init() ts_secret_init()	1800	K	Networking (Core)	Each function generates a random 128-bit secret key, both used for the kernel's SipHash implementation.	✗	✓	✓
inet_ehashfn()	1800	K	Networking (TCP/IP)	Set up a random secret hash seed.‡	✗	✓	✓
flow_hash_from_keys()	4785	K	BPF Flow Dissector	Set up a random secret hash seed.‡	✗	✓	✓
__prandom_timer()	42048	K	General-Purpose PRNG	Seed the kernel's PRNG (prandom) periodically.	✗	✓	✓

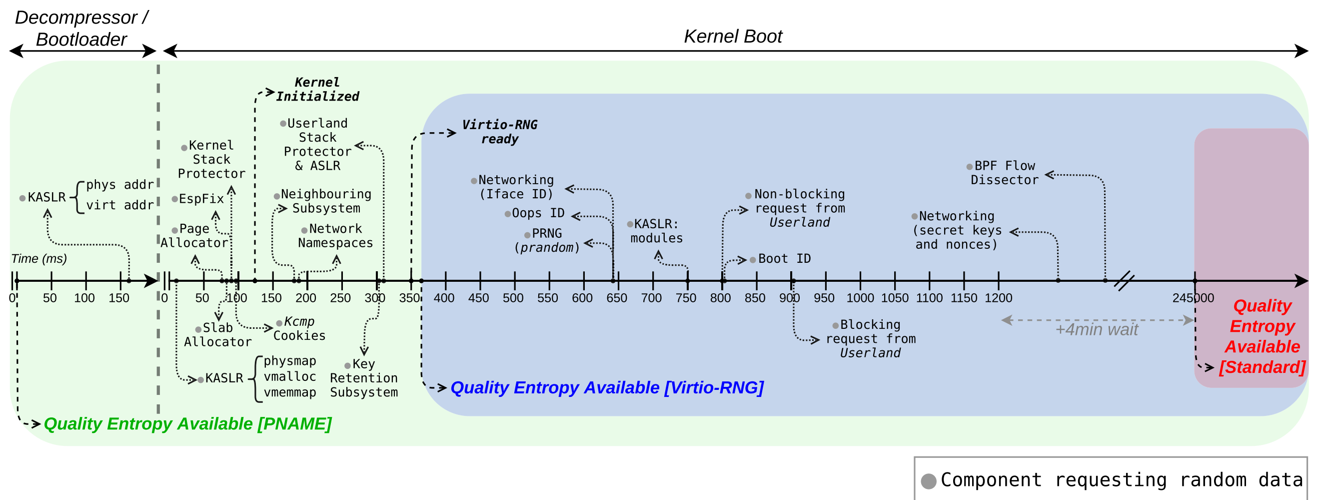

FIGURE 4. Chronology of the kernel components requiring entropy at early boot-time, comparing availability of quality entropy with Standard, virtio-rng and E-Boot.

and 32 GiB of SDRAM memory. The hypervisor used is KVM (Linux kernel 4.19-lts) along with Qemu v4.2.0.

Table 3 shows a list of Linux kernel functions requiring early random data while there is not enough entropy as to initialize the CSPRNG. We have identified a total of 33 kernel functions that are affected by the lack of quality entropy with a standard Linux compilation. When E-Boot is used, all random data requests by those 33 functions are satisfied and the starvation problem is addressed.

In order to assess the effectiveness of our implementation, we compare E-Boot with current state-of-the-art alternatives that provide entropy to the kernel, considering the worst and best-case scenarios. On the one hand, we consider a *standard* kernel without any other entropy source than OS external events as the worst-case scenario. The reason is that although true randomness obtained by OS external events is a basic and built-in entropy source available in any standard Linux kernel, it takes a while to produce enough quality entropy solely with this source. This is because it is slow and unable to produce large streams of random bits in a short period of time. In our experiments up to 4 minutes as shown in figure 4.

On the other hand, the current recommended approach to relax boot-time entropy starvation situations is to use virtio-rng as a virtual HRNG. Virtio-rng is able to provide fast-rate streams of random data to guest virtual machines, and therefore initialize the CSPRNG long before that the standard Linux. Hence, virtio-rng can be considered as the best-case scenario. It is important to note that virtio-rng can be compiled as a loadable kernel module or built-in. Although the default option is to compile it as a module, we have built virtio-rng as built-in to have it available as soon as possible during the kernel boot process. Therefore, we are comparing E-Boot against the best case of the best available solution.

The experiments results are summarized in the table 3. The three last columns indicate which solution provides entropy for each random data request. From the total of 33 affected kernel functions, virtio-rng is able to provide entropy earlier than standard, fixing the last 12 cases (36.36%), but there are still 21 being affected because of the insufficient entropy available. The reason is that virtio-rng needs the kernel to be initialized prior to be ready to supply random data. Therefore, kernel components needing entropy before that point are unable to benefit from it. An example is kernel randomization, which needs entropy just one millisecond after the kernel starts. In contrast, our solution addresses all of them (100%) because it is able to provide the needed entropy from the very beginning, including components requiring entropy in the boot-loader before Linux starts.

Figure 4 represents a chronology of boot-loader and kernel execution time. It shows kernel components requiring entropy at early boot-time, along with important events as the kernel initialization and the moment that virtio-rng is ready. It is important to note that, in this figure, kernel functions have been grouped into components for sake of clarity and only the first request is represented. For example, `Userland Stack Protector & ASLR` consists of several kernel functions (`arch_rnd()`, `create_elf_tables()`, etc.). Also, time count is reset when the kernel is executed because they are two different execution environments.

With the standard configuration, the kernel is not able to generate enough quality entropy until after more than 4 minutes (245 seconds in average). To illustrate the example presented in section VI, we have measured the start-up time of OpenSSH v8.1. Our experiments showed that the SSHd server was around 4 minutes blocked in the `getrandom()` system call (244 seconds in average), and did not start accepting incoming connections until then. Both E-Boot and virtio-rng effectively remove this delay, because when the random requests from SSH arrive (at 1500 ms) both virtio-rng and E-Boot have initialized the CSPRNG, and therefore the userspace entropy requests were successfully satisfied.

TABLE 4. Spatial overhead of the proposed E-Boot with a total of 32 random bytes of random data provided by the hypervisor to be used by the Linux boot-loader and kernel.

Image	Size (bytes)		
	Standard	E-Boot	Overhead
Linux Kernel	58424752	58425088	336
Linux Boot-loader	8817272	8817376	104
Compressed bzImage	8999808	8999808	0

Consequently, user experience is considerably improved, since they can use those userland services as soon as the system boots, without having to wait more than 4 minutes to the CSPRNG initialization.

Although in userspace applications there is not much difference between using virtio-rng or E-Boot, virtio-rng is not able to provide entropy to bootloaders or initialize the CSPRNG at early stages of the Linux boot process. As shown in figure 4, virtio-rng is only able to provide entropy approximately after 360 ms, and therefore is not able to provide entropy to previous Linux components requiring it. Table 3 shows the detailed list of those components. In our experiments 63.63% of the functions were not satisfied by virtio-rng when it was compiled as a built-in, but this number reached 75.75% when it was compiled as a module.

A. SPATIAL OVERHEAD

E-Boot introduces a negligible spatial overhead. The code added to the Linux kernel is simple and small. The pre-reserved areas increase the size of the generated kernel image file. The increment is caused by the addition of the E-Boot section in both images, boot-loader and kernel, which are statically allocated. This overhead is the sum of the header (4 bytes) plus the number of random bytes required.

Table 4 shows the spatial overhead in bytes. In our implementation, we use 36 bytes for the boot-loader (4 bytes for the header + 32 for random data) and 20 bytes for the E-Boot area of the Linux kernel (4 bytes for the header + 16 for random data). Note that the bootloader does not consume the 32 bytes but it contains the 16 bytes of random data of the Linux kernel.

The spatial overhead of the boot-loader is 68 bytes, including the 36 bytes of the E-Boot area. The overhead of the implementation is 316 bytes for the kernel image, including the 36 bytes of the E-Boot area. Therefore, the total spatial cost of the E-Boot areas including the code necessary to implement the E-Boot is 440 bytes, which is less than 0.01% of overhead.

Although the overhead is negligible, we implemented E-Boot within the `init` sections to ensure that the memory is freed after the kernel is booted. Therefore, the actual memory overhead when Linux is running is zero.

B. TEMPORAL OVERHEAD

The temporal overhead introduced by E-Boot is almost negligible. Our experiments showed that 2 μs is the total time overhead when we add the boot-loader and the kernel overhead introduced by E-Boot.

TABLE 5. Detailed temporal overhead of the E-Boot implementation, showing time in nanoseconds.

Order	Description	Hypervisor/ Bootloader/ Kernel	Overhead (ns)
1	Parse the boot-loader image to locate its E-Boot pre-reserved area.	H	492
2	Fill with random data, using `getrandom()` system call.	H	733
3	Load two 8-byte random values for kernel randomization (first use case).	B	91
4	Parse the kernel image to locate its E-Boot pre-reserved area.	B	482
5	Copy remaining 16 bytes from boot-loader to kernel's E-Boot.	B	163
6	Load 16 bytes of random data for kernel's entropy pool (second use case).	K	90
	TOTAL:		2051

The temporal overhead introduced by E-Boot slightly affects the latency of the virtual machine start-up, including the hypervisor and the guest kernel. However, once the guest is booted, the run-time cost is zero, since the code and the memory used are freed. The implementation of E-Boot introduces overhead in the following parts. 1) In the hypervisor, when it parses the guest image file to locate the E-Boot section. 2) In the hypervisor, when it populates the pre-reserved area with random data. 3) In the boot-loader, when it accesses to the pre-reserved memory region containing the random values to load two 8-byte values for kernel randomization. 4) In the bootloader, after kernel decompression the kernel image file is parsed to locate its E-Boot section. 5) In the bootloader, when it copies the remaining entropy to the kernel pre-reserved memory region. 6) Finally, in the kernel, whet it accesses to the 16 random bytes to feed its internal CSPRNG.

After running one million different executions of each of the above parts, as table 5 shows, we obtained a total overhead of about 2 μs. The hypervisor overhead is around 1.2 μs and for the guest is around 826 ns. The kernel only performs a simple load and write operation and its overhead is 90 ns. Therefore, we can conclude that the total temporal overhead introduced by E-Boot is negligible.

C. ENTROPY ASSESSMENT

In virtualization technologies, the hypervisor must be trusted by design. Likewise, E-Boot relies on the hypervisor's ability to provide quality entropy through the pre-reserved area. Consequently, it is the hypervisor's responsibility to ensure the quality of the entropy provided. However, the design of E-Boot contemplates the scenario where the hypervisor cannot provide entropy to their guests. The guests will use the entropy provided by the hypervisor only if the hypervisor set the proper flag that will be later read by the virtual machine.

Therefore, there is no risk or necessity to assess the entropy generated by the hypervisor. In our implementation we use the entropy provided by E-Boot directly to feed the CSPRNG of Linux. To ensure that this process is correct and we are not introducing any weakness into the CSPRNG of Linux,

TABLE 6. NIST statistical tests results showing the p-values and the proportion of passing sequences, with a significance level $\alpha = 0.01$. The total data size is 2.91 GiB, consisting of 1000 samples of $25 * 10^6$ random bit-streams each. To pass a test, the p-value must be at least α, and the proportion of passing sequences must be at least 0.980.

Statistical Test	P-value	Pass Rate	Assessment
Frequency	0.897763	0.990	PASSED
Block Frequency	0.536163	0.991	PASSED
Cumulative Sums	0.905546	0.991	PASSED
Runs	0.319084	0.992	PASSED
Longest Run	0.371941	0.984	PASSED
Rank	0.864494	0.993	PASSED
Discrete Fourier Transform	0.363593	0.991	PASSED
Non-Overlapping Template	0.190726	0.990	PASSED
Overlapping Template	0.016261	0.993	PASSED
Universal	0.253122	0.994	PASSED
Approximate Entropy	0.614226	0.991	PASSED
Random Excursions	0.075354	0.990	PASSED
Random Excursions Variant	0.519006	0.989	PASSED
Serial	0.089620	0.992	PASSED
Linear Complexity	0.707513	0.989	PASSED

we have assessed its output using the NIST Statistical Test Suite (STS) [108].

The null hypothesis is that the Linux random number generator in the guest virtual machines is working correctly after seeding it with the E-Boot provided entropy. For this tests, we have executed 1000 virtual machine instances providing different random values in the pre-reserved area. After seeding and initializing the guest random number generator, we extracted 25 millions of random bits from each virtual machine execution.

Table 6 presents the empirical results of the NIST's STS, showing that all tests were passed successfully and confirming the null hypothesis. P-values of different sub-tests (e.g., Non-Overlapping Template) have been combined using the Fisher's method [109]. The obtained results confirm that E-Boot is not introducing weaknesses in the guest's randomness generation.

X. CONCLUSION

In this paper, we analyzed the boot-time entropy starvation problem and how it affects bootloaders, operating systems and userspace applications. We analyzed the Linux kernel boot process, revealing that the problem not only affects userland applications but up to 33 kernel functions were weakly fed by random numbers.

To overcome this problem, we proposed E-Boot, a novel technique that provides high-quality random numbers to guest virtual machines. E-Boot is the first technique that completely satisfies the entropy demand of virtualized bootloaders, early boot stages of operating systems and userland applications.

We have implemented E-Boot in Linux v5.3 and our experiments showed that it effectively solves the boot-time entropy starvation problem. Our proposal successfully feeds boot time Linux kernel hardening techniques and also reduces to zero the number of userspace blocks and delays. The evaluation results showed that the total time overhead introduced by E-Boot is around 2 μs and has zero memory overhead since the memory is freed before the kernel boot ends, which makes E-boot a practical solution for cloud systems.

REFERENCES

[1] P. Krugman, "Scale economies, product differentiation, and the pattern of trade," *Amer. Econ. Rev.*, vol. 70, no. 5, pp. 950–959, 1980.
[2] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Q. Dang, and K. Pentikousis, "Energy-efficient cloud computing," *Comput. J.*, vol. 53, no. 7, pp. 1045–1051, 2010.
[3] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, and A. Slominski, "Serverless computing: Current trends and open problems," in *Res. Adv. Cloud Comput.* Cham, Switzerland: Springer, 2017, pp. 1–20.
[4] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2017, pp. 405–410.
[5] P. Mell and T. Grance, "The NIST definition of cloud computing," U.S. Dept. Commerce, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. Special Publication 800-145, 2011.
[6] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 2658–2659.
[7] J. Barr. *Firecracker—Lightweight Virtualization for Serverless Computing*. Accessed: Dec. 2019. [Online]. Available: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/
[8] K. Mowery, M. Wei, D. Kohlbrenner, H. Shacham, and S. Swanson, "Welcome to the entropics: Boot-time entropy in embedded devices," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 589–603.
[9] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining your PS and QS: Detection of widespread weak keys in network devices," in *Proc. 21st USENIX Secur. Symp.*, 2012, pp. 205–220.
[10] S. H. Kim, D. Han, and D. H. Lee, "Predictability of Android openssl's pseudo random number generator," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 659–668.
[11] T. Yoo, J.-S. Kang, and Y. Yeom, "Recoverable random numbers in an Internet of Things operating system," *Entropy*, vol. 19, no. 3, p. 113, 2017.
[12] J. Edge. (2019). *Fixing Getrandom, LWN*. Accessed: Oct. 2019. [Online]. Available: https://lwn.net/Articles/800509/
[13] M. Popovic, "Researchers in an entropy wonderland: A review of the entropy concept," 2017, *arXiv:1711.07326*. [Online]. Available: http://arxiv.org/abs/1711.07326
[14] I. K. Sethi, "Entropy nets: From decision trees to neural networks," *Proc. IEEE*, vol. 78, no. 10, pp. 1605–1613, Oct. 1990.
[15] C. Cachin, "Entropy measures and unconditional security in cryptography," Ph.D. dissertation, ETH Zürich, Zürich, Switzerland, 1997.
[16] A. Moreno-Gomez, J. Amezquita-Sanchez, M. Valtierra-Rodriguez, C. Perez-Ramirez, A. Dominguez-Gonzalez, and O. Chavez-Alegria, "EMD-Shannon entropy-based methodology to detect incipient damages in a truss structure," *Appl. Sci.*, vol. 8, no. 11, p. 2068, 2018.
[17] F. Maes, D. Vandermeulen, and P. Suetens, "Medical image registration using mutual information," *Proc. IEEE*, vol. 91, no. 10, pp. 1699–1722, Oct. 2003.
[18] J. Von Neumann, "13. Various techniques used in connection with random digits," *Appl. Math Ser*, vol. 12, nos. 36–38, p. 5, 1951.
[19] J. Edge. (2014). *A System Call for Random Numbers: Getrandom, LWN*. [Online]. Available: https://lwn.net/Articles/606141/
[20] K. Binder, D. M. Ceperley, J.-P. Hansen, M. Kalos, D. Landau, D. Levesque, H. Mueller-Krumbhaar, D. Stauffer, and J.-J. Weis, *Monte Carlo Simulation in Statistical Physics*, vol. 7. Cham, Switzerland: Springer, 2012.
[21] H. S. Sacks, J. Berrier, D. Reitman, V. Ancona-Berk, and T. C. Chalmers, "Meta-analyses of randomized controlled trials," *New England J. Med.*, vol. 316, no. 8, pp. 450–455, 1987.
[22] T. Woollings and J. Thuburn, "Entropy sources in a dynamical core atmosphere model," *Quart. J. Roy. Meteorological Soc.*, vol. 132, no. 614, pp. 43–59, Jan. 2006.
[23] M. Turan, E. Barker, J. Kelsey, K. McKay, M. Baish, and M. Boyle, "Nist special publication 800-90b: Recommendation for the entropy sources used for random bit generation," U.S. Dept. Commerce, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. Special Publication 800-90b, 2012.

[24] S. Müller, "CPU time jitter based non-physical true random number generator," in *Proc. Ottawa Linux Symp.*, 2014, p. 1.

[25] C. Plesiuk, "Evaluating a GPU based TRNG in an entropy starved virtual linux environment," M.S. thesis, Dept. Comput. Sci., Univ. Manitoba, Winnipeg, MB, Canada, 2016. Accessed: Apr. 1, 2020. [Online]. Available: https://mspace.lib.umanitoba.ca/xmlui/handle/1993/31227

[26] J. S. Kim, M. Patel, H. Hassan, L. Orosa, and O. Mutlu, "D-RaNGe: Using commodity DRAM devices to generate true random numbers with low latency and high throughput," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 582–595.

[27] J. Bonneau, J. Clark, and S. Goldfeder, "On bitcoin as a public randomness source," in *Proc. IACR Cryptol. ePrint Arch.*, vol. 2015, 2015, p. 1015.

[28] PaX. (2003). *PaX Address Space Layout Randomization (ASLR)*. Accessed: Sep. 2018. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[29] H. Marco-Gisbert and I. R. Ripoll, "Address space layout randomization next generation," *Appl. Sci.*, vol. 9, no. 14, p. 2928, 2019.

[30] S. Müller. (2018). *Documentation and Analysis of the Linux Random Number Generator*. Accessed: Dec. 2019. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/LinuxRNG/LinuxRNG_EN.pdf?__blob=publicationFile&v=19

[31] D. J. Bernstein, "Chacha, a variant of SALSA20," in *Proc. Workshop Rec. SASC*, vol. 8, 2008, pp. 3–5.

[32] J. B. Lacy, D. P. Mitchell, and W. M. Schell, "Cryptolib: Cryptography in software," in *Proc. USENIX Secur. Symp.*, 1993, p. 1.

[33] D. Kaminsky. *Dakarand 1.0: Revisiting Clock Drift for Entropy Generation*. Accessed: Apr. 1, 2020. [Online]. Available: https://dankaminsky.com/2012/08/15/dakarand/

[34] R. Finnie. *Twuewand, a Truerand Algorithm for Generating Entropy*. Accessed: Apr. 1, 2020. [Online]. Available: https://www.finnie.org/software/twuewand/

[35] F. van Heusden and D. Miller. *Timer Entropy Daemon*. Accessed: Apr. 1, 2020. [Online]. Available: https://vanheusden.com/te/

[36] J. Edge. (2014). *Adding CPU Randomness to the Entropy Pool, LWN*. Accessed: Apr. 1, 2020. [Online]. Available: https://lwn.net/Articles/586427/

[37] H. Zhun and C. Hongyi, "A truly random number generator based on thermal noise," in *Proc. 4th Int. Conf. ASIC (ASICON)*, 2001, pp. 862–864.

[38] N. C. Laurenciu and S. D. Cotofana, "Low cost and energy, thermal noise driven, probability modulated random number generator," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 2724–2727.

[39] A. A. Abbott, L. Bienvenu, and G. Senno, "Non-uniformity in the quantis random number generator," Dept. Comput. Sci., Univ. Auckland, Auckland, New Zealand, Tech. Rep. CDMTCS Research Reports CDMTCS-472 (2014), 2014. Accessed: Apr. 1, 2020. [Online]. Available: http://hdl.handle.net/2292/23904

[40] X. Ma, X. Yuan, Z. Cao, B. Qi, and Z. Zhang, "Quantum random number generation," *NPJ Quantum Inf.*, vol. 2, p. 16021, 2016.

[41] B. Valtchanov, V. Fischer, A. Aubert, and F. Bernard, "Characterization of randomness sources in ring oscillator-based true random number generators in FPGAs," in *Proc. 13th IEEE Symp. Design Diag. Electron. Circuits Syst.*, Apr. 2010, pp. 48–53.

[42] M. Bucci, L. Germani, R. Luzzi, A. Trifiletti, and M. Varanonuovo, "A high-speed oscillator-based truly random number source for cryptographic applications on a smart card ic," *IEEE Trans. Comput.*, vol. 52, no. 4, pp. 403–409, Apr. 2003.

[43] V. Fischer and M. Drutarovskỳ, "True random number generator embedded in reconfigurable hardware," in *Proc. Int. Workshop Cryptograph. Hardw. Embedded Syst.*, in Lecture Notes in Computer Science, vol. 2523. Springer, 2002, pp. 415–430. Accessed: Apr. 1, 2020. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-36400-5_30

[44] B. Jun and P. Kocher, "The Intel random number generator," Cryptogr. Res., San Francisco, CA, USA, White Paper, 1999, vol. 27, pp. 1–8. [Online]. Available: https://www.rambus.com/wp-content/uploads/2015/08/IntelRNG.pdf

[45] 2000. *Intel 82802ab/82802ac Firmware Hub (FWH) Datasheet*. Accessed: Nov. 2019. [Online]. Available: https://www.dataman.com/media/datasheet/Intel/82802Ax.pdf

[46] M. Kinney, "Solving bios boot issues with EFI," Intel Developer UPDATE Magazine, 2000.

[47] (2017). *Unified Extensible Firmware Interface Specification*. [Online]. Available: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_6_Errata_B.pdf

[48] T. Lewis. *Uefi 2.4 Review, Part 12: Random Number Generator Protocol*. Accessed: Dec. 2019. [Online]. Available: https://www.insyde.com/press_news/blog/uefi-24-review-part-12-random-number-generator-protocol

[49] R. N. Akram, K. Markantonakis, and K. Mayes, "Pseudorandom number generation in smart cards: An implementation, performance and randomness analysis," in *Proc. 5th Int. Conf. New Technol., Mobility Secur. (NTMS)*, May 2012, pp. 1–7.

[50] M. Merhi, J. C. Hernandez-Castro, and P. Peris-Lopez, "Studying the pseudo random number generator of a low-cost RFID tag," in *Proc. IEEE Int. Conf. RFID-Technologies Appl.*, Sep. 2011, pp. 381–385.

[51] A. Boorghany, S. B. Sarmadi, P. Yousefi, P. Gorji, and R. Jalili, "Random data and key generation evaluation of some commercial tokens and smart cards," in *Proc. 11th Int. ISC Conf. Inf. Secur. Cryptol.*, Sep. 2014, pp. 49–54.

[52] A. Beaupré. (LWN). (2017). *A Comparison of Cryptographic Keycards*. [Online]. Available: https://lwn.net/Articles/736231/

[53] A. Metrum, "Chaoskey true random number generator," Altus Metrum, Tech. Rep., 2008. Accessed: Apr. 1, 2020. [Online]. Available: https://altusmetrum.org/ChaosKey/

[54] Yubico. (2015). *Yubihsm User Manual*. Accessed: Dec. 2019. [Online]. Available: https://www.yubico.com/wp-content/uploads/2015/04/YubiHSM-Manual_1_5_0.pdf

[55] J. Walker. (2001). *Hotbits: Genuine Random Numbers, Generated by Radioactive Decay*. [Online]. Available: http://www.fourmilab.ch/hotbits

[56] F. van Heusden and D. Miller. *Audio Entropy Daemon*. Accessed: Apr. 1, 2020. [Online]. Available: https://vanheusden.com/aed/

[57] *Video Entropy Daemon*. Accessed: Apr. 1, 2020. [Online]. Available: https://vanheusden.com/ved/

[58] *Lavarng Number Generator*. Accessed: Apr. 1, 2020. [Online]. Available: http://www.lavarnd.org/

[59] P. Warren. *An Entropy Generator Using SDR Peripherals, Including RTL-SDR and Bladerf*. Accessed: Apr. 1, 2020. [Online]. Available: https://github.com/pwarren/rtl-entropy

[60] "Evaluation of via C3 Nehemiah random number generator," Cryptogr. Res. Inc., San Francisco, CA, USA, White Paper, 2003. [Online]. Available: https://www.rambus.com/wp-content/uploads/2015/08/VIA_rng.pdf

[61] (2018). *Intel Digital Random Number Generator (DRNG) Software Implementation Guide*. Accessed: Nov. 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide

[62] E. B. Barker and J. M. Kelsey, "Nist special publication 800-90a: Recommendation for random number generation using deterministic random bit generators," U.S. Dept. Commerce, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep., 2007.

[63] O. Lascu. (2018). *IBM z14 (3906) Technical Guide*. Accessed: Nov. 2019. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg248451.pdf

[64] (2014). *Central Processor Assist for Cryptographic Functions*. Accessed: Nov. 2019. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.csfb300/csfb3za218.htm

[65] (2017). *Power Isac Version 3.0 b*. Accessed: Nov. 2019. [Online]. Available: https://wiki.raptorcs.com/w/images/c/cb/PowerISA_public.v3.0B.pdfs

[66] waywardgeek. *Infinite Noise Trng (True Random Number Generator)*. Accessed: Apr. 1, 2020. [Online]. Available: https://github.com/waywardgeek/infnoise

[67] J. Larson, N. Perlroth, and S. Shane, *Revealed: The NSA's Secret Campaign to Crack, Undermine Internet Security*. New York, NY, USA: ProPublica, Sep. 2013.

[68] J. Ball, J. Borger, and G. Greenwald, "US and UK spy agencies defeat privacy and security on the Internet," The Guardian, 2013.

[69] B. Gellman and E. Nakashima, "US spy agencies mounted 231 offensive cyber-operations in 2011, documents show," The Washington Post, Tech. Rep. The Washington Post–30/08/2013, 2013, vol. 30. Accessed: Apr. 1, 2020. [Online]. Available: http://www.washingtonpost.com/world/national-security/us-spy-agencies-mounted-231-offensive-cyber-operations-in-2011-documents-show/2013/08/30/d090a6ae-119e-11e3-b4cbfd7ce041d814_story.html

[70] S. Gold, "Backdoors to the future?[communications cyber security]," *Eng. Technol.*, vol. 9, no. 9, pp. 59–63, 2014.

[71] S. Sethumadhavan, A. Waksman, M. Suozzo, Y. Huang, and J. Eum, "Trustworthy hardware from untrusted components," *Commun. ACM*, vol. 58, no. 9, pp. 60–71, Aug. 2015.
[72] J. Robertson and M. Riley, "The big hack: How China used a tiny chip to infiltrate us companies," *Bloomberg Businessweek*, vol. 4, 2018. Accessed: Apr. 1, 2020. [Online]. Available: https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tinychip-to-infiltrate-america-s-top-companies
[73] Yubico. *Security Advisory 2019-06-13—Reduced Initial Randomness on Fips Keys*. Accessed: Jun. 13, 2019. [Online]. Available: https://www.yubico.com/support/security-advisories/ysa-2019-02/
[74] C. Keller, F. Gurkaynak, H. Kaeslin, and N. Felber, "Dynamic memory-based physically unclonable function for the generation of unique identifiers and true random numbers," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Jun. 2014, pp. 2740–2743.
[75] S. Sutar, A. Raha, D. Kulkarni, R. Shorey, J. Tew, and V. Raghunathan, "D-PUF: An intrinsically reconfigurable dram PUF for device authentication and random number generation," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 1, p. 17, 2018.
[76] C. Eckert, F. Tehranipoor, and J. A. Chandy, "DRNG: DRAM-based random number generation using its startup value behavior," in *Proc. IEEE 60th Int. Midwest Symp. Circuits Syst. (MWSCAS)*, Aug. 2017, pp. 1260–1263.
[77] F. Tehranipoor, W. Yan, and J. A. Chandy, "Robust hardware true random number generators using DRAM remanence effects," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2016, pp. 79–84.
[78] *Haveged—A Simple Entropy Daemon*. Accessed: Nov. 2019. [Online]. Available: http://www.issihosts.com/haveged/index.html
[79] A. Seznec and N. Sendrier, "Hardware volatile entropy gathering and expansion: Generating unpredictable random number at user level," Inst. Nat. Recherche Informatique Automatique, Le Chesnay, France, Tech. Rep. ISSN 0249-6399, ISRN INRIA/RR-4592, 2002. Accessed: Apr. 1, 2020. [Online]. Available: https://hal.inria.fr/inria-00071993
[80] A. Seznec and N. Sendrier, "HAVEGE: A user-level software heuristic for generating empirically strong random numbers," *ACM Trans. Model. Comput. Simul.*, vol. 13, no. 4, pp. 334–346, 2003.
[81] *Gnu.Org-RNG-Tools*. Accessed: Nov. 2019. [Online]. Available: https://www.gnu.org/software/hurd/user/tlecarrour/rng-tools.html
[82] (Nov. 2019). *Sourceforge—EGD: The Entropy Gathering Daemon*. [Online]. Available: http://egd.sourceforge.net/
[83] D. Silverstone. *Randomsound—Alsa Sound Card Related Entropy Gathering Daemon*. Accessed: Nov. 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man8/randomsound.8.html
[84] *Systemd—Random Seeds*. Accessed: Nov. 2019. [Online]. Available: https://systemd.io/RANDOM_SEEDS
[85] J. Corbet. (LWN). (2016). *Kernel Building With GCC Plugins*. [Online]. Available: https://lwn.net/Articles/691102/
[86] PaX Team. (2012). *[GRSEC] New GCC Plugin: Latent Entropy Extraction*. Aug. 2019. [Online]. Available: https://grsecurity.net/pipermail/grsecurity/2012-July/001093.html
[87] E. Revfy. (2016). (LWN). *Introduce the Latent_Entropy GCC Plugin*. [Online]. Available: https://lwn.net/Articles/688492/
[88] A. Vassilev and R. Staples, "Entropy as a service: Unlocking cryptography's full potential," *Computer*, vol. 49, no. 9, pp. 98–102, Sep. 2016.
[89] D. Kirkland. *Pollinate—An Entropy-as-a-Service Client*. Accessed: Nov. 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man1/pollinate.1.html
[90] D. Kirkland. *Pollen—An Entropy-as-a-Service Web Server*. Accessed: Nov. 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/bionic/man1/pollen.1.html
[91] M. J. Fischer, M. Iorga, and R. Peralta, "A public randomness service," in *Proc. Int. Conf. Secur. Cryptogr.*, 2011, pp. 434–438.
[92] (Dec. 2018). *Virtual I/O Device (Virtio) Version 1.1*. [Online]. Available: https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html
[93] R. Russell, "Virtio: Towards a de-facto standard for virtual I/O devices," *ACM SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 95–103, 2008.
[94] M. T. Jones, "Virtio: An I/O virtualization framework for linux," IBM, Armonk, NY, USA, White Paper, 2010. Accessed: Apr. 1, 2020. [Online]. Available: https://developer.ibm.com/articles/l-virtio/
[95] C. S. Inc. *Xentropyd Source Code*. Accessed: Dec. 2019. [Online]. Available: https://github.com/mirage/xentropyd
[96] M. Dobler. (2017). *Long Startup Delay Caused by Random Generator on V3.5*. [Online]. Available: https://gitlab.alpinelinux.org/alpine/aports/issues/6635
[97] D. Pitsioris. (2018). *Openssh-Server: Slow Startup After the Upgrade to 7.9p1*. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=912087
[98] B. Blank. (2019). *Stalls Due to Insufficient Randomness in Cloud Images*. [Online]. Available: https://lists.debian.org/debian-devel/2019/06/msg00027.html
[99] A. Stamos, A. Becherer, and N. Wilcox, "Cloud computing models and vulnerabilities: Raining on the trendy new parade," BlackHat USA, 2009.
[100] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *IEEE Secur. Privacy Mag.*, vol. 9, no. 2, pp. 50–57, Mar. 2011.
[101] *CVE-2018-1108, MITRE, CVE-ID CVE-2018-1108*. Accessed: Apr. 12, 2018. [Online]. Available: https://access.redhat.com/security/cve/cve-2018-1108
[102] D. Gillmor. (2018). *Linux: Ratelimiting Hides Warnings About Uninitialized CRNG Usage*. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=907060
[103] S. K. Das, K. Kant, and N. Zhang, *Handbook on Securing Cyber-Physical Critical Infrastructure*. Amsterdam, The Netherlands: Elsevier, 2012.
[104] P. Bothner, S. Chamberlain, I. L. Taylor, and D. Delorie, "A guide to the internals of the GNU linker," *Free Softw. Found.*, 2001.
[105] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. USENIX Annu. Tech. Conf., Freenix Track*, vol. 41, 2005, pp. 46–50.
[106] K. Cook. (2013). *Kernel Address Space Layout Randomization*. Accessed: Sep. 2019. [Online]. Available: https://outflux.net/slides/2013/lss/kaslr.pdf
[107] F. Vano-Garcia and H. Marco-Gisbert, "KASLR-MT: Kernel address space layout randomization for multi-tenant cloud systems," *J. Parallel Distrib. Comput.*, vol. 137, pp. 77–90, Mar. 2020.
[108] S. NIST. *800–22: Random Bit Generation—Download Documentation and Software*. Accessed: Apr. 1, 2020. [Online]. Available: https://csrc.nist.gov/projects/random-bit-generation/documentation-and-software
[109] S. R. A. Fisher, *Statistical Methods for Research Workers...-Revised and Enlarged*. London, U.K.: Edinburgh, 1932.

FERNANDO VANO-GARCIA (Graduate Student Member, IEEE) received the B.Sc. degree in computer engineering from the Universitat Politècnica de València, and the M.Sc. degree in cybersecurity from the Universidad Carlos III de Madrid, Spain. He is currently a Ph.D. Researcher with the University of the West of Scotland, U.K. He has participated in research projects as a Co-Investigator. He is the author of many articles of computer security in operating systems and cloud computing. He is also a technical program committee member of international scientific conferences. His main research interests include cybersecurity, memory management in cloud computing, critical infrastructures and virtualization technologies, and among others.

HECTOR MARCO-GISBERT (Senior Member, IEEE) received the Ph.D. degree in computer science, cybersecurity from the Universitat Politècnica de València, Spain. He was a Research Associate with the Universitat Politecnica de Valencia, where he co-founded the Cybersecurity Research Group. He is currently an Associate Professor and a Cybersecurity Researcher with the University of the West of Scotland, U.K. He was a part of the team developing the multiprocessor version of the XtratuM hypervisor to be used by the European Space Agency in its space crafts. He has participated in multiple research projects as a Principal Investigator and a Co-Investigator. He is the author of many articles of computer security and cloud computing. He has been invited multiple times to reputed cybersecurity conferences, such as Black Hat and DeepSec. He has published more than ten Common Vulnerabilities and Exposures (CVE) affecting important software, such as the Linux kernel. He is a member of the Engineering and Physical Sciences Research Council (EPSRC), U.K. He received honors and awards from Google, Packet Storm Security, and IBM for his security contributions to the design and implementation of the Linux ASLR.