Providing Secure Randomness in Applied Cryptography

Hannes Salin
Omegapoint AB
Stockholm, SWEDEN
May 2017
Email: hannes.salin@omegapoint.com

***Abstract*—Securing an application or a network always involves cryptography in one way or another, it is the inevitable parameter for a secure system. System developers and administrators have the tools necessary for doing this right, but what is often forgotten is that strong cryptography relies on being provided with high quality entropy.**

This paper introduces briefly the theory of pseudo random functions, the notion of entropy, how it is used today in industry and how testing of such random data can be done. Finally, some pointers on how to ensure cryptographically secure randomness is presented along a list of best-practices and tools.

I. Introduction

"The security of SSL, like that of any other cryptographic protocol, depends crucially on the unpredictability of this secret key. If an attacker can predict the key's value or even narrow down the number of keys that must be tried, the protocol can be broken with much less effort than if truly random keys had been used. Therefore, it is vital that the secret keys be generated from an unpredictable random-number source." [1]

Randomness, or more strictly the usage of randomly generated bits, is fundamental for all cryptography. For any cryptographic protocol there is a need of random bits: generating nonces, session identifiers, secret keys – all of which relies on the property that it is impossible to make a computation that predicts the generated bit string better than just guessing. Most of the time (due to effectiveness) pseudo random generators are used to feed cryptograhic systems.

However, a weak or predictable generator can have serious consequences [1] [2] [3].

Shannon showed that using a so called one-time pad i.e. computing XOR of the plaintext and a uniformly distributed key of same lenght as the plaintext, ensures perfect secrecy [7]. The impracticality of using keys as long as plaintexts (due to the hardness of finding true random bits to construct large keys) created a need of finding a solution where one could use a shorter key than the plaintext and expand it into arbitrarily length without loosing too much of the security. This implied the invention of pseudo random generators; still relying on a small portion of real randomness but being able to efficiently generate arbitrary length "random looking" bit strings. Today we can use such generators in many different applications, but do we use them in a secure way? And how secure are they?

Programmers often use built-in algorithms to generate random bits, without making any extra effort to ensure the generated bit strings are cryptographically secure. This is probably due to ignorance but even for anyone who carefully studies the subject it may seem non-trivial to do this. In fact, as this paper will show, this is a surprisingly hard task and perhaps even impossible. Moreover, assuming we can generate random bits, can we do this efficiently? Can we test and verify that our bits really are random? This paper will explore ways to do such things, but also give the reader a comprehensive overview and best-practice guiding in the area of generating cryptographically secure randomness.

II. Cryptographic randomness

Section II and III require some knowledge in probability theory and familiarity of theoretical computer science; you may skip to section IV without loosing context.

A. Random and pseudo random functions

Following subsections give a brief introduction to the mathematical framework used for random and pseudo random functions in cryptography. We introduce the notion of computational indistinguashibility between pseudo random probability ensambles, which underlies the construction of *pseudo random generators* (PRG). These functions then extends into implementation of *pseudo random number generators* (PRNG) which are used in applications and hardware. The difference is that a PRNG generally output bytes instead of single bits in every request, which then can be used to represent integers. A PRG is the generalization of a family of algorithms capable of producing pseudo random bits. No matter what our implementations of a PRG produces: integers, bytes, booleans and so forth, they still produce bits or bit-strings just represented as a certain data type.

The rest of this and next subsection is based on the material found in [4], [5] and [6], and the focus has been to distill the most important parts of the theory and exclude all technical and tedious details in order to present a more readable and succinct version.

The first distinction we need to make is the difference

between a *random function* f and a *pseudo random function* f_p. A random function gives an output for which every bit is uniformly distributed, i.e. it is of equal probability that the next outputted bit is 1 or 0. This also implies that given every outputted bit $b_1, b_2, ..., b_n$ from a random function f, there is no better way to predict bit b_{n+1} than guessing. It turns out that implementing a true random function f over n-bit strings in terms of a computer program is completely infeasible due to the enourmous space required, even for rather small n. It stems from the fact that such function would consist of a large table (precisely of size 2^{n2^n}) from which the output strings are chosen from.

Next, a pseudo random function f_p should have the property of *indistinguishability* from a random function. Informally this means that given f and f_p, it should be practically impossible to tell which one is a true random function. We start by defining discrete random variables and probability ensambles:

Definition 1. *A* discrete random variable X *is a function that is defined over a probability space* Ω*, i.e. a set of all possible outcomes for a certain process, and each element associated with a probability. We denote* $Pr[X = x]$ *as the probability that x was chosen from* Ω*.*

Definition 2. *A* probability ensamble $\{X_n\}_{n\in\mathbb{N}}$ *is a sequence of random variables, indexed with* n *meaning that* X_n *is a random variable bounded over n-bit strings.*

An ensamble represents the output from some process generating n-bit strings associated with some probability distribution. Let $\{U_n\}_{n\in\mathbb{N}}$ be the uniform distribution ensamble. One could see this as the possible outputs of all n-bit strings from a random function.

We define a *probabilistic polynomial time* (PPT) adversary A (sometimes also called *distinguisher*) informally as an attacker that uses some probabilistic algorithm, i.e. has the ability to "flip coins" internally in order to generate true random choices, and runs in polynomial time. Given a sequence of bits as input to the adversary's algorithm, it either outputs 1 if it determine that the sequence is uniformely distributed, or 0 if not. Now, using this type of adversary we can define indistinguishability as follows:

Definition 3. *Let* $\{X_n\}_{n\in\mathbb{N}}$ *and* $\{Y_n\}_{n\in\mathbb{N}}$ *be probability ensambles over* $\{0,1\}^n$*. If for every PPT A there exists a negligible function* $\epsilon(n)$ *such that for all* $n \in \mathbb{N}$

$$|Pr[A(X_n = 1] - Pr[A(Y_n = 1]| \leq \epsilon(n)$$

then we say that $\{X_n\}_{n\in\mathbb{N}}$ *and* $\{Y_n\}_{n\in\mathbb{N}}$ *are computationally* indistinguishable*.*

Informally a negligible function ϵ is a function which practically converge to such small numbers that it could be considered to be "almost 0". For completeness we also define a negligible function:

Definition 4. *A function* $\epsilon(x) : \mathbb{N} \to \mathbb{R}$ *such that for every positive integer* c *there exists an integer* N_c *such that for all* $x > N_c$

$$|\epsilon(x)| < \frac{1}{x^c}$$

Definition 5. *We say that* $\{X_n\}_{n\in\mathbb{N}}$ *is* pseudo random *if it is indistinguishable from* $\{U_n\}_{n\in\mathbb{N}}$*.*

The difference between two processes generating n-bit strings, where one is truly random, f, and the other is pseudo random, f_p, is the associated probability ensamble. The process of choosing strings for a pseudo random ensamble can now be formalized in terms of a deterministic algorithm with certain properties that extends f_p into a pseudo random generator:

Definition 6. *A function* $G : \{0,1\}^k \to \{0,1\}^m$ *is called a* pseudo random generator *if following properties are fulfilled:*

1) *G can be computed efficiently (polynomial time)*
2) *G expand its output:* $|G(x)| > |x|$
3) *The ensamble* $\{G(x) : x \in U_k\}_{m\in\mathbb{N}}$ *is pseduo random*

Here x is called a *seed* which is a randomly chosen bit string from U_k. So we still need to get those true random bits! The good part is that we only need a small amount of such bits but the task of harvest seeds may still be a headace.

Now, to ensure (cryptographical) security for a PRG it must be unpredictable for any PPT adversary. In some sense a PRG is unpredictable if no statistical test can take the output and distinguish it from uniformely distributed bits. It turns out that the so called *next-bit test* is a complete test, i.e. if a PRG would pass the next-bit test it will pass any (efficient) statistical test [4].

Definition 7. *(Next-bit test) A PRG* $G : \{0,1\}^k \to \{0,1\}^n$ *is unpredictable if for every PPT A*

$$Pr[G(x) = b_1 b_2 ... b_n,\ \hat{b} = b_i,\ A(b_1, ..., b_{i-1}) = \hat{b}] < \frac{1}{2} + \epsilon(k)$$

where $x \in U_k$ and $\epsilon(k)$ is some negligible function. Again, this is the same as saying that given $i-1$ bits generated from a PRG, any efficient adversary algorithm will fail to do better than just guessing the next bit b_i.

B. Pseudo random number generators

Since it seems to be an impossible task to implement a true random function programmers had to use the next best thing: the PRNG. Note in contrast that a random function f may sometimes be called a *true random number generators* (TRNG), e.g. when we consider a "function" which reads cosmic microwave background radiation and translate the data into bits.

We may view the properties of a PRNG that an attacker given full access to query outputs from a computer, which is either connected to a PRNG or a true random generator, should not be able to tell which generator he is interacting with. Also, for a PRNG to be secure and generate good enough randomness it is of great importance to carefully seed the algorithm. Given a bad (or biased) seed a PRNG would output completely predictable numbers, thus a seed should be extracted from a source of true randomness.

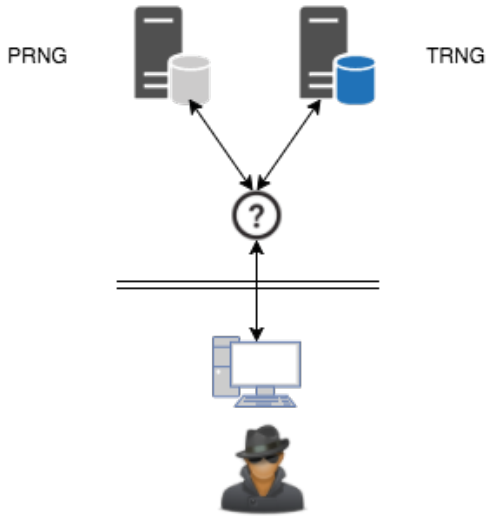

Fig. 1: An attacker should not be able to distinguish a PRNG from a true random generator

Example 1. *Assume we use a random 8-bit integer as seed and an attacker gets a few bytes generated from the PRNG. It would be possible to brute force the PRNG by testing all* 2^8 *possible seeds by computing the first bytes from the PRNG and compare with the attacker's stolen bytes, in order to find the sequence and then be able to completely predict all future bytes. Given that the seed is small and the PRNG never re-seeds, this opens up for above brute force attack, therefore consider at least 128 bits (or 16 bytes) seeds if possible.*

To summarize: we want to engineer a deterministic algorithm that is computationally indistinguishable from a true random function, effeciently to run and expands a small seed of true random data into a large output of pseudo random data. Most importantly, the algorithm need to be resistant to predictions from any adversary meaning the algorithm must pass the next-bit test. Before going into how such algorithms can be constructed we must investigate how seeds can be extracted and how operating systems today handle this task. Next section introduces the concept of entropy and descibes how it is implemented in various systems.

III. Entropy

A. Sources of entropy

The seed is often collected from an *entropy pool* provided by the underlying operating system. The entropy pool may consist of harvested bits from multiple sources such as keyboard strokes, CPU cycles, mouse movements and so forth. Typically such entropy pool is by definition not really true random but deterministic, which is why it would be better to use an entropy pool connected to some hardware that can harvest true random bits, e.g. measurement of cosmic microwave background radiation, electrical noise or quantum phenomenon. These options are often not considered as feasible in practice or may even be susceptible to external malicius influence, why so many systems rely on the operating system's (deterministic) entropy pool.

B. Information theory and entropy

Now, what exactly is entropy? In the field of information theory, Claude Shannon came up with the notion of *measurement of unpredicitability* [7]. Given a bitstring $\sigma = b_1 b_2 ... b_n$, how unpredictable is the information transferred by this string? – that is the measurement of entropy. If we know that σ has a uniform distribution for every bit b_i we need to receive every bit in order to get all information. On the other hand, if we know that the last bit is always 0 then we only need $n-1$ bits to get the same information, thus σ has lower entropy. Similarily when fliping a fair coin, encoding the information of the result must require 1 bit (heads or tails) to transfer. An unfair coin that always gets tails contain no entropy since the information is completely predictable.

Shannon obtained follwing formula for computing the entropy of a random variable X:

$$H(X) = -\sum_{i=1}^{n} Pr[X = x_i] \log_2 Pr[X = x_i]$$

where each $Pr[X = x_i]$ is the probability associated for corresponding $x_i \in X$. For elements that has zero probability it still works with the logarithm since $\lim_{p \to 0+} p \log(p) = 0$.

Note that X is a discrete random variable for a bit string, e.g. the space of outputs from a Linux entropy pool or a key space for a secret AES key.

Example 2. *Consider a fair die (6 sides). Let* X *be the discrete random variable for the outcome of a throw of the die. Since the die is fair, in other words has a uniform distribution, we get that* $Pr[X = x_i] = \frac{1}{6}$*, therefore the entropy of transferring the information of the outcome of a throw of this die is*

$$H(X) = -\sum_{i=1}^{6} \frac{1}{6} log_2 \left(\frac{1}{6}\right) = log_2(6) = 2.584962...$$

thus approximately 2.6 bits are required due to the high amount of unpredictability.

C. Kolmogorov complexity

So far we only considered the unpredictability of bit strings as being chosen randomly from a source. In contrast to the Shannon entropy which assume some probability distribution over the generating soruce, another aspect is to analyze a bit string σ regardless of the source. *Kolmogorov complexity* is a property of the string itself in terms of the length of the shortest possible description of σ with some alphabet [8]. Thus, Kolmogorov complexity has the advantage that it is also defined even if we do not know the probability distribution over the source.

Let σ be a bit string and $d(\sigma)$ the minimal length of σ's description (as in some programming language), measured in bits.

Definition 8. *Given a language L, the* Kolmogorov complexity *of a bit string σ is denoted*

$$K_L(\sigma) = |d(\sigma)|$$

with respect to L, and is the shortest possible program which outputs σ.

Somewhat informal and with lack of technical detail we say that σ is "random" if there is no shorter description than $|\sigma|$, i.e. $K_L(\sigma) \geq |\sigma|$.

Example 3. *Consider a bit string $\sigma = 1010...10$ of length n, then we could write a program such as*

$$print\ "10"\ \frac{n}{2}\ times$$

which for most n would be a shorter description for σ than just print it:

$$print\ "10101010...10"$$

thus $K_L(\sigma) < n$ and not random.

Unfortunately it turns out that it is uncomputable (undecidable) wether a string σ is Kolmogorov random [8], i.e. we cannot construct a general algorithm that decides if a string is random or not. This makes us more prone to investigate statistical tests over random data instead, which many of current randomness testing suites do.

IV. Implementations

A. Microsoft Windows entropy pool and PRNG

The entropy pool and internal working of Microsoft Windows 2000 PRNG was not officially published but later reverse engineered by Dorrendorf, Gutterman and Pinkas [9]. It was shown that multiple sources depending both on user input and system behaviour was used to harvest bits into the pool, for some examples see Table 1. Typically bad sources would be process ids, timestamps, computer names and user names. The reason is that such sources are both easy to guess within small intervalls and even possible to tamper with, and as seen in Table 1 all of these parameters were included. The paper also report that RC-4 and SHA-1, both now deprecated due to the possibility of real-world attacks, was used in the PRNG seeded by the entropy pool. However, no further analysis could determine if that would be a problem for the PRNG. A possible attack on the PRNG includes computing future outputs if the internal state of the PRNG is known, hence no forward secrecy.

Source	Bytes requested
CircularHash	256
KSecDD	256
GetCurrentProcessID()	8
GetCurrentThreadID()	8
GetTickCount()	8
GetLocalTime()	16
QueryPerformanceCounter()	24
GlobalMemoryStatus()	16
GetDiskFreeSpace()	40
GetComputerName()	16
GetUserName()	257
GetCursorPos()	8
GetMessageTime()	16

TABLE I: Some of the reversed engineered entropy sources found in Microsoft Windows 2000

Microsoft Windows used the *Cryptographic Application Programming Interface* (CryptoAPI) to provide cryptographic functions to programmers such as encryption, decryption, hash functions and pseudo random number generators [16]. Due to the vulnerabilities found in [9] the Microsoft Windows Vista release had an update to the API, the *Cryptography API: Next Generation* (CNG) which replaced CryptoAPI. This replacement contained several new algorithms in which one was the Dual_EC_DRBG; later discovered to be faulty due to a kleptographic backdoor. Also, CNG provides the ability to replace the default PRNG with a user specific one.

The CNG uses *Cryptographic Service Providers* (CSPs) which are modules installed in the operating system and implements all cryptographic functions, i.e. if an application calls the CNG it will in turn call a CSP's implementation of that specific function (for a list of CSPs see [17]).

For the PRNG specifically, according to Microsoft [16] the CNG stores an intermediate random seed for each user which is used when creating a seed to the the PRNG: *"...a calling application supplies bits it might have – for instance, mouse or keyboard timing input – that are then combined with both the stored seed and various system data and user data such as the process ID and thread ID, the system clock, the system time, the system counter, memory status, free disk clusters, the hashed user environment block. This result is used to seed the pseudorandom number generator (PRNG)."*

Unfortunately no known attempts of reverse engineering newer versions of Microsoft Windows has been published, but a quite recent attempt to investigate the quality of the PRNG has been done [18]. Microsoft Windows implements cryptographic algorithms following standards by FIPS 140-2 and NIST SP800-(90A,90B), and NIST has a list of documents describing those implementations [10] [11] [12]. These documents provide technical details for describing the architecture of entropy gathering and associated PRNGs for Windows 7 and 10 which is probably today's most used Microsoft Windows versions. Since no objective research has analyzed these systems (at least in terms of what is published today), the following description gathered from above mentioned

documents should be valued carefully by the reader.

The cryptographic module `CNG.sys` is accessed by both kernel- and user mode applications when PRNG material is needed, the former mode via `BCrypt` or legacy FIPS APIs, and the latter via the `SystemPrng` interface. `CNG.sys` itself gets feeded with entropy from several entropy APIs. Another module, the *Microsoft Windows Cryptographic Primitives Library*, is for user mode applications only and is accessed via dynamic linked library *bcryptprimitives.dll*. Both modules serves pseudo random data by identical services.

The Entropy pool is gathering bits from the Trusted Platform Module (TPM) when present but also by periodically querying the values from a large set of operating system variables. Many of these are the ones listed in Table 1, e.g. process id and current local date and time. Interestingly, other variables such as *"a hash of the environment block for the current process", "Some hardware CPU-specific cycle counters"* and *"processor power information of Thermal Limit Frequency"* are listed, but also (almost) completely non-entropy sources such as number of CD-ROMs, number of floppies and time zone.

`BCryptGenRandom` is an exported function for user mode, found in `bcryptprimitives.dll` module. This function is the interface betweeen the programmer and the `SystemPrng` interface.

A function called `EntropyProvideData` is used for providing entropy to the entropy pool [10]. This function has one parameter `entropyEstimateInMilliBits` which implies that Microsoft Windows may have some sort of entropy estimator algorithm in order to ensure that PRNGs are feeded with "good enough" bytes.

Each time `CNG.sys` is loaded it performs a set of self-tests, both on several cryptographic encryption/decryption algorithms and hash functions, but also the SP800-90 AES-256 counter mode DRBG and SP800-90 Dual-EC DRBG are reseeded.

B. Linux entropy pool and PRNG

The Linux PRNG is part of the kernel and incorporates three entropy pools: *input pool, blocking pool* and *non-blocking pool*. These pools is the embodiment of the internal state of the PRNG and the input pool asynchronously and independently harvest bits from system events inside the kernel. Sources of entropy includes user inputs (such as keyboard and mouse movements), disk timings and interrupt timings. Every such event is captured by three 32-bit values: a value specific to event type, CPU cycle count and *jiffies* count which is an interupt counter from kernel boot time. Each such value do not consist of 32 bits entropy; empirical tests showed that jiffies and cycle counts only gave 3.4 and 14.8 bits of entropy respectively [14]. Harvested bits are processed in a linear mixing function before being distributed into the blocking and non-blocking pool.

The Linux user space has two device interfaces that reads random bytes from the blocking and non-blocking pools: `dev/random` and `dev/urandom` respectively. Each entropy pool transfer its content as seeds to a PRNG before outputting to these interfaces. The blocking interface, `dev/random`, simply halt until enough entropy is collected from the input pool. This block is triggered by an entropy esitmation algorithm computed by the kernel, as stated in [14]: *"Entropy estimation is based on a few reasonable assumptions. It is assumed that most of the entropy of the input samples is contained in their timings. Both the cycle and jiffies counts can be seen as a measure of timing, however the jiffies count has a much coarser granularity. The Linux PRNG bases its entropy estimation on the jiffies count only, which leads to a pessimistic estimation"*.

There is a handle called `entropy_avail` to check currently available entropy estimation in the blocking pool. Following command will show the entropy estimation continuously in a system:

```
watch -n 1 cat /proc/sys/kernel/random/
entropy_avail clear
```

When the Linux User space requests random bytes from either device interface the entropy pools process the required bits in a somewhat complicated "mix and hash" function using SHA-1, where parts of the data is injected back into the entropy pool. The generated bits are requested in 10 byte blocks, and if the request size is not a multiple of 10 the last block is truncated. SHA-1 is used several times from the point where entropy bits are harvested up till the output stage. The final steps consists of using XOR to scramble the output further.

The output can be read from any of the interfaces, but as mentioned earlier `dev/random` will block until the estimation is high enough. The blocking property often lead to the fact that developers and system administrators configure applications to use the non-blocking device `dev/urandom` instead. A common misconception about the differences of the blocking and non-blocking devices is that the latter is "less secure" since it just continues outputting bytes regardless of the entropy estimation, but all bytes comes from the same source internally, it just may happen that `dev/random` need to get more bytes for the estimation threshold. One could argue wether the estimation itself is correct since it predominantly measures the timing as a factor and not so much the actual source from where the bits were generated.

Note that new kernels (starting from version 4.8) have different implementations between the blocking- and non-blocking devices [15], i.e. the non-blocking device uses a symmetric cipher called ChaCha20 to generate data. Although this potentially more secure change in the kernel, in most cases many systems in "the wild" tend to not upgrade frequently, thus using the old way of generating data.

On the other hand, the Linux PRNG is considered more secure even since kernel versions 2.6.30.7 [14] but still has

some flaws, e.g. SHA-1 is still used (but unknown if the impact for the PRNG is fatal) and a Linear Feedback Shift Register (LFSR) PRNG which is used internally does not provide a full period; if the period is low the generator will start over computing same bits more frequently. Also, a user may write data to `dev/random` in order to provide more bytes to the pool, but this will not add any entropy [19] and may affect the output from the PRNG in a highly risky way since a user could infect the byte stream with low entropy and repetetive patterns.

Since Linux is part of the open source community the entropy pool architecture and PRNG can be reviewed independently and allow researchers to analyze the security in more detail. In contrast to Microsoft Windows no validation or compliance to NIST (or equivalent) is established.

C. Mac OS and iOS entropy pool and PRNG

Mac OS and iOS implement interfaces `/dev/random` and `/dev/urandom`, both using the same algorithms. Previously a Yarrow-based algorithm was used as PRNG [21] but according to Apple at least the iOS 9.3 (and later) PRNG is based on a CTR_DRBG (Counter mode Deterministic Random Byte Generator) standardized by NIST in SP800-90A. This algorithm is based on a block cipher in counter mode, most likely AES [23]. Apple's security white paper continues: *"System entropy is generated from timing variations during boot, and additionally from interrupt timing once the device has booted. Keys generated inside the Secure Enclave use its true hardware random number generator based on multiple ring oscillators post processed with CTR_DRBG"* [22]. The Secure Enclave is a coprocessor in Apple A7 (or later) that includes a hardware random generator.

For Mac OS, examination of the XNU open source github repository (Apple's underlying kernel) shows that the Yarrow-CoreLib files from the `dev/random` is removed since version 10.11.

For iOS devices an API called *Randomization Services* can be used, which has a function `SecRandomCopyBytes` that read bytes from `/dev/random`. For Mac OS applications the Apple developer documentation recommends reading directly from `/dev/random` or `/dev/urandom` to ensure cryptographically secure randomness.

Naturally, mobile phones could be regarded as low-entropy devices since the hardware is more limited compared to web servers. In fact, in 2012 it was observed that many embedded devices were generating low-entropy RSA keys, most likely due to a faulty PRNG flow during boot [24]. A typical attack scenario for mobile devices is kernel-level exploatation, i.e. accessing the kernel address space. Mitigating these type of attacks require randomness in order to shuffle the address space (ASLR) and make it unpredictable for an attacker to find certain spaces, see for example the *Common Vulnerabilities and Exposures* findings CVE-2016-4655, CVE-2016-4656 and CVE-2016-4657 regarding iOS kernel leakage attacks. This implies that randomization is even more crucial in mobile devices compared to desktop and web applications.

D. Android entropy pool and PRNG

The Android operating system builds on the Linux-kernel, which is open source and largely widespread in many devices. Regarding the entropy pool and Linux PRNG the same principles described in section IV B applies. In both Andorid 4.2 and 4.3 it was possible to determine the state of the PRNG given a leaked value from the non-blocking pool at boot time and several attacks were presented, including bypassing stack canarys [25] [26]. The attacks utilized the fact that Android's low boot-time did not enable the PRNG to initialize properly which in turn made it possible to guess the canary value and thereby bypass it. A stack canary is a certain value inserted into memory in order to discover buffer overflow attacks (verification of the canary value checks if an overflow occured or not). To mitigate such attacks, either change how the initialization is performed or feed external entropy (e.g. from web APIs) into the boot start phase [25].

E. Entropy in virtualization

Virtualization is a common activity today, both as in personal usage – the ability to have several operating systems available at hand at any time – and as a mechanism to deploy applications, e.g. much of cloud based activitiy relies on virtualization in one form or another. The basis is that a host operating system, i.e. the underlying computer's operating system, has a *hypervisor* which provides a virtualization layer on which *virtual machines* (VMs) can be installed. A VM consists of a guest operating system that runs on emulated hardware. A typical feature for a VM manager is to create snapshots of a VM, i.e. a clone which copies the complete internal state and memory, including any entropy that may have been collected.

Given the structure of how virtualization works, two main problems regarding entropy collection should be considered:

1) Is the hypervisor streaming entropy from the host operating system to all VMs?
2) Does every snapshot contain the exact state for the entropy pool / PRNG ?

Naturally, for the first case it could be problematic if each VM shares the host system's entropy pool when the host system is relatively inactive, thus not generating much entropy. The other way around, when each VM uses its own entropy pool but due to the virtualization layer, not much entropy is collected because many entropy soruces are not used/excluded may also be problematic. The second problem is quite fatal and previous research shows practical attacks due to such vulnerabilities including compromising TLS [27]. Security experts recommend users to surf the net in a VM in order to increase security due to sandboxing and more control of the web-environment, but as shown in [27] it happens that if a TLS pre-master secret is generated and then a snapshot is

taken, the same pre-master secret is used for each cloned VM. This means that the same symmetric key is used in multiple HTTPS-connections, at least one for each cloned VM, since the pre-master secret is a randomly chosen value which derives a key for encryption used in the TLS protocol.

Another aspect is (as usual) the performance of generating entropy in relation to the quality. When studying the entropy generation process using Linux-based guest operating systems in a cloud environment it was shown that it had significant slower performance due to virtualization [28].

V. Cryptographically secure pseudo random number generators

A. Overview

As mentioned previously, a PRNG should be indistinguishable from a generator of uniformly distributed bit strings. Moreover, a PRNG is considered cryptographically secure if it also is resistant against predictability, e.g. given the current state of the PRNG an adversary should not be able to reconstruct previous bits or prediciting the next bit (recall the next-bit test).

Now, in contrast to cryptograhically secure PRNGs, we mention the (unfortuntely) more common generators which are not considered secure at all. For example, many programming languages implement some of these algorithms in their standard libraries [20] [31]. Two common types are the linear congruential generators (LCG) and linear feedback shift registers (LFSR). Due to the linearity an attacker can perform algebraic attacks and solve equations in order to break the PRNG. For the curious reader on how these algorithms work, see Knuth's work [33].

B. Cryptographically secure generators

We distinguish a PRNG from a soruce of entropy, whereas the latter is considered to not necessarily be an algorithm but rather a physcial source such as a reader of a harddisk's cache timings or cosmic microwave background radiation. This section only considers PRNGs (since we could also use sources of entropy directly if the entropy can be estimated accurately enough).

In this paper we present a few considered secure PRNGs, some for which formal security proofs exists. Note that these proofs are based on provable security which only guarantee security within some specific model and by reduction to a assumed computationally hard problem; no proof of the hardness for that considered problem exists.

The *Blum Blum Shub generator* (BBS) is a PRNG for which the security proof [34] is based on the quadratic residuosity problem: given two integers a and N, it is belived hard to decide if a is a qudratic residue modulo N, i.e. $a \equiv x^2 \bmod N$ for some integer x. Now, the BBS generator takes the form

$$x_{n+1} = x_n^2 \bmod N$$

where $N = pq$ for some primes p and q. The output from BBS is the parity bit from the bit encoding of x_{n+1}.

Another provable secure PRNG is the *Blum-Micali generator* (BM) [35] which relies on the discrete logarithm problem: given $a^x \bmod N$ it is belived hard to find x. Many schemes and protocols in cryptography builds upon this hard problem and security proofs reduces to that if an adversary can break the protocol it would imply that the attack also gives an algorithm for solving the discrete logarithm problem. The BM generator is as follows: let p be a prime and g a so called primitive root modulo p, then the generator is described as

$$x_{n+1} = g^{x_n} \bmod p$$

and the output is also parity bits similiar as in the BBS generator: x_i transforms to output bit 1 if $x_i < \frac{p-1}{2}$, otherwise 0.

Another approach is to use encryption schemes as PRNGs, e.g. the *RSA generator* which is provably secure under the hardness of RSA [36], which in turn means that we belivie it is computationally hard with integer factorization. The idea of the RSA generator is to iterate the RSA encryption mapping, i.e. computing

$$x_{n+1} = x_n^e \bmod N$$

where e is the public exponent (key) and $N = pq$ for primes p and q. Again, the output is a subset of the bit representation of x_i, e.g. a number of the least significant bits for example. In practice, such generator should perform poorly since asymetrical encryption in general suffers from performance issues. Instead, block ciphers and hash functions can also be used as PRNGs which has significant computantional advantages. On the other hand, many hash- och block cipher based PRNGs are proved under the assumption that the corresponding block cipher is a pseduo random permutation [37], which more or less means a circular argument.

The *Yarrow algorithm* is a well known family of PRNG algorithms devised by Kelsey, Schneier and Ferguson [38] which has been incorporated in iOS and Mac OS X. The Yarrow algorithms consists of two components: a hash function and a block cipher, specifically for Yarrow-160 which is found in implementations, SHA-1 and 3DES with triple keys are used. To generate output the block cipher is used in counter mode, and throughout the generation the algorithm reseed itself frequently. Yarrow also uses entropy estimation internally in order to collect "sufficient enough" randomness so that the state of the PRNG can be stable from a security point of view; to handle this a set of pools are used for entropy collection. Estimating entropy is not a trivial task and improvements have been done which resulted in another PRNG called *Fortuna*, also devised by Schneier and Ferguson [39].

VI. Entropy testing

A. Testing

It is fairly easy to implement and integrate cryptography in software, but ensuring the quality of the underlying entropy feeded into the software is more complicated. In an ideal world the development of secure software using cryptography should consist of thoroughly and carefully anlysis of entropy sources and random data usage. There are some attempts for this type of analysis and NIST even have some requirements for entropy testing in SP 800-90B [23]. Furthermore, no official tests for entropy quality exists but some independent tools are available such as Diehard [40], Dieharder [41], ENT [42], NIST test suite [43] and TestU01 [44]. These tests however are mostly statistical tests that does not take in consideration the underlying distribution of an entropy source (or that the distribution may change over time), or simply assumes a uniform distribution. There are attempts for computing predictions of data i.e. complementary to statistical testing also try to predict future outputs given a set of data. If such tests could be performed we may get closer to evalute cryptographically secure PRNGs and entropy sources more adequately. There is research investigating this type of predictive analysis and the method in general is called *predictors* [45], [46] and [47].

In summary, we can perform a lot of statistical tests and apply predictors if available, but more research is needed and the phenomenen of entropy testing is not yet fully understood, thus we cannot get exact results weather our entropy sources or PRNGs are cryptographically secure. On the other hand we may use what tools exist and put some effort into analysis of statistical- and predictable patterns. If we can detect recurring patterns and eliminate them we have at least made it a bit harder for an attacker. Therefore, it is of high importance that we test our pseudo random generators and entropy pools before deploying high-level security applications using cryptography. Furthermore, we must **take time** to investigate, implement and verify that we use sufficient entropy pools and PRNGs.

VII. Generating randomness

A. Entropy tools

Complementary to testing entropy sources using tools and methods described in section VI, one can deploy tools to (potentially) increase the quality if needed. Following section gives a short description of what tools can be used, but no deeper analysis is given into how well they perform over time (that would be a separate paper in itself).

Disclaimer

Not all tools listed here are tested properly, thus only given as examples in order to give the reader an overview of what tools are available.

For Linux environments specifically there exists some tools that provide entropy to `dev/random` and `dev/urandom` interfaces by measuring distinct sources and convert the output into entropy bits:

1) The *timer entropy deamon*, `timer_entropyd`, uses timers to measure sleep jitters and require no additional hardware (only relies on the underlying CPU clocks).
2) The *clock randomness gathering deamon*, `clrngd`, which measure timing-differences between different physical high-frequency clocks. All data is tested against FIPS standard tests before adding any entropy.
3) The *randomsound* application that uses the soundcard to extract bits, thus require a soundcard.
4) The *Haveged* algorithm [29] use timings between different hardware interrupts but also cache misses and similiar. Implementations are available for both Linux and Windows.

Another solution, more used in industry and for real-world applications, is the *Hardware Security Module* (HSM). These are often hardware devices securing crypto keys and/or computes cryptographical processing. A HSM should then be able to generate entropy by some hidden and secured process and feed a system securely. Note that if an application or system need strong cryptography it is most often for generating keys, thus being able to generate good entropy is implicit by the key generating process in the HSM.

Best practice for using third party entropy software

The author's recommendation is not to deploy any of these presented tools unless the needed entropy is used for other contexts than cryptography; if strong cryptography is deployed one should not need that many bytes for seeding material. In such circumstiances for large scale systems a set of HSMs should be deployed instead to provide more entropy.

B. How to generate secure randomness in Java

There are two ways of producing pseudo random data in Java, either using the weak `Random` class or the more secure `SecureRandom` class. We only discuss `SecureRandom` since `Random` should never be used in the context of security and cryptography due to weak PRNGs.

Algorithm	OS
NativePRNG	Solaris, OS X, Linux
NativePRNGBlocking	Solaris, OS X, Linux
NativePRNGNonBlocking	Solaris, OS X, Linux
SHA1PRNG	Solaris, OS X, Linux, Windows
Windows-PRNG	Windows
PKCS11	Solaris

TABLE II: Available PRNG algorithms for different operating systems in Java

`SecureRandom` is dependent on which implementation is used; a *Service Provider Interface* (SPI) called SecureRandomSpi specifies the PRNG to use. Note that a SecureRandomSpi may not be stated explicitly when creating an instance of `SecureRandom`. The SPI is associated to a certain provider, e.g. for Sun's Java implementation the SUN provider is default. If no provider is explicitly given, a list of all available providers is traversed in order to find the default or the most preferred one. The default choice for Unix-like operating systems is NativePRNG and for Microsoft Windows SHA1PRNG. For a complete list of providers see [30].

Each implementation of `SecureRandom` have a `nextBytes()` method that outputs the generated pseudo random bytes. The NativePRNG and NativePRNGNonBlocking outputs from `dev/urandom` directly, NativePRNGBlocking from `dev/random` and SHA1PRNG from a SHA-1 based function.

The `getInstanceStrong()` method was introduced in Java 8 which returns an instance of the strongest `SecureRandom` implementation available. Oracle recommendations are: *"It [getInstanceStrong()] should be used in cases when you need to create a high-value and long-lived secret, such as an RSA private and public key pair."* [30]

Example 4. *Assuming we operate in a Linux environment, following examples show how to correctly instantiate and seed the SecureRandom class. To use the non-blocking device dev/urandom:*

```
SecureRandom random = new SecureRandom();
byte[] values = new byte[20];
random.nextBytes(values);
```

To use the blocking device dev/random:

```
SecureRandom random =
SecureRandom.getInstanceStrong();
byte[] values = new byte[20];
random.nextBytes(values);
```

To explicitly use a preferred PRNG:

```
SecureRandom random =
SecureRandom.getInstance("SHA1PRNG",
"SUN");
byte[] values = new byte[20];
random.nextBytes(values);
```

Note that 20 bytes are used to perform an initial read. This is to enforce the PRNG to internaly seed itself with enough entropy from either `dev/random` or `dev/urandom` if using Unix-like operating systems or the Windows CryptAPI: CNG for Microsoft Windows. This method is to prefer instead of manually "find" entropy to seed the PRNG; it is possible to configure the seeding mechanism which is somewhat provider dependent, but this is not recommended.

Best practice for Java's SecureRandom

1) Always specify the exact PRNG and provider that you wish to use
2) Always call `SecureRandom.nextBytes()` immediately after instantiation, to force the PRNG to seed itself securely
3) Periodically reseed your PRNG if it is used frequently (if good enough seeding material is available). One way to reseed is to periodically throw away the existing `SecureRandom` instance and create a new one since this will generate a new instance with a new seed

C. How to generate secure randomness in C++

We only consider C++ version 11 and newer due to the introduction of `random.h` header. The concept of generating random bytes in the standard library builds upon using an *engine* together with a *distribution*. This is a natural approach since a random variable that generates output must be associated to a certain distribution. The most interesting distribution is `uniform_int_distribution` which produces integers uniformly distributed within a closed interval specified by the programmer. There is a variety of PRNGs included in the engine concept, however the `random_device` class is implementation-specific and functions as an entropy source: `random_device` may use the underlying (possibly blocking) devices such as `dev/random` for Unix-like operating systems, or it may use something completely different. Thus if porting an applictation to another operating system one needs to fully investigate how `random_device` is implemented to avoid failure. Now, the constructor is defined as

```
explicit random_device(const std::string&
token = /*implementation-defined*/ );
```

where `token` is a string for which a programmer can try to prioritize which device to read from, e.g. libc++ and libstdc++ uses `dev/urandom` as default and if the CPU instruction RDRND is available libstdc++ will use that as default.

Example 5. *Following two examples shows how the `random_device` can be used, first using default source, e.g. RDRND or `/dev/urandom`:*

```
std::uniform_int_distribution<int> d(0,
2147483647);
std::random_device rd1;
std::cout << d(rd1);
```

and then explicitly state a source:

```
std::uniform_int_distribution<int> d(0,
```

```
2147483647);
std::random_device rd2("/dev/random");
std::cout << d(rd2);
```

The ISO C++ Standard states that it is not required for the class to generate uniformly distributed or cryptographically secure bits, but Microsoft MSDN for example guarantee non-determinism [16]. According to reference [32]: *"...certain library implementations may lack the ability to produce such numbers [uniformly distributed] and employ a random number engine to generate pseudo-random values instead. In this case, entropy returns zero."* The function `entropy` is part of the `random_device` class and returns a value estimating the entropy in bits. Beware that this function is not fully implemented in some standard libraries, e.g GNU libstdc++ and LLVM libc++ always return zero even though the device is non-deterministic. Also, Microsoft Visual C++ implementation always returns 32, and `boost.random` returns 10 [32].

Another apporoach is to read directly from `/dev/urandom` or `/dev/random` if such device interfaces are available in the operating system.

The Microsoft Windows CryptoAPI: CNG have the `BCryptGenRandom` function as mentioned earlier (previously the Crypto API used `CryptGetRandom`). It may be convenient to interact directly with the cryptographic API if using Microsoft Windows. `BCryptGenRandom` function is defined as:

```
NTSTATUS WINAPI BCryptGenRandom(
_Inout_ BCRYPT_ALG_HANDLE hAlgorithm,
_Inout_ PUCHAR pbBuffer,
_In_ ULONG cbBuffer,
_In_ ULONG dwFlags
);
```

where the parameter `pbBuffer` stores the received pseudo random bytes and `hAlgorithm` specifies which PRNG to use (by the means of a BCryptOpenAlgorithmProvider object). Following PRNGs are available:

PRNG	Implementation standard
BCRYPT_RNG_ALGORITHM	FIPS 186-2 FIPS 140-2 NIST SP 800-90
BCRYPT_RNG_DUAL_EC _ALGORITHM	NIST SP 800-90
BCRYPT_RNG_FIPS186_DSA _ALGORITHM	FIPS 186-2

TABLE III: Available PRNG implementations in Windows cryptography API

BCRYPT_RNG_ALGORITHM is based on the AES block cipher with counter mode, beginning from Windows Vista with SP1 and Windows Server 2008, and BCRYPT_RNG_DUAL_EC _ALGORITHM has been removed beginning from Windows 10 [16].

Best practice for generating randomness using C++

1) Investigate what source `random_device` uses
2) Use `uniform_int_distribution` togheter with `random_device`
3) Consider read directly from `dev/random` if using Linux
4) Periodically reseed your PRNG if it is used frequently

D. Conclusion

This paper only gives a first introduction to several different areas spanning over entropy and randomness used in a security context; each section could easily be further developed into white papers on their own. The aim is to inform and introduce anyone developing, auditing or code reviewing secure software, into the high risk of undermine cryptography by using bad sources of entropy.

Most importantly one needs to be aware of how randomness is collected and used in software. Spending some effort on statistical and predictive testing is recommended but cannot guarantee security. We may see further developments in this domain but today it is nothing we can rely on.

REFERENCES

[1] Goldberg I., Wagner D., *Randomness and the Netscape browser*, Dr. Dobbs Journal, 1996, pp. 6670

[2] Gutterman Z., Malkhi D. (2005) *Hold Your Sessions: An Attack on Java Session-Id Generation.* In: Menezes A. (eds) Topics in Cryptology CT-RSA 2005. CT-RSA 2005. Lecture Notes in Computer Science, vol 3376. Springer, Berlin, Heidelberg

[3] CVE-2008-0166, *Debian generated SSH-Keys working exploit* [Online]. Available: http://www.securityfocus.com/archive/1/archive/1/492112/100/0/threaded

[4] Bellare M., Rogaway P., *Introduction to Modern Cryptography*, 2005. Lecture notes.

[5] Bellare M., Goldwasser S., *Lecture Notes on Cryptography*, 2008, Lecture notes.

[6] Stinson R D., *Cryptography, theory and practice*, 2006, 3d edition

[7] Shannon C., *Communication Theory of Secrecy Systems*, 1949, Bell System Technical Journal

[8] Grunwald P., and Vitanyi P., *Shannon Information and Kolmogorov Complexity*, 2010

[9] Dorrendorf L., *Cryptanalysis of the Windows Random Number Generator*, 2007

[10] NIST, *Microsoft Windows 7 Kernel Mode Cryptographic Primitives Library (cng.sys) Security Policy Document*, 2013

[11] NIST, *Cryptographic Primitives Library (bcryptprimitives.dll and ncryptsslp.dll) in Microsoft Windows 10*, 2016

[12] NIST, *Kernel Mode Cryptographic Primitives Library (cng.sys) in Microsoft Windows 10*, 2016

[13] Gutterman Z., Pinkas B., Reinman T., *Analysis of the Linux Random Number Generator*, 2006

[14] Lacharme,Patrick, Rck, Andrea, Strubel, Vincent and Videau, Marion, *The Linux Pseudorandom Number Generator Revisited*, published 2012

[15] Linux kernel, feb 2017. https://github.com/torvalds/linux.

[16] Microsoft MSDN, *CryptGenRandom*, https://msdn.microsoft.com

[17] Microsoft MSDN, *Cryptographic Service Providers*, https://msdn.microsoft.com/en-us/library/aa386983.aspx

[18] Alzhrani, Khudran and Aljaedi, Amer *Windows and Linux Random Number Generation Process: A Comparative Analysis*, published 2015

[19] Linux man-pages, random, *random(4)*, http://man7.org/linux/man-pages/man4/random.4.html

[20] Linux man-pages, rand, *rand(3)*, http://man7.org/linux/man-pages/man3/rand.3.html

[21] Mandt, Tarjei, *Revisiting iOS Kernel (In)Security: Attacking the early random() PRNG*, 2016

[22] Apple, *iOS Security Guide White Paper*, 2016, https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[23] National Institute of Standards and Technology, Special Publications, http://csrc.nist.gov/publications/PubsSPs.html

[24] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J Alex Halderman. *Mining your Ps and Qs: detection of widespread weak keys in network devices.* In Proceedings of the 21st USENIX conference on Security symposium, Security12, page 35. USENIX Association, 2012

[25] Kaplan, David , Kedmi, Sagi, Hay, Roee and Dayan, Avi, *Attacking the Linux PRNG on Android: Weaknesses in Seeding of Entropic Pools and Low Boot-time Entropy*, Proceedings of the 8th USENIX Conference on Offensive Technologies, WOOT'14, 2014

[26] Ding, Yu, Peng, Zhuo, Zhou, Yuanyuan and Zhang, Chao *Android Low Entropy Demystified*, IEEE International Conference on Communications (ICC) , Sydney, Australia, 2014

[27] Ristenpart, Thomas and Yilek, Scott, *When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography*, Proceedings of the Network and Distributed System Security Symposium, NDSS 2010

[28] Diogo A. B. Fernandes, Liliana F. B. Soares, Mario M. Freire and Pedro R. M. Inacio, *Randomness in Virtual Machines*

[29] Haveged project, http://www.issihosts.com/haveged/

[30] Oracle, Java 8 Security, https://docs.oracle.com/javase/8/docs/technotes /guides/security/

[31] Oracle, Java 8 Documentation, https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

[32] Cplusplus.com reference, http://www.cplusplus.com/reference /random/random_device/

[33] Donald E. Knuth. *The Art of Computer Programming, volume 2:Seminumerical Algorithms.* Addison-Wesley, 1969.

[34] Blum, Lenore; Blum, Manuel; Shub, Mike, *A Simple Unpredictable Pseudo-Random Number Generator*, SIAM Journal on Computing.

[35] M. Blum and S. Micali. *How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits*

[36] Steinfeld, R., Pieprzyk, J., Wang, H.: *On the provable security of an efficient RSA-based pseudorandom generator.* ASIACRYPT 2006. LNCS, vol. 4284, pp. 194209. Springer, Heidelberg (2006)

[37] A. Sidorenko, *Design and analysis of provably secure pseudorandom generators*, Technische Universiteit Eindhoven, 2007

[38] J. Kelsey, B. Schneier, and N. Ferguson, *Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator*, Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999

[39] Ferguson, Niels; Schneier, Bruce; Kohno, Tadayoshi (2010). *Chapter 9: Generating Randomness. Cryptography Engineering: Design Principles and Practical Applications.* Wiley Publishing

[40] The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/

[41] Dieharder: A Random Number Test Suite: http://webhome.phy.duke.edu/ rgb/General/dieharder.php

[42] http://www.fourmilab.ch/random/

[43] http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

[44] LÉcuyer, Pierre and Simard, Richard, *TestU01: A C Library for Empirical Testing of Random Number Generators*, ACM Trans. Math. Softw., 2007

[45] John Kelsey, Kerry A. McKay and Meltem Sonmez Turan, *Predictive Models for Min-Entropy Estimation*, 2015

[46] John Kelsey, Kerry A. McKay and Meltem Sonmez Turan, *How Random is Your RNG?*, 2015

[47] Gärtner, Joel, *Analysis of Entropy Usage in Random Number Generators*, 2017, Royal Institute of Technology, Stockholm, Sweden