Recommendations for Randomness in the Operating System
or, How to Keep Evil Children Out of Your Pool and Other Random Facts

Henry Corrigan-Gibbs and Suman Jana
Stanford University

Abstract

Common misconceptions about randomness underlie the design and implementation of randomness sources in popular operating systems. We debunk these fallacies with a survey of the "realities of randomness" and derive a number of new architectural principles for OS randomness subsystems.

1 Introduction

Randomness is at the heart of the security of a modern operating system: cryptographic keys, TLS nonces, ASLR offsets, password salts, TCP sequence numbers, and DNS source port numbers all rely on a source of hard-to-predict random bits. Unfortunately, misuse and abuse of random numbers and random number generators has led to a jaw-dropping number of bugs and security holes of late [9–32, 44, 55].

The blame for many such failures lies not with application developers, but with the faulty designs and error-prone interfaces for randomness common in popular OSes. The documentation of OS randomness sources is often misleading or incorrect and serves to spread myths about randomness (e.g., that a well-seeded randomness pool can "run out" of random bits), rather than dispel them. In addition, the special-file interface to the OS randomness subsystem (i.e., `/dev/random`) makes it difficult, if not impossible, for applications to safely use randomness under adversarial conditions. Given the state of randomness sources in popular OSes, it is no surprise that developers often misunderstand and misuse randomness.

In the first part of this paper, we identify and debunk a number of common misconceptions about OS randomness. Along the way, we point out weaknesses in and attacks against the randomness sources of existing OSes. In the second part, we outline a new architecture for OS randomness that solves many of the problems with existing designs. In sum, we attempt to "set the record straight" on randomness for designers of future OSes, with the hope that new systems will take a more principled view towards this important but counter-intuitive piece of the OS.

2 Preliminaries

We first define a few key terms and concepts.

Entropy. For our purposes, entropy is a measure of an adversary's uncertainty about the state of a particular value. To make this notion precise, let S be a discrete random variable representing, for example, the state of a cryptographic random number generator. Let $G(S \mid \mathcal{A}\text{'s knowledge})$ be a random variable representing the number of guesses required to recover the value of S given an adversary $\mathcal{A}$'s knowledge of the distribution of S, following an optimal strategy [48]. We say that the distribution of the state S has k bits of *guessing entropy* ("entropy") with respect to an adversary $\mathcal{A}$ if the expected number of guesses $E[\,G(S \mid \mathcal{A}\text{'s knowledge})\,]$ is greater than or equal to 2^k [7, Section 3.2.4]. A value S sampled uniformly at random from a set of 2^k values has just over $k-1$ bits of guessing entropy, since $E[G(S)]$ is equal to $(2^k+1)/2$.[1] When we say that a particular value "has k bits of entropy," we mean that the value is sampled from a distribution with k bits of entropy from the perspective of some adversary.

Pseudo-random Bit Generator. A *pseudo-random bit generator* (PRG) is a family of deterministic algorithms mapping short bitstrings ("seeds") to long bitstrings ("outputs") [6, 49]. To be useful in a cryptographic setting, the output of a PRG must be indistinguishable from random, as long as the seed is sampled from a distribution over the seed space with "enough" entropy.[2]

One suitable PRG is the AES block cipher instantiated in counter mode [3]. The seed for the PRG is the AES key k and the output of the PRG is the concatenation of the AES encryptions of the bitstrings representing "0," "1," "2," and so on. As long as the seed for this PRG is sampled from a distribution with many bits of entropy (e.g., 128 bits) it appears infeasible to distinguish the output of this PRG from random.

OS Randomness Subsystem. All modern operating systems implement some sort of *randomness pool*. The pool contains a bitstring derived from other values in the system that are ostensibly difficult for user-level processes to guess. These values typically include the CPU's cycle counter, disk seek times, outputs from a hardware randomness source (e.g., Intel's `RdRand` instruction), network packet arrival times, and other fast-changing values

[1] The guessing entropy of this distribution is *not* equal to its Shannon entropy or min entropy [7, Section 3.4].

[2] To be precise, the seed must be sampled from a distribution with k bits of entropy such that k has size polynomial in the security parameter.

known to the operating system. The operating system periodically updates this pool with fresh values.

In Unix-like operating systems, user processes interact with the randomness subsystem through a special file named `/dev/random`.[3] The `CryptGenRandom` function serves a similar purpose in recent versions of Windows. When we say that a process reads random bytes from the OS, we mean that it requests bytes from `/dev/random` or `CryptGenRandom`. Applications and kernel threads use randomness to generate cryptographic secrets, run randomized algorithms, and derive unpredictable values for other purposes (e.g., DNS source port numbers).

All common flavors of Unix also allow user-space processes to *write* random bytes to the OS by writing to `/dev/random`, which mixes the user-provided bits into the OS randomness pool.

3 Realities of Randomness

We review a number of realities of randomness, which inform the randomness architecture introduced in the next section.

3.1 Your Randomness Won't Run Out

A common misconception is that randomness in the entropy pool can somehow be "used up"—that the OS must constantly add new hard-to-predict bits to the pool for its output to remain unpredictable. For example, a post on the CloudFlare Security Blog about the Linux randomness pool states:

> When random numbers are generated from the pool the entropy of the pool is diminished (because the person receiving the random number has some information about the pool itself). [53]

This statement is *false* as long as widespread cryptographic assumptions hold. Once there is enough randomness in the pool to seed a pseudo-random generator, the OS can produce an endless string of random-looking bits.[4]

Deriving an endless stream of bits from a random-enough randomness pool is straightforward: use a cryptographic hash function (e.g., SHA-256) to hash the pool's contents into the space of PRG seeds (e.g., AES keys). Then, use the PRG (e.g., AES in counter mode) to expand the short seed into a long bitstring. Application of the random-oracle model [4] allows a rigorous analysis of the security of this extract-then-expand technique [45]. This procedure leads to the following finding:

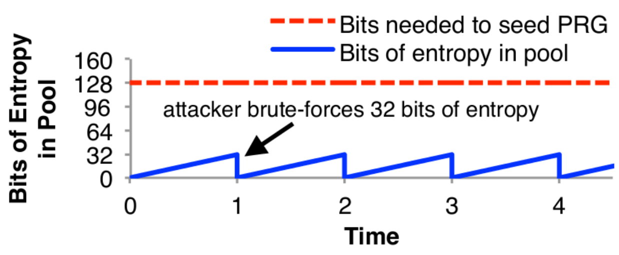


Figure 1: Under unfavorable conditions, the system will never accumulate enough entropy to generate strong cryptographic keys (using the 128-bit security level).

Reality 1. *Once the randomness pool has accumulated enough entropy to seed a PRG, the pool can never "run out" of random bits (if the pool implementation is sane).*

As we note, Reality 1 is only true as long as the randomness pool uses a "sane" implementation. The extract-than-expand technique outlined above is an example of a "sane" implementation [45]. An example of an "insane" implementation is any one that returns bits of the randomness pool state to a user-space processes without running it through a hash function and cryptographic PRG. Implementations like these may reveal the entire state of the randomness pool to an adversarial user-space process and can thus "run out" of entropy.

Corollary. *The only time that the randomness pool is vulnerable to compromise is in the period* before *it has accumulated enough entropy to seed a PRG.*

Before the randomness pool has enough entropy to seed a PRG, an adversary who can read random bytes from OS can learn the internal state of the OS's randomness pool. To mount this attack, the adversary first reads a random bitstring b from the OS, and guesses the state of the randomness pool that would have produced the string b. The average number of guesses required is bounded above by 2^k, where k is the number of bits of entropy in the randomness pool.

Figure 1 graphically depicts this process: the pool starts out with zero bits of entropy (e.g., after the machine's first boot) and the OS harvests 32 bits of entropy from hardware sources in every time unit. If a malicious process can read several bytes from `/dev/random` at every time unit and brute-force through the 2^{32} possible pool states, the malicious process can always recover the state of the randomness pool and the system will *never* accumulate enough entropy to seed a PRG.

3.2 Entropy Estimation is Hopeless

A tempting way to address the problem depicted in Figure 1 is to just disallow reads from the OS randomness source until the pool accumulates enough entropy to seed a PRG (e.g., 128 bits). Several operating systems—including Linux, NetBSD, and Solaris—try to *estimate*

[3]In Linux and Solaris, `/dev/random` can block, so there is also a similar, but non-blocking, file called `/dev/urandom`.

[4]Well, "endless" at least from the perspective of all polynomial-time adversaries.

how many bits of entropy are in the randomness pool and will block `/dev/random` until there is enough entropy to prevent the attack of Figure 1. Unfortunately, this strategy is misguided.

Reality 2. *Building an accurate entropy estimator is infeasible.*

To see why entropy estimation is infeasible, recall the definition of entropy: it is a measure of the *adversary*'s uncertainty about the value of a certain variable. The OS generally has no way of knowing what the adversary knows about the system and thus has no hope of estimating how many bits of entropy are in the randomness pool [1]. (Of course, in the degenerate case in which no string has *ever* been added to the pool, it is clear that there are zero bits of entropy in the pool.)

For example, OS designers might reason that the low-order bit of the arrival time of every network packet is a good source of randomness. An entropy estimator might then increase the entropy count by one bit upon receipt of every packet. If an adversary can monitor the machine's network connection, however, the packet arrival times would be a poor source of randomness with respect to this adversary. Real-world entropy estimators have similar weaknesses [33, Lemma 3].

Since the OS cannot *ever* accurately estimate how many bits are in the pool, and since the OS should only block reads to the randomness source when there are "too few" bits of entropy in the randomness pool, we conclude:

Corollary. *Reads to the OS randomness source should never block.*

After booting, the OS should initialize the randomness subsystem with values from I/O sources as best it can and then make `/dev/random` available to user-space processes. If the system's design is sound, the OS's randomness pools will eventually accumulate entropy, even if they were not well-seeded initially. FreeBSD and Mac OS use this strategy to avoid entropy estimation.

3.3 All Bits Should Be Treated Equally

The widespread use of entropy estimation techniques has given rise to the misconception that the OS should differentiate between "trusted" and "untrusted" inputs to the randomness pool. In Linux, for example, only kernel threads and administrators can write to the randomness pools in a way that increases the entropy estimate.

The intuition behind this design is clear: internal entropy sources, like the cycle counter and disk seek times, are considered "more random" than user-provided bitstrings, which might be adversarially crafted. However, there is no need to make such a distinction, since adding data to the randomness pool should *never* decrease the amount of entropy in the pool:

Reality 3. *Adding bits to the randomness pool will* never decrease *the amount of entropy in the pool, as long as the implementation is sane.*

For an example of a "sane" implementation: let the state of the randomness pool be an ℓ-bit string and let $E_k(m)$ be an ideal cipher that encrypts a message m with a key k, such that m and k are both ℓ-bit strings [5, 52]. We can calculate the new state s_{i+1} of the pool by hashing the (possibly long) input string into a short cipher key k and computing $s_{i+1} \leftarrow E_k(s_i)$ [45]. Since E defines a family of permutations (indexed by k), the adversary's uncertainty about the value of s_{i+1} is at least as large as the adversary's uncertainty about the value of s_i, no matter whether the input string is adversarially chosen.

Corollary. *The OS should allow any user and any process to contribute to the randomness pool.*

Adding bits to the randomness pool can only increase the adversary's uncertainty about the pool contents, so it can never hurt to allow writes into the pool.

3.4 User-Space is a Danger Zone

Cryptographic folklore, as well as the OpenBSD and Linux randomness manpages, suggest that it is good practice to maintain a user-space randomness pool when an application needs many random bytes. Many popular cryptography libraries (including OpenSSL) follow this advice and implement their own user-space randomness pools and PRGs.

Reality 4. *User-space randomness pools are often unsafe.*

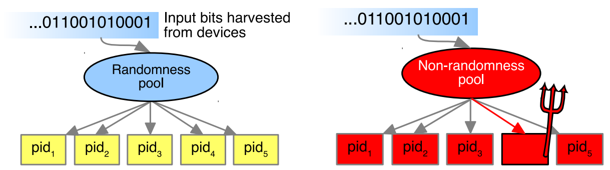
Maintaining a randomness pool in user-space entails a number of risks:

Fork (un)safety. The implementation must be sure to reseed the child of a `fork()` call with fresh randomness to make sure that the child's pool differs from its parent's. Neglecting to reseed after forking is easy to do and can lead to dangerous security vulnerabilities [26, 30, 31].

Pool leakage. It is more difficult to maintain the privacy of the randomness pool state in user-space than it is in kernel-space. Chow et al. [8] showed that user-space secrets can leak through swap and other unexpected sources.

Reseeding required. If the user-space randomness pool seeds itself from `/dev/random` soon after boot, the OS's pool may not yet have enough entropy to provide a strong seed. Even if the amount of entropy in the kernel pool eventually increases, the amount of entropy in the user-space pool will not increase unless the user-space implementation periodically reseeds itself from `/dev/random`. Many libraries, including OpenSSL, do not reseed themselves automatically.

There are legitimate reasons for maintaining a user-space randomness pool: it avoids the system-call over-



(a) Today's OSes use a single entropy pool shared amongst all processes.

(b) A single malicious process can prevent entropy accumulation indefinitely.

Figure 2: Today's OS randomness subsystems.

head of reading a special file and it allows an application to use a different PRG than the one the kernel supports. Even so, the risks of maintaining a user-space pool are much greater than the benefits in many cases.

3.5 Use a System Call, Not a Special File

Most OSes limit the number of open file descriptors, so if either the per-user or global file descriptor limit is reached, no process can access system randomness via `/dev/random`. In this case, applications have to choose between two bad alternatives: (a) fail and halt, or (b) proceed without randomness from the OS.

An attack. Most applications we have inspected choose option (b) above, which opens them up to a file descriptor denial-of-service attack: a coalition of malicious users opens enough files to exceed the system's global file descriptor limit. When an honest user subsequently tries to generate a cryptographic secret key, the cryptography library may produce a non-random (and thus vulnerable) key [27, 28]. Many components of OpenSSL, including SSL/TLS pre-master secret generation and RSA key generation, are vulnerable to this attack, as is the `arc4random` function used for cryptographic randomness in FreeBSD, OpenBSD, and Mac OS.

Reality 5. *Special-file interfaces for randomness are often unsafe.*

The special-file interface for randomness allows malicious processes to starve honest processes of randomness under certain circumstances. To prevent these attacks, OSes should provide a system-call interface to the OS randomness subsystem and security-critical applications running on OSes without a system-call interface should "fail closed" when they cannot read `/dev/random`. The latest versions of Windows, Linux, and OpenBSD offer a system call for randomness, but FreeBSD, Mac OS X, and Solaris do not.

4 Improving OS Randomness

Our proposed architecture for OS randomness uses *per-process* randomness pools to ensure entropy accumulation, even under adversarial conditions. The design does not require entropy estimation and eliminates the distinction between "trusted" and "untrusted" inputs to the pools.

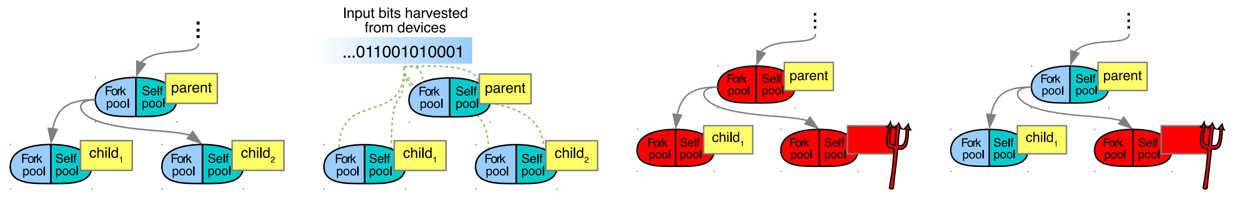
The corollary to Reality 1 indicates that the system's randomness pool is vulnerable *only* in the time before the pool has accumulated enough entropy to seed a PRG. The focus of the randomness subsystem, then, is to ensure that eventually there is enough entropy in the pool to seed a PRG, even in the presence of many adversarial processes.

To our knowledge, *no current OS* provides this property. Existing OSes use a single common entropy pool shared amongst all processes, as shown in Figure 2. If the system starts out in a known state (e.g., after first boot), the hardware supplies only a few bits of entropy per time unit, and an adversarial process can read a few bits of OS-supplied randomness in every time unit, then the adversary will *always* know the contents of the entropy pool via a brute-force guessing attack (Figure 1) [32–34]. In these conditions, benign processes will not ever be able to generate strong cryptographic secrets.

To prevent this failure case, our design allocates separate entropy pools to each process in the system. By limiting the number of pools from which an adversarial process can read, we minimize the number of pools that an adversary can attack. This ensures that benign processes eventually accumulate enough entropy to generate strong cryptographic keys (though, by Reality 2 we will never know precisely *when* this has happened).

In our architecture (Figure 3), the OS maintains two randomness pools for each process in the system: one for generating random values for the process itself and another for initializing the pools of forked child processes. When a process asks the OS for random bits, the OS derives these bits from the first pool, the "self" pool. When a process forks a child process, the OS initializes the child's two randomness pools using bits derived from the parent's "fork" pool. The OS feeds new input bits into all active pools in the system on a round-robin schedule. Under the minimal assumption that the random input bits are not perfectly correlated with each other, all non-adversarial pools in the system will eventually accumulate enough entropy to seed a PRG.

The use of two separate pools per process provides protection in scenarios in which the pool of a parent process contains few bits of entropy and an adversary can repeatedly read from the randomness pools of the parent's recently forked child processes. For example, if the parent is an (honest) forking TLS server, an adversarial client can learn the state of the parent's randomness pool by connecting to a forked child server process and inspecting the nonces in the child process' TLS messages. If the parent's pool started out with few bits of entropy, the adversary will be able to learn the state of the parent's pool and will keep the parent's pool from ever accumulating entropy.



(a) Children of `fork()`s have pools seeded from the parent's fork pool.

(b) The OS replenishes *all* pools in the system in a round-robin fashion.

(c) An evil child might initially be able to learn the state of other pools.

(d) However, the parent's other children can eventually accumulate entropy (recover).

Figure 3: A system using our pool-per-process design can accumulate entropy in the presence of a malicious process.

By having a separate forking pool, we ensure that the child processes cannot read from the parent's "self" pool and thus the child cannot recover the state of the parent's "self" pool. This allows the parent's "self" pool to eventually accumulate entropy with respect to the adversarial client, even though the parent's "fork" pool never will.

Instead of having a separate "fork" pool, we could just initialize the child's randomness pools to all zeros and let the child's pool accumulate entropy over time from input sources. This addresses the forking attack above, but introduces a more serious problem: a child will not have *any* access to randomness immediately after forking, even if the parent's pool has many bits of entropy. In the case of a forking TLS server, in which a child needs to generate cryptographic secrets immediately after forking, zero-initializing the child's pool is unsatisfactory.

The downside of having separate randomness pools is that the system as a whole will accumulate entropy more slowly than today's systems do—$2P$ times more slowly, where P is the number of active processes.

To increase the rate of accumulation, the system could have a single global randomness pool *in addition* to the per-process pools described above. The OS would insert half of the input bits into the global pool and the other half into the hierarchy of per-process pools. When a process requests random bits, the OS would derive these bits from both the process' "self" pool *and* from the global pool. In the absence of a malicious process, the system would accumulate entropy almost as quickly as it would with today's designs. In the presence of many malicious processes, the honest processes' pools would be still able to accumulate entropy eventually (albeit at a slower rate). There are many possible combinations of global, local, and intermediate pools, but we leave a full investigation of these ideas to future work.

5 Related Work

There is a long and fruitful line of work investigating randomness failures in Debian [19, 55], Java [40], Linux [39, 41], Netscape [38], Windows [35]. Weak hardware entropy sources [42, 47] and use of virtual machine snapshotting [36, 37, 51] have also led to randomness failures in the past. Lazar et al. investigate the source of bugs in cryptographic software, including cases involving misuse of randomness [46].

Another line of work has proposed techniques to protect against weak randomness. Barak and Halevi [1] and Dodis et al. [33, 34] rigorously analyze random number generators as used in operating systems and offer improved constructions. Mowery et al. conjecture that even embedded devices have potentially rich sources of entropy at boot time [50]. Hedged public-key cryptography, developed by Bellare et al. [2, 51], allow for a graceful degradation of security in the face of bad randomness.

Many of the arguments of Section 3 are in the folklore and some have been discussed in prior work [1, 43, 54]. To our knowledge, no one has proposed using per-process randomness pools as we do in Section 4.

6 Conclusion

We have debunked a number of common misconceptions about randomness in the OS. From the counter-intuitive realities of randomness, we derive a number of straightforward design principles for the OS's randomness subsystem. We also make the unorthodox recommendation that OSes maintain two separate randomness pools for every process. We hope that our paper will encourage OS designers to question common myths about randomness and to rethink how they provide randomness to users.

Acknowledgements

Dan Boneh encouraged us to gather our random thoughts into this paper and offered comments on an early draft. Bryan Ford, David Terei, Tal Garfinkel, and the anonymous reviewers gave useful tips on how to improve the presentation. Bryan also suggested the idea, discussed in Section 4, of using a combination of global and per-process randomness pools. An NDSEG Fellowship and the National Science Foundation funded this work in part.

References

[1] Boaz Barak and Shai Halevi. A model and architecture for pseudo-random generation with applications to `/dev/random`. In *CCS*, pages 203–212. ACM, 2005.

[2] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In *ASIACRYPT*, pages 232–249. Springer, 2009.

[3] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *Foundations of Computer Science*, pages 394–403. IEEE, 1997.

[4] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, pages 62–73. ACM, 1993.

[5] John Black. The ideal-cipher model, revisited: An uninstantiable blockcipher-based hash function. In *Fast Software Encryption*, pages 328–340, 2006.

[6] Lenore Blum, Manuel Blum, and Mike Shub. A simple pnpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, 1986.

[7] Christian Cachin. *Entropy Measures and Unconditional Security in Cryptography*. PhD thesis, ETH Zürich, 1997.

[8] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *USENIX Security*, pages 22–22, 2005.

[9] CVE-2000-0357: ORBit and esound in Red Hat Linux do not use sufficiently random numbers, December 1999.

[10] CVE-2001-0950: ValiCert Enterprise Validation Authority uses insufficiently random data, January 2001.

[11] CVE-2001-1141: PRNG in SSLeay and OpenSSL could be used by attackers to predict future pseudo-random numbers, July 2001.

[12] CVE-2001-1467: `mkpasswd`, as used by Red Hat Linux, seeds its random number generator with its process ID, April 2001.

[13] CVE-2003-1376: WinZip uses weak random number generation for password protected ZIP files, December 2003.

[14] CVE-2005-3087: SecureW2 TLS implementation uses weak random number generators during generation of the pre-master secret, September 2005.

[15] CVE-2006-1378: PasswordSafe uses a weak random number generator, March 2006.

[16] CVE-2006-1833: Intel RNG Driver in NetBSD may always generate the same random number, April 2006.

[17] CVE-2007-2453: Random number feature in Linux kernel does not properly seed pools when there is no entropy, June 2007.

[18] CVE-2008-0141: WebPortal CMS generates predictable passwords containing only the time of day, January 2008.

[19] CVE-2008-0166: OpenSSL on Debian-based operating systems uses a random number generator that generates predictable numbers, January 2008.

[20] CVE-2008-2108: `GENERATE_SEED` macro in php produces 24 bits of entropy and simplifies brute force attacks against the rand and mt_rand functions, May 2008.

[21] CVE-2008-5162: The `arc4random` function in FreeBSD does not have a proper entropy source for a short time period immediately after boot, November 2008.

[22] CVE-2009-0255: TYPO3 creates the encryption key with an insufficiently random seed, January 2009.

[23] CVE-2009-3238: Linux kernel produces insufficiently random numbers, September 2009.

[24] CVE-2009-3278: QNAP uses rand library function to generate a certain recovery key, September 2009.

[25] CVE-2011-3599: `Crypt::DSA` for Perl, when `/dev/random` is absent, uses the `data::random` module, October 2011.

[26] CVE-2013-1445: The `crypto.random.atfork` function in PyCrypto before 2.6.1 does not properly reseed the pseudo-random number generator (PRNG) before allowing a child process to access it, October 2013.

[27] CVE-2013-4442: Password generator (aka Pwgen) before 2.07 uses weak pseudo generated numbers when `/dev/urandom` is unavailable, December 2013.

[28] CVE-2013-5180: The `srandomdev` function in Libc in Apple Mac OS X before 10.9, when the kernel random-number generator is unavailable, produces predictable values instead of the intended random values, October 2013.

[29] CVE-2013-7373: Android before 4.4 does not properly arrange for seeding of the OpenSSL PRNG, April 2013.

[30] CVE-2014-0016: tunnel before 5.00, when using fork threading, does not properly update the state of the OpenSSL pseudo-random number generator, March 2014.

[31] CVE-2014-0017: The `rand_bytes` function in libssh before 0.6.3, when forking is enabled, does not properly reset the state of the OpenSSL pseudo-random number generator, March 2014.

[32] CVE-2014-4422: The kernel in Apple iOS before 8 and Apple TV before 7 uses a predictable random number generator during the early portion of the boot process, October 2014.

[33] Yevgeniy Dodis, David Pointcheval, Sylvain Ruhault, Damien Vergniaud, and Daniel Wichs. Security analysis of pseudo-random number generators with input: `/dev/random` is not robust. In *CCS*, pages 647–658. ACM, 2013.

[34] Yevgeniy Dodis, Adi Shamir, Noah Stephens-Davidowitz, and Daniel Wichs. How to eat your entropy and have it too-optimal recovery strategies for compromised rngs. In *CRYPTO*, volume 2014.

[35] Leo Dorrendorf, Zvi Gutterman, and Benny Pinkas. Cryptanalysis of the Windows random number generator. In *CCS*, pages 476–485, 2007.

[36] Adam Everspaugh, Yan Zhai, Robert Jellinek, Thomas Ristenpart, and Michael Swift. Not-so-random numbers in virtualized Linux and the Whirlwind RNG. In *IEEE Security and Privacy*, 2014.

[37] Tal Garfinkel and Mendel Rosenblum. When virtual is harder than real: Security challenges in virtual machine based computing environments. In *HotOS*, 2005.

[38] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal–Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[39] Peter Gutmann. Random number generation. *Cryptographic Security Architecture: Design and Verification*, pages 215–273, 2004.

[40] Zvi Gutterman and Dahlia Malkhi. Hold your sessions: An attack on Java `session-id` generation. In *CT-RSA*, pages 44–57, 2005.

[41] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux random number generator. In *IEEE Security and Privacy*, pages 371–385, 2006.

[42] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, pages 205–220, 2012.

[43] Thomas Hühn. Myths about `/dev/urandom`. `http://www.2uo.de/myths-about-urandom/`, November 2014.

[44] Alex Klyubin. Some SecureRandom thoughts. `http://android-developers.blogspot.com/2013/08/some-securerandom-thoughts.html`, August 2013.

[45] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *CRYPTO*, pages 631–648. Springer, 2010.

[46] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *APSys*, page 7. ACM, 2014.

[47] Arjen K Lenstra, James P Hughes, Maxime Augier, Joppe W Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, Whit is right. *IACR ePrint archive*, 64, 2012.

[48] James L. Massey. Guessing and entropy. In *International Symposium on Information Theory*, page 204. IEEE, 1994.

[49] Alfred J. Menezes, Paul C. Van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC press, 2010.

[50] Keaton Mowery, Michael Wei, David Kohlbrenner, Hovav Shacham, and Steven Swanson. Welcome to the Entropics: Boot-time entropy in embedded devices. In *IEEE Security and Privacy*, 2013.

[51] Thomas Ristenpart and Scott Yilek. When good randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *NDSS*, 2010.

[52] Claude E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[53] Nick Sullivan. Ensuring randomness with Linux's random number generator. `http://blog.cloudflare.com/ensuring-randomness-with-linuxs-random-number-generator/`, October 2014.

[54] John Viega. Practical random number generation in software. In *ACSAC*. IEEE, 2003.

[55] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *IMC*, pages 15–27, November 2009.