Chapter 11

Random number generation

Random bits form an essential ingredient of nearly every cryptographic protocol. They are required to produce cryptographic keys, initialization vectors for symmetric encryption, padding for public-key encryption schemes, nonce values, session identifiers, and many other undpredictable values. Modern cryptographic protocols are thirsty for random inputs. A key challenge in designing cryptographic systems is the need to produce these bits in large quantities while also ensuring that they are of very high quality.

To give just one illustration of the challenge, consider the establishment of a single TLS connection. For a connection that uses elliptic curve Diffie-Hellman and ECDSA signatures, the server's side of the protocol requires more than 800 random bits just to establish a single session. Whichever randomness source is used, it will be drawn upon to generate secret values such as ephemeral secret keys, as well as non-secret values such as nonces that will be sent over the wire in the clear. There is essentially no room for error in generating these bits. If an adversary can predict or guess any of these values, or sometimes even detect a small bias in their distribution, attackers might be able to decrypt communications, replay connections, or even extract long-term signing keys.

DRAFT

What is randomness? In one mathematical view of randomness, random numbers (or variables) simply exist: they can be *sampled* from some distribution, such that the probability of any given outcome is well-defined but the actual value is not known in advance. In this view, we can invoke concepts such as unbiased "perfect" coins without worrying very much about where we will find such ideal objects. (In fact, many theoretical cryptography papers refer to random bits as "coins.") Even mathematicians have found different ways to think about randomness. Kolmogorov [105], for example, defined randomness in terms of *compressibility*: a string variable can be considered "random" if it is incompressible by any means, *i.e.,* the original string is one of the shortest ways to represent its data.[1]

[1]This is a beautiful definition, but it's important to note that it's not a very practical one. Compression in this setting does not refer to some specific data compression algorithm such as LZW. Instead it refers to *any possible* strategy for representing a string that could be devised before the string itself is generated, including the uncomputable task of enumerating every possible Turing machine that could produce it.

There is also a physical view of randomness. In this view, there exist processes in the universe that are highly unpredictable. These might include the peaks in a waveform captured by a microphone aimed at a noisy city street, thermal noise in an electrical circuit, or measurements of the decay of a radioactive particle. It is likely that some of these physical processes are not *truly* unpredictable, meaning that with a precise measurement of the initial conditions and an impractical amount of computing power, some attacker could predict future outcomes. Some physical processes appear to be unpredictable in more fundamental ways. For example, standard interpretations of quantum physics hold that it is not possible to predict the outcome of certain quantum measurements in advance, even if their probability distributions can be calculated with great accuracy.

The good news is that the mathematical and physical views can coexist. Many cryptographic protocol designers will happily assume the availability of perfect random bits as an input, and base the security of the system on the assumption that these bits are truly unpredictable. The problem of manufacting these bits is left to engineers and hardware designers, who use a wide variety of techniques to measure physical processes and then *extract* random bits from the messy data. As inelegant as this approach may seem, it seems to work most of the time, unless something goes wrong – at which point things can get exciting!

DRAFT

Random vs. pseudorandom. While this chapter is about random number generation, it's worth pointing out that much of the "randomness" we use in cryptographic systems isn't random at all. It is instead generated by first producing modest amounts of (hopefully) random bits from some physical process, and then "stretching" this into a large number of bits using a deterministic cryptographic algorithm known as a (cryptographically-secure) *pseudorandom (bit) generator*, or PRG.[2] These PRGs are clearly not random at all! In the famous words of John von Neumann, "anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

Despite von Neumann's warning, there is logic behind the widespread use of PRG algorithms. Many high-quality hardware randomness generators operate slowly, meaning that their output may be insufficient to supply hungry cryptographic algorithms. Moreover, modern PRG algorithms are designed to offer a strong *computational* guarantee: namely, if the input "seed" is random, then the output of a secure PRG should be *indistinguishable* from truly random bits, at least to a computationally-bounded adversary who does not know the seed. As a final matter, the use of PRGs is often mandated even in situations where the seed is generated by a relatively fast hardware generator: this is because the "post-processing" of a PRG can sometimes provide an extra layer of security in cases where a hardware generator malfunctions (for example by producing slightly predictable or biased outputs.) A common architecture of this form is depicted in Figure 11.1.

[2]Sometimes this is also called a pseudorandom *number* generator (PRNG) or a *cryptographically-secure* PRNG (CSPRNG), or even a *deterministic random bit generator* (DRBG) in U.S. government parlance [22]. There are also non-cryptographic generators that are sometimes referred to as "pseudorandom", but security professionals should never, ever use this terminology.

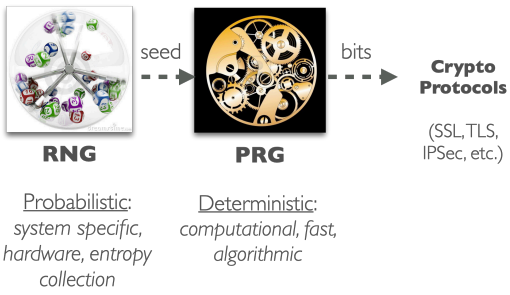


Figure 11.1: Overview of the random number generation architecture used in nearly every deployed system. Here a "true" RNG generates unpredictable bits. These are conditioned, then used to seed a pseudorandom generator. Cryptographic protocols then consume the resulting bits.

11.1 "True" Random Number Generation

Non-determinism was the enemy of computing's early pioneers. These founders worked doggedly to remove unpredictable behavior from the earliest computing systems, and their achievement is reflected in the fact that modern processors nearly always produce the same results from a given input. The downside of this accomplishment is that determinism greatly limits a processor's access to the sorts of unpredictable bits required for cryptography.

Historically, system designers have worked around this limitation in a variety of *ad hoc* ways. The most common approaches involve measuring unpredictable hardware or software processes that occur on the machine. These can include the precise timing of interrupts, human input events such as keystrokes and mouse movements, or high-precision measurements of the seek times of a rotating hard drive. While these measurements are not entirely unpredictable, they are believed to contain a significant amount of entropy that can be distilled into effectively random bits. Today this "entropy gathering" approach remains the most practical solution for low-cost processors used in inexpensive embedded devices.

Unfortunately, the march of technological progress has reduced the availability of high-quality entropy sources (in particular, rotating hard drives), and not every device has access to human input. To address this problem, mid-tier and high-end processors/SoCs have begun to include built-in hardware random number generators. The most common design in these systems amplifies low-level thermal noise in a circuit, then uses a simple arrangement of amplifiers and oscillators to convert this noise into unpredictable binary data. The resulting entropy is typically used in as one input to a modern system RNG. For more expensive systems with security-critical applications, it is also possible to purchase high-quality external RNG devices that use even more advanced technology, including some that advertise the use of "quantum" randomness. Paranoid developers will often combine as many unpredictable inputs as possible. Although it is more "show business" than security, the company Cloudflare famously uses a wall of lava lamps as one input to their network's RNG.

DRAFT

Figure 11.2: The Cloudflare lava lamp wall. A camera measures entropy from the image of these lamps and folds this into other RNG inputs used by the company. (Photo credit: Martin J. Levy)

11.1.1 Entropy gathering

On systems that lack internal RNG hardware, a typical approach to generating randomness is to collect unpredictable measurements from various sources. This is referred to as "entropy gathering", reflecting the fact that we are interested only in the unpredictable component, the *entropy*, of these measurements.

The process of producing random bits from gathered entropy consists of three sub-stages: (1) entropy collection, in which raw measurements are collected from various sources and aggregated together into a "pool", (2) entropy estimation, in which the total number of entropy bits are estimated using various heuristics, (3) pool "stirring" and randomness extraction, where the measurements are combined together and processed in various ways to distill them into uniformly-random bits.

DRAFT

Sources of entropy. Modern OS random number generators collect entropy from a variety of sources. These include:

- *Interrupt and scheduler timings.* Modern processors typically provide a cycle-accurate processor counter that runs at the CPU clock frequency. This can be used to compute high-resolution timestamps for a number of somewhat-unpredictable events that occur on the machine, such as interrupt timings and the timing of events from the OS system scheduler.

- *Human input.* On systems with input devices such as a keyboard and mouse, RNGs can collect readings from these movements. Measurements typically include both the identity of the event (such as the key pressed, or the mouse direction), combined with the timestamp of the measurement.

- *Disk events.* A common source of entropy comes from *seek timings* on spinning hard drives, due to the fact that chaotic turbulence in the air or fluid under the drive head can make precise timings difficult to predict. To capture this information, OS RNGs

often collect a high-resolution timestamp each time a disk event occurs. This is less useful on systems with non-spinning drives such as SSDs, which may not produce much entropy at all.

- *Hardware RNG inputs.* Systems with high-quality hardware RNGs will typically draw input from these devices whenever it is available. This can include random bits drawn from the processor RNG, or from the RNG located on a system device such as a Trusted Platform Module (TPM).

- *Seed files.* Most operating systems require random numbers at boot time. Unfortunately this is often the time when systems may be in the *most* predictable state, since they have had little time to gather fresh entropy. To address this, some operating system RNGs will write a *seed file* that contains output from a seeded RNG so that the OS can use this data to seed the RNG at the next boot.

- *Virtual Machine seeds.* Some VM hypervisors provide random seeds to guest operating systems, for example via the Advanced Configuration and Power Interface (ACPI). This is useful to ensure that VMs do not use predictable random seeds when two copies of the same snapshot are restarted.

- *Public randomness sources.* While it is not standard any major OS, I have seen examples in which users manually inject entropy from public sources such as "randomness beacons" [23]. While this is usually not harmful in a well-designed RNG architecture, an attacker may also have access to these bits: hence they may not contribute any meaningful entropy to the system.

Measurements from these sources are typically collected and appended to a buffer known as an *entropy pool.* This pool may be *stirred* periodically, a process that typically involves hashing the pool's contents and replacing the pool data with the resulting hash output.

DRAFT

Entropy estimation. A good threat model for an entropy gathering system is to assume that the initial state of the system is entirely known to the adversary, *i.e.,* that there is no entropy in the pool. As the system gathers measurements, the number of potential states for the system increases until it is no longer feasible for the adversary to enumerate every possible state and predict the outputs of the system. Typically at this point the pool is hashed, and the resulting hash is used to seed a pseudorandom generator (PRG) that generates the actual random bits used by the system.

The challenge in filling an entropy pool is that even "unpredictable" events such as drive seek times are, in fact, largely predictable. For example: a typical processor counter measurement produces 48 bits of raw data; however, the majority of these bits will be predictable to an attacker with some information about the system's boot time. In practice the truly unpredictable component of a timing measurement, that is, the pure "entropy" in each measurement, may only comprise only a fraction of a single bit. The goal of a well-functioning entropy gathering system is to estimate how many measurements will be

required in order to collect sufficient entropy, meaning that the *estimated* min-entropy of all of the collected material is on the order of 128 bits or greater. At this point it should be safe to use the pool to construct seed material.

The problem with entropy estimation is that it is very hard to accurately estimate the entropy in any measurement made across different hardware platforms. Most estimates are thus based on some *heuristic*, which is a fancy word for "guess." For example, the designers of the Linux RNG system conservatively estimate that interrupt timings should include approximately 1 bit of entropy for every 64 interrupt timings measured, with the exception of interrupts generated by human input devices, which count as having substantially greater entropy. Naturally, most RNG designers are security-conscious and try to aim their estimates towards the conservative end of the range.

Should entropy estimates decrease*?* In some older RNG designs, particularly earlier implementations of Linux's `/dev/random` device, entropy was viewed as a consumable resource. In this design, the pool's entropy estimate would increase as the system collected unpredictable material. It would then *decrease* as applications read pseudorandom bits from the device. The device would block whenever the available entropy estimate dropped below some threshold.

This behavior never really made any sense. While it's very reasonable to block an application from reading bits from a generator that has not been properly seeded (*e.g.,* at early phases of system boot), once a generator is properly seeded it should stay that way. This is because most OS RNGs use the collected entropy to seed a pseudorandom generator, and the consumption of the resulting pseudorandom bits does not "un-seed" the generator in any meaningful sense. A well-designed PRG should be able to produce a genuinely huge number of output bits from a relatively short seed without any noticeable loss in security: hence, monotonically-decreasing entropy estimates are not really useful.[3]

The unfortunate legacy of the blocking behavior in `/dev/random` was that developers were, for many years, advised to use an alternative *non-blocking* device called `/dev/urandom`. While this bypassed the imaginary problem of blocking due to overconsumption, the `urandom` device would not block at early boot phases *when there was no entropy in the pool!* The result of this mess was that applications running near system boot time could sometimes generate keys using highly-predictable RNG output [87]. The mainline Linux RNG architecture has mostly discarded these anachronisms and offers a much safer interface called `getrandom()` (see §**??** for details.) Unfortunately, the earlier design lives on in some embedded Linux distributions, and its legacy of confusion and bad practices lingers to this day.

Stirring, whitening and randomness extraction. An important challenge with entropy gathering is that the resulting pool data cannot be directly used to seed a pseudorandom generator, since most generators require high-quality (uniformly-random) seed material. Unfortunately, raw entropy measurements typically produce large quantities of

[3]While many standard PRG algorithms do restrict the number of bits that can be requested from a given seed, these limits are usually quite high, on the order of 2^{64} bits or higher. Where the limit is smaller, this is almost always due to limitations in a specific PRG's design.

DRAFT

semi-predictable and biased data. Before we can use this entropy data for cryptographic purposes, the measurements in the pool must be processed to obtain a smaller but high-quality seed.

Randomness extractors. From a theory perspective, the tool required for this purpose is known as a *randomness extractor* [168]. These constructions realize an efficient algorithm that processes a (potentially long) string containing a substantial amount of entropy, then outputs a shorter and uniformly-distributed string of bits.[4] A standard requirement of this approach is that the *min-entropy* of the input data (measured in bits) must be at least as large as the total length of the output required.

Randomness extraction is a place where the theoretical cryptography literature diverges from real-world practice. In the literature, extractors are typically built from ingredients such as error correcting codes and universal hash functions [**?**]. In the real world, developers extract randomness by feeding raw entropy data into a cryptographic hash function or cipher-based construction such as SHA-2, SHA-3, BLAKE2 or AES-CBC-MAC, usually under some stronger assumption about the properties of the underlying hash functions or ciphers (*e.g.,* that they behave like random functions, see §**??**.) While this solution is theoretically unsatisfying, it's "good enough for government work" as evidenced by NIST recommendations [138].

Pool stirring. Entropy pools are normally constructed by appending new measurements to a pool buffer, and then periodically applying an extractor (hash function or MAC) to the pool data in order to extract a random seed. Even when new seed material is not immediately required, it is often advisable to limit the growth of the pool by hashing it and replacing the existing pool data with the hash digest. New measurements can then be appended to this digest and the stirring process can be repeated on the new contents. Alternatively, the pool can be stirred each time a new measurement arrives, eliminating the need to store large amounts of raw measurement data.

From a theoretical perspective, it is worth pointing out that each "stir" of the pool may reduce the pool's entropy by a small but non-zero amount. The first reason for this is easy to explain: no matter how much entropy is in the pre-stirred pool, the entropy (number of possible states) of the stirred result cannot possibly be higher than the number of possible output values from the hash function: for example, a pool estimated at 2,000 bits of min-entropy that is hashed using a single application of SHA-512 will end up with at most 512 bits of entropy following the stir. The more subtle reason is that each application of a hash function technically "costs" some entropy due to the existence of hash collisions that could map many possible pre-stirred inputs into a slightly smaller number of hash digest values. None of this has any real practical impact, provided the hash function output length is much

[4]Note that the formal definition of an extractor requires that also must take in a uniformly random *seed*. This raises a chicken-and-egg problem: if we don't have random output yet, *where do we get this seed from?* In practice, most deployed systems simply ignore this theoretical requirement and use an unkeyed hash function, or they follow the NIST recommendation and use a keyed MAC with a hard-coded key, or a key derived from *unprocessed* entropy from the generator [138]. From a scientific perspective this is all pretty squirrelly, but everything seems to work fine anyway.

DRAFT

larger than the security level you care about.[5]

11.1.2 Statistical testing

From time to time someone approaches me with a new idea for a (T)RNG. Their next question is typically: *how do I prove that my generator actually produces random output?* I get depressed when this happens. Usually if someone is asking me these questions, it means that they are about to make some poor life choices.

The problem here is that it is extremely difficult to prove that a process is unpredictable, let alone random. Here I use the word "difficult" not in the sense of "it might take some time", or "it will cost a lot of computing power," but rather in the sense of: *it may be computationally intractable.* The challenge is that there exist many entirely non-random, predictable processes that will satisfy nearly every public statistical test that we have ever devised. For example, modern cryptographic algorithms such as hash functions and block ciphers are explicitly designed to produce cryptographically pseudorandom output, and generators that post-process their output using these components will pass statistical test suites with flying colors — even if they are highly insecure when analyzed more deeply.

What statistical tests can do is *detect* a limited set of non-random behavior. If you're simply designing a security system, you should never, ever need to use these tools — just use a well-studied RNG design and/or a pseudorandom generator that has a security analysis. If you really do need these tools, you're probably doing something advanced and/or possibly dangerous. Please don't tell me about it.

DRAFT

If you want to test randomness anyway. With these admonitions out of the way, many statistical tests do exist that can detect non-random behavior in generated bit sequences. These tests can be valuable for detecting obvious biases, non-random sequences and other defects. Although this list is very incomplete, some tests include:

1. **Frequency tests.** The simplest metric for testing long sequences is to measure the frequency with which "0" and "1" symbols appear. This test can be applied to an entire sequence, or to smaller substrings within the sequence. In each case, the fraction of bits matching either value should converge towards 1/2 as the measured sequence grows longer.

2. **Number and length of "runs."** A *run* is just a sequence of multiple consecutive digits where all digits have the same value, such as `11111`. Tests sometimes measure the number of runs, or the length of the longest run of one specific digit. The measured result is compared to the expected value of the longest sequence in the same number of random bits. This test helps to detect "stuttering" generators that produce long sequences of the same output bit.

[5] Some recent designs such as SHA3's SHAKE (see §11.2.3) use a "sponge" design that replaces the hash function with an unkeyed function that permutes the input: this ensures that no entropy is lost due to collisions, but also creates a new risk that an attacker who compromises the state might be able to "rewind" the current state into past states. Life is full of tradeoffs.

3. **Fourier transform.** Fourier transforms can be used to convert random bits into *spectral* components, and can be used to detect unusual characteristics, such as repetitive patterns in data. This data can be compared with expected random spectral data by measuring the number of components that exceed some threshold.

4. **Random excursions tests.** This test uses an input sequence to define a random walk, and then calculates statistics such as the numbers of visits to each state.

5. **Compression tests.** As Kolmogorov noted, true random sequences should be largely incompressible. Some statistical test suites take this admonition literally by applying standard compression algorithms such as LZW [**?**] or Maurer's algorithm [117] and measuring the size of the result. It's worth noting that this approach will only detect bad sequences that possess specific characteristics that align with the compression algorithm's design (e.g., frequently-repeated sequences, non-uniform distribution of bytes.)

In general, these statistical tests will calculate a measured result over a given sequence, then will compare this to an expected result calculated based on the expected properties of a true random sequence. The two values are compared using a metric (for example, a *chi squared* test), and the test is considered successful if the metric meets some threshold chosen by the test designer.

Modern statistical test suites will run many additional tests beyond the ones listed above. For example, NIST's Special Publication 800-90B describes dozens of tests [138], and NIST publishes an automated test suite comprising fifteen different tests [153]. Some deprecated suites such as DIEHARDER [46] (an updated version of DIEHARD [115]) include still further tests of varying sophistication.

DRAFT

11.1.3 Hardware RNGs

A number of hardware approaches can be used to derive unpredictable numbers. These generators break down into various categories.[6]

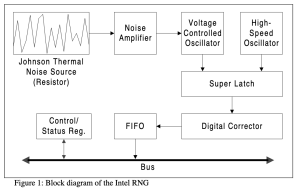


Figure 11.3: Block diagram of an early version of the Intel SecureKey hardware random number generator. Matt: Cite Kocher paper

[6]For an excellent and more complete discussion of these designs, see [163].

Noise-based RNGs. A common approach to designing electrical RNGs is to employ some source of thermal noise, such as the Johnson-Nyquist noise created in a resistor, and then *amplify* this signal. This amplified signal can then be further processed using a comparator circuit that signals when the amplified noise signal exceeds some voltage threshold. This is a relatively simple design, but it has several challenges. First, the level of noise and the amplifier need to be tightly correlated in order to ensure correct performance. Moreover, these designs may be sensitive to external influences such as temperature. Finally, and more critically, this design does not necessarily produce uncorrelated random bits without some further processing.

Intel SecureKey RNG. One example of a noise-based RNG design can be seen in the Intel SecureKey generator (Figure 11.3) [159, 163, 55], which was introduced in the early 2000s to most Intel processors.[7] The Intel RNG uses amplified noise and a threshold comparator to generate an unpredictable low-frequency signal. This signal is used as a trigger: each time the comparator activates, the circuitry samples an an output "bit" from a second signal that is generated by a high-frequency oscillator. It is difficult to evaluate the exact quality of the raw output bits produced by this generator, since they are not provided to software developers via any public API. We can infer that the raw result must not consist of high-quality uniform bits, since Intel post-processes this output using (1) a von-Neumann debiasing process, and (2) a cryptographic "conditioning" procedure based on AES (see further below) before any results are provided to applications.

Intel Ivy Bridge RNG. The previous Intel design has some disadvantages due to the use of analog components, which increase power consumption and create manufacturing difficulties. More recent Intel processors employ a novel design that uses entirely "digital" components [165, 80]. This design, shown in Figure 11.4 uses a pair of inverters organized in a feedback loop. Under ordinary circumstances, these gates will stabilize so that if the output if the first gate is "1", then the output of the second gate is "0", or vice versa. A set of transistors can override this stable arrangement: when switched on, they force the inputs and outputs of both inverters into the 1 state. When the transistors are switched off, the circuit will then fall back into one of the two possible stable arrangements. However, the choice of *which* of the two possible states to fall into is unpredictable, and should largely be determined by electrical effects such as thermal noise. While this overview sounds simple, the challenge in this design is that it requires both gates to be perfectly balanced: in practice, small manufacturing differences can cause one state to be more likely than another, and the circuit must be "tuned" to reduce bias. The output of this generator is also post-processed using the de-biasing and conditioning techniques described above.

The conditioned output of both Intel generators is made available to software via a special instruction named RDSEED. These bytes are also used to seed a pseudorandom generator based on CTR_DRBG (see §11.2.3), and the resulting pseudorandom bits are available via the RDRAND instruction.

[7] More recent processors use a slightly different design.

DRAFT

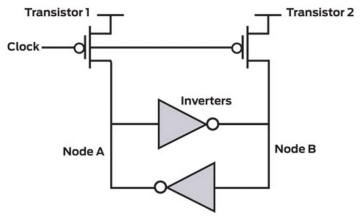


Figure 11.4: Block diagram of the Intel Ivy Bridge hardware random number generator, from [165].

Quantum RNGs. According to standard interpretations of quantum mechanics, many quantum properties of physical systems are fundamentally unpredictable and random. At the same time, physics offers us the tools to calculate the probability distributions from which these random events will be sampled. Examples of quantum effects include electrical effects such as shot noise, the polarization of photons in a laser stream, or even the decay of radioactive particles. While in principle all hardware RNGs are to some extent reliant on quantum effects (for example, shot noise and flicker noise in electrical circuits can contribute to the noise used by electrical RNGs), some "quantum" RNG designs have proposed to more directly measure these effects.

While quantum RNGs often sound like a good idea, they have many downsides. The most obvious of these is cost. However, from a security perspective, the biggest challenge is that they rely on sensitive measurements that can often be swamped by macroscopic effects such as thermal noise or RF interference, either due to manufacturing defects or poor calibration. Moreover, there is no demonstrated need for quantum randomness. That is: there exists no general attack against non-quantum RNG designs such as electrical noise generators, and any attacks against specific equipment is usually due to manufacturing defects or miscalibration, both of which can occur in "quantum" generators as well. This does not prevent the industry from selling many quantum designs of unknown quality. I refer the reader to [163] for a discussion of many specific techniques.

Bit pattern	Allowable number of occurrences per 256-bit sample
1	109 < n < 165
01	46 < n < 84
010	8 < n < 58
0110	2 < n < 35
101	8 < n < 58
1001	2 < n < 35

Table 1: Health bounds for 256-bit samples

Figure 11.5: Examples of statistical "health tests" run on input samples by the Intel Ivy Bridge random number generator, excerpted from [80].

DRAFT

Health testing. A danger with hardware generators is the possibility of malfunction due to environmental effects or manufacturing defects. To detect these conditions, generators typically contain circuitry that is designed to monitor the quality of raw generator output in order to detect obviously problematic conditions, such as continuous runs of the same digit or an unusual numbers of occurrences of a specific substring. Standards such as FIPS 140 and ISO/IEC 19790 mandate this form of runtime test, both at startup and continuously during operation [133, 95].

In the event of a test failure, generators can take two possible actions. In some cases, generators will simply discard output strings that fail to satisfy tests. For example, the Intel RNG performs a series of simple statistical tests (see Figure 11.5) that are anticipated to fail with approximately 1% probability even when run on uniformly random strings: this guarantees that occasional failures are an expected and routine occurrence [80]. In the event of a more significant test failure, the RNG may enter an error state and cease production of further outputs. This can be catastrophic for cryptographic systems: to give just one example, a Linux Kernel RNG may cause a kernel panic if the RNG runtime tests fail [51]. Note that it almost never makes sense to perform health tests on bits that have been *conditioned* using cryptographic algorithms, but I have seen people do this anyway.

Post-processing and debiasing. Most hardware generators do not produce uniform and unbiased output bits. This means that the output bits need to be further processed before use in cryptographic protocols. In the event that output bits are *biased* but otherwise independent (meaning that the probability of sampling a "0" bit is p, for $p \neq 1/2$) then this bias can be eliminated using a simple technique known as von Neumann debiasing [**?**]. In this approach, input bits are considered in sequential pairs (b_0, b_1). If $b_0 = b_1$ then the pair is discarded. Otherwise, the pair is replaced with the single bit b_0. This technique will work for any value of p, although the output rate drops precipitously as p grows far from $1/2$.[8]

Unfortunately the assumption that individual bits are uncorrelated and independent is often invalid. In these cases, the designers of hardware RNGs will typically post-process (or "condition") the output of a generator by applying some cryptographic technique. A common approach is to process the output using some hash function or block cipher, such as AES. For example, the Intel generators each process data by collecting sequences of bits and using AES-CBC-MAC to produce conditioned output [159].[9]

11.2 Pseudo-random Number Generation

When true random numbers prove insufficient, most systems will produce additional bits using a pseudorandom generator, or PRG. A PRG is not random at all, but is instead a

[8] It is easy to see why this works. Assuming the probability of sampling a "0" bit is p, and the probability of sampling a "1" bit is $1 - p$, then there are four possible outcomes for each pair (b_0, b_1). With the two $b_0 = b_1$ cases discarded, the probability of $(0, 1)$ and $(1, 0)$ are each identical, at $p \cdot (1 - p)$ respectively.

[9] In [159], Shrimpton and Terashima discuss some theoretical weaknesses of this design. As is common in applied cryptography, these appear to have mostly been ignored.

DRAFT

deterministic algorithm. In principle, PRGs "stretch" a random string into a much longer sequence of output bits, with the property that the resulting bits cannot be distinguished from true random bits.

Non-cryptographic constructions. Before discussing pseudorandom generators, we need to address an elephant in the room: there are many highly-insecure, *non-cryptographic* algorithms that are often referred to as "pseudorandom number generators." These algorithms are wholly inappropriate to use in any security-critical application, and will often turn up in places where they can do great harm. Common examples include the C language's Linear Congruential Generator (LCG), as well as the Mersenne Twister algorithm which is deployed as the default `random` library in languages such as Python. Aside from producing biased output in some cases, a key danger with these algorithms is that an attacker can perform *state recovery attacks* that recover the generator's state/seed, given a reasonable number of output bits.Matt: Do we want to include more on these in an appendix?

In the real world you'll sometimes see developers use the term *cryptographically-secure* PRNG (CSPRNG) to label generators that are safe for security applications. I won't use that nomenclature in this book because I believe that cryptographic security is already bound up in the term *pseudorandom.* I also feel strongly that non-cryptographic generators should *never* be exposed to software developers as a default choice in any language or programming framework. My reasoning is as follows: a cryptographically-secure generator will work perfectly for any non-security application (such as statistical simulation or graphics rendering), whereas a non-secure generator can cause grave harm when used in security-critical applications. The only possible reason to select an insecure generator is to speed up certain performance-critical (but non-security) applications. While I can appreciate the need to optimize, I believe that security should come first. It is perfectly fine for a knowledgeable developer to select an insecure-but-fast generator: they should simply do so intentionally.

DRAFT

11.2.1 Security and simple constructions

As discussed above, the core feature of a pseudorandom generator is the fact that its output is cryptographically indistinguishable from true randomness. Still, this is a vague definition. Before we get to the details of how real PRGs work, it's helpful to obtain a better idea of what security actually means. This is usually stated using one of two possible definitions:

1. **Yao's Indistinguishability.** This definition demands that there is no efficiently-computable "statistical test" that can determine the difference between pseudorandom bits from a properly-seeded PRG (where the seed is kept secret), and an equal-length string of truly random bits.[10] Note that "statistical test" here does not indicate some a

[10]There is a lot of technical detail being hidden in this explanation! As usual, "efficiently-computable" typically means *probabilistic polynomial time* in the security parameter, and in practice requires that any attack be super-polynomial time in this value (see §**??** for details.) It is also important to recognize that given enough runtime, some attacks do exist: for example, an attacker can simply brute-force guess the PRG seed until she finds one that matches the output. Fortunately if the seed space is reasonably-sized (for

specific algorithm from a statistics textbook, but instead refers to *any* algorithm that can evaluate the given string and make some determination.

2. **Blum and Micali's next-bit unpredictability.** In this definition, an attacker is given many bits of output from the PRG, and (at a point in the output stream the attacker chooses) must predict the next bit that will emerge from the generator. The PRG is secure if there is no efficiently algorithm that can predict the next bit with probability meaningfully different from 1/2.

Both definitions can give practitioners a good intuition for what properties they can obtain from a generator. For example, Yao's definition explains why it is safe to use the output of a correctly-seeded PRG in place of true random numbers in your protocol. Just as a thought experiment, imagine that some cryptographic protocol is secure when it is run using true random bits, and yet there exists an (efficient) algorithm that can "break" the security of the protocol when the protocol is executed using pseudorandom bits. Since we specified that this attack algorithm only works when the protocol is executed using pseudorandom bits, we can build a "statistical test" to determine whether a stream of bits is random or pseudorandom as follows: (1) run the protocol using the given bits as input, (2) use the attack algorithm to try to break the protocol, and (3) announce that the bits are pseudorandom if the attack succeeds. Since Yao's definition tells us that no efficient statistical test should be able to distinguish secure PRG output from true random bits, we know that this thought experiment cannot be valid: any attack algorithm must work about as well when the protocol is run with true random bits as it does when the protocol is run with pseudorandom bits.

I also find that the next-bit definition is useful for understanding the way PRGs are used in common software implementations. Often attackers will use a single generator to generate both public values (such as nonces) that will be sent over the wire — and hence the adversary will see — as well as cryptographic secrets that must remain unpredictable. A generator that satisfies Blum and Micali's definition ensures that no matter how many public values the attacker sees, they should not be able to predict any subsequent bits.

The good news is that you do not have to choose one or the other of these definitions: they turn out to be equivalent, meaning that any generator that satisfies one of them will necessarily satisfy the other [**?**]. It should go without saying that these promises only hold if the seed is sufficiently long, perfectly random and obviously not known to the adversary. If a system leaks its PRG seed or uses non-random seeding material, then all bets are off. We'll talk more about these exceptions further below.

Cycle length and state size. A pseudorandom generator is initialized with a seed, which produces its initial internal "state." If the generator outputs many values, this internal state must necessarily change in some manner, to ensure that it does not simply repeat the same output. One consequence of this fact is that — unless it is reseeded — every fully deterministic generator with a finite amount of memory will eventually enter a "cycle" and

example, 128 bits, this attack is unlikely to be practical.

DRAFT

begin to repeat previous output. This occurs because there are only a finite number of internal states that can exist, and after a sufficient number of state updates, the generator must necessarily arrive at an internal state that it has used before.

Fortunately, modern cryptographic PRGs are designed to have so many possible internal states that this concern can mostly be ignored. Typically the number of internal states $\mathcal{S}$ in a generator is on the order of 2^{128} to 2^{256} or more.[11] Some designs update their internal state using a pseudorandom approach — for example, by generating the next internal state based on the output of the generator using the previous state value. In those designs, the birthday paradox (§4.2.1) can apply: this means one should expect to encounter a repeated state value with high probability after the internal state has been updated $\sqrt{\mathcal{S}}$ times. For example, a generator with 2^{128} states is likely to enter a cycle well before 2^{64} state updates. This is why it is useful to employ generators with large internal state sizes.[12]

Some example PRGs. PRGs are relatively easy to build using standard cryptographic primitives such as block ciphers, stream ciphers and hash functions. There are also designs exist that use more exciting mathematics such as elliptic curve cryptography, RSA and lattices; however these alternatives are rarely used in deployed systems.

PRGs from stream and block ciphers. Although it might seem obvious, most stream ciphers are already designed to be pseudorandom generators: their purpose is to generate a long pseudorandom *keystream* that can be combined with a plaintext using XOR. These keystreams can be directly used as pseudorandom bits. In practice, this is one of the most popular approaches to building PRGs: ciphers such as ChaCha20 (§**??**) or AES in CTR mode (§**??**) are used as the building block for most operating system PRGs.

PRGs from hash functions. Many cryptographically-secure hash functions can be used to generate pseudorandom bits as well. NIST's HASH-DRBG and HMAC-DRBG standards [22] both convert a hash function such as SHA-2 into a secure PRG (see Figure 11.6.) The more recent SHA-3 standard [**?**] even defines a special mode called SHAKE that can be used to ingest a high-entropy seed value and then produce arbitrary amounts of pseudorandom output (see §11.2.3.)

PRGs from mathematical primitives. In principle, any one-way function can be used to produce a pseudorandom generator (see §**??**.) Many such functions have been constructed from ingredients such as elliptic curves, factoring-based cryptosystems such as RSA, and lattice-based cryptography. At least two PRGs have been standardized based on these problems, with some unexpected results (see §11.3.1.) The main argument against these algorithms is that they tend to be much slower than hash- or cipher-based constructions, in exchange for no real security benefit.

We discuss some concrete examples further below.

[11] This corresponds to an internal state value of (at least) 128-256 bits.

[12] Aside from state size limitations, many standard generators have much lower limits due to other constraints on the design. For example, NIST's CTR-DRBG using 128-bit AES specifies a limit of 2^{32} outputs.

DRAFT

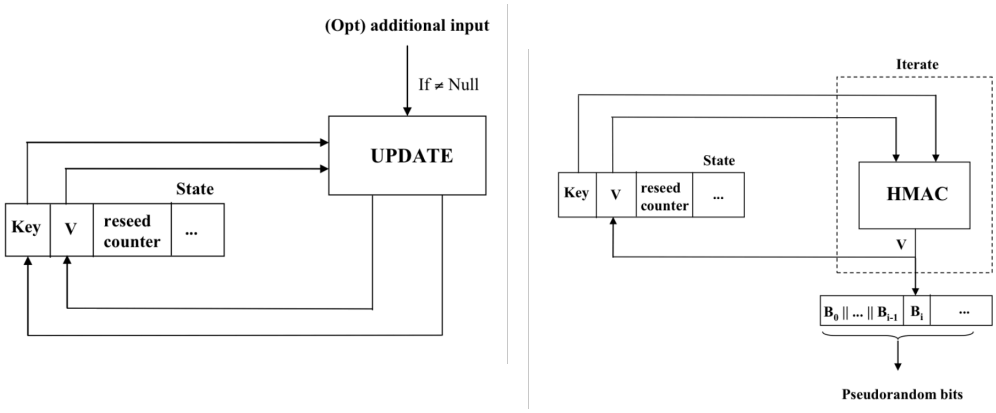


Figure 11.6: The seed update (left) and output stage (right) of the HMAC-DRBG generator from NIST SP800-90A [22]. The state consists of a key K and a value V, along with a reseed counter. In the output phase (right), HMAC is repeatedly invoked to generate output bits and update a portion of the seed, V. Periodically the update procedure (left) updates the HMAC key K as well, using optional additional input material.

11.2.2 Real-world considerations

While it's possible to build a simple PRG using the basic approaches described above, deployed cryptographic systems often add many additional requirements. These include:

Forward security. Some older operating system RNGs would sample a cryptographic seed at system boot, then use this seed to generate all outputs until the next reboot. This can be very risky in practice! The main concern in this design is the possibility that the fixed seed could become known to an attacker, who would then be able to re-generate all *past and future* pseudorandom bits used by the system. In a worst case outcome, this could allow the attacker to decrypt every TLS connection ever made by the machine.[13]

To prevent a seed compromise from revealing *past* generator outputs, many standardized PRG designs employ a *forward-secure* construction. In this approach, the generator periodically updates its current seed (or *state*) in a one-way manner. In a well-designed construction, even if an attacker obtains the full current state of the generator at some point in time, they should not be able to compute previous versions of the generator state. The result is that all outputs produced by the generator *prior* to the state compromise should

[13]While the idea that an attacker could compromise a seed (or current *state*) of a generator might seem paranoid, my colleagues and I have discovered such flaws in security-critical systems [**?**, 87]. One potential vector for seed compromise is an information disclosure software vulnerability that enables the attacker to read portions of kernel or application memory space. Seed leakage can also occur when seeds are included in VM snapshots, and the VM snapshot itself is compromised or simply restored to production multiple times. In this case, each time the system is restored from the VM image, it will use the same sequence of pseudorandom numbers. In some instances this can be catastrophic [87]. The most extreme example I have ever seen was due to a manufacturer that hard-coded a fixed seed (drawn from FIPS test vectors) into all of their VPN devices, rendering all connections exploitable (see §11.3.2.)

DRAFT

remain unpredictable (and indistinguishable from random.)

These forward secure designs are relatively easy to build. A common technique is to periodically update the seed/state of the generator by (1) employing the generator to generate some pseudorandom output bits, and then (2) using these output bits as a new seed/state. The older state is then destroyed, or "zeroized".[14] For example, each of the standardized generators in NIST's Special Publication 800-90A [22] incorporates this idea into their design under the name "backtracking resistance" (see §11.2.3 below for more on these designs.)

Post-compromise security. While forward-secure designs can protect past outputs following state compromise, this will not prevent an attacker from computing *future* outputs of the generator. The problem stems from the fact that generators are deterministic: the attacker can simply run the algorithm forward from any internal state to produce all future states and outputs. A standard solution to this problem is to periodically *re-seed* the generator with fresh randomness (or entropy) from whatever source the system has available. Modern generator algorithms will often "mix" this fresh seed material into the current state.

Note that it is critically important that these re-seeding attempts include a sufficient amount of entropy! Consider an attacker who compromises the full generator state at time t, and then sees generator output bits at time $t+1$ after some fresh seed material has been added. If the quantity of added seed material is small enough, the attacker may be able to (1) brute-force guess every possible value of the new seeding material, and (2) verify whether a guess is correct by computing new output and comparing it to real output generated at the later period. Even if, say, 128 bits of seed material is added to the generator over some time period, this attacker may be able to guess the current state with much less than 2^{128} effort, provided the new seed material is added just a few bits at a time.[15] This is why it is important to collect fresh entropy in a location where the attacker cannot see it (*e.g.,* in an entropy pool) and then add it to the generator only when a large quantity has been collected.

Personalization strings and multiple generators. An interesting recent trend is to use multiple independent PRGs to generate different values used by the system. This can occur naturally when different applications seed their own PRG instance from the system generator. However, in some more opinionated implementations such as Amazon's s2n TLS library, one generator instance is used to produce secret keys, while a different instance is used to generate non-secret values such as nonces. The operative theory of this design is that a flaw (or even a deliberate backdoor!) in the non-secret instance should not leak information about the data produced by the secret instance.

While it is debatable whether this threat model is worth the trouble, it is not particularly expensive to implement. Some other designs such as NIST's SP800-90A framework [22]

[14] This approach works because PRGs are necessarily "one-way". This is due to the fact that PRG output expands its input and is indistinguishable from random. An efficient adversary that can recover the seed used to generate PRG output will necessarily violate this property, since it should not be possible to find such a seed that works for truly random bits.

[15] For example, if 128 bits of fresh seed material is added, but only one bit at a time (and the attacker sees generator output after each addition) then she will only need to guess at most $2 \cdot 128 = 256$ different values, as opposed to 2^{128} if all 128 bits are added in one go!

DRAFT

attempt to achieve a similar purpose by allowing the caller to provide a "personalization string" that is cryptographically folded into the output.

11.2.3 Deployed PRG designs

A number of formal and de-facto standards exist for pseudorandom generation. Here we provide a short list.

CTR_DRBG, HMAC_DRBG and HASH_DRBG

NIST's Special Publication 800-90A [22] specifies the design of three pseudorandom generators, also called *Deterministic Random Bit Generators* (DRBGs) in NIST parlance. These generators are based on block ciphers, hash-based MACs, and hash functions, with designations CTR_DRBG, HMAC_DRBG, HASH_DRBG respectively.[16]

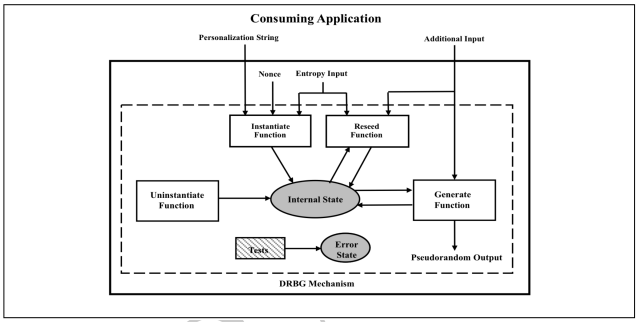


Figure 11.7: Architecture diagram for the DRBGs of NIST Special Publication 800-90A [22].

Functional specification. Each SP800-90A generator features the common API illustrated in Figure 11.7. The Instantiate function takes in an initial seed for the generator and sets up its state, Generate(n) produces a stream of n pseudorandom output bits, Reseed folds new seed material into the generator's internal state, and Uninstantiate "zeroizes" the state. Within the Generate function, each generator can also fold in any amount of additional entropy (called *additional input*) at each step of generation.

All three generators share a common template for the Generate function. An abstract diagram of this template is shown in Figure 11.8. In this generic design, the generator contains two distinct pseudorandom (irreversable) functions labeled F and G. An initial seed/state is drawn in at the beginning of generation, and then this state is repeatedly updated using the F function to produce new internal states. The internal state is also used to produce pseudorandom output bits via the abstract G function. In practice, both functions

[16] A previous version of the document also specified a fourth generator, called the Dual Elliptic Curve DRBG, or Dual_EC_DRBG. This deprecated generator has an exciting history, which we discuss in §11.3.1.

are implemented using different calls to the same underlying cryptographic primitive, such as a block cipher or hash function.

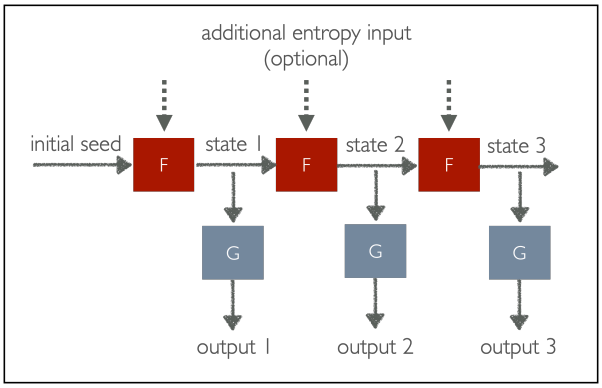


Figure 11.8: Abstract diagram of the Generate algorithm for all generators in NIST Special Publication 800-90A [22]. The functions F and G are instantiated with various cryptographic primitives.

In the interest of preserving the reader's sanity, we will describe only one algorithm in detail. The others are quite similar.

CTR_DRBG. The most widely-implemented generator from SP800-90A is CTR_DRBG, which is typically instantiated with AES on processors with hardware support for the cipher.

As the name implies, CTR_DRBG realizes the generation algorithm using AES in CTR mode. The internal state compromises a pair of strings $(\mathbf{Key}, \mathbf{V})$. When a user requests the generation of n output bits, the Generate algorithm enciphers $\mathbf{V}$ under $\mathbf{Key}$ to produce a block of pseudorandom output bits equal to the block size of the cipher. If additional bits are required, the (counter) value $\mathbf{V}$ is incremented and the process is repeated until a sufficient number of bits has been produced.

At the conclusion of each Generate call, the generation process calls the Update function, which uses a similar generation process to produce fresh pseudorandom values to replace each of $(\mathbf{Key}, \mathbf{V})$, ensuring that forward security holds following each call to Generate. Figure 11.9 gives a block diagram of the Generate and Update procedures.

Linux's ChaCha20-based PRG

In versions of the Linux kernel from 4.8 onwards (released in 2016), Linux incorporates a built-in pseudorandom generator based on the ChaCha20 cipher. In most Linux deployments, this generator is periodically seeded with material drawn from the system entropy pool, and then used to generate all bits produced via the `getrandom()` system call as well as the `/dev/random` and `/dev/urandom` block devices [70].

The core PRG algorithm is somewhat similar to the AES-based CTR_DRBG algorithm above. The generator is initially seeded with 256 bits drawn from the OS entropy pool. For

DRAFT

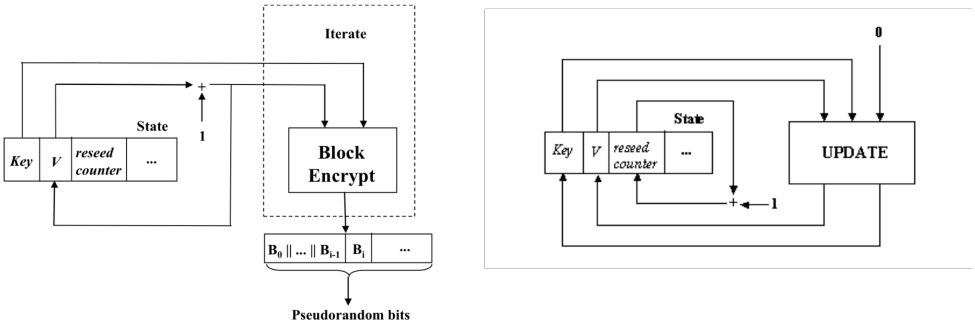


Figure 11.9: Generate (left) and Update (right) algorithms for CTR_DRBG. Note that Update is called as the final operation in each call to Generate. Source: [22].

each generate call, a 32-bit counter and 96-bit nonce is initialized, and then the generator produces output bits in 512-bit chunks until the desired output length has been produced: the counter is incremented for each block of output. To ensure forward secrecy, at the conclusion of the generation process a final block of 256 bits is generated to use as a new key and the original key is zeroized.

In systems with multiple CPUs, Linux maintains a single base instance of the ChaCha20 PRG, as well as a distinct secondary instance of the generator for each CPU. These secondary instances are seeded from the output of the core instance.

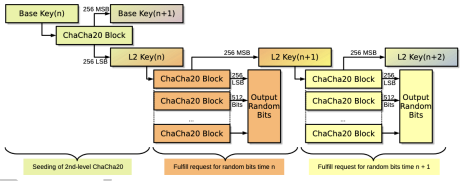


Figure 11.10: Diagram of Linux's ChaCha20-based PRG, showing the base generator seeding a secondary generator. Source: [70].

Sponge-based PRGs

Some recent hash function designs such as SHA-3 employ a *sponge* construction (§**??**). An important characteristic of this design is that it can be used to extend a seed into an arbitrary-length pseudorandom output, provided that the internal function is cryptographically strong.

Review of the sponge design. The main novelty in sponge constructions is the use of an internal function that does not compress its input. Instead, this function is typically implemented via a fixed *permutation* that draws in b bits and outputs b bits. To construct a sponge, the internal function is iteratively applied in two phases: in the *absorbing* phase,

DRAFT

data is fed in over multiple rounds; in the *squeezing* phase, pseudorandom data is read out (see Figure 11.11.)

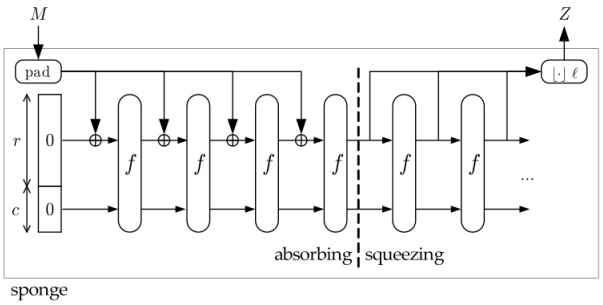


Figure 11.11: Overview of a sponge construction. A message is drawn in ("absorbed") on the left, and data is generated ("squeezed") on the right. Source: [32].

In more detail, each invocation of the internal function splits the internal state array into two portions: the first r bits are known as the *rate*, and the final c bits are referred to as the *capacity*. During the absorbing phase, input bits are added to the rate via XOR. During the squeezing process, the rate portion comprises the output of the construction, while the capacity bits are left to make up the secret internal state.[17]

Extensible Output Functions: SHAKE. In some applications such as the implementation of encryption padding (§**??**), a random seed (or appropriate high-entropy input) must be stretched into an arbitrary-length sequence of pseudorandom bits. One solution is to use an *extensible output function* (or XOF) such as the SHAKE construction given in the SHA-3 standard [**?**].[18] Each of these functions applies the sponge construction to absorb the input seed material, and then to squeeze an arbitrary length output from the resulting state.

While the SHAKE construction may be suitable for generating short pseudorandom sequences, it should not be used to implement a system or application-level PRG. This is mainly because the basic sponge construction does not provide forward or post-compromise security. The problem here is twofold: first, since the permutation is invertible, any compromise of the full internal state will allow an attacker to "rewind" the generator to obtain past states and outputs. Second, the SHAKE construction does not allow the inclusion of new seed material during the squeezing phase, which means that the generator cannot recover security following a state compromise.[19]

[17] It is critical that c be large enough that the internal state is computationally unpredictable to an attacker.

[18] Two such functions are defined in the SHA-3 standard, known as SHAKE128 and SHAKE256 respectively. Here the numerical term refers to the internal security level of the function.

[19] It is worth nothing that some research works (*e.g.,* [33]) have proposed improved sponge-based constructions that allow for fresh seed material to be added during the squeezing process, which partially addresses the post-compromise security concerns. However, these constructions may still lack forward security due to the invertible permutation. Moreover, they are not certified by standards organizations.

DRAFT

PRGs from Number Theoretic Problems

It is possible to build secure pseudorandom generators that base their security on well-known mathematical problems related to integer factorization, lattices, or elliptic curve cryptography. This guarantee can appeal to naïve practitioners, since these "mathematical" problems may feel more natural than assumptions about human-constructed primitives like block ciphers.

My view is that any perceived security advantage from these techniques is mostly an illusion: while it is possible that we will find flaws in our hash functions and ciphers that make current PRG designs insecure, it seems equally likely that we will devise breakthrough cryptanalytic results against factoring or elliptic curve assumptions. The computational cost of these generators is also typically quite high. This is due to the fact that the underlying mathematical operations (modular multiplication, elliptic curve point multiplication, matrix multiplications) tend to be much more time-consuming than most symmetric algorithms such as block ciphers.

Nonetheless: for information purposes, I will give a brief overview of some of these algorithms, with the warning that you shouldn't use any of them in real systems.

Factoring-based generators: Blum-Blum-Shub (BBS). An early result from by Lenore Blum, Manuel Blum and Michael Shub [39] provides an inefficient construction for obtaining pseudorandom bits based on the factoring problem. In this construction, the private parameters for the generator consist of an RSA-style modulus of the form $N = p \cdot q$ where p, q are large primes with some specific properties, as well as an initial seed $x_0 \in [2, N)$ that should be uniformly sampled and co-prime to N. Generation is now simple: for $i = 1$ to n, the generator simply computes $x_i \leftarrow x_{i-1}^2 \ mod \ N$ and outputs the least significant bit of x_i.

The provable security of the BBS generator is based on two results. The first is that inverting the function $F(x) = x^2 \ mod \ N$ (for random x and properly-constructed N) is provably as hard as factoring N. The second is that the least significant bit of x is a *hardcore bit* for this function (§**??**): this means that an attacker given $F(x)$ should not be able to compute this bit with non-negligible advantage, unless they can also invert $F(x)$. An obvious downside of the BBS generator is that it is extremely costly: each squaring operation produces only one bit of output,[20] and the modulus N must be large (*e.g.,* at least 3,072 bits to ensure 128-bit security), which further slows the generator. As a final note, factoring-based PRGs are not quantum secure.

Micali-Schnorr and Dual_EC_DRBG. Although it is possible to generate pseudorandom bits using a variety of lattice- and number-theoretic constructions, most of these ideas remain firmly within the research literature. Only two such generators have been proposed in production standards. The most famous of these is the Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG), which was deprecated from NIST SP800-90A

[20] In practice it is possible to output $log \ log \ N$ bits of output, but this is still a small number.

DRAFT

due to credible allegations that it contains an NSA backdoor (see §11.3.1 further below.) The second appears in an older proposed ISO standard for an RSA-based generator called Micali-Schnorr [**?**], which can be viewed as a more efficient variant of the Blum-Blum-Shub generator. Neither technique is currently included in any active standard, nor should either generator ever be used.

11.2.4 Testing PRGs

A nice feature of pseudorandom generators is that they are deterministic algorithms: compared to "true" RNGs, their correct operation is relatively easy to verify.

Known Answer Tests. A standard approach to testing PRG implementations is to use test vectors, also called *Known Answer Tests* (or KATs.) A KAT is simply a list of seed inputs that can be provided to the generator, as well as the known output bits that should emerge. FIPS and ISO security standards provide guidelines for implementing KATs, as well as KAT vectors for most standard PRG designs.

The challenge in using KATs is that a PRG implementation must be extended to include a "testing" harness in which the normal seeding functions are overridden, and chosen seed material can be injected into the generator. Developers must be very careful that testing mode does not inadvertently become active during production usage, since this can produce a major security incident (see §11.3.2 for an example of a manufacturer who used a test key in production.) A second challenge is that it is hard to build a testing pathway that guarantees correct behavior in normal operation. I am aware of at least one FIPS-certified module where the RNG failed to operate during normal usage, but still passed all FIPS KAT tests because the test harness bypassed a bug in the initialization logic [114].

Self tests. Many security standards also mandate that PRGs pass runtime "self tests", such as ensuring that the generator does not produce sequential repeated outputs. While this form of runtime testing makes sense for hardware RNGs, it probably offers much less utility when applied to PRG outputs. While I have no doubt that there exists bugs that can cause these tests to fail, the cryptographic nature of PRG algorithms ensures that they will often produce statistically "plausible" output even when they are unseeded or seriously malfunctioning.

11.3 RNG nightmares

The failure of an RNG can lead to catastrophe for all cryptographic software that uses it. In this section we give several examples of accidental and *engineered* RNG failures. While these specific vulnerabilities have since been mitigated, these failure modes can be instructive when thinking about contemporary RNG designs.

DRAFT

11.3.1 Dual EC DRBG

Beginning in the late 1990s, NIST and the National Security Agency began collaborating on a new set of pseudorandom generators, called *deterministic random bit generators* (DRBGs.) This work eventually resulted in two standards that were published in 2006: ISO 18031 [93] and NIST's Special Publication 800-90A [21]. Both of the standards incorporated a new, NSA-designed algorithm known as the *Dual Elliptic Curve DRBG*, or Dual_EC_DRBG for short. The unusual aspect of this generator was its use of mathematics (elliptic curve operations) that are traditionally associated with public-key cryptosystems. This aspect of the generator is widely believed to have enabled a sophisticated cryptographic backdoor.

Overview of Dual EC. The generation process in Dual EC follows the basic framework shown in Figure 11.8. The standard specifies an elliptic curve subgroup of prime order q, as well as two hard-coded generator points P, Q that are explicitly specified in the standard. To generate bits, an initial seed s_0 is recursively processed by a function F to produce new internal states $s_1, s_2, s_3, \ldots$, while a second (distinct) function G produces pseudorandom output bits from each internal state. In Dual EC, every internal state is a scalar integer in the range $[0, q)$. The implementation of both functions uses elliptic curve scalar multiplication as follows:

1. The internal state update function $F : s_{i-1} \rightarrow s_i$ is implemented by computing $S \leftarrow s_{i-1}P$, and setting s_i to be the x-coordinate of S.

2. The output generation function $G : s_i \rightarrow r_i$ takes an internal state s_i and computes $R = s_iQ$. It then truncates the most significant 16 bits off of the x-coordinate of R, and outputs the remaining bits as r_i.

The security argument behind the Dual EC design was outlined in an analysis by Brown and Gjøsteen in 2007 [45]: provided that s is a random scalar and P, Q are random generators of the curve subgroup, then the resulting points (sP, sQ) should themselves indistinguishable from two randomly-sampled points in the subgroup.[21] In this argument, both the output of the generator and the next internal state are pseudorandom. Of course, a limitation of this analysis is that while indistinguishability holds for the output values viewed as *points* in the subgroup, but does not necessarily hold for the *bits* that are output by the generator. The problem is that there are many values for x in NIST's Weierstrass curves that will not satisfy the curve equation $y^2 = x^3 + ax + b \ mod \ p$, which has the effect of producing a measurable *bias* if the full x-coordinate is output at each cycle. Dual EC attempts to compensate for this issue by discarding a small number of the most significant bits of each x-coordinate before producing the remainder as output, although even this approach leaves a small but detectable bias in the generated bits.[22]

[21] This argument relies on two assumptions. The first is that the elliptic curve decisional Diffie-Hellman problem (§**??**) is hard in the given elliptic curve subgroups. The second, more problematic assumption, is that no party knows the discrete log relationship between P and Q.

[22] This bias was acknowledged by the generator's designers, who conducted a detailed security analysis and ultimately argued that it would not cause a problem for protocol implementations.

DRAFT

A backdoor in Dual EC? In 2007, the Microsoft cryptographers Dan Shumow and Neils Ferguson stunned the research community by presenting a short talk at a major cryptography research conference entitled "On the Possibility of a Back Door in the NIST SP800-90 Dual Ec Prng" [160]. The talk made the remarkable case that the *designers* of Dual EC could potentially hold a master "trapdoor' that would allow them to compromise the security of any Dual EC implementation.

Shumow and Ferguson's concern was the provenance of the specific elliptic curve points P and Q that are specified in the NIST standard. They observed that, while security might hold if these points were generated at random, the NIST standard did not actually contain any evidence that they were. Without this assurance, it was possible that the points could be generated so that the NSA, as an attacker, would know a *discrete logarithm relationship* between P and Q.[23] They argued that this secret knowledge would form a master "trapdoor" for the generator. An attacker with knowledge of the trapdoor could use it to recover the internal state of the generator given only a single short output, which would allow the attacker to *predict all future outputs of the generator*, at least until it was reseeded.

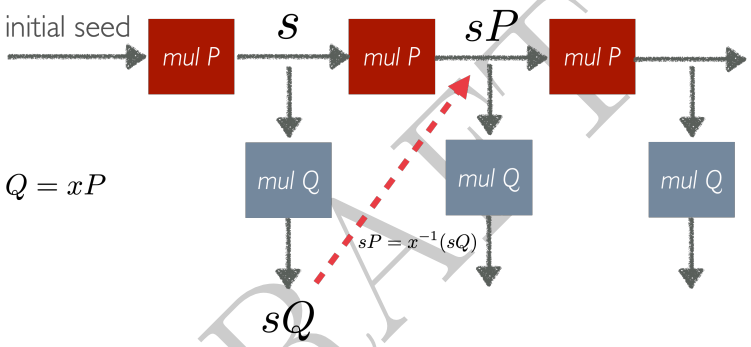


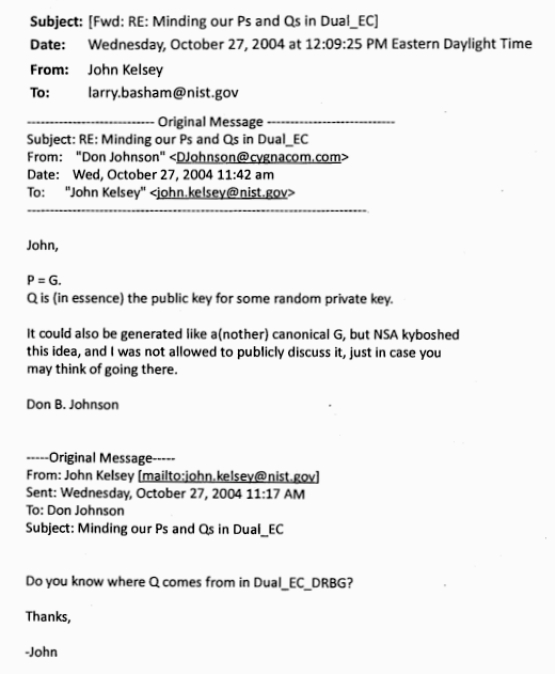
Figure 11.12: Illustration of the Dual EC generation process (grey arrows) with the backdoor illustrated (red arrow.) Knowledge of the discrete log relationship x such that $Q = xP$ allows an attacker to transform the output point sQ into the next internal state sP. This diagram shows only the point multiplications and omits the truncation of the output values, but this can be inverted by brute-force guessing the truncated bits.

The impact of such a vulnerability would be devastating to any cryptographic system built using Dual EC. For example, it could potentially allow a passive eavesdropper to decrypt every TLS and IPSec connection made or received by a device, thus rendering all communications transparent. This is because most implementations of these protocols will use the same generator to compute public nonces, such as the "Server Random" value used in TLS connections, prior to producing secret key material such as Diffie-Hellman ephemeral

[23] This implies that the attacker knows the (elliptic curve) discrete logarithm of Q base P, for example. It is easy to produce a pair of values with a known relationship: an attacker can select P at random, and then sample a random scalar x, setting $Q = xP$ (here multiplication indicates scalar multiplication of elliptic curve points.) There is no way to determine from the given points whether the author retained knowledge of x.

DRAFT

secrets. A passive eavesdropper with the trapdoor could simply observe a nonce from a TLS connection, then use the trapdoor to efficiently recover the secret keys and then decrypt the session (see [49] for a detailed impact analysis on several TLS libraries.)



Subject: [Fwd: RE: Minding our Ps and Qs in Dual_EC]
Date: Wednesday, October 27, 2004 at 12:09:25 PM Eastern Daylight Time
From: John Kelsey
To: larry.basham@nist.gov

---------- Original Message ----------
Subject: RE: Minding our Ps and Qs in Dual_EC
From: "Don Johnson" <DJohnson@cygnacom.com>
Date: Wed, October 27, 2004 11:42 am
To: "John Kelsey" <john.kelsey@nist.gov>

John,

P = G.
Q is (in essence) the public key for some random private key.

It could also be generated like a(nother) canonical G, but NSA kyboshed this idea, and I was not allowed to publicly discuss it, just in case you may think of going there.

Don B. Johnson

-----Original Message-----
From: John Kelsey [mailto:john.kelsey@nist.gov]
Sent: Wednesday, October 27, 2004 11:17 AM
To: Don Johnson
Subject: Minding our Ps and Qs in Dual_EC

Do you know where Q comes from in Dual_EC_DRBG?

Thanks,

-John

Figure 11.13: An email exchange between NIST's John Kelsey and Don Johnson at Cygnacom, an NSA contractor. This conversation about the provenance of P, Q occurred during the Dual EC standardization process in 2004.

The challenge for Shumow and Ferguson was that there was no way to definitively *prove* that the NSA had kept such a backdoor. Of course, any thoughtful security expert would view this standard as absurd — the mere suspicion of a master backdoor should have been sufficient to immediately revise the standard. Unfortunately, NIST leadership stonewalled critics, claiming that that it had "no evidence that anyone has, or will ever have, the 'secret numbers' for the back door that were hypothesized." To the great shame of everyone involved, NIST's refusal to act succeeded in ending the discussion. In large part this was due to the fact that the research community was unaware of Dual EC actually being widely deployed, and so many researchers assumed the warning had been sufficient to address the matter.

The Snowden leaks. In 2013, a former NSA contractor named Edward Snowden revealed a series of classified documents describing various NSA programs. One major revelation was the existence of a $250 million/year effort to subvert cryptographic standards, known as the "SIGINT Enabling Project". This project explicitly sought to incorporate vulnerabilities in commercial encryption products and public standards, with the intent of "enabling"

decryption by the NSA [144]. The New York Times article that reported on the leak specifically called out a classified reference to a successful tampering effort involving the 2006 ISO standard that contained Dual EC.

Events moved quickly following this leak. In late 2013, the security community began to take stock of the impact of a possible Dual EC compromise. Shockingly, they discovered that use of Dual EC was widespread due to the fact that it was included as the default generator in the popular RSA BSAFE library, which had been used in many FIPS-certified cryptographic products during the 2000s. This realization was followed by a Reuters report alleging that RSA Security had been received government contracts for incorporating Dual EC, even prior to the algorithm's standardization in 2006 [121]. Both NIST and RSA quickly acted to deprecate Dual EC, though this did little to undo any past damage, or to repair older versions of security products that included the library.

The Juniper Dual EC discovery. In December 2015, Juniper Networks announced the discovery of two critical CVEs in their Netscreen line of firewall/VPN devices [181]. These vulnerabilities were remarkable for the fact that they were not accidental vulnerabilities, but had been added by an outside threat actor that had compromised and tampered with Juniper's codebase beginning in 2012. One of the two vulnerabilities, CVE-2015-7756, was described as potentially "allowing a knowledgeable attacker who can monitor VPN traffic to decrypt that traffic." Although the vulnerabilities were quickly patched, disassembly by the public research community quickly revealed several surprising facts about both the original Netscreen implementation and the modified one [123, 48].

First, firmware analysis revealed that the original (unmodified) version of the Netscreen codebase contained an undocumented instance of the Dual EC generator, which had been present in the code since 2008.[24] This is quite surprising, since Juniper's CMVP certification documents make no mention of Dual EC, and instead claim that the generator uses only ANSI X9.31 for deterministic bit generation. This undocumented instance of Dual EC has two further unusual features: (1) it does not use the standard NIST Q point, instead using a new value Q' of unknown provenance, and (2) while all outputs from the Dual EC generator *appear* to be post-processed using ANSI X9.31 (which should make exploitation of a Dual EC backdoor infeasible) the code for the ANSI generator contains a bug (Figure 11.14) that completely disables this post-processing. The result is that in the default configuration, all Netscreen PRG outputs were produced using Dual EC. These findings have astounding implications: any attacker with knowledge of the discrete logarithm of Q' would have been able to efficiently decrypt Netscreen VPN traffic worldwide.[25]

These findings are so surprising on their own that the second portion of the story is only slightly remarkable. Sometime in 2012, unauthorized attackers gained access to the

[24] While the existence of Dual EC was not disclosed in FIPS documents, Juniper did quietly release a knowledge-based article in 2013, immediately following NIST's deprecation of Dual EC. At the time, the company acknowledged the existence of Dual EC in the code (and noted the use of non-standard curve points), but claimed that the output of Dual EC was post-processed by a second ANSI generator, thus removing any risk that the generator backdoor could be exploited.

[25] For a detailed analysis and proof-of-concept exploit, see Checkoway *et al.* [48].

DRAFT

```
1  void prng_reseed(void) {
2    blocks_generated_since_reseed = 0;
3    if (dualec_generate(prng_temporary, 32) != 32)
4      error_handler("FIPS ERROR: PRNG failure, unable to reseed\n", 11);
5    memcpy(prng_seed, prng_temporary, 8);
6    prng_output_index = 8;
7    memcpy(prng_key, &prng_temporary[prng_output_index], 24);
8    prng_output_index = 32;
9  }
10 void prng_generate(void) {
11   int time[2];
12
13   time[0] = 0;
14   time[1] = get_cycles();
15   prng_output_index = 0;
16   ++blocks_generated_since_reseed;
17   if (!one_stage_rng())
18     prng_reseed();
19   for (; prng_output_index <= 0x1F; prng_output_index += 8) {
20     // FIPS checks removed for clarity
21     x9_31_generate_block(time, prng_seed, prng_key, prng_block);
22     // FIPS checks removed for clarity
23     memcpy(&prng_temporary[prng_output_index], prng_block, 8);
24   }
25 }
```

Figure 11.14: Decompiled segment of code from the Juniper Netscreen 6.2 RNG. The loop in lines 19–24 implements the ANSI X9.31 PRG, but this loop never runs due to the fact that the loop variable prng_output_index has been set to 32 during a subroutine call at line 8. Source: [48].

DRAFT

Netscreen ScreenOS codebase and replaced the original value Q' with a new value Q'' of their own devising. If we presume that the attackers knew the discrete logarithm of this new value, then this attack essentially "re-keyed" an existing backdoor to create one that the attackers could exploit for themselves. The vulnerability remained undiscovered within Netscreen routers until its discovery in late 2015. To this day we do not know why the attackers did this, and whether they used their capability to attacks some other target. The FBI investigation remains open as of this writing, and no further details have been released.

11.3.2 ANSI X9.31 PRG

Juniper Netscreen hardware is not the only mass-market VPN device that has been known to contain an exploitable RNG backdoor. In 2017 a related flaw was discovered in Fortinet devices.

Prior to the release of NIST SP 800-90A, many FIPS-certified devices made use of a now-deprecated pseudorandom generator, which was standardized by the ANSI X9.31 committee in 1998. The ANSI X9.31 generator pre-dates the development of NIST SP 800-90A [22], and contains a much more vulnerable design. The ANSI design makes use of a block cipher, typically either 3DES (in earlier implementations) or AES. Figure 11.15 shows one round of the generation process. There are two "seeds" in this design: a fixed key K that does not change between rounds, and a "state" (seed) V that is initially set to a random value, but is updated within each round of generation. The generator also takes as input a unique "timestamp" that can be drawn from a system clock, or can simply be an ascending counter that does not repeat.

For nearly as long as ANSI X9.31 has existed, researchers have known that it was vulnerable to a specific attack: if an attacker can learn the value K and guess the timestamp T_i, then the generator's internal state can be recovered from seeing a single block of output R_i [103]. With further ability to guess timestamp values, the attacker can then wind the generator both forwards *and* backwards to recover future and past generator outputs!

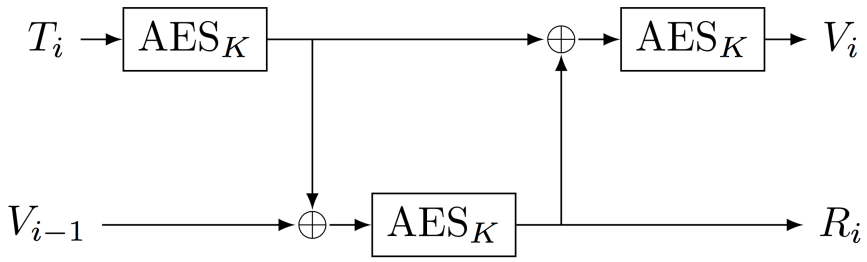


Figure 11.15: PRG generation loop for the ANSI X9.31 generator, here implemented using AES. T_i is a "timestamp" value (in practice, this can simply be an incrementing counter). V_{i-1} and V_i represent the seed/state, which must be initialized to a random value at V_0 and will update at each iteration. K is a fixed AES key that does not change or update. R_i represents a block of random output. All values are strings of the form $\{0,1\}^b$ where b is the cipher's block size (128 bits in this example.) Source: [52].

DRAFT

Of course, this requires an apparently strong assumption: that the key K is known to the attacker. Unfortunately, during the time ANSI X9.31 was allowed in devices, the U.S. FIPS standards allowed many vendors to use a *fixed and hard-coded* value of K in devices, which meant that K could potentially be recovered through simple reverse-engineering. Even worse, a large class of Fortinet firewall/VPN devices using software released from 2009–2014 included a hard-coded key drawn from public FIPS test vectors, which made tens of thousands of devices vulnerable to state-recovery attacks and TLS/IPsec decryption [52].

It is quite remarkable to note that two major U.S. VPN device vendors, Juniper/Netscreen and Fortinet, both contained critical RNG vulnerabilities during the same time period, both of which enabled efficient decryption of VPN connections. One might even call it bad luck.

The Debian PRG

A challenge with RNG implementations is that it is relatively easy to break them in ways that are hard to detect. Back in 2008, a version of OpenSSL that shipped with the Debian distribution of Linux was found to contain a subtle flaw that resulted in the generation of thousands of low-entropy (but apparently valid) TLS/SSL keys [154]. The accidental flaw (CVE-2008-0166) was ironically the result of an attempt to improve the security of OpenSSL's code.

The issue in this version of OpenSSL was triggered by the fact that the library's internal PRG used a large quantity of seed material to initialize the PRG, some of which included

sections of *uninitialized memory*. This was not by itself a security vulnerability, since the remainder of the seed material was high-entropy and sufficient to properly seed the generator. (Indeed, the use of uninitialized memory was believed by the OpenSSL developers to be a harmless, even possibly helpful element, since this memory might contain additional entropy beyond the normal seed material.) Unfortunately, during a security analysis, a developer using the Valgrind memory-safety analysis tool detected the use of uninitialized memory, and flagged it as a potential security vulnerability [128]. The "fix" was to disable specific lines of code that were, unfortunately, essential to ensuring that the remainder of the seed material was properly hashed into the OpenSSL PRG seed.

The consequence was that the only seed material used in this OpenSSL version was the *process ID* (pid). Since at the time there were only 32,768 possible values for this variable, the result was that the OpenSSL seed could be easily brute-forced. Unfortunately, since the PRG itself is cryptographic, the resulting output bits appeared random to simple manual inspection. Even worse: OpenSSL re-seeded the PRG immediately before generating TLS/SSL keypairs, which meant that there were at most 32,768 possible keypairs that could be produced. The flaw was detected when multiple devices were found to be using identical TLS/SSL keys, a condition that should never have occurred in practice. Unfortunately, by this point the affected devices were widely used across the Internet, and all generated keys had to be revoked and re-generated.

DRAFT

Factorable RSA keys

A related flaw was detected in 2012 by researchers who conducted Internet-wide scans of publicly-available SSL/TLS and IPSec public keys [87, 109]. These researchers discovered that many RSA public moduli shared common factors. Recall that an RSA public modulus has the form $N = p \cdot q$, where p, q are supposed to be randomly-generated prime numbers. Since these primes are usually large (e.g., 512 bits or greater), the probability of two distinct moduli N_1, N_2 sharing a single prime factor should be negligible. Unfortunately, these scans revealed that thousands of deployed machines used either identical values of N, or else shared a single factor (e.g., p.) The latter vulnerability renders both RSA keys completely insecure, since an attacker can easily compute a GCD of the two moduli to obtain p, then divide to obtain the remaining factors (see §**??**.)

Detailed investigation revealed that the common element in many cases was the use of improperly-seeded random generators, mainly those used in embedded devices that generate keys immediately following device boot [87]. A common pattern was as follows: a device, recently booted and containing little-to-no entropy in the kernel entropy pool, would use a non-blocking RNG such as `/dev/urandom` to generate a first factor p.[26] Subsequently, a small amount of additional entropy would be collected, and the device would generate the second factor q. The result was that many different devices would share p while using different values of q. In most cases this flaw occurred on "headless" embedded devices that

[26] In practice, the more common pattern was that a cryptographic library such as OpenSSL would seed its own PRG from the OS RNG.

had limited access to entropy sources. We hope that embedded devices have improved as a result of this example, but they probably haven't.

11.4 Other randomness sources

To conclude this chapter, it is worth noting that there exist other sources of randomness on the Internet, and these are sometimes used in real applications.

Randomness beacons

Several parties have proposed to construct public *randomness beacons* that provide unpredictable but non-secret outputs. While these services must not be used for generating key material, they can be useful for applications such as conducting verifiable low-value lotteries and running consensus protocols. NIST operates an experimental public beacon service that periodically generates random strings that can be used for various purposes [132, 23]. The danger in using these services is that the users must implicitly trust the beacon operator to behave honestly (or not to be hacked), which rules out their use in many high-value applications.

Distributed randomness generation

A number of decentralized protocols require access to high-quality *non-secret* randomness for their operation. Common examples include distributed consensus protocols that perform operations such as randomized committee election. In these systems, a large number of nodes cooperate to perform some process, such as determining which subset of the nodes will perform essential operations during some time period. This can typically done using some form of "lottery" process, but it requires that the group must jointly produce some unpredictable random numbers (often, simply a "seed" that can be stretched into a longer string of values using a PRG.)

A typical approach is for the various members of the group to each contribute some randomness, which is then combined together to produce a single joint random value. The challenge in this process is that the *last* member to contribute randomness may be able to see the other members' contributions and adjust its own contribution to influence the final result. For example, consider a simple protocol where each of N members outputs a random string, and the resulting seed is obtained by combining the strings together using exclusive-OR: here, the final member who sees the other members' strings can effectively choose the final result! This can be somewhat thwarted if the resulting string is combined by hashing the members' contributions together using a cryptographic hash function: but even in this case, the final mover can still test and discard many possible strings in order to "bias" the resulting hash towards a value it chooses.[27]

[27]For example, if the attacker wishes to force the low-order bits of the hash function to a specific value, it can simply choose many possible contribution values and internally compute the expected hash of its value

DRAFT

One approach to solving this problem is to use a distributed *coin-flipping protocol*. A common approach in these protocols is for each player to perform the process in two rounds. In the first round, each player *commits* to its input by sending a digital commitment (e.g., a salted cryptographic hash) to its input. In the second round, each player "opens" the commitment by sending its actual input value as well as any randomness (salt) used in the commitment. The inputs are then combined together. This approach nominally prevents the later players from changing their inputs after the other players have revealed their own values. Unfortunately this simply raises new problems: some users might selectively withhold their inputs when they detect that the outcome will not be as desired. A great deal of research has been done on ways to address this problem, but the results are beyond the scope of this book.

Astronomical noise

A fun (if slightly goofy) proposal for shared randomness generation is to use *astronomical noise*, such as the radio signals emitted by pulsars (see [57] for one, but not the first such proposal.) Since these signals appear to be somewhat unpredictable, and can also be received at multiple points throughout the globe, researchers have occasionally proposed to use them as a form of nature-made randomness beacon. There are numerous challenges with this idea, not the least of which is the need to obtain access to a relatively large radio telescope in order to access the signal. A further downside of this approach is the need to consider extra-terrestrial interference within your threat model.

DRAFT

combined with the other members' values, discarding any values that do not produce the desired result.