2

RANDOMNESS

Randomness is found everywhere in cryptography: in the generation of secret keys, in encryption schemes, and even in the attacks on cryptosystems. Without randomness, cryptography would be impossible because all operations would become predictable and therefore insecure.

This chapter introduces the concept of randomness in the context of cryptography and its applications. We discuss pseudorandom number generators and how operating systems can produce reliable randomness, and we conclude with real examples showing how flawed randomness can impact security.

Random or Nonrandom?

You've probably heard the phrase *random bits* before, but strictly speaking, there is no such thing as a series of random bits. What *is* random is the

algorithm, or process, that produces a series of random bits; therefore, when we say "random bits," we actually mean randomly generated bits.

What do random bits look like? For example, the 8-bit string 11010110 might look more random than 00000000, although both have the same chance of being generated (namely, 1/256). The value 11010110 looks more random than 00000000 because it has the signs typical of a randomly generated value. That is, 11010110 has no obvious pattern.

When we see the string 11010110, our brain registers that it has 3 zeros and 5 ones, just like 55 other 8-bit strings (11111000, 11110100, 11110010, and so on), but only one 8-bit string has 8 zeros. Because the pattern 3-zeros-and-5-ones is more likely to occur than the pattern 8-zeros, we identify 11010110 as random and 00000000 as nonrandom, even if they're not.

This example illustrates two types of errors people often make when identifying randomness:

Mistaking nonrandomness for randomness Thinking that an object was randomly generated simply because it *looks* random

Mistaking randomness for nonrandomness Thinking that patterns appearing by chance are there for a reason other than chance

The distinction between random-looking and actually random is crucial. Indeed, in crypto, nonrandomness is often synonymous with insecurity.

The saying "it happened by chance" reflects the property that from a complex system (in this case, our universe that obeys the laws of physics, deterministic at the macroscopic level and truly random at the subatomic, quantum level) can emerge specific patterns, such as the string 00000000. By the law of large numbers, if many events occur, some won't look random—such as a series of sequential numbers in a lottery draw. Many pseudosciences and belief systems are in fact cases of mistaking randomness for nonrandomness.

Randomness as a Probability Distribution

Any randomized process is characterized by a *probability distribution*, which gives all there is to know about the randomness of the process. A probability distribution, or simply *distribution*, lists the outcomes of a randomized process where each outcome is assigned a probability.

A *probability* measures the likelihood of an event occurring. It's expressed as a real number between 0 and 1 where a probability of 0 means impossible and a probability of 1 means certain. For example, when tossing a two-sided coin, each side has a 1/2 (or 0.5) probability of landing face up, and the probability of a coin landing on its edge has a probability close to 0.

A probability distribution must include all possible outcomes such that the sum of all probabilities is 1. Specifically, if there are N possible events, there are N probabilities $p_1, p_2, \ldots, p_N$ with $p_1 + p_2 + \ldots + p_N = 1$. In the case of the coin toss, the distribution is 1/2 for heads and 1/2 for tails. The sum

of both probabilities is equal to 1/2 + 1/2 = 1, because the coin will fall on one of its two faces.

A *uniform distribution* occurs when all probabilities in the distribution are equal, meaning that all outcomes are equally likely to occur. If there are N events, then each event has probability $1/N$. For example, if a 128-bit key is picked uniformly at random—that is, according to a uniform distribution—then each of the 2^{128} possible keys should have a probability of $1/2^{128}$.

In contrast, when a distribution is *nonuniform*, probabilities aren't all equal. A coin toss with a nonuniform distribution is said to be biased and may yield heads with probability 1/4 and tails with probability 3/4, for example.

NOTE *It's possible to cheat with a loaded die, preventing the probabilities of each of the six faces to be 1/6; however, one can't bias a coin. Coin tosses can be biased only if "the coin is allowed to bounce or be spun rather than simply flipped in the air," as described in the article "You Can Load a Die but You Can't Bias a Coin" (available at* https://www.stat.berkeley.edu/~nolan/Papers/dice.pdf*).*

Entropy: A Measure of Uncertainty

Entropy is the measure of uncertainty, or disorder, in a system. The higher the entropy, the less certainty found in the result of a randomized process.

We can compute the entropy of a probability distribution. If your distribution consists of probabilities p_1, p_2, . . . , p_N, then its entropy is the negative sum of all probabilities multiplied by their logarithm, as shown in this expression:

$$-p_1 \times \log(p_1) - p_2 \times \log(p_2) - \ldots\ p_N \times \log(p_N)$$

Here the function *log* is the *binary logarithm*, or logarithm in base two. Unlike the natural logarithm, the binary logarithm expresses the information in bits and yields integer values when probabilities are powers of two. For example, $\log(1/2) = -1$, $\log(1/4) = -2$, and more generally $\log(1/2^n) = -n$. (We actually take the *negative sum* to end up with a positive number.) Random 128-bit keys produced using a uniform distribution therefore have the following entropy:

$$2^{128} \times \left(-2^{-128} \times \log\left(2^{-128}\right)\right) = -\log\left(2^{-128}\right) = 128 \text{ bits}$$

If you replace 128 with any integer n, the entropy of a uniformly distributed n-bit string will be n bits.

Entropy is maximized when the distribution is uniform because a uniform distribution maximizes uncertainty: no outcome is more likely than the others. Therefore, n-bit values can't have more than n bits of entropy.

By the same token, when the distribution is not uniform, entropy is lower. Consider the coin toss example. The entropy of a fair toss is the following:

$$-(1/2) \times \log(1/2) - (1/2) \times \log(1/2) = 1/2 + 1/2 = 1 \text{ bit}$$

What if one side of the coin has a higher probability of landing face up than the other? Say heads has a probability of 1/4 and tails 3/4. (Remember that the sum of all probabilities should be 1.)

The entropy of such a biased toss is this:

$$-(3/4) \times \log(3/4) - (1/4) \times \log(1/4) \approx -(3/4) \times (-0.415) - (1/4) \times (-2) \approx 0.81 \text{ bits}$$

The fact that 0.81 is less than the 1-bit entropy of a fair toss tells us that the more biased the coin, the less uniform the distribution and the lower the entropy. Taking this example further, if heads has a probability of 1/10, the entropy is 0.469; if the probability drops to 1/100, the entropy drops to 0.081.

NOTE *Entropy can also be viewed as a measure of information. For example, the result of a fair coin toss gives you exactly 1 bit of information—heads or tails—and you're unable to predict the result of the toss in advance. In the case of the unfair coin toss, you know in advance that tails is more probable, so you can predict the outcome. The result of the unfair coin toss gives you the information needed to predict the result with certainty.*

Random and Pseudorandom Number Generators

Cryptosystems need randomness to be secure and therefore need a component from which to get their randomness. The job of this component is to return random bits when requested to do so. To perform this randomness generation, you'll need two things:

- A source of entropy, provided by random number generators.
- A cryptographic algorithm to produce high-quality random bits from the source of entropy. This is found in pseudorandom number generators.

Using both random and pseudorandom number generators is the key to making cryptography practical and secure. Let's briefly look at how random number generators work before exploring pseudorandom number generators in depth.

Randomness comes from the environment, which is analog, chaotic, uncertain, and hence unpredictable. Randomness can't be generated by computer-based algorithms alone. In cryptography, randomness usually comes from *random number generators (RNGs)*, which are software or hardware components that leverage entropy in the analog world to produce unpredictable bits in a digital system. For example, an RNG might directly

sample bits from measurements of temperature, acoustic noise, air turbulence, or electrical static. Unfortunately, such analog entropy sources aren't always available, and their entropy is often difficult to estimate.

RNGs can also harvest the entropy in a running operating system by drawing from attached sensors, I/O devices, network or disk activity, system logs, running processes, and user activities such as key presses and mouse movement. Such system- and human-generated activities can be a good source of entropy, but they can be fragile and manipulated by an attacker. Also, they're slow to yield random bits.

NOTE Quantum random number generators (QRNGs) *are a type of RNG that rely on the randomness arising from quantum mechanical phenomena, such as radioactive decay, photon polarization, or thermal noise. These phenomena, not being characterized by equations that determine the future state from the current state, are random in the absolute sense. In practice, however, the raw bits extracted from a QRNG may be biased and tend to be slow to produce. Like the previously cited entropy sources, they require postprocessing to generate reliable bits at high speed.*

Pseudorandom number generators (PRNGs) address the challenge in generating randomness by reliably producing many artificial random bits from a few true random bits. For example, an RNG that translates mouse movements to random bits would stop working if you stop moving the mouse, whereas a PRNG always returns pseudorandom bits when requested to do so.

PRNGs rely on RNGs but behave differently: RNGs produce true random bits relatively slowly from analog sources, in a nondeterministic way, and with no guarantee of uniform distribution or of high entropy per bit. In contrast, PRNGs produce random-looking bits quickly from digital sources, in a deterministic way, uniformly distributed, and with an entropy guaranteed to be high enough for cryptographic applications. Essentially, PRNGs transform a few unreliable random bits into a long stream of reliable pseudorandom bits suitable for crypto applications, as Figure 2-1 shows.

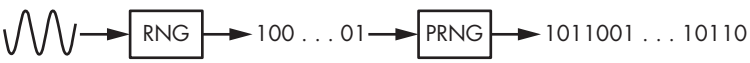

Figure 2-1: RNGs produce few unreliable bits from analog sources, whereas PRNGs expand those bits to a long stream of reliable bits.

How PRNGs Work

A PRNG receives random bits from an RNG at regular intervals and uses them to update the contents of a large memory buffer, called the *entropy pool*. The entropy pool is the PRNG's source of entropy, just like the physical environment is to an RNG. When the PRNG updates the entropy pool, it mixes the pool's bits together to help remove any statistical bias.

To generate pseudorandom bits, the PRNG runs a deterministic random bit generator (DRBG) algorithm that expands some bits from the entropy pool into a much longer sequence. As its name suggests, a DRBG is deterministic, not randomized: given one input, you will always get the same output. The PRNG ensures that its DRBG never receives the same input twice so it can generate unique pseudorandom sequences.

In the course of its work, the PRNG performs three operations:

***init*()** Initializes the entropy pool and the internal state of the PRNG

***refresh*(*R*)** Updates the entropy pool using some data, *R*, usually sourced from an RNG

***next*(*N*)** Returns *N* pseudorandom bits and updates the entropy pool

The *init* operation resets the PRNG to a fresh state, reinitializes the entropy pool to some default value, and initializes any variables or memory buffers used by the PRNG to carry out the *refresh* and *next* operations.

The *refresh* operation is often called *reseeding*, and its argument *R* is called a *seed*. When no RNG is available, seeds may be unique values hard-coded in a system. The *refresh* operation is typically called by the operating system, whereas *next* is typically called or requested by applications. The *next* operation runs the DRBG and modifies the entropy pool to ensure that the next call will yield different pseudorandom bits.

Security Concerns

Let's talk briefly about how PRNGs address high-level security concerns. Specifically, PRNGs should guarantee *backtracking resistance* and *prediction resistance*. Backtracking resistance (also called *forward secrecy*) means that previously generated bits are impossible to recover, whereas prediction resistance (*backward secrecy*) means that future bits should be impossible to predict.

To achieve backtracking resistance, the PRNG should ensure that the transformations performed when updating the state through the *refresh* and *next* operations are irreversible. This way, if an attacker compromises the system and obtains the entropy pool's value, they can't determine the previous values of the pool or the previously generated bits. To achieve prediction resistance, the PRNG should call *refresh* regularly with *R* values that are unknown to an attacker and are difficult to guess, thus preventing an attacker from determining future values of the entropy pool, even if the whole pool is compromised. (If the list of *R* values were known, you'd need to know the order in which *refresh* and *next* calls were made to reconstruct the pool.)

The PRNG Fortuna

Fortuna is a PRNG construction used in Windows originally designed in 2003 by Niels Ferguson and Bruce Schneier. Fortuna superseded *Yarrow*, a 1998 design by John Kelsey and Bruce Schneier that was for a long time used in the macOS and iOS operating systems and has been replaced by

Fortuna. I won't provide the Fortuna specification here or show you how to implement it, but I will try to explain how it works. You'll find a complete description of Fortuna in Chapter 9 of *Cryptography Engineering* by Ferguson, Schneier, and Kohno (Wiley, 2010).

Fortuna's internal memory includes the following:

- Thirty-two entropy pools, P_1, P_2, . . . , P_{32}, such that P_i is used every 2^i reseeds.
- A key, K, and a counter, C (both 16 bytes). These form the internal state of Fortuna's DRBG.

In simplest terms, Fortuna works like this:

- *init*() sets K and C to zero and empties the 32 entropy pools P_i, where $i = 1 \ldots 32$.
- *refresh*(R) appends the data, R, to one of the entropy pools. The system chooses the RNGs used to produce R values, and it should call *refresh* regularly.
- *next*(N) updates K using data from one or more entropy pools, where the choice of the entropy pools depends mainly on how many updates of K have already been done. The N bits requested are then produced by encrypting C using K as a key. If encrypting C is not enough, Fortuna encrypts $C + 1$, then $C + 2$, and so on, to get enough bits.

Although Fortuna's operations look fairly simple, implementing them correctly is hard. For one, you need to get all the details of the algorithm right—how entropy pools are chosen, the type of cipher to be used in *next*, how to behave when no entropy is received, and so on. Although the specs define most of the details, they don't include a comprehensive test suite to check that an implementation is correct, which makes it difficult to ensure that your implementation of Fortuna will behave as expected.

Even if Fortuna is correctly implemented, security failures may occur for reasons other than the use of an incorrect algorithm. For example, Fortuna might not notice if the RNGs fail to produce enough random bits, and as a result Fortuna will produce lower-quality pseudorandom bits, or it may stop delivering pseudorandom bits altogether.

Another risk inherent in Fortuna implementations lies in the possibility of exposing associated *seed files* to attackers. The data in Fortuna seed files is used to feed entropy to Fortuna through *refresh* calls when an RNG is not immediately available—for example, immediately after a system reboot and before the system's RNGs have recorded any unpredictable events. However, if an identical seed file is used twice, Fortuna will produce the same bit sequence twice. Seed files should therefore be erased after use to ensure they aren't reused.

Finally, if two Fortuna instances are in the same state because they're sharing a seed file (meaning the same data in the entropy pools, including C and K), then the *next* operation will return the same bits in both instances.

Cryptographic vs. Noncryptographic PRNGs

There are cryptographic and noncryptographic PRNGs. Noncrypto PRNGs are designed to produce uniform distributions for applications such as scientific simulations or video games. However, you should never use noncrypto PRNGs in crypto applications, because they're insecure; they're concerned only with the quality of the bits' probability distribution and not with their predictability. Crypto PRNGs, on the other hand, are unpredictable because they're also concerned with the strength of the underlying *operations* used to deliver well-distributed bits.

Unfortunately, most PRNGs exposed by programming languages—such as libc's `rand` and `drand48`, PHP's `rand` and `mt_rand`, Python's `random` module, and Java's `java.util.Random` class—are noncryptographic. Defaulting to a noncrypto PRNG is a recipe for disaster because it often ends up being used in crypto applications, so be sure to use only crypto PRNGs when generating randomness related to cryptographic or security applications.

A Popular Noncrypto PRNG: Mersenne Twister

The *Mersenne Twister (MT)* algorithm is a noncryptographic PRNG used (at the time of this writing) in PHP, Python, R, Ruby, and many other systems. It's even been used (unfortunately) in blockchain wallet key generators. MT generates uniformly distributed random bits without statistical bias, but it's predictable: given a few bits produced by MT, one can guess which bits will follow.

Let's look under the hood to see what makes the Mersenne Twister insecure. The MT algorithm is much simpler than that of crypto PRNGs: its internal state is an array, S, consisting of 624 32-bit words. This array is initially set to $S_1, S_2, \ldots, S_{624}$ and evolves to $S_2, \ldots, S_{625}$, then $S_3, \ldots, S_{626}$, and so on, according to this equation:

$$S_{k+624} = S_{k+397} \oplus \mathbf{A}\left(\left(S_k \wedge \text{0x80000000}\right) \vee \left(S_{k+1} \wedge \text{0x7fffffff}\right)\right)$$

Here, $\oplus$ denotes the bitwise XOR (^ in the C programming language), $\wedge$ denotes the bitwise AND (& in C), $\vee$ denotes the bitwise OR (| in C), and $\mathbf{A}$ is a function that transforms some 32-bit word, x, to $(x \gg 1)$ if x's most significant bit is 0, or to $(x \gg 1) \oplus$ 0x9908b0df otherwise.

In this equation, bits of S interact with each other only through XORs. The operators $\wedge$ and $\vee$ never combine 2 bits of S together but instead combine bits of S with bits from the constants 0x80000000 and 0x7fffffff. This way, any bit from S_{625} can be expressed as an XOR of bits from S_{398}, S_1, and S_2, and any bit from any future state can be expressed as an XOR combination of bits from the initial state $S_1, \ldots, S_{624}$. (When you express, say, $S_{228+624} = S_{852}$ as a function of S_{625}, S_{228}, and S_{229}, you can in turn replace S_{625} by its expression in terms of S_{398}, S_1, and S_2.)

Because there are exactly $624 \times 32 = 19{,}968$ bits in the initial state (or 624 32-bit words), any output bit can be expressed as an equation with at most 19,969 terms (19,968 bits plus one constant bit). That's about 2.5KB of

data. The converse is also true: bits from the initial state can be expressed as an XOR of output bits.

Linearity Insecurity

We call an XOR combination of bits a *linear combination*. For example, if X, Y, and Z are bits, then the expression $X \oplus Y \oplus Z$ is a linear combination, whereas $(X \wedge Y) \oplus Z$ is not because there's an AND ($\wedge$). If you flip a bit of X in $X \oplus Y \oplus Z$, then the result changes as well, regardless of Y's and Z's values. In contrast, if you flip a bit of X in $(X \wedge Y) \oplus Z$, the result changes only if Y's bit at the same position is 1. The upshot is that linear combinations are predictable because you don't need to know the value of the bits in order to predict how a change in their value will affect the result.

For comparison, if the MT algorithm were cryptographically strong, its equations would be *nonlinear* and would involve not only single bits but also AND combinations (*products*) of bits, such as $S_1S_{15}S_{182}$ or $S_{17}S_{256}S_{257}S_{354}S_{498}S_{601}$. Although linear combinations of those bits include at most 624 variables, nonlinear combinations allow for up to 2^{624} variables. It would be impossible to solve, let alone write down, the whole of these equations. (Note that 2^{305}, a much smaller number, is the estimated information capacity of the observable universe.)

The key here is that linear transformations lead to short equations (comparable in size to the number of variables), which are easy to solve, whereas nonlinear transformations give rise to equations of exponential size, which are practically unsolvable. The game of cryptographers is thus to design PRNG algorithms that emulate such complex nonlinear transformations using only a small number of simple operations.

NOTE *Linearity is just one of many security criteria. Although necessary, nonlinearity alone does not make a PRNG cryptographically secure.*

The Uselessness of Statistical Tests

Statistical test suites like TestU01, Diehard, or the National Institute of Standards and Technology (NIST) test suite are one way to test the quality of pseudorandom bits. These tests take a sample of pseudorandom bits produced by a PRNG (say, 1MB's worth), compute some statistics on the distribution of certain patterns in the bits, and compare the results with the typical results obtained for a uniform distribution. For example, some tests count the number of 1 bits versus the number of 0 bits, or the distribution of 8-bit patterns. But statistical tests are largely irrelevant to cryptographic security, and it's possible to design a cryptographically weak PRNG that fools any statistical test.

When you run statistical tests on randomly generated data, you will usually see a bunch of statistical indicators as a result. These are typically p-values, a common statistical indicator. These results aren't always easy to interpret because they're rarely as simple as passed or failed. If your first results seem abnormal, don't worry: they may be the result of some

accidental deviation, or you may be testing too few samples. To ensure that the results you see are normal, compare them with those obtained for some reliable sample of identical size—for example, one generated with the OpenSSL toolkit using the following command:

```
$ openssl rand <number of bytes> -out <output file>
```

Real-World PRNGs

Let's turn our attention to implementing PRNGs in the real world. You'll find crypto PRNGs in the operating systems (OSs) of most platforms, from desktops and laptops to embedded systems such as routers and set-top boxes, as well as virtual machines, mobile phones, and so on. Most of these PRNGs are software based, but those that are pure hardware are used by applications running on the OS and sometimes by other PRNGs running on top of cryptographic libraries or applications.

Next, we'll look at the most widely deployed PRNGs: for Linux, Android, and many other Unix-based systems; in Windows; and in recent Intel microprocessors, whose PRNG is hardware based.

Random Bits in Linux

The device file */dev/urandom* is the userland interface to the crypto PRNG in operating systems based on the Linux kernel. You'll typically use it to generate reliable random bits. Because it's a device file, you request random bits from */dev/urandom* by reading it as a file. For example, the following command uses */dev/urandom* to write 10MB of random bits to a file:

```
$ dd if=/dev/urandom of=<output file> bs=1M count=10
```

The Wrong Way to Use /dev/urandom

You could write a naive and insecure C program like the one shown in Listing 2-1 to read random bits and hope for the best, but that would be a bad idea.

```
int random_bytes_insecure(void *buf, size_t len)
{
    int fd = open("/dev/urandom", O_RDONLY);
    read(fd, buf, len);
    close(fd);
    return 0;
}
```

Listing 2-1: An insecure use of /dev/urandom

This code is insecure; it doesn't even check the return values of `open()` and `read()`, which means your expected random buffer could end up filled with zeros or left unchanged.

A Safer Way to Use /dev/urandom

Listing 2-2, copied from the LibreSSL library, shows a safer way to use */dev/urandom*.

```
int random_bytes_safer(void *buf, size_t len)
{
    struct stat st;
    size_t i;
    int fd, cnt, flags;
    int save_errno = errno;

start:
    flags = O_RDONLY;
#ifdef O_NOFOLLOW
    flags |= O_NOFOLLOW;
#endif
#ifdef O_CLOEXEC
    flags |= O_CLOEXEC;
#endif
  ❶ fd = open("/dev/urandom", flags, 0);
    if (fd == -1) {
        if (errno == EINTR)
            goto start;
        goto nodevrandom;
    }
#ifndef O_CLOEXEC
    fcntl(fd, F_SETFD, fcntl(fd, F_GETFD) | FD_CLOEXEC);
#endif

    /* Lightly verify that the device node looks sane. */
    if (fstat(fd, &st) == -1 || !S_ISCHR(st.st_mode)) {
        close(fd);
        goto nodevrandom;
    }
    if (ioctl(fd, RNDGETENTCNT, &cnt) == -1) {
        close(fd);
        goto nodevrandom;
    }
    for (i = 0; i < len; ) {
        size_t wanted = len - i;
      ❷ ssize_t ret = read(fd, (char *)buf + i, wanted);

        if (ret == -1) {
            if (errno == EAGAIN || errno == EINTR)
                continue;
            close(fd);
            goto nodevrandom;
        }
        i += ret;
    }
    close(fd);
    if (gotdata(buf, len) == 0) {
        errno = save_errno;
        return 0; /* Satisfied */
    }
```

```
nodevrandom:
    errno = EIO;
    return -1;
}
```

Listing 2-2: A safe use of /dev/urandom

Unlike Listing 2-1, Listing 2-2 makes several sanity checks. Compare, for example, the calls to `open()` ❶ and to `read()` ❷ with those in Listing 2-1: the safer code checks the return values of those functions and upon failure closes the file descriptor and returns –1.

Differences Between /dev/urandom and /dev/random, Before 2022

The Linux PRNG, defined in *drivers/char/random.c* in the Linux kernel, underwent major changes in 2022 (since kernel version 5.17).

First, the general structure of the PRNG, which is similar in the old and new versions, is based on a collection of entropy from various sources (including system activity, such as keyboard, mouse, and disk accesses), as well as from an entropy pool that can be seen as a large array, which is filled by hashing data collected from the entropy sources. Next, a DRBG is responsible for producing the pseudorandom data streams returned when */dev/random* or */dev/urandom* is read or when the `getrandom()` system call is made.

Historically, prior to kernel version 5.17, the Linux PRNG behaved as follows: unlike */dev/urandom*, the */dev/random* interface was *blocking*; if the kernel estimated that the PRNG had an insufficient level of entropy, then */dev/random* would stop returning bytes ("block") when it was read, until a sufficient level of entropy was estimated by the kernel. This was not a good idea. For one thing, entropy estimators are notoriously unreliable and can be fooled by attackers (which is one reason why Fortuna ditched Yarrow's entropy estimation). Furthermore, */dev/random* ran out of estimated entropy pretty quickly, which could produce a denial-of-service condition, slowing applications that were forced to wait for more entropy. The upshot is that in practice, */dev/random* was no better than */dev/urandom* and created more problems than it solved.

Differences Between /dev/urandom and /dev/random, Since 2022

In versions of the Linux kernel from 2022 (5.17 onward), several improvements have been incorporated. First, the SHA-1 hash function has been replaced by BLAKE2 when creating the contents of the pool. The biggest change is the modification of the relative behavior of */dev/random* and */dev/urandom*; it has even been proposed to eliminate their differences altogether. At the time of writing, on most platforms both interfaces will detect if there isn't enough entropy, but */dev/urandom* will resume producing pseudorandom bits if the kernel fails to collect enough entropy, whereas */dev/random* will block.

In addition, the kernel's entropy estimation logic has been greatly improved: instead of considering that entropy decreases when PRNG bits are read (a cryptographic nonsense), the kernel just looks for the point when enough uncertainty (that is, entropy) has been collected—for example, at system startup.

You can read the entropy value of a Linux system in the */proc/sys/kernel/random/entropy_avail* file. In older versions of the kernel, this value was a maximum of 4,096 bits and decreased with the generation of PRNG bits. In the new kernels, the value is capped at 256 bits and therefore no longer decreases.

The CryptGenRandom() Function in Windows

In Windows, the legacy userland interface to the system's PRNG is the `CryptGenRandom()` function from the Cryptography application programming interface (API). Recent Windows versions replace the `CryptGenRandom()` function with the `BcryptGenRandom()` function in the Cryptography API: Next Generation (CNG). The Windows PRNG takes entropy from the kernel mode driver *cng.sys* (formerly *ksecdd.sys*), whose entropy collector is loosely based on Fortuna. As is usually the case in Windows, the process is complicated.

Listing 2-3 shows a typical C++ invocation of `CryptGenRandom()` with the required checks.

```
int random_bytes(unsigned char *out, size_t outlen)
{
    static HCRYPTPROV handle = 0; /* Only freed when the program ends */
    if(!handle) {
        if(!CryptAcquireContext(&handle, 0, 0, PROV_RSA_FULL,
                                CRYPT_VERIFYCONTEXT | CRYPT_SILENT)) {
            return -1;
        }
    }
    while(outlen > 0) {
        const DWORD len = outlen > 1048576UL ? 1048576UL : outlen;
        if(!CryptGenRandom(handle, len, out)) {
            return -2;
        }
        out    += len;
        outlen -= len;
    }
    return 0;
}
```

Listing 2-3: Using the Windows `CryptGenRandom()` PRNG interface

Prior to calling the actual PRNG, you need to declare a *cryptographic service provider* (`HCRYPTPROV`) and then acquire a *cryptographic context* with `CryptAcquireContext()`, which increases the likelihood that things will go wrong. For instance, the final version of the TrueCrypt encryption software was found to call `CryptAcquireContext()` in a way that could silently fail, leading to suboptimal randomness without notifying the user. Fortunately, the

newer and simpler `BCryptGenRandom()` interface for Windows doesn't require the code to explicitly open a handle (or at least makes it much easier to use without a handle).

A Hardware-Based PRNG: Intel Secure Key

We've discussed only software PRNGs so far, so let's take a look at a hardware one. The *Intel Digital Random Number Generator*, or *Intel Secure Key*, is a hardware PRNG introduced in 2012 in Intel's Ivy Bridge microarchitecture. It's based on NIST's SP 800-90 guidelines with the Advanced Encryption Standard (AES) in CTR_DRBG mode. Intel's PRNG is accessed through the `RDRAND` assembly instruction, which offers an interface independent of the operating system and is in principle faster than software PRNGs.

Whereas software PRNGs try to collect entropy from unpredictable sources, Intel Secure Key has a single entropy source that provides a serial stream of entropy data as zeros and ones. In hardware engineering terms, this entropy source is a dual differential jamb latch with feedback—essentially, a small hardware circuit that jumps between two states (0 or 1) depending on thermal noise fluctuations, at a frequency of 3 GHz. This is usually pretty reliable.

The `RDRAND` assembly instruction takes as an argument a register of 16, 32, or 64 bits and then writes a random value. When invoked, `RDRAND` sets the carry flag to 1 if the data set in the destination register is a valid random value, and to 0 otherwise; be sure to check the `CF` flag if you write assembly code directly. Note that the C intrinsics available in common compilers don't check the `CF` flag but do return its value.

NOTE *Intel's PRNG framework provides an assembly instruction other than `RDRAND`: the `RDSEED` assembly instruction returns random bits directly from the entropy source, after some conditioning or cryptographic processing. It's intended to be able to seed other PRNGs.*

Intel Secure Key is only partially documented, but it's built on known standards and has been audited by the well-regarded company Cryptography Research (see its report titled "Analysis of Intel's Ivy Bridge Digital Random Number Generator"). Nonetheless, there have been some concerns about its security, especially following Edward Snowden's revelations about cryptographic backdoors: PRNGs are indeed the perfect target for sabotage. If you're concerned but still want to use `RDRAND` or `RDSEED`, mix them with other entropy sources. Doing so will prevent effective exploitation of a hypothetical backdoor in Intel Secure Key's hardware or in the associated microcode in all but the most far-fetched scenarios.

How Things Can Go Wrong

To conclude, I'll present a few examples of randomness failures. There are countless examples to choose from, but I've chosen four that are simple enough to understand and illustrate different problems.

Poor Entropy Sources

In 1996, the SSL implementation of the Netscape browser was computing 128-bit PRNG seeds according to the pseudocode shown in Listing 2-4, copied from Goldberg and Wagner's page at *https://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html*.

```
global variable seed;

RNG_CreateContext()
    (seconds, microseconds) = time of day; /* Time elapsed since 1970 */
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
  ❶ b = mklcpr(pid + seconds + (ppid << 12));
    seed = MD5(a, b); /* Derivation of a 128-bit value using the hash MD5 */

mklcpr(x) /* Not cryptographically significant; shown for completeness */
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);

MD5() /* A very good standard mixing function, source omitted */
```

Listing 2-4: Pseudocode of the Netscape browser's generation of 128-bit PRNG seeds

The problem here is that the PIDs and microseconds are guessable values. Assuming that you can guess the value of `seconds`, `microseconds` has only 10^6 possible values and thus an entropy of $\log(10^6)$, or about 20 bits. The process ID (PID) and parent process ID (PPID) are 15-bit values, so you'd expect 15 + 15 = 30 additional entropy bits. But looking at how `b` is computed ❶ shows that the overlap of 3 bits yields an entropy of about 15 + 12 = 27 bits, for a total entropy of only 47 bits, whereas a 128-bit seed should have 128 bits of entropy.

Insufficient Entropy at Boot Time

In 2012, researchers scanned the internet and harvested public keys from TLS certificates and SSH hosts. They found that a handful of systems had identical public keys, and in some cases very similar keys (namely, RSA keys with shared prime factors)—in short, two numbers, $n = pq$ and $n' = p'q'$, with $p = p'$, whereas normally all ps and qs should be different in distinct modulus values.

It turned out that many devices generated their public key early, at first boot, before having collected enough entropy, despite using an otherwise-decent PRNG (typically */dev/urandom*). PRNGs in different systems produced identical random bits due to having the same entropy source (for example, a hardcoded seed).

At a high level, the presence of identical keys is due to key-generation schemes like the following, in pseudocode:

```
prng.seed(seed)
p = prng.generate_random_prime()
q = prng.generate_random_prime()
n = p*q
```

If two systems run this code given an identical seed, they'll produce the same p, the same q, and therefore the same n.

The presence of shared primes in different keys is due to key-generation schemes where additional entropy is injected during the process, as demonstrated here:

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_entropy()
q = prng.generate_random_prime()
n = p*q
```

If two systems run this code with the same seed, they'll produce the same p, but the injection of entropy through `prng.add_entropy()` will ensure distinct qs.

The problem with shared prime factors is that given $n = pq$ and $n' = pq'$, it's trivial to recover the shared p by computing the greatest common divisor (GCD) of n and n'. For details, see the paper "Mining Your Ps and Qs" by Heninger, Durumeric, Wustrow, and Halderman, available at *https://factorable.net*.

Noncryptographic PRNG

Earlier we discussed the difference between crypto and noncrypto PRNGs and why the latter should never be used for crypto applications. Alas, many systems overlook that detail, so we'll look at one such example.

The popular MediaWiki application runs on Wikipedia and many other wikis. It uses randomness to generate things like security tokens and temporary passwords, which should be unpredictable. Unfortunately, a now obsolete version of MediaWiki used a noncrypto PRNG, the Mersenne Twister, to generate these tokens and passwords. Here's a snippet from the vulnerable MediaWiki source code; look for the function called to get a random bit, and read the comments:

```
        /**
         * Generate a hex-y looking random token for various uses.
         * Could be made more cryptographically sure if someone cares.
         * @return string
         */
function generateToken( $salt = '' ) {
    $token = dechex(mt_rand()).dechex(mt_rand());
    return md5( $token . $salt );
}
```

Did you notice `mt_rand()` in the preceding code? Here, `mt` stands for Mersenne Twister. In 2012, researchers showed how to exploit the predictability of Mersenne Twister to predict future tokens and temporary passwords, given a couple of security tokens. MediaWiki was patched to use a crypto PRNG.

Sampling Bug with Strong Randomness

The next bug shows how even a strong crypto PRNG with sufficient entropy can produce a biased distribution. The chat program Cryptocat was designed to offer secure communication. It used a function that attempted to create a uniformly distributed string of decimal digits—namely, numbers in the range 0 through 9. However, just taking random bytes modulo 10 doesn't yield a uniform distribution; when taking all numbers between 0 and 255 and reducing them modulo 10, you don't get an equal number of values in 0 to 9.

Cryptocat did the following to address that problem and obtain a uniform distribution:

```
Cryptocat.random = function() {
    var x, o = '';
    while (o.length < 16) {
        x = state.getBytes(1);
        if (x[0] <= 250) {
            o += x[0] % 10;
        }
    }
    return parseFloat('0.' + o)
}
```

And that was almost perfect. By taking only the numbers up to a multiple of 10 and discarding others, you'd expect a uniform distribution of the digits 0 through 9. Unfortunately, there was an off-by-one error in the `if` condition. I'll leave the details to you as an exercise. You should find that there is a small statistical bias in favor of the index 0 (hint: `<=` should have been `<`).

Further Reading

I've just scratched the surface of randomness in cryptography. There is much more to learn about the theory of randomness, including different entropy notions, randomness extractors, and even the power of randomization and derandomization in complexity theory. To learn more about PRNGs and their security, read the classic 1998 paper "Cryptanalytic Attacks on Pseudorandom Number Generators" by Kelsey, Schneier, Wagner, and Hall. Then look at the implementation of PRNGs in your favorite applications and try to find their weaknesses. (Search online for "random generator bug" to find plenty of examples.)

We're not done with randomness, though. We'll encounter it multiple times throughout this book, and you'll discover the many ways it helps to construct secure systems.