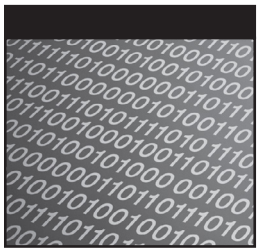

Cryptography Engineering

Design Principles and Practical Applications

Niels Ferguson
Bruce Schneier
Tadayoshi Kohno

WILEY
Wiley Publishing, Inc.

Cryptography Engineering: Design Principles and Practical Applications

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Published by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-47424-2

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2010920648

CHAPTER 9

Generating Randomness

To generate key material, we need a random number generator, or RNG. Generating good randomness is a vital part of many cryptographic operations. Generating good randomness is also very challenging.

We won't go into a detailed discussion of what randomness really is; an informal discussion suffices for our purposes. A good informal definition is that random data is unpredictable to the attacker, even if he is taking active steps to defeat our randomness.

Good random number generators are necessary for many cryptographic functions. Part II discussed the secure channel and its components. We assumed there to be a key known to both Alice and Bob. That key has to be generated somewhere. Key management systems use random number generators to choose keys. If you get the RNG wrong, you end up with a weak key. This is exactly what happened to one of the early versions of the Netscape browser [54].

The measure for randomness is called *entropy* [118]. Here's the high-level idea. If you have a 32-bit word that is completely random, it has 32 bits of entropy. If the 32-bit word takes on only four different values, and each value has a 25% chance of occurring, the word has 2 bits of entropy. Entropy does not measure how many bits are in a value, but how *uncertain* you are about the value. You can think of entropy as the average number of bits you would need to specify the value if you could use an ideal compression algorithm. Note that the entropy of a value depends on how much you know. A random 32-bit word has 32 bits of entropy. Now suppose you happen to know that the value has exactly 18 bits that are 0 and 14 bits that are 1. There are about $2^{28.8}$ values that satisfy these requirements, and the entropy is also limited to 28.8 bits. In other words, the more you know about a value, the smaller its entropy is.

It is a bit more complicated to compute the entropy for values that have a nonuniform probability distribution. The most common definition of entropy for a variable X is

$$H(X) := -\sum_x P(X = x) \log_2 P(X = x)$$

where $P(X = x)$ is the probability that the variable X takes on the value x. We won't use this formula, so you don't need to remember it. This definition is what most mathematicians refer to when they talk about entropy. There are a few other definitions of entropy that mathematicians use as well; which one they use depends on what they are working on. And don't confuse our entropy definition with the entropy that physicists talk about. They use the word for a concept from thermodynamics that is only tangentially related to our definition of entropy.

9.1 Real Random

In an ideal world we would use ''real random'' data. The world is not ideal, and real random data is extremely hard to find.

Typical computers have a number of sources of entropy. The exact timing of keystrokes and the exact movements of a mouse are well-known examples. There has even been research into using the random fluctuations in hard-disk access time caused by turbulence inside the enclosure [29]. All of these sources are somewhat suspect because there are situations in which the attacker can influence or perform measurements on the random source.

It is tempting to be optimistic about the amount of entropy that can be extracted from various sources. We've seen software that will generate 1 or 2 bytes of supposedly random data from the timing of a single keystroke. Cryptographers in general are far more pessimistic about the amount of entropy in a single keystroke. A good typist can keep the time between consecutive keystrokes predictable to within a dozen milliseconds. And the keyboard scan frequency limits the resolution with which keystroke timings can be measured. The data being typed is not very random either, even if you ask the user just to hit some keys to generate random data. Furthermore, there is always a risk that the attacker has additional information about the ''random'' events. A microphone can pick up the sounds of the keyboard, which helps to determine the timing of keystrokes. Be very careful in estimating how much entropy you think a particular piece of data contains. We are, after all, dealing with a very clever and active adversary.

There are many physical processes that behave randomly. For example, the laws of quantum physics force certain behavior to be perfectly random. It would be very nice if we could measure such random behavior and use it. Technically, this is certainly possible. However, the attacker has a few lines of attack on this type of solution. First of all, the attacker can try to influence the behavior of the quantum particles in question to make them behave predictably. The attacker can also try to eavesdrop on the measurements we make; if he gets a copy of our measurements, while the data might still be random, it won't have any entropy from the attacker's point of view. (If he knows the value, then it has no entropy for him.) Maybe the attacker can set up a strong RF field in an attempt to bias our detector. There are even some quantum physics–based attacks that can be contemplated. The Einstein-Podolsky-Rosen paradox could be used to subvert the randomness we are trying to measure [11, 19]. Similar comments apply to other sources of entropy, such as thermal noise of a resistor and tunneling and breakdown noise of a Zener diode.

Some modern computers have a built-in real random number generator [63]. This is a significant improvement over a separate real random generator, as it makes some of the attacks more difficult. The random number generator is still only accessible to the operating system, so an application has to trust the operating system to handle the random data in a secure manner.

9.1.1 Problems with Using Real Random Data

Aside from the difficulty of collecting real random data, there are several other problems with its practical use. First of all, it is not always available. If you have to wait for keystroke timings, then you cannot get any random data unless the user is typing. That can be a real problem when your application is a Web server on a machine with no keyboard connected to it. A related problem is that the amount of real random data is always limited. If you need a lot of random data, then you have to wait; something that is unacceptable for many applications.

A second problem is that real random sources, such as a physical random number generator, can break. Maybe the generator will become predictable in some way. Because real random generators are fairly intricate things in the very noisy environment of a computer, they are much more likely to break than the traditional parts of the computer. If you rely on the real random generator directly, then you're out of luck when it breaks. What's worse, you might not know when it breaks.

A third problem is judging how much entropy you can extract from any specific physical event. Unless you have specially designed dedicated hardware

for the random generator it is extremely difficult to know how much entropy you are getting. We'll discuss this in greater detail later.

9.1.2 Pseudorandom Data

An alternative to using real random data is to use pseudorandom data. Pseudorandom data is not really random at all. It is generated from a seed by a deterministic algorithm. If you know the seed, you can predict the pseudorandom data. Traditional pseudorandom number generators, or PRNGs, are not secure against a clever adversary. They are designed to eliminate statistical artifacts, not to withstand an intelligent attacker. The second volume of Knuth's *The Art of Computer Programming* contains an extensive discussion of random number generators, but all generators are analyzed for statistical randomness only [75]. We have to assume that our adversary knows the algorithm that is used to generate the random data. Given some of the pseudorandom outputs, is it possible for him to predict some future (or past) random bits? For many traditional PRNGs the answer might be yes. For a proper cryptographic PRNG the answer is no.

In the context of a cryptographic system, we have more stringent requirements. Even if the attacker sees much of the random data generated by the PRNG, she should not be able to predict anything about the rest of the output of the PRNG. We call such a PRNG cryptographically strong. As we have no need for a traditional PRNG, we will only talk about cryptographically strong PRNGs.

Forget about the normal random function in your programming library, because it is almost certainly not a cryptographic PRNG. Unless the cryptographic strength is explicitly documented, you should never use a library PRNG.

9.1.3 Real Random Data and PRNGs

We only use real random data for a single thing: to seed a PRNG. This construction resolves some of the problems of using real random data. Once the PRNG is seeded, random data is always available. You can keep adding the real random data that you receive to the PRNG seed, thereby ensuring that it never becomes fully predictable even if the seed becomes known.

There is a theoretical argument that real random data is better than pseudorandom data from a PRNG. In certain cryptographic protocols you can prove that certain attacks are impossible if you use real random data. The protocol is unconditionally secure. If you use a PRNG, the protocol is only secure as long as the attacker cannot break the PRNG; the protocol is computationally secure. This distinction, however, is only of theoretical interest. All

cryptographic protocols use computational assumptions for almost everything. Removing the computational assumption for one particular type of attack is an insignificant improvement, and generating real random data, which you need for the unconditional security, is so difficult that you are far more likely to reduce the system security by trying to use real random data. Any weakness in the real random generator immediately leads to a loss of security. However, if you use real random data to seed a PRNG, you can afford to be far more conservative in your assumptions about the entropy sources, which makes it much more likely that you will end up with a secure system in the end.

9.2 Attack Models for a PRNG

The task of generating pseudorandom numbers from a seed is fairly simple. The problem is how to get a random seed, and how to keep it secret in a real-world situation [71]. One of the best designs up to now that we know of is called Yarrow [69], a design we created a few years ago together with John Kelsey. Yarrow tries to prevent all the known attacks.

At any point in time the PRNG has an internal state. Requests for random data are honored by using a cryptographic algorithm to generate pseudorandom data. This algorithm also updates the internal state to ensure that the next request does not return the same random data. This process is easy; any hash function or block cipher can be used for this step.

There are various forms of attack on a PRNG. There is a straightforward attack where the attacker attempts to reconstruct the internal state from the output. This is a classical cryptographic attack, and rather easy to counter using cryptographic techniques.

Things become more difficult if the attacker is at some point able to acquire the internal state. For the purposes of this discussion, it is unimportant how that happens. Maybe there is a flaw in the implementation, or maybe the computer was just booted for the first time and has had no random seed yet, or maybe the attacker managed to read the seed file from disk. Bad things happen, and you have to be able to handle them. In a traditional PRNG, if the attacker acquires the internal state, she can follow all the outputs and all the updates of the internal state. This means that if the PRNG is ever attacked successfully, then it can never recover to a secure state.

Another problem arises if the same PRNG state is used more than once. This can happen when two or more virtual machines (VMs) are booted from the same state and read the same seed file from disk.

Recovering a PRNG whose state has been compromised is difficult, as is avoiding the re-use of the same state across VMs booted from the same

instance. We will need some source of entropy from a real random number generator. To keep this discussion simple, we will assume that we have one or more sources that provide some amount of entropy (typically in small chunks that we call events) at unpredictable times.

Even if we mix the small amounts of entropy from an event into the internal state, this still leaves an avenue of attack. The attacker simply makes frequent requests for random data from the PRNG. As long as the total amount of entropy added between two such requests is limited to, say, 30 bits, the attacker can simply try all possibilities for the random inputs and recover the new internal state after the mixing. This would require about 2^{30} tries, which is quite practical to do.[1] The random data generated by the PRNG provides the necessary verification when the attacker hits upon the right solution.

The best defense against this particular attack is to pool the incoming events that contain entropy. You collect entropy until you have enough to mix into the internal state without the attacker being able to guess the pooled data. How much is enough? Well, we want the attacker to spend at least 2^{128} steps on any attack, so you want to have 128 bits of entropy. But here is the real problem: making any kind of estimate of the amount of entropy is extremely difficult, if not impossible. It depends heavily on how much the attacker knows or can know, but that information is not available to the developers during the design phase. This is Yarrow's main problem. It tries to measure the entropy of a source using an entropy estimator, and such an estimator is impossible to get right for all situations.

9.3 Fortuna

In practice you are probably best off using a cryptographic PRNG provided by a well-accepted cryptographic library. For illustrative purposes, we focus now on the design of a PRNG we call Fortuna. Fortuna is an improvement on Yarrow and is named after the Roman goddess of chance.[2] Fortuna solves the problem of having to define entropy estimators by getting rid of them. The rest of this chapter is mostly about the details of Fortuna.

There are three parts to Fortuna. The generator takes a fixed-size seed and generates arbitrary amounts of pseudorandom data. The accumulator collects and pools entropy from various sources and occasionally reseeds the generator. Finally, the seed file control ensures that the PRNG can generate random data even when the computer has just booted.

[1] We are being sloppy with our math here. In this instance we should use guessing entropy, rather than the standard Shannon entropy. For extensive details on entropy measures, see [23].

[2] We thought about calling it Tyche, after the Greek goddess of chance, but nobody would know how to pronounce it.

9.4 The Generator

The generator is the part that converts a fixed-size state to arbitrarily long outputs. We'll use an AES-like block cipher for the generator; feel free to choose AES (Rijndael), Serpent, or Twofish for this function. The internal state of the generator consists of a 256-bit block cipher key and a 128-bit counter.

The generator is basically just a block cipher in counter mode. CTR mode generates a random stream of data, which will be our output. There are a few refinements.

If a user or application asks for random data, the generator runs its algorithm and generates pseudorandom data. Now suppose an attacker manages to compromise the generator's state after the completion of the request. It would be nice if this would not compromise the previous results the generator gave. Therefore, after every request we generate an extra 256 bits of pseudorandom data and use that as the new key for the block cipher. We can then forget the old key, thereby eliminating any possibility of leaking information about old requests.

To ensure that the data we generate will be statistically random, we cannot generate too much data at one time. After all, in purely random data there can be repeated block values, but the output of counter mode never contains repeated block values. (See Section 4.8.2 for details.) There are various solutions; we could use only half of each ciphertext block, which would hide most of the statistical deviation. We could use a different building block called a *pseudorandom function*, rather than a block cipher, but there are no well-analyzed and efficient proposals that we know of. The simplest solution is to limit the number of bytes of random data in a single request, which makes the statistical deviation much harder to detect.

If we were to generate 2^{64} blocks of output from a single key, we would expect close to one collision on the block values. A few repeated requests of this size would quickly show that the output is not perfectly random; it lacks the expected block collisions. We limit the maximum size of any one request to 2^{16} blocks (that is, 2^{20} bytes). For an ideal random generator, the probability of finding a block value collision in 2^{16} output blocks is about 2^{-97}, so the complete absence of collisions would not be detectable until about 2^{97} requests had been made. The total workload for the attacker ends up being 2^{113} steps. Not quite the 2^{128} steps that we're aiming for, but reasonably close.

We know we are being lax here and accepting a (slightly) reduced security level. There seems to be no good alternative. We don't have any suitable cryptographic building blocks that give us a PRNG with a full 128-bit security level. We could use SHA-256, but that would be much slower. We've found that people will argue endlessly not to use a good cryptographic PRNG, and

speed has always been one of the arguments. Slowing down the PRNG by a perceptible factor to get a few bits more security is counterproductive. Too many people will simply switch to a really bad PRNG, so the overall system security will drop.

If we had a block cipher with a 256-bit block size, then the collisions would not have been an issue at all. This particular attack is not such a great threat. Not only does the attacker have to perform 2^{113} steps, but the computer that is being attacked has to perform 2^{113} block cipher encryptions. So this attack depends on the speed of the user's computer, rather than on the speed of the attacker's computer. Most users don't add huge amounts of extra computing power just to help an attacker. We don't like these types of security arguments. They are more complicated, and if the PRNG is ever used in an unusual setting, this argument might no longer apply. Still, given the situation, our solution is the best compromise we can find.

When we rekey the block cipher at the end of each request, we do not reset the counter. This is a minor issue, but it avoids problems with short cycles. Suppose we were to reset the counter every time. If the key value ever repeats, and all requests are of a fixed size, then the next key value will also be a repeated key value. We could end up in a short cycle of key values. This is an unlikely situation, but by not resetting the counter we can avoid it entirely. As the counter is 128 bits, we will never repeat a counter value (2^{128} blocks is beyond the computational capabilities of our computers), and this automatically breaks any cycles. Furthermore, we use a counter value of 0 to indicate that the generator has not yet been keyed, and therefore cannot generate any output.

Note that the restriction that limits each request to at most 1 MB of data is not an inflexible restriction. If you need more than 1 MB of random data, just do repeated requests. In fact, the implementation could provide an interface that automatically performs such repeated requests.

The generator by itself is an extremely useful module. Implementations could make it available as part of the interface, not just as a component, of Fortuna. Take a program that performs a Monte Carlo simulation.[3] You really want the simulation to be random, but you also want to be able to repeat the exact same computation, if only for debugging and verification purposes. A good solution is to call the operating system's random generator once at the start of the program to get a random seed. This seed can be logged as part of the simulator output, and from this seed our generator can generate all the random data needed for the simulation. Knowing the original seed of the generator also allows all the computations to be verified by running the program again using the same input data and seed. And for debugging, the

[3] A Monte Carlo simulation is a simulation that is driven by random choices.

same simulation can be run again and again, and it will behave exactly the same every time, as long as the starting seed is kept constant.

We can now specify the operations of the generator in detail.

9.4.1 Initialization

This is rather simple. We set the key and the counter to zero to indicate that the generator has not been seeded yet.

function InitializeGenerator
output: $\mathcal{G}$ Generator state.

Set the key K and counter C to zero.
$(K, C) \leftarrow (0, 0)$
Package up the state.
$\mathcal{G} \leftarrow (K, C)$
return $\mathcal{G}$

9.4.2 Reseed

The reseed operation updates the state with an arbitrary input string. At this level we do not care what this input string contains. To ensure a thorough mixing of the input with the existing key, we use a hash function.

function Reseed
input: $\mathcal{G}$ Generator state; modified by this function.
s New or additional seed.

Compute the new key using a hash function.
$K \leftarrow \text{SHA}_d\text{-256}(K \parallel s)$
Increment the counter to make it nonzero and mark the generator as seeded. Throughout this generator, C is a 16-byte value treated as an integer using the LSByte first convention.
$C \leftarrow C + 1$

The counter C is used here as an integer. Later it will be used as a plaintext block. To convert between the two we use the least-significant-byte-first convention. The plaintext block is a block of 16 bytes $p_0, \ldots, p_{15}$ that corresponds to the integer value

$$\sum_{i=0}^{15} p_i 2^{8i}$$

By using this convention throughout, we can treat C both as a 16-byte string and as an integer.

9.4.3 Generate Blocks

This function generates a number of blocks of random output. This is an internal function used only by the generator. Any entity outside the PRNG should not be able to call this function.

function GENERATEBLOCKS
input: $\mathcal{G}$ Generator state; modified by this function.
k Number of blocks to generate.
output: r Pseudorandom string of $16k$ bytes.

assert $C \neq 0$
Start with the empty string.
$r \leftarrow \epsilon$
Append the necessary blocks.
for $i = 1, \ldots, k$ **do**
$r \leftarrow r \parallel E(K, C)$
$C \leftarrow C + 1$
od
return r

Of course, the $E(K, C)$ function is the block cipher encryption function with key K and plaintext C. The GENERATEBLOCKS function first checks that C is not zero, as that is the indication that this generator has never been seeded. The symbol ϵ denotes the empty string. The loop starts with an empty string in r and appends each newly computed block to r to build the output value.

9.4.4 Generate Random Data

This function generates random data at the request of the user of the generator. It allows for output of up to 2^{20} bytes and ensures that the generator forgets any information about the result it generated.

function PSEUDORANDOMDATA
input: $\mathcal{G}$ Generator state; modified by this function.
n Number of bytes of random data to generate.
output: r Pseudorandom string of n bytes.

Limit the output length to reduce the statistical deviation from perfectly random outputs. Also ensure that the length is not negative.
assert $0 \leq n \leq 2^{20}$
Compute the output.
$r \leftarrow$ *first-n-bytes*(GENERATEBLOCKS($\mathcal{G}$, $\lceil n/16 \rceil$))
Switch to a new key to avoid later compromises of this output.
$K \leftarrow$ GENERATEBLOCKS($\mathcal{G}$, 2)
return r

The output is generated by a call to GenerateBlocks, and the only change is that the result is truncated to the correct number of bytes. (The $\lceil \cdot \rceil$ operator is the round-upwards operator.) We then generate two more blocks to get a new key. Once the old K has been forgotten, there is no way to recompute the result r. As long as PseudoRandomData does not keep a copy of r, or forget to wipe the memory r was stored in, the generator has no way of leaking any data about r once the function completes. This is exactly why any future compromise of the generator cannot endanger the secrecy of earlier outputs. It does endanger the secrecy of future outputs, a problem that the accumulator will address.

The function PseudoRandomData is limited in the amount of data it can return. One can specify a wrapper around this that can return larger random strings by repeated calls to PseudoRandomData. Note that you should not increase the maximum output size per call, as that increases the statistical deviation from pure random. Doing repeated calls to PseudoRandomData is quite efficient. The only real overhead is that for every 1 MB of random data produced, you have to generate 32 extra random bytes (for the new key) and run the key schedule of the block cipher again. This overhead is insignificant for all of the block ciphers we suggest.

9.4.5 Generator Speed

The generator for Fortuna that we just described is a cryptographically strong PRNG in the sense that it converts a seed into an arbitrarily long pseudorandom output. It is about as fast as the underlying block cipher; on a PC-type CPU it should run in less than 20 clock cycles per generated byte for large requests. Fortuna can be used as a drop-in replacement for most PRNG library functions.

9.5 Accumulator

The accumulator collects real random data from various sources and uses it to reseed the generator.

9.5.1 Entropy Sources

We assume there are several sources of entropy in the environment. Each source can produce events containing entropy at any point in time. It does not matter exactly what you use as your sources, as long as there is at least one source that generates data that is unpredictable to the attacker. As you cannot know how the attacker will attack, the best bet is to turn anything that looks like unpredictable data into a random source. Keystrokes and mouse movements make reasonable sources. In addition, you should add as many timing sources

as practical. You could use accurate timing of keystrokes, mouse movements and clicks, and responses from the disk drives and printers, preferably all at the same time. Again, it is not a problem if the attacker can predict or copy the data from some of the sources, as long as she cannot do it for all of them.

Implementing sources can be a lot of work. The sources typically have to be built into the various hardware drivers of the operating system. This is almost impossible to do at the user level.

We identify each source by a unique source number in the range $0 \ldots 255$. Implementors can choose whether to allocate the source numbers statically or dynamically. The data in each event is a short sequence of bytes. Sources should only include the unpredictable data in each event. For example, timing information can be represented by the two or four least significant bytes of an accurate timer. There is no point including the day, month, and year. It is safe to assume that the attacker knows those.

We will be concatenating various events from different sources. To ensure that a string constructed from such a concatenation uniquely encodes the events, we have to make sure the string is parsable. Each event is encoded as three or more bytes of data. The first byte contains the random source number. The second byte contains the number of additional bytes of data. The subsequent bytes contain whatever data the source provided.

Of course, the attacker will know the events generated by some of the sources. To model this, we assume that some of the sources are completely under the attacker's control. The attacker chooses which events these sources generate at which times. And like any other user, the attacker can ask for random data from the PRNG at any point in time.

9.5.2 Pools

To reseed the generator, we need to pool events in a pool large enough that the attacker can no longer enumerate the possible values for the events in the pool. A reseed with a ''large enough'' pool of random events destroys the information the attacker might have had about the generator state. Unfortunately, we don't know how many events to collect in a pool before using it to reseed the generator. This is the problem Yarrow tried to solve by using entropy estimators and various heuristic rules. Fortuna solves it in a much better way.

There are 32 pools: $P_0, P_1, \ldots, P_{31}$. Each pool conceptually contains a string of bytes of unbounded length. In practice, the only way that string is used is as the input to a hash function. Implementations do not need to store the unbounded string, but can compute the hash of the string incrementally as it is assembled in the pool.

Each source distributes its random events over the pools in a cyclical fashion. This ensures that the entropy from each source is distributed more or

less evenly over the pools. Each random event is appended to the string in the pool in question.

We reseed the generator every time pool P_0 is long enough. Reseeds are numbered $1, 2, 3, \ldots$. Depending on the reseed number r, one or more pools are included in the reseed. Pool P_i is included if 2^i is a divisor of r. Thus, P_0 is used every reseed, P_1 every other reseed, P_2 every fourth reseed, etc. After a pool is used in a reseed, it is reset to the empty string.

This system automatically adapts to the situation. If the attacker knows very little about the random sources, she will not be able to predict P_0 at the next reseed. But the attacker might know a lot more about the random sources, or she might be (falsely) generating a lot of the events. In that case, she probably knows enough of P_0 that she can reconstruct the new generator state from the old generator state and the generator outputs. But when P_1 is used in a reseed, it contains twice as much data that is unpredictable to her; and P_2 will contain four times as much. Irrespective of how many fake random events the attacker generates, or how many of the events she knows, as long as there is at least one source of random events she can't predict, there will always be a pool that collects enough entropy to defeat her.

The speed at which the system recovers from a compromised state depends on the rate at which entropy (with respect to the attacker) flows into the pools. If we assume this is a fixed rate ρ, then after t seconds we have in total ρt bits of entropy. Each pool receives about $\rho t/32$ bits in this time period. The attacker can no longer keep track of the state if the generator is reseeded with a pool with more than 128 bits of entropy in it. There are two cases. If P_0 collects 128 bits of entropy before the next reseed operation, then we have recovered from the compromise. How fast this happens depends on how large we let P_0 grow before we reseed. The second case is when P_0 is reseeding too fast, due to random events known to (or generated by) the attacker. Let t be the time between reseeds. Then pool P_i collects $2^i\rho t/32$ bits of entropy between reseeds and is used in a reseed every $2^i t$ seconds. The recovery from the compromise happens the first time we reseed with pool P_i where $128 \leq 2^i\rho t/32 < 256$. (The upper bound derives from the fact that otherwise pool P_{i-1} would contain 128 bits of entropy between reseeds.) This inequality gives us

$$\frac{2^i\rho t}{32} < 256$$

and thus

$$2^i t < \frac{8192}{\rho}$$

In other words, the time between recovery points ($2^i t$) is bounded by the time it takes to collect 2^{13} bits of entropy ($8192/\rho$). The number 2^{13} seems a bit large, but it can be explained in the following way. We need at least $128 = 2^7$ bits to recover from a compromise. We might be unlucky if the system reseeds

just before we have collected 2^7 bits in a particular pool, and then we have to use the next pool, which will collect close to 2^8 bits before the reseed. Finally, we divide our data over 32 pools, which accounts for another factor of 2^5.

This is a very good result. This solution is within a factor of 64 of an ideal solution (it needs at most 64 times as much randomness as an ideal solution would need). This is a constant factor, and it ensures that we can never do terribly badly and will always recover eventually. Furthermore, we do not need to know how much entropy our events have or how much the attacker knows. That is the real advantage Fortuna has over Yarrow. The impossible-to-construct entropy estimators are gone for good. Everything is fully automatic; if there is a good flow of random data, the PRNG will recover quickly. If there is only a trickle of random data, it takes a long time to recover.

So far we've ignored the fact that we only have 32 pools, and that maybe even pool P_{31} does not collect enough randomness between reseeds to recover from a compromise. This could happen if the attacker injected so many random events that 2^{32} reseeds would occur before the random sources that the attacker has no knowledge about have generated 2^{13} bits of entropy. This is unlikely, but to stop the attacker from even trying, we will limit the speed of the reseeds. A reseed will only be performed if the previous reseed was more than 100 ms ago. This limits the reseed rate to 10 reseeds per second, so it will take more than 13 years before P_{32} would ever have been used, had it existed. Given that the economic and technical lifetime of most computer equipment is considerably less than ten years, it seems a reasonable solution to limit ourselves to 32 pools.

9.5.3 Implementation Considerations

There are a couple of implementation considerations in the design of the accumulator.

9.5.3.1 Distribution of Events Over Pools

The incoming events have to be distributed over the pools. The simplest solution would be for the accumulator to take on that role. However, this is dangerous. There will be some kind of function call to pass an event to the accumulator. It is quite possible that the attacker could make arbitrary calls to this function, too. The attacker could make extra calls to this function every time a ''real'' event was generated, thereby influencing the pool that the next ''real'' event would go to. If the attacker manages to get all ''real'' events into pool P_0, the whole multi-pool system is ineffective, and the single-pool attacks apply. If the attacker gets all ''real'' events into P_{31}, they essentially never get used.

Our solution is to let every event generator pass the proper pool number with each event. This requires the attacker to have access to the memory of the program that generates the event if she wants to influence the pool choice. If the attacker has that much access, then the entire source is probably compromised as well.

The accumulator could check that each source routes its events to the pools in the correct order. It is a good idea for a function to check that its inputs are properly formed, so this would be a good idea in principle. But in this situation, it is not always clear what the accumulator should do if the verification fails. If the whole PRNG runs as a user process, the PRNG could throw a fatal error and exit the program. That would deprive the system of the PRNG just because a single source misbehaved. If the PRNG is part of the operating system kernel, it is much harder. Let's assume a particular driver generates random events, but the driver cannot keep track of a simple 5-bit cyclical counter. What should the accumulator do? Return an error code? Chances are that a programmer who makes such simple mistakes doesn't check the return codes. Should the accumulator halt the kernel? A bit drastic, and it crashes the whole machine because of a single faulty driver. The best idea we've come up with is to penalize the driver in CPU time. If the verification fails, the accumulator can delay the driver in question by a second or so.

This idea is not terribly useful, because the reason we let the caller determine the pool number is that we assume the attacker might make false calls to the accumulator with fake events. If this happens and the accumulator checks the pool ordering, the real event generator will be penalized for the misbehavior of the attacker. Our conclusion: the accumulator should not check the pool ordering, because there isn't anything useful the accumulator can do if it detects that something is wrong. Each random source is responsible for distributing its events in cyclical order over the pools. If a random source screws up, we might lose the entropy from that source (which we expect), but no other harm will be done.

9.5.3.2 Running Time of Event Passing

We want to limit the amount of computation necessary when an event is passed to the accumulator. Many of the events are timing events, and they are generated by real-time drivers. These drivers do not want to call an accumulator if once in a while the call takes a long time to complete.

There is a certain minimum number of computations that we will need to do. We have to append the event data to the selected pool. Of course, we are not going to store the entire pool string in memory, because the length of a pool string is potentially unbounded. Recall that popular hash functions are iterative? For each pool we will have a short buffer and compute a partial hash

as soon as that buffer is full. This is the minimum amount of computation required per event.

We do not want to do the whole reseeding operation, which uses one or more pools to reseed the generator. This takes an order of magnitude more time than just adding an event to a pool. Instead, this work will be delayed until the next user asks for random data, when it will be performed before the random data is generated. This shifts some of the computational burden from the event generators to the users of random data, which is reasonable since they are also the ones who are benefiting from the PRNG service. After all, most event generators are not benefiting from the random data they help to produce.

To allow the reseed to be done just before the request for random data is processed, we must encapsulate the generator. In other words, the generator will be hidden so that it cannot be called directly. The accumulator will provide a RandomData function with the same interface as PseudoRandomData. This avoids problems with certain users calling the generator directly and bypassing the reseeding process that we worked so hard to perfect. Of course, users can still create their own instance of the generator for their own use.

A typical hash function, like SHA-256, and hence SHA_d-256, processes message inputs in fixed-size blocks. If we process each block of the pool string as soon as it is complete, then each event will lead to at most a single hash block computation. However, this also has a disadvantage. Modern computers use a hierarchy of caches to keep the CPU busy. One of the effects of the caches is that it is more efficient to keep the CPU working on the same thing for a while. If you process a single hash code block, then the CPU must read the hash function code into the fastest cache before it can be run. If you process several blocks in sequence, then the first block forces the code into the fastest cache, and the subsequent blocks take advantage of this. In general, performance on modern CPUs can be significantly increased by keeping the CPU working within a small loop and not letting it switch between different pieces of code all the time.

Considering the above, one option is to increase the buffer size per pool and collect more data in each buffer before computing the hash. The advantage is a reduction in the total amount of CPU time needed. The disadvantage is that the maximum time it takes to add a new event to a pool increases. This is an implementation trade-off that we cannot resolve here. It depends too much on the details of the environment.

9.5.4 Initialization

Initialization is, as always, a simple function. So far we've only talked about the generator and the accumulator, but the functions we are about to define

are part of the external interface of Fortuna. Their names reflect the fact that they operate on the whole PRNG.

function INITIALIZEPRNG
output: $\mathcal{R}$ PRNG state.

Set the 32 pools to the empty string.
for $i = 0, \ldots, 31$ **do**
$P_i \leftarrow \epsilon$
od
Set the reseed counter to zero.
RESEEDCNT $\leftarrow 0$
And initialize the generator.
$\mathcal{G} \leftarrow$ INITIALIZEGENERATOR()
Package up the state.
$\mathcal{R} \leftarrow (\mathcal{G}, \text{RESEEDCNT}, P_0, \ldots, P_{31})$
return $\mathcal{R}$

9.5.5 Getting Random Data

This is not quite a simple wrapper around the generator component of the PRNG, because we have to handle the reseeds here.

function RANDOMDATA
input: $\mathcal{R}$ PRNG state, modified by this function.
n Number of bytes of random data to generate.
output: r Pseudorandom string of bytes.

if $length(P_0) \geq$ MINPOOLSIZE $\wedge$ last reseed $>$ 100 ms ago **then**
We need to reseed.
RESEEDCNT $\leftarrow$ RESEEDCNT $+ 1$
Append the hashes of all the pools we will use.
$s \leftarrow \epsilon$
for $i \in 0, \ldots, 31$ **do**
if $2^i \mid$ RESEEDCNT **then**
$s \leftarrow s \parallel \text{SHA}_d\text{-256}(P_i)$
$P_i \leftarrow \epsilon$
fi
od
Got the data, now do the reseed.
RESEED$(\mathcal{G}, s)$
fi

if ReseedCnt $= 0$ **then**
Generate error, prng *not seeded yet*
else
Reseeds (if needed) are done. Let the generator that is part of $\mathcal{R}$ do the work.
return PseudoRandomData($\mathcal{G}, n$)
fi

This function starts by checking the size of pool P_0 against the parameter MinPoolSize to see if it should do a reseed. You can use a very optimistic estimate of how large the pool size has to be before it can contain 128 bits of entropy. Assuming that each event contains 8 bits of entropy and takes 4 bytes in the pool (this corresponds to 2 bytes of event data), a suitable value for MinPoolSize would be 64 bytes. It doesn't matter much, although choosing a value smaller than 32 seems inadvisable. Choosing a much larger value is not good, either, because that will delay the reseed even if there are very good random sources available.

The next step is to increment the reseed count. The count was initialized to 0, so the very first reseed uses the value 1. This automatically ensures that the first reseed uses only P_0, which is what we want.

The loop appends the hashes of the pools. We could also have appended the pools themselves, but then every implementation would have to store entire pool strings, not just the running hash-computation of each pool. The notation $2^i \mid$ ReseedCnt is a divisor test. It is true if 2^i is a divisor of the value ReseedCnt. Once an i value fails this test, all tests of the subsequent loop iterations will also fail, which suggests an optimization.

9.5.6 Add an Event

Random sources call this routine when they have another random event. Note that the random sources are each uniquely identified by a source number. We will not specify how to allocate the source numbers because the solution depends on the local situation.

function AddRandomEvent
input: $\mathcal{R}$ prng state, modified by this function.
s Source number in range $0, \ldots, 255$.
i Pool number in range $0, \ldots, 31$. Each source must distribute its events over all the pools in a round-robin fashion.
e Event data. String of bytes; length in range $1, \ldots, 32$.

Check the parameters first.
assert $1 \le \mathit{length}(e) \le 32 \wedge 0 \le s \le 255 \wedge 0 \le i \le 31$
Add the data to the pool.
$P_i \leftarrow P_i \parallel s \parallel \mathit{length}(e) \parallel e$

The event is encoded in $2 + length(e)$ bytes, with both s and $length(e)$ being encoded as a single byte. This concatenation is then appended to the pool. Note that our specifications just append data to the pool, but do not mention any hash computation. We only specify the hashing of the pool at the point in time where we use it. A real implementation should compute the hashes on the fly. That is functionally equivalent and easier to implement, but specifying it directly would be far more complicated.

We have limited the length of the event data to 32 bytes. Larger events are fairly useless; random sources should not pass large amounts of data, but rather, only those few bytes that contain unpredictable random data. If a source has a large amount of data that contains some entropy spread throughout it, the source should hash the data first. The AddRandomEvent function should always return quickly. This is especially important because many sources—by their very nature—perform real-time services. These sources cannot spend too much time calling AddRandomEvent. Even if a source produces small events, it should not have to wait on other callers whose events are large. Most implementations will need to serialize the calls to AddRandomEvent by using a mutex of some sort to ensure that only one event is being added at the same time.[4]

Some random sources might not have the time to call AddRandomEvent. In this case, it might be necessary to store the events in a buffer and have a separate process pick the events from the buffer and feed them to the accumulator.

An alternative architecture allows the sources to simply pass the events to the accumulator process, and has a separate thread in the accumulator perform all the hash computations. This is a more complex design, but it does have advantages for the entropy sources. The choice depends very much on the actual situation.

9.6 Seed File Management

Our PRNG so far will collect entropy and generate random data after the first reseed. However, if we reboot a machine we have to wait for the random sources to produce enough events to trigger the first reseed before any random data is available. In addition, there is no guarantee that the state after the first reseed is, in fact, unpredictable to the attacker.

The solution is to use a seed file. The PRNG keeps a separate file full of entropy, called the seed file. This seed is not made available to anyone else. After a reboot, the PRNG reads the seed file and uses it as entropy to get into an

[4] In a multithreaded environment, you should always be very careful to ensure that different threads do not interfere with each other.

unknown state. Of course, once the seed file has been used in this manner, it needs to be rewritten with new data.

We will describe seed file management, first under the assumption that the file system supports atomic operations; later we will discuss the issues involved with implementing seed file management on real systems.

9.6.1 Write Seed File

The first thing to do is generate a seed file. This is done with a simple function.

function WRITESEEDFILE
input: $\mathcal{R}$ PRNG state, modified by this function.
f File to write to.
write(f, RANDOMDATA($\mathcal{R}$, 64))

This function simply generates 64 bytes of random data and writes it to the file. This is slightly more data than absolutely needed, but there is little reason to be parsimonious with the bytes here.

9.6.2 Update Seed File

Obviously we need to be able to read a seed file, too. For reasons explained below, we always update the seed file in the same operation.

function UPDATESEEDFILE
input: $\mathcal{R}$ PRNG state, modified by this function.
f File to be updated.
$s \leftarrow$ *read*(f)
assert *length*(s) = 64
RESEED($\mathcal{G}$, s)
write(f, RANDOMDATA($\mathcal{R}$, 64))

This function reads the seed file, checks its length, and reseeds the generator. It then rewrites the seed file with new random data.

This routine must ensure that no other use is made of the PRNG between the reseed it causes and the writing of the new data to the seed file. Here is the problem: after a reboot, the seed file is read by this function, and the data is used in a reseed. Suppose the attacker asks for random data before the seed file has been updated. As soon as this random data is returned, but before the seed file is updated, the attacker resets the machine. At the next reboot, the same seed file data will be read and used to reseed the generator. This time, an innocent user asks for random data before the seed file has been rewritten. He will get the same random data that the attacker got earlier. This violates

the secrecy of the random data. As we often use random data to generate cryptographic keys, this is a rather serious problem.

The implementation should ensure that the seed file is kept secret. Also, all updates to the seed file must be atomic (see Section 9.6.5).

9.6.3 When to Read and Write the Seed File

When the computer is rebooted, the PRNG does not have any entropy to generate random data from. This is why the seed file is there. Thus, the seed file should be read and updated after every reboot.

As the computer runs, it collects entropy from various sources. We eventually want this entropy to affect the seed file as well. One obvious solution is to rewrite the seed file just as the machine is shutting down. As some computers will never be shut down in an orderly fashion, the PRNG should also rewrite the seed file at regular intervals. We won't spell out the details here, as they are quite uninteresting and often depend on the platform. It is important to ensure that the seed file is updated regularly from the PRNG after it has collected a fair amount of entropy. A reasonable solution would be to rewrite the seed file at every shutdown and every 10 minutes or so.

9.6.4 Backups and Virtual Machines

Trying to do the reseeding correctly opens a can of worms. We cannot allow the same state of the PRNG to be repeated twice. We use the file system to store a seed file to prevent this. But most file systems are not designed to avoid repeating the same state twice, and this causes us a lot of trouble.

First of all, there are backups. If you make a backup of the entire file system and then reboot the computer, the PRNG will be reseeded from the seed file. If you later restore the entire file system from the backup and reboot the computer, the PRNG will be reseeded from the very same seed file. In other words, until the accumulator has collected enough entropy, the PRNG will produce the same output after the two reboots. This is a serious problem, as an attacker can do this to retrieve the random data that another user got from the PRNG.

There is no direct defense against this attack. If the backup system is capable of recreating the entire permanent state of the computer, there is nothing we can do to prevent the PRNG state from repeating itself. Ideally, we would fix the backup system to be PRNG-aware, but that is probably too much to ask. Hashing the seed file together with the current time would solve the problem as long as the attacker does not reset the clock to the same time. The same solution could be used if the backup system were guaranteed to keep a counter of how many restore-operations it had done. We could hash the seed file with the restore counter.

Virtual machines pose a similar problem to backups. If a VM's state is saved and then restarted twice, both instances would begin with the same PRNG state. Fortunately, some of the same solutions for backups also apply to multiple VM instances starting from the same state.

The issues with backups and virtual machines deserve further study, but because they are highly platform-dependent, we do not give a general treatment here.

9.6.5 Atomicity of File System Updates

Another important problem associated with the seed file is the atomicity of file system updates. On most operating systems, if you write a seed file, all that happens is that a few memory buffers get updated. The data is not actually written to disk until much later. Some file systems have a ''flush'' command that purports to write all data to disk. However, this can be an extremely slow operation, and we have seen cases where the hardware lied to the software and simply refused to implement the ''flush'' command properly.

Whenever we reseed from our seed file, we must update it before allowing any user to ask for random data. In other words, we must be absolutely sure that the data has been modified on the magnetic media. Things become even more complicated when you consider that many file systems treat file data and file administration information separately. So rewriting the seed file might make the file administration data temporarily inconsistent. If the power fails during that time, we could get a corrupted seed file or even lose the seed file entirely—not a good idea for a security system.

Some file systems use a journal to solve some of these problems. This is a technique originally developed for large database systems. The journal is a sequential list of all the updates that have been done to the file system. When properly used, a journal can ensure that updates are always consistent. Such a file system is always preferable from a reliability point of view. Unfortunately, some of the very common file systems only apply the journal to the administrative information, which isn't quite good enough for our goals.

As long as the hardware and operating system do not support fully atomic and permanent file updates, we cannot create a perfect seed file solution. You will need to investigate the particular platform that you work on and do the best you can to reliably update the seed file.

9.6.6 First Boot

When we start the PRNG for the very first time, there is no seed file to use for a reseed. Take, for example, a new PC that had its OS installed in the factory. The OS is now generating some administrative cryptographic keys for the

installation, for which it needs the PRNG. For ease of production, all machines are identical and loaded with identical data. There is no initial seed file, so we cannot use that. We could wait for enough random events to trigger one or more reseeds, but that takes a long time, and we'd never know when we had collected enough entropy to be able to generate good cryptographic keys.

A good idea would be for the installation procedure to generate a random seed file for the PRNG during the configuration. It could, for example, use a PRNG on a separate computer to generate a new seed file for each machine. Or maybe the installation software could ask the tester to wiggle the mouse to collect some initial entropy. The choice of solution depends on the details of the environment, but somehow initial entropy has to be provided. Not providing initial entropy is not an option. The entropy accumulator can take quite a while to seed the PRNG properly, and it is quite likely that some very important cryptographic keys will be generated by the PRNG shortly after the installation of the machine.

Keep in mind that the Fortuna accumulator will seed the generator as soon as it *might* have enough entropy to be really random. Depending on how much entropy the sources actually deliver—something that Fortuna has no knowledge about—it could take quite a while before enough entropy has been gathered to properly reseed the generator. Having an outside source of randomness to create the first seed file is probably the best solution.

9.7 Choosing Random Elements

Our PRNG produces sequences of random bytes. Sometimes this is exactly what you need. In other situations you try to pick a random element from a set. This requires some care to do right.

Whenever we choose a random element, we implicitly assume that the element is chosen uniformly at random from the specified set (unless we specify another distribution). This means that each element should have exactly the same probability of being chosen.[5] This is harder than one might think.

Let n be the number of elements in the set we are choosing from. We will only discuss how to choose a random element from the set $0, 1, \ldots, n-1$. Once you can do this, you can choose elements from any set of size n.

If $n = 0$, there are no elements to choose from, so this is a simple error. If $n = 1$ you have no choice; again a simple case. If $n = 2^k$, then you just get k bits of random data from the PRNG and interpret them as a number in the range $0, \ldots, n-1$. This number is uniformly random. (You might have to get

[5] If we are designing for a 128-bit security level, we could afford a deviation from the uniform probability of 2^{-128}, but it is easier to do it perfectly.

a whole number of bytes from the PRNG and throw away a few bits of the last byte until you're left with k bits, but this is easy.)

What if n is not a power of two? Well, some programs choose a random 32-bit integer and take it modulo n. But that algorithm introduces a bias in the resulting probability distribution. Let's take $n = 5$ as an example and define $m := \lfloor 2^{32}/5 \rfloor$. If we take a uniformly random 32-bit number and reduce it modulo 5, then the results 1, 2, 3, and 4 each occur with a probability of $m/2^{32}$, while the result 0 occurs with a probability of $(m + 1)/2^{32}$. The deviation in probability is small, but could very well be significant. It would certainly be easy to detect the deviation within the 2^{128} steps we allow the attacker.

The proper way to select a random number in an arbitrary range is to use a trial-and-error approach. To generate a random value in the range $0, \ldots, 4$, we first generate a random value in the range $0, \ldots, 7$, which we can do since 8 is a power of 2. If the result is 5 or larger, we throw it away and choose a new random number in the range $0, \ldots, 7$. We keep doing this until the result is in the desired range. In other words, we generate a random number with the right number of bits in it and throw away all the improper ones.

Here is a more formal specification for how to choose a random number in the range $0, \ldots, n - 1$ for $n \geq 2$.

1. Let k be the smallest integer such that $2^k \geq n$.
2. Use the PRNG to generate a k-bit random number K. This number will be in the range $0, \ldots, 2^k - 1$. You might have to generate a whole number of bytes and throw away part of the last byte, but that's easy.
3. If $K \geq n$ go back to step 2.
4. The number K is the result.

This can be a bit of a wasteful process. In the worst case, we throw away half our attempts on average. Here is an improvement. As $2^{32} - 1$ is a multiple of 5, we could choose a random number in the range $0, \ldots, 2^{32} - 2$ and take the result modulo 5 for our answer. To choose a value in the range $0, \ldots, 2^{32} - 2$, we use the ''inefficient'' try-and-throw-away algorithm, but now the probability of having to throw the intermediate result away is very low.

The general formulation is to choose a convenient k such that $2^k \geq n$. Define $q := \lfloor 2^k/n \rfloor$. First choose a random number r in the range $0, \ldots, nq - 1$ using the try-and-throw-away rules. Once a suitable r has been generated, the final result is given by (r mod n).

We don't know of any way to generate uniformly random numbers on sizes that are not a power of two without having to throw away some random bits now and again. That is not a problem. Given a decent PRNG, there is no shortage of random bits.

9.8 Exercises

Exercise 9.1 Investigate the random number generators built into three of your favorite programming languages. Would you use these random number generators for cryptographic purposes?

Exercise 9.2 Using an existing cryptography library, write a short program that generates a 256-bit AES key using a cryptographic PRNG.

Exercise 9.3 For your platform, language, and cryptography library of choice, summarize how the cryptographic PRNG works internally. Consider issues including but not limited to the following: how the entropy is collected, how reseeding occurs, and how the PRNG handles reboots.

Exercise 9.4 What are the advantages of using a PRNG over an RNG? What are the advantages of using an RNG over a PRNG?

Exercise 9.5 Using a cryptographic PRNG that outputs a stream of bits, implement a random number generator that outputs random integers in the set $0, 1, \ldots, n-1$ for any n between 1 and 2^{32}.

Exercise 9.6 Implement a naive approach for generating random numbers in the set $0, 1, \ldots, 191$. For this naive approach, generate a random 8-bit value, interpret that value as an integer, and reduce that value modulo 192. Experimentally generate a large number of random numbers in the set $0, 1, \ldots, 191$ and report on the distribution of results.

Exercise 9.7 Find a new product or system that uses (or should use) a cryptographic PRNG. This might be the same product or system you analyzed for Exercise 1.8. Conduct a security review of that product or system as described in Section 1.12, this time focusing on the issues surrounding the use of random numbers.