Editors: Peter Gutmann, pgut001@cs.auckland.ac.nz
David Naccache, david.naccache@ens.fr
Charles C. Palmer, ccpalmer@us.ibm.com

Randomness in Cryptography

Rosario Gennaro
IBM T.J. Watson Research Center

Although more formal definitions of randomness exist, a colloquial one will suffice here: a random process is one whose consequences are unknown. Intuitively, this is why randomness is crucial in cryptographic applications—because it provides a way to create information that an adversary can't learn or predict. It's then the task of a good protocol designer to leverage this power in the best possible way to protect data and communication.

In this installment of Crypto Corner, we'll look at some basic uses of randomness in cryptography and briefly review the process of securely generating randomness.

Randomly unpredictable?

Although it's useful in some circumstances, the informal definition from the introductory paragraph seems to confuse the concept of randomness with the concept of unpredictability. A random process is unpredictable, but the reverse isn't completely true—some unpredictable processes aren't fully random.

The canonical example of a random process is tossing an unbiased coin that will land on either side with the same chance (1/2). Repeated coin tosses are *independent*, meaning the outcome of one toss doesn't affect the outcome of the others. If we associate the bit 0 with the event that the coin lands on its head, then by repeatedly flipping a coin N times, we obtain a uniformly distributed N-bit string: each possible string has the same chance of being produced (2^{-N}).

Consider now the process of rolling a die: its output is the binary expansion of the number on which the die rolls, so if the die rolls a 1, the output will be 001, for 2, it's 010, and so on, until for 6, it's 110. To produce an N-bit string, we would need to roll the die $N/3$ times. Although this string is unpredictable, it isn't fully random—in particular, it isn't uniformly distributed. A roll of the die would never produce strings 000 and 111, for example, which means this process can't produce all possible N-bit strings; the ones that *are* produced will appear with a probability higher than 2^{-N}.

Entropy

Mathematicians often use entropy to measure unpredictability. Entropy is the negative logarithm of the probability of the process's most likely output, so the higher a process's entropy, the more unpredictable the process. A process of maximal entropy is completely random—its output is uniformly distributed. The process of tossing N coins, for example, has entropy N because $-\log 2^{-N} = N$. On the other hand, the process of rolling a die $N/3$ times has entropy $\sim 0.86\,N$ because each 3-bit string appears with probability 1/6; they're independent, so the probability of each string appearing is $(1/6)^{N/3}$, thus $-\log(1/6)^{N/3} = N/3 \log 6 \sim 0.86N$.

Cryptographic applications require uniformly distributed random strings, but physical random processes usually aren't perfect because they generate highly unpredictable, yet not uniformly distributed, values. Fortunately, the mathematical and computer science communities have developed ways to extract pure randomness from high-entropy sources via the use of *entropy-smoothing* hash functions.[1]

One-time pads

The most secure encryption scheme is the one-time pad, which is a key that can be used only once: Alice and Bob share a secret key k, a long binary string whose bits are selected uniformly and independently. When Alice wants to send a message m to Bob, she sends the ciphertext $c = m \oplus k$, where $\oplus$ is the bitwise exclusive-OR operator (we assume that m is a shorter binary string than k). On receiving c, Bob computes $m = c \oplus k$ by applying the bitwise exclusive-OR operator to both the ciphertext and the key. But what if the adversary can see the ciphertext c? Will it help him or her understand the message m?

The answer is no: if Alice sent m_0, then the key could be $k_0 = c \oplus m_0$, which yields ciphertext c. Similarly, if Alice sent a different message m_1, the key could be $k_1 = c \oplus m_1$, which also yields c. Indeed, for any possible message m', a possible key k' exists such that $c = m' \oplus k'$. In other words, being able to see c doesn't provide any information about m's contents.

Notice, however, that the truth in this statement is predicated on k's true randomness. If k has more zeroes than ones, for example, we know that most of the bits in c are identical to the bits in m. Alternatively, if we assume that the bits of k in position i tend to agree with the bits in position $i + 1$ (for $i = 1, 3, 5, 7, \ldots$), we would be able to infer interesting relationships about the bits of message m by looking at c (in particular, the relationship between the i and $i + 1$ bits of m is the same as the corresponding bits of c). More generally, if the parties didn't select k in a completely uniform way, then certain values might not be generated (for example, k_0), which means that seeing c would exclude certain messages (in this case, m_0).

Key generation

The previous example points to the most important use of randomness in cryptography, which is to generate keys, the secret pieces of information that let legitimate key holders communicate securely. As its name implies, the one-time pad is a key that can be used only once, but other cryptographic algorithms (such as the Data Encryption Standard [DES] and the newly introduced Advanced Encryption Standard [AES]) safely allow key reuse multiple times.

For any cipher, it's extremely important that communicating parties choose the key at random, without any possible bias or correlation between bits. The one-time pad's simple weaknesses to a nonrandom key might not translate directly to encryption algorithms such as DES and AES, which use more complex and iterated operators, but a nonrandom key could substantially weaken a cipher to the point of making other attacks feasible.

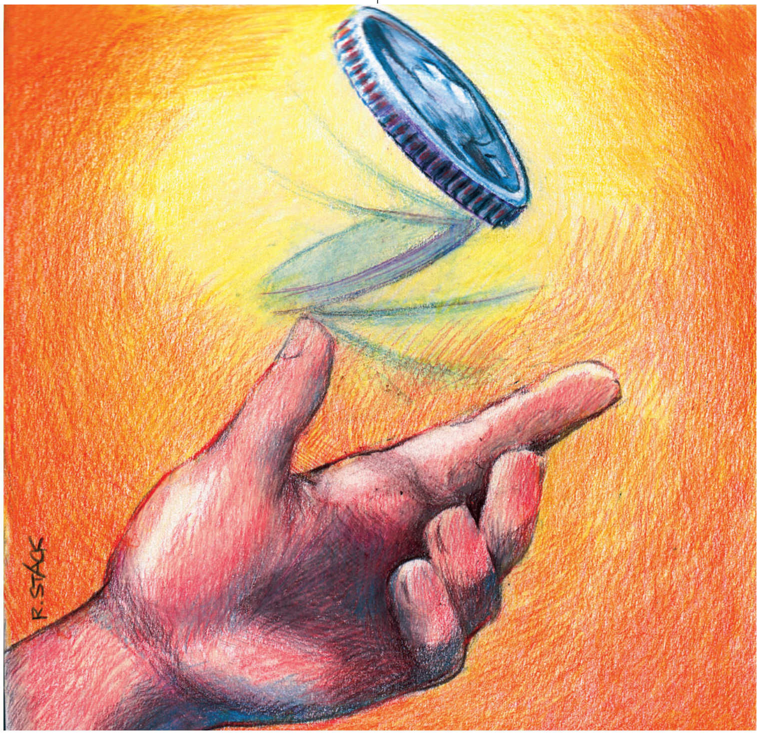
In the DES case, for example, the key indexes random-looking tables (the famous S-boxes, which are tables with 4-bit entries on each of their 64 rows; six key bits affect the selection of a row to obtain a 4-bit output). If the key isn't random, the DES encryption process would use some entries in the S-boxes more often, whereas others might not be used at all; this could eventually lead to unforeseen attacks and weaknesses. However, we have an even more important and direct reason to make sure the parties generate keys randomly: preventing brute-force attacks. Given a message–ciphertext pair (m, c), an adversary can always discover the correct key by brute force—testing all possible keys and encrypting m to see if he or she can get c.

If the cipher selects the key uniformly at random, then a brute-force attack will take 2^N steps, where N is the key's length. In the case of AES, this takes 2^{128} steps—way beyond the ability of even the fastest known computer. But, again, the assumption that the key is random plays a fundamental role in bounding a brute-force attack's cost. An adversary can discover a nonrandom key much more quickly—if the adversary knows that the key's bits are biased toward zero, he or she won't bother testing keys with a lot of ones in them. Similarly, if the even-positioned key bits tend to agree with the previous bit in the key, the search space is immediately cut by a square root down to $2^{N/2}$. Indeed, only $N/2$ bits of the key are relevant (the odd-positioned ones, because the even-positioned ones tend to agree with the bits).

Message randomization

Randomness should also affect the entire encryption process, not just key generation. If encryption were deterministic, an adversary could infer some interesting information by simply observing the ciphertexts—if the sender transmits the same ciphertext twice, the adversary knows the same message was sent twice.

In the case of a public-key scheme, a deterministic encryption procedure provides the adversary with a way to test if a given message is the encrypted one or not. Given a ciphertext c encrypted under public key pk, an adversary can test if c is an encryption of m by simply re-encrypting m under pk and testing if the obtained ciphertext is in fact c. In the case of small message spaces (such as buy/sell orders for a stock broker), the adversary can guess the



correct plaintext immediately, without knowing the secret key.

Starting with Shafi Goldwasser and Silvio Micali's seminal paper on the subject,[2] the crypto literature offers various secure ways to use randomized encryption. However, we must remember that all these methods assume and require access to perfect sources of randomness. An imperfect or biased source of randomness will likely compromise the security of the entire encryption scheme.

Clearly we can't build a specialized cryptochip that will flip a perfectly unbiased coin every time we need a random bit, so where can we get randomness?

Nonces

Another important use of randomness is the generation of unique values, or *nonces*. If we repeatedly choose long strings at random, the chances that we'll generate the same string twice are very small. Specifically, an N-bit random string appears with probability 2^{-N}, thus when N is long enough (say, 160 bits), we can assume the strings will be unique. Unique strings are important in cryptographic protocols to prevent replay attacks, in which an adversary tries to reuse correct values from previous executions of the protocol, hoping that they'll seem correct in the current attempt.

Randomness is the easiest way to create nonces. Other possible options include counters and time stamps, but both of these approaches have drawbacks. Time stamps require a third-party service trusted with the computation and timesetting, and counters can only be used in certain applications in which parties can keep a common, synchronized, tamper-proof state.

Identification and zero knowledge

Perhaps the most fascinating example of the cryptographic use of randomness is zero knowledge[3] and its application to identification protocols. In a zero-knowledge proof, say Alice wants to prove to Bob that she knows a certain secret without revealing that secret to him. The application to identification is immediate: Alice's public key is an RSA modulus N, and she proves her identity by proving that she knows its factorization (without revealing it, to prevent Bob from impersonating her later).

Describing a zero-knowledge protocol in full detail is beyond this article's scope, but the following example gives a general idea of it. Alice wants to prove to Bob that she has extraordinary powers: she can correctly guess the number of leaves on a tree just by looking at it. Bob decides to test her, so they choose a tree, Alice guesses the number of leaves on it, and then they count the leaves one by one. However, Alice doesn't want to reveal the number of leaves to Bob (he might go around claiming to have the same powers), and, besides, Bob doesn't want to spend hours counting leaves.

They agree on the following protocol: Alice will look at the tree, guess the number of leaves, and then turn around. While she's not looking, Bob will remove a random number of leaves from the tree, and then Alice will turn back around and tell Bob how many leaves he pulled. If her guess is correct, Bob will believe her claimed powers.

Bob has no idea how many leaves are on the tree at the end of the protocol because all he knows is the number of leaves he removed. On the other hand, he'll be convinced of Alice's powers if she can guess the number of leaves—if she couldn't, she would have had to have been extremely lucky in her guess of the number of leaves he removed. Indeed, if Bob removes k leaves, where k is randomly chosen between 0 and N, and if Alice can't count leaves, she can only guess the right answer with probability $1/N$.

Notice how randomness plays a crucial role here. If Bob's choice of k isn't random and uniform, Alice's probability of guessing the correct k would increase, and she could convince him of her powers, even if she's just an ordinary person.

Randomness is scarce

Now that we've examined some of the uses of randomness in cryptography, let's look more closely at some possible ways to implement it in a cryptographic protocol. Clearly we can't build a specialized cryptochip that will flip a perfectly unbiased coin every time we need a random bit, so where can we get randomness?

The best source is a hardware device explicitly built to produce random bits. A very common device for this task is a Zener diode, which uses noise voltage at the diode's terminals. This process is highly unpredictable, but it isn't necessarily uniform, so its output is usually fed to an entropy-smoothing procedure to obtain truly random bits.

If a specialized hardware device isn't available, some operating systems today provide procedures that will attempt to generate random bits from environmental noise. An example is `/dev/random`, which resides in Unix-like operating systems: this generator keeps an estimate of the number of bits of noise in the entropy pool and, when queried, extracts truly random bits from the pool, again using a mathematical procedure to smooth entropy into real randomness. This entropy pool's quality is quite high, and the obtained random bits are suitable for cryptographic applications, provided the adversary doesn't have access to the entropy's source.

But how can we be sure that the machine's internal state is sufficiently hidden from an adversary? The first step is to ensure that the entropy's sources aren't trivial. Recall the 1996 Netscape incident, in which the browser generated encryption keys using "random" bits computed from the current time and the IDs of the process requesting the randomness. Ian Goldberg and David Wagner[4] (graduate students at the University of California, Berkeley, at the time) showed that these bits weren't random at all and that an adversary could easily predict them and thus compute the encryption keys. Unix's `/dev/random` does a good job at looking for randomness sources that aren't deterministic yet are hard for an outside observer to measure.

Care must be taken if the machine might have been compromised at some point; the inside knowledge the adversary learns about the machine's internal state might be helpful in predicting the random bits, and thus a rebooting or at least a refreshing step might be necessary.[5]

No matter the source of the randomness, the rate of these sources is likely to be small and insufficient for generating the number of random bits a cryptographic application requires. Fortunately, we can stretch random bits into pseudorandom ones by using cryptographically strong pseudorandom-bit generators. These deterministic algorithms take as input a small number of random bits and output a much longer string. Although this string isn't random in the information-theoretic sense, its crucial property is that it "looks" random to a computationally bounded adversary. In other words, these bits are good enough for cryptographic applications in which we always assume the adversary doesn't have sufficient computational power to, say, factor large numbers or test all possible 2^{128} AES keys.

The important point here is to use a secure (cryptographically strong) pseudorandom number generator. Notice that the generator's security is a very stringent condition: plenty of generators are good enough for noncryptographic applications, such as simulation and approximation, but they aren't cryptographically strong (their output can eventually be predicted). The literature is full of rich mathematical theories for cryptographically strong generators[6] and how to implement them. Although provably secure pseudorandom generators remain a little too slow for practical applications, several efficient constructions are based on cryptographic hash functions and sufficiently secure blockciphers.

Randomness is a fascinating topic in theoretical computer science, and its practical uses are countless, from simulation to approximation to optimization. However, in cryptographic applications, a high-quality and secure entropy source is key (pun intended). We also must control the environment in which the randomness source and the pseudorandom generator run: a compromise can have devastating consequences. ▢

Acknowledgments

I am indebted to Michael Rabin for his pedagogical exposition of the concept of zero knowledge.

References

1. N. Nisan, "Extracting Randomness: How and Why, A Survey," *IEEE Conf. Computational Complexity*, IEEE CS Press, 1996, pp. 44–58.
2. S. Goldwasser and S. Micali, "Probabilistic Encryption," *J. Computer System Sciences*, vol. 28, no. 2, 1984, pp. 270–299.
3. S. Goldwasser, S. Micali, and C. Rackoff, "The Knowledge Complexity of Interactive Proof Systems," *SIAM J. on Computing*, vol. 18, no. 1, 1989, pp. 186–208.
4. I. Goldberg and D. Wagner, "Randomness and the Netscape Browser," *Dr. Dobb's J.*, Jan. 1996, pp. 66–70; www.ddj.com/articles/1996/9601/.
5. B. Barak and S. Halevi, "A Model and Architecture for Pseudo-Random Generation with Applications to /dev/random," *ACM Conf. Computer and Comm. Security*, ACM Press, 2005, pp. 203–212.
6. M. Blum and S. Micali, "How to Generate Cryptographically Strong Sequences of Pseudo-Random Bits," *SIAM J. on Computing*, vol. 13, no. 4, 1984, pp. 850–864.

***Rosario Gennaro** is a research staff member at the IBM T.J. Watson Research Center. His research interests include the design and implementation of efficient yet provable cryptographic algorithms. Gennaro has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a member of the International Association for Cryptologic Research (ICAR). Contact him at rosario@us.ibm.com.*