NTNU | Norwegian University of Science and Technology

RANDOMNESS 1: ENTROPY

TTM4205 – Lecture 3

Tjerand Silde

21.08.2026

Contents

Contents

Reference Material

These slides are based on:

- JPA: parts of chapter 2 and 3
- DW: parts of chapter 8

Randomness

Randomness is the foundation in designing secure schemes:

- parameter and key generation
- probabilistic encryption and signatures
- modeling of hash functions
- analyzing attacks and security
- protecting implementations

Entropy

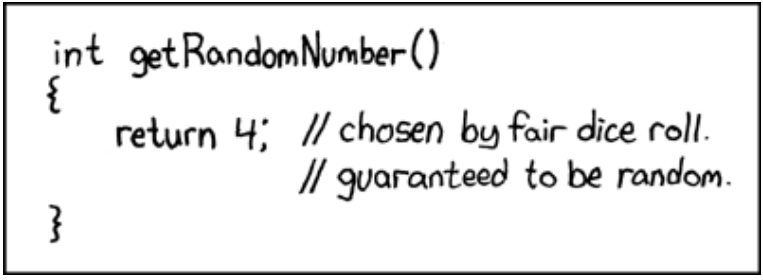
Let $\mathcal{X}$ be an alphabet. For example, we have $\mathcal{X} = \{0, 1\}$ when flipping a coin and $\mathcal{X} = \{1, 2, 3, 4, 5, 6\}$ when rolling a die.

Let p be a probability distribution over $\mathcal{X}$, that is, $p \colon \mathcal{X} \to [0, 1]$ with $\sum_{x \in \mathcal{X}} p(x) = 1$. For example, p can be the uniform distribution.

Let X be a random variable taking values in $\mathcal{X}$ according to p. The *(Shannon) entropy* $\mathbb{H}$ of X, measured in bits, is defined as

$$\mathbb{H}(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x) = \mathbb{E}[-\log_2 p(X)].$$

Entropy


int getRandomNumber()
{
return 4; // chosen by fair dice roll.
// guaranteed to be random.
}

Entropy

Examples

- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = 0, p(1) = 1$. The entropy is 0 bits.
- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = p(1) = 1/2$. The entropy is 1 bit.
- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = 1/3, p(1) = 2/3$. The entropy is

$$-(1/3 \cdot -1.585 + 2/3 \cdot -0.585) \approx 0.92 \text{ bits.}$$

- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = 1/4, p(1) = 3/4$. The entropy is

$$-(1/4 \cdot -2 + 3/4 \cdot -0.415) \approx 0.81 \text{ bits.}$$

- Let $\mathcal{X} = \{0, 1\}^\lambda$, uniform distribution. The entropy is λ bits.

Min-Entropy

Shannon entropy is an *average*, but an adversary guessing a key does not guess on average: the adversary starts with the most likely value.

The *min-entropy* $\mathbb{H}_\infty$ of X is defined from that best guess as

$$\mathbb{H}_\infty(X) = -\log_2\left(\max_{x\in\mathcal{X}} p(x)\right).$$

If $\mathbb{H}_\infty(X) = k$, then no single guess succeeds with probability better than 2^{-k}, so the adversary needs about 2^k guesses. This is the measure to use for keys, nonces, and seeds, and the one NIST SP 800-90B requires when a physical entropy source is assessed.

Min-Entropy

Examples

- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = p(1) = 1/2$. Then $\mathbb{H} = \mathbb{H}_\infty = 1$ bit.
- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = 1/4, p(1) = 3/4$. Then $\mathbb{H} \approx 0.81$ bits, while

$$\mathbb{H}_\infty = -\log_2(3/4) \approx 0.415 \text{ bits.}$$

- Let $\mathcal{X} = \{0, 1\}^{128}$ where the all-zero key has probability $1/2$ and the rest is uniform. Then $\mathbb{H} \approx 65$ bits, but

$$\mathbb{H}_\infty = -\log_2(1/2) = 1 \text{ bit.}$$

- The last key looks strong on average and is guessed on the first attempt half of the time.

Entropy vs. Min-Entropy

We always have $\mathbb{H}_\infty(X) \leq \mathbb{H}(X)$, with equality if and only if p is uniform.

- $\mathbb{H}$ answers: how many bits do I need to *store* X on average?
- $\mathbb{H}_\infty$ answers: how likely is the adversary to *guess* X?

Report key material in min-entropy: to get a 128-bit key, collect at least 128 bits of min-entropy and condition it with a hash function or a PRNG.

Averages hide the failure modes we care about in this course, so a source that passes an entropy estimate on average can still be trivially guessable.

Contents

Security Parameter

We estimate the security of a scheme by how many bit-operations are needed to break it. If an AES-128 key is sampled uniformly at random, then it takes 2^{128} trials to guess the right key.

We do not know more efficient attacks against AES-128 than brute force.

Security Parameter

We have more efficient algorithms for computing discrete logarithms. The generic algorithms run in time $\approx \sqrt{p}$ operations in a generic group $\mathbb{G}_p$ of prime order p, and this is optimal for a group with no exploitable structure.

For well-chosen elliptic curves, no attack better than the generic ones is known, so we need groups of order 256 bits to get 128 bits of security (as AES-128).

Security Parameter

We have even more efficient algorithms for computing discrete logarithms in finite fields and for factoring bi-primes (an RSA modulus).

For finite field Diffie-Hellman (DH) and Digital Signature Algorithm (DSA) over $\mathbb{F}_p$ and for RSA encryption and signatures over $\mathbb{Z}_n$ we need p and n to be of roughly 3072 bits to provide 128 bits of security (as AES-128).

It is due to sub-exponential index calculus and number field sieve algorithms.

Security Parameter

The Bitcoin network computes roughly 2^{70} hashes per second, which is about 2^{95} hashes per year. It would spend about 20 minutes on 2^{80} hashes.

RSA-1024 has only 80 bits of security, and 1024-bit Diffie-Hellman is within reach of a well-funded agency; see `https://weakdh.org`. At 2^{70} operations per second, 2^{128} operations still take 2^{58} seconds, which is about 9 billion years.

Quantum computers are not simply faster, but specific quantum algorithms forces us to re-evaluate the hardness of certain cryptographic problems.

Contents

Sources of Randomness

What are good sources for extracting randomness? Discuss!

Sources of Randomness

To generate *true* randomness, we need a source of entropy:

- temperature measurements
- acoustic noise
- air turbulence
- electromagnetic radiation

These sources (TRNGs) are hard to come by, to measure, and to analyze.

Random Numbers - Numberphile

Figure: https://www.youtube.com/watch?v=SxP30euw3-0

Sources of Randomness

What modern computers do today:

- keyboard timings
- mouse movements
- disk and network activity
- lava lamps (yes, really)

It is recommended to use more than one source. The operating system in modern computers usually mixes several of the above.

Sources of Randomness

Figure: https://www.cloudflare.com/en-gb/learning/ssl/lava-lamp-encryption

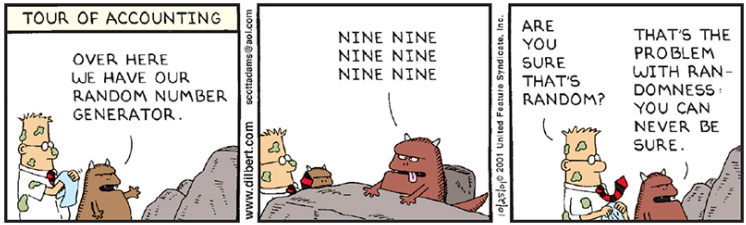
Bad Sources of Randomness

Sources that you should not use:

- the (exact) time of day, even in μs
- precomputed factory seed files
- process id or other environment variables
- whatever your “new friend” told you to use

Warning: the hard case is the first boot. Embedded devices and cloned virtual machines make keys before any entropy has been collected.

Sources of Randomness



Contents

Pseudorandom Number Generators

Pseudorandom Number Generators are deterministic algorithms that take as input a small sequence of *true* random bits and expand it into long sequences of *pseudorandom* bits streams.

A PRNG can perform three operations:

1. **init()** Initializes the internal state of the PRNG
2. **refresh(*R*)** Updates the state with randomness *R*
3. **next(*N*)** Returns *N* pseudorandom bits and updates the state

Security Concerns

We want the following security properties of a PRNG:

1. *forward secrecy*, or backtracking resistance, means that previously generated bits cannot be recovered from the state
2. *prediction resistance* means that future bits cannot be predicted, even after the state has been compromised

Under the Hood

Given a source of *true* randomness, the PRNGs we use today take that as seed and use symmetric ciphers (e.g. AES or ChaCha) or hash functions (e.g. SHA-2) to generate pseudorandom bits.

They are standardized as DRBGs in NIST SP 800-90A, and the entropy sources feeding them in NIST SP 800-90B.

Standardization is no guarantee: one of these generators turned out to contain a deliberate backdoor, which we return to later in the course.

Non-Cryptographic PRNGs

Be aware that most programming languages provide non-cryptographic PRNGs by default. These PRNGs output *random-looking* numbers that might be predictable given a few samples or by running statistical tests on the output.

Some classic non-cryptographic PRNGs that people use:

- *Mersenne Twister* (Python, Ruby, PHP, R, MATLAB,...)
- *Linear Congruential Generator* (Java, *rand* and *drand48* in libc,...)
- *Xorshift* and *PCG* (JavaScript engines, Rust `SmallRng`,...)

They are fine for simulations, but never for keys, nonces, or salts.

Non-Cryptographic PRNGs

The Go Blog

Secure Randomness in Go 1.22

Russ Cox and Filippo Valsorda
2 May 2024

Computers aren't random. On the contrary, hardware designers work very hard to make sure computers run every program the same way every time. So when a program does need random numbers, that requires extra effort. Traditionally, computer scientists and programming languages have distinguished between two different kinds of random numbers: statistical and cryptographic randomness. In Go, those are provided by `math/rand` and `crypto/rand`, respectively. This post is about how Go 1.22 brings the two closer together, by using a cryptographic random number source in `math/rand` (as well as `math/rand/v2`, as mentioned in our previous post). The result is better randomness and far less damage when developers accidentally use `math/rand` instead of `crypto/rand`.

Figure: `https://go.dev/blog/chacha8rand`

What to Actually Call

Ask the operating system, and let it do the seeding and reseeding:

- Linux, macOS, BSD: `getrandom()`, `getentropy()`, `/dev/urandom`
- Windows: `BCryptGenRandom()`
- Python `secrets`, Go `crypto/rand`, Java `SecureRandom`, Rust `OsRng`, libsodium `randombytes_buf()`

Never `time()`, `rand()`, `Math.random()`, a process id, or a seed you chose yourself, and never a fixed seed to make a test reproducible.
Hardware instructions such as `RDSEED` are one input among several, and not a replacement for the pool maintained by the operating system.

Contents

Schnorr Signatures

Let $\mathbb{G}_p$ be a group of prime order p and g be a generator for $\mathbb{G}_p$. Denote by pp the public parameters $(\mathbb{G}_p, g, p)$.

Let H be a cryptographic hash function outputting random elements in $\mathbb{Z}_p$.

Let the secret key $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p$ be sampled uniformly at random, and let the public key be $\mathsf{pk} = g^{\mathsf{sk}}$, where pk is made public.

Schnorr Signatures

The Schnorr signature of message m is computed as:

1. Sample uniformly random $r \leftarrow\!\!\$\ \mathbb{Z}_p$ and compute $R = g^r$.
2. Compute the hashed challenge $c = \mathsf{H}(\mathsf{pp}, \mathsf{pk}, m, R)$.
3. Compute the response $z = r - c \cdot \mathsf{sk} \mod p$. Output $\sigma = (c, z)$.

To verify the signature, compute $R' = g^z \cdot \mathsf{pk}^c$ and check if $c \stackrel{?}{=} \mathsf{H}(\mathsf{pp}, \mathsf{pk}, m, R')$. If correct, accept, and otherwise, reject.

Contents

ElGamal Encryption

Let $\mathsf{pp} = (\mathbb{G}_p, g, p)$ be as above. Let the secret key $\mathsf{sk} \leftarrow\!\!\$\ \mathbb{Z}_p$ be sampled uniformly at random, and the public key be $\mathsf{pk} = g^{\mathsf{sk}}$, where pk is made public.

The ElGamal encryption scheme, with message $m \in \mathbb{G}_p$, works as follows:

Enc: Sample uniform $x \leftarrow\!\!\$\ \mathbb{Z}_p$ and compute $\mathsf{ctx} = (X, Y) = (g^x, \mathsf{pk}^x \cdot m)$.

Dec: Decrypt the ciphertext $\mathsf{ctx} = (X, Y)$ as $m = Y \cdot X^{-\mathsf{sk}}$.

Contents

RSA Cryptosystem

Sample large random prime numbers p and q and compute product $n = p \cdot q$. Compute $\phi(n) = (p-1) \cdot (q-1)$.

Choose integer e (co-prime with $\phi(n)$) and compute d such that $e \cdot d \equiv 1 \mod \phi(n)$. Let $\mathsf{sk} = (p, q, d)$ be the secret key and $\mathsf{pk} = (n, e)$ the public key.

The RSA encryption scheme, with $m \in \mathbb{Z}_n$, works as follows:

Enc: Use padding scheme pad to compute $\mathsf{ctx} \equiv \mathsf{pad}(m)^e \mod n$.

Dec: Decrypt the ciphertext ctx to get $m = \mathsf{pad}^{-1}(\mathsf{ctx}^d \mod n)$.

Contents

Secure Hash Functions

When proving the security of cryptographic schemes that use a hash function H as an underlying building block, we often model H as a *random oracle*: a publicly available oracle that returns a fresh uniformly random value for every new input, and the same value whenever the input repeats.

In practice, a hash function is a public function that we can implement and analyze. The random oracle is an idealization of the behavior and security properties we want from the hash function.

You can read more about random oracles at Matthew Green's blog on cryptographic engineering: `https://blog.cryptographyengineering.com/2011/09/29/what-is-random-oracle-model-and-why-3`.

Contents

What to Remember

- Randomness is the foundation: keys, nonces, padding, and proofs all fail when it is weak, as we will see for the rest of the course
- Measure key material in *min-entropy*, not in Shannon entropy
- Aim for 128 bits of security; 80 bits can potentially be broken in minutes
- Use a TRNG for the seed and a CSPRNG for everything after that
- The dangerous moments are the first boot, cloned virtual machines, and any seed picked by a developer to make the output reproducible

Questions?