NTNU | Norwegian University of Science and Technology

RANDOMNESS 1: ENTROPY

TTM4205 – Lecture 2

Tjerand Silde

23.08.2024

Contents

Contents

Reference Group

We are looking for (at least) four students to form a reference group in this course, preferably students from different programs. We will meet three times during the semester, and your feedback is extremely valuable.

Thank you to Adrian Tokle Storset (MSTCNNS) and Daniel Nils Braun (exchange student) for volunteering already! We need at least one student from MTKOM, and preferable someone from SECCLO as well.

Send me an email and/or talk to me in the break :)

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.

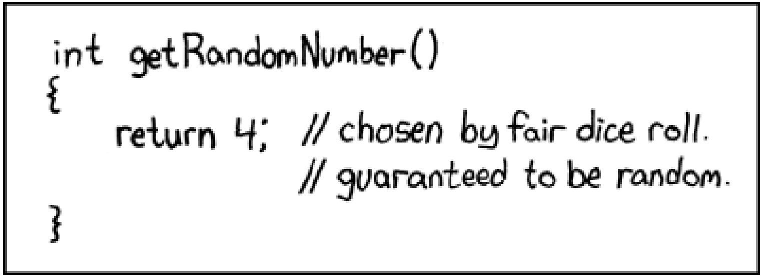
Let $p(x)$ be a probability distribution over $\mathcal{X}$ s.t. $p\colon \mathcal{X} \to [0, 1]$. We can e.g. assume that p is the uniform distribution over $\mathcal{X}$.

Let X be a random variable. The *(bit) entropy H* of X with respect to probability distribution p over alphabet $\mathcal{X}$ is defined as

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

.

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.

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.

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

$$-(1/3 \cdot -1.584 + 2/3 \cdot -0.584) = 0.92 \text{ bits.}$$

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
$$-(1/3 \cdot -1.584 + 2/3 \cdot -0.584) = 0.92 \text{ bits.}$$
- Let $\mathcal{X} = \{0, 1\}$ where $p(0) = 1/4, p(1) = 3/4$. The entropy is
$$-(1/4 \cdot -2 + 3/4 \cdot -0.42) = 0.81 \text{ bits.}$$

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
$$-(1/3 \cdot -1.584 + 2/3 \cdot -0.584) = 0.92 \text{ bits.}$$
- Let $\mathcal{X} = \{0,1\}$ where $p(0) = 1/4, p(1) = 3/4$. The entropy is
$$-(1/4 \cdot -2 + 3/4 \cdot -0.42) = 0.81 \text{ bits.}$$
- Let $\mathcal{X} = \{0,1\}^\lambda$, uniform distribution. The entropy is λ bits.

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 $\mathbb{G}_p$.

Elliptic curves are generic groups without more structure. We need groups of size 256 bits to provide 128 bits of security (equivalent to 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 (equivalent to AES-128).

Security Parameter

The computing power of the Bitcoin blockchain network is roughly 2^{60} operations per second. This is roughly 2^{85} operations per year.

RSA-1024 has only 80 bits of security. We think this has been breakable by powerful agencies like the NSA for at least ten years already.

We estimate that 2^{128} operations are infeasible even when using all computing power on Earth for the rest of the universe's lifetime. Quantum computers are not faster, but quantum algorithms might be more efficient.

Contents

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, measure, and analyse.

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*

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

Sources of Randomness

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

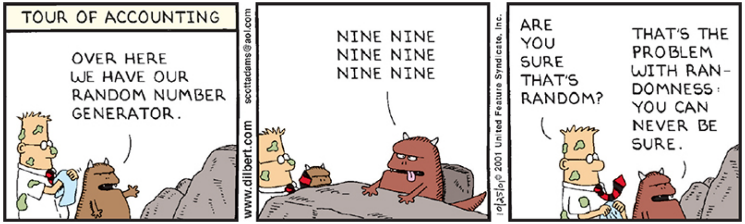
Bad Sources of Randomness

Sources that you should not use:

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

Warning: if you extract too much randomness within a short time frame, then the entropy of fresh samples might go down.

Sources of Randomness


TOUR OF ACCOUNTING
OVER HERE WE HAVE OUR RANDOM NUMBER GENERATOR.
NINE NINE NINE NINE NINE NINE
ARE YOU SURE THAT'S RANDOM?
THAT'S THE PROBLEM WITH RANDOMNESS: YOU CAN NEVER BE SURE.
scottadams@aol.com
www.dilbert.com
© 2001 United Feature Syndicate, Inc.
10/25/01

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 refresh

Security Concerns

We want the following security properties of a PRNG:

1. *forward secrecy* means that previously generated pseudorandom bits are impossible to recover
2. *prediction resistance* means that future pseudorandom bits are impossible to predict

Under the Hood

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

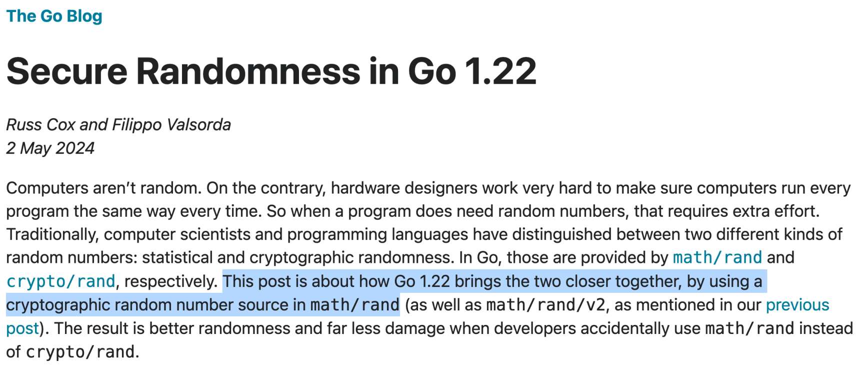
Non-Cryptographic PRNGs

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, PHP, Ruby, Pascal,...)
- *Linear Congruential Generator* (Java, Python, Rust,...)
- *rand* and *drand48* (libc), *rand* and *mt_rand* (PHP)

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`

Contents

Schnorr Signatures

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

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

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

Contents

ElGamal Encryption

Let $\mathsf{pp} = (\mathbb{G}, g, p)$ as above. 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.

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

Enc : Sample uniform $x \leftarrow\$ \mathbb{Z}_p$ and encrypt as $X = g^x$ and $Y = \mathsf{pk}^x \cdot m$.

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

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 μ to compute ciphertext $c \equiv \mu(m)^e \mod n$.

Dec : Decrypt the ciphertext c to get the message m as $\mu^{-1}(c^d \mod n)$.

Contents

Secure Hash Functions

When proving the security of cryptographic schemes that use hash functions *H* as underlying building blocks, we often model *H* as *random oracles* with an internal table of values.

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`

Questions?