(Applied) Cryptography

Week #2: Randomess and Cryptographic Security

Manuel Barbosa, mbb@fc.up.pt

MSI/MCC/MERSI – 2022/2023

DCC-FCUP

Part #0: Cryptographic keys

Where do keys come from?

For symmetric crypto:

- Generated uniformly at random and pre-shared, or
- Derived using a key derivation function (KDF):
  - From a password or low entropy secret
  - From a high-entropy *master key* from key exchange protocol

For asymmetric crypto:

- Key generation algorithm: uniform random coins $\Rightarrow$ key pair
- Private key holder generates both keys; publishes public key

Asymmetric keys are typically much larger:

- RSA keys take roughly 4096-bits for 128-bit security
- Elliptic-curve keys take roughly 400-bits for 128-bit security

Where are keys generated and stored?

Ideally in a secure hardware element:

- Hardware Security Module (HSM)
- Smartcard or other crypto token

Long-term keys are often **wrapped** before (standard) storage:

- Wrapping means encrypted with another key
- Password-Based Encryption (low security)
- Wrap with HW-protected master key (standard-security)
- (Master) key directly stored in secure hardware (high-security)

What are we talking about really?

What is it?

- **To generate uniformly at random**?
- **High/low entropy**?
- **Uniform random coins**?
- **128-bit security**?
- A key derivation function?
- A master key?
- A key exchange protocol?
- RSA and Elliptic curves?
- ...

Today we will cover the **bold** questions.

Understanding these is crucial before we move on to the other ones.

Part #1: Randomness?

Introduction

Is this bit string random?

- 00000000

What about this one?

- 10101010

And this one?

- 001010111

"Randomness" is not a property of a bit string.

It is a property of

- a bit string **generation process**, i.e.,
- a bit string **sampling procedure**

Randomness Distributions

We describe randomized processes using *randomness distributions*.

Everything starts with the **uniform distribution** over a finite set S.

A process U samples from the uniform distribution if

$$\forall s^* \in S, \Pr[s = s^* : s \twoheadleftarrow U] = \frac{1}{|S|}$$

For bit strings of length λ:

- Every string occurs with probability $\frac{1}{2^\lambda}$
- Can be implemented by sampling each bit as a perfect coin

Randomness Distributions (2)

We can think of an arbitrary distribution X as follows:

- Fix a randomness space $R := \{0,1\}^\lambda$
- Sample coins C uniformly at random from R
- Compute output deterministically as $s = X(C)$

The probability of any output s^* can be computed as

$$\Pr[s = s^* : s \twoheadleftarrow X] = \frac{\#\{C \in \{0,1\}^\lambda : X(C) = s^*\}}{2^\lambda}$$

Entropy

We will mostly use "entropy" as an intuitive concept:

- It measures the **uncertainty** wrt to a sampling output

Mathematically it can be defined as for a distribution X as:

$$H(X) := \sum_{s^* \in S} - \Pr[s^*] \cdot \log \Pr[s^*]$$

This definition is maximized by the uniform distribution, which has entropy λ.

It is minimized by the constant distribution, which has entropy 0.

This definition relates the uncertainty of any distribution X to that of the uniform distribution over $\{0,1\}^{H(X)}$.

Random Number Generators (RNG)

How do we get uniform random coins?

Everything starts with a physical process:

- a source of entropy, e.g., some natural process that is believed to sample ℓ-bit strings from a high-entropy distribution
- typically $\ell \gg \lambda$, where λ is the assumed entropy
- randomness extractors (often a hash function) compress such bit strings down to λ bits
- the resulting bit strings are assumed to be uniform

This combined process is called a Random Number Generator.

High-security RNGs currently exploit quantum effects.

Pseudorandom Number Generators (PRG) in Theory

Good randomness is hard to come by, so RNGs are typically slow.

PRGs are crypto's response to this problem:

- PRG takes a small uniform random seed of length λ
- Generates long, random-looking bit strings $\ell \gg \lambda$

PRGs are deterministic algorithms!

An attacker must be incapable of distinguishing PRG(s) from a truly random string (not knowing s).

Stateful random number generation in operating systems

In modern OS randomness generation is stateful:

- PRG keeps a state
- OS mixes output of entropy source into PRG state

Such PRG constructions extract and expand randomness:

- $st \leftarrow \textbf{init}(\,)$ SO initializes state
- $st \leftarrow \textbf{refresh}(R, st)$ SO adds entropy (reseeds)
- $(C, st) \leftarrow \textbf{next}(N, st)$ SO returns N random bits

In some cases the OS tries to estimate the amount of entropy and blocks if not enough.

Security properties of stateful PRG

Security needs to consider **state compromise**.

Backtracking resistance:

- Suppose adversary corrupts PRG state
- Past randomness should not be compromised
- Aka forward secrecy (for past secret keys)

Prediction resistance:

- Suppose adversary corrupts PRG state
- SO adds extra (hidden) entropy to PRG state
- Future output should look random once more
- Hence **refresh** must be called regularly

A perfect entropy estimator would be great for security: in practice they are not reliable.

Caution: statistical tests are not sufficient

Statistical tests:

- count number of 1/0
- check distribution of 8-bit words
- ...

Irrelevant for security:

- possible to pass statistical tests
- totally insecure for crypto

Cryptographic PRGs come with a **proof of security**:

- if attacker breaks PRG (e.g., predicts)
- we can use it to perform (assumed) impossible computation

Linux systems

The PRG is accessible at /dev/urandom:

- in *nix-style, PRG is mapped to a file
- careful to make sure system call successful

Link to code from LibreSSL.

In some *nix variants there is a blocking /dev/random based on an entropy estimator.

The current consensus seems to be that for most applications this is not useful; see, e.g., **this link**.

Part #2: Quantifying Cryptographic Security

Introduction

Cryptographic security is mathematically quantified.

This is different from other security analyses:

- Often security is about eliminating known vulnerabilities
- The attack model is not formally defined

Cryptographic security is about *defining the impossible*.

Impossible is a relative notion:

- Fix the power of relevant attackers (data + computation)
- Fix a small enough advantage/probability measure ϵ

Impossible means no attacker of given power succeeds with advantage/probability above *epsilon***.**

One-Time-Pad (the past)

Perfectly secure encryption scheme:

- Key is a random permutation over the entire plaintext space
- Key is only used once

How can one implement such a permutation?

- Same as Vigenère but $\ell =$ plaintext length

When working with bits:

- Key is a random bit string of the same size as plaintext
- To encrypt/decrypt, perform bit-wise XOR

Why is it *perfectly secure*?

- Ciphertext is distributed uniformly at random if key is hidden
- All plaintexts are equally likely
- Even *unbounded* attacker cannot learn anything about plaintext

One-Time-Pad (the present)

Can't be generally used in practice

- Huge keys that can only be used once
- How does one pre-share and store such keys?

Still, idea of the one-time-pad is still central to modern crypto:

- Prove that a bit string is "as good as" a random string
- Use the bit string instead of a one-time-pad key

Today's encryption schemes

A "good" keyed permutation is designed:

- it is called a *block cipher* or *pseudorandom permutation*
- large input size (unrelated to plaintext size), 128 or 256 bits
- large key space (unrelated to plaintext size), 128 or 256 bits

Encryption:

- gradually transform plaintext by
- applying an iterative process that
- uses the permutation and the key repeatedly

This is called a *mode of operation*.

Ideal (Block) Cipher

What is a "good" permutation?

The ideal cipher is a permutation sampled uniformly at random.

Why can't we use that instead of a keyed algorithm?

- Huge table, cannot be compressed into compact description

Pseudorandom permutations aka block ciphers:

- Compact public description
- Efficiently computable
- Small secret key (still large key space)
- If key is hidden, outputs "look like" from random permutation

Security in Practice

Redefine impossible to break:

- With **reasonable resources** (time, memory, HW power)
- With probability higher than **negligible**

Practical schemes are **computationally impossible** to break.

E.g., take a block cipher and an attacker that does not know K

- attacker chooses non-repeat inputs X_i and gets
  - Y_i chosen uniformly at random if $b = 1$
  - $Y_i = \mathbf{E}(K, X_i)$ if $b = 0$
- attacker guesses b' and we define ϵ as

$$|\Pr[b' = 1|b = 1] - \Pr[b' = 1|b = 0]|$$

Best attack for $\epsilon = 2^{-40}$ takes 2^{80} steps.

Security in Practice (2)

Some numbers for scale:

- The estimated age of the universe in nanoseconds is around 2^{88}
- Event occurs once every 10000 years: approx. 2^{-16}

Estimated computational power today:

- Very entertaining reading **in this link**

For practical schemes we usually say (t, ϵ)-security:

- For some well-defined attack model
- Any attacker running in at most t steps
- Has at most ϵ success advantage/probability

What if attacker can do $t + 1$ steps?

Security in Practice (3)

When we have (t, ϵ)-security:

- t is a lower-bound on the work needed to break the scheme

E.g., what is the best possible block cipher with key space 2^{128}?

- Best attack should be brute-forcing key
- This means $(t, t/2^{128})$ security
  - If I try all keys in $t = 2^{128}$: $\epsilon = 1$
  - If I try one key in $t = 1$: $\epsilon = 1/2^{128}$
  - If I try a fraction of the keys $t = 2^{64}$: $\epsilon = 2^{64}/2^{128}$

Note that no real-world cipher can be better than this! (Why?)

Quantifying Security

Lower bound on the amount of work required for a successful attack.

Number of steps of the best attack:

- n-bits security means
- best attack to break the scheme requires 2^n steps

In encryption:

- n bit keys cannot give more than n-bit security
- brute-force attack allows finding the correct key

In general:

- ℓ bit keys could lead to n-bit security where $n \ll \ell$
- Best attack is more efficient than brute-force attack

Quantifying using n-bit security permits comparing schemes.

What bit-security should we use?

Today, 128-bit security is the norm:

- e.g., designs target best attacks at $(t, \epsilon) = (2^{88}, 2^{40})$

For how long do we need security to hold?

- Moore: computational power doubles each 2 years
- This means one $n + 1$ bit security every 2 years
- This no longer seems to be true, but ...
- Maybe we will have quantum computers

For long-term security:

- Defense in depth (e.g., onion encryption) + 256-bit keys

For very short-term security:

- Shorter, e.g., 80-bit keys may be OK

Part #3: Achieving Cryptographic Security

Security assurance

How do we guarantee n-bit security?

- **Heuristic security**
  - Large community constantly trying to break scheme
  - I.e., cryptanalysts try to disprove n-bit security
- **Provable security**
  - Mathematical proof
  - Breaking a scheme implies solving hard problem

Provable security

Assume we believe mathematical problem P is impossible to solve.

Suppose we want to prove breaking scheme C is impossible.

We build a *reduction*:

- Take any (hypothetical) attacker $\mathcal{A}$ that breaks C
- Construct (concrete) reduction $\mathcal{B}^{\mathcal{A}}$
- I.e., $\mathcal{B}$ uses $\mathcal{A}$ as a subroutine.
- $\mathcal{B}$ solves P whenever $\mathcal{A}$ succeeds.

Corollary:

- If no successful $\mathcal{B}^{\mathcal{A}}$ can exist (P cannot be solved)
- Then $\mathcal{A}$ cannot exist (we know $\mathcal{B}$ exists)
- So scheme C must be secure

Provable security (caveats)

Problem P is called a *hardness assumption*:

- It can be a mathematical problem such as factoring
- It can be some other cryptographic construction

Proof assurance $\leq$ Assumption assurance:

- Proofs of security are relative to assumptions
- Security only holds if assumptions are true

All practically used assumptions are validated via heuristic security.

Heuristic security

Validating hardness assumptions is crucial for modern cryptography.

Methodology for heuristic security has been progressing:

- Standards take years to define
- Competitions where proposals are scrutinized
- "My construction wins if I break your construction"
- AES, SHA-3, post-quantum, encryption, etc., competitions

Thank you!

mbb@fc.up.pt

http://www.dcc.fc.up.pt/~mbb