Cryptology complementary

✧

Introduction

Pierre Karpman
pierre.karpman@univ-grenoble-alpes.fr
https://www-ljk.imag.fr/membres/Pierre.Karpman/tea.html

2018–02–08

First things first

Main goals of this course: "practical" complement to the other half

- Introduce some constructions (what's a block cipher, a key exchange?...)
- Introduce some implementation aspects (how do you do finite field arithmetic?...)
- Introduce some attacks (how do yo compute a discrete logarithm?...)
- Introduce some real-life usage (e.g. SSH)

Organisation

- Course format: mix of lectures/TDs/TPs
- (Probably) A contrôle continu evaluation (lab session/homework, details T.B.D.)
- A final exam (ditto)

What's crypto?

Quick answer: it's about protecting secret data from *adversaries*

- In a communication (encrypted email, text messages; on the web; when paying by credit card)
- On a device (encrypted hard-drive)
- During a computation (online voting)
- Etc.

Where does crypto run?

Crypto needs on various platforms

- High-end CPUs (Server/Desktop/Laptop computers,...)
- Mobile processors (Phones,...)
- Microcontrollers (Smartcards,...)
- Dedicated hardware (accelerating coprocessors, cheap chips,...)

Techno constraints

Varying contexts, varying requirements

- Speed (throughput)
- Speed (latency)
- Code/circuit size
- Energy/power consumption
- Protection v. physical attacks

⇒ Implementation plays a big part in crypto

Quick example

A protocol (e.g. TLS) uses among others

- A key exchange algorithm (e.g. Diffie-Hellman) — "public-key" cryptography
  - instantiated with a secure group (e.g. ANSSI FRP256V1)
- An authenticated-encryption mode of operation (e.g. GCM) — "symmetric-key" cryptography
  - instantiated with a secure block cipher (e.g. the AES)
- A digital signature algorithm (e.g. ECDSA) — "public-key" + "symmetric-key" cryptography
  - instantiated with a secure group and a secure hash function (e.g. SHA-3)

Protocols can be complex

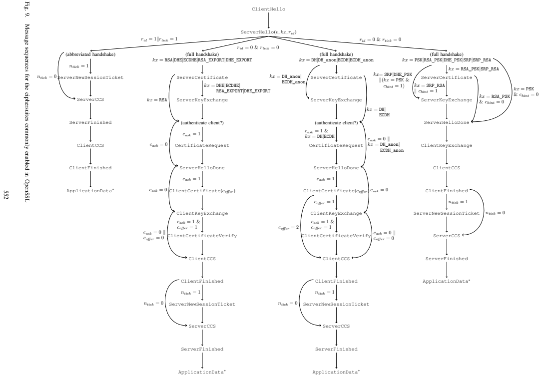

Figure: Part of the TLS state machine, Beurdouche et al., 2015

“Doing crypto”

- Designing new primitives/constructions(/protocols)
- Analysing existing primitives/...
- Deploying crypto in products

- Different goals, different techniques
  - Ad-hoc analysis, discrete mathematics, algorithmics
  - Computational number theory/algebraic geometry
  - Low-level implementation (assembly, hardware)
  - Formal methods
  - Following “good practice”

Scope of an analysis

Many types of adversary

- Passive ("eavesdropper = Eve")
- Not passive, i.e. active
- With or w/o physical access
  - Side channels
  - Fault attacks
- With varying scenarios ("one-time" or long-term secret?)
- With varying objectives

Security objectives?

Security objectives?

- Hard to find the secret (the key)
- Hard to find the message (confidentiality)
- Hard to change/forge a message (integrity/authenticity)
- Etc.

Small informal focus

Example: indistiguishability (IND-CPA)

1. Submit messages to an *oracle* $\mathfrak{O}$ to be encrypted, & get the result
2. Choose, m_0, m_1, send both to $\mathfrak{O}$
3. Receive $\mathfrak{O}(m_b)$ for a random $b \in \{0, 1\}$
4. Goal: determine the value of b (better than by guessing)

- $\mathfrak{O}$ has to be *randomized*

A code that's not IND-CPA

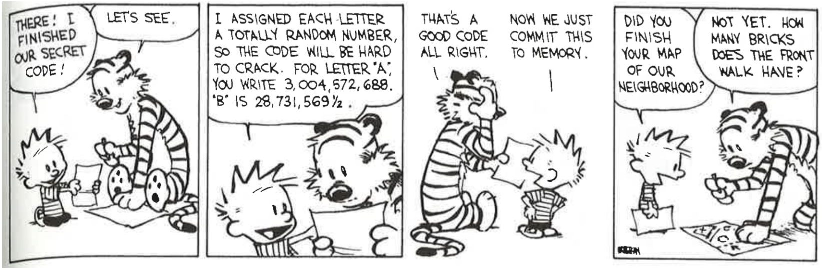

Figure: Calvin & Hobbes' code (Watterson)

Randomness is key in crypto

Random numbers always needed

- To generate keys
- To generate *initialization vectors* (IVs) or *nonces*
- To generate random masks (to protect against some attacks)
- Etc.

Random number generation is not easy

Lead to severe vulnerabilities, several times. For instance:

- Debian's OpenSSL key generation (2006–2008)
- WWW RSA private keys with shared factors (Lenstra et al., 2012)
- Smartcard RSA w/ biased private keys (Bernstein et al., 2013)
- Smartcard RSA w/ biased private keys (Nemec et al., 2017)

How not to generate random numbers

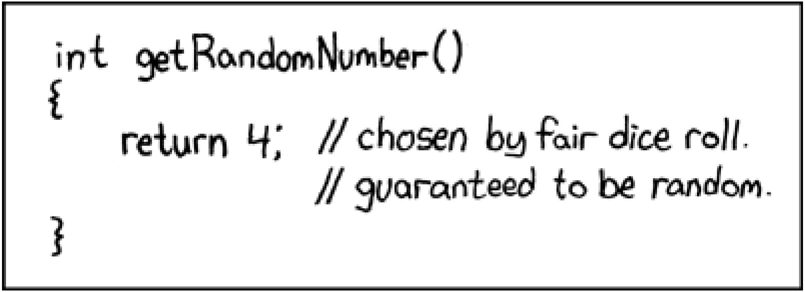

Figure: XKCD's PRNG (Munroe)

How not to generate random numbers

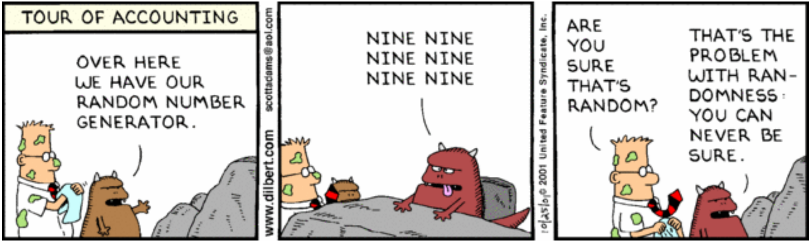

Figure: Dilbert's PRNG (Adams)

Terrible Kolmogorov complexity!

How to generate them, then?

A basic idea (e.g. `/dev/random`)

- Set up a “random” state (from e.g. physical sources)
- Refresh it continuously as randomness comes by
- Extract and refresh when outputs are needed

Random numbers are all you need?

A "perfect" encryption scheme, the one-time pad

1. Let the message m be in $\{0,1\}^n$, n maybe large (say, 2^{40})
2. Let the key k be $\xleftarrow{\$} \{0,1\}^n$
3. The ciphertext $c = m \oplus k$

- Knowing c does not give information about m (Exercise)

Problems:

- Integrity not guaranteed
- Needs very large keys
- Needs "perfect" randomness too!

⇒ Later, we'll see how to solve such issues practically

Keys: secret, private, public...

What are crypto keys like?

- Stream/Block cipher: a binary string
- Hash functions: $\varnothing$
- RSA: a prime number (secret), an integer (public)
- Diffie-Hellman: an integer (secret), a group element (public)
- Code-based: a (generating) matrix (of a code) (one secret, one public)
- Etc.

Secrets large and small

What should the secret/public key size be (in bits)?

- Stream ciphers?
- Block ciphers?
- RSA?
- Diffie-Hellman (well-chosen $\mathbf{F}_q^*$)?
- Diffie-Hellman (well-chosen $E(\mathbf{F}_q)$)?
- Code-based (McEliece, Binary Goppa codes)?

Secrets large and small

What should the secret/public key size be (in bits)?

- Stream ciphers: e.g. 128 bits (+ a large (e.g. 128 bits) IV necessary)
- Block ciphers: e.g. 128 bits
- RSA: e.g. 3072 bits
- Diffie-Hellman (well-chosen $\mathbf{F}_q^*$): e.g. 3072 bits
- Diffie-Hellman (well-chosen $E(\mathbf{F}_q)$): e.g. 256 bits
- Code-based (McEliece, Binary Goppa codes)? e.g. 200 000 *bytes*

Secrets large and small

What should the secret/public key size be (in bits)?

⇒ Quite a complex matter! (Follow recommendations, e.g. from ANSSI!)

What's 128 bits anyway?

Objective: run a function 2^{128} times within 34 years ($\approx 2^{30}$ seconds), assuming:

- Hardware at 2^{50} iterations/s (that's pretty good)
- Trivially parallelizable (that's not always the case in practice)
- 1000 W per device, no overhead (that's pretty good)

$\Rightarrow$

- $2^{128-50-30} \approx 2^{48}$ machines needed
- $\approx 280\,000\,000$ GW 'round the clock
  - $\approx 170\,000\,000$ EPR nuclear power plants

(Of course, technology may improve, but here's quite a safe margin)