(Applied) Cryptography

Week #8: Hard Problems and Public-Key Cryptography

Manuel Barbosa, mbb@fc.up.pt

MSI/MCC/MIERSI – 2020/2021

DCC-FCUP

Part #1: Complexity Theory (Very) Basics

Why does cryptography use complexity theory?

What does it means for a problem to be hard?

- We know of no way to solve it?
- Someone showed there is no way to solve it?
- Someone showed there is no way to solve it *efficiently*?

Surely all *small enough* problems can be solved. (How?)

Why does cryptography use complexity theory?

What does it means for a problem to be hard?

- We know of no way to solve it?
- Someone showed there is no way to solve it?
- Someone showed there is no way to solve it *efficiently*?

Surely all *small enough* problems can be solved. (How?)

Complexity theory looks at problems as the size of instances grows:

- **Easy problems**: efficient solution for all input sizes
- **Hard problems**: best known solution inefficient for moderate input sizes
- **Hardest problems**: if we can solve these, we can solve all the hard ones

What does *efficient* and *inefficient* mean?

Execution time

We define efficient/inefficient in a relative way:

- **Efficient**: degrades slowly as input grows
- **Inefficient**: degrades quickly as input grows

Size of input := size it takes in memory (bits).

Example:

- For loop that prints all k-bit numbers
- Try to run it in your computer.
- Runs in time $2^k * \text{op}$
- Where op is machine-specific.

Execution time

We define efficient/inefficient in a relative way:

- **Efficient**: degrades slowly as input grows
- **Inefficient**: degrades quickly as input grows

Size of input := size it takes in memory (bits).

Example:

- For loop that prints all k-bit numbers
- Try to run it in your computer.
- Runs in time $2^k * \text{op}$
- Where op is machine-specific.

Brute-force search is an exponential-time algorithm.

Execution time

We define efficient/inefficient in a relative way:

- **Efficient**: degrades slowly as input grows
- **Inefficient**: degrades quickly as input grows

Size of input := size it takes in memory (bits).

Example:

- For loop that prints all k-bit numbers
- Try to run it in your computer.
- Runs in time $2^k * \text{op}$
- Where op is machine-specific.

Brute-force search is an exponential-time algorithm.

Super-polynomial algorithms: inefficient in **any** computer.
(Quantum computers do not exist yet.)

Execution time (2)

Super-polynomial-time algorithms:

- small inputs (e.g., 60-bit keys)
- reach the limit of feasible computation, say 2^{60}.

Security in crypto: best (known) attack at least super-polynomial.

Execution time (2)

Super-polynomial-time algorithms:

- small inputs (e.g., 60-bit keys)
- reach the limit of feasible computation, say 2^{60}.

Security in crypto: best (known) attack at least super-polynomial.

Useful algorithms execute in polynomial-time.

Say, e.g., op $* k^3$ at most.

Encryption, decryption, signing, etc.:

- They all should execute in small poly time
- If we increase key size: slightly slower
- Best attack: exponentially harder

Notation

Algorithm is $\mathcal{O}(2^n)$:

- not worse than exponential (really bad)

Algorithm is $\mathcal{O}(n^c)$:

- cannot be excluded as feasible

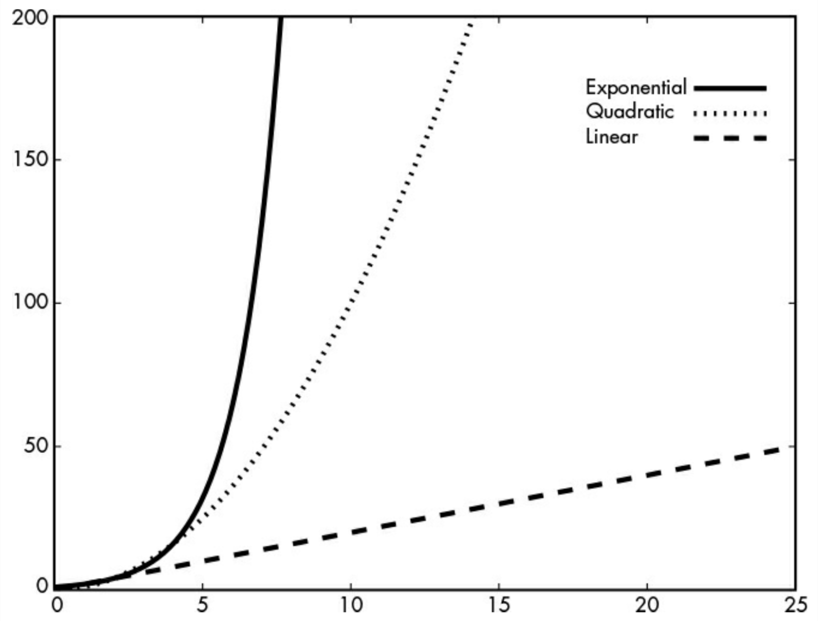
Algorithm is $\mathcal{O}(n)$:

- not worse than linear (really good)

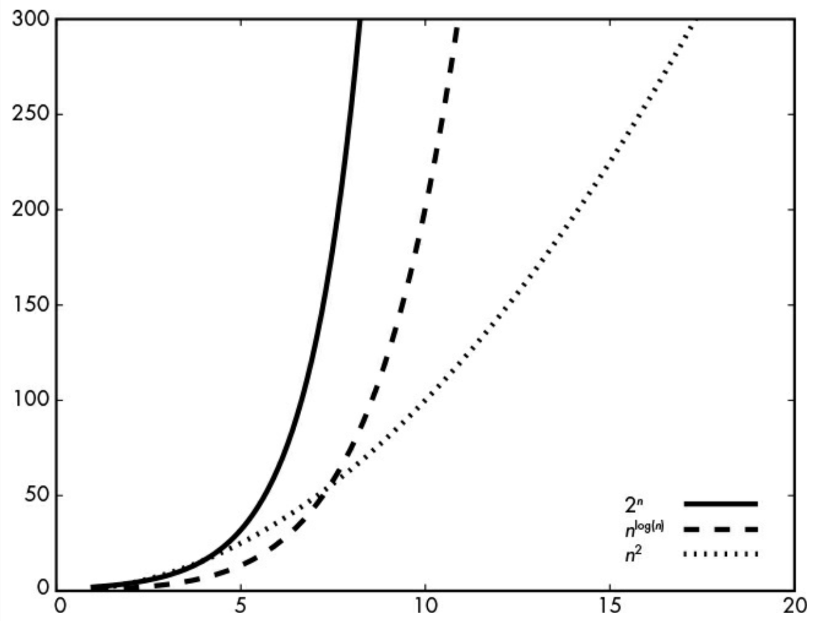
Algorithm is $\mathcal{O}(1)$:

- constant, i.e., does not depend on input size

Execution time in a plot



Super-polynomial time in a plot



Should we use the hardest problems in crypto?

Crypto is built on top of computational assumptions:

- problems that no one knows how to solve efficiently
- problems that we believe are very hard to solve

Should we use the hardest problems in crypto?

Crypto is built on top of computational assumptions:

- problems that no one knows how to solve efficiently
- problems that we believe are very hard to solve

Complexity theory has a classification system for problems:

- Class P: poly-time algorithm solves it (using polynomial space)
- Class NP: non-deterministic poly-time algorithm solves it
  - Enough that one can check correct solution in poly-time
  - All problems in P are in NP

Should we use the hardest problems in crypto?

Crypto is built on top of computational assumptions:

- problems that no one knows how to solve efficiently
- problems that we believe are very hard to solve

Complexity theory has a classification system for problems:

- Class P: poly-time algorithm solves it (using polynomial space)
- Class NP: non-deterministic poly-time algorithm solves it
  - Enough that one can check correct solution in poly-time
  - All problems in P are in NP

Some problems in NP are special (NP-complete):

- They are as hard as any problem in NP
- If they can be solved in poly-time, then P=NP
- Most important open question in Computer Science?

Should we use the hardest problems for crypto?

Should we be building crypto on NP-complete problems?

Not so easy:

- Complexity theory: worst-case complexity (some inputs)
- Crypto: problems hard to solve for most inputs (average-case)

Should we use the hardest problems for crypto?

Should we be building crypto on NP-complete problems?

Not so easy:

- Complexity theory: worst-case complexity (some inputs)
- Crypto: problems hard to solve for most inputs (average-case)

Problems underlying today's crypto not known to be NP-complete.

But, same problems can be efficiently solved using quantum computers!

Should we use the hardest problems for crypto?

Should we be building crypto on NP-complete problems?

Not so easy:

- Complexity theory: worst-case complexity (some inputs)
- Crypto: problems hard to solve for most inputs (average-case)

Problems underlying today's crypto not known to be NP-complete.

But, same problems can be efficiently solved using quantum computers!

Post-quantum cryptography introduced great changes:

- Lattice-based crypto uses new computational assumptions
- Average-case hardness related to worst-case hardness
- Underlying problems closely related to NP-complete problems

Part #2: Some hard problems used in crypto

Factoring

Consider the following integer generation algorithm:

$$\begin{aligned} &p \twoheadleftarrow \mathrm{RandomPrime}(\lambda) \\ &q \twoheadleftarrow \mathrm{RandomPrime}(\lambda) \\ &N \leftarrow p \cdot q \\ &\text{Return } N \end{aligned}$$

Here RandomPrime is an algorithm that samples random primes:

- of growing bit length as security parameter λ grows
- For example $\lambda = 128$ yields 2048-bit primes

The factoring assumption states that:

- The best algorithm for finding (p, q) given N as above
- Executes in super-polynomial time in λ

What we know about factoring

Not believed to be NP-complete.

Believed not to be in P (yet a quantum computer could factor).

Best known factoring algorithm:

- General Number Field Sieve uses advanced mathematics
- Executes in $\approx \exp(1.91 \times n^{1/3} (\log n)^{2/3})$
- $\exp =$ exponential function; n is the bit-length of N

What this means:

- Factoring 1024-bit N as above $\Rightarrow 2^{70}$ steps
- Factoring 2024-bit N as above $\Rightarrow 2^{90}$ steps
- Done in 2005: 663 bit number in 18 months
- Done in 2009: 768 bit number in 24 months
- 1024-bit numbers believed to be within reach in 2020

RSA problem (Rivest, Shamir, Adleman 1977)

Let e be a fixed (small) prime number, typically 0x10001.

Let N be an integer generated as in the previous slides.

RSA function for $x \in \mathbb{Z}_N^*$ is: $\mathsf{RSA}(N, e, x) := x^e \mod N$.

RSA function can be efficiently inverted if factorization is known:

$$\begin{aligned}
&\mathsf{Invert}(y):\\
&\quad d \leftarrow e^{-1} \mod \Phi(N)\\
&\quad x \leftarrow y^d \mod N\\
&\quad \text{Return } x
\end{aligned}$$

Here

- $\Phi(N)$: number of integers in $\mathbb{Z}_N^*$ co-prime to N
- In this case we have $\Phi(N) = (p–1)(q–1)$
- d is such that $e \cdot d = 1 \mod \Phi(N)$
- This means that $x^{e \cdot d} \mod N = x$

RSA problem (2)

Consider the following *one-wayness* experiment:

- Fix e and sample N as above
- Sample x uniformly at random in $\mathbb{Z}_N^*$
- Compute $y \leftarrow \mathrm{RSA}(N, e, x)$
- Run adversary on (N, e, y)
- Adversary wins if it outputs $x' = x$

Note attacker is trying to compute an e-th root modulo N.

Attacker could win by factoring N and inverting as above.

This is believed to be the best attack against RSA.

One of the most widely used problems in cryptography
(Why?)

RSA trapdoor permutation

RSA is intrinsically an asymmetric function:

- Publish (N, e) as public key
- Keep (N, e, d, p, q) as private key
- Everyone can compute (your instance of) the RSA function
- Only you can compute its inverse
- This is called a **trapdoor**

The RSA function can be used to construct:

- Digital signature schemes
- Public-key encryption schemes
- Key agreement protocols
- Authentication protocols

Everything that can be constructed from **one-way trapdoor permutations**.

Discrete Logarithms

Take any (finite) group:

- Set of elements $\mathcal{G}$
- Group operation $\circ : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$
- $\circ$ is closed, associative (, commutative)
- Neutral element $\mathbf{1}$ and all elements have an inverse

Some elements are *group generators*: g generates $\mathcal{G}$ if

$$\mathcal{G} = \left\{ g^k \mid k \in [\, 0 \ldots |\mathcal{G}| - 1 \,] \right\}$$

If a generator exists, the group is called **cyclic**.

The discrete logarithm problem is defined as:

- choose k uniformly at random in $[\, 0 \ldots |\mathcal{G}| - 1 \,]$
- ask adversary to find k given only g^k

Discrete Logarithms

The discrete logarithm problem is trivial in some groups:

- Fix a large prime number p
- Take the additive group in the ring of integers modulo p
- *Exponentiation* in this group is integer multiplication modulo p
- Extended Euclidean algorithm computes division efficiently
- So the discrete logarithm problem is easy in this group

We do not know how to solve it in other groups:

- Large prime number $p = k * q + 1$ for q large prime
- Choose g that generates multiplicative subgroup of size/order q
- For such primes, $g = 2$ works (see here for discussion on sizes)

We will cover the presently most popular DL groups in crypto:

- $\mathcal{G}$ is the set of points in an Elliptic Curve
- Group operation: simple/efficient formulae over coordinates

DH Problem (Diffie, Hellman 1976)

Closely related to the DL problem, but more flexible.

Computational version (CDH):

- Choose x, y uniformly at random in $[\,0 \ldots |\mathcal{G}| - 1\,]$
- Give g^x and g^y to the adversary
- Adversary must find g^{xy}

Decisional version (DDH):

- Choose x, y, z uniformly at random in $[\,0 \ldots |\mathcal{G}| - 1\,]$
- Choose a random coin b
- Set $w = xy$ if $b = 0$ and $w = z$ if $b = 1$
- Give g^x, g^y and g^w to the adversary
- Adversary must output a guess for b

On assumption strength

Is it more plausible to believe factoring is hard or RSA is hard?

- If you can factor you can solve RSA
- But there could be an easier way to solve RSA
- So it is more plausible that factoring is hard
- We say factoring is a weaker assumption

Is it more plausible to believe DL is hard or CDH is hard?

- If you can solve DL you can solve CDH
- But there could be an easier way to solve CDH
- We say DL is a weaker assumption

How about CDH vs DDH?

On assumption strength (2)

We would like to design constructions based on weaker assumptions.

All assumptions should be heuristically validated.

Usually:

- Stronger the assumptions are more complex
- They are harder to validate heuristically
- Resulting cryptosystems are more efficient

The strongest assumption (absurdum):

- Validate the entire cryptosystem heuristically

Modern crypto:

- Best cryptosystem uses the weakest assumption
- Proof shows that assumptions imply security

Key lengths and bit-security

The best attack on DL/CDH is also based on the GNFS algorithm.

This means that key sizes are essentially the same as RSA.

Disadvantage:

- generating a DH modulus slower than generating RSA modulus. (Why?)

Advantage:

- many users can share the same group parameters
- indeed, some sets of such parameters are standardized

Standardizing parameters creates a fixed and high-profile target:

- pre-computation over many years may hurt long-term security
- for dynamically generated ones: honest generation guarantee

How things go wrong (common misconceptions)

Small enough problems are easy:

- Factoring a 512-bit number was hard in the 70s but not now
- Similarly, discrete logs are easy modulo p for 512-bit prime

Some large numbers are easy to factor:

- if N has small factors (smooth)
- if N_1 and N_2 are RSA moduli and share a prime factor

In some groups DDH is easy whereas CDH remains unsolved:

- Such groups should be used with care

Thank you!

mbb@fc.up.pt

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