Cryptograph Fundamentals

Cryptography is the study of the scenario where we have:

1. A set of honest parties (ex: ourselves)
2. A set of adversaries (ex: trying to steal from you)
3. A goal
4. A set of assumptions (ex: what can the adversary do? More importantly, what can we do securely - our model of the world)

A lot of cryptography is: the study of what set of assumptions
allow us
to build a protocol to achieve our goal
despite potential adversaries

(goal) a common language to talk about *how secure* something is

(define) bits of security

Key size	# combinations
1	2^1
2	2^2
32	2^{32} (≈4.2 billion)
256	2^{256} (≈ # atoms)
128	2^{128}

← still very secure, and faster than 256

(define) security parameter (κ)

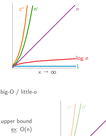
(goal) map an algorithm's security into # of bits of security

ex: factorization of an n-digit number
has a different # of bits of security depending on:
- classical adversary
- quantum adversary

allows to update mapping as tech improves { better computers, better techniques

(define) function complexity

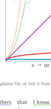
(goal) classify functions based on behavior as $x \to \infty$



- 2^x — exponential (ex: 2^n)
- x^2 — polynomial (ex: n^2)
- n — linear (ex: 3n)
- $\log n$ — logarithmic (ex: $\log(3n)$)
- 3 — constant (ex: 5)

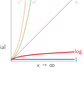
(define) big-O / little-o

upper bound
ex: $O(n)$



strict upper bound
ex: $o(n^2)$

pronounced "sub"
ex: $o(n^2)$ = subpolynomial



TODO: *merge into the complexity explainer file, or link it from here*

(simple example) I want to prove to others that I know a specific secret value without revealing it
where nobody can recover the message or forge a fake proof

want: a way to convert secret value → publicly shareable value

(idea) we will publicly share the hash of our secret

(define) Hash Algorithm
a function that maps arbitrary data (often arbitrary size) → the pre-image to numerical output (often small, fixed size) → the digest

ex: assign every letter a number

A → 0
B → 1
⋮
Z → 26

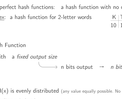
in practice, there are standards for this like ASCII tables

then, add all numbers in a text together (ex: "AB" = 0 + 1)

note: multiple strings may give the same result (a "collision")
"AB" = 0 + 1 = 1
"BA" = 1 + 0 = 1

however, many hash functions are optimized for efficiency, simplicity over security
or other properties (ex: avalanching)

ex: distribution may not be very random



perfect randomness — example hash function

in fact, some are even *explicitly designed* to be reversible

(define) perfect hash functions: a hash function with no collisions
ex: a hash function for 2-letter words

K	T
10	19

← never collides, but doesn't scale to large inputs

(define) Cryptographic Hash Function
a hash function with *a fixed output size* → n bits output → *n bits of security*

with properties:
- given x, H(x) is evenly distributed (any value equally possible. No avalanching)
- given H(x), finding x is hard (one-way function)
- given H(x), finding different y s.t. H(x)=H(y) is hard (collision resistant) (harder)
- (not given anything) finding any x ≠ y s.t. H(x)=H(y) is hard (birthday problem) (easier)

rationale:
can leverage full n bits of security
can't find somebody's secret
can't lie about pre-image. Every output is unique (usable as a key in different data types)

ex: SHA-256 ← Bitcoin
→ 256 bits of security

Keccak-256 ← Ethereum
Blake2b ← Cardano, Polkadot

however, this is not always enough, as even cryptographic hash functions can be attacked

(define) Rainbow Table
a *precomputed* table of results for cryptographic hash functions

ex: passwords are often predictable

Rank	Content
# 1	123456
# 2	qwerty
# 2	password

are there are not *that many commonly* used hash functions (ex: sha256)

∴ just precompute sha256("123456")
sha256("qwerty")
sha256("password")
} store these, and compare against any database leaks online

to solve this, project use a salt

(define) salt
a *random value* fed into hash functions in order to avoid rainbow table attacks
→ hash(password + salt)

ex: instead of storing *hash(password)*,
we store salt and *hash(password)*

Rank	Content	Salt	Hash
User # 1	123456	%yL9TS	0x0C9B
User # 2	qwerty	jLeZzy4	0xB625
User # 2	password	Pa5U1d	0xE947

→ long & random

however, even this is not perfect

Password length	Time to crack
12	20 yrs
13	1,000 yrs
14	53,000 yrs

← passwords containing
- uppercase
- lowercase
- letters & Z

not to mention
- implementation bugs
- theoretical vulnerabilities discovered
- only secure up to n bits (where n increases as computers get better)

common approach in cryptography:

(define) Ideal Functionality
if something is doable in theory (for some security parameter)
and it just depends on:
- ∃ a proper implementation → ∃ params for target security
- secure communication/composition with other components

then, just assume its correctness as part of *your model* ← often referred as an *ideal*

(define) Universal Composability (UC)
If an implementation is indistinguishable from its ideal functionality
(none in different environments and when composed)

then it's secure under the UC model

(define) Random Oracle Model (ROM)
Models an ideal hash function where:
- input: any data
- output: a *perfectly random* response
unique to that input

Hashes are great when all we want to model is a *one-way function*
However, we often care about composing functions and protocols
often with multiple network participants

but hash functions are often *too one-way*

ex: H(x) + H(y) ≠ H(x+y)
→ we cannot combine our computations without revealing to each-other x, y (requires an additional ideal functionality)
→ another partner computes this
→ we compute this

ex: how do I securely exchange messages with somebody else

To achieve this, we need extra structure
→ careful: structure can leads to attacks!

Two key results for quantum computers:

1. Lack of structure can easily be attacked

(define) Grover's Algorithm
For any "needle in the haystack" where any input is possible.
(unstructured search)

halve the # bits of security
even when finding hash pre-image!

← 64	2^{64} (≈18 quintillion)
← 128	2^{128}
256	2^{256} (≈ # atoms)

Intuition: "try all inputs all once"

recall: $\text{sqrt}(n) = n^{\frac{1}{2}}$
∴ $\text{sqrt}(2^n) = 2^n$

2. Too much structure can easily be attacked

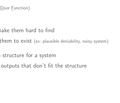
(define) Shor's Algorithm

For any periodic substructure that can be extracted,
quantum runtime: $O(\log(\text{classical}))$ (approximation) ← due to Quantum Fourier Transform (QFT)

$O(N \log N)$ (classical)
↓
$O(n \log n)$ (quantum) $n = \log N$

Intuition: integer factoring
(classical) $O\left(e^{(c \log N)^{\frac{1}{3}}}\right)$
(quantum) $O(\log N)^2$

leverage periodicity of $a^x \pmod N$



To achieve this, we look at ideal functions that loosen one-way functions:

#1: Invertible (not only one-way), but hard to invert

It turns out this is enough to build *asymmetric cryptography*

(define) Public Key Cryptography (asymmetric cryptography)

A type of cryptography where there exists two types of keys:
- private keys (secret keys): can efficiently invert the trapdoor function
- public keys: public data that anybody can use to verify an output was generated to you

so anybody can compute f(x)
- parameters
- setup
- sometimes even f(x) itself

ex: digital signatures
- private keys (secret keys): private data only you know used to initialize a one-way function unique to you
- public keys: public data that anybody can use to verify an output was generated to you

To achieve this, we look at ideal functions that loosen one-way functions:

TODO: hardness-based inversion
TODO: what ideal functionality do we want
TODO: group operations, number theory
TODO: *modular arithmetic*, and why it's the best implementation of the ideal
TODO: public-key cryptography
TODO: existence of real one-way function (of non-trapdoor variety) *would imply* P ⊆ NP. Therefore, one-way function itself is an ideal

TODO: but we want non-interactive ways to invert (discrete log is interactive,
because you need to do a key exchange to be able to invert somebody else's things.
This is not the case for FHE for example)
TODO: argument of knowledge
TODO: don't want to share trapdoor secret, but just knowledge you know
TODO: sampleable trapdoors

(define) Trapdoor function: a function easy in one direction, yet difficult to invert (TDF: Trap Door Function)

(note) TDF may
- have collisions (not injective) → sometimes we just make them hard to find / sometimes we want them to exist (ex: plausible deniability, noisy system)
- not cover every output (not surjective) → when imposing extra structure for a system / we don't care about outputs that don't fit the structure

ex: knowing path to a point a plane's clings (latticies)

(define) TrapDoor Permutation (TDP): a TDF that is one-to-one invertible (bijective)
i.e. it's defined on all possible outputs (a lot stricter than TDF)

ex: knowing factorization of a large number (RSA)

Notably, trapdoors allow us to build *asymmetric cryptography*,
which powers a lot of the modern world