Security II: Cryptography

Markus Kuhn

Computer Laboratory, University of Cambridge

https://www.cl.cam.ac.uk/teaching/1516/SecurityII/

These notes are provided as an aid for following the lectures, and are not a substitute for attending

Lent 2016 – Part II

1

Related textbooks

Main reference:

▶ Jonathan Katz, Yehuda Lindell:
Introduction to Modern Cryptography
Chapman & Hall/CRC, 2nd ed., 2014

Further reading:

▶ Christof Paar, Jan Pelzl:
Understanding Cryptography
Springer, 2010

http://www.springerlink.com/content/978-3-642-04100-6/
http://www.crypto-textbook.com/

▶ Douglas Stinson:
Cryptography – Theory and Practice
3rd ed., CRC Press, 2005

▶ Menezes, van Oorschot, Vanstone:
Handbook of Applied Cryptography
CRC Press, 1996
http://www.cacr.math.uwaterloo.ca/hac/

Encryption schemes

Encryption schemes are algorithm triples $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$:

Private-key (symmetric) encryption scheme

- $K \leftarrow \mathsf{Gen}$ key generation
- $C \leftarrow \mathsf{Enc}_K(M)$ encryption
- $M := \mathsf{Dec}_K(C)$ decryption

Public-key (asymmetric) encryption scheme

- $(PK, SK) \leftarrow \mathsf{Gen}$ public/secret key-pair generation
- $C \leftarrow \mathsf{Enc}_{PK}(M)$ encryption using public key
- $M := \mathsf{Dec}_{SK}(C)$ decryption using secret key

Probabilistic algorithms: Gen and (often also) Enc access a random-bit generator that can toss coins (uniformly distributed, independent).

Notation: $\leftarrow$ assigns the output of a probabilistic algorithm, $:=$ that of a deterministic algorithm.

Message-integrity schemes

Private key (symmetric):

Message authentication code (MAC)

- $K \leftarrow \mathsf{Gen}$ private-key generation
- $C \leftarrow \mathsf{Mac}_K(M)$ MAC generation
- $\mathsf{Vrfy}_K(M', C) = 1$ MAC verification
  $\Leftrightarrow M \overset{?}{=} M'$

Public key (asymmetric):

Digital signature

- $(PK, SK) \leftarrow \mathsf{Gen}$ public/secret key-pair generation
- $S \leftarrow \mathsf{Sign}_{SK}(M)$ signature generation using secret key
- $\mathsf{Vrfy}_{PK}(M', S) = 1$ signature verification using public key
  $\Leftrightarrow M \overset{?}{=} M'$

Hash functions

A *hash function* $h : \{0,1\}^* \rightarrow \{0,1\}^\ell$ efficiently maps arbitrary-length input strings onto fixed-length "hash values" such that the output is uniformly distributed in practice.

Typical applications of hash functions:

- ▶ hash table: data structure for fast $t = O(1)$ table lookup; storage address of a record containing value x is determined by $h(x)$.
- ▶ Bloom filter: data structure for fast probabilistic set membership test
- ▶ fast probabilistic string comparison (record deduplication, diff, rsync)
- ▶ Rabin–Karp algorithm: substring search with rolling hash

Closely related: checksums (CRC, Fletcher, Adler-32, etc.)

A good hash function h is one that minimizes the chances of a *collision* of the form $h(x) = h(y)$ with $x \neq y$.

But constructing collisions is not difficult for normal hash functions and checksums, e.g. to modify a file without affecting its checksum.

Algorithmic complexity attack: craft program input to deliberately trigger worst-case runtime (denial of service). Example: deliberately fill a server's hash table with colliding entries.

Secure hash functions

A *secure, collision-resistant hash function* is designed to make it infeasible for an adversary who knows the implementation of the hash function to find *any* collision

$$h(x) = h(y) \quad \text{with} \quad x \neq y$$

Examples for applications of secure hash functions:

- ▶ message digest for efficient calculation of digital signatures
- ▶ fast message-authentication codes (HMAC)
- ▶ tamper-resistant checksum of files

```
$ sha1sum security?-slides.tex
2c1331909a8b457df5c65216d6ee1efb2893903f  security1-slides.tex
50878bcf67115e5b6dcc866aa0282c570786ba5b  security2-slides.tex
```

- ▶ git commit identifiers
- ▶ P2P file sharing identifiers
- ▶ key derivation functions
- ▶ password verification
- ▶ hash chains (e.g., Bitcoin, timestamping services)
- ▶ commitment protocols

Secure hash functions: standards

- MD5: $\ell = 128$ (Rivest, 1991)
  insecure, collisions were found in 1996/2004, collisions used in
  real-world attacks (Flame, 2012) $\rightarrow$ avoid (still ok for HMAC)

  `http://www.ietf.org/rfc/rfc1321.txt`

- SHA-1: $\ell = 160$ (NSA, 1995)
  widely used today (e.g., git), but 2^{69}-step algorithm to find collisions
  found in 2005 $\rightarrow$ being phased out (still ok for HMAC)

- SHA-2: $\ell = 224, 256, 384,$ or 512
  close relative of SHA-1, therefore long-term collision-resistance
  questionable, very widely used standard

  FIPS 180-3 US government secure hash standard,
  `http://csrc.nist.gov/publications/fips/`

- SHA-3: KECCAK wins 5-year NIST contest in October 2012
  no length-extension attack, arbitrary-length output,
  can also operate as PRNG, very different from SHA-1/2.
  (other finalists: BLAKE, Grøstl, JH, Skein)

  `http://csrc.nist.gov/groups/ST/hash/sha-3/`
  `http://keccak.noekeon.org/`

Collision resistance – a formal definition

Hash function

A hash function is a pair of probabilistic polynomial-time (PPT)
algorithms (Gen, H) where

- Gen reads a security parameter 1^n and outputs a key s.
- H reads key s and input string $x \in \{0,1\}^*$ and outputs
  $H_s(x) \in \{0,1\}^{\ell(n)}$ (where n is a security parameter implied by s)

Formally define collision resistance using the following game:

1. Challenger generates a key $s = \text{Gen}(1^n)$
2. Challenger passes s to adversary $\mathcal{A}$
3. $\mathcal{A}$ replies with x, x'
4. $\mathcal{A}$ has found a collision iff $H_s(x) = H_s(x')$ and $x \neq x'$

A hash function (Gen, H) is *collision resistant* if for all PPT adversaries
$\mathcal{A}$ there is a negligible function negl such that

$$\mathbb{P}(\mathcal{A} \text{ found a collision}) \leq \text{negl}(n)$$

Recall "negligible function" (Security I): approaches zero faster than any polynomial, e.g 2^{-n}.
A fixed-length *compression function* is only defined on $x \in \{0,1\}^{\ell'(n)}$ with $\ell'(n) > \ell(n)$.

Unkeyed hash functions

Commonly used collision-resistant hash functions (SHA-256, etc.) do *not* use a key s. They are fixed functions of the form $h : \{0, 1\}^* \to \{0, 1\}^\ell$.

Why do we need s in the security definition?

Any fixed function h where the size of the domain (set of possible input values) is greater than the range (set of possible output values) will have collisions x, x'. There always exists a constant-time adversary $\mathcal{A}$ that just outputs these hard-wired values x, x'.

Therefore, a complexity-theoretic security definition must depend on a key s (and associated security parameter 1^n). Then H becomes a recipe for defining ever new collision-resistant fixed functions H_s.

So in practice, s is a publicly known fixed constant, embedded in the secure hash function h.

Also, without any security parameter n, we could not use the notion of a negligible function.

Weaker properties implied by collision resistance

Second-preimage resistance

For a given s and input value x, it is infeasible for any polynomial-time adversary to find x' with $H_s(x') = H_s(x)$ (except with negligible probability).

If there existed a PPT adversary $\mathcal{A}$ that can break the second-preimage resistance of H_s, than $\mathcal{A}$ can also break its collision resistance. Therefore, collision resistance implies second-preimage resistance.

Preimage resistance

For a given s and output value y, it is infeasible for any polynomial-time adversary to find x' with $H_s(x') = y$ (except with negligible probability).

If there existed a PPT adversary $\mathcal{A}$ that can break the pre-image resistance of H_s, than $\mathcal{A}$ can also break its second-preimage resistance (with high probability). Therefore, either collision resistance or second-preimage resistance imply preimage resistance.
How?

Note: collision resistance does *not* prevent H_s from leaking information about x ($\to$ CPA).

Merkle–Damgård construction

Wanted: variable-length hash function (Gen, H).

Given: (Gen, C), a fixed-length hash function with
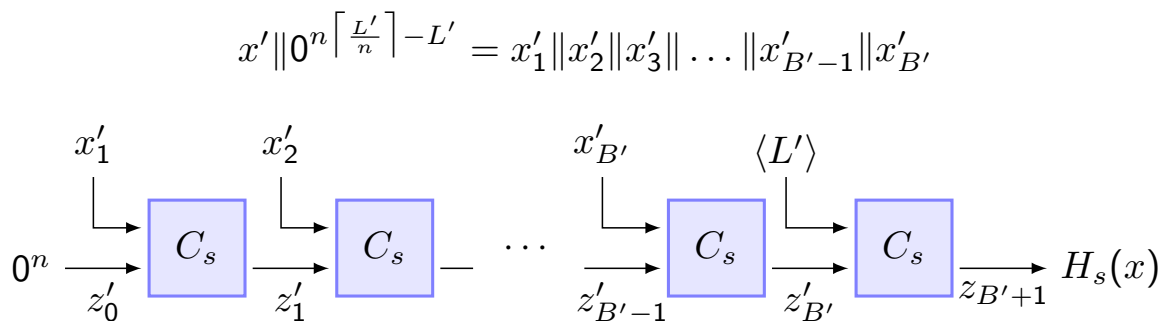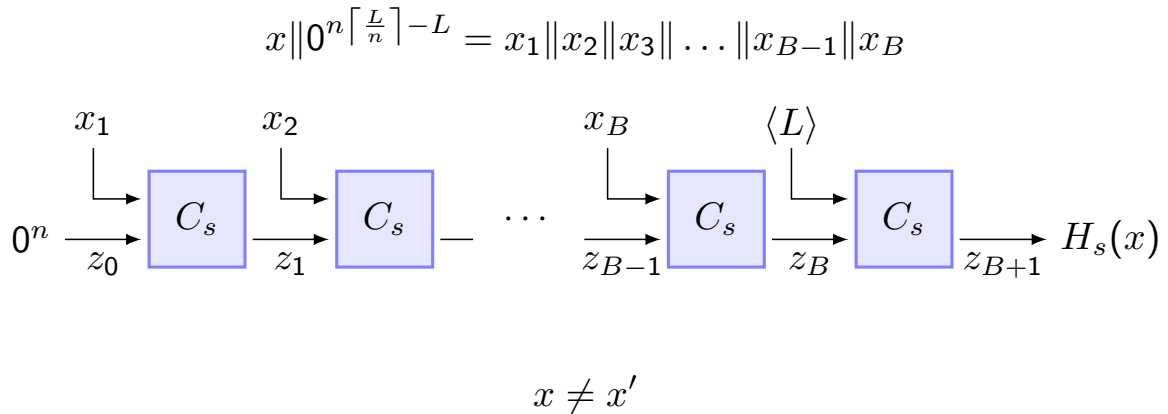$C : \{0,1\}^{2n} \to \{0,1\}^n$ ("compression function")

Input of H: key s, string $x \in \{0,1\}^L$ with length $L < 2^n$

❶ Pad x to length divisible by n by appending "0" bits, then split the result into $B = \lceil \frac{L}{n} \rceil$ blocks of length n each:

$$x \| 0^{n \lceil \frac{L}{n} \rceil - L} = x_1 \| x_2 \| x_3 \| \dots \| x_{B-1} \| x_B$$

❷ Append a final block $x_{B+1} = \langle L \rangle$, which contains the n-bit binary representation of input length $L = |x|$.

❸ Set $z_0 := 0^n$ (initial vector, IV)

❹ compute $z_i := C_s(z_{i-1} \| x_i)$ for $i = 1, \dots, B+1$

❺ Output $H_s(x) := z_{B+1}$

$$x \| 0^{n \lceil \frac{L}{n} \rceil - L} = x_1 \| x_2 \| x_3 \| \dots \| x_{B-1} \| x_B$$

$$x \neq x'$$

$$x' \| 0^{n \lceil \frac{L'}{n} \rceil - L'} = x'_1 \| x'_2 \| x'_3 \| \dots \| x'_{B'-1} \| x'_{B'}$$

Merkle–Damgård construction – security proof

If the fixed-length compression function C is collision resistant, so will be the variable-length hash function H resulting from the Merkle–Damgård construction.

Proof outline:

Assume C_s is collision resistant, but H is not, because some PPT adversary $\mathcal{A}$ outputs $x \neq x'$ with $H_s(x) = H_s(x')$.

Let $x_1, \ldots, x_B$ be the n-bit blocks of padded L-bit input x, and $x'_1, \ldots, x'_{B'}$ be those of L'-bit input x', and $x_{B+1} = \langle L \rangle$, $x'_{B'+1} = \langle L' \rangle$.

Case $L \neq L'$: Then $x_{B+1} \neq x'_{B'+1}$ but $H_s(x) = z_{B+1} = C_s(z_B \| x_{B+1}) = C_s(z'_{B'} \| x'_{B'+1}) = z'_{B'+1} = H_s(x')$, which is a collision in C_s.

Case $L = L'$: Now $B = B'$. Let $i \in \{1, \ldots, B+1\}$ be the largest index where $z_{i-1} \| x_i \neq z'_{i-1} \| x'_i$. (Such i exists as due to $|x| = |x'|$ and $x \neq x'$ there will be at least one $1 \leq j \leq B$ with $x_j \neq x'_j$.) Then $z_k = z'_k$ for all $k \in \{i, \ldots, B+1\}$ and $z_i = C_s(z_{i-1} \| x_i) = C_s(z'_{i-1} \| x'_i) = z'_i$ is a collision in C_s.

So C_s was not collision resistant, invalidating the assumption.
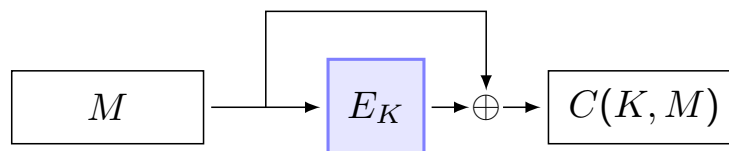
Compression function from block ciphers

Davies–Meyer construction

One possible technique for obtaining a collision-resistant compression function C is to use a block cipher $E : \{0,1\}^\ell \times \{0,1\}^n \to \{0,1\}^n$ in the following way:

$$C(K, M) = E_K(M) \oplus M$$

or in the notation of slide 11 (with $K = x_i$ and $M = z_{i-1}$):

$$C(z_{i-1} \| x_i) = E_{x_i}(z_{i-1}) \oplus z_{i-1}$$

However, the security proof for this construction requires E to be an *ideal cipher*, a keyed random permutation. It is not sufficient for E to merely be a strong pseudo-random permutation.

Warning: use only block ciphers that have specifically been designed to be used this way. Other block ciphers (e.g., DES) may have properties that can make them unsuitable here (e.g., related key attacks, block size too small).

SHA-1 structure

Merkle–Damgård construction, block length $n = 512$ bits.

Compression function:

- Input $= 160$ bits $=$ five 32-bit registers A–E
- each block $= 16$ 32-bit words $W_0, \ldots, W_{15}$
- LFSR extends that sequence to 80 words: $W_{16}, \ldots, W_{79}$
- 80 rounds, each fed one W_i
- Round constant K_i and non-linear function F_i change every 20 rounds.
- four 32-bit additions $\boxplus$ and two 32-bit rotations per round, 2–5 32-bit Boolean operations for F.
- finally: 32-bit add round 0 input to round 79 output (Davies–Meyer)

One round:
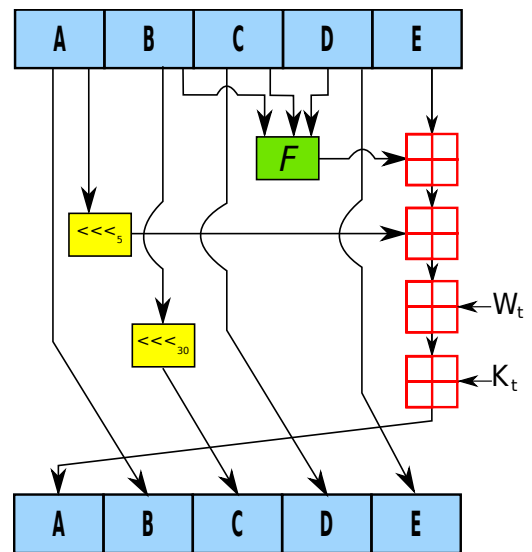
commons.wikimedia.org, CC SA-BY

Random oracle model

Many applications of secure hash functions have no security proof that relies only on the collision resistance of the function used.

The known security proofs require instead a much stronger assumption, the strongest possible assumption one can make about a hash function:

Random oracle

- A random oracle H is a device that accepts arbitrary length strings $X \in \{0,1\}^*$ and consistently outputs for each a value $H(X) \in \{0,1\}^\ell$ which it chooses uniformly at random.
- Once it has chosen an $H(X)$ for X, it will always output that same answer for X consistently.
- Parties can privately query the random oracle (nobody else learns what anyone queries), but everyone gets the same answer if they query the same value.
- No party can infer anything about $H(X)$ other than by querying X.

Ideal cipher model

An random-oracle equivalent can be defined for block ciphers:

> **Ideal cipher**
>
> Each key $K \in \{0,1\}^\ell$ defines a random permutation E_K, chosen uniformly at random out of all $(2^n)!$ permutations. All parties have oracle access to both $E_K(X)$ and $E_K^{-1}(X)$ for any (K, X). No party can infer any information about $E_K(X)$ (or $E_K^{-1}(X)$) without querying is value for (K, X).

We have encountered random functions and random permutations before, as a tool for defining pseudo-random functions/permutations.

Random oracles and ideal ciphers are different:

If a security proof is made "in the random oracle model", then a hash function is replaced by a random oracle or a block cipher is replaced by an ideal cipher.

In other words, the security proof makes much stronger assumptions about these components: they are not just indistinguishable from random functions/permutations by any polynomial-time distinguisher, they are actually assumed to be random functions/permutations.

Davies–Meyer construction – security proof

$$C(K, X) = E_K(X) \oplus X$$

If E is modeled as an *ideal cipher*, then C is a collision-resistant hash function. Any attacker $\mathcal{A}$ making $q < 2^{\ell/2}$ oracle queries to E finds a collision with probability not higher than $q^2/2^\ell$. (negligible)

Proof: Attacker $\mathcal{A}$ tries to find $(K, X), (K', X')$ with $E_K(X) \oplus X = E_{K'}(X') \oplus X'$. We assume that, before outputting $(K, X), (K', X')$, $\mathcal{A}$ has previously made queries to learn $E_K(X)$ and $E_{K'}(X')$. We also assume (wlog) $\mathcal{A}$ never makes redundant queries, so having learnt $Y = E_K(X)$, $\mathcal{A}$ will not query $E_K^{-1}(Y)$ and vice versa.

The i-th query (K_i, X_i) to E only reveals

$$c_i = C_i(K_i, X_i) = E_{K_i}(X_i) \oplus X_i.$$

A query to E^{-1} instead would only reveal $E^{-1}(K_i, Y_i) = X_i$ and therefore

$$c_i = C_i(K_i, X_i) = Y_i \oplus E_{K_i}^{-1}(Y_i).$$

$\mathcal{A}$ needs to find $c_i = c_j$ with $i > j$.

For some fixed pair i, j with $i > j$, what is the probability of $c_i = c_j$?

A collision at query i can only occur as one of these two query results:

- $E_{K_i}(X_i) = c_j \oplus X_i$
- $E_{K_i}^{-1}(Y_i) = c_j \oplus Y_i$

Each query will reveal a new uniformly distributed ℓ-bit value, except that it may be constrained by (at most) $i - 1$ previous query results (since E_{K_i} must remain a permutation).

Therefore, the ideal cipher E will answer query i by uniformly choosing a value out of at least $2^\ell - (i - 1)$ possible values.

Therefore, each of the above two possibilities for reaching $c_i = c_j$ can happen with probability no higher than $1/(2^\ell - (i - 1))$.

With $i \le q < 2^{\ell/2}$ and $\ell > 1$, we have

$$\mathbb{P}(c_i = c_j) \le \frac{1}{2^\ell - (i - 1)} \le \frac{1}{2^\ell - 2^{\ell/2}} \le \frac{2}{2^\ell}$$

There are $\binom{q}{2} < q^2/2$ pairs $j < i \le q$, so the collision probability after q queries cannot be more than $\frac{2}{2^\ell} \cdot \frac{q^2}{2} = \frac{q^2}{2^\ell}$. $\qquad\square$

Random oracle model – controversy

Security proofs that replace the use of a hash function with a query to a random oracle (or a block cipher with an ideal cipher) remain controversial.

Cons

- ▶ Real hash algorithms are publicly known. Anyone can query them privately as often as they want, and look for shortcuts.
- ▶ No good justification to believe that proofs in the random oracle model say anything about the security of a scheme when implemented with practical hash functions (or pseudo-random functions/permutations).
- ▶ No good criteria known to decide whether a practical hash function is "good enough" to instantiate a random oracle.

Pros

- ▶ A random-oracle model proof is better than no proof at all.
- ▶ Many efficient schemes (especially for public-key crypto) only have random-oracle proofs.
- ▶ No history of successful real-world attacks against schemes with random-oracle security proofs.
- ▶ If such a scheme were attacked successfully, it should still be fixable by using a better hash function.

Sponge functions

Another way to construct a secure hash function $H(M) = Z$:
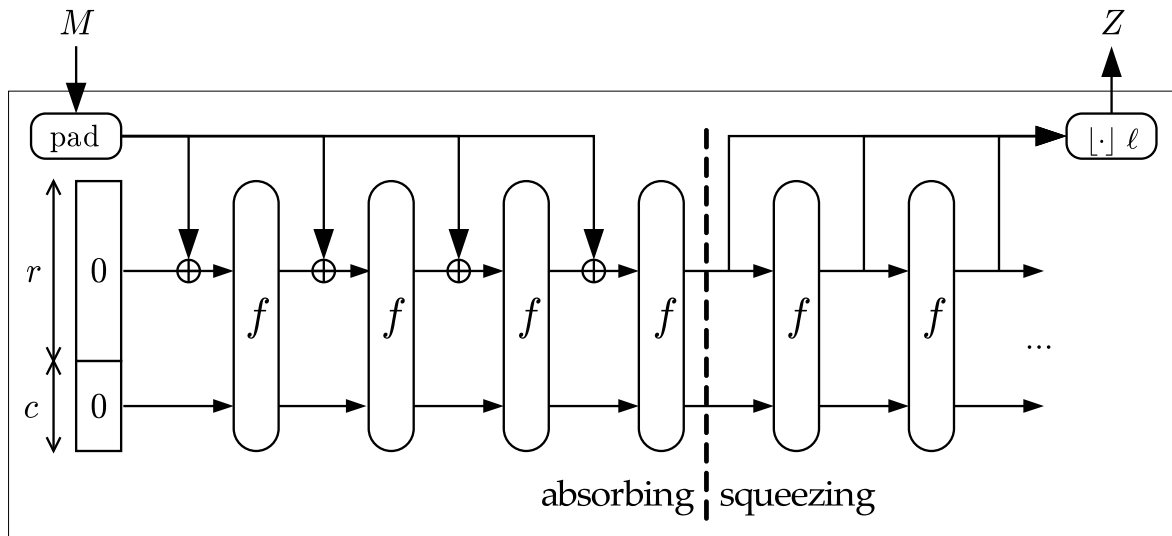
sponge

$(r + c)$-bit internal state, XOR r-bit input blocks at a time, stir with pseudo-random permutation f, output r-bit output blocks at a time.

Versatile: secure hash function (variable input length) and stream cipher (variable output length)

Advantage over Merkle–Damgård: internal state > output, flexibility.

Duplex construction

init.   duplexing   duplexing   duplexing
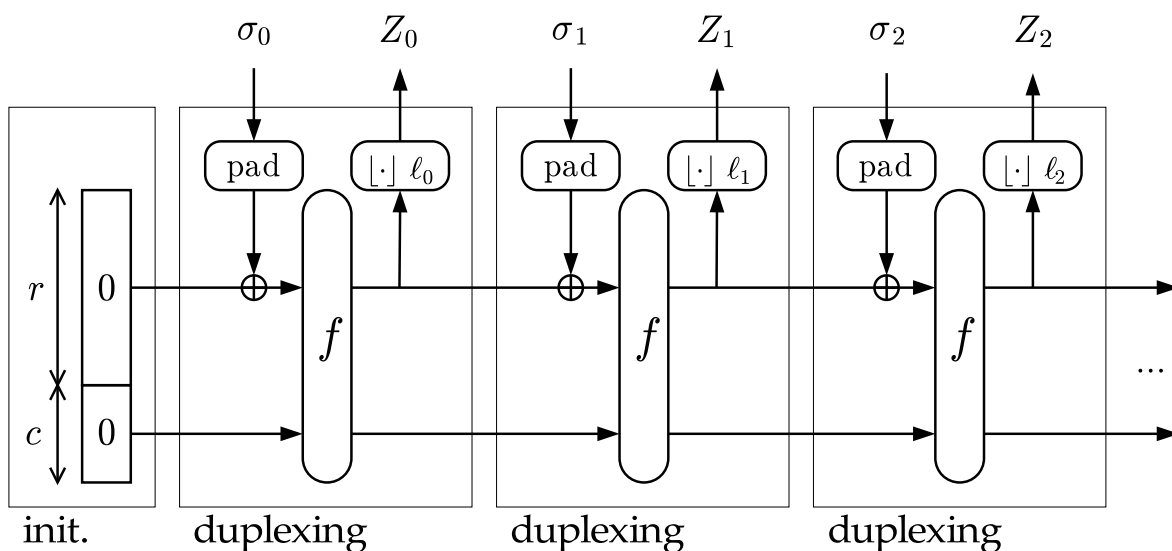
A variant of the sponge construction, proposed to provide

- ▶ authenticated encryption (basic idea: $\sigma_i = C_i = M_i \oplus Z_{i-1}$)
- ▶ reseedable pseudo-random bit sequence generator (for post-processing and expanding physical random sources)

G. Bertoni, J. Daemen, et al.: Duplexing the sponge: single-pass authenticated encryption and other applications. SAC 2011. http://dx.doi.org/10.1007/978-3-642-28496-0_19 http://sponge.noekeon.org/SpongeDuplex.pdf

SHA-3

Latest NIST secure hash algorithm

- Sponge function with $b = r + c = 1600 = 5 \times 5 \times 64$ bits of state

- Standardized (SHA-2 compatible) output sizes:
  $\ell \in \{224, 256, 384, 512\}$ bits

- Internal capacity: $c = 2\ell$

- Input block size: $r = b - 2\ell \in \{1152, 1088, 832, 576\}$ bits

- Padding: append 10*1 to extend input to next multiple of r

NIST also defined two related extendable-output functions (XOFs),
SHAKE128 and SHAKE256, which accept arbitrary-length input and can
produce arbitrary-length output. PRBG with 128 or 256-bit security.

SHA-3 standard: permutation-based hash and extendable-output functions. August 2015.
`http://dx.doi.org/10.6028/NIST.FIPS.202`

Probability of collision / birthday problem

Throw b balls into n bins, selecting each bin uniformly at random.
With what probability do at least two balls end up in the same bin?
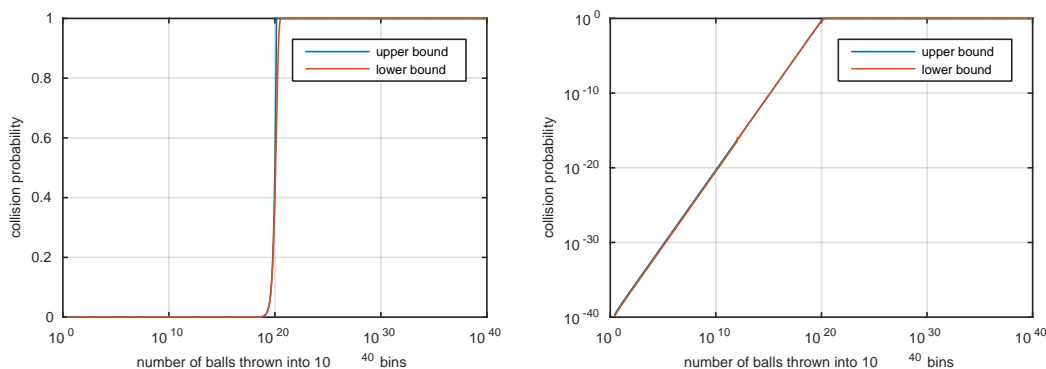
Remember: for large n the collision probability

- is near 1 for $b \gg \sqrt{n}$

- is near 0 for $b \ll \sqrt{n}$, growing roughly proportional to $\frac{b^2}{n}$

Expected number of balls thrown before first collision: $\sqrt{\frac{\pi}{2}n}$ (for $n \to \infty$)

No simple, efficient, and exact formula for collision probability, but good approximations:
`http://cseweb.ucsd.edu/~mihir/cse207/w-birthday.pdf`

"Birthday attacks"

If a hash function outputs ℓ-bit words, an attacker needs to try only
        different input values, before there is a better than 50% chance of
finding a collision.

Computational security

Attacks requiring 2^{128} steps considered infeasible $\implies$ use hash function
that outputs $\ell = 256$ bits (e.g., SHA-256). If only second pre-image
resistance is a concern, shorter $\ell = 128$-bit may be acceptable.

Finding useful collisions

An attacker needs to generate a large number of plausible input
plaintexts to find a practically useful collision. For English plain text,
synonym substitution is one possibility for generating these:

  A: Mallory is a {good,hardworking} and {honest,loyal} {employee,worker}
  B: Mallory is a {lazy,difficult} and {lying,malicious} {employee,worker}

Both A and B can be phrased in 2^3 variants each $\implies 2^6$ pairs of phrases.

With a 64-bit hash over an entire letter, we need only   such
sentences for a good chance to find a collision in   steps.

Low-memory collision search

A normal search for an ℓ-bit collision uses $O(2^{\ell/2})$ memory and time.

Algorithm for finding a
collision with $O(1)$ memory
and $O(2^{\ell/2})$ time:

Input: $H : \{0,1\}^* \to \{0,1\}^\ell$
Output: $x \neq x'$ with $H(x) = H'(x)$

$x_0 \leftarrow \{0,1\}^{\ell+1}$
$x' := x := x_0$
$i := 0$
loop
   $i := i + 1$
   $x := H(x)$ $// \ x = H^i(x_0)$
   $x' := H(H(x'))$ $// \ x' = H^{2i}(x_0)$
until $x = x'$
$x' := x, \ x := x_0$
for $j = 1, 2, \ldots, i$
   if $H(x) = H(x')$ **return** (x, x')
   $x := H(x)$ $// \ x = H^j(x_0)$
   $x' := H(x')$ $// \ x' = H^{i+j}(x_0)$

Basic idea:

- Tortoise x goes at most once
  round the cycle, hare x' at
  least once

- loop 1: ends when x'
  overtakes x for the first time
  $\Rightarrow x'$ now i steps ahead of x
  $\Rightarrow i$ is now an integer
  multiple of the cycle length

- loop 2: x back at start, x' is i
  steps ahead, same speed
  $\Rightarrow$ meet at cycle entry point

Wikipedia: Cycle detection

Constructing meaningful collisions

Tortoise-hare algorithm gives no direct control over content of x, x'.

Solution:

Define a text generator function $g : \{0,1\}^\ell \to \{0,1\}^*$, e.g.

$$g(0000) = \text{Mallory is a good and honest employee}$$
$$g(0001) = \text{Mallory is a lazy and lying employee}$$
$$g(0010) = \text{Mallory is a good and honest worker}$$
$$g(0011) = \text{Mallory is a lazy and lying worker}$$
$$g(0100) = \text{Mallory is a good and loyal employee}$$
$$g(0101) = \text{Mallory is a lazy and malicious employee}$$
$$\cdots$$
$$g(1111) = \text{Mallory is a difficult and malicious worker}$$

Then apply the tortoise-hare algorithm to $H(x) = h(g(x))$, if h is the hash function for which a meaningful collision is required.

With probability $\frac{1}{2}$ the resulting x, x' ($h(g(x)) = h(g(x'))$) will differ in the last bit $\Rightarrow$ collision between two texts with different meanings.

Hash and MAC

A secure hash function can be combined with a fixed-length MAC to provide a variable-length MAC $\mathsf{Mac}_k(H(m))$. More formally:

Let $\Pi = (\mathsf{Mac}, \mathsf{Vrfy})$ be a MAC for messages of length $\ell(n)$ and let $\Pi_H = (\mathsf{Gen}_H, H)$ be a hash function with output length $\ell(n)$. Then define variable-length MAC $\Pi' = (\mathsf{Gen}', \mathsf{Mac}', \mathsf{Vrfy}')$ as:

▶ Gen': Read security parameter 1^n, choose uniform $k \in \{0,1\}^n$, run $s := \mathsf{Gen}_H(1^n)$ and return (k, s).

▶ Mac': read key (k, s) and message $m \in \{0,1\}^*$, return tag $\mathsf{Mac}_k(H_s(m))$.

▶ Vrfy' : read key (k, s), message $m \in \{0,1\}^*$, tag t, return $\mathsf{Vrfy}_k(H_s(m), t)$.

If Π offers existential unforgeability and Π_H is collision resistant, then Π' will offer existential unforgeability.

Proof outline: If an adversary used Mac' to get tags on a set Q of messages, and then can produce a valid tag for $m^* \notin Q$, then there are two cases:

▶ $\exists m \in Q$ with $H_s(m) = H_s(m^*) \Rightarrow H_s$ not collision resistant

▶ $\forall m \in Q : H_s(m) \neq H_s(m^*) \Rightarrow \mathsf{Mac}$ failed existential unforgeability

Hash-based message authentication code

Initial idea: hash a message M prefixed with a key K to get

$$\text{MAC}_K(M) = h(K\|M)$$

This construct is secure in the random oracle model (where h is a random function). Is is also generally considered secure with fixed-length m-bit messages $M \in \{0,1\}^m$ or with sponge-function based hash algorithm h, such as SHA-3.

Danger: If h uses the Merkle–Damgård construction, an adversary can call the compression function again on the MAC to add more blocks to M, and obtain the MAC of a longer M' without knowing the key!

To prevent such a message-extension attack, variants like

$$\text{MAC}_K(M) = h(h(K\|M))$$

or

$$\text{MAC}_K(M) = h(K\|h(M))$$

could be used to terminate the iteration of the compression function in a way that the adversary cannot continue. $\Rightarrow$ HMAC

HMAC

HMAC is a standard technique widely used to form a message-authentication code using a Merkle–Damgård-style secure hash function h, such as MD5, SHA-1 or SHA-256:

$$\text{HMAC}_K(x) = h(K \oplus \text{opad}\|h(K \oplus \text{ipad}\|x))$$

Fixed padding values ipad, opad extend the key to the input size of the compression function, to permit precomputation of its first iteration.

$$x\|\text{padding}(n + |x|) = x_1\|x_2\|x_3\|\ldots\|x_{B-1}\|x_B$$

http://www.ietf.org/rfc/rfc2104.txt
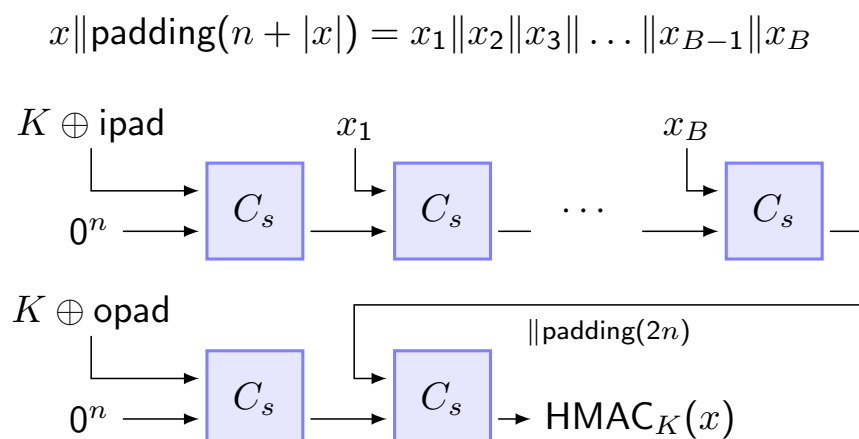
Secure commitment

You have today an idea that you write down in message M. You do not want to publish M yet, but you want to be able to prove later that you knew M already today. Initial idea: you publish $h(M)$ today.

Danger: if the entropy of M is small (e.g., M is a simple choice, a PIN, etc.), there is a high risk that your adversary can invert the collision-resistant function h successfully via brute-force search.

Solution:

- Pick (initially) secret $N \in \{0,1\}^{128}$ uniformly at random.
- Publish $h(N, M)$ (as well as h and $|N|$).
- When the time comes to reveal M, also reveal N.

You can also commit yourself to message M, without yet revealing it's content, by publishing $h(N, M)$.

Applications: online auctions with sealed bids, online games where several parties need to move simultaneously, etc.

Tuple (N, M) means any form of unambiguous concatenation, e.g. $N \| M$ if length $|N|$ is agreed.
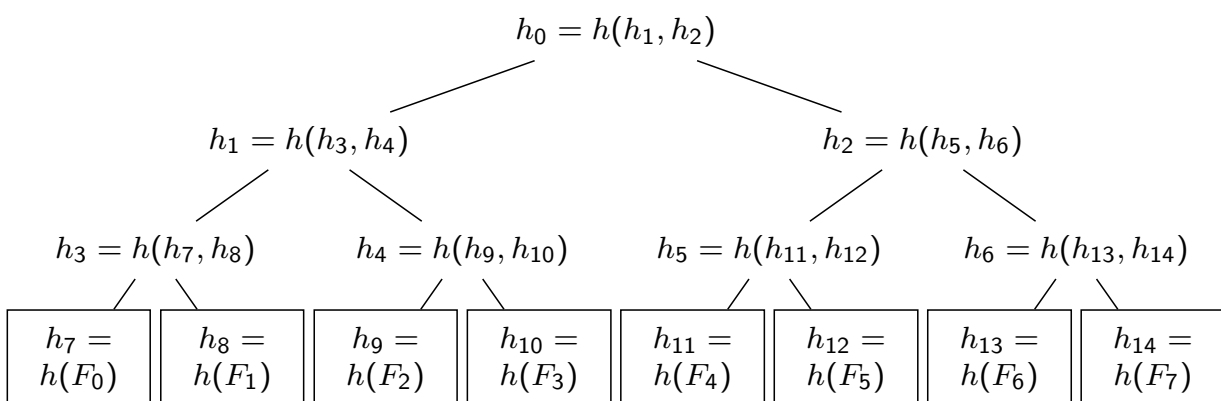
Merkle tree

Problem: Untrusted file store, small trusted memory. Solution: hash tree.

Leaves contain hash values of files $F_0, \ldots, F_{k-1}$. Each inner node contains the hash of its children. Only root h_0 (and number k of files) needs to be stored securely.

Advantages of tree (over naive alternative $h_0 = h(F_0, \ldots, F_{k-1})$):

- Update of a file F_i requires only $O(\log k)$ recalculations of hash values along path from $h(F_i)$ to root (not rereading every file).
- Verification of a file requires only reading $O(\log k)$ values in all direct children of nodes in path to root (not rereading every node).

$$h_0 = h(h_1, h_2)$$

$$h_1 = h(h_3, h_4) \qquad h_2 = h(h_5, h_6)$$

$$h_3 = h(h_7, h_8) \quad h_4 = h(h_9, h_{10}) \quad h_5 = h(h_{11}, h_{12}) \quad h_6 = h(h_{13}, h_{14})$$

$h_7 = h(F_0)$	$h_8 = h(F_1)$	$h_9 = h(F_2)$	$h_{10} = h(F_3)$	$h_{11} = h(F_4)$	$h_{12} = h(F_5)$	$h_{13} = h(F_6)$	$h_{14} = h(F_7)$

One-time passwords from a hash chain

Generate hash chain: (h is preimage resistant, with ASCII output)

$$
\begin{aligned}
R_0 &\leftarrow \text{random} \\
R_1 &:= h(R_0) \\
&\vdots \\
R_{n-1} &:= h(R_{n-2}) \\
R_n &:= h(R_{n-1})
\end{aligned}
$$

Equivalently: $R_i := \underbrace{h(h(h(\ldots h(\, R_0)\ldots)))}_{i \text{ times}} = h^i(R_0) \quad (0 < i \le n)$

Store last chain value $H := R_n$ on the host server. Give the remaining list $R_{n-1}, R_{n-2}, \ldots, R_0$ as one-time passwords to the user.

When user enters password R_i, compare $h(R_i) \overset{?}{=} H$. If they match:

▶ Update $H := R_{i-1}$ on host

▶ grant access to user

Leslie Lamport: *Password authentication with insecure communication.* CACM 24(11)770–772, 1981. http://doi.acm.org/10.1145/358790.358797

Broadcast stream authentication

Alice sends to a group of recipients a long stream of messages $M_1, M_2, \ldots, M_n$. They want to verify Alice's signature on each packet immediately upon arrival, but it is too expensive to sign each message.

Alice calculates

$$
\begin{aligned}
C_1 &= h(C_2, M_1) \\
C_2 &= h(C_3, M_2) \\
C_3 &= h(C_4, M_3) \\
&\cdots \\
C_{n-2} &= h(C_{n-1}, M_{n-2}) \\
C_{n-1} &= h(C_n, M_{n-1}) \\
C_n &= h(0, M_n)
\end{aligned}
$$

and then broadcasts the stream

$$C_1, \mathsf{Sign}(C_1), (C_2, M_1), (C_3, M_2), \ldots, (0, M_n).$$

Only the first check value is signed, all other packets are bound together in a hash chain that is linked to that single signature.

Problem: Alice needs to know M_n before she can start to broadcast C_1. Solution: TESLA

Timed Efficient Stream Loss-tolerant Authentication

TESLA uses a hash chain to authenticate broadcast data, without any need for a digital signature for each message.

Timed broadcast of data sequence $M_1, M_2, \ldots, M_n$:

- $t_0 : \mathsf{Sign}(R_0), R_0$ where $R_0 = h(R_1)$

- $t_1 : (\mathsf{Mac}_{R_2}(M_1), M_1, R_1)$ where $R_1 = h(R_2)$

- $t_2 : (\mathsf{Mac}_{R_3}(M_2), M_2, R_2)$ where $R_2 = h(R_3)$

- $t_3 : (\mathsf{Mac}_{R_4}(M_3), M_3, R_3)$ where $R_3 = h(R_4)$

- $t_4 : (\mathsf{Mac}_{R_5}(M_4), M_4, R_4)$ where $R_4 = h(R_5)$

- $\ldots$

Each R_i is revealed at a pre-agreed time t_i. The MAC for M_i can only be verified after t_{i+1} when key R_{i+1} is revealed.

By the time the MAC key is revealed, everyone has already received the MAC, therefore the key can no longer be used to spoof the message.

Hash chains, block chains, time-stamping services

Clients continuously produce transactions M_i (e.g., money transfers).

Block-chain time-stamping service: receives client transactions M_i, may order them by dependency, validates them (payment covered by funds?), batches them into groups

$$G_1 = (M_1, M_2, M_3)$$
$$G_2 = (M_4, M_5, M_6, M_7)$$
$$G_3 = (M_8, M_9)$$
$$\ldots$$

and then publishes the hash chain (with timestamps t_i)

$$B_1 = (G_1, t_1, 0)$$
$$B_2 = (G_2, t_2, h(B_1))$$
$$B_3 = (G_3, t_3, h(B_2))$$
$$\ldots$$
$$B_i = (G_i, t_i, h(B_{i-1}))$$

New blocks are broadcast to and archived by clients. Clients can

- verify that $t_{i-1} \leq t_i \leq$ now
- verify $h(B_{i-1})$
- frequently compare latest $h(B_i)$ with other clients

to ensure consensus that

- each client sees the same serialization order of the same set of validated transactions
- every client receives the exact same block-chain data
- nobody can later rewrite the transaction history

The **Bitcoin** crypto currency is based on a *decentralized* block-chain:

- accounts identified by single-use public keys
- each transaction signed with the payer's private key
- new blocks broadcast by "miners", who are allowed to mint themselves new currency as incentive for operating the service
- issuing rate of new currency is limited by requirement for miners to solve cryptographic puzzle (adjust a field in each block such that $h(B_i)$ has a required number of leading zeros, currently ≈ 68 bits)

`https://blockchain.info/` `https://en.bitcoin.it/`

Hashing passwords

Password storage

Avoid saving a user's password P as plaintext. Saving the hash $h(P)$ instead helps to protect the passwords after theft of the database. Verify password by comparing it's hash with the database record.

Better: hinder dictionary attacks by adding a random salt value S and by iterating the hash function C times to make it computationally more expensive. The database record then stores $(S, h^C(P, S))$ or similar.

PBKDF2 iterates HMAC C times for each output bit.

Typical values: $S \in \{0, 1\}^{128}$, $10^3 < C < 10^7$

Password-based key derivation

Passwords have low entropy per bit (e.g. only ≈ 95 graphical characters per byte from keyboard) and therefore make bad cryptographic keys.

Preferably use a true random bit generator to generate cryptographic keys. If you must derive keys from a password, encourage users to choose passwords much longer than the key length, then hash the password to generate a uniform key from it. (Dictionary-attack: see above)

Recommendation for password-based key derivation. NIST SP 800-132, December 2010.

Inverting unsalted password hashes: "rainbow tables"

Target: invert $h(p)$, where $p \in P$ is a password from an assumed finite set P of passwords (e.g., $h = \text{MD5}$, $|P| = 95^8 \approx 2^{53}$ 8-char ASCII strings)

Idea: define a "rainbow" of n reduction functions $r_i : \{0,1\}^{128} \to P$, then iterate $h(r_i(\cdot))$ to avoid loops. (For example: $r_i(x) := r(h(x\|\langle i\rangle))$.)

$$m \left\downarrow \quad x_0 \overset{r_1}{\to} p_1 \overset{h}{\to} x_1 \overset{r_2}{\to} p_2 \overset{h}{\to} \cdots \overset{h}{\to} x_{n-1} \overset{r_n}{\to} p_n \overset{h}{\to} x_n \quad \Rightarrow \quad L[x_n] := x_0 \right.$$
$$\vdots$$

$\text{PRECOMPUTE}(h, r, m, n):$
  for $j := 1$ **to** m
    $x_0 \in_{\mathsf{R}} \{0,1\}^{128}$
    for $i := 1$ **to** n
      $p_i := r_i(x_{i-1})$
      $x_i := h(p_i)$
    store $L[x_n] := x_0$
  return L

$\text{INVERT}(h, r, n, L, x):$
  for $k := n$ **downto** 1
    $x_{k-1} := x$
    for $i := k$ **to** n
      $p_i := r_i(x_{i-1})$
      $x_i := h(p_i)$
    if $L[x_n]$ exists
      $p_1 := r_1(L[x_n])$
      for $j := 1$ **to** n
        if $h(p_j) = x$
          return p_j
        $p_{j+1} := r_{j+1}(h(p_j))$

Trade-off
time:
$$n \approx |P|^{1/3}$$
memory:
$$m \approx |P|^{2/3}$$

Philippe Oechslin: Making a faster cryptanalytic time–memory trade-off. CRYPTO 2003.
https://dx.doi.org/10.1007/978-3-540-45146-4_36

# Other applications of secure hash functions

- ▶ deduplication – quickly identify in a large collection of files duplicates, without having to compare all pairs of files, just compare the hash of each files content.

- ▶ file identification – in a peer-to-peer filesharing network or cluster file system, identify each file by the hash of its content.

- ▶ distributed version control systems (git, mercurial, etc.) – name each revision via a hash tree of all files in that revision, along with the hash of the parent revision(s). This way, each revision name securely identifies not only the full content, but its full revision history.

- ▶ key derivation – avoid using the same key K for more than one purpose. Better use a secure hash function to derive multiple other keys $K_1, K_2, \ldots$, one for each application: $K_i = h(K, i)$

Key distribution problem

In a group of n participants, there are $n(n-1)/2$ pairs who might want to communicate at some point, requiring $O(n^2)$ private keys to be exchanged securely in advance.

This gets quickly unpractical if $n \gg 2$ and if participants regularly join and leave the group.
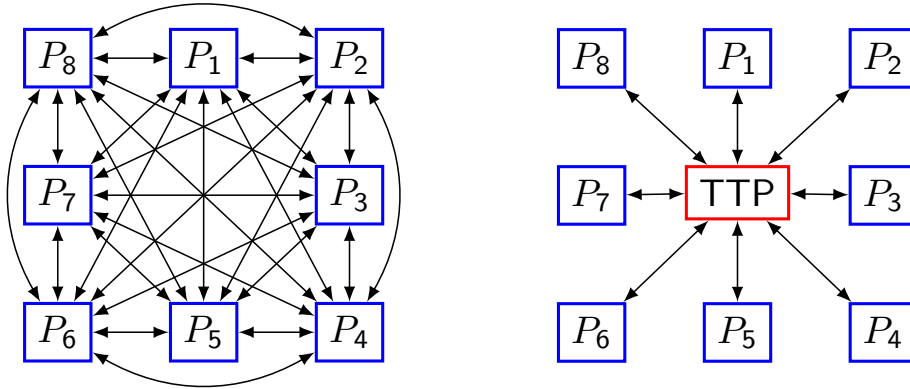
**Alternative 1:** introduce an intermediary "trusted third party"

Trusted third party – key distribution centre

Needham–Schroeder protocol

Communal trusted server S shares key K_{PS} with each participant P.

① A informs S that it wants to communicate with B.

② S generates K_{AB} and replies to A with
$\mathsf{Enc}_{K_{AS}}(B, K_{AB}, \mathsf{Enc}_{K_{BS}}(A, K_{AB}))$
Enc is a symmetric authenticated-encryption scheme

③ A checks name of B, stores K_{AB}, and forwards the "ticket"
$\mathsf{Enc}_{K_{BS}}(A, K_{AB})$ to B

④ B also checks name of A and stores K_{AB}.

⑤ A and B now share K_{AB} and communicate via $\mathsf{Enc}_{K_{AB}}/\mathsf{Dec}_{K_{AB}}$.
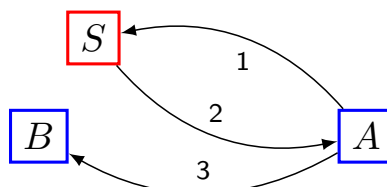
Kerberos

An extension of the Needham–Schroeder protocol is now widely used in corporate computer networks between desktop computers and servers, in the form of Kerberos and Microsoft's Active Directory. K_{AS} is generated from A's password (hash function).

Extensions include:

- timestamps and nonces to prevent replay attacks

- a "ticket-granting ticket" is issued and cached at the start of a session, replacing the password for a limited time, allowing the password to be instantly wiped from memory again.

- a pre-authentication step ensures that S does not reply with anything encrypted under K_{AS} unless the sender has demonstrated knowledge of K_{AS}, to hinder offline password guessing.

- mechanisms for forwarding and renewing tickets

- support for a federation of administrative domains ("realms")

Problem: ticket message enables eavesdropper off-line dictionary attack.

Key distribution problem: other options

Alternative 2: hardware security modules + conditional access
1. A trusted third party generates a global key K and embeds it securely in tamper-resistant hardware tokens (e.g., smartcard)
2. Every participant receives such a token, which also knows the identity of its owner and that of any groups they might belong to.
3. Each token offers its holder authenticated encryption operations $\mathrm{Enc}_K(\cdot)$ and $\mathrm{Dec}_K(A, \cdot)$.
4. Each encrypted message $\mathrm{Enc}_K(A, M)$ contains the name of the intended recipient A (or the name of a group to which A belongs).
5. A's smartcard will only decrypt messages addressed this way to A.

Commonly used for "broadcast encryption", e.g. pay-TV, navigation satellites.

Alternative 3: Public-key cryptography

- Find an encryption scheme where separate keys can be used for encryption and decryption.

- Publish the encryption key: the "public key"

- Keep the decryption key: the "secret key"

Some form of trusted third party is usually still required to certify the correctness of the published public keys, but it is no longer directly involved in establishing a secure connection.

Public-key encryption

A **public-key encryption scheme** is a tuple of PPT algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ such that

- the **key generation algorithm** Gen receives a security parameter ℓ and outputs a pair of keys $(PK, SK) \leftarrow \mathsf{Gen}(1^\ell)$, with key lengths $|PK| \geq \ell$, $|SK| \geq \ell$;

- the **encryption algorithm** Enc maps a public key PK and a plaintext message $M \in \mathcal{M}$ to a ciphertext message $C \leftarrow \mathsf{Enc}_{PK}(M)$;

- the **decryption algorithm** Dec maps a secret key SK and a ciphertext C to a plaintext message $M := \mathsf{Dec}_{SK}(C)$, or outputs $\bot$;

- for all ℓ, $(PK, SK) \leftarrow \mathsf{Gen}(1^\ell)$: $\mathsf{Dec}_{SK}(\mathsf{Enc}_{PK}(M)) = M$.

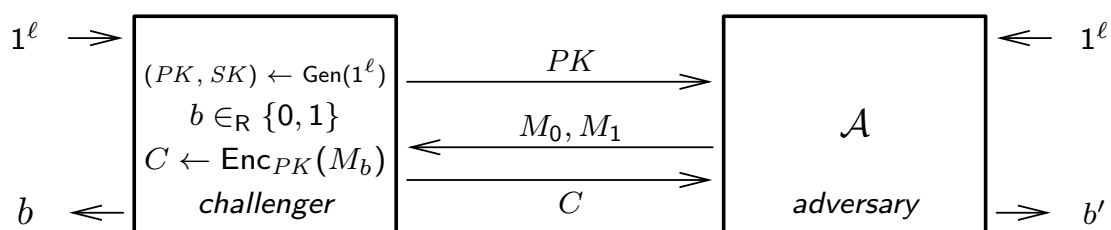In practice, the message space $\mathcal{M}$ may depend on PK.

In some practical schemes, the condition $\mathsf{Dec}_{SK}(\mathsf{Enc}_{PK}(M)) = M$ may fail with negligible probability.

Security against chosen-plaintext attacks (CPA)

Public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$

Experiment/game $\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell)$:

Setup:

❶ The challenger generates a bit $b \in_{\mathsf{R}} \{0, 1\}$ and a key pair $(PK, SK) \leftarrow \mathsf{Gen}(1^\ell)$.

❷ The adversary $\mathcal{A}$ is given input 1^ℓ

Rules for the interaction:

❶ The adversary $\mathcal{A}$ is given the public key PK

❷ The adversary $\mathcal{A}$ outputs a pair of messages: $M_0, M_1 \in \{0, 1\}^m$.

❸ The challenger computes $C \leftarrow \mathsf{Enc}_{PK}(M_b)$ and returns C to $\mathcal{A}$

Finally, $\mathcal{A}$ outputs b'. If $b' = b$ then $\mathcal{A}$ has succeeded $\Rightarrow \mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell) = 1$
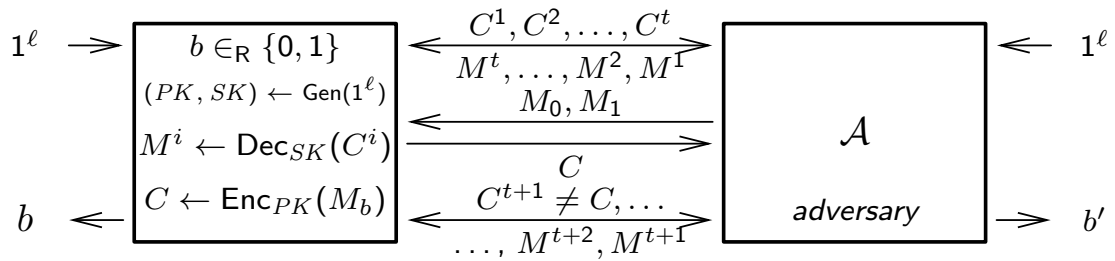
Note that unlike in $\mathsf{PrivK}^{\mathsf{cpa}}$ we do not need to provide $\mathcal{A}$ with any oracle access: here $\mathcal{A}$ has access to the encryption key PK and can evaluate $\mathsf{Enc}_{PK}(\cdot)$ itself.

Security against chosen-ciphertext attacks (CCA)

Public-key encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$

Experiment/game $\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(\ell)$:

Setup:

  ▶ handling of ℓ, b, PK, SK as before

Rules for the interaction:

  ❶ The adversary $\mathcal{A}$ is given PK and oracle access to Dec_{SK}:
  $\mathcal{A}$ outputs C^1, gets $\mathsf{Dec}_{SK}(C^1)$, outputs C^2, gets $\mathsf{Dec}_{SK}(C^2)$, $\ldots$

  ❷ The adversary $\mathcal{A}$ outputs a pair of messages: $M_0, M_1 \in \{0,1\}^m$.

  ❸ The challenger computes $C \leftarrow \mathsf{Enc}_{SK}(M_b)$ and returns C to $\mathcal{A}$

  ❹ The adversary $\mathcal{A}$ continues to have oracle access to Dec_{SK}
  but is not allowed to ask for $\mathsf{Dec}_{SK}(C)$.

Finally, $\mathcal{A}$ outputs b'. If $b' = b$ then $\mathcal{A}$ has succeeded $\Rightarrow \mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(\ell) = 1$

Security against chosen-plaintext attacks (cont'd)

Definition: A public-key encryption scheme Π has *indistinguishable encryptions under a chosen-plaintext attack* (*"is CPA-secure"*) if for all probabilistic, polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl, such that

$$\mathbb{P}(\mathsf{PubK}^{\mathsf{cpa}}_{\mathcal{A},\Pi}(\ell) = 1) \leq \frac{1}{2} + \mathsf{negl}(\ell)$$

Definition: A public-key encryption scheme Π has *indistinguishable encryptions under a chosen-ciphertext attack* (*"is CCA-secure"*) if for all probabilistic, polynomial-time adversaries $\mathcal{A}$ there exists a negligible function negl, such that

$$\mathbb{P}(\mathsf{PubK}^{\mathsf{cca}}_{\mathcal{A},\Pi}(\ell) = 1) \leq \frac{1}{2} + \mathsf{negl}(\ell)$$

What about ciphertext integrity / authenticated encryption?

Since the adversary has access to the public encryption key PK, there is no useful equivalent notion of authenticated encryption for a public-key encryption scheme.

Number theory: integers, divisibility, primes, gcd

Set of integers: $\mathbb{Z} := \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ $a, b \in \mathbb{Z}$

If there exists $c \in \mathbb{Z}$ such that $ac = b$, we say "a divides b" or "$a \mid b$".

- ▶ if $0 < a$ then a is a "divisor" of b
- ▶ if $1 < a < b$ then a is a "factor" of b
- ▶ if a does not divide b, we write "$a \nmid b$"

If integer $p > 1$ has no factors (only 1 and p as divisors), it is "prime", otherwise it is "composite". Primes: $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots$

- ▶ every integer $n > 1$ has a unique prime factorization $n = \prod_i p_i^{e_i}$, with primes p_i and positive integers e_i

The *greatest common divisor* $\gcd(a, b)$ is the largest c with $c \mid a$ and $c \mid b$.

- ▶ examples: $\gcd(18, 12) = 6$, $\gcd(15, 9) = 3$, $\gcd(15, 8) = 1$
- ▶ if $\gcd(a, b) = 1$ we say a and b are "relatively prime"
- ▶ $\gcd(a, b) = \gcd(b, a)$
- ▶ if $c|ab$ and $\gcd(a, c) = 1$ then $c|b$
- ▶ if $a|n$ and $b|n$ and $\gcd(a, b) = 1$ then $ab|n$

Integer division with remainder

For every integer a and positive integer b there exist unique integers q and r with $a = qb + r$ and $0 \leq r < b$.

The modulo operator performs integer division and outputs the remainder:

$$a \bmod b = r \quad \Rightarrow \quad 0 \leq r < b \ \wedge \ \exists q \in \mathbb{Z} : a - qb = r$$

Examples: $7 \bmod 5 = 2$, $-1 \bmod 10 = 9$

If

$$a \bmod n = b \bmod n$$

we say that "a and b are congruent modulo n", and also write

$$a \equiv b \pmod{n}$$

This implies $n|(a - b)$. Being congruent modulo n is an equivalence relationship:

- ▶ reflexive: $a \equiv a \pmod{n}$
- ▶ symmetric: $a \equiv b \pmod{n} \Rightarrow b \equiv a \pmod{n}$
- ▶ transitive: $a \equiv b \pmod{n} \wedge b \equiv c \pmod{n} \Rightarrow a \equiv c \pmod{n}$

Modular arithmetic

Addition, subtraction, and multiplication work the same under congruence modulo n:

If $a \equiv a'$ (mod n) and $b \equiv b'$ (mod n) then

$$a + b \equiv a' + b' \pmod{n}$$
$$a - b \equiv a' - b' \pmod{n}$$
$$ab \equiv a'b' \pmod{n}$$

Associative, commutative and distributive laws also work the same:

$$a(b + c) \equiv ab + ac \equiv ca + ba \pmod{n}$$

When evaluating an expression that is reduced modulo n in the end, we can also reduce any intermediate results. Example:

$$(a - bc) \bmod n = \Big((a \bmod n) - \big((b \bmod n)(c \bmod n)\big) \bmod n \Big) \bmod n$$

Reduction modulo n limits intermediate values to
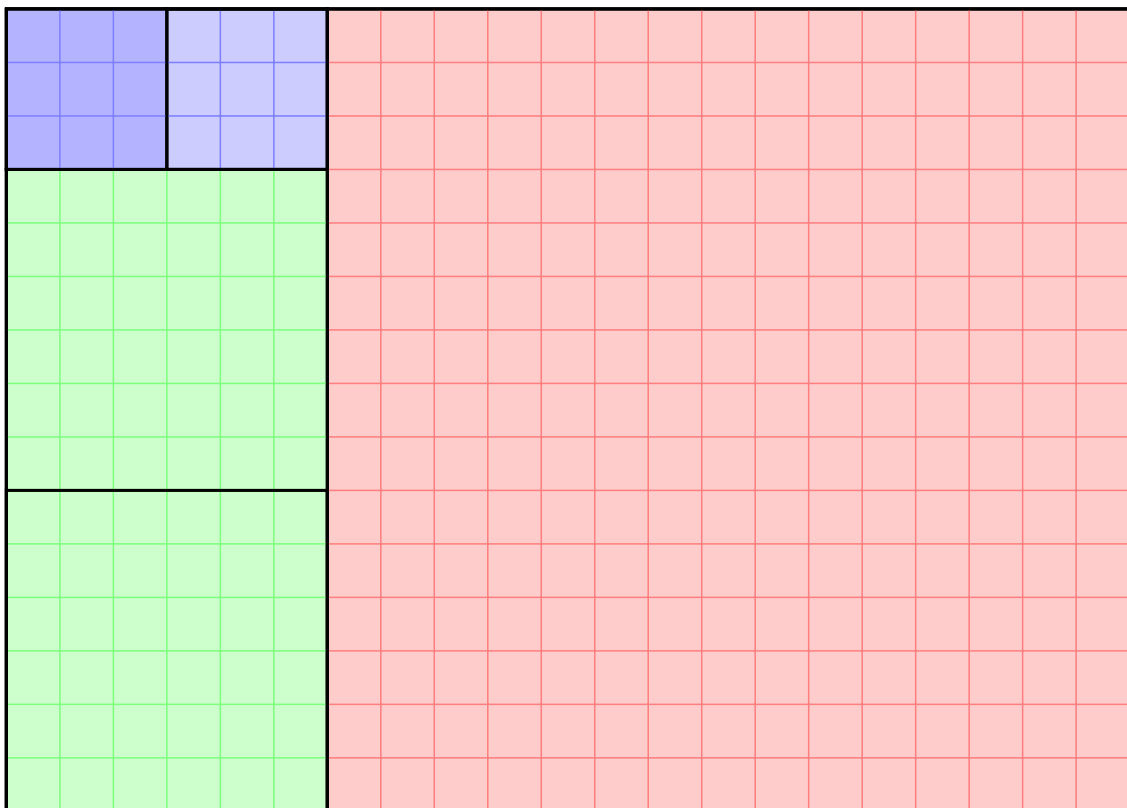
$$\mathbb{Z}_n := \{0, 1, 2, \ldots, n - 1\},$$

the "set of integers modulo n".

Staying within $\mathbb{Z}_n$ helps to limit register sizes and can speed up computation.

Euclid's algorithm

$\gcd(21, 15) = \gcd(15, 21 \bmod 15) = \gcd(15, 6) = \gcd(6, 15 \bmod 6) = \gcd(6, 3) = 3 = -2 \times 21 + 3 \times 15$

Euclid's algorithm

Euclidean algorithm: (WLOG $a \geq b > 0$, since $\gcd(a,b) = \gcd(b,a)$)

$$\gcd(a,b) = \begin{cases} b, & \text{if } b \mid a \\ \gcd(b, a \bmod b), & \text{otherwise} \end{cases}$$

For all positive integers a, b, there exist integers x and y such that $\gcd(a,b) = ax + by$.

Euclid's extended algorithm also provides x and y: (WLOG $a \geq b > 0$)

$(\gcd(a,b), x, y) :=$

$$\text{egcd}(a,b) = \begin{cases} (b, 0, 1), & \text{if } b \mid a \\ (d, y, x - yq), & \text{otherwise,} \\ & \text{with } (d, x, y) := \text{egcd}(b, r), \\ & \text{where } a = qb + r,\ 0 \leq r < b \end{cases}$$

Groups

A **group** $(\mathbb{G}, \bullet)$ is a set $\mathbb{G}$ and an operator $\bullet : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$ that have

closure: $a \bullet b \in \mathbb{G}$ for all $a, b \in \mathbb{G}$

associativity: $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ for all $a, b, c \in \mathbb{G}$

neutral element: there exists an $e \in \mathbb{G}$ such that for all $a \in \mathbb{G}$:
$a \bullet e = e \bullet a = a$

inverse element: for each $a \in \mathbb{G}$ there exists some $b \in \mathbb{G}$ such that
$a \bullet b = b \bullet a = e$

If $a \bullet b = b \bullet a$ for all $a, b \in \mathbb{G}$, the group is called **commutative** (or **abelian**).

Examples of abelian groups:

- $(\mathbb{Z}, +)$, $(\mathbb{R}, +)$, $(\mathbb{R} \setminus \{0\}, \cdot)$
- $(\mathbb{Z}_n, +)$ – set of integers modulo n with addition $a + b := (a + b) \bmod n$
- $(\{0,1\}^n, \oplus)$ where $a_1 a_2 \ldots a_n \oplus b_1 b_2 \ldots b_n = c_1 c_2 \ldots c_n$ with $(a_i + b_i) \bmod 2 = c_i$ (for all $1 \leq i \leq n$, $a_i, b_i, c_i \in \{0,1\}$)   "bit-wise XOR"

If there is no inverse element for each element, $(\mathbb{G}, \bullet)$ is a **monoid** instead.

Examples of monoids:

- $(\mathbb{Z}, \cdot)$ – set of integers under multiplication
- $(\{0,1\}^*, ||)$ – set of variable-length bit strings under concatenation

Permutations and groups

Permutation groups

A set P of permutations over a finite set S forms a group under concatenation if

▶ closure: for any pair of permutations $g, h : S \leftrightarrow S$ in P their concatenation $g \circ h : x \mapsto g(h(x))$ is also in P.

▶ neutral element: the identity function $x \mapsto x$ is in P

▶ inverse element: for each permutation $g \in P$, the inverse permutation g^{-1} is also in P.

Note that function composition is associative: $f \circ (g \circ h) = (f \circ g) \circ h$

The set of all permutations of a set S forms a permutation group called the "symmetric group" on S. Non-trivial symmetric groups ($|S| > 1$) are not abelian.

Each group is isomorphic to a permutation group

Given a group $(\mathbb{G}, \bullet)$, map each $g \in \mathbb{G}$ to a function $f_g : x \mapsto x \bullet g$. Since $g^{-1} \in \mathbb{G}$, f_g is a permutation, and the set of all f_g for $g \in \mathbb{G}$ forms a permutation group isomorphic to $\mathbb{G}$. ("Cayley's theorem")

Encryption schemes are permutations.
Which groups can be used to form encryption schemes?

Subgroups

$(\mathbb{H}, \bullet)$ is a *subgroup* of $(\mathbb{G}, \bullet)$ if

▶ $\mathbb{H}$ is a subset of $\mathbb{G}$ ($\mathbb{H} \subset \mathbb{G}$)

▶ the operator $\bullet$ on $\mathbb{H}$ is the same as on $\mathbb{G}$

▶ $(\mathbb{H}, \bullet)$ is a group, that is

  • for all $a, b \in \mathbb{H}$ we have $a \bullet b \in \mathbb{H}$

  • each element of $\mathbb{H}$ has an inverse element in $\mathbb{H}$

  • the neutral element of $(\mathbb{G}, \bullet)$ is also in $\mathbb{H}$.

Examples of subgroups

▶ $(n\mathbb{Z}, +)$ with $n\mathbb{Z} := \{ni | i \in \mathbb{Z}\} = \{\ldots, -2n, -n, 0, n, 2n, \ldots\}$ – the set of integer multiples of n is a subgroup of $(\mathbb{Z}, +)$

▶ $(\mathbb{R}^+, \cdot)$ – the set of positive real numbers is a subgroup of $(\mathbb{R} \setminus \{0\}, \cdot)$

▶ $(\mathbb{Q}, +)$ is a subgroup of $(\mathbb{R}, +)$, which is a subgroup of $(\mathbb{C}, +)$

▶ $(\mathbb{Q} \setminus \{0\}, \cdot)$ is a subgroup of $(\mathbb{R} \setminus \{0\}, \cdot)$, etc.

▶ $(\{0, 2, 4, 6\}, +)$ is a subgroup of $(\mathbb{Z}_8, +)$

Notations used with groups

When the definition of the group operator is clear from the context, it is often customary to use the symbols of the normal arithmetic addition or multiplication operators ("$+$", "$\times$", "$\cdot$", "ab") for the group operation.

There are two commonly used alternative notations:

"Additive" group: think of group operator as a kind of "$+$"

- ▶ write 0 for the neutral element and $-g$ for the inverse of $g \in \mathbb{G}$.

- ▶ write $g \cdot i := \underbrace{g \bullet g \bullet \cdots \bullet g}_{i \text{ times}}$ $(g \in \mathbb{G}, i \in \mathbb{Z})$

"Multiplicative" group: think of group operator as a kind of "$\times$"

- ▶ write 1 for the neutral element and g^{-1} for the inverse of $g \in \mathbb{G}$.

- ▶ write $g^i := \underbrace{g \bullet g \bullet \cdots \bullet g}_{i \text{ times}}$ $(g \in \mathbb{G}, i \in \mathbb{Z})$

Rings

A **ring** $(\mathbf{R}, \boxplus, \boxtimes)$ is a set $\mathbf{R}$ and two operators $\boxplus : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ and $\boxtimes : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ such that

- ▶ $(\mathbf{R}, \boxplus)$ is an abelian group
- ▶ $(\mathbf{R}, \boxtimes)$ is a monoid
- ▶ $a \boxtimes (b \boxplus c) = (a \boxtimes b) \boxplus (a \boxtimes c)$ and $(a \boxplus b) \boxtimes c = (a \boxtimes c) \boxplus (b \boxtimes c)$ (distributive law)

If also $a \boxtimes b = b \boxtimes a$, then we have a **commutative ring**.

Examples for rings:

- ▶ $(\mathbb{Z}[x], +, \cdot)$, where

$$\mathbb{Z}[x] := \left\{ \sum_{i=0}^{n} a_i x^i \,\middle|\, a_i \in \mathbb{Z}, n \geq 0 \right\}$$

  is the set of polynomials with variable x and coefficients from $\mathbb{Z}$ – commutative

- ▶ $\mathbb{Z}_n[x]$ – the set of polynomials with coefficients from $\mathbb{Z}_n$
- ▶ $(\mathbb{R}^{n \times n}, +, \cdot)$ – $n \times n$ matrices over $\mathbb{R}$ – not commutative

Fields

A **field** $(\mathbb{F}, \boxplus, \boxtimes)$ is a set $\mathbb{F}$ and two operators $\boxplus : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ and $\boxtimes : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ such that

- $(\mathbb{F}, \boxplus)$ is an abelian group with neutral element $0_{\mathbb{F}}$

- $(\mathbb{F} \setminus \{0_{\mathbb{F}}\}, \boxtimes)$ is also an abelian group with neutral element $1_{\mathbb{F}} \neq 0_{\mathbb{F}}$

- $a \boxtimes (b \boxplus c) = (a \boxtimes b) \boxplus (a \boxtimes c)$ and $(a \boxplus b) \boxtimes c = (a \boxtimes c) \boxplus (b \boxtimes c)$ (distributive law)

In other words: a field is a commutative ring where each element except for the neutral element of the addition has a multiplicative inverse.

Field means: division works, linear algebra works, solving equations, etc.

Examples for fields:

- $(\mathbb{Q}, +, \cdot)$

- $(\mathbb{R}, +, \cdot)$

- $(\mathbb{C}, +, \cdot)$

Ring $\mathbb{Z}_n$

Set of *integers modulo* n is $\mathbb{Z}_n := \{0, 1, \ldots, n-1\}$

When we refer to $(\mathbb{Z}_n, +)$ or $(\mathbb{Z}_n, \cdot)$, we apply after each addition or multiplication a reduction modulo n. (No need to write out "mod n" each time.)

We add/subtract the integer multiple of n needed to get the result back into $\mathbb{Z}_n$.

$(\mathbb{Z}_n, +)$ is an abelian group:

- neutral element of addition is 0

- the inverse element of $a \in \mathbb{Z}_n$ is $n - a \equiv -a \pmod{n}$

$(\mathbb{Z}_n, \cdot)$ is a monoid:

- neutral element of multiplication is 1

$(\mathbb{Z}_n, +, \cdot)$, with its "mod n" operators, is a ring, which means commutative, associative and distributive law works just like over $\mathbb{Z}$.

From now on, when we refer to $\mathbb{Z}_n$, we usually imply that we work with the commutative ring $(\mathbb{Z}_n, +, \cdot)$.

Examples in $\mathbb{Z}_5$: $4 + 3 = 2$, $4 \cdot 2 = 3$, $4^2 = 1$

Division in $\mathbb{Z}_n$

In ring $\mathbb{Z}_n$, element a has a multiplicative inverse a^{-1} (with $aa^{-1} = 1$) if and only if $\gcd(n, a) = 1$.

In this case, the extended Euclidian algorithm gives us

$$nx + ay = 1$$

and since $nx = 0$ in $\mathbb{Z}_n$ for all x, we have $ay = 1$.

Therefore $y = a^{-1}$ is the inverse needed for dividing by a.

- ▶ We call the set of all elements in $\mathbb{Z}_n$ that have a multiplicative inverse the "multiplicative group" of $\mathbb{Z}_n$:

$$\mathbb{Z}_n^* := \{a \in \mathbb{Z}_n \mid \gcd(n, a) = 1\}$$

- ▶ If p is prime, then $(\mathbb{Z}_p^*, \cdot)$ with

$$\mathbb{Z}_p^* = \{1, \ldots, p-1\}$$

is a group, and $(\mathbb{Z}_p, +, \cdot)$ is a (finite) field, that is every element except 0 has a multiplicative inverse.

Example: Multiplicative inverses of $\mathbb{Z}_7^*$:
$1 \cdot 1 = 1$, $2 \cdot 4 = 1$, $3 \cdot 5 = 1$, $4 \cdot 2 = 1$, $5 \cdot 3 = 1$, $6 \cdot 6 = 1$

Finite fields (Galois fields)

$(\mathbb{Z}_p, +, \cdot)$ is a finite field with p elements, where p is a prime number. Also written as $\mathsf{GF}(p)$, the "Galois field of order p".

We can also construct finite fields $\mathsf{GF}(p^n)$ with p^n elements:

- ▶ **Elements:** polynomials over variable x with degree less than n and coefficients from the finite field $\mathbb{Z}_p$

- ▶ **Modulus:** select an *irreducible* polynomial $T(x) \in \mathbb{Z}_p[x]$ of degree n

$$T(x) = c_n x^n + \cdots + c_2 x^2 + c_1 x + c_0$$

where $c_i \in \mathbb{Z}_p$ for all $0 \leq i \leq n$. An irreducible polynomial cannot be factored into two other polynomials from $\mathbb{Z}_p[x] \setminus \{0, 1\}$.

- ▶ **Addition:** $\oplus$ is normal polynomial addition (i.e., pairwise addition of the coefficients in $\mathbb{Z}_p$)

- ▶ **Multiplication:** $\otimes$ is normal polynomial multiplication, then divide by $T(x)$ and take the remainder (i.e., multiplication modulo $T(x)$).

Theorem: any finite field has p^n elements (p prime, $n > 0$)
Theorem: all finite fields of the same size are isomorphic

GF(2^n)

GF(2) is particularly easy to implement in hardware:

- ▶ addition = subtraction = XOR gate

- ▶ multiplication = AND gate

- ▶ division can only be by 1, which merely results in the first operand

Of particular practical interest in modern cryptography are larger finite fields of the form GF(2^n):

- ▶ Polynomials are represented as bit words, each coefficient = 1 bit.

- ▶ Addition/subtraction is implemented via bit-wise XOR instruction.

- ▶ Multiplication and division of binary polynomials is like binary integer multiplication and division, but *without carry-over bits*. This allows the circuit to be clocked much faster.

Recent Intel/AMD CPUs have added instruction PCLMULQDQ for 64×64-bit carry-less multiplication. This helps to implement arithmetic in GF(2^{64}) or GF(2^{128}) more efficiently.

GF(2^8) example

The finite field GF(2^8) consists of the 256 polynomials of the form

$$c_7 x^7 + \cdots + c_2 x^2 + c_1 x + c_0 \qquad c_i \in \{0, 1\}$$

each of which can be represented by the byte $c_7 c_6 c_5 c_4 c_3 c_2 c_1 c_0$.

As modulus we chose the irreducible polynomial

$$T(x) = x^8 + x^4 + x^3 + x + 1 \quad \text{or} \quad 1\,0001\,1011$$

Example operations:

- ▶ $(x^7 + x^5 + x + 1) \oplus (x^7 + x^6 + 1) = x^6 + x^5 + x$
  or equivalently $1010\,0011 \oplus 1100\,0001 = 0110\,0010$

- ▶ $(x^6 + x^4 + 1) \otimes_T (x^2 + 1) = [(x^6 + x^4 + 1)(x^2 + 1)] \bmod T(x) =$
  $(x^8 + x^4 + x^2 + 1) \bmod (x^8 + x^4 + x^3 + x + 1) =$
  $(x^8 + x^4 + x^2 + 1) \ominus (x^8 + x^4 + x^3 + x + 1) = x^3 + x^2 + x$
  or equivalently
  $0101\,0001 \otimes_T 0000\,0101 = 1\,0001\,0101 \oplus 1\,0001\,1011 = 0000\,1110$

Finite groups

Let $(\mathbb{G}, \bullet)$ be a group with a finite number of elements $|\mathbb{G}|$.

Practical examples here: $(\mathbb{Z}_n, +)$, $(\mathbb{Z}_n^*, \cdot)$, $(\mathrm{GF}(2^n), \oplus)$, $(\mathrm{GF}(2^n) \setminus \{0\}, \otimes)$

Terminology:

- The *order of a group* $\mathbb{G}$ is its size $|\mathbb{G}|$

- *order of group element* g in $\mathbb{G}$ is $\mathrm{ord}_{\mathbb{G}}(g) = \min\{i > 0 \mid g^i = 1\}$.

Related notion: the *characteristic of a ring* is the order of 1 in its additive group, i.e. the smallest i with $\underbrace{1 + 1 + \cdots + 1}_{i \text{ times}} = 0$.

Useful facts regarding any element $g \in \mathbb{G}$ in a group of order $m = |\mathbb{G}|$:

❶ $g^m = 1$, $g^x = g^{x \bmod m}$

❷ $g^x = g^{x \bmod \mathrm{ord}(g)}$

❸ $g^x = g^y \Leftrightarrow x \equiv y \pmod{\mathrm{ord}(g)}$

❹ $\mathrm{ord}(g) \mid m$ "Lagrange's theorem"

❺ if $\gcd(e, m) = 1$ then $g \mapsto g^e$ is a permutation, and $g \mapsto g^d$ its inverse (i.e., $g^{ed} = g$) if $ed \bmod m = 1$

Proofs

⓿ In any group $(\mathbb{G}, \cdot)$ with $a, b, c \in \mathbb{G}$ we have $ac = bc \Rightarrow a = b$.
Proof: $ac = bc \Rightarrow (ac)c^{-1} = (bc)c^{-1} \Rightarrow a(cc^{-1}) = b(cc^{-1}) \Rightarrow a \cdot 1 = b \cdot 1 \Rightarrow a = b$.

❶ Let $\mathbb{G}$ be an abelian group of order m with elements $g_1, \ldots, g_m$. We have
$$g_1 \cdot g_2 \cdots g_m = (gg_1) \cdot (gg_2) \cdots (gg_m)$$
for arbitrary fixed $g \in \mathbb{G}$, because $gg_i = gg_j \Rightarrow g_i = g_j$ (see ⓿), which implies that each of the (gg_i) is distinct, and since there are only m elements of $\mathbb{G}$, the right-hand side of the above equation is just a permutation of the left-hand side. Now pull out the g:
$$g_1 \cdot g_2 \cdots g_m = (gg_1) \cdot (gg_2) \cdots (gg_m) = g^m \cdot g_1 \cdot g_2 \cdots g_m \Rightarrow g^m = 1.$$
(Not shown here: $g^m = 1$ also holds for non-commutative groups.)

Also: $g^m = 1 \Rightarrow g^x = g^x \cdot (g^m)^n = g^{x-nm} = g^{x \bmod m}$ for any $n \in \mathbb{Z}$.

❷ Likewise: $i = \mathrm{ord}(g) \Rightarrow g^i = 1 \Rightarrow g^x = g^x \cdot (g^i)^n = g^{x+ni} = g^{x \bmod i}$ for any $n \in \mathbb{Z}$.

❸ Let $i = \mathrm{ord}(g)$.
"⇐": $x \equiv y \pmod{i} \Leftrightarrow x \bmod i = y \bmod i \Rightarrow g^x = g^{x \bmod i} = g^{y \bmod i} = g^y$.
"⇒": Say $g^x = g^y$, then $1 = g^{x-y} = g^{(x-y) \bmod i}$. Since $(x - y) \bmod i < i$, but i is the smallest positive integer with $g^i = 1$, we must have $(x - y) \bmod i = 0$. $\Rightarrow x \equiv y \pmod{i}$.

❹ $g^m = 1 = g^0$ therefore $m \equiv 0 \pmod{\mathrm{ord}(g)}$ from ❸, and so $\mathrm{ord}(g) \mid m$.

❺ $(g^e)^d = g^{ed} = g^{ed \bmod m} = g^1 = g$ means that $g \mapsto g^d$ is indeed the inverse of $g \mapsto g^e$ if $ed \bmod m = 1$. And since $\mathbb{G}$ is finite, the existence of an inverse operation implies that $g \mapsto g^e$ is a permutation.

Katz/Lindell, sections 8.1 and 8.3

Cyclic groups

Let $\mathbb{G}$ be a finite (multiplicative) group of order $m = |\mathbb{G}|$.

For $g \in \mathbb{G}$ consider the set

$$\langle g \rangle := \{g^0, g^1, g^2, \ldots\}$$

Note that $|\langle g \rangle| = \mathrm{ord}(g)$ and $\langle g \rangle = \{g^0, g^1, g^2, \ldots, g^{\mathrm{ord}(g)-1}\}$.

Definitions:

- ▶ We call g a *generator* of $\mathbb{G}$ if $\langle g \rangle = \mathbb{G}$.
- ▶ We call $\mathbb{G}$ *cyclic* if it has a generator.

Useful facts:

- ❶ Every cyclic group of order m is isomorphic to $(\mathbb{Z}_m, +)$. $(g^i \leftrightarrow i)$
- ❷ $\langle g \rangle$ is a subgroup of $\mathbb{G}$ (subset, a group under the same operator)
- ❸ If $|\mathbb{G}|$ is prime, then $\mathbb{G}$ is cyclic and all $g \in \mathbb{G} \setminus \{1\}$ are generators.

  Recall that $\mathrm{ord}(g) \mid |\mathbb{G}|$. We have $\mathrm{ord}(g) \in \{1, |\mathbb{G}|\}$ if $|\mathbb{G}|$ is prime, which makes g either 1 or a generator.

Katz/Lindell, section 8.3

How to find a generator?

Let $\mathbb{G}$ be a cyclic (multiplicative) group of order $m = |\mathbb{G}|$.

- ▶ If m is prime, any non-neutral element is a generator. Done.
  But $|\mathbb{Z}_p^*| = p - 1$ is not prime (for $p > 3$)!

- ▶ Directly testing for $|\langle g \rangle| \overset{?}{=} m$ is infeasible for crypto-sized m.

- ▶ Fast test: if $m = \prod_i p_i^{e_i}$ is composite, then $g \in \mathbb{G}$ is a generator if and only if $g^{m/p_i} \neq 1$ for all i.

- ▶ Sampling a polynomial number of elements of $\mathbb{G}$ for the above test will lead to a generator in polynomial time (of $\log_2 m$) with all but negligible probability.

$\Rightarrow$ Make sure you pick a group of an order with known prime factors.

One possibility for $\mathbb{Z}_p^*$ (commonly used):

- ▶ Chose a "strong prime" $p = 2q + 1$, where q is also prime
  $\Rightarrow |\mathbb{Z}_p^*| = p - 1 = 2q$ has prime factors 2 and q.

$(\mathbb{Z}_p, +)$ is a cyclic group

For every prime p every element $g \in \mathbb{Z}_p \setminus \{0\}$ is a generator:

$$\mathbb{Z}_p = \langle g \rangle = \{g \cdot i \bmod p \,|\, 0 \le i \le p-1\}$$

Note that this follows from fact 3 on slide 67: $\mathbb{Z}_p$ is of order p, which is prime.

Example in $\mathbb{Z}_7$:

$$(1 \cdot 0,\ 1 \cdot 1,\ 1 \cdot 2,\ 1 \cdot 3,\ 1 \cdot 4,\ 1 \cdot 5,\ 1 \cdot 6) = (0,\ 1,\ 2,\ 3,\ 4,\ 5,\ 6)$$
$$(2 \cdot 0,\ 2 \cdot 1,\ 2 \cdot 2,\ 2 \cdot 3,\ 2 \cdot 4,\ 2 \cdot 5,\ 2 \cdot 6) = (0,\ 2,\ 4,\ 6,\ 1,\ 3,\ 5)$$
$$(3 \cdot 0,\ 3 \cdot 1,\ 3 \cdot 2,\ 3 \cdot 3,\ 3 \cdot 4,\ 3 \cdot 5,\ 3 \cdot 6) = (0,\ 3,\ 6,\ 2,\ 5,\ 1,\ 4)$$
$$(4 \cdot 0,\ 4 \cdot 1,\ 4 \cdot 2,\ 4 \cdot 3,\ 4 \cdot 4,\ 4 \cdot 5,\ 4 \cdot 6) = (0,\ 4,\ 1,\ 5,\ 2,\ 6,\ 3)$$
$$(5 \cdot 0,\ 5 \cdot 1,\ 5 \cdot 2,\ 5 \cdot 3,\ 5 \cdot 4,\ 5 \cdot 5,\ 5 \cdot 6) = (0,\ 5,\ 3,\ 1,\ 6,\ 4,\ 2)$$
$$(6 \cdot 0,\ 6 \cdot 1,\ 6 \cdot 2,\ 6 \cdot 3,\ 6 \cdot 4,\ 6 \cdot 5,\ 6 \cdot 6) = (0,\ 6,\ 5,\ 4,\ 3,\ 2,\ 1)$$

- ▶ All the non-zero elements of $\mathbb{Z}_7$ are generators
- ▶ $\mathrm{ord}(0) = 1$, $\mathrm{ord}(1) = \mathrm{ord}(2) = \mathrm{ord}(3) = \mathrm{ord}(4) = \mathrm{ord}(5) = \mathrm{ord}(6) = 7$

$(\mathbb{Z}_p^*, \cdot)$ is a cyclic group

For every prime p there exists a *generator* $g \in \mathbb{Z}_p^*$ such that

$$\mathbb{Z}_p^* = \{g^i \bmod p \,|\, 0 \le i \le p-2\}$$

Note that this does **not** follow from fact 3 on slide 67: $\mathbb{Z}_p^*$ is of order $p-1$, which is even (for $p > 3$), not prime.

Example in $\mathbb{Z}_7^*$:

$$(1^0, 1^1, 1^2, 1^3, 1^4, 1^5) = (1,1,1,1,1,1)$$
$$(2^0, 2^1, 2^2, 2^3, 2^4, 2^5) = (1,2,4,1,2,4)$$
$$(3^0, 3^1, 3^2, 3^3, 3^4, 3^5) = (1,3,2,6,4,5)$$
$$(4^0, 4^1, 4^2, 4^3, 4^4, 4^5) = (1,4,2,1,4,2)$$
$$(5^0, 5^1, 5^2, 5^3, 5^4, 5^5) = (1,5,4,6,2,3)$$
$$(6^0, 6^1, 6^2, 6^3, 6^4, 6^5) = (1,6,1,6,1,6)$$

- ▶ 3 and 5 are generators of $\mathbb{Z}_7^*$
  Fast generator test (p. 68), using $|\mathbb{Z}_7^*| = 6 = 2 \cdot 3$:
  $3^{6/2} = 6, 3^{6/3} = 2, 5^{6/2} = 6, 5^{6/3} = 4$, all $\ne 1$.
- ▶ 1, 2, 4, 6 generate *subgroups* of $\mathbb{Z}_7^*$: $\{1\}, \{1,2,4\}, \{1,2,4\}, \{1,6\}$
- ▶ $\mathrm{ord}(1) = 1$, $\mathrm{ord}(2) = 3$, The *order* of g in $\mathbb{Z}_p^*$ is the size of the subgroup $\langle g \rangle$.
  $\mathrm{ord}(3) = 6$, $\mathrm{ord}(4) = 3$, Lagrange's theorem: $\mathrm{ord}_{\mathbb{Z}_p^*}(g) \mid p-1$ for all $g \in \mathbb{Z}_p^*$
  $\mathrm{ord}(5) = 6$, $\mathrm{ord}(6) = 2$

Fermat's and Euler's theorem

Fermat's little theorem: (1640)

$$p \text{ prime and } \gcd(a,p) = 1 \quad \Rightarrow \quad a^{p-1} \bmod p = 1$$

Recall from Lagrange's theorem: for $a \in \mathbb{Z}_p^*$, $\mathrm{ord}(a)|(p-1)$ since $|\mathbb{Z}_p^*| = p - 1$.

Euler's phi function:

$$\varphi(n) = |\mathbb{Z}_n^*| = |\{a \in \mathbb{Z}_n \mid \gcd(n,a) = 1\}|$$

- Example: $\varphi(12) = |\{1, 5, 7, 11\}| = 4$
- primes p, q:
$$\varphi(p) = p - 1$$
$$\varphi(p^k) = p^{k-1}(p-1)$$
$$\varphi(pq) = (p-1)(q-1)$$

- $\gcd(a, b) = 1 \Rightarrow \varphi(ab) = \varphi(a)\varphi(b)$

Euler's theorem: (1763)

$$\gcd(a, n) = 1 \quad \Leftrightarrow \quad a^{\varphi(n)} \bmod n = 1$$

- this implies that in $\mathbb{Z}_n$: $a^x = a^{x \bmod \varphi(n)}$ for any $a \in \mathbb{Z}_n, x \in \mathbb{Z}$

Recall from Lagrange's theorem: for $a \in \mathbb{Z}_n^*$, $\mathrm{ord}(a)|\varphi(n)$ since $|\mathbb{Z}_n^*| = \varphi(n)$.

Chinese remainder theorem

Definition: Let $(\mathbb{G}, \bullet)$ and $(\mathbb{H}, \circ)$ be two groups. A function $f : \mathbb{G} \to \mathbb{H}$ is an *isomorphism* from $\mathbb{G}$ to $\mathbb{H}$ if

- f is a 1-to-1 mapping (bijection)
- $f(g_1 \bullet g_2) = f(g_1) \circ f(g_2)$ for all $g_1, g_2 \in \mathbb{G}$

Chinese remainder theorem:
For any p, q with $\gcd(p, q) = 1$ and $n = pq$, the mapping

$$f : \mathbb{Z}_n \leftrightarrow \mathbb{Z}_p \times \mathbb{Z}_q \qquad f(x) = (x \bmod p, x \bmod q)$$

is an isomorphism, both from $\mathbb{Z}_n$ to $\mathbb{Z}_p \times \mathbb{Z}_q$ and from $\mathbb{Z}_n^*$ to $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.

Inverse: To get back from $x_p = x \bmod p$ and $x_q = x \bmod q$ to x, we first use Euclid's extended algorithm to find a, b such that $ap + bq = 1$, and then $x = (x_p bq + x_q ap) \bmod n$.

Application: arithmetic operations on $\mathbb{Z}_n$ can instead be done on both $\mathbb{Z}_p$ and $\mathbb{Z}_q$ after this mapping, which may be faster.

Example: $n = pq = 3 \times 5 = 15$

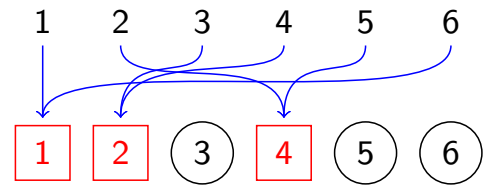| x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$x \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
$x \bmod 5$	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

Quadratic residues in $(\mathbb{Z}_p^*, \cdot)$

In $\mathbb{Z}_p^*$, the squaring of an element, $x \mapsto x^2$ is a 2-to-1 function:

$$y = x^2 = (-x)^2$$

Example in $\mathbb{Z}_7^*$:

$$(1^2, 2^2, 3^2, 4^2, 5^2, 6^2) = (1, 4, 2, 2, 4, 1)$$

If y is the square of a number in $x \in \mathbb{Z}_p^*$, that is if y has a square root in $\mathbb{Z}_p^*$, we call y a "quadratic residue".

Example: $\mathbb{Z}_7^*$ has 3 quadratic residues: $\{1, 2, 4\}$.

If p is an odd prime: $\mathbb{Z}_p^*$ has $(p-1)/2$ quadratic residues.
$\mathbb{Z}_p$ would have one more: 0

Euler's criterion:

$$c^{(p-1)/2} \bmod p = 1 \quad \Leftrightarrow \quad c \text{ is a quadratic residue in } \mathbb{Z}_p^*$$

Example in $\mathbb{Z}_7$: $(7-1)/2 = 3$, $\quad (1^3, 2^3, 3^3, 4^3, 5^3, 6^3) = (1, 1, 6, 1, 6, 6)$

$c^{(p-1)/2}$ is also called the *Legendre symbol*

Taking roots in $\mathbb{Z}_p^*$

If $x^e = c$ in $\mathbb{Z}_p$, then x is the "e^{th} root of c", or $x = c^{1/e}$.

Case 1: $\gcd(e, p-1) = 1$
Find d with $de = 1$ in $\mathbb{Z}_{p-1}$ (Euclid's extended), then $c^{1/e} = c^d$ in $\mathbb{Z}_p^*$.
Proof: $(c^d)^e = c^{de} = c^{de \bmod \varphi(p)} = c^{de \bmod (p-1)} = c^1 = c.$

Case 2: $e = 2$ (taking square roots)
$\gcd(2, p-1) \neq 1$ if p odd prime $\Rightarrow$ Euclid's extended alg. no help here.

- If $p \bmod 4 = 3$ and $c \in \mathbb{Z}_p^*$ is a quadratic residue: $\sqrt{c} = c^{(p+1)/4}$

  Proof: $\left[c^{(p+1)/4}\right]^2 = c^{(p+1)/2} = \underbrace{c^{(p-1)/2}}_{=1} \cdot c = c.$

- If $p \bmod 4 = 1$ this can also be done efficiently (details omitted).

Application: solve quadratic equations $ax^2 + bx + c = 0$ in $\mathbb{Z}_p$

Solution: $x = \dfrac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Algorithms: $\sqrt{b^2 - 4ac}$ as above, $(2a)^{-1}$ using Euclid's extended

Taking roots in $\mathbb{Z}_n^*$: If n is composite, then we know how to test whether $c^{1/e}$ exists, and how to compute it efficiently, **only** if we know the prime factors of n. Basic Idea: apply Chinese Remainder Theorem, then apply above techniques for $\mathbb{Z}_p^*$.

Working in subgroups of $\mathbb{Z}_p^*$

How can we construct a cyclic finite group $\mathbb{G}$ where all non-neutral elements are generators?

Recall that $\mathbb{Z}_p^*$ has $q = (p-1)/2$ quadratic residues, exactly half of its elements.

Quadratic residue: an element that is the square of some other element.

Choose p to be a *strong prime*, that is where q is also prime.

Let $\mathbb{G} = \{g^2 \mid g \in \mathbb{Z}_p^*\}$ be the set of quadratic residues of $\mathbb{Z}_p^*$. $\mathbb{G}$ with operator "multiplication mod p" is a subgroup of $\mathbb{Z}_p^*$, with order $|\mathbb{G}| = q$.

$\mathbb{G}$ has prime order $|\mathbb{G}| = q$ and $\mathrm{ord}(g)|q$ for all $g \in \mathbb{G}$ (Lagrange's theorem):
$\Rightarrow \mathrm{ord}(g) \in \{1, q\} \Rightarrow \mathrm{ord}(g) = q$ for all $g > 1 \Rightarrow$ for all $g \in \mathbb{G} \setminus \{1\}$ $\langle g \rangle = \mathbb{G}$.

If p is a strong prime, then each quadratic residue in $\mathbb{Z}_p^*$ other than 1 is a generator of the subgroup of quadratic residues of $\mathbb{Z}_p^*$.

$\textsc{Generate\_group}(1^\ell)$:
  $p \in_{\mathsf{R}} \{(\ell+1)\text{-bit strong primes}\}$
  $q := (p-1)/2$
  $x \in_{\mathsf{R}} \mathbb{Z}_p^* \setminus \{-1, 1\}$
  $g := x^2 \bmod p$
  return p, q, g

Example: $p = 11$, $q = 5$
$g \in \{2^2, 3^2, 4^2, 5^2\} = \{4, 9, 5, 3\}$
$\langle 4 \rangle = \{4^0, 4^1, 4^2, 4^3, 4^4\} = \{1, 4, 5, 9, 3\}$
$\langle 9 \rangle = \{9^0, 9^1, 9^2, 9^3, 9^4\} = \{1, 9, 4, 3, 5\}$
$\langle 5 \rangle = \{5^0, 5^1, 5^2, 5^3, 5^4\} = \{1, 5, 3, 4, 9\}$
$\langle 3 \rangle = \{3^0, 3^1, 3^2, 3^3, 3^4\} = \{1, 3, 9, 5, 4\}$

Modular exponentiation

In cyclic group $(\mathbb{G}, \bullet)$ (e.g., $\mathbb{G} = \mathbb{Z}_p^*$):
How do we calculate g^e efficiently? ($g \in \mathbb{G}$, $e \in \mathbb{N}$)

Naive algorithm: $g^e = \underbrace{g \bullet g \bullet \cdots \bullet g}_{e \text{ times}}$

Far too slow for crypto-size e (e.g., $e \approx 2^{256}$)!

Square and multiply algorithm:

Binary representation: $e = \sum_{i=0}^{n} e_i \cdot 2^i$, $\quad n = \lfloor \log_2 e \rfloor$, $\quad e_i = \left\lfloor \frac{e}{2^i} \right\rfloor \bmod 2$

Computation:

$$g^{2^0} := g, \quad g^{2^i} := \left(g^{2^{i-1}}\right)^2$$

$$g^e := \prod_{i=0}^{n} \left(g^{2^i}\right)^{e_i}$$

Side-channel vulnerability: the **if** statement leaks the binary representation of e. "Montgomery's ladder" is an alternative algorithm with fixed control flow.

$\textsc{Square\_and\_multiply}(g, e)$:
  $a := g$
  $b := 1$
  for $i := 0$ to n do
    if $\lfloor e/2^i \rfloor \bmod 2 = 1$ then
      $b := b \bullet a$ $\leftarrow$ multiply
    $a := a \bullet a$ $\leftarrow$ square
  return b

Discrete logarithm problem

Let $(\mathbb{G}, \bullet)$ be a given cyclic group of order $q = |\mathbb{G}|$ with given generator g ($\mathbb{G} = \{g^0, g^1, \ldots, g^{q-1}\}$). The "discrete logarithm problem (DLP)" is finding for a given $y \in \mathbb{G}$ the number $x \in \mathbb{Z}_q$ such that

$$g^x = \underbrace{g \bullet g \bullet \cdots \bullet g}_{x \text{ times}} = y$$

If $(\mathbb{G}, \bullet)$ is clear from context, we can write $x = \log_g y$. For any x' with $g^{x'} = y$, we have $x = x' \bmod q$. Discrete logarithms behave similar to normal logarithms: $\log_g 1 = 0$ (if 1 is the neutral element of $\mathbb{G}$), $\log_g h^r = (r \cdot \log_g h) \bmod q$, and $\log_g h_1 h_2 = (\log_g h_1 + \log_g h_2) \bmod q$.

For cryptographic applications, we require groups with

- ▶ a probabilistic polynomial-time group-generation algorithm $\mathcal{G}(1^\ell)$ that outputs a description of $\mathbb{G}$ with $\lceil \log_2 |\mathbb{G}| \rceil = \ell$;

- ▶ a description that defines how each element of $\mathbb{G}$ is represented uniquely as a bit pattern;

- ▶ efficient (polynomial time) algorithms for $\bullet$, for picking an element of $\mathbb{G}$ uniformly at random, and for testing whether a bit pattern represents an element of $\mathbb{G}$;

Hard discrete logarithm problems

The discrete logarithm experiment $\mathrm{DLog}_{\mathcal{G},\mathcal{A}}(\ell)$:

1. Run $\mathcal{G}(1^\ell)$ to obtain $(\mathbb{G}, q, g)$, where $\mathbb{G}$ is a cyclic group of order q ($2^{\ell-1} < q \leq 2^\ell$) and g is a generator of $\mathbb{G}$

2. Choose uniform $h \in \mathbb{G}$.

3. Give $(\mathbb{G}, q, g, h)$ to $\mathcal{A}$, which outputs $x \in \mathbb{Z}_q$

4. Return 1 if $g^x = h$, otherwise return 0

We say "the discrete-logarithm problem is hard relative to $\mathcal{G}$" if for all probabilistic polynomial-time algorithms $\mathcal{A}$ there exists a negligible function negl, such that $\mathbb{P}(\mathrm{DLog}_{\mathcal{G},\mathcal{A}}(\ell) = 1) \leq \mathrm{negl}(\ell)$.

Diffie–Hellman problems

Let $(\mathbb{G}, \bullet)$ be a cyclic group of order $q = |\mathbb{G}|$ with generator g ($\mathbb{G} = \{g^0, g^1, \ldots, g^{q-1}\}$). Given elements $h_1, h_2 \in \mathbb{G}$, define

$$\mathsf{DH}(h_1, h_2) := g^{\log_g h_1 \cdot \log_g h_2}$$

that is if $g^{x_1} = h_1$ and $g^{x_2} = h_2$, then $\mathsf{DH}(h_1, h_2) = g^{x_1 \cdot x_2} = h_1^{x_2} = h_2^{x_1}$.

These two problems are related to the discrete logarithm problem:

▶ **Computational Diffie–Hellman (CDH) problem:** the adversary is given uniformly chosen $h_1, h_2 \in \mathbb{G}$ and has to output $\mathsf{DH}(h_1, h_2)$.

The problem is hard if for all PPT $\mathcal{A}$ we have $\mathbb{P}(\mathcal{A}(\mathbb{G}, q, g, g^x, g^y) = g^{xy}) \le \mathsf{negl}(\ell)$.

▶ **Decision Diffie–Hellman (DDH) problem:** the adversary is given $h_1, h_2 \in \mathbb{G}$ chosen uniformly at random, plus another value $h' \in \mathbb{G}$, which is either equal to $\mathsf{DH}(h_1, h_2)$, or was chosen uniformly at random, and has to decide which of the two cases applies.

The problem is hard if for all PPT $\mathcal{A}$ and uniform $x, y, z \in \mathbb{G}$ we have $|\mathbb{P}(\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1) - \mathbb{P}(\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1)| \le \mathsf{negl}(\ell)$.

If the discrete-logarithm problem is not hard for $\mathbb{G}$, then neither will be the CDH problem, and if the latter is not hard, neither will be the DDH problem.

Diffie-Hellman key exchange

How can two parties achieve message confidentiality who have no prior shared secret and no secure channel to exchange one?

Select a cyclic group $\mathbb{G}$ of order q and a generator $g \in \mathbb{G}$, which can be made public and fixed system wide. A generates x and B generates y, both chosen uniformly at random out of $\{1, \ldots, q-1\}$. Then they exchange two messages:

$$\begin{aligned} A \to B : & \quad g^x \\ B \to A : & \quad g^y \end{aligned}$$

Now both can form $(g^x)^y = (g^y)^x = g^{xy}$ and use a hash $h(g^{xy})$ as a shared private key (e.g. with an authenticated encryption scheme).

The eavesdropper faces the computational Diffie–Hellman problem of determining g^{xy} from g^x, g^y and g.

The DH key exchange is secure against a passive eavesdropper, but not against middleperson attacks, where g^x and g^y are replaced by the attacker with other values.

W. Diffie, M.E. Hellman: New Directions in Cryptography. IEEE IT-22(6), 1976-11, pp 644–654.

Discrete logarithm algorithms

Several generic algorithms are known for solving the discrete logarithm problem for any cyclic group $\mathbb{G}$ of order q:

- **Trivial brute-force algorithm:** try all g^i, time $|\langle g \rangle| = \text{ord}(g) \leq q$.

- **Pohlig–Hellman algorithm:** if q is not prime, and has a known (or easy to determine) factorization, then this algorithm reduces the discrete-logarithm problem for $\mathbb{G}$ to discrete-logarithm problems for prime-order subgroups of $\mathbb{G}$.
  $\Rightarrow$ the difficulty of finding the discrete logarithm in a group of order q is no greater than that of finding it in a group of order q', where q' is the largest prime factor dividing q.

- **Shank's baby-step/giant-step algorithm:** requires $O(\sqrt{q} \cdot \text{polylog}(q))$ time and $O(\sqrt{q})$ memory.

- **Pollard's rho algorithm:** requires $O(\sqrt{q} \cdot \text{polylog}(q))$ time and $O(1)$ memory.

$\Rightarrow$ choose $\mathbb{G}$ to have a **prime order** q, and make q large enough such that no adversary can be expected to execute $\sqrt{q}$ steps (e.g. $q \gg 2^{200}$).

Baby-step/giant-step algorithm

Given generator $g \in \mathbb{G}$ ($|\mathbb{G}| = q$) and $y \in \mathbb{G}$, find $x \in \mathbb{Z}_q$ with $g^x = y$.

- Powers of g form a cycle $1 = g^0, g^1, g^2, \ldots, g^{q-2}, g^{q-1}, g^q = 1$, and $y = g^x$ sits on this cycle.

- Go around cycle in "giant steps" of $n = \lfloor \sqrt{q} \rfloor$:

$$g^0, g^n, g^{2n}, \ldots, g^{\lceil q/n \rceil n}$$

  Store all values encountered in a lookup table $L[g^{kn}] := k$.
  Memory: $\sqrt{q}$, runtime: $\sqrt{q}$, (times log. lookup table insertion)

- Go around cycle in "baby steps", starting at y

$$y \cdot g^1, y \cdot g^2, \ldots, y \cdot g^n$$

  until we find one of these values in the table L: $L[y \cdot g^i] = k$.
  Runtime: $\sqrt{q}$ (times log. table lookup)

- Now we know $y \cdot g^i = g^{kn}$, therefore $y = g^{kn-i}$ and can return $x := (kn - i) \bmod q = \log_g y$.

Compare with time–memory tradeoff on slide ??.

Discrete logarithm algorithms for $\mathbb{Z}_p^*$

The Index Calculus Algorithm computes discrete logarithms in the cyclic group $\mathbb{Z}_p^*$. Unlike the generic algorithms, it has sub-exponential runtime

$$2^{O(\sqrt{\log p \log \log p})}$$

Therefore, prime p bit-length in cyclic group $\mathbb{Z}_p^*$ has to be *much* longer than a symmetric key of equivalent attack cost. In contrast, the bit-length of the order q of the subgroup used merely has to be doubled.

Elliptic-curve groups over $\mathbb{Z}_p^*$ or $\mathrm{GF}(p^n)$ exist that are not believed to be vulnerable to the Index Calculus Algorithm.

Equivalent key lengths: (NIST)

private key length	RSA factoring $n = pq$ modulus n	Discrete logarithm problem in $\mathbb{Z}_p^*$ modulus p	Discrete logarithm problem in $\mathbb{Z}_p^*$ order q	in EC order q
80 bits	1024 bits	1024 bits	160 bits	160 bits
112 bits	2048 bits	2048 bits	224 bits	224 bits
128 bits	3072 bits	3072 bits	256 bits	256 bits
192 bits	7680 bits	7680 bits	384 bits	384 bits
256 bits	15360 bits	15360 bits	512 bits	512 bits

Schnorr groups – working in subgroups of $\mathbb{Z}_p^*$

Schnorr group: cyclic subgroup $\mathbb{G} = \langle g \rangle \subset \mathbb{Z}_p^*$ with prime order $q = |\mathbb{G}| = (p-1)/r$, where (p, q, g) are generated with:

❶ Choose primes $p \gg q$ with $p = qr + 1$ for $r \in \mathbb{N}$

❷ Choose $1 < h < p$ with $h^r \bmod p \neq 1$

❸ Use $g := h^r \bmod p$ as generator for $\mathbb{G} = \langle g \rangle = \{h^r \bmod p \,|\, h \in \mathbb{Z}_p^*\}$

Advantages:

▶ Select bit-length of p and q independently, based on respective security requirements (e.g. 128-bit security: 3072-bit p, 256-bit q)

Difficulty of Discrete Logarithm problem over $\mathbb{G} \subseteq \mathbb{Z}_p^*$ with order $q = |\mathbb{G}|$ depends on both p (subexponentially) and q (exponentially).

▶ Some operations faster than if $\log_2 q \approx \log_2 p$.

Square-and-multiply exponentiation $g^x \bmod p$ (with $x < q$) run-time $\sim \log_2 x < \log_2 q$.

▶ Prime order q has several advantages:

• simple choice of generator (pick any element $\neq 1$)

• $\mathbb{G}$ has no (non-trivial) subgroups $\Rightarrow$ no small subgroup confinement attacks

• q with small prime factors can make Decision Diffie–Hellman problem easy to solve (Exercise 13)

Compare with slide 75 where $r = 2$.

Schnorr groups (proofs)

Let $p = rq + 1$ with p, q prime and $\mathbb{G} = \{h^r \bmod p \,|\, h \in \mathbb{Z}_p^*\}$. Then

❶ $\mathbb{G}$ is a subgroup of $\mathbb{Z}_p^*$.

Proof: $\mathbb{G}$ is closed under multiplication, as for all $x, y \in \mathbb{G}$ we have
$x^r y^r \bmod p = (xy)^r \bmod p = (xy \bmod p)^r \bmod p \in \mathbb{G}$ as $(xy \bmod p) \in \mathbb{Z}_p^*$.
In addition, $\mathbb{G}$ includes the neutral element $1^r = 1$
For each h^r, it also includes the inverse element $(h^{-1})^r \bmod p$.

❷ $\mathbb{G}$ has $q = (p-1)/r$ elements.

Proof: The idea is to show that the function $f_r : \mathbb{Z}_p^* \to \mathbb{G}$ with $f_r(x) = x^r \bmod p$ is an r-to-1 function, and then since $\mathbb{Z}_p^* = p - 1$ this will show that $|\mathbb{G}| = q = (p-1)/r$.
Let g be a generator of $\mathbb{Z}_p^*$ such that $\{g^0, g^1, \ldots, g^{p-2}\} = \mathbb{Z}_p^*$. Under what condition for i, j is $(g^i)^r \equiv (g^j)^r \pmod{p}$? $(g^i)^r \equiv (g^j)^r \pmod{p} \Leftrightarrow ir \equiv jr \pmod{p-1} \Leftrightarrow (p-1)|(ir - jr) \Leftrightarrow rq|(ir - jr) \Leftrightarrow q|(i - j)$.
For any fixed $j \in \{0, \ldots, p-2\} = \mathbb{Z}_{p-1}$, what values of $i \in \mathbb{Z}_{p-1}$ fulfill the condition $q|(i - j)$, and how many such values i are there? For each j, there are exactly the r different values $i \in \{j, j+q, j+2q, \ldots, j+(r-1)q\}$ in $\mathbb{Z}_{p-1}$, as $j + rq \equiv j \pmod{p-1}$. This makes f_r an r-to-1 function.

❸ For any $h \in \mathbb{Z}_p^*$, h^r is either 1 or a generator of $\mathbb{G}$.

Proof: $h^r \in \mathbb{G}$ (by definition) and $|\mathbb{G}|$ prime $\Rightarrow \mathrm{ord}_{\mathbb{G}}(h^r) \in \{1, |\mathbb{G}|\}$ (Lagrange).

❹ $h \in \mathbb{G} \Leftrightarrow h \in \mathbb{Z}_p^* \wedge h^q \bmod p = 1$. (Useful security check!)

Proof: Let $h = g^i$ with $\langle g \rangle = \mathbb{Z}_p^*$ and $0 \le i < p - 1$. Then
$h^q \bmod p = 1 \Leftrightarrow g^{iq} \bmod p = 1 \Leftrightarrow iq \bmod (p-1) = 0 \Leftrightarrow rq|iq \Leftrightarrow r|i$.

Katz/Lindell section 8.3.3

Elliptic-curve groups

elliptic curve over $\mathbb{R}$ ($A = -1$, $B = 1$) elliptic curve over $\mathbb{Z}_{11}$ ($A = -1$, $B = 1$)
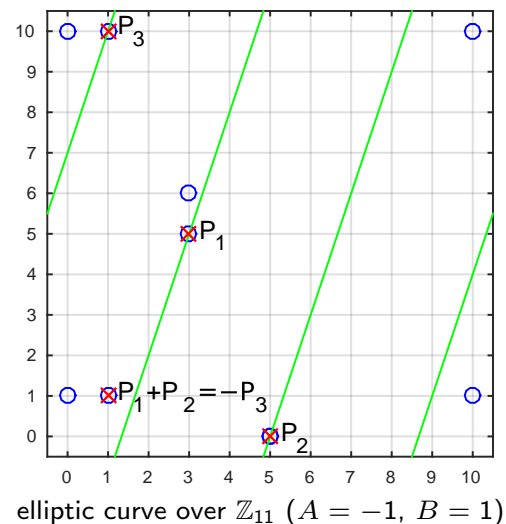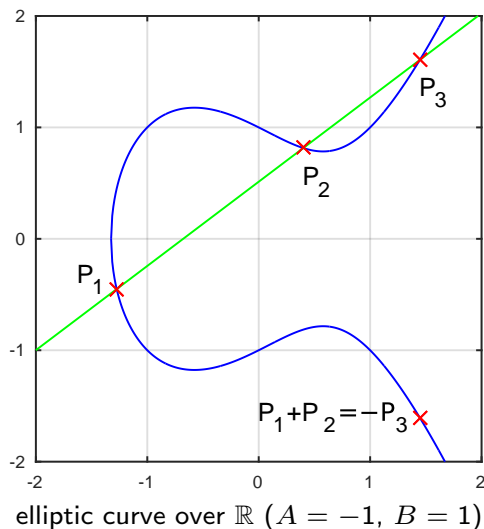
Elliptic curves are sets of 2-D coordinates (x, y) with

$$y^2 = x^3 + Ax + B$$

plus one additional "point at infinity" $\mathcal{O}$.

Group operation $P_1 + P_2$: draw line through curve points P_1, P_2, intersect with curve to get third point P_3, then negate the y coordinate of P_3 to get $P_1 + P_2$.

Neutral element: $\mathcal{O}$ − intersects any vertical line. Inverse: $-(x, y) = (x, -y)$

Curve compression: for any given x, encoding y requires only one bit

Elliptic-curve group operator

Elliptic curve: ("short Weierstrass equation")

$$\mathbb{E}(\mathbb{Z}_p, A, B) := \{(x, y) \mid x, y \in \mathbb{Z}_p \text{ and } y^2 \equiv x^3 + Ax + B \ (\mathrm{mod}\ p)\} \cup \{\mathcal{O}\}$$

where $p > 5$ prime, parameters A, B with $4A^3 + 27B^2 \not\equiv 0 \ (\mathrm{mod}\ p)$.

Neutral element: $P + \mathcal{O} = \mathcal{O} + P = P$

For $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$, $P_1, P_2 \neq \mathcal{O}$, $x_1 \neq x_2$:

$$m = \frac{y_2 - y_1}{x_2 - x_1} \qquad\qquad \text{line slope}$$

$$y = m \cdot (x - x_1) + y_1 \qquad\qquad \text{line equation}$$

$$\left(m \cdot (x - x_1) + y_1\right)^2 = x^3 + Ax + B \qquad\qquad \text{intersections}$$

$$x_3 = m^2 - x_1 - x_2 \qquad\qquad \text{third-point solution}$$

$$y_3 = m \cdot (x_3 - x_1) + y_1$$

$$(x_1, y_1) + (x_2, y_2) = (m^2 - x_1 - x_2, m \cdot (x_1 - x_3) - y_1) \quad \text{(all of this mod } p)$$

If $x_1 = x_2$ but $y_1 \neq y_2$ then $P_1 = -P_2$ and $P_1 + P_2 = \mathcal{O}$.
If $P_1 = P_2$ and $y_1 = 0$ then $P_1 + P_2 = 2P_1 = \mathcal{O}$.
If $P_1 = P_2$ and $y_1 \neq 0$ then use tangent $m = (3x_1^2 + A)/2y_1$.

$(x, y) =$ affine coordinates; projective coordinates (X, Y, Z) with $X/Z = x$, $Y/Z = y$ add faster

Elliptic-curve groups with prime order

How large are elliptic curves over $\mathbb{Z}_p$?

Equation $y^2 = f(x)$ has two solutions if $f(x)$ is a quadratic residue, and one solution if $f(x) = 0$. Half of the elements in $\mathbb{Z}_p^*$ are quadratic residues, so expect around $2 \cdot (p-1)/2 + 1 = p$ points on the curve.

Hasse bound: $p + 1 - 2\sqrt{p} \leq |E(\mathbb{Z}_p, A, B)| \leq p + 1 + 2\sqrt{p}$

Actual group order approximately uniformly spread over Hasse bound.

Elliptic curves became usable for cryptography with the invention of efficient algorithms for counting the exact number of points on them.

Generate a cyclic elliptic-curve group (p, q, A, B, G) with:

❶ Choose uniform n-bit prime p

❷ Choose $A, B \in \mathbb{Z}_p$ with $4A^3 + 27B^2 \neq 0 \bmod p$, determine $q = |E(\mathbb{Z}_p, A, B)|$, repeat until q is an n-bit prime

❸ Choose $G \in E(\mathbb{Z}_p, A, B) \setminus \{\mathcal{O}\}$ as generator

Easy to find a point $G = (x, y)$ on the curve: pick uniform $x \in \mathbb{Z}_p$ until $f(x)$ is a quadratic residue or 0, then set $y = \sqrt{f(x)}$.

Elliptic-curve discrete-logarithm problem

The elliptic-curve operation is traditionally written as an additive group, so the "exponentiation" of the elliptic-curve discrete-logarithm problem (ECDLP) becomes multiplication:

$$x \cdot G = \underbrace{G + G + \cdots + G}_{x \text{ times}} \qquad x \in \mathbb{Z}_q$$

So the square-and-multiply algorithm becomes double-and-add.

Many curve parameters and cyclic subgroups for which ECDLP is believed to be hard have been proposed or standardised.
NIST FIPS 186-2, RFC 5639, SEC 2, Curve25519, etc.

Example: NIST P-256

$p = \mathtt{0xffffffff00000001000000000000000000000000ffffffffffffffffffffffff}$

$q = \mathtt{0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551}$

$A = 3$

$B = \mathtt{0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b}$

$G = (\mathtt{0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296},$

$\qquad \mathtt{0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5})$

Note: the NIST parameters have come under suspicion for potentially having been carefully selected by the NSA, to embed a vulnerability. `http://safecurves.cr.yp.to/rigid.html`

ElGamal encryption scheme

The DH key exchange requires two messages. This can be eliminated if everyone publishes their g^x as a *public key* in a sort of phonebook. Assume $((\mathbb{G}, \cdot), q, g)$ are fixed for all participants.

A publishes g^x as her *public key* and keeps x as her *secret key*.
B generates for each message a new nonce y and then sends

$$B \to A: \qquad g^y, \ (g^x)^y \cdot M$$

where $M \in \mathbb{G}$ is the message that B sends to A in this asymmetric encryption scheme. Then A calculates

$$[(g^x)^y \cdot M] \cdot [(g^y)^{q-x}] = M$$

to decrypt M.

In practice, this scheme is rarely used because of the difficulty of fitting M into $\mathbb{G}$. Instead, B only sends g^y. Then both parties calculate $h(A\|B\|g^y\|g^{xy})$ and use that as the private session key for an efficient block-cipher based authenticated encryption scheme that protects the confidentiality and integrity of the bulk of the message. B digitally signs g^y to establish his identity.

Number theory: easy and difficult problems

Easy:

- given integer n, i and $x \in \mathbb{Z}_n^*$: calculate $x^{-1} \in \mathbb{Z}_n^*$ or $x^i \in \mathbb{Z}_n^*$
- given prime p and polynomial $f(x) \in \mathbb{Z}_p[x]$:
  find $x \in \mathbb{Z}_p$ with $f(x) = 0$
  runtime grows linearly with the degree of the polynomial

Difficult:

- given safe prime p, generator $g \in \mathbb{Z}_p^*$ (or large subgroup):
  - given value $a \in \mathbb{Z}_p^*$: find x such that $a = g^x$.
    $\rightarrow$ Discrete Logarithm Problem
  - given values $g^x, g^y \in \mathbb{Z}_p^*$: find g^{xy}.
    $\rightarrow$ Computational Diffie–Hellman Problem
  - given values $g^x, g^y, z \in \mathbb{Z}_p^*$: tell whether $z = g^{xy}$.
    $\rightarrow$ Decision Diffie–Hellman Problem
- given a random $n = p \cdot q$, where p and q are ℓ-bit primes ($\ell \geq 1024$):
  - find integers p and q such that $n = p \cdot q$ in $\mathbb{N}$
    $\rightarrow$ Factoring Problem
  - given a polynomial $f(x)$ of degree > 1:
    find $x \in \mathbb{Z}_n$ such that $f(x) = 0$ in $\mathbb{Z}_n$

"Textbook" RSA encryption

Key generation

- Choose random prime numbers p and q (each ≈ 1024 bits long)
- $n := pq$ (≈ 2048 bits = key length) $\varphi(n) = (p-1)(q-1)$
- pick integer values e, d such that: $ed \bmod \varphi(n) = 1$
- public key $PK := (n, e)$
- secret key $SK := (n, d)$

Encryption

- input plaintext $M \in \mathbb{Z}_n^*$, public key (n, e)
- $C := M^e \bmod n$

Decryption

- input ciphertext $C \in \mathbb{Z}_n^*$, secret key (n, d)
- $M := C^d \bmod n$

In $\mathbb{Z}_n$: $(M^e)^d = M^{ed} = M^{ed \bmod \varphi(n)} = M^1 = M$.

Common implementation tricks to speed up computation:

- Choose small e with low Hamming weight (e.g., 3, 17, $2^{16} + 1$) for faster modular encryption
- Preserve factors of n in $SK = (p, q, d)$, decryption in both $\mathbb{Z}_p$ and $\mathbb{Z}_q$, use Chinese remainder theorem to recover result in $\mathbb{Z}_n$.

"Textbook" RSA is not secure

There are significant security problems with a naive application of the basic "textbook" RSA encryption function $C := P^e \bmod n$:

- ▶ deterministic encryption: cannot be CPA secure

- ▶ malleability:
  - adversary intercepts C and replaces it with $C' := X^e \cdot C$
  - recipient decrypts $M' = \mathsf{Dec}_{SK}(C') = X \cdot M \bmod n$

- ▶ chosen-ciphertext attack recovers plaintext:
  - adversary intercepts C and replaces it with $C' := R^e \cdot C \bmod n$
  - decryption oracle provides $M' = \mathsf{Dec}_{SK}(C') = R \cdot M \bmod n$
  - adversary recovers $M = M' \cdot R^{-1} \bmod n$

- ▶ Small value of M (e.g., 128-bit AES key), small exponent $e = 3$:
  - if $M^e < n$ then $C = M^e \bmod n = M^e$ and then $M = \sqrt[3]{C}$ can be calculated efficiently in $\mathbb{Z}$ (no modular arithmetic!)

- ▶ many other attacks exist ...

Trapdoor permutations

A **trapdoor permutation** is a tuple of polynomial-time algorithms $(\mathsf{Gen}, F, F^{-1})$ such that

- ▶ the **key generation algorithm** Gen receives a security parameter ℓ and outputs a pair of keys $(PK, SK) \leftarrow \mathsf{Gen}(1^\ell)$, with key lengths $|PK| \geq \ell$, $|SK| \geq \ell$;

- ▶ the **sampling function** F maps a public key PK and a value $x \in \mathcal{X}$ to a value $y := F_{PK}(x) \in \mathcal{X}$;

- ▶ the **inverting function** F^{-1} maps a secret key SK and a value $y \in \mathcal{X}$ to a value $x := F_{SK}^{-1}(y) \in \mathcal{X}$;

- ▶ for all ℓ, $(PK, SK) \leftarrow \mathsf{Gen}(1^\ell)$, $x \in \mathcal{X}$: $F_{SK}^{-1}(F_{PK}(x)) = x$.

In practice, the domain $\mathcal{X}$ may depend on PK.

This looks almost like the definition of a public-key encryption scheme, the difference being
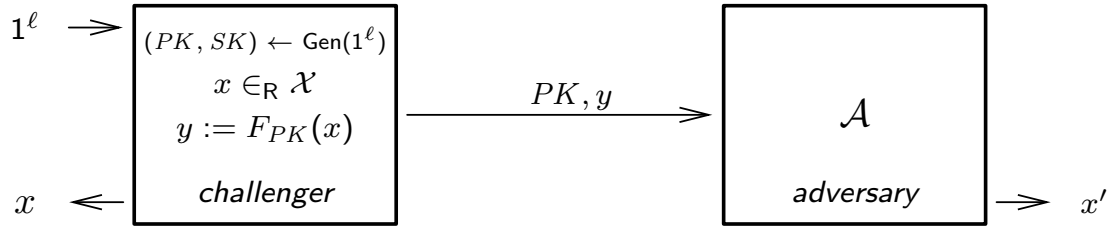
- ▶ F is deterministic;

- ▶ the associated security definition.

Secure trapdoor permutations

Trapdoor permutation: $\Pi = (\text{Gen}, F, F^{-1})$

Experiment/game $\text{TDInv}_{\mathcal{A}, \Pi}(\ell)$:

❶ The challenger generates a key pair $(PK, SK) \leftarrow \text{Gen}(1^\ell)$ and a random value $x \in_{\mathsf{R}} \mathcal{X}$ from the domain of F_{PK}.

❷ The adversary $\mathcal{A}$ is given inputs PK and $y := F_{PK}(x)$.

❸ Finally, $\mathcal{A}$ outputs x'.

If $x' = x$ then $\mathcal{A}$ has succeeded: $\text{TDInv}_{\mathcal{A}, \Pi}(\ell) = 1$.

A trapdoor permutation Π is secure if for all probabilistic polynomial time adversaries $\mathcal{A}$ the probability of success $\mathbb{P}(\text{TDInv}_{\mathcal{A}, \Pi}(\ell) = 1)$ is negligible.

While the definition of a trapdoor permutation resembles that of a public-key encryption scheme, its security definition does not provide the adversary any control over the input (plaintext).

Public-key encryption scheme from trapdoor permutation

Trapdoor permutation: $\Pi_{\mathsf{TD}} = (\text{Gen}_{\mathsf{TD}}, F, F^{-1})$ with $F_{PK} : \mathcal{X} \leftrightarrow \mathcal{X}$
Authentic. encrypt. scheme: $\Pi_{\mathsf{AE}} = (\text{Gen}_{\mathsf{AE}}, \text{Enc}, \text{Dec})$, key space $\mathcal{K}$
Secure hash function $h : \mathcal{X} \to \mathcal{K}$

We define the private-key encryption scheme $\Pi = (\text{Gen}', \text{Enc}', \text{Dec}')$:

▶ Gen': output key pair $(PK, SK) \leftarrow \text{Gen}_{\mathsf{TD}}(1^\ell)$

▶ Enc': on input of plaintext message M, generate random $x \in_{\mathsf{R}} \mathcal{X}$, $y = F(x)$, $K = h(x)$, $C \leftarrow \text{Enc}_K(M)$, output ciphertext (y, C);

▶ Dec': on input of ciphertext message $C = (y, C)$, recover $K = h(F^{-1}(y))$, output $\text{Dec}_K(C)$

$$\boxed{\text{Encrypted message:} \quad F(x), \text{Enc}_{h(x)}(M)}$$

The trapdoor permutation is only used to communicate a "session key" $h(x)$, the actual message is protected by a symmetric authenticated encryption scheme. The adversary $\mathcal{A}$ in the $\text{PubK}_{\mathcal{A}, \Pi'}^{\mathsf{cca}}$ game has no influence over the input of F.

If hash function h is replaced with a "random oracle" (something that just picks a random output value for each input from $\mathcal{X}$), the resulting public-key encryption scheme Π' is CCA secure.

Using RSA as a CCA-secure encryption scheme

**Solution 1:** use only as trapdoor function to build encryption scheme

- Pick random value $x \in \mathbb{Z}_n^*$

- Ciphertext is $(x^e \bmod n, \mathsf{Enc}_{h(x)}(M))$, where Enc is from an authenticated encryption scheme

Solution 2: Optimal Asymmetric Encryption Padding

Make M (with zero padding) the left half, and a random string R the right half, of the input of a two-round Feistel cipher, using a secure hash function as the round function.

Interpret the result (X, Y) as an integer M'.

Then calculate $C := M'^e \bmod n$.

PKCS #1 v2.0

Wikipedia/Ozga
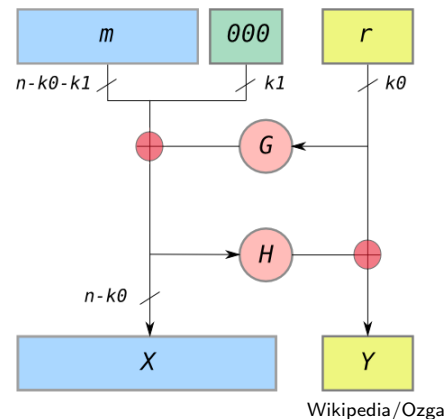
Practical pitfalls with implementing RSA

- low entropy of random-number generator seed when generating p and q (e.g. in embedded devices):
  - take public RSA modulus n_1 and n_2 from two devices
  - test $\gcd(n_1, n_2) \overset{?}{=} 1 \Rightarrow$ if no, n_1 and n_2 share this number as a common factor
  - February 2012 experiments: worked for many public HTTPS keys

    Lenstra et al.: Public keys, CRYPTO 2012
    Heninger et al.: Mining your Ps and Qs, USENIX Security 2012.

One-time signatures

A simple digital signature scheme can be built using a one-way function h (e.g., secure hash function):

Secret key: $2n$ random bit strings $R_{i,j}$ $(i \in \{0,1\}, 1 \leq j \leq n)$

Public key: $2n$ bit strings $h(R_{i,j})$

Signature: $(R_{b_1,1}, R_{b_2,2}, \ldots, R_{b_n,n})$, where $h(M) = b_1 b_2 \ldots b_n$

Digital Signature Algorithm (DSA)

Let $(\mathbb{G}, q, g)$ be system-wide choices of a cyclic group $\mathbb{G}$ of order q with generator g. In addition, we need to functions $H : \{0,1\}^* \to \mathbb{Z}_q$ and $F : \mathbb{G} \to \mathbb{Z}_q$ where H must be collision resistant.

Both H and F are random oracles in security proofs, but common F not even preimage resistant.

Key generation: uniform secret key $x \in \mathbb{Z}_q$, then public key $y := g^x \in \mathbb{G}$.

Signing: On input of a secret key $x \in \mathbb{Z}_q$ and a message $m \in \{0,1\}^*$, first choose (for each message!) uniformly at random $k \in \mathbb{Z}_q^*$ and set $r := F(g^k)$. Then solve the linear equation

$$k \cdot s - x \cdot r \equiv H(m) \pmod{q} \tag{1}$$

for $s := k^{-1} \cdot (H(m) + xr) \bmod q$. If $r = 0$ or $s = 0$, restart with a fresh k, otherwise output $\mathsf{Sign}_x(m) \leftarrow (r, s)$.

Verification: On input of public key y, message m, and signature (r, s), verify equation (1) after both sides have been turned into exponents of g:

$$g^{ks}/g^{xr} = g^{H(m)} \tag{2}$$

$$(g^k)^s = g^{H(m)} y^r \tag{3}$$

$$g^k = g^{H(m)s^{-1}} y^{rs^{-1}} \tag{4}$$

$$\Longrightarrow \text{ actually verify:} \quad r \stackrel{?}{=} F\big(g^{H(m)s^{-1}} y^{rs^{-1}}\big) \tag{5}$$

DSA variants

ElGamal signature scheme

The DSA idea was originally proposed by ElGamal with $\mathbb{G} = \mathbb{Z}_p^*$, $\mathrm{ord}(g) = q = p - 1$ and $F(x) = x$.

Unless the p and g are chosen more carefully, ElGamal signatures can be vulnerable to forgery: D. Bleichenbacher: Generating ElGamal signatures without knowing the secret key. EUROCRYPT '96. http://www.springerlink.com/link.asp?id=xbwmv0b564gwlq7a

NIST DSA

In 1993, the US government standardized the Digital Signature Algorithm, a modification of the ElGamal signature scheme where

- ▶ $\mathbb{G}$ is a prime-order subgroup of $\mathbb{Z}_p^*$
- ▶ prime number p (1024 bits), prime number q (160 bits) divides $p - 1$
- ▶ $g = h^{(p-1)/q} \bmod p$, with $1 < h < p - 1$ so that $g > 1$ (e.g., $h = 2$)
- ▶ H is SHA-1
- ▶ $F(x) = x \bmod q$

Generate key: random $0 < x < q$, $y := g^x \bmod p$.
Signature $(r, s) := \mathsf{Sign}_x(m)$: random $0 < k < q$,
$r := (g^k \bmod p) \bmod q$, $s := (k^{-1}(H(m) + x \cdot r)) \bmod q$

Later versions of the DSA standard FIPS 186 added larger values for (p, q, g), as well as ECDSA, where $\mathbb{G}$ is one of several elliptic-curve groups over $\mathbb{Z}_p$ or $\mathsf{GF}(2^n)$ and $F((x, y)) = x \bmod q$.

Proper generation of k is important

DSA fails catastrophically if the adversary can ever guess k:

$$s \equiv k^{-1} \cdot (H(m) + xr) \quad \Rightarrow \quad x \equiv (k \cdot s - H(m)) \cdot r^{-1} \pmod{q}$$

All that is needed for k to leak is two messages $m \neq m'$ signed with the same $k = k'$ (easily recognized from $r = r' = F(g^k)$):

$$s \equiv k^{-1} \cdot (H(m) + xr)$$
$$s' \equiv k^{-1} \cdot (H(m') + xr)$$
$$s - s' \equiv k^{-1} \cdot (H(m) - H(m'))$$
$$k \equiv (H(m) - H(m'))(s - s')^{-1} \pmod{q}$$

Sony used a fixed k in firmware signatures for their PlayStation 3 (fail0verflow, 27th Chaos Communication Conf., Berlin 2010).

Without a good random-bit generator to generate k, use e.g. $k := \mathsf{SHA-3}(x\|m) \bmod q$ (with hash output longer than q).

Public-key infrastructure I

Public key encryption and signature algorithms allow the establishment of confidential and authenticated communication links with the owners of public/private key pairs.

Public keys still need to be reliably associated with identities of owners. In the absence of a personal exchange of public keys, this can be mediated via a trusted third party. Such a *certification authority* C issues a digitally signed *public key certificate*

$$\mathsf{Cert}_C(A) = (A, PK_A, T, L, \mathsf{Sign}_{PK_C}(A, PK_A, T, L))$$

in which C confirms that the public key PK_A belongs to entity A, starting at time T and that this confirmation is valid for the time interval L, and all this is digitally signed with C's private signing key SK_C.

Anyone who knows C's public key K_C from a trustworthy source can use it to verify the certificate $\mathsf{Cert}_C(A)$ and obtain a trustworthy copy of A's public key PK_A this way.

Public-key infrastructure II

We can use the operator $\bullet$ to describe the extraction of A's public key PK_A from a certificate $\mathsf{Cert}_C(A)$ with the certification authority public key PK_C:

$$PK_C \bullet \mathsf{Cert}_C(A) = \begin{cases} PK_A & \text{if certificate valid} \\ \text{failure} & \text{otherwise} \end{cases}$$

The $\bullet$ operation involves not only the verification of the certificate signature, but also the validity time and other restrictions specified in the signature. For instance, a certificate issued by C might contain a reference to an online *certificate revocation list* published by C, which lists all public keys that might have become compromised (e.g., the smartcard containing SK_A was stolen or the server storing SK_A was broken into) and whose certificates have not yet expired.

Public-key infrastructure III

Public keys can also be verified via several trusted intermediaries in a *certificate chain*:

$$PK_{C_1} \bullet \mathsf{Cert}_{C_1}(C_2) \bullet \mathsf{Cert}_{C_2}(C_3) \bullet \cdots \bullet \mathsf{Cert}_{C_{n-1}}(C_n) \bullet \mathsf{Cert}_{C_n}(B) = PK_B$$

A has received directly a trustworthy copy of PK_{C_1} (which many implementations store locally as a certificate $\mathsf{Cert}_A(C_1)$ to minimise the number of keys that must be kept in tamper-resistant storage).

Certification authorities could be made part of a hierarchical tree, in which members of layer n verify the identity of members in layer $n-1$ and $n+1$. For example layer 1 can be a national CA, layer 2 the computing services of universities and layer 3 the system administrators of individual departments.

Practical example: A personally receives K_{C_1} from her local system administrator C_1, who confirmed the identity of the university's computing service C_2 in $\mathsf{Cert}_{C_1}(C_2)$, who confirmed the national network operator C_3, who confirmed the IT department of B's employer C_3 who finally confirms the identity of B. An online directory service allows A to retrieve all these certificates (plus related certificate revocation lists) efficiently.

In today's Transport Layer Security (TLS) practice (HTTPS, etc.), most private users use their web-browser or operating-system vendor as their sole trusted source of PK_{C_1} root keys.

Outlook

Goals of this part of the course were

- ▶ introduce secure hash functions and some of their applications
- ▶ introduce some of the number-theory and abstract-algebra concepts behind the main public-key encryption and signature schemes, in particular the discrete logarithm problem, the Diffie-Hellman key exchange, RSA encryption, and the Digital Signature Algorithm

Modern cryptography is still a young discipline (born in the early 1980s), but well on its way from a collection of tricks to a discipline with solid theoretical foundations.

Some important concepts that we did not touch here for time reasons:

- ▶ password-authenticated key exchange
- ▶ identity-based encryption
- ▶ side-channel and fault attacks
- ▶ application protocols: electronic voting, digital cash, etc.
- ▶ secure multi-party computation
- ▶ post-quantum cryptography