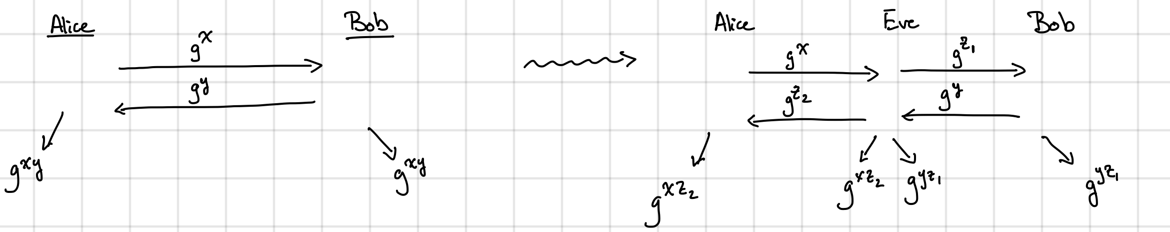
Diffie-Hellman key-exchange is an anonymous key-exchange protocol: neither side knows who they are talking to
↳ vulnerable to a "man-in-the-middle" attack



Observe Eve can now decrypt all of the messages between Alice and Bob and Alice + Bob have no idea!

What we require: authenticated key-exchange (not anonymous) and relies on a root of trust (e.g., a certificate authority)
↳ On the web, one of the parties will authenticate themself by presenting a certificate

To build authenticated key-exchange, we require more ingredients — namely, an integrity mechanism [e.g., a way to bind a message to a sender — a "public-key MAC" or digital signature]

We will revisit when discussing the TLS protocol

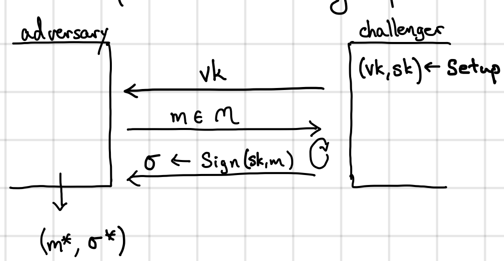
Digital signature scheme: Consists of three algorithms:
- Setup → (vk, sk): Outputs a verification key vk and a signing key sk
- Sign(sk, m) → σ: Takes the signing key sk and a message m and outputs a signature σ
- Verify(vk, m, σ) → 0/1: Takes the verification key vk, a message m, and a signature σ, and outputs a bit 0/1

Two requirements:
- Correctness: For all messages $m \in \mathcal{M}$, $(vk, sk) \leftarrow \text{Setup}$, then

$$\Pr[\text{Verify}(vk, m, \text{Sign}(sk, m)) = 1] = 1.$$

[Honestly-generated signatures always verify]

- Unforgeability: Very similar to MAC security. For all efficient adversaries $\mathcal{A}$, $\text{SigAdv}[\mathcal{A}] = \Pr[W = 1] = \text{negl}(\lambda)$, where W is the output of the following experiment:



Let $m_1, \dots, m_Q$ be the signing queries the adversary submits to the challenger Then, $W = 1$ if and only if:

$$\text{Verify}(vk, m^*, \sigma^*) = 1 \quad \text{and} \quad m^* \notin \{m_1, \dots, m_Q\}$$

Adversary cannot produce a valid signature on a new message.

Exact analog of a MAC (slightly weaker unforgeability: require adversary to not be able to forge signature on new message)
↳ MAC security required that no forgery is possible on any message [needed for authenticated encryption]

digital signature algorithm

elliptic-curve DSA

} standards (widely used on the web – eg, TLS)

It is possible to build digital signatures from discrete log based assumptions (DSA, ECDSA)
↳ But construction not intuitive until we see zero knowledge proofs
↳ We will first construct from RSA (trapdoor permutations)

We will now introduce some facts on composite-order groups:

Let $N = pq$ be a product of two primes p, q. Then, $\mathbb{Z}_N = \{0, 1, \ldots, N-1\}$ is the additive group of integers modulo N. Let $\mathbb{Z}_N^*$ be the set of integers that are invertible (under multiplication) modulo N.

$$x \in \mathbb{Z}_N^* \text{ if and only if } \gcd(x, N) = 1$$

Since $N = pq$ and p, q are prime, $\gcd(x, N) = 1$ unless x is a multiple of p or q:

$$|\mathbb{Z}_N^*| = N - p - q + 1 = pq - p - q + 1 = (p-1)(q-1) = \varphi(N)$$

Euler's phi function (Euler's totient function)

Recall Lagrange's Theorem:

for all $x \in \mathbb{Z}_N^*$: $x^{\varphi(N)} = 1 \pmod{N}$ [called Euler's theorem, but special case of Lagrange's theorem]

important: "ring of exponents" operate modulo $\varphi(N) = (p-1)(q-1)$

Hard problems in composite-order groups:

- Factoring: given $N = pq$ where p and q are sampled from a suitable distribution over primes, output p, q
- Computing cube roots: Sample random $x \xleftarrow{R} \mathbb{Z}_N^*$. Given $y = x^3 \pmod{N}$, compute $x \pmod{N}$.
  - ↳ This problem is easy in $\mathbb{Z}_p^*$ (when $3 \nmid p-1$). Namely, compute $3^{-1} \pmod{p-1}$, say using Euclid's algorithm, and then compute $y^{3^{-1}} \pmod{p} = (x^3)^{3^{-1}} \pmod{p} = x \pmod{p}$.
  - ↳ Why does this procedure not work in $\mathbb{Z}_N^*$. Above procedure relies on computing $3^{-1} \pmod{|\mathbb{Z}_N^*|} = 3^{-1} \pmod{\varphi(N)}$. But we do not know $\varphi(N)$ and computing $\varphi(N)$ is as hard as factoring N. In particular, if we know N and $\varphi(N)$, then we can write

$$\begin{cases} N = pq \\ \varphi(N) = (p-1)(q-1) \end{cases} \quad \text{[both relations hold over the integers]}$$

and solve this system of equations over the integers (and recover p, q)

Hardness of computing cube roots is the basis of the RSA assumption:

RSA assumption: Take $p, q \leftarrow \text{Primes}$ (distribution over prime numbers (size determined by security parameter λ)), and set $N = pq$. Then, for all efficient adversaries A,

$$\Pr[x \xleftarrow{R} \mathbb{Z}_N^*;\ y \leftarrow A(N, x) : y^3 = x] = \text{negl.}$$

more generally, can replace 3 with any e where $\gcd(e, \varphi(N)) = 1$

common choices:
$e = 3$
$e = 65537$

↳ Hardness of RSA relies on $\varphi(N)$ being hard to compute, and thus, on hardness of factoring (Reverse direction factoring $\overset{?}{\Rightarrow}$ RSA is not known)

Hardness of factoring / RSA assumption:

- Best attack based on general number field sieve (GNFS) — runs in time $\sim 2^{\tilde{O}(\sqrt[3]{\log N})}$ (same algorithm used to break discrete log over $\mathbb{Z}_p^*$)
- For 112-bits of security, use RSA-2048 (N is product of two 1024-bit primes)
  128-bits of security, use RSA-3072

  large key-sizes and computational cost $\Rightarrow$ ECC generally preferred over RSA
- Both prime factors should have similar bit length (ECM algorithm factors in time that scales with smaller factor)

RSA problem gives an instantiation of more general notion called a trapdoor permutation:

$$F_{RSA}: \mathbb{Z}_N^* \to \mathbb{Z}_N^*$$

$$F_{RSA}(x) := x^e \pmod N \quad \text{where } \gcd(N, e) = 1$$

Given $\varphi(N)$, we can compute $d = e^{-1} \pmod{\varphi(N)}$. Observe that given d, we can invert F_{RSA}:

$$F_{RSA}^{-1}(x) := x^d \pmod N.$$

Then, for all $x \in \mathbb{Z}_N^*$:

$$F_{RSA}^{-1}(F_{RSA}(x)) = (x^e)^d = x^{ed \pmod{\varphi(N)}} = x^1 = x \pmod N.$$

Trapdoor permutations: A trapdoor permutation (TDP) on a domain $\mathcal{X}$ consists of three algorithms:

- Setup $\to$ (pp, td): Outputs public parameters pp and a trapdoor td
- $F(pp, x) \to y$: On input the public parameters pp and input x, outputs $y \in \mathcal{X}$
- $F^{-1}(td, y) \to x$: On input the trapdoor td and input y, output $x \in \mathcal{X}$

Requirements:

- Correctness: for all pp output by Setup:
  - $F(pp, \cdot)$ implements a permutation on $\mathcal{X}$.
  - $F^{-1}(td, F(pp, x)) = x$ for all $x \in \mathcal{X}$.
- Security: $F(pp, \cdot)$ is a one-way function (to an adversary who does not see the trapdoor)

Naive approach (common "textbook" approach) to build signatures:

Let (F, F^{-1}) be a trapdoor permutation

- Verification key will be pp
- Signing key will be td

} to sign a message m, compute $\sigma \leftarrow F^{-1}(td, m)$
to verify a signature, check $m \stackrel{?}{=} F(pp, \sigma)$

Correct because:

$$F(pp, \sigma) = F(pp, F^{-1}(td, m)) = m$$

Secure because F^{-1} is hard to compute without trapdoor (signing key) FALSE!

↳ This is not true! Security of TDP just says that F is one-way. One-wayness just says function is hard to invert on a random input. But in the case of signatures, the message is the input. This is not only not random, but in fact, adversarially chosen!

↳ Very easy to attack. Consider the 0-query adversary:

Given verification key $vk = pp$, compute $F(pp, \sigma)$ for any $\sigma \in \mathcal{X}$

Output $m = F(pp, \sigma)$ and σ

↳ By construction, σ is a valid signature on the message m, and the adversary succeeds with advantage 1.

Textbook RSA signatures: [NEVER USE THIS!]

Setup: Sample (N, e, d) where $N = pq$ and $ed = 1 \pmod{\varphi(N)}$
Output $vk = (N, e)$ and $sk = d$

Sign$(\overset{=d}{sk}, m)$: Output $\sigma \leftarrow m^d \pmod N$

Verify$(\overset{=e}{vk}, m, \sigma)$: Output 1 if $\sigma^e = m \pmod N$

} Looks tempting (and simple)... but totally broken!

Signatures from trapdoor permutations (the full domain hash):

In order to appeal to security of TDP, we need that the argument to $F^{-1}(td, \cdot)$ to be <u>random</u>

<u>Idea</u>: hash the message first and sign the hash value (often called "hash-and-sign")

↳ <u>Another benefit</u>: Allows signing long messages (much larger than domain size of TDF)

<u>FDH construction</u>:

- Setup: Sample $(pp, td) \leftarrow$ Setup for the TDP and output $vk = pp$, $sk = td$
- Sign(sk, m): Output $\sigma \leftarrow F^{-1}(td, H(m))$
- Verify(vk, m, σ): Output 1 if $F(pp, \sigma) = H(m)$ and 0 otherwise

<u>Theorem</u>. If F is a trapdoor permutation and H is an ideal hash function (i.e., "random oracle") then the full domain hash signature scheme defined above is secure.

<u>Proof Idea</u>: Signature is <u>deterministic</u>, so to succeed, adversary has to forge on an unqueried message m.

Signature on m is preimage of F at $H(m)$

↳ Adversary has to invert F at <u>random</u> input (when H is modeled as a random oracle)

How to simulate signing queries?

↳ Relies on "programming" the random oracle

<u>Recap</u>: RSA-FDH signatures:

Setup: Sample modulus N, e, d such that $ed = 1 \pmod{\varphi(N)}$ — typically $e = 3$ or $e = 65537$

Some (partial) attacks can exploit very small public exponent $(e=3)$

Output $vk = (N, e)$ and $sk = (N, d)$

Sign(sk, m): $\sigma \leftarrow H(m)^d$ [Here, we are assuming that H maps into $\mathbb{Z}_N^*$]

Verify(vk, m, σ): output 1 if $H(m) = \sigma^e$ and 0 otherwise

<u>Standard</u>: PKCS1 v1.5 (typically used for signing certificates)

↳ Standard cryptographic hash functions hash into a 256-bit space (e.g., SHA-256), but FDH requires <u>full domain</u>

↳ PKCS 1 v1.5 is a way to <u>pad</u> hashed message before signing:

00 01	FF FF ··· FF FF	00	DI	H(m)
16 bits	pad		digest info (e.g., which hash function was used)	message hash (e.g., computed using SHA-256)

↳ Padding important to protect against chosen message attacks (e.g., preprocess to find messages m_1, m_2, m_3 where $H(m_1) = H(m_2) \cdot H(m_3)$)

(but this is <u>not</u> a full-domain hash and <u>cannot</u> prove security under RSA — can make stronger assumption...)