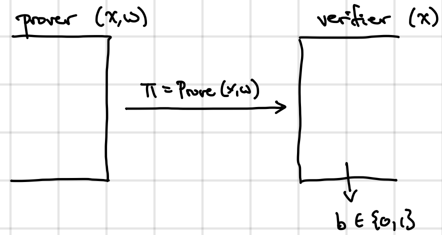
(NIZK)
Non-interactive zero-knowledge: Can we construct a zero-knowledge proof system where the proof is a single message from the prover to the verifier?



Why do we care? Interaction in practice is expensive!

Unfortunately, NIZKs are only possible for sufficiently-easy languages (i.e., languages in BPP). ← languages that can be decided by a randomized polynomial-time algorithm (w.h.p.)

↳ The simulator (for ZK property) can essentially be used to decide the language

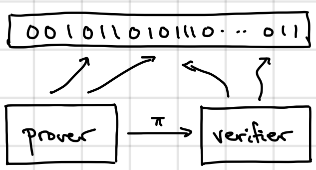
if $x \in \mathcal{L}$: $S(x) \to \pi$ and π should be accepted by the verifier (by ZK)

if $x \notin \mathcal{L}$: $S(x) \to \pi$ but π should not be accepted by verifier (by soundness)

} NIZK impossible for NP unless NP ⊆ BPP (unlikely!)

Impossibility results tell us where to look! If we cannot succeed in the "plain" model, then move to a different one:

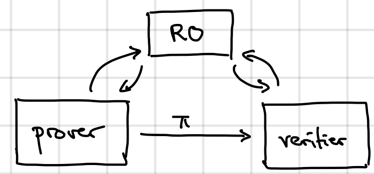
common random/reference string (CRS) model:



prover and verifier have access to shared randomness (could be a uniformly random string or a structured string)

in this model, simulator is allowed to choose (i.e., simulate) the CRS in conjunction with the proof, but soundness is defined with respect to an honestly-generated CRS [asymmetry between the capabilities of the real prover and the simulator]

random oracle model:

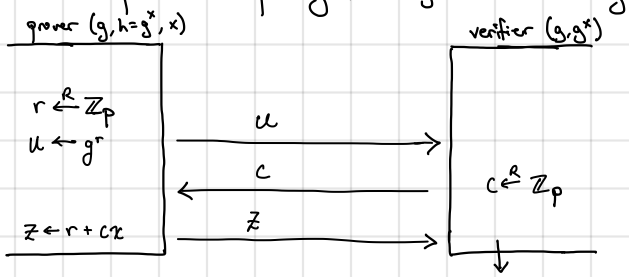


in this model, simulator can "program" the random oracle (again, asymmetry between real prover and the simulator)

⟹ In both cases, simulator has additional "power" than the real prover, which is critical for enabling NIZK constructions for NP.

Fiat-Shamir heuristic: NIZKs in random oracle model

Recall Schnorr's protocol for proving knowledge of discrete log:



verify that $g^z = u \cdot h^c$

In this protocol, verifier's message is uniformly random (and in fact, is "public coin" — the verifier has no secrets)

Key idea: Replace the verifier's challenge with a hash function $H: \{0,1\}^* \to \mathbb{Z}_p$

Namely, instead of sampling $c \xleftarrow{R} \mathbb{Z}_p$, we sample $c \leftarrow H(g, h, u)$. ← prover can now compute this quantity on its own!

Completeness, zero knowledge, proof of knowledge follow by a similar analysis as Schnorr [will rely on random oracle]

Signatures from discrete log in RO model (Schnorr):

- Setup: $x \xleftarrow{R} \mathbb{Z}_p$

  vk: $(g, h = g^x)$ sk: x

- Sign(sk, m): $r \xleftarrow{R} \mathbb{Z}_p$

  $u \leftarrow g^r$ $c \leftarrow H(g, h, u, m)$ $z \leftarrow r + cx$

  $\sigma = (u, z)$

  } signature is a NIZK proof of knowledge of discrete log of h (with challenge derived from the message m)

- Verify(vk, m, σ): write $\sigma = (u, z)$, vk = h, compute $c \leftarrow H(g, h, u, m)$ and accept if $g^z = u \cdot h^c$

Security essentially follows from security of Schnorr's identification protocol (together with Fiat-Shamir)

↳ forged signature on a new message m is a <u>proof of knowledge</u> of the discrete log (can be <u>extracted</u> from adversary)

Length of Schnorr's signature: vk: $(g, h = g^x)$ sk: x

$\sigma: (g^r, \underbrace{c = H(g, h, g^r, m)}_{\text{can be computed given other components, so do not need to include}}, z = r + cx)$

Verification checks that $g^z = g^r h^c$

$\Rightarrow |\sigma| = 2 \cdot |\mathbb{G}|$ [512 bits if $|\mathbb{G}| = 2^{256}$]

But, can do better... observe that challenge c only needs to be 128-bits (the knowledge error of Schnorr is $1/|C|$ where C is the set of possible challenges), so we can sample a 128-bit challenge rather than 256-bit challenge. Thus, instead of sending (g^r, z), instead send (c, z) and compute $g^r = g^z / h^c$ and that $c = H(g, h, g^r, m)$. Then resulting signatures are <u>384 bits</u>

128 bit challenge + 256 bit group element

<u>Important note</u>: Schnorr signatures are <u>randomized</u>, and security relies on having <u>good</u> randomness

↳ What happens if randomness is reused for two different signatures?

Then, we have

$$\left.\begin{aligned} \sigma_1 &= (g^r, c_1 = H(g, h, g^r, m_1), z_1 = r + c_1 x) \\ \sigma_2 &= (g^r, c_2 = H(g, h, g^r, m_2), z_2 = r + c_2 x) \end{aligned}\right\} \quad z_1 - z_2 = (c_1 - c_2)x \implies x = (c_1 - c_2)^{-1}(z_1 - z_2)$$

This is precisely the set of relations the knowledge extractor uses to recover the discrete log x (i.e., the signing key)!

<u>Deterministic Schnorr</u>: We want to replace the random value $r \xleftarrow{R} \mathbb{Z}_p$ with one that is deterministic, but which does not compromise security

↳ Derive randomness from message using a PRF. In particular, signing key includes a secret PRF key k, and signing algorithm computes $r \leftarrow F(k, m)$ and $\sigma \leftarrow \mathrm{Sign}(sk, m; r)$.

↳ Avoids randomness reuse/misuse vulnerabilities.

In practice, we (TLS protocol) use a variant of Schnorr's signature scheme called $\overbrace{\text{DSA / ECDSA}}^{\text{digital signature algorithm / elliptic-curve DSA}}$

↳ larger signatures (2 group elements - 512 bits) and proof only in "generic group" model

[but we use it because Schnorr was patented ... until 2008]

ECDSA signatures (over a group $\mathbb{G}$ of prime order p):

- Setup: $x \xleftarrow{R} \mathbb{Z}_p$

  $vk: (g, h = g^x)$ $\quad$ $sk: x$

- Sign(sk, m): $\alpha \xleftarrow{R} \mathbb{Z}_p$

  $u \leftarrow g^\alpha$ $\quad$ $r \leftarrow f(u) \in \mathbb{Z}_p$ $\quad$ deterministic function specified by ECDSA

  $s \leftarrow (H(m) + r \cdot x)/\alpha \in \mathbb{Z}_p$

  $\sigma = (r, s)$

[specifically, $f(u)$ parses $u = (\hat{x}, \hat{y}) \in \mathbb{F}_q^2$ where $\mathbb{F}_q$ is the base field over which the elliptic curve is defined, and outputs $\hat{x} \pmod p$, where $\hat{x}$ is viewed as a value in $[0, q)$]

- Verify(vk, m, σ): write $\sigma = (r, s)$, compute $u \leftarrow g^{H(m)/s} h^{r/s}$, accept if $r = f(u)$

  $vk = h$

<u>Correctness</u>: $u = g^{H(m)/s} h^{r/s} = g^{[H(m) + rx]/s} = g^{[H(m)+rx]/[H(m)+rx]\alpha^{-1}} = g^\alpha$ and $r = f(g^\alpha)$

Security analysis non-trivial: requires either strong assumptions or modeling $\mathbb{G}$ as an "ideal" group

Signature size: $\sigma = (r, s) \in \mathbb{Z}_p^2$ — for 128-bit security, $p \sim 2^{256}$ so $|\sigma| = 512$ bits (can use P-256 or Curve25519)