In a zero-knowledge proof system, a prover can convince the verifier that some statement \( x \) is true (without revealing anything more about \( x \)).

In many cases, we want a stronger property: the prover actually "knows" why a statement is true (e.g., it knows a "witness").

For instance, consider the following language:

\[
L = \{ h \in G \mid \exists x \in \mathbb{Z}_p : h = g^x \} = G
\]

This definition of \( L \) implicitly defines an NP relation \( R \):

\[
R(h, x) = 1 \iff h = g^x \in G
\]

In this case, all statements in \( G \) are true (i.e., contained in \( L \)), but we can still consider a notion of proving knowledge of the discrete log of an element \( h \in G \) — conceptually stronger property than proof of membership.

Philosophical question: What does it mean to "know" something?

If a prover is able to convince an honest verifier that it "knows" something, then it should be possible to extract that quantity from the prover.

Definition. An interactive proof system \((P, V)\) is a proof of knowledge for an NP relation \( R \) if there exists an efficient extractor \( \mathcal{E} \) such that for any \( x \) and any prover \( P^* \),

\[
\Pr[w \leftarrow \mathcal{E}(x) : R(x, w) = 1] \geq \Pr[\langle P^*, V \rangle(x) = 1] - \epsilon
\]

more generally,

\[
\epsilon \text{ could be polynomially smaller}
\]

knowledge error

Trivial proof of knowledge: prover sends witness in the clear to the verifier

\[ \Rightarrow \text{In most applications, we additionally require zero-knowledge} \]

Note: knowledge is a strictly stronger property than soundness

\[ \Rightarrow \text{if protocol has knowledge error } \epsilon \Rightarrow \text{it also has soundness error } \epsilon \text{ (i.e. a dishonest prover convinces an honest verifier of a false statement with probability at most } \epsilon) \]
Proving knowledge of discrete log ( Schnorr's protocol )

Assume a prover wants to prove it knows $x$ such that $h = g^x$ (i.e. prove knowledge of discrete log of $h$ base $g$)

prover
\[
\begin{aligned}
& r \xleftarrow{\$} \mathbb{Z}_p \\
& u \leftarrow g^r \\
& z \leftarrow r + cx \\
& c \xleftarrow{\$} \mathbb{Z}_p
\end{aligned}
\]

vector
\[
\begin{aligned}
& c \cdot g^z \\
& \text{verify that } g^z = u \cdot h^c
\end{aligned}
\]

Completeness: if $z = r + cx$, then $g^z = g^{r + cx} = g^r \cdot g^{cx} = u \cdot h^c$ zero knowledge only required to hold against an honest verifier (e.g. view of the honest verifier can be simulated)

Honest-Verifier Zero-Knowledge: build a simulator as follows (similar strategy: run the protocol "in reverse"): on input $(g, h)$:
1. Sample $z \xleftarrow{\$} \mathbb{Z}_p$
2. Sample $c \xleftarrow{\$} \mathbb{Z}_p$
3. Set $u = \frac{g^z}{c}$ and output $(u, c, z)$ - simulated transcript is identically distributed as the real transcript with an honest verifier
   - $u$ chosen so that $g^z = u \cdot h^c$
   - Simulated transcript is statistically indistinguishable from the real transcript with an honest verifier

What goes wrong if the challenge is not sampled uniformly at random (i.e., if the verifier is dishonest)
Above simulation no longer works (since we cannot sample $z$ first)

\( \Rightarrow \) To get general zero-knowledge, we require that the verifier first commit to its challenge (using a statistically hiding commitment)

Knowledge: Suppose $P^e$ is (possibly malicious) prover that convinces honest verifier with probability 1. We construct an extractor as follows:

1. Run the prover $P^e$ to obtain an initial message $u$
2. Send a challenge $c_1 \xleftarrow{\$} \mathbb{Z}_p$ to $P^e$. The prover replies with a response $z_1$.
3. "Rewind" the prover $P^e$ so its internal state is the same as it was at the end of Step 1. Then, send another challenge $c_2 \xleftarrow{\$} \mathbb{Z}_p$ to $P^e$. Let $z_2$ be the response of $P^e$.
4. Compute and output $\chi = (z_1 - z_2)(c_1 - c_2)^{-1} \in \mathbb{Z}_p$.

Since $P^e$ succeeds with probability 1 and the extractor perfectly simulates the honest verifier's behavior with probability 1, both $(u, c_1, z_1)$ and $(u, c_2, z_2)$ are both accepting transcripts. This means that

\[
\begin{aligned}
g^{z_1} &= u \cdot h^{c_1} \\
g^{z_2} &= u \cdot h^{c_2}
\end{aligned}
\]

\[
\Rightarrow \quad \frac{g^{z_1}}{h^{c_1}} = \frac{g^{z_2}}{h^{c_2}} \quad \Rightarrow \quad g^{z_1 + c_1 \chi} = g^{z_2 + c_2 \chi}
\]

with overwhelming probability

\[
\Rightarrow \quad \chi = (z_1 - z_2)(c_1 - c_2)^{-1} \in \mathbb{Z}_p
\]

Thus, extractor succeeds with overwhelming probability.
If $P^*$ succeeds with probability $E$, then need to rely on "Rewinding Lemma" to argue that extractor obtains two accepting transcripts with probability at least $2E^* - 1/4$.

How can a prover both prove knowledge and yet be zero-knowledge at the same time?

- Extractor operates by "rewinding" the prover (if the prover has good success probability, it can answer most challenges correctly).
- But in the real (actual) protocol, verifier cannot rewind (i.e., verifier only sees prover on fresh protocol executions), which cannot provide zero-knowledge.

Identification Protocol from Discrete Log

Suppose a client wants to authenticate to the server.

Goal: security against active adversaries (adversary sees contents of the server and can interact arbitrarily with the client) in this setting.

Can directly build such a scheme from Schnorr’s protocol:

\[
\begin{array}{c}
\text{Client (X)} \\
\text{secret } x \\
\downarrow \\
\text{server (g, h=g^x)} \\
\uparrow \\
\text{protocol is precisely 3-round} \\
\text{Schnorr proof of knowledge of discrete log}
\end{array}
\]

Correctness of this protocol follows from completeness of Schnorr's protocol.

(Active) security follows from knowledge property and zero-knowledge.

Intuitively: knowledge says that any client that successfully authenticates must have secret $X$.

Zero-knowledge says that interactions with honest client (i.e., the prover) do not reveal anything about $X$ (for active security, require protocol that provides general zero-knowledge rather than just HVZK).

More general view: $\Sigma$-protocols (Sigma protocols):

\[
\begin{array}{c}
\text{prover (X)} \\
g, h = g^x \\
\downarrow \\
\text{verifier} \\
g^r \\
\uparrow \\
\text{"commitment"} \\
c \\
\downarrow \\
\text{"challenge"} \\
\text{(random string, "public coin")]}
\end{array}
\]

Protocol also resembles a $\Sigma$.

Properties: 1. Completeness
2. Honest-Verifier Zero-Knowledge
3. Proof of Knowledge

Protocols with this structure (commitment-challenge-response) are called $\Sigma$-protocols (Sigma protocols).

Many variants of Schnorr protocols can be used to prove knowledge of statements like:

- Common discrete log: $X$ such that $h_1 = g^x$ and $h_2 = g^k$ (useful for building a verifiable random function).
- DDH tuple: $(g, u, v, w)$ is a DDH tuple — namely, that $u = g^x$, $v = g^y$, and $w = g^{xy}$ for $x, g \in \mathbb{Z}_q$.
  
  - Useful for proving relations on ElGamal ciphertexts (e.g., that a particular ElGamal ciphertext encrypts either 0 or 1).
  
  - Useful building block in constructions of DDH-based oblivious transfer (OT) protocols — Naor-Pinkas (more details next week).
- Reduces to proving common discrete log: $(g, u, v, w)$ is a DDH tuple if and only if there is an $X$ such that $v = g^x$ and $w = u^x$.
Showing that \( h_1 = g_1^x \) and \( h_2 = g_2^x \):

\[
\begin{array}{c}
\text{prover} \quad r \in \mathbb{Z}_q^* \\
\begin{array}{c}
u_1 = g_1^r \\
u_2 = g_2^r \\
\end{array} \quad t \in \mathbb{Z}_q \\
\begin{array}{c}
t \\
z = r + tx \\
\end{array} \quad \text{check that } g_1^z = u_1 \cdot h_1^t \quad \text{and} \quad g_2^z = u_2 \cdot h_2^t
\end{array}
\]

Completeness and HVZK follows as in Schnorr's protocol.

Knowledge: Two scenarios:

1. If prover uses inconsistent commitment (i.e., \( u_1 = g_1^r \) and \( u_2 = g_2^s \) where \( r \neq s \)), then over choice of honest verifier's randomness, then prover can only succeed with probability at most \( 1/4 \):

\[
z = r_1 + x_1 t = r_2 + x_2 t \quad (\text{if verifier accepts})
\]

This means that

\[
(r_1 - r_2) = t (x_2 - x_1)
\]

If \( r_1 \neq r_2 \), there is at most \( 1 \) in \( \mathbb{Z}_q^* \), where this relation holds. Since \( t \) is uniform over \( \mathbb{Z}_q \), the verifier accepts with probability at most \( 1/4 \).

2. If prover succeeds with \( 1/4 \) probability, then it must use a "consistent" commitment. Can build extractor just as in Schnorr's protocol. Knowledge error larger by additive \( 1/4 \) term (from above analysis).
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?

\[
\begin{array}{c}
\text{prover (x)} \\
\pi = \text{Prove}(x) \\
\text{verifier (x)} \\
\end{array}
\]

Why do we care? Interaction in practice is expensive!

\[ b \in \{0,1\} \]

languages that can be decided by a randomized polynomial-time algorithm (BPP)

\[ \text{The simulator (for 2K property) can essentially be used to decide the language} \]

\[ \text{if } x \in L : S(x) \rightarrow \pi \text{ and } \pi \text{ should be accepted by the verifier (by 2K)} \]

\[ \text{if } x \notin L : S(x) \rightarrow \pi \text{ but } \pi \text{ should not be accepted by verifier (by soundness)} \]

\[ \text{NP \leq BPP \space (whp)} \]

Unfortunately, NIZKs are only possible for sufficiently-easy languages (i.e., languages in BPP).

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

1. **common random/reference string (CRS) model:**
   - **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 (symmetry between the capabilities of the real prover and the simulator).**

2. **random oracle model:**
   - **In this model, simulator can "probe" the random oracle (again, asymmetry between real prover and the simulator).**

\[ \Rightarrow \text{In both cases, simulator has additional "power" than the real prover, which is critical for enabling NIZK constructions for NP.} \]

Fiat-Shamir heuristic: from \( \Sigma \)-protocols to NIZK in RO model

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

\[ \begin{array}{c}
g_{\text{pub}}(g, s, r) \\
\text{prover} \\
r \in \mathbb{Z}_p \\
u \leftarrow g^r \\
z = u + c \omega \\
\text{verifier (g^s \mod \mathbb{Z}_p)} \\
\text{verify that } g^z = u \cdot h^c \\
\end{array} \]

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 \( H: \{0,1\}^n \rightarrow \mathbb{Z}_p \)

Namely, instead of sampling \( g^z \mod \mathbb{Z}_p \), we sample \( c \leftarrow H(g, h, u) \). — prover can now compute this quantity on its own!
Security of Fiat-Shamir:

1. Completeness: Same as Schnorr protocol.
2. Zero-knowledge: Same as in Schnorr protocol; namely, simulator samples \( c \approx Z_p, z \approx Z_p \), computes \( u \), and then queries RO at \((y,h,u)\) to \(c\).
3. Knowledge: Construct extractor as follows: given (possibly malicious) prover \( P^* \):
   1. Run \( P^* \) to obtain proof \( \pi = (u,z) \) where challenge \( c = H(y,h,u) \) at verification time.
   2. Note that at some point, \( P^* \) must have queried the random oracle on input \((y,h,u)\) and need to argue that with high probability, \( P^* \) will output proof with some committed value \( u \) (follow-by "fooling lemma")
   3. Run \( P^* \) again, but when it queries RO, use different responses.
   4. Can extract discrete log as before.

Signatures from discrete log in RO model (Schnorr):
- Verification key is \((y,h,g^y)\) and signing key is \(x\).
- To sign a message \( m \), signer proves knowledge of \( x \) (discrete log of \( h \)) using Fiat-Shamir (and where challenge is derived from message): e.g., \( c = H(y,h,u,m)\).
- Security essentially follows from security of Schnorr's identification protocol (together with Fiat-Shamir):
  1. Specifically, challenger answers signing queries using the ZK simulator (programming RO or needed for consistency).
  2. Forged signature on a new message \( m \) is a proof of knowledge of the discrete log (can be extracted from adversary).

More generally, any \( \Sigma \)-protocol can be used to build a signature scheme using the Fiat-Shamir heuristic (by using the message to derive the challenge via RO).

Length of Schnorr's signature:
- \( \text{vk: } (y,h,g^y)\)
- \( \text{sk: } x\)
- \( \sigma: (g^r, c = H(y,h,g^y,m), z = r + cx)\)
- Verification checks that \(g^{2z} \cdot g^x = c\), can be computed given other components so do not need to include.

But, can do better... observe that challenge \( c \) only needs to be 256-bit (the knowledge error of Schnorr is \( 1/161 \) where \( c \) is the set of possible challenges), so we can sample a 128-bit challenge rather than 256-bit challenge. Then instead of sending \((g^r, z)\), instead send \((c, z)\) and compute \(g^r \cdot g^{2z} \cdot g^x\) and that \(c = H(y,h,g^y,m)\). Then resulting signatures are 288 bits (128 bit challenge + 256 bit group element)

In practice, we use a variant of Schnorr's signature scheme called \( \text{DSA/ECDSA} \)

- Larger signatures (3 group elements = 512 bits) and proof only in "generic group" model [but we use it because Schnorr was patented... until 2008]

Important note: Schnorr signatures (and DSA/ECDSA) are randomized, and security relies on having good randomness.

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

  Then, we have
  \[
  \sigma_1 = \left( g^r, c_1 = H(y,h,g^y,m), z_1 = r + cx \right) \quad \Rightarrow \quad Z_1 - Z_2 = (c - c_1)x \quad \Rightarrow \quad x = (c - c_1)^{-1}(Z_1 - Z_2)
  \]

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

- In PlayStation 3, the randomness was a fixed constant! Enables hackers to deliver arbitrary firmware updates to device!

- TLS protocol
Deterministic Scheme: We want to replace the random value $r \in \mathbb{Z}_p$ 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 $\sigma \leftarrow \text{Sign}(sk, m, r)$.

Avoids randomness reuse/misuse vulnerabilities.