Zero-Knowledge Proofs from Scratch: Schnorr Identification and the Sigma Protocol

April 8, 2026

In 1985, Shafi Goldwasser, Silvio Micali, and Charles Rackoff asked a question that seemed like a non-starter: can you *prove* you know something without revealing what you know [Goldwasser, Micali, & Rackoff, 1985] (https://dl.acm.org/doi/10.1145/22145.22178)? Not encrypt it. Not hash it. But, prove it. Prove it, with mathematical certainty, while the person or system verifying it learns absolutely nothing about the secret itself.

The answer is yes. The framework they introduced, zero-knowledge proofs (ZKPs), is now foundational to systems ranging from blockchain privacy to password-authenticated key exchange to anonymous credentials.

The issue with most introductions to ZKPs is that they stop at analogies. “Imagine a cave with two paths.” “Imagine a colorblind alien with two balls.”

Analogies build intuition. They do not build understanding.

This post *attempts* to bridge that gap with a different approach. We’ll construct a ZKP from scratch using the Schnorr identification protocol, one of the first practical ZKP schemes. Every step uses numbers small enough to verify by hand. By the end, you will understand not just *that* ZKPs work, but *why* they work, and the general pattern that makes the entire field possible.

The Trapdoor: Discrete Logarithms

Every ZKP rests on a trapdoor: a mathematical operation that’s easy to perform but nearly impossible to reverse. Schnorr’s trapdoor is the **discrete logarithm**.

In ordinary arithmetic, logarithms are easy: if $2^5 = 32$, then $\log_2(32) = 5$. You just undo the exponentiation. Modular arithmetic breaks that symmetry. When you compute $g^x \bmod p$ — raise a number g to a power x, then take the remainder after dividing by a prime p — the forward direction is fast. But the reverse? Given the result, g, and p, try recovering the exponent x. This is the **discrete logarithm problem (DLP)**, and for large primes, no efficient *classical* algorithm exists to solve it.

> *This algorithm and those centered around DLP are **not** quantum-resistant: Shor’s algorithm solves DLP in polynomial time on a sufficiently large quantum computer.*

Let’s see this concretely. Pick a prime $p = 23$ and a base $g = 2$. Raise 2 to successive powers mod 23:

x	$2^x \bmod 23$
0	1
1	2
2	4
3	8
4	16
5	9
6	18
7	13
8	3
9	6
10	12

The results cycle through exactly 11 values before looping back: a **cyclic subgroup** of order $q = 11$, generated by $g = 2$[1].

Now: given that $2^x \bmod 23 = 13$, find x.

You can scan the table and find $x = 7$ trivially. But for a 256-bit prime, the table would have roughly 10^{77} entries. No table scan. No shortcut. This one-way property is the foundation of everything that follows.

Schnorr Identification

In 1989, Claus Schnorr had a practical problem: smart cards needed to prove their identity to terminals [Schnorr, 1991] (https://link.springer.com/article/10.1007/BF00196725). The card holds a secret key. The terminal needs to verify the card is legitimate. The obvious approach (send the secret, let the terminal check it, done) has a fatal flaw. The moment the key touches the terminal's memory, you are trusting that terminal not to store it, replay it, or get compromised while holding it. Even if the terminal promises to discard the key immediately, you have no way to verify that it did. Schnorr's solution eliminates the trust assumption entirely: a three-move conversation where the card proves it *knows* the key without the key ever leaving the card.

The protocol has four phases. First, the card and terminal agree on public parameters: a prime, a generator, and the order of the subgroup they generate[2]. The card uses these to derive a public key from its secret key; the public key is what the terminal will test against. The secret key never leaves the card.

From there, the conversation is three moves:

1. **Commit**. The card picks a random number, uses it to compute a commitment, and sends the commitment to the terminal. This locks in the card's randomness before it knows what the terminal will ask. *Think of sealing your answer in an envelope before hearing the question. Once the envelope is on the table, you can't magically swap what's inside.*
2. **Challenge**. The terminal sends back a random challenge. Because the card already committed, it cannot reverse-engineer a favorable response. *Like an auditor showing up unannounced and asking to see one specific ledger entry at random. You already filed the books; you don't get to choose which entry they inspect.*
3. **Respond**. The card combines its secret key, its random number, and the challenge into a single response value and sends it back. The secret is baked into the response but blended with randomness, so the response alone reveals nothing. *Like mixing a secret ingredient into a smoothie: the flavor proves it's there, but no one can separate it back out.*

The terminal then checks whether the response is consistent with the public key and the commitment. If both sides of the equation are equal, the card is authenticated. If not, it is rejected.

Now let's run through this with actual numbers. The table below assigns a concrete value to every symbol in the protocol, using the same prime $p = 23$ and generator $g = 2$ from the discrete logarithm section:

Symbol	Role	Type	Value
p	Prime modulus (public)	Parameter (prime $\in \mathbb{Z}$)	23
g	Generator (public)	Group element	2
q	Subgroup order (public)	Parameter (prime $\in \mathbb{Z}$, $q \mid p - 1$)	11
$\mathbb{Z}$_q	Scalar space = integers mod q	Finite field ($\mathbb{F}$_q)	{0, 1, …, 10}
($\mathbb{Z}$/p$\mathbb{Z}$)*	Multiplicative group mod p	Finite group (cyclic, order $p - 1$)	{1, 2, …, 22}
⟨g⟩	Cyclic subgroup generated by g	Finite group (cyclic, prime order q)	{1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18}
1	Identity element of ⟨g⟩ ($g^0 = g^\varphi = 1$)	Group element	1
x	Secret key (card only)	Scalar	7
h	Public key = g^x mod p	Group element	13
r	Random nonce (card only)	Scalar	4
a	Commitment = g^r mod p	Group element	16
c	Challenge (terminal picks)	Scalar	3
z	Response = $r + cx$ mod q	Scalar	3

Two kinds of values appear in the table: **scalars** (elements of $\mathbb{Z}$_q, the integers {0, 1, …, $q - 1$} with arithmetic mod q) and **group elements** (members of the **cyclic subgroup**, the set of values produced by exponentiating the generator mod p).

Just remember: exponents are scalars. Results of exponentiation are group elements[3].

With these values in hand, let's walk through each phase.

Setup. The card holds its secret key $x = 7$ and publishes public key h:

$$h = g^x \equiv 2^7 \equiv 128 \equiv 13 \pmod{23}$$

h = gx ≡ 27 ≡ 128 ≡ 13 (mod 23)

The terminal sees $h = 13$. The secret $x = 7$ never leaves the card.

1. Commit. The card picks random nonce $r = 4$, a scalar from our finite field, and computes commitment a:

$$a = g^r \equiv 2^4 \equiv 16 \pmod{23}$$

a = gr ≡ 24 ≡ 16 (mod 23)

It sends $a = 16$ to the terminal.

A fresh r must be sampled for every interaction. This one-time randomness is what will make the protocol zero-knowledge; without it, repeated runs with the same nonce would leak the secret.

2. Challenge. The terminal fires back random challenge $c = 3$ (another scalar from our finite field). Because the card already committed to a before seeing c, it cannot tailor its response or forge authenticity.

3. Respond. The card combines its nonce, the challenge, and its secret into response z:

$$z = r + c \cdot x \equiv 4 + 3 \cdot 7 \equiv 25 \equiv 3 \pmod{q}$$

z = r + c · x ≡ 4 + 3 · 7 ≡ 25 ≡ 3 (mod q)

Why mod q? Three facts, building on each other:

1. The generator g has **order** $q = 11$. This means that $g^{11} \equiv 1 \pmod{p}$: raising g to the q-th power returns the **identity element** of the group. The identity element is the element the cycle returns to, the starting point at exponent 0 in the discrete log table above. The **order** of g is defined as the smallest positive exponent that maps g back to this element.
2. Because the cycle length is q, exponents that differ by a multiple of q produce the same group element, i.e., $g^{14} \equiv g^3 \pmod{23}$ (since $14 \equiv 3 \bmod 11$).
3. So we can safely reduce any exponent mod q without changing the group element it produces. The response $z = 25$ and $z = 3$ map to the same $g^z \pmod{p}$ because $25 \equiv 3 \pmod{11}$. This is why the Respond step computes mod q: it keeps the exponent within ℤ_q while preserving the result.

The secret $x = 7$ is baked into z, but blended with $r = 4$. Without knowing r, extracting x is impossible. The response is indistinguishable from a random element of ℤ_q.

Verify. The terminal never learns r or x. It has only h, a, c, and z. It checks one equation:

$$g^z \equiv? \; a \cdot h^c \pmod{p}$$

gz ≡? a · hc (mod p)

Left side:

$$2^3 \equiv 8 \pmod{23}$$

23 ≡ 8 (mod 23)

Right side. We need $a \cdot h^c \equiv 16 \cdot 13^3 \pmod{23}$a · hc ≡ 16 · 133 (mod 23). Computing $h^c = 13^3$ hc = 133 by repeated multiplication:

1. h^2 : $13^2 \equiv 169 \equiv$ **8** (mod 23)
2. h^3 : $13 \cdot 8 \equiv 104 \equiv$ **12** (mod 23)
3. $a \cdot h^3$: $16 \cdot 12 \equiv 192 \equiv$ **8** (mod 23)

Both sides equal 8.

Why this particular equation? The verifier needs a check that uses only public values and that ties the response z back to both the commitment a and the public key h. The chosen equation is the minimal relation that does this: it lifts z into the group so that r (hidden inside a) and x (hidden inside h) can be checked against each other without either being revealed.

And it isn't a coincidence that the check balances. Substitute the definitions into both sides of the verification equation:

- **Left side**: $g^z \equiv g^{r+cx} \pmod{p}$gz ≡ gr+cx (mod p)
- **Right side**: $a \cdot h^c \equiv g^r \cdot (g^x)^c \equiv g^r \cdot g^{xc} \equiv g^{r+cx} \pmod{p}$
  a · hc ≡ gr · (gx)c ≡ gr · gxc ≡ gr+cx (mod p)

Both sides reduce to $g^{r+cx} \pmod{p}$gr+cx (mod p). The equation holds if and only if the prover used the correct x to compute z. A prover who doesn't know x succeeds against a random challenge with probability at most $1/q$, the **soundness error**, negligible for real parameters[4]. The terminal is now convinced the card knows the secret behind the public key they provided.

But *what* did it learn about the secret itself? Nothing. And we can prove it.

The Punchline: Zero-Knowledge

A zero-knowledge proof must ensure the verifier gains no information about the secret beyond the single bit: "the prover knows it." The formal tool for proving this is the **simulator argument**. The idea: if someone who doesn't know x can produce transcripts indistinguishable from real protocol runs, then real transcripts contain no information about x.

The simulator runs the protocol in reverse. Sample z and c uniformly from ℤ_q, then derive the commitment:

$$a = g^z \cdot h^{-c} \bmod p$$

a = gz · h−c mod p

With $z = 3$ and $c = 3$, this yields $a = 16$: exactly the transcript the real prover produced, without ever touching $x = 7$.

It works because the joint distribution of (a, c, z) is identical in both cases. In a real run, r is uniform in ℤ_q, which makes a uniform in $\langle g \rangle$ and $z = r + cx$ uniform in ℤ_q. In the simulation, z and c are sampled uniform and a is determined, but still uniform in $\langle g \rangle$. Same distribution, no distinguisher[5].

If transcripts carry no information about x, the verifier, who sees only transcripts, learns nothing. That is zero-knowledge.

The Sigma Protocol

We've demonstrated three properties. Let's name them.

Completeness. If the prover knows x, the verifier always accepts. The verification equation is an algebraic identity when z is correctly computed.

Special soundness. If the prover does *not* know x, the verifier rejects (except with negligible probability). More precisely: given two accepting transcripts (a, c_1, z_1) and (a, c_2, z_2) with the same commitment but different challenges, the secret is extractable. Subtract the response equations:

$$z_1 - z_2 = (r + c_1 x) - (r + c_2 x) = (c_1 - c_2)\, x$$

z1 − z2 = (r + c1x) − (r + c2x) = (c1 − c2) x

The nonce r cancels. Therefore $x = (z_1 - z_2)(c_1 - c_2)^{-1} \bmod q$x = (z1 − z2)(c1 − c2)−1 mod q. Two correct answers for one commitment means the secret is computable from the answers.

Honest-verifier zero-knowledge (HVZK). A simulator produces indistinguishable transcripts without the secret, as demonstrated above. The "honest-verifier" qualifier means the simulation assumes the verifier samples challenges uniformly from ℤ_q; see [5] for the distinction from full zero-knowledge[6].

Ronald Cramer, in his 1997 PhD thesis at CWI Amsterdam, named any three-move protocol satisfying these properties a **sigma protocol** (Σ-protocol) [Cramer, 1997] (https://ir.cwi.nl/pub/21438). The name comes from the shape of the letter sigma, which traces the three-move flow between prover and verifier.

The power of this abstraction is **composability**. Sigma protocols for individual statements combine into sigma protocols for compound statements:

- **AND composition**: prove you know x_1 AND x_2 (run both protocols in parallel with the same challenge)
- **OR composition**: prove you know x_1 OR x_2 without revealing which one (the CDS technique [Cramer, Damgård, & Schoenmakers, 1994] (https://link.springer.com/chapter/10.1007/3-540-48658-5_19))

This composability is what makes sigma protocols the workhorse of ZK systems, from anonymous credentials to blockchain proofs.

From Interactive to Non-Interactive

The Schnorr protocol as described is **interactive**: it needs a live verifier to supply the challenge. In 1986, Fiat and Shamir proposed eliminating the verifier entirely [Fiat & Shamir, 1986] (https://link.springer.com/chapter/10.1007/3-540-47721-7_12). The prover computes the challenge as a hash of the commitment, $c = H(g \parallel h \parallel a)$c = H(g ∥ h ∥ a), so the proof becomes a single message (a, c, z),

verifiable by anyone who can compute the hash. Because a hash function behaves like a random oracle, the prover cannot manipulate the challenge after committing. This **Fiat-Shamir transformation** converts any sigma protocol into a **non-interactive zero-knowledge proof (NIZK)**. Applied to Schnorr identification, it yields the Schnorr NIZK proof. The closely related Schnorr *signature* scheme extends this by binding a message into the hash, $c = H(g \parallel h \parallel a \parallel m)$ c = H(g ‖ h ‖ a ‖ m), tying the proof to a specific payload. The same Fiat-Shamir mechanism is at the root of most modern ZKP systems, including pairing-based SNARKs like Groth16, PLONK, and FFLONK. Those systems apply Fiat-Shamir over polynomial commitment schemes rather than sigma protocols directly, but the core idea, replacing a live verifier with a hash, is the same.

But Schnorr proves one specific statement: "I know the discrete logarithm of h with respect to g." What if you need to prove something more complex? How do you prove that you successfully executed an arbitrary computation correctly, or that your encrypted vote is valid and should be counted? These require encoding computations as systems of constraints, specifically **rank-1 constraint systems (R1CS)**, where each constraint takes the form $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$. Solving an R1CS means finding values for every variable such that every $A \cdot B = C$ constraint holds. That full assignment is called the **witness**. In a ZKP, the witness is what the prover knows and keeps hidden; the **proof** is the short message they send to convince the verifier that a valid witness exists, without revealing it. In the next post, we'll open the files that Circom generates and take a look at what these constraints look like.

Key Takeaways

- **Zero-knowledge proofs** let you prove knowledge of a secret without revealing it.
- **The sigma protocol** underpins most deployed discrete-log ZKPs.
- **The Schnorr identification protocol** proves knowledge in three moves: one exponentiation to commit, one multiplication and addition mod q to respond, one exponentiation and one multiplication to verify.
- **The simulator argument** is the formal mechanism proving zero-knowledge: if valid transcripts can be forged without the secret, real transcripts leak nothing.
- **Fiat-Shamir** eliminates the verifier by replacing the challenge with a hash, converting any sigma protocol into a non-interactive proof.

References

Goldwasser, S., Micali, S., & Rackoff, C. (1985). "The Knowledge Complexity of Interactive Proof-Systems." STOC '85. Link (https://dl.acm.org/doi/10.1145/22145.22178)

Fiat, A. & Shamir, A. (1986). "How to Prove Yourself: Practical Solutions to Identification and Signature Problems." CRYPTO '86. Link (https://link.springer.com/chapter/10.1007/3-540-47721-7_12)

Schnorr, C. P. (1991). “Efficient Signature Generation by Smart Cards.” Journal of Cryptology, 4(3). Link (https://link.springer.com/article/10.1007/BF00196725)

Cramer, R., Damgård, I., & Schoenmakers, B. (1994). “Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols.” CRYPTO ‘94. Link (https://link.springer.com/chapter/10.1007/3-540-48658-5_19)

Cramer, R. (1997). “Modular Design of Secure yet Practical Cryptographic Protocols.” PhD Thesis, CWI / University of Amsterdam. Link (https://ir.cwi.nl/pub/21438)

Hao, F. (2017). “Schnorr Non-interactive Zero-Knowledge Proof.” RFC 8235. Link (https://datatracker.ietf.org/doc/html/rfc8235)

Damgård, I. “On Σ-protocols.” Lecture notes, Aarhus University. Link (https://www.cs.au.dk/~ivan/Sigma.pdf)

Krenn, S. & Orrù, M. “Proposal: Σ-protocols.” ZKProof.org Standards Track. Link (https://docs.zkproof.org/pages/standards/accepted-workshop4/proposal-sigma.pdf)

Footnotes

1. The full multiplicative group mod 23 has order 22. We need a prime-order subgroup for Schnorr. The derivation chain runs top-down: (1) choose the subgroup order q (e.g., a 256-bit prime for ~128-bit security); (2) find a prime $p = kq + 1$, so Lagrange’s theorem guarantees an order-q subgroup of $(\mathbb{Z}/p\mathbb{Z})^*$ exists; (3) find a primitive root α mod p (an element whose powers hit every nonzero residue mod p), in practice by choosing p of a structured form such as a safe prime $p = 2q + 1$, so that $p - 1$ has only two prime factors and verifying a candidate requires just two exponentiations; no efficient deterministic algorithm is known for finding primitive roots in general, so real implementations rely on randomized sampling over a structured p; (4) derive $g = \alpha^{((p-1)/q)} \bmod p$, which collapses α’s full-group cycle down to a cycle of length q and produces a generator of the order-q subgroup. In our example: $q = 11$, $p = 23$ (since $22 = 2 \times 11$), $\alpha = 5$ (a primitive root mod 23), and $g = 5^2 \bmod 23 = 2$, the same generator used throughout the post. Modern implementations typically use elliptic curve groups where the entire group has prime order, avoiding subgroup considerations entirely. ↩
2. These three values (p, g, q) are public system parameters, not secrets. See footnote 1 for how they’re derived in practice. ↩
3. Why two different moduli? The cyclic subgroup has q elements, so exponents repeat every q steps: $g^{11} \bmod 23 = 1$, meaning $g^{14} \bmod 23 = g^3 \bmod 23$. Scalar arithmetic wraps at q because that’s where the cycle resets. The mod p reduction happens on the group element side, after exponentiation. ↩
4. The formal soundness guarantee is stronger than “guessing is hard.” Special soundness means that *any* algorithm capable of answering two different challenges for the same commitment can be used to extract the secret x. For the Schnorr protocol with challenge space of size q, the soundness error is $1/q$, roughly 2^{-256} for real parameters. ↩
5. Technically, the Schnorr protocol achieves **honest-verifier zero-knowledge (HVZK)**, not full zero-knowledge against arbitrary malicious verifiers. The simulation argument works because we assume the verifier selects its challenge uniformly at random. After applying the Fiat-Shamir transformation, the distinction collapses: the challenge is determined by a hash function, so the verifier has no freedom to choose adversarially. Fiat-Shamir’d sigma protocols achieve zero-knowledge in the random oracle model. ↩ ↩2

6. The three-move structure dominates practical ZKP systems, but it's not universal. Some constructions use multiple rounds of interaction (Bulletproofs), and non-interactive schemes built on probabilistically checkable proofs (PCPs) don't map cleanly to commit-challenge-respond. Still, the sigma protocol pattern underpins many deployed systems, including Schnorr signatures, Bulletproofs, and a range of anonymous-credential schemes. Pairing-based SNARKs like Groth16, PLONK, and FFLONK use the Fiat-Shamir transform too, but are not sigma protocols underneath. ↩
