ZERO KNOWLEDGE PROOF?

THAT SOUNDS USELESS!

Philipp Kant Ensurable Systems, Co-Founder and CTO

BobKonf 2026 · Berlin

KEYNOTE: DIGITAL SOVEREIGNTY

Data Sovereignty ***as an engineering problem***

DATENSPARSAMKEIT

EVERY DAY YOU PROVE THINGS ABOUT YOURSELF

- Your **age** → access content, buy things, enter venues
- Your **income** → loans, apartments
- Your **credentials** → jobs, professions
- Your **identity** → votes, contracts, borders

TODAY'S MODEL

```
You want to prove you're over 18.

You hand over:   your passport

They now have:   your full name
   your date of birth
   your nationality
   your address
   your passport number
   your photo
```

They needed: **one bit.** age ≥ 18

THIS IS DELEGATION OF TRUST.

- Every party you share data with becomes a target
- Breaches are inevitable
- You have no control once you've shared

ZERO-KNOWLEDGE ARGUMENT OF KNOWLEDGE

Instead of…	Just prove…
Your date of birth	*I know when I was born, and it's more than 18 years ago*
Your bank statements	*I know my financial history, and I know it meets the lender's criteria*
Your licence number	*I have a government-signed licence, and I know it hasn't expired or been revoked*

ZERO KNOWLEDGE PROOF?
THAT SOUNDS USELESS!

Actually, it's what makes the internet safe for secrets.

LET'S FORMALISE!

A zero-knowledge argument of knowledge (or zero-knowledge proof) is a protocol between a **prover** and a **verifier**.

The prover convinces the verifier they know a secret *without revealing the secret*.

We need the following properties:

- [ ] `Soundness` `dishonest provers always fail`
- [ ] `Zero-knowledge` `the prover reveals nothing about the secret`
- [ ] `Completeness` `honest provers always succeed`

A WEIRDLY SPECIFIC EXAMPLE

Let **C** and **Z** be public polynomials over $\mathbb{F}_p$

*"I know a private polynomial **w** such that **C(w) is divisible by Z**."*

Knowing **w**
⇔ knowing all its coefficients
⇔ being able to evaluate it at any point.

SCHWARTZ-ZIPPEL

*Two distinct polynomials of degree **d** agree at **at most d** points.*

```
|𝔽_p|  ≈  2²⁵⁶      degree  ≈  thousands

probability of a random point being one of them:  d / |𝔽_p|  ≈  10⁻⁷⁴
```

Knowing w ≡ evaluating it correctly at **one** random point.

THE PROTOCOL

1. Verifier picks random r
2. Prover reveals w(r) and t(r)
3. Verifier checks: C(w(r)) = t(r) · Z(r)

THE PROTOCOL

1. Verifier picks random r
2. Prover reveals w(r) and t(r)
3. Verifier checks: C(w(r)) = t(r) · Z(r)

The prover sees `r` before revealing anything — they could choose `w` and `t` *after* seeing `r`, without knowing a valid polynomial.

The prover must commit to `w` and `t` before seeing `r`.

COMMITMENT SCHEMES

A hash gives you **binding**: publish `H(w)` → you can't change `w` afterwards.
But the verifier needs `w` to check — **no hiding**.

Polynomial commitments add both **hiding** and **evaluation proofs**:

- **Binding**: can't change `w` after committing
- **Hiding**: `commit(w)` reveals nothing about `w`
- **Evaluation proofs**: prove `w(r) = v` without revealing `w`

The extra ingredient: **elliptic curve pairings**

THE PROTOCOL

```
0.  Prover publishes  commit(w),  commit(t)
1.  Verifier picks random r
2.  Prover reveals  w(r),  t(r)  + evaluation proofs
3.  Verifier checks:  C(w(r))  =  t(r) · Z(r)
```

☑ SOUNDNESS

☑	Soundness	dishonest provers always fail
☐	Zero-knowledge	the prover reveals nothing about the secret
☐	Completeness	honest provers always succeed

ZERO KNOWLEDGE?

What the verifier sees	What it reveals
`commit(w)`, `commit(t)`	nothing — coefficients are hidden
`w(r)`, `t(r)`	one point on a degree-d polynomial

ZERO KNOWLEDGE?

What the verifier sees	What it reveals
`commit(w)`, `commit(t)`	nothing — coefficients are hidden
`w(r)`, `t(r)`	one point on a degree-d polynomial

One point cannot determine a degree-d polynomial.

The verifier learns *very little*. Only nearly **zero** knowledge.

ZERO KNOWLEDGE?

What the verifier sees	What it reveals
`commit(w)`, `commit(t)`	nothing — coefficients are hidden
`w(r)`, `t(r)`	one point on a degree-d polynomial

One point cannot determine a degree-d polynomial.

The verifier learns *very little*. Only nearly **zero** knowledge.

A technique called **blinding** removes the "nearly": the prover adds a fresh random term to `w` before committing, making the evaluation point completely uninformative. **Zero knowledge** ✓

☑ ZERO-KNOWLEDGE

☑	Soundness	dishonest provers always fail
☑	Zero-knowledge	the prover reveals nothing about the secret
☐	Completeness	honest provers always succeed

COMPLETENESS

We can prove **"I know `w` such that `C(w)` is divisible by `Z`"**.

COMPLETENESS

We can prove **"I know `w` such that `C(w)` is divisible by `Z`"**.

That's great, I guess?

COMPLETENESS

We can prove **"I know `w` such that `C(w)` is divisible by `Z`"**.

That's great, I guess?

But what about more general secrets?

COMPLETENESS

We can prove **"I know `w` such that `C(w)` is divisible by `Z`"**.

That's great, I guess?

But what about more general secrets?

Claim: any secret input to any computation can be turned into a claim of exactly this form.

START SIMPLE: `age = 18`

```
assert current_year - birth_year = 18
```

One constraint. One private input. Let's turn this into a circuit.

THE CIRCUIT

```
birth_year ──→ [ assert current_year − birth_year − 18 = 0 ]
```

- One **gate**, asserting one equation
- One private **wire value**: birth_year
- current_year and 18 are public — baked into the gate, not wires

ENCODING THE WIRE VALUES — W

One private wire, one evaluation point:

```
w(ω⁰) = birth_year       (private)
```

w is the unique polynomial of degree 0 through this one point.

ENCODING THE WIRE VALUES — W

One private wire, one evaluation point:

$$w(\omega^0) = \text{birth\_year} \qquad \text{(private)}$$

w is the unique polynomial of degree 0 through this one point.

With **n wires**, we'd have n evaluation points:

$$w(\omega^0) = \text{wire}_0, \quad w(\omega^1) = \text{wire}_1, \quad \ldots, \quad w(\omega^{n-1}) = \text{wire}_{n-1}$$

w is the unique polynomial of degree n−1 through these n points.

ENCODING THE CONSTRAINTS — C AND C(W)

C — the gate's constraint rule, as a polynomial in the wire value (public):

```
C(v)  =  current_year − v − 18
```

C(w) — function composition: substitute the wire polynomial into C:

```
C(w)(x)   =  C(w(x))  =  current_year − w(x) − 18

C(w)(ω⁰)  =  C(w(ω⁰))  =  current_year − birth_year − 18
```

Z AND THE DIVISIBILITY CLAIM

```
w(ω⁰) = birth_year

C(w)(ω⁰)  =  C(w(ω⁰))  =  current_year − birth_year − 18
```

Z — one root at the gate position:

```
Z(x) = (x − ω⁰)          zero at ω⁰, nowhere else
```

```
C(w) divisible by Z
    ⟺  C(w)(ω⁰) = 0          ← factor theorem
    ⟺  gate equation holds
    ⟺  birth_year = current_year − 18
```

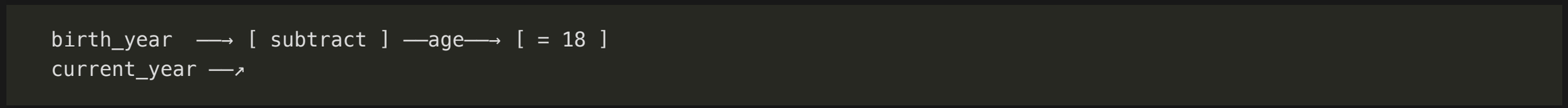
(factor theorem: (x−a) divides P iff P(a) = 0)

THIS IS THE WEIRDLY SPECIFIC EXAMPLE

```
"I know w and t  such that  C(w) = t · Z"
```

- w — the wire polynomial: $w(\omega^0)$ = birth_year
- C — the gate constraint: C(v) = current_year − v − 18, public
- Z — $(x - \omega^0)$, zero at the gate position
- t — quotient: certificate that the constraint passes

THE SAME SCHEME, MORE WIRES



- **w** — one evaluation point per wire: birth_year, current_year, age
- **C** — one constraint per gate, in terms of the wire values
- **Z** — one root per gate: $(x - \omega^0)(x - \omega^1)$
- **C(w) = t · Z** — same claim, same verification

GENERALISING TO age ≥ 18

age = 18 needed **1 gate**. age ≥ 18 requires a range check:

```
diff = age − 18
diff = b₇·128 + b₆·64 + … + b₀·1
bᵢ · (1 − bᵢ) = 0,  for each i
```

1 gate for the subtraction, 1 for the recombination, 8 boolean gates — one per bit. Each $b_i \in \{0,1\}$ enforced by the boolean gate. Maximum 8-bit value: **255**.

In $\mathbb{F}_p$, subtraction wraps: $age < 18 \to diff \approx 2^{256}$ — too large for any 8-bit assignment. ✗ $age \geq 18 \to diff \leq 255$ — a valid witness exists. ✓

~10 gates. Same structure — one equation per gate, one root in Z.

☑ COMPLETENESS

```
☑  Soundness       dishonest provers always fail
☑  Zero-knowledge  the prover reveals nothing about the secret
☑  Completeness    honest provers always succeed
```

What we just built has a name: a **SNARK** — **S**uccinct **N**on-interactive **AR**gument of **K**nowledge.

so.

Who is looking forward to writing circuits and polynomial constraints?

so.

Who is looking forward to writing circuits and polynomial constraints?

We could just drop it all into an LLM and hope for the best.

so.

Who is looking forward to writing circuits and polynomial constraints?

We could just drop it all into an LLM and hope for the best.

Or we could use a language **designed** for exactly this.

THE ZKP DSL LANDSCAPE

Before DSLs: every gate specified by hand — Rank-1 Constraint Systems (R1CS). **Now:** write code. Get circuits.

DSL	Syntax	Notes
Noir	Rust-like	Clean, readable, Aztec-backed
Cairo	Rust-like	Starknet ecosystem
o1js	TypeScript	Mina protocol
Leo	Rust-like	Aleo network

EU DIGITAL ID WALLET

Prove you're over 18 to a service — without revealing your date of birth.

THE NOIR CIRCUIT

```
fn main(
    birth_year:   u32,         // PRIVATE: stays in your wallet

    credential:   pub Field,  // PUBLIC: pedersen_hash([birth_year])
    current_year: pub u32,    // PUBLIC: verifier provides
    required_age: pub u32,    // PUBLIC: verifier provides
) {
    let computed = std::hash::pedersen_hash([birth_year as Field]);
    assert(computed == credential);

    let age = current_year - birth_year;
    assert(age >= required_age);
}
```

THIS ONE LINE...

```
assert(age >= required_age);
```

...compiles to the **entire bit-decomposition circuit**.

THE CREDENTIAL

```
credential:   pub Field,  // pedersen_hash([birth_year])
```

```
GOVERNMENT ISSUES:
  credential = pedersen_hash([1987])
             = 0x19dd02231fae...
  Stores in your wallet.

VERIFIER RECEIVES:
  credential   = 0x19dd02231fae...  ← can't reverse this
  current_year = 2026
  required_age = 18
  proof        = <blob>
```

PRODUCTION CREDENTIALS

```
CREDENTIAL ISSUANCE (once, by government)
  fields      = { birth_date, name, nationality, device_pubkey, ... }
  merkle_root = MerkleTree(fields).root
  signature   = gov_key.sign(merkle_root)
  => stored in your wallet
```

```
PROOF GENERATION (per use, on your device)
  ZKP circuit proves all at once:
    a) birth_date is a leaf in the Merkle tree    ← real credential
    b) merkle_root is signed by the government    ← real credential
    c) device_pubkey is a leaf in the same tree   ← this is mine
    d) I can sign the nonce with device_privkey   ← I am present now
    e) current_year − birth_year >= 18            ← age claim holds

  Output: one proof, ~16KB
```

WHAT EACH LAYER ADDS

Layer	Proves	Prevents
Hash commitment	You know birth_year	Lying about birth_year
Issuer signature	Credential signed by authority	Forging credentials
Merkle proof	birth_year is one field of many	Mixing fields across creds
Device binding	Credential tied to your device key	Credential theft/sharing
Biometric	Device key unlocked by rightful user	Lost/stolen device

DEMO: PROVER SIDE

```
node prove.mjs
```

```
Generating witness...
  birth_year   = 1987  <-- stays here, never sent
  current_year = 2026
  required_age = 18
  age          = 39

Generating proof...
Proof size:    2192 bytes
Public inputs: ["0x19dd02231fae...", "0x7e6", "0x12"]

proof.json written -- send this to the verifier.
```

DEMO: VERIFIER SIDE

```
node verify.mjs
```

```
Verifying proof...
Public inputs from proof:
  credential   = 0x19dd02231fae...
  current_year = 2026
  required_age = 18

VERIFIED: age >= 18.
The prover's birth_year was never revealed.
```

HOW DOES THE VERIFIER KNOW WHAT'S BEING PROVED?

```
Same Noir source
     ↓
Same circuit  →  same C and Z
     ↓
Same verification key
     ↓
Can verify any proof from that circuit
```

The circuit is the **auditable specification**.

THE PATTERN GENERALISES

```
Age verification    →  birth_year satisfies age ≥ N

Credit score        →  score satisfies score ≥ 700

Election integrity  →  ballot is valid without revealing vote

Credential check    →  member of a set, without revealing which

Compliance          →  computation was done correctly,
          without revealing the data
```

THE DESIGN SHIFT

Old model:

Give me your data. I'll check it.

New model:

You check it. Give me the proof.

YOUR PHONE BECOMES THE ROOT OF YOUR IDENTITY

Not a government database. Not a service's server.

YOU.

☑ ALL THREE PROPERTIES

```
☑  Soundness        dishonest provers always fail
☑  Zero-knowledge   the proof reveals nothing
☑  Completeness     honest provers always succeed
```

ZERO KNOWLEDGE PROOF?
THAT SOUNDS USELESS.

Actually, it's what makes the internet safe for secrets.

THANK YOU

Philipp Kant

ensurable.systems

@philippkant.bsky.social

HOW BLINDING WORKS

Replace w with $\tilde{w}$ before committing:

$$\tilde{w}(x) = w(x) + b(x) \cdot Z(x)$$

$b(x)$ — a fresh random polynomial, chosen by the prover.

HOW BLINDING WORKS

Replace w with $\tilde{w}$ before committing:

$$\tilde{w}(x) = w(x) + b(x) \cdot Z(x)$$

$b(x)$ — a fresh random polynomial, chosen by the prover.

Correctness preserved — at the gate position ω^0:

$$Z(\omega^0) = 0 \quad \rightarrow \quad \tilde{w}(\omega^0) = w(\omega^0) + b(\omega^0) \cdot 0 = \text{birth\_year}$$

Wire value unchanged → $C(\tilde{w})(\omega^0) = C(w)(\omega^0) = 0$ → quotient t still exists. ✓

HOW BLINDING WORKS

Replace w with $\tilde{w}$ before committing:

```
w̃(x)  =  w(x)  +  b(x) · Z(x)
```

b(x) — a fresh random polynomial, chosen by the prover.

Correctness preserved — at the gate position ω^0:

```
Z(ω⁰) = 0   →   w̃(ω⁰) = w(ω⁰) + b(ω⁰)·0 = birth_year
```

Wire value unchanged → $C(\tilde{w})(\omega^0) = C(w)(\omega^0) = 0$ → quotient t still exists. ✓

Zero knowledge achieved — at the random challenge r (outside the domain):

$$Z(r) \neq 0 \quad \rightarrow \quad \tilde{w}(r) = w(r) + b(r) \cdot Z(r)$$

$b(r) \cdot Z(r)$ is a fresh random mask → $\tilde{w}(r)$ reveals nothing about $w(r)$. ✓