Halo2

zk-snark

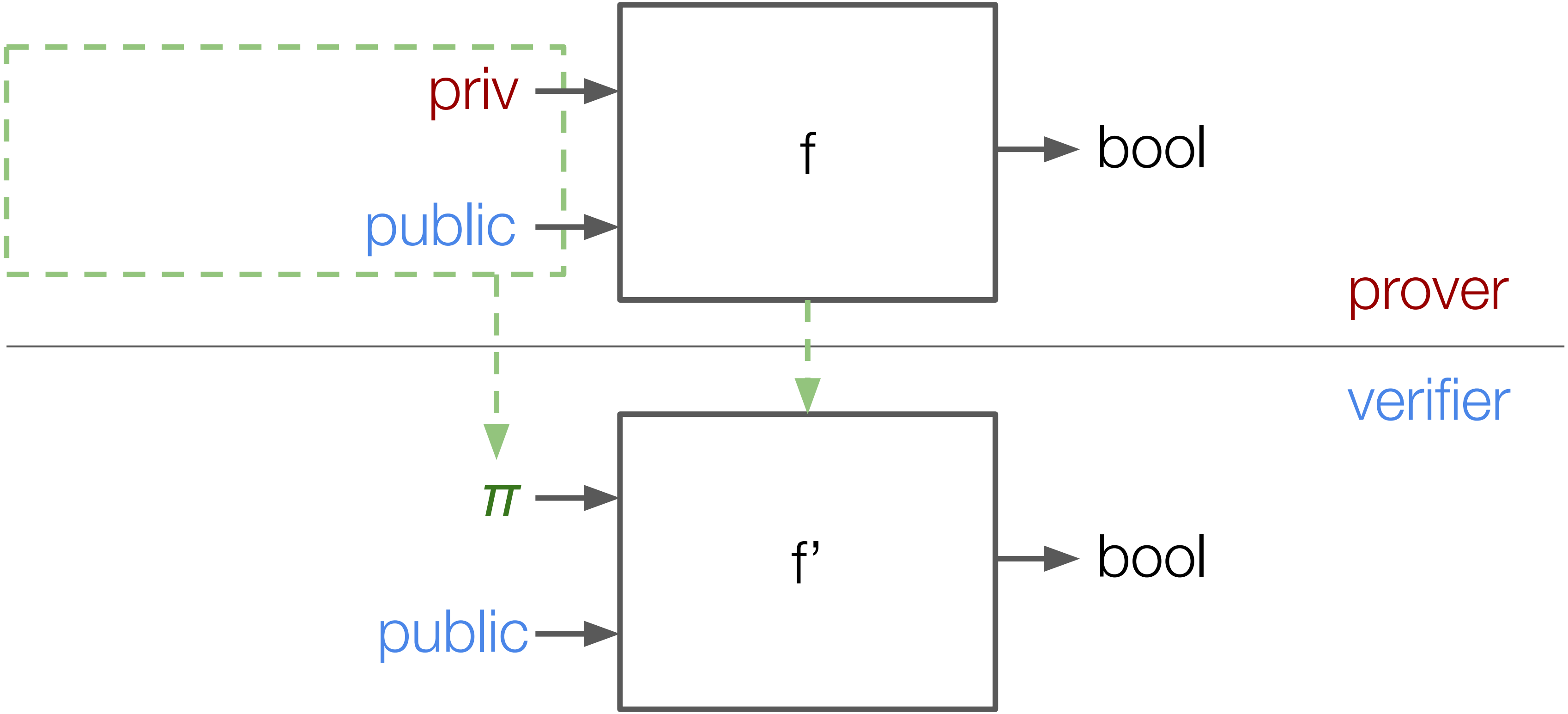
- Zero Knowledge
- Succinct
- Non-interactive
- ARgument of Knowledge

“Zero-knowledge” proofs allow one party (the prover) to prove to another (the verifier) that a statement is true, without revealing any information beyond the validity of the statement itself. For example, given the hash of a random number, the prover could convince the verifier that there indeed exists a number with this hash value, without revealing what it is. (Zcash)

zk-snark

- “Programmable” zero knowledge proofs
- Compatible with ethereum via bn128
- Growing set of tools
- Compression / scalability
- Privacy & Anonymity

zk-snark



priv

public

f

bool

prover

verifier

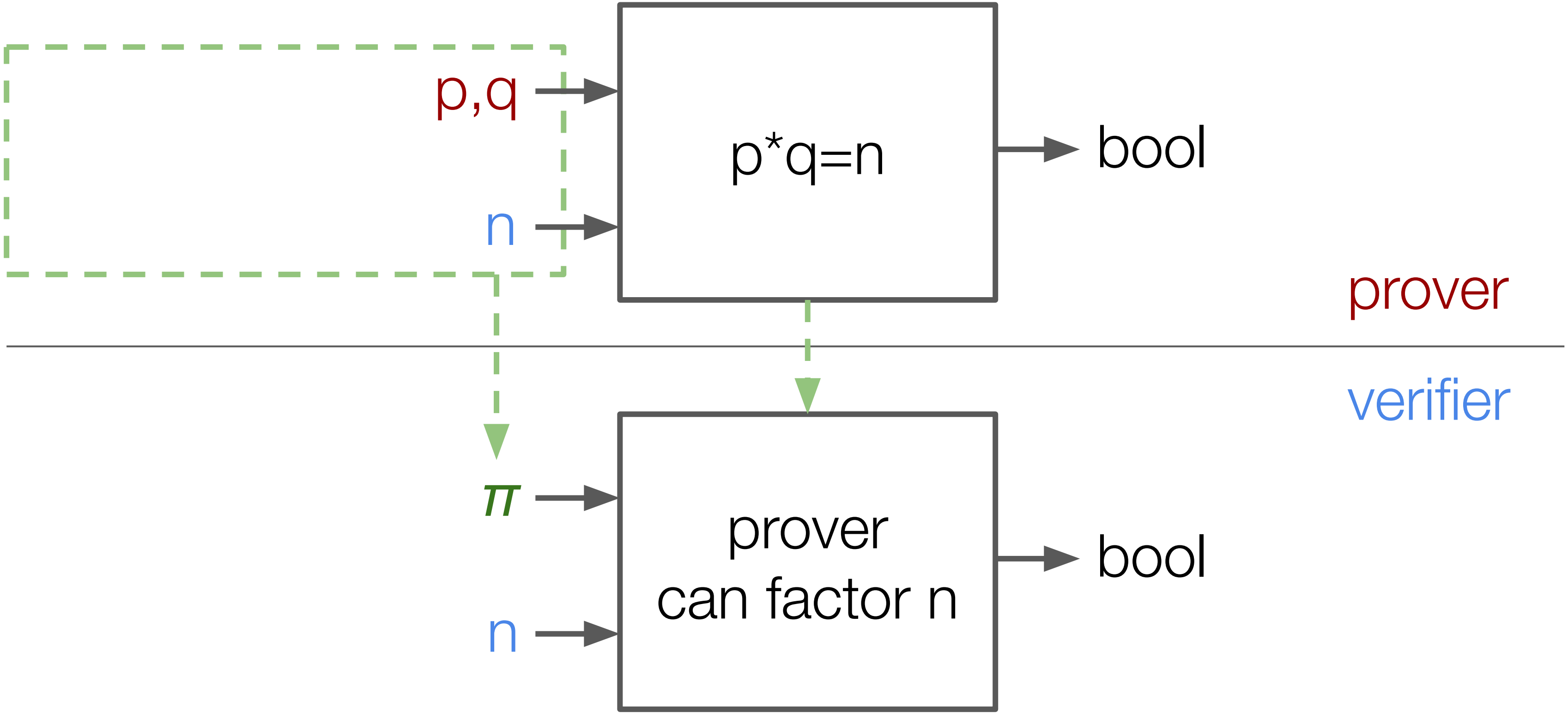
π

f’

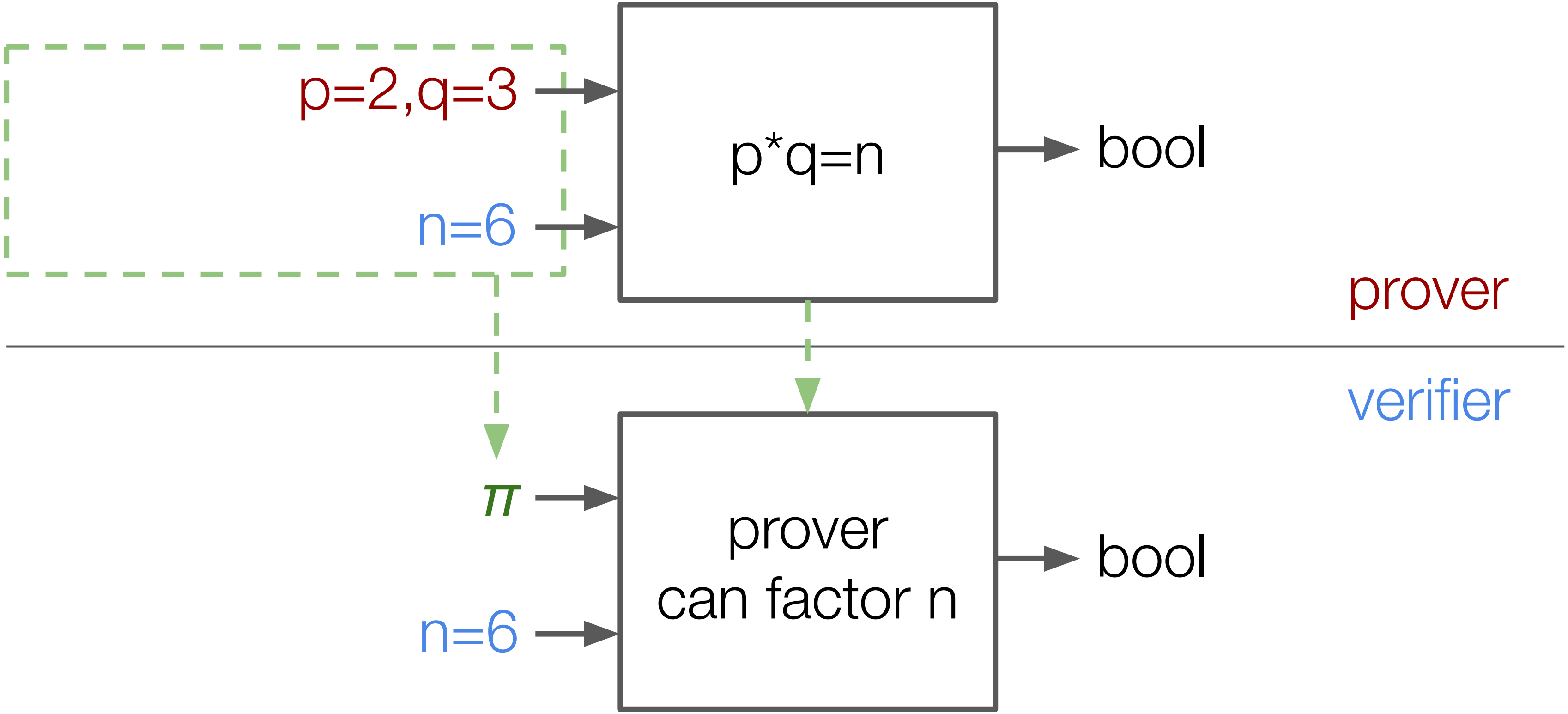
public

bool

proof-of-factor



proof-of-factor



p=2,q=3

n=6

p*q=n

bool

prover

verifier

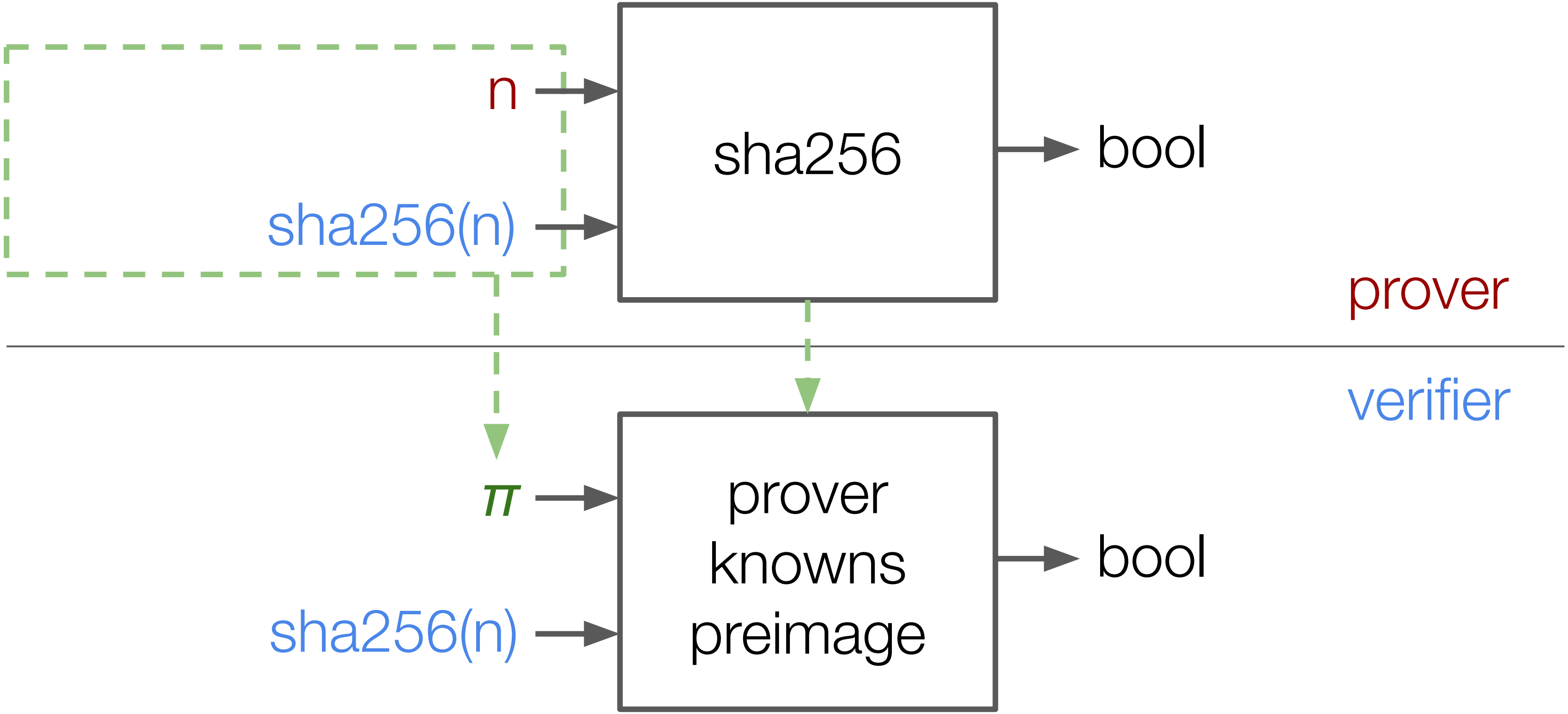
π

n=6

prover
can factor n

bool

knowledge of preimage



n

sha256(n)

sha256

bool

prover

verifier

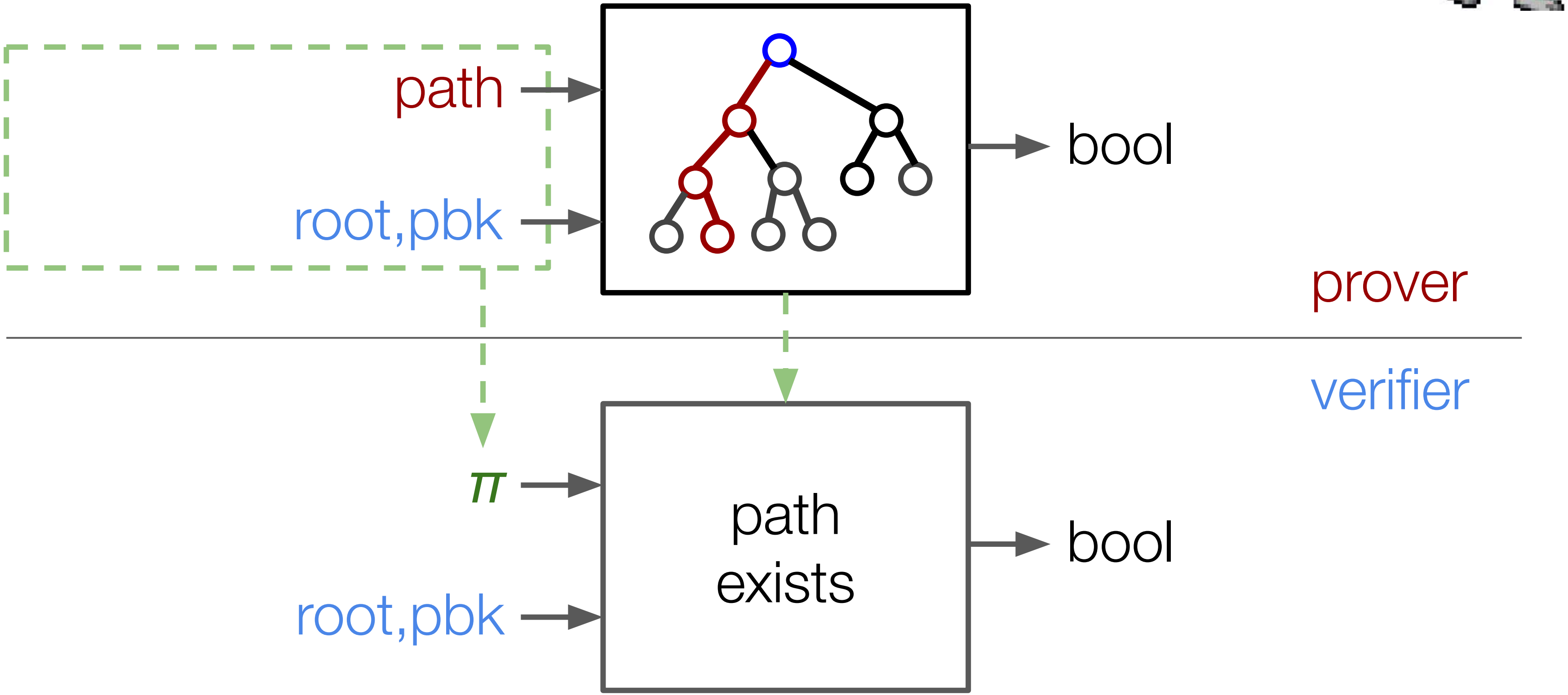
π

sha256(n)

prover knowns preimage

bool

proof-of-existence



path

root,pbk

bool

prover

verifier

π

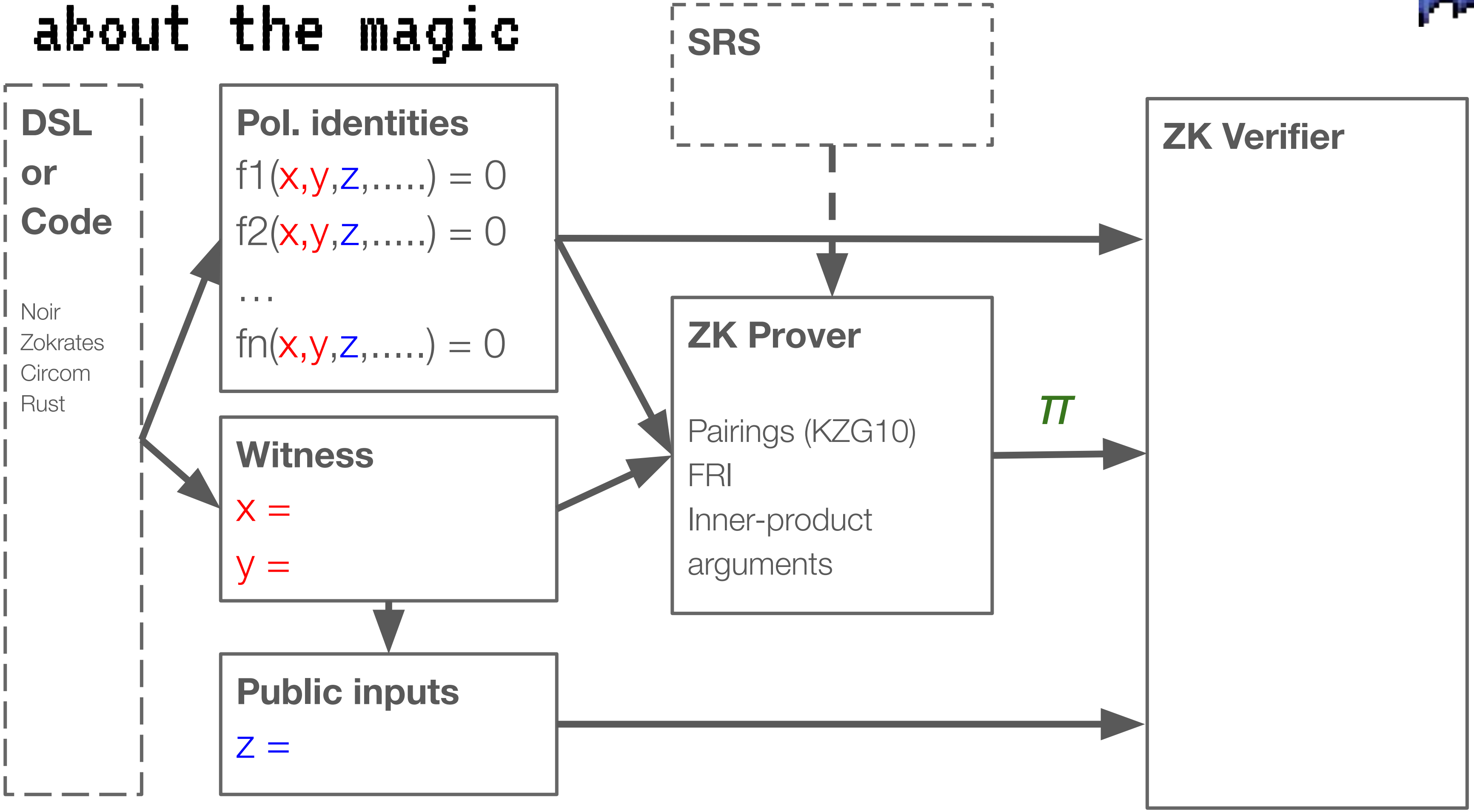
path exists

root,pbk

bool

check https://github.com/zcash-hackworks/babyzoe/blob/master/talks/2016-07-27-IC3---SNARKs-for-Ethereum.pdf !

about the magic



DSL or Code

Noir
Zokrates
Circom
Rust

Pol. identities

$f1(x,y,z,.....) = 0$

$f2(x,y,z,.....) = 0$

…

$fn(x,y,z,.....) = 0$

Witness

$x =$

$y =$

Public inputs

$z =$

SRS

ZK Prover

Pairings (KZG10)
FRI
Inner-product arguments

π

ZK Verifier

Constrains

R1CS

$[a_1S_1+..+a_nS_n]$ *

$[b_1S_1+..+b_nS_n]$ +

$[c_1S_1+..+c_nS_n]=0$

PLONK Custom Gates

$f(x)=0$

PLONK

$a(X)b(X)qM(X)+a(X)qL(X)+b(X)qR$

$(X)+c(X)qO(X)+PI(X)+qC(X)$

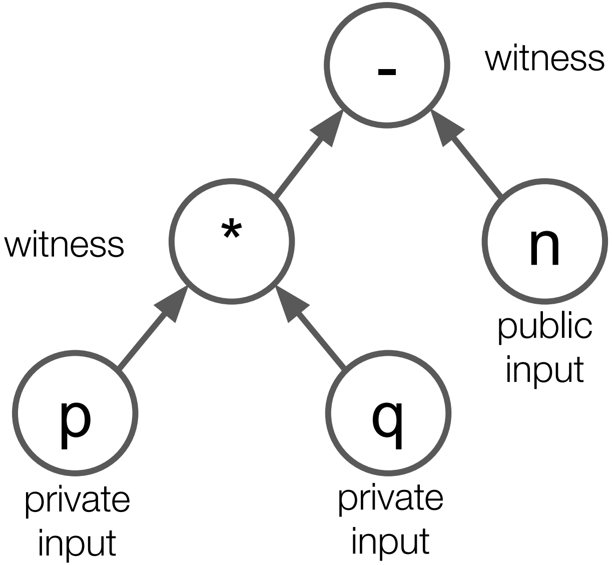
+ copy constraints

PLOOKUP

$A \subseteq B$

f

- No turing-complete
- Build an arithmetic circuits with +,-,*, that evals to zero
- $[a_1S_1+..+a_nS_n] * [b_1S_1+..+b_nS_n] + [c_1S_1+..+c_nS_n] = 0$
- E.g n=p*q
- $1S_p * 1S_q + -1S_n = 0$



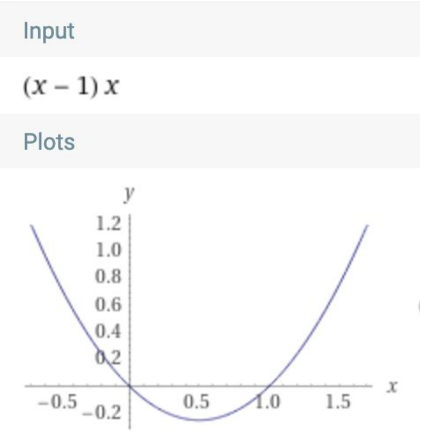
f

- n=p AND q

$(-1S_p + 1S_{ONE})*(1S_p) = 0$

$(-1S_q + 1S_{ONE)}]*(1S_q) = 0$

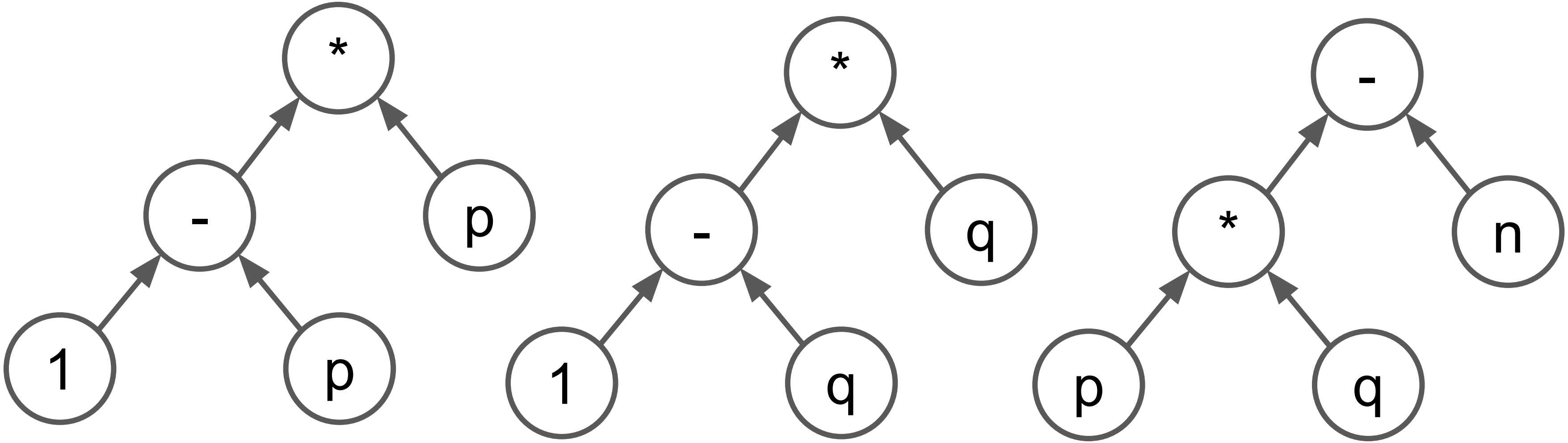
$(1S_p)*(1S_q)+(-1S_n) = 0$



Input

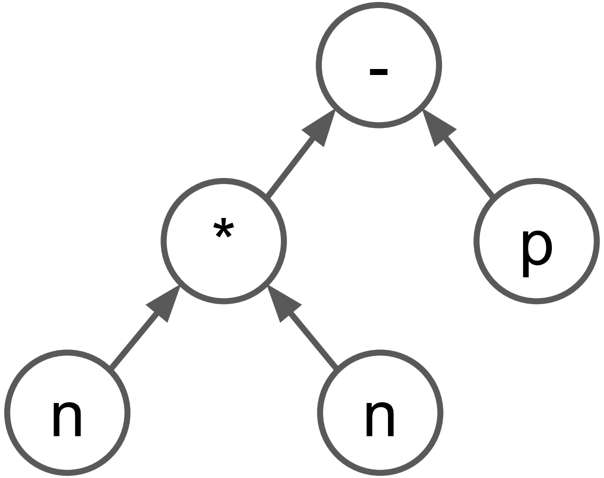
$(x - 1)\,x$

Plots



f

- $n=\sqrt{p}$
- $(1S_n)*(1S_n)+(-1S_p)$



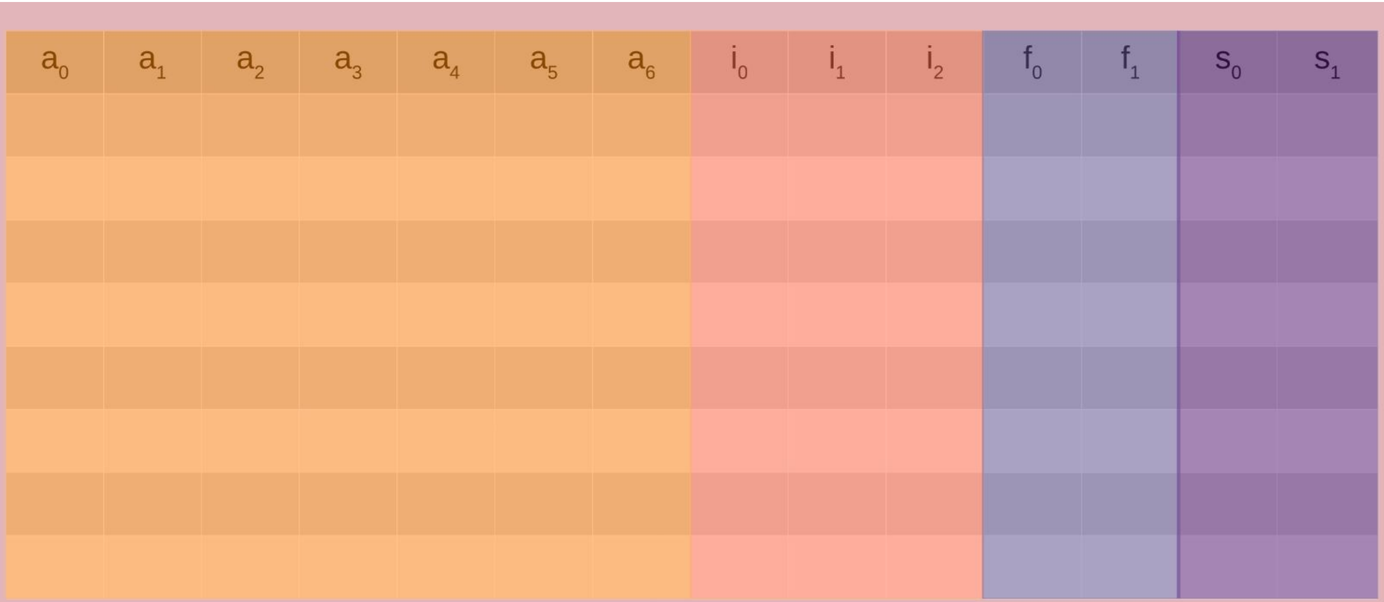
HALO2

- Inner product arguments (no trusted setup)
- All is “custom gates” + PLOOKUP
- ECC
- ZCash / EF ZKEvm / Filecoin / Dark.fi
- Rust
- Can be used with KZG
- PLONKish arithmatization

HALO2

- Create your (zk) chips
- Create your circuit
  - Define the inputs
  - Load constants
  - Load private inputs
  - Connect chips
  - Expose cells as public inputs

PLONKish



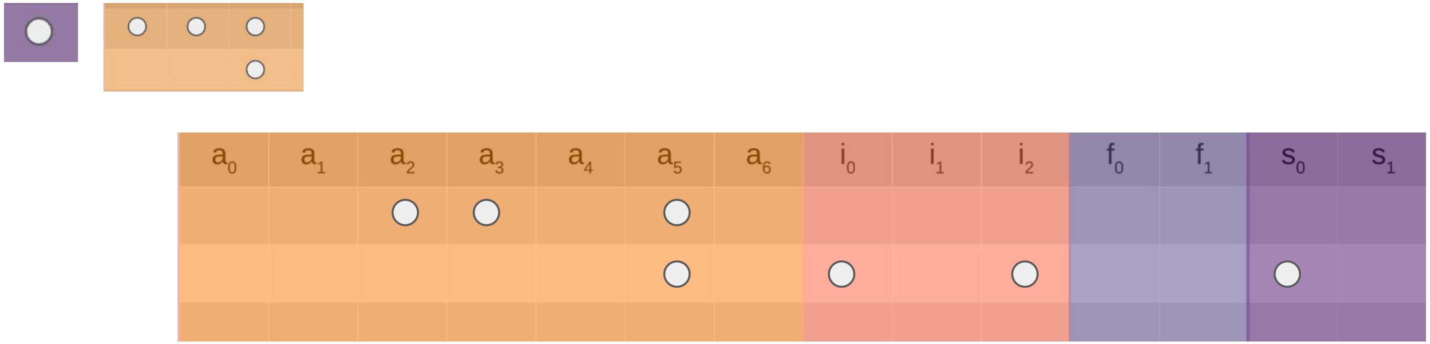
a_0	a_1	a_2	a_3	a_4	a_5	a_6	i_0	i_1	i_2	f_0	f_1	s_0	s_1

advice columns & instance columns
vary with each proof instance

fixed columns & selector columns
do not vary across instances

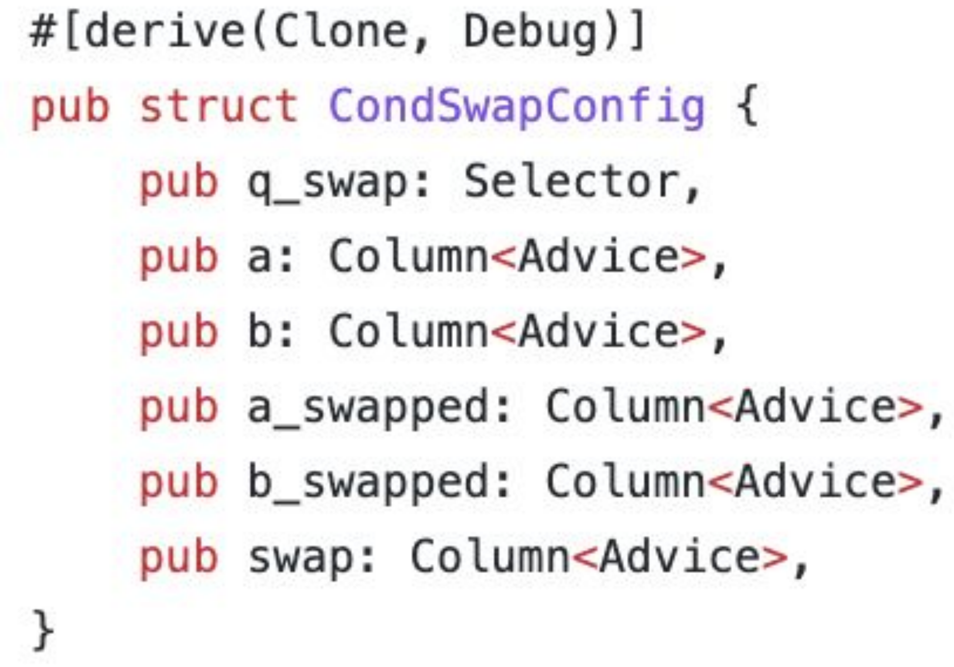
PLONKish chips

- You create your own “chips”
  - `struct{}` - Define how many cells needs to use
  - `configure()`
    - Define the custom gates - poly relationship between cells
    - The offset and “properties” of the cells used
  - `Custom{}` - Define the “instructions” for the chip
    - Copy constrains
    - How witness is generated (if in witness generation mode)



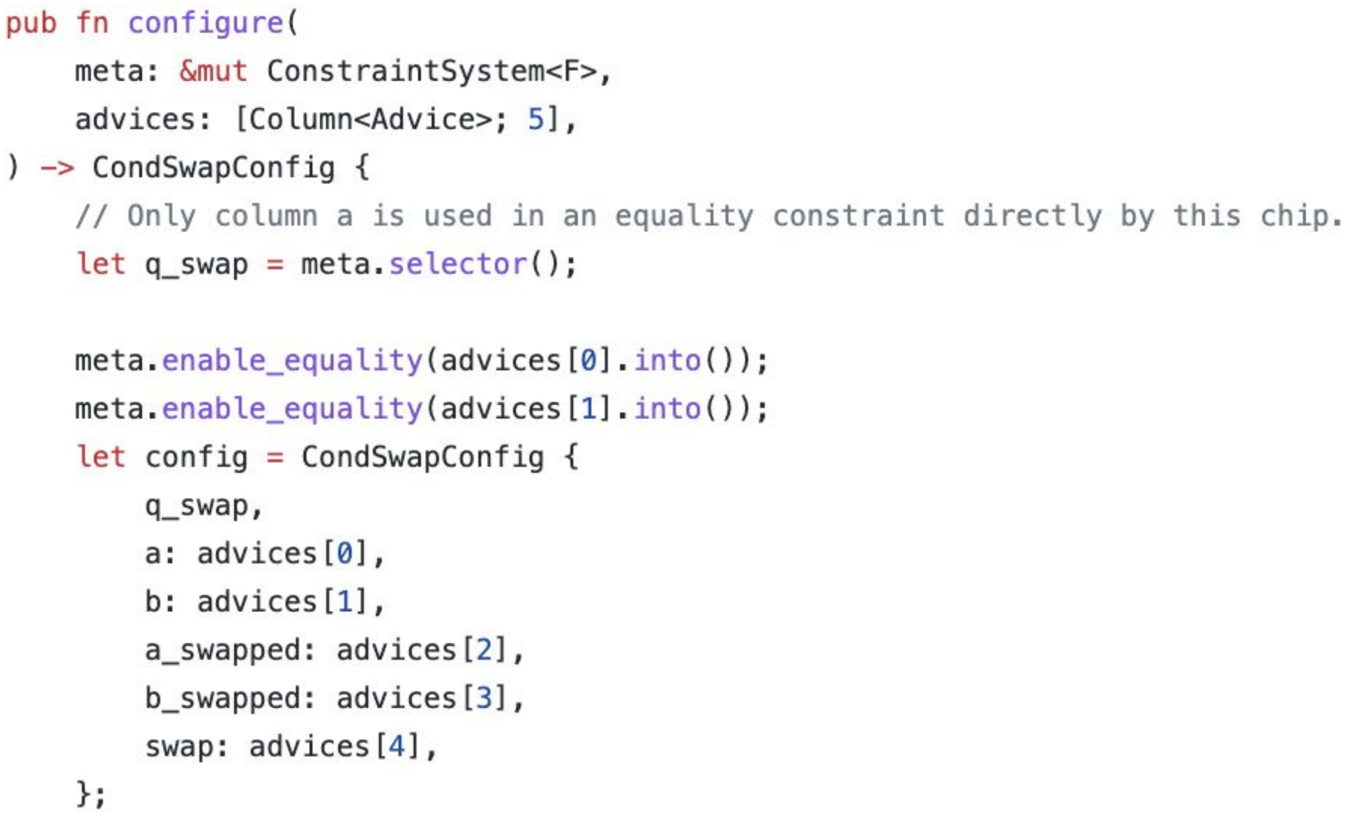
a_0	a_1	a_2	a_3	a_4	a_5	a_6	i_0	i_1	i_2	f_0	f_1	s_0	s_1
		○	○		○								
					○		○		○			○	

CondSwapChip



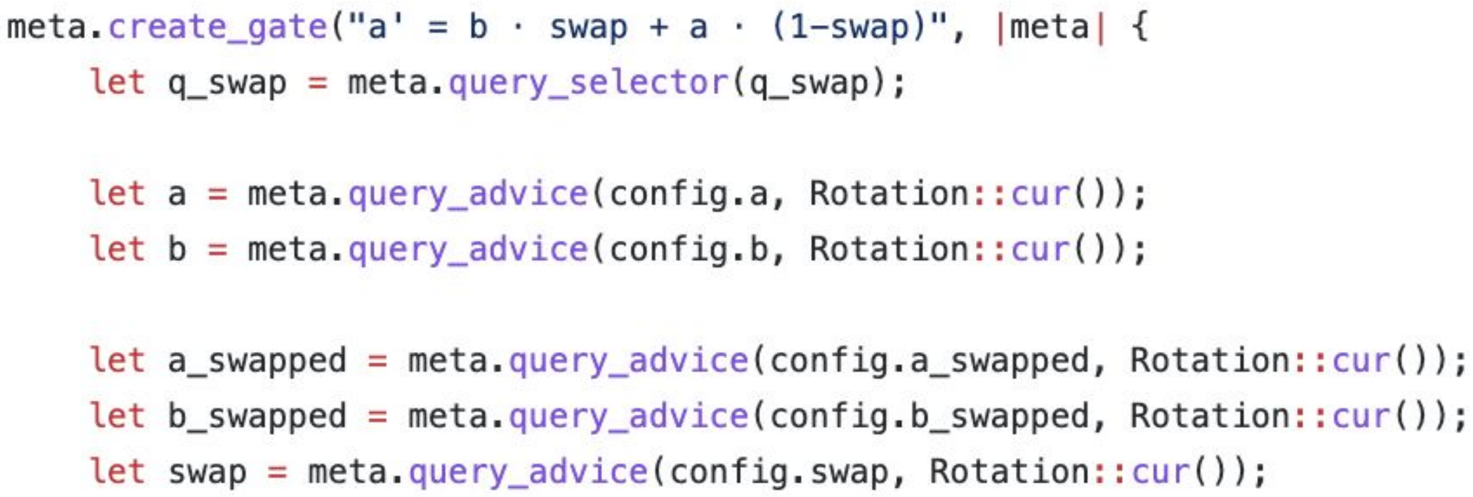
```
#[derive(Clone, Debug)]
pub struct CondSwapConfig {
    pub q_swap: Selector,
    pub a: Column<Advice>,
    pub b: Column<Advice>,
    pub a_swapped: Column<Advice>,
    pub b_swapped: Column<Advice>,
    pub swap: Column<Advice>,
}
```

CondSwapChip - configure



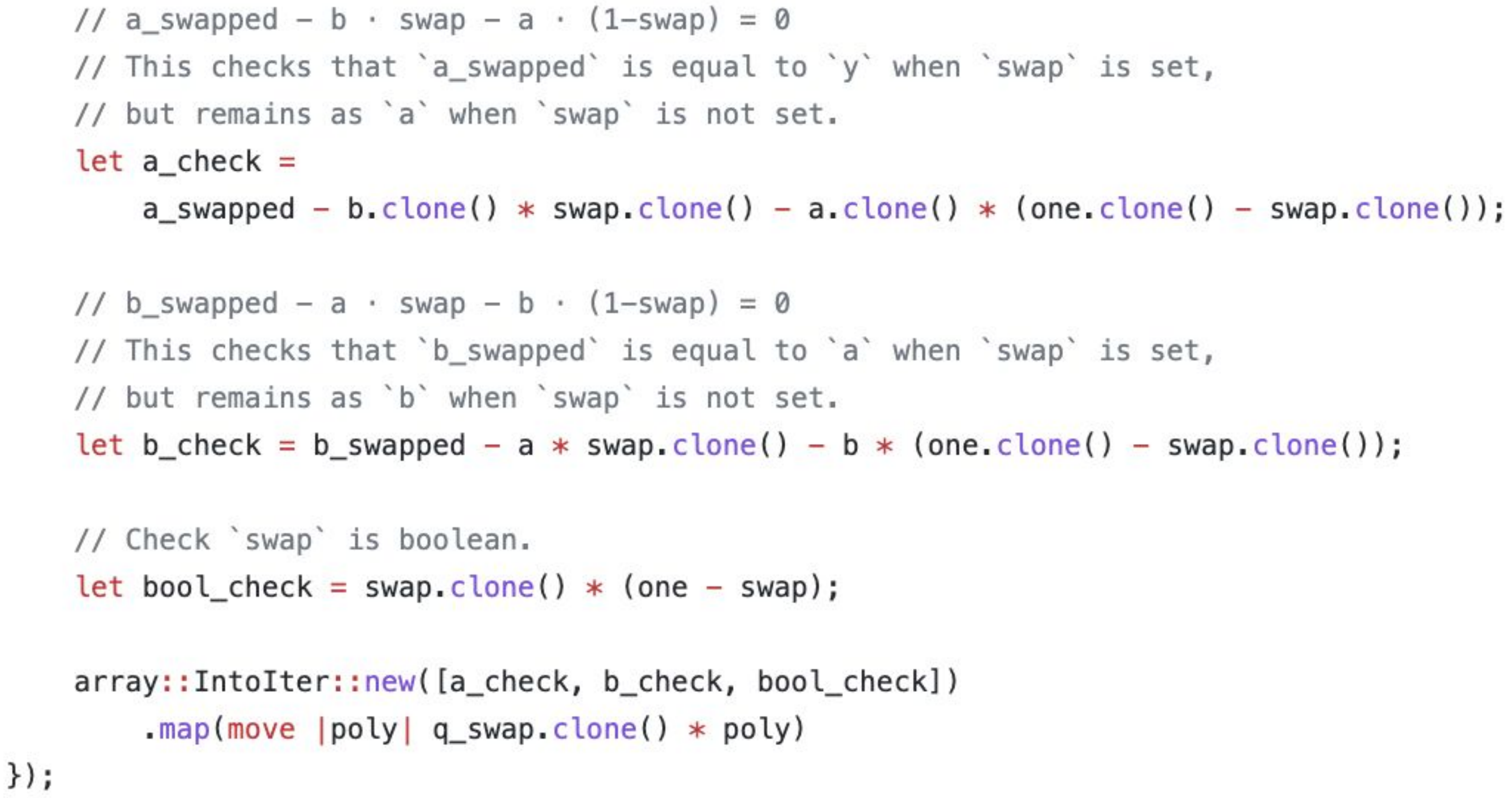
```rust
pub fn configure(
    meta: &mut ConstraintSystem<F>,
    advices: [Column<Advice>; 5],
) -> CondSwapConfig {
    // Only column a is used in an equality constraint directly by this chip.
    let q_swap = meta.selector();

    meta.enable_equality(advices[0].into());
    meta.enable_equality(advices[1].into());
    let config = CondSwapConfig {
        q_swap,
        a: advices[0],
        b: advices[1],
        a_swapped: advices[2],
        b_swapped: advices[3],
        swap: advices[4],
    };
```

CondSwapChip - configure - custom gate/1



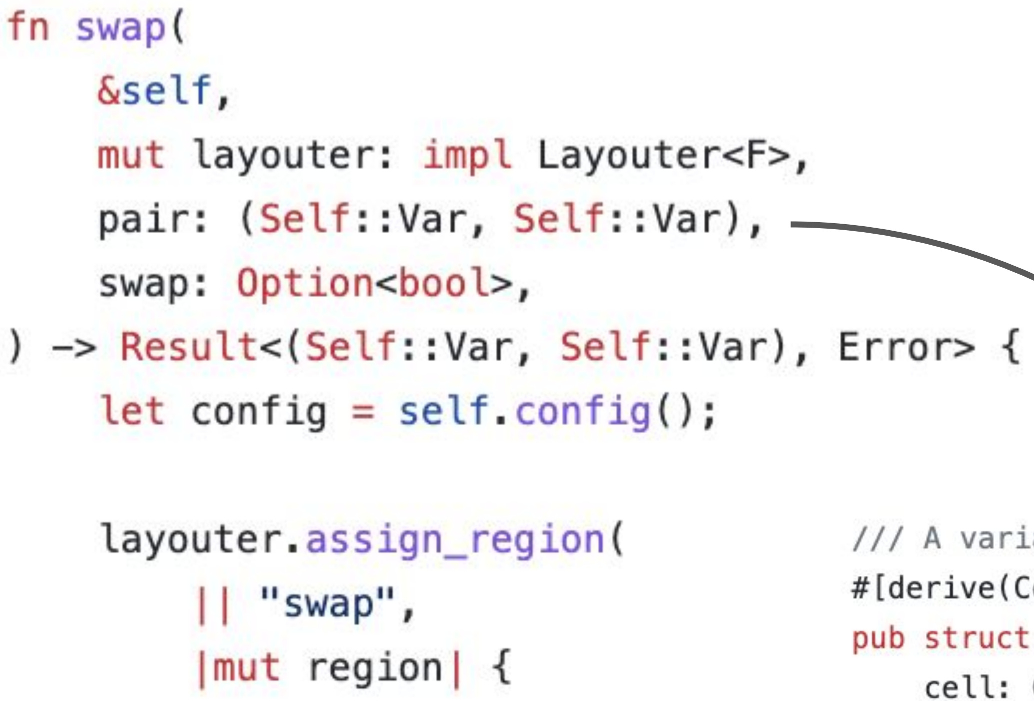
```
meta.create_gate("a' = b · swap + a · (1-swap)", |meta| {
    let q_swap = meta.query_selector(q_swap);

    let a = meta.query_advice(config.a, Rotation::cur());
    let b = meta.query_advice(config.b, Rotation::cur());

    let a_swapped = meta.query_advice(config.a_swapped, Rotation::cur());
    let b_swapped = meta.query_advice(config.b_swapped, Rotation::cur());
    let swap = meta.query_advice(config.swap, Rotation::cur());
```

CondSwapChip - configure - custom gate/2

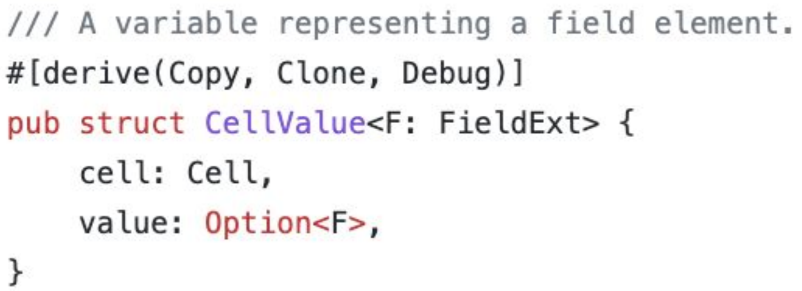


```
        // a_swapped - b · swap - a · (1-swap) = 0
        // This checks that `a_swapped` is equal to `y` when `swap` is set,
        // but remains as `a` when `swap` is not set.
        let a_check =
            a_swapped - b.clone() * swap.clone() - a.clone() * (one.clone() - swap.clone());

        // b_swapped - a · swap - b · (1-swap) = 0
        // This checks that `b_swapped` is equal to `a` when `swap` is set,
        // but remains as `b` when `swap` is not set.
        let b_check = b_swapped - a * swap.clone() - b * (one.clone() - swap.clone());

        // Check `swap` is boolean.
        let bool_check = swap.clone() * (one - swap);

        array::IntoIter::new([a_check, b_check, bool_check])
            .map(move |poly| q_swap.clone() * poly)
    });
```

CondSwapChip - swap instruction

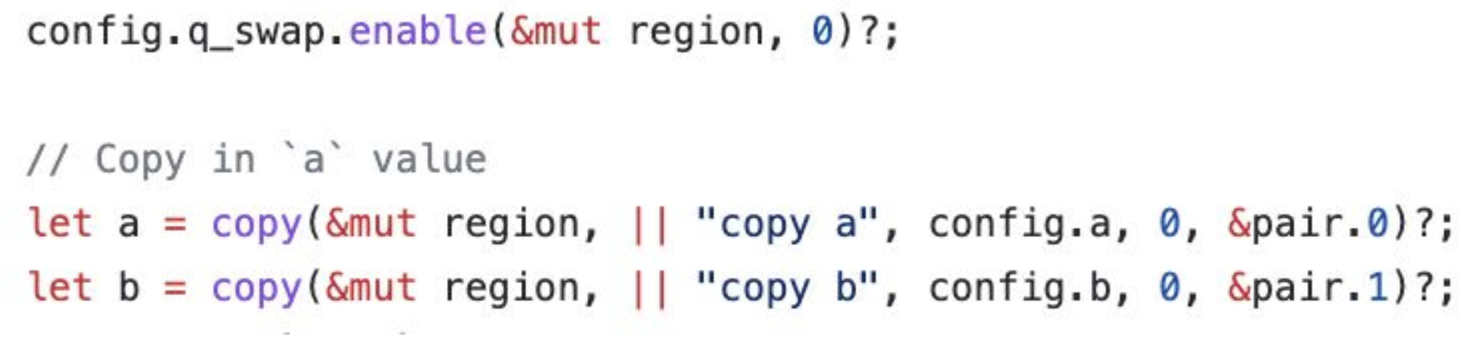


```rust
fn swap(
    &self,
    mut layouter: impl Layouter<F>,
    pair: (Self::Var, Self::Var),
    swap: Option<bool>,
) -> Result<(Self::Var, Self::Var), Error> {
    let config = self.config();

    layouter.assign_region(
        || "swap",
        |mut region| {
```



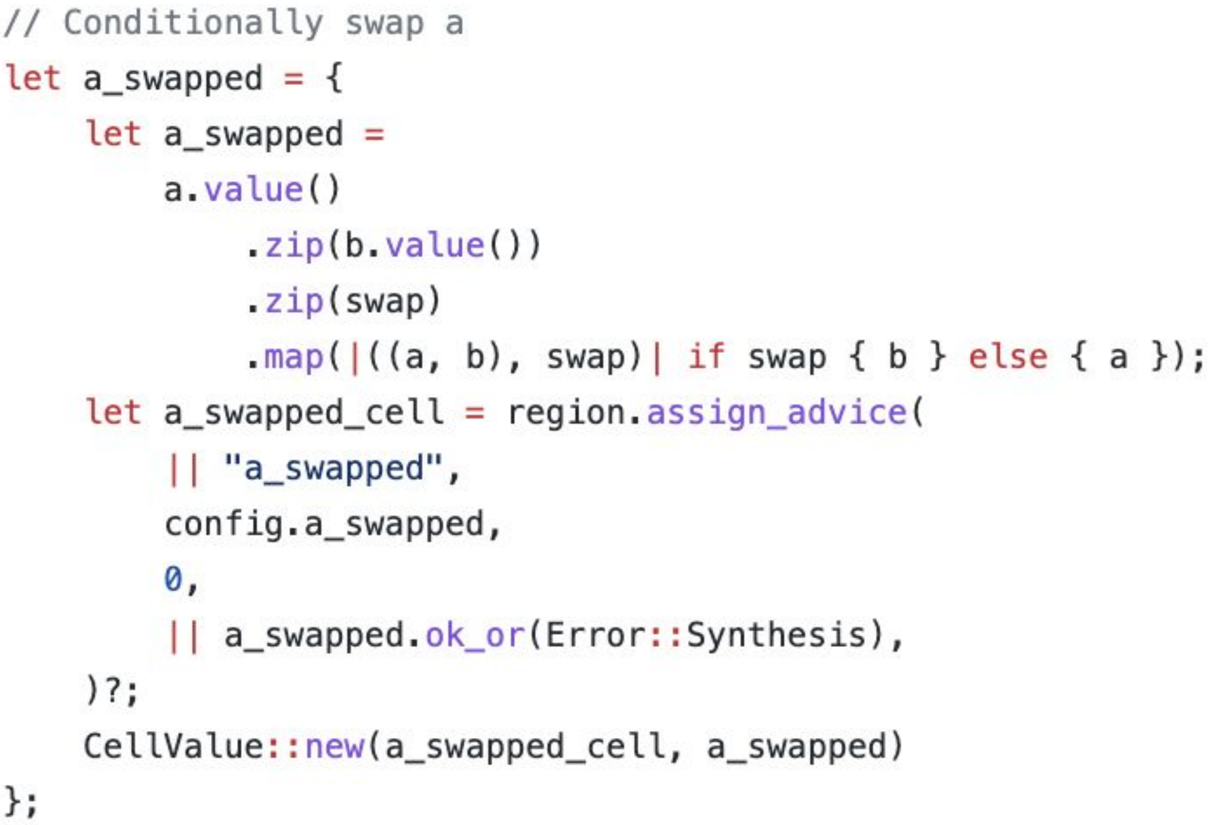
```rust
/// A variable representing a field element.
#[derive(Copy, Clone, Debug)]
pub struct CellValue<F: FieldExt> {
    cell: Cell,
    value: Option<F>,
}
```

CondSwapChip - swap instruction



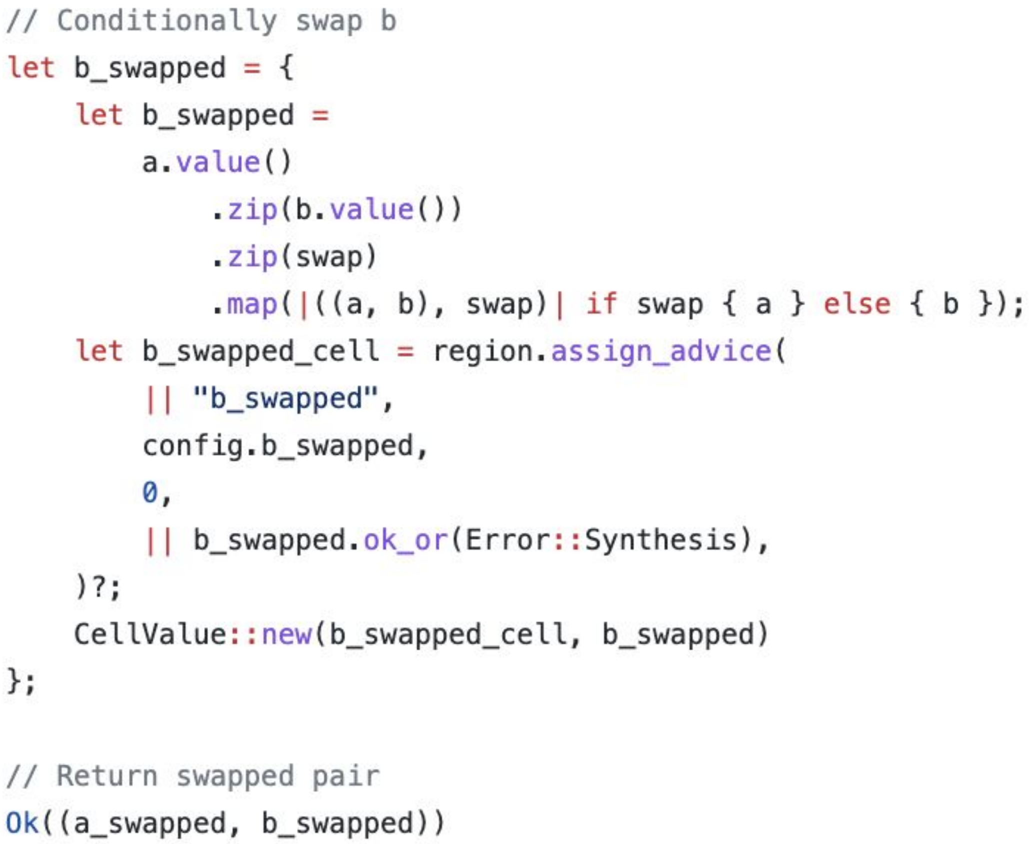
```
config.q_swap.enable(&mut region, 0)?;

// Copy in `a` value
let a = copy(&mut region, || "copy a", config.a, 0, &pair.0)?;
let b = copy(&mut region, || "copy b", config.b, 0, &pair.1)?;
```

CondSwapChip - swap instruction



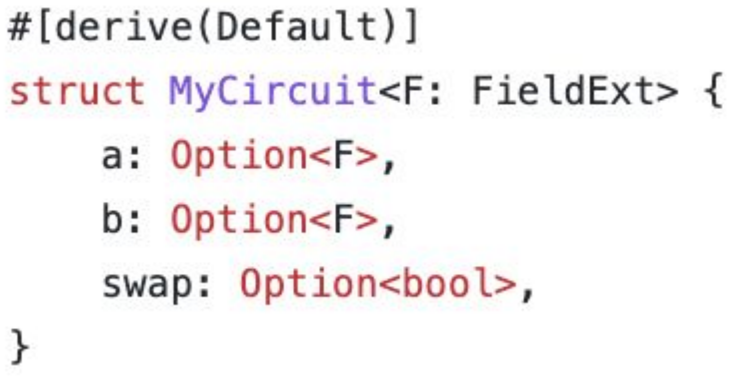
```
// Conditionally swap a
let a_swapped = {
    let a_swapped =
        a.value()
            .zip(b.value())
            .zip(swap)
            .map(|((a, b), swap)| if swap { b } else { a });
    let a_swapped_cell = region.assign_advice(
        || "a_swapped",
        config.a_swapped,
        0,
        || a_swapped.ok_or(Error::Synthesis),
    )?;
    CellValue::new(a_swapped_cell, a_swapped)
};
```

CondSwapChip - swap instruction



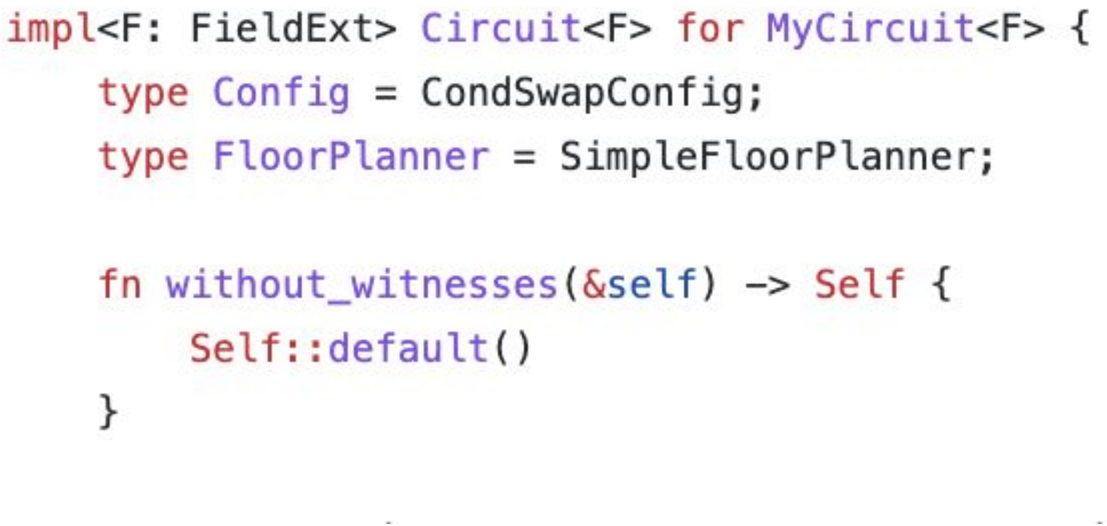
```
// Conditionally swap b
let b_swapped = {
    let b_swapped =
        a.value()
            .zip(b.value())
            .zip(swap)
            .map(|((a, b), swap)| if swap { a } else { b });
    let b_swapped_cell = region.assign_advice(
        || "b_swapped",
        config.b_swapped,
        0,
        || b_swapped.ok_or(Error::Synthesis),
    )?;
    CellValue::new(b_swapped_cell, b_swapped)
};

// Return swapped pair
Ok((a_swapped, b_swapped))
```

CondSwapChip - circuit - struct



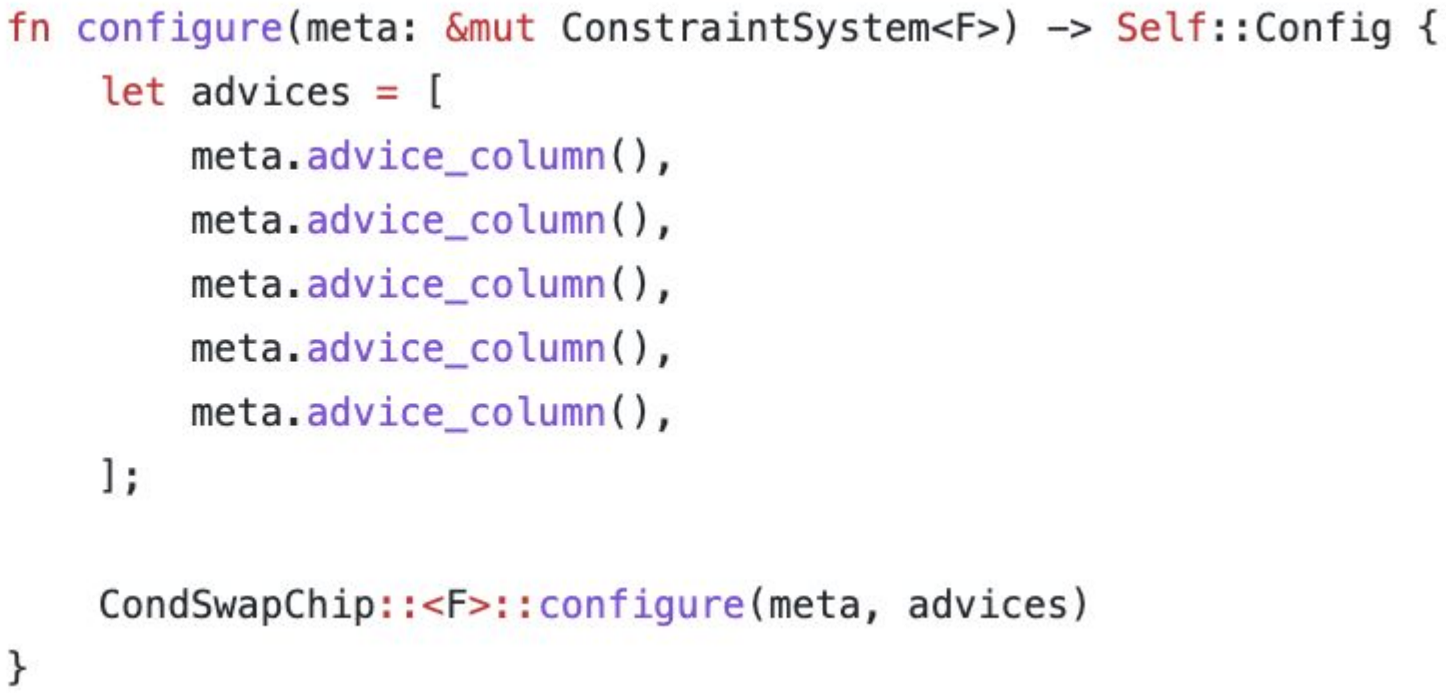
```
#[derive(Default)]
struct MyCircuit<F: FieldExt> {
    a: Option<F>,
    b: Option<F>,
    swap: Option<bool>,
}
```

CondSwapChip - circuit - Circuit<F> impl



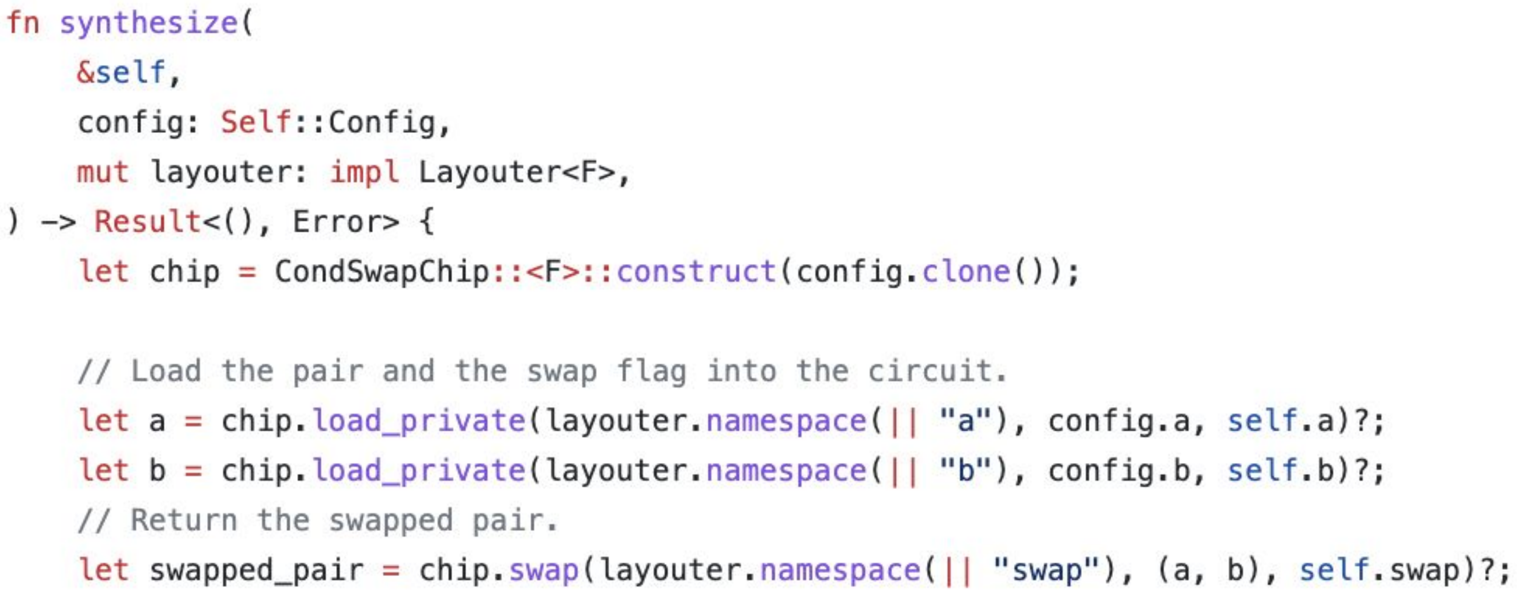
```
impl<F: FieldExt> Circuit<F> for MyCircuit<F> {
    type Config = CondSwapConfig;
    type FloorPlanner = SimpleFloorPlanner;

    fn without_witnesses(&self) -> Self {
        Self::default()
    }
```

CondSwapChip - circuit - Circuit<F> impl



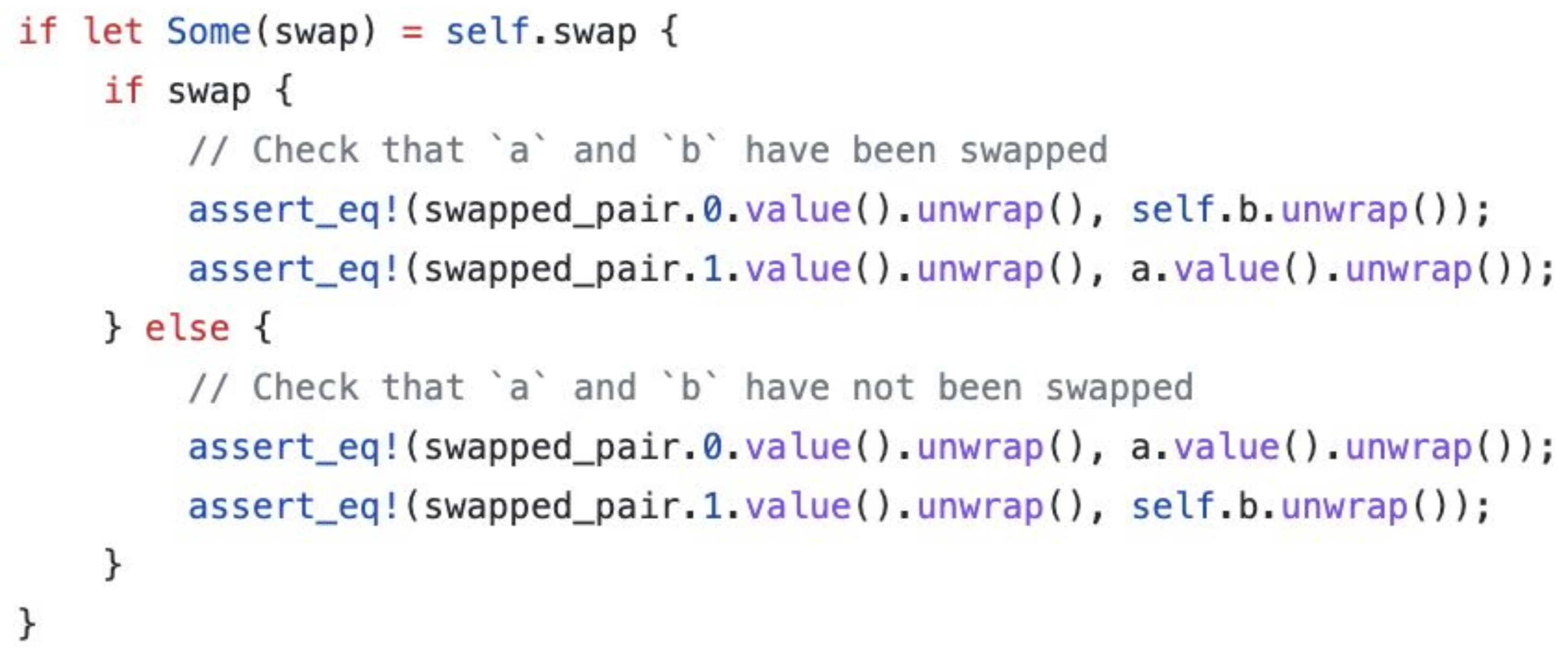
```
fn configure(meta: &mut ConstraintSystem<F>) -> Self::Config {
    let advices = [
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
        meta.advice_column(),
    ];

    CondSwapChip::<F>::configure(meta, advices)
}
```

CondSwapChip - circuit - synthetize



```
fn synthesize(
    &self,
    config: Self::Config,
    mut layouter: impl Layouter<F>,
) -> Result<(), Error> {
    let chip = CondSwapChip::<F>::construct(config.clone());

    // Load the pair and the swap flag into the circuit.
    let a = chip.load_private(layouter.namespace(|| "a"), config.a, self.a)?;
    let b = chip.load_private(layouter.namespace(|| "b"), config.b, self.b)?;
    // Return the swapped pair.
    let swapped_pair = chip.swap(layouter.namespace(|| "swap"), (a, b), self.swap)?;
```

CondSwapChip - circuit - synthetize



```
if let Some(swap) = self.swap {
    if swap {
        // Check that `a` and `b` have been swapped
        assert_eq!(swapped_pair.0.value().unwrap(), self.b.unwrap());
        assert_eq!(swapped_pair.1.value().unwrap(), a.value().unwrap());
    } else {
        // Check that `a` and `b` have not been swapped
        assert_eq!(swapped_pair.0.value().unwrap(), a.value().unwrap());
        assert_eq!(swapped_pair.1.value().unwrap(), self.b.unwrap());
    }
}
```

Another synthetize

https://github.com/adria0/halo2-franchise-proof/blob/58d82a9a27d6b92c9ff8fa2e429b5de765c601f4/src/franchise.rs#L192

Prove / verify

https://github.com/adria0/halo2-franchise-proof/blob/main/benches/franchise.rs

twitter adria0

@ethdevbcn

thanks!