Clap 👏
safe & fast circuits

or CLAP: a Semantic-Preserving Optimizing eDSL for Plonkish Proof Systems

Marco Stronati with Denis Firsov, Antonio Locascio, Ben Livshits
developed at Matter Labs



https://arxiv.org/abs/2405.12115

Critical systems and security bugs

Application: video game

Bugs: fewer the better

Trigger: random

Testing

Move fast and break things



Application: wallet

Bugs: None

Trigger: intentional

Formal Verification

Move faster w/o breaking anything



Bad fame of Formal Methods

Developers:

- need to ship yesterday
- FM are slow and expensive

Proof engineer:

- After-the-fact (untangle spaghetti)
- End-to-end properties

Proof engineers should make tools not systems

Formally verifiABLE

A design that could be formally verified, by a kind soul, one day.

Recipe:

1. Break down the problem
2. Define your properties
3. Sketch how to prove them

Proof engineers should be part of the design

Constraint systems & witness generators

Proof systems:

- CS
- Witness
- Sat: cs → witness → bool

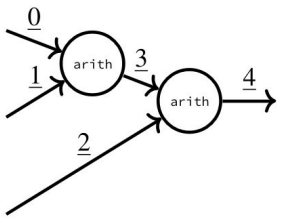
Performance goals:

- CS - small
- Witness generator - fast

Why we talk about circuits?

```
fn chained_add<F:Field>(
  e : &mut Env<F>,
  i1 : Repr<F, SValue>,
  i2 : Repr<F, SValue>,
  i3 : Repr<F, SValue>)
  -> Repr<F, SValue> {
  let m = e.mul(i0, i1);
  e.add(m, i3)
}
```

(a) Program



(b) Circuit

$$0 * t[\underline{0}] + 0 * t[\underline{1}] - 1 * t[\underline{3}] + 1 * t[\underline{0}] * t[\underline{1}] + 0 = 0$$

$$1 * t[\underline{2}] + 1 * t[\underline{3}] - 1 * t[\underline{4}] + 0 * t[\underline{0}] * t[\underline{1}] + 0 = 0$$

(c) Constraint system.

l	r	o	ql	qr	qo	qm	qc
0	1	3	0	0	-1	1	0
2	3	4	1	1	-1	0	0

(d) Table.

Circuit =? CS

- CS are relations
- Circuits are (partial) functions
- Circuits = witgen

Determinism is the goal

Properties

- Over-constrained / not complete

  witgen $\supset$ sat

  problem for an honest prover
- Under-constrained / not sound

  witgen $\subset$ sat

  attack for a dishonest prover

Testing:

- very hard for over-constrained
- impossible for under-constrained

Verification approach:

- given a CS (NOT the witgen)
- Re-write a specification
- Prove equivalence

Our approach: *Correct by construction*

Oracles are not CS!

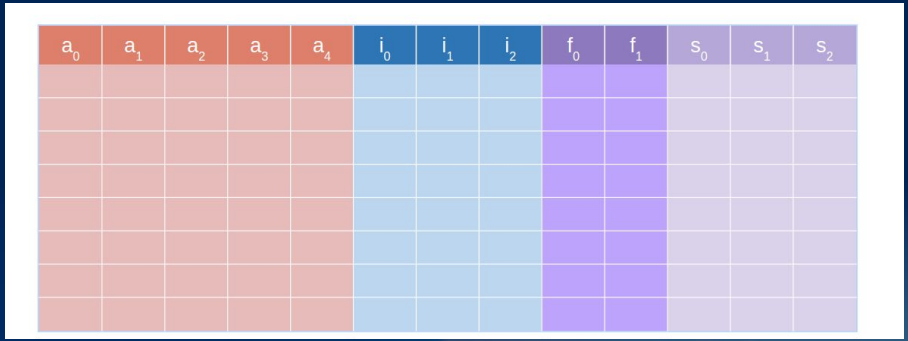
Circuits can compute anything that can be ***checked*** by a CS.

Examples:

- Inversion
- Limb decomposition

Boojum (Halo2, Plonky2, Bellman)

- forget circuits
- you can write small correct gates
  (witgen + cs)
- compose by hand



safe inefficient gates vs dangerous optimized gates

Duplicate checks

c = { b0 = isZero x ;
b1 = bNot b0 }

isZero : field → bool |cs|=2
bNot : field → bool |cs|=2

c′ = { b0 = isZero x ;
b1 = bNot_unsafe b0 }

bNot_unsafe : bool → bool |cs|=1

Much worse in limb decomposition!

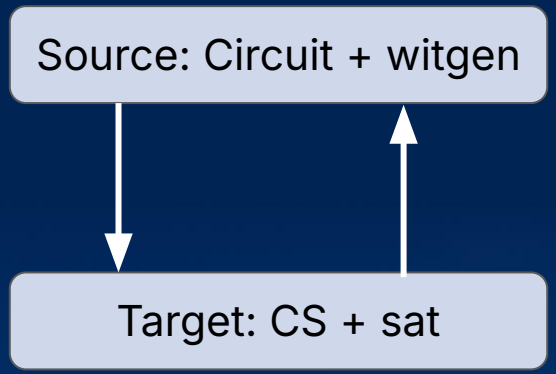
Clap



Intuition 1: sem-pre compilation

Completeness

(Not over-constrained)



Soundness

(Not under-constrained)

Definitions are hard already!

Intuition 2: write safe&simple, optimize later

- Developer has only access to sound and complete gates
- Optimizer reduces the size
  - whole circuit
  - access to unsound gates
- No manual optimization

Experimental results

	Constant	FMA	Linear Combination (size 4)	Total
Boojum non-custom-gate	5	854	304	1,163
Clap non-optimized	119	1756	0	1,875
Clap optimized	119	423	483	1,025

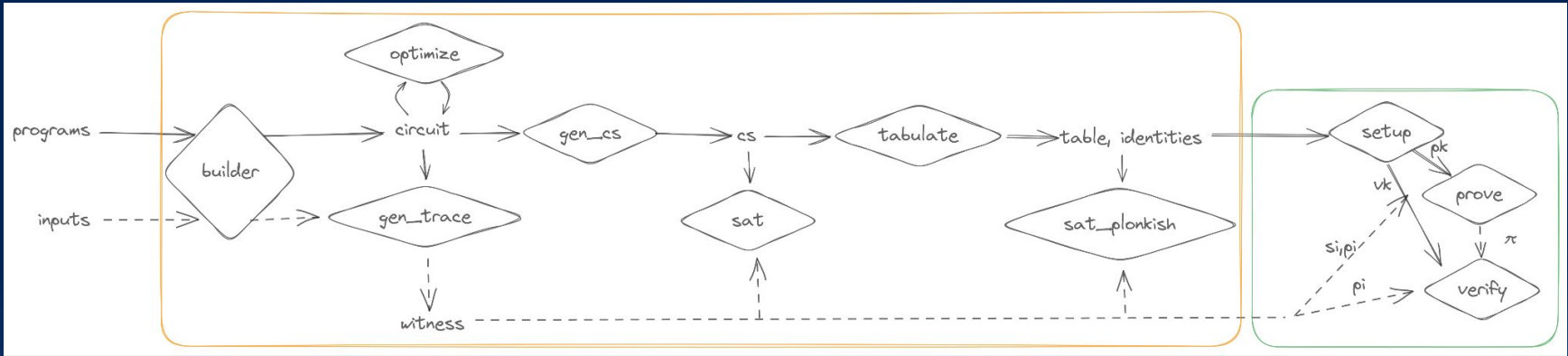
Fig. 7. Circuit sizes (per gate) for Poseidon2 circuit implementations.

	Constant	FMA	Linear Combination (size 4)	Lookup	Total
Boojum	76	1,268	3,617	3,885	8,845
Clap non-optimized	75	2,434	4,832	6,802	14,143
Clap optimized	75	998	3,251	3,602	7,926

Fig. 8. Circuit sizes (per gate) for (one round of) SHA-256 circuit implementations.

safer, slightly smaller cs, more readable, reusable, verifiAble

Clap architecture

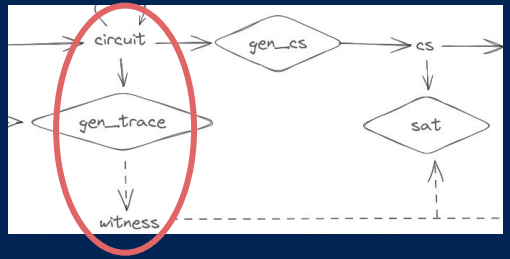


- simple untrusted builder from Rust to Circuit
- optimizer (reduce size, convert gates)
- gen_trace for witness generation
- gen_cs goes to CVs, not yet table
- tabulate: have uniform CVs (selectors aggregation, random challenge composition)

Semantics-preserving compilation

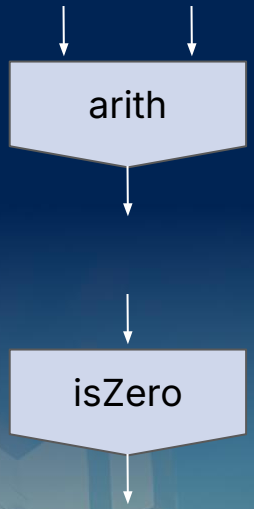


Source: circuit



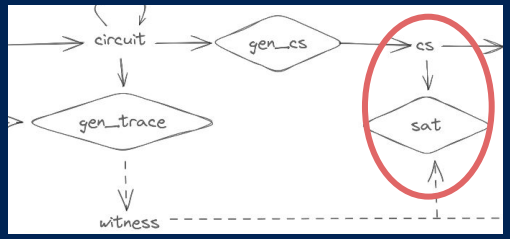

```
circuit ::=
  | nil
  | gate(gate)
  | seq(circuit, circuit)
  | par(circuit, circuit)
```

```
gate ::=
  | isZero (i) (o)
  | arith (l r) (o) (ql qr qo qm qc)
```



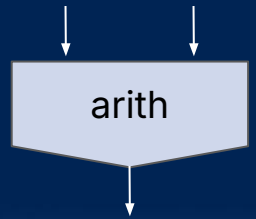
gen_trace : circuit → trace → option trace

Target: constrained vectors

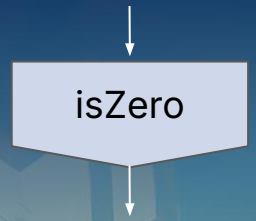


wires/variables constants identities

(w, q, id)

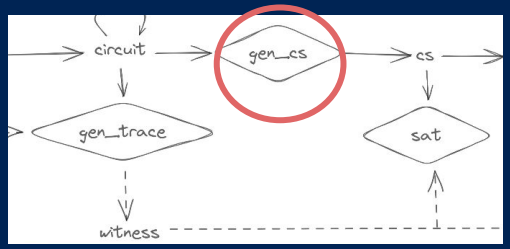


sat : cs → trace → bool

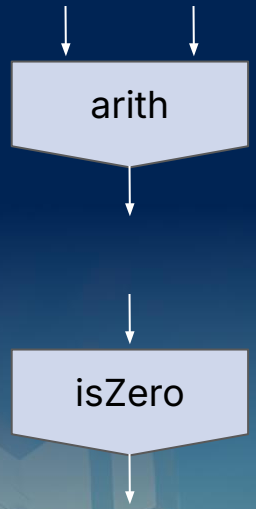



```
arith_cv    := ((wl wr wo), (ql qr qo qm qc), id_arith)
isZero_cv   := ((wi wr wo), (0 0 -1 -1 1), (id_arith id_bool))
```

Compilation



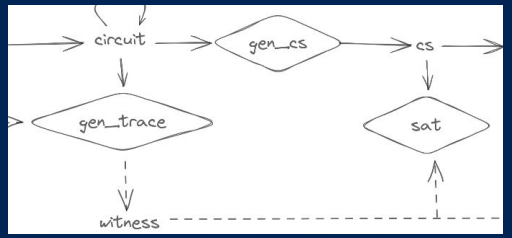
gen_cs : circuit → cs




```
arith_cv        := ((wl wr wo), (ql qr qo qm qc), id_arith)
isZero_cv       := ((wi wr wo), (0 0 -1 -1 1), (id_arith id_bool))
```

Completeness

Not over-constrained



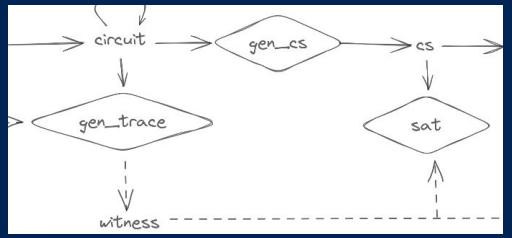
∀ circuit inputs witness.

gen_trace circuit inputs = Some witness

⇒ sat (gen_cs circuit) witness = true

Soundness

Not under-constrained

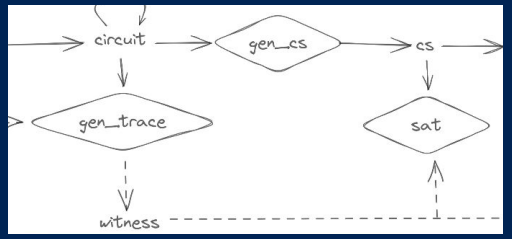


∀ circuit witness.

sat (gen_cs circuit) witness = true

⇒ gen_trace circuit (prefix witness) $\approx_c$ Some witness

Compositional approach



Sound and complete circuits
compose into
sound and complete circuits

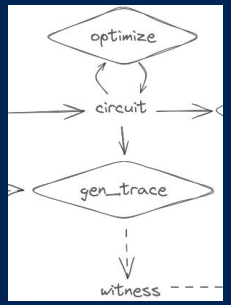
✅verifiED

Simplified case for gates:
- Gen_trace
- Gen_cs
- Identities
- Prove soundness & completeness

Optimizer



Sem-Pre Optimizations



∀ circuit inputs.

gen_trace (**opt** circuit) inputs

≈

gen_trace circuit inputs

Only for sound and complete gates!

Inlining

- Reduce intermediate variables
- Watch the degree!

- Reduce prematurely-optimized circuits
- Poseidon has nice linear segments

	Constant	FMA	Linear Combination (size 4)	Total
Boojum non-custom-gate	5	854	304	1,163
CLAP non-optimized	119	1756	0	1,875
CLAP optimized	119	423	483	1,025

Fig. 7. Circuit sizes (per gate) for Poseidon2 circuit implementations.

```
fn xor(&mut self,
       l: Repr<F, bool>,
       r: Repr<F, bool>) -> Repr<F, bool> {
  // (l /\ ~r) \/ (~l /\ r)
  let nr = self.not(r);
  let lnr = self.and(l, nr);
  let nl = self.not(l);
  let nlr = self.and(nl, r);
  self.or(lnr, nlr)
}
```

(a) CLAP version.

```
pub fn xor(self, cs: &mut CS, other: Self) ->
    Self {
    // c = ~(a /\ b) /\ ~(~a /\ ~b)
    // =(1 - (a * b)) * (1 - ((1 - a) * (1 - b)))
    // =(1 - ab) * (1 - (1 - a - b + ab))
    // =(1 - ab) * (a + b - ab)
    // =a + b - ab - (a^2)b - (b^2)a + (a^2)(b^2)
    // =a + b - ab - ab - ab + ab
    // =a + b - 2ab
    // -2a * b = c - a - b
    // 2a * b = a + b - c
    // 2a*b - a - b + c = 0
    let a_plus_b = compute_fma(cs, ...);
    let c = compute_fma(cs,...);
    Boolean::from_variable_checked(cs, c)
}
```

(b) Boojum version.

Remove duplicate checks

For soundness, each gate introduces checks.
Sometimes over the same variables!

We allow ***only*** the optimizer to introduce locally-unsound gates.

```
c = { x1 = isZero x0 ;
      x2 = bNot x1}

c′ = { x1 = isZero x0 ;
       x2 = bNot_unsafe x1}
```

Limb decomposition

	Constant	FMA	Linear Combination (size 4)	Lookup	Total
Boojum	76	1,268	3,617	3,885	8,845
CLAP non-optimized	75	2,434	4,832	6,802	14,143
CLAP optimized	75	998	3,251	3,602	7,926

Fig. 8. Circuit sizes (per gate) for (one round of) SHA-256 circuit implementations.

Remove duplicate checks

```rust
pub fn round_function<F: SmallField>(
    cs: &mut Env<F>, state: &mut State<F>,
    message_block: &[Repr<F, u32>; 16],
    range_check_final_state: bool,
) -> Option<[Repr<F, U4>; 64]> {
```

(a) CLAP version.

```rust
// we only have variables in the signature:
// - if it's a first round then state is made of
    constants
// - if it's not the last round then next round
    will range check when rounds happen
// - if it's the last round then it's done by
    caller
pub fn round_function<F: SmallField>(
    cs: &mut CS, state: &mut [Variable; 8],
    message_block: &[Variable; 16],
    range_check_final_state: bool,
) -> Option<[Variable; 64]> {
```

(b) Boojum version.

```rust
for idx in 16..SHA256_ROUNDS {
  ...
  let (u32_part, _) = range_check_36(cs,
    expanded_word);
  expanded[idx] = u32_part;
}
```

(a) CLAP version.

```rust
// we assume inputs to be range checked, so
// idx - 15 is range checked by xor
// idx - 2 is range-checked by xor
// so we are "safe" as long as our index over
    which we work right now will become
// idx - 2 for some next round
for idx in 16..SHA256_ROUNDS {
  ...
  // we only need to fully range check if it's
    one of the last
  let u32_part = if idx + 2 >= 64 {
    let (u32_part, _) = range_check_36(cs,
    expanded_word);
    u32_part
    } else {
      let (u32_part, high_unchecked) =
    split_36_bits_unchecked(cs, expanded_word);
      yet_unconstrained_chunks.push(
    high_unchecked);
      u32_part
  };
  expanded[idx] = u32_part;
}
// range check small pieces
...
```

(b) Boojum version.

Gate conversion

Watch the field!

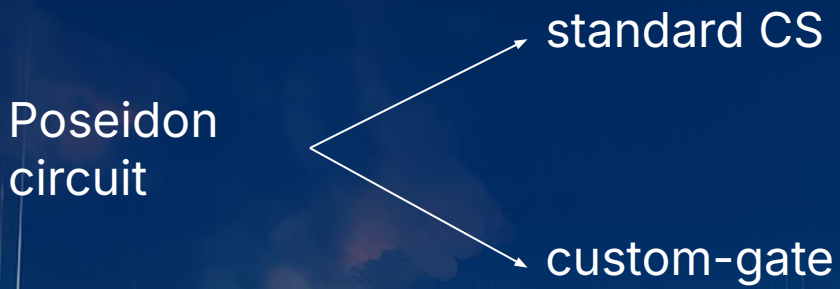


wider/narrower tables ⇒ faster proving/verification

Custom gate generation

- custom gates are the defining characteristic of PlonKish
- they can be huge e.g. Poseidon round
- trade-off cost / usage

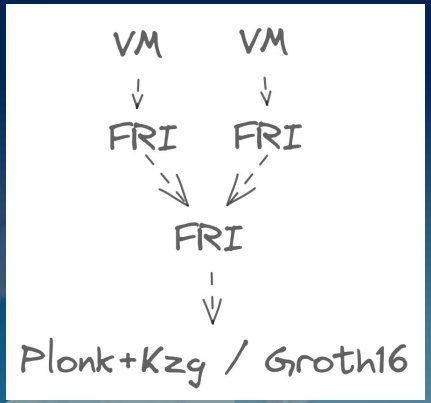
- inliner can take a normal circuit and inline it fully (with a degree limit)



Relation to W and future work

- Highly recursive systems: after on-chain verifier there are multiple circuit verifiers
- Embed in Agda/Coq/Lean: functional correctness

- Summer internship: embed builder in Agda
- Target Plonky3 for Air circuits?



Thanks