Automated Analysis of Halo2 Circuits

Fatemeh Heidari Soureshjani	Quantstamp & Polytechnique Montreal, Canada
Mathias Hall-Andersen	Aarhus University, Denmark
Mohammad Mahdi Jahanara	Quantstamp
Jeffrey Kam	Quantstamp
Jan Gorzny	**Quantstamp**
Mohsen Ahmadvand	Quantstamp

21st International Workshop on Satisfiability Modulo Theories, Italy

Plan

1. **Introduction**: Zero-Knowledge Proofs, Halo2, and Related Work
2. **Abstract Interpretation Approach**: Introduction & Use
3. **SMT Approach**: Use
4. **Conclusion**: Summary & Future Work

Plan

1. **Introduction**: Zero-Knowledge Proofs, Halo2, and Related Work
2. **Abstract Interpretation Approach**: Introduction & Use
3. **SMT Approach**: Use
4. **Conclusion**: Summary & Future Work

Zero-Knowledge Proofs

Introduced by Goldwasser et al. 1989

Prove that you know something without revealing it.

“For function *f* and public input *x*, I know a private witness *w* such that *f(x,w)* = *y*”

zk-SNARK

Zero-Knowledge — Succinct — Non-interactive — Argument of Knowledge

ZK DSLs & PLONK

- Some ZK DSLs and frameworks exist
  - Circom (Bellés-Muñoz et al. 2022)
  - ZoKrates (Eberhardt and Tai 2018)
  - **Halo2** (ZCash; no paper yet?)

- Under the hood, they typically compile to one of the following constraint systems:
  - Rank 1 Constraint System (R1CS)
    - Groth16 (Groth 2016)

  - **PLONKish arithmetic**
    - PLONK (Gabizon and Williamson 2019)
    - TurboPLONK (Gabizon and Williamson 2019)
    - plookup (Gabizon and Williamson 2020)
    - UltraPLONK (Aztec 2021; no paper yet?)
    - HyperPLONK (Chen et al. 2022)

Halo2

- Popular zero-knowledge proof system library in **Rust**
- Uses **PLONKish arithmetization** to express circuits: circuits are tables, and we add constraints over the table

	Verifier + Prover (input)	*Prover (witness)*	*Verifier + Prover (constant)*
	Selector (s)	**Instance (b)**	**Advice (a)**	**Fixed (C)**

	Selector (s)	Instance (b)	Advice (a)	Fixed (C)
		Verifier + Prover (input)	*Prover (witness)*	*Verifier + Prover (constant)*
row *i*	1	15	5	83

+ Constraints:

$$s_i\,(C_i \cdot b_i - a_i) = 0$$

$$f(\langle a,b,C\rangle, s) = s\,(C \cdot a - b)$$

Shaded area is a **region** and a **gate** (entire row in this example)

Vanishing Polynomials

	Verifier + Prover (input)	*Prover (witness)*	*Verifier + Prover (constant)*
Selector (s)	**Instance (b)**	**Advice (a)**	**Fixed (C)**

	Selector (s)	Instance (b)	Advice (a)	Fixed (C)
row *i*	1	15	5	83

+ Constraints:

$$s_i (C_i \cdot b_i - a_i) = 0$$

A polynomial **vanishes** if it evaluates to 0 over all rows. *All* polynomial constraints in a Halo2 proof system should vanish over all rows for a valid witness and public input pair.

Do this by either

- setting the irrelevant selector variables to 0, or
- providing (possibly secret) assignments to the table cells that result in the polynomial's evaluation to 0.

ZK Problems

ZK Bug Tracker https://github.com/0xPARC/zk-bug-tracker

A community-maintained collection of bugs, vulnerabilities, and exploits in apps using ZK crypto.

Bugs in the Wild

1. Dark Forest v0.3: Missing Bit Length Check
2. BigInt: Missing Bit Length Check
3. Circom-Pairing: Missing Output Check Constraint
4. Semaphore: Missing Smart Contract Range Check
5. Zk-Kit: Missing Smart Contract Range Check
6. Aztec 2.0: Missing Bit Length Check / Nondeterministic Nullifier
7. 0xPARC StealthDrop: Nondeterministic Nullifier
8. MACI 1.0: Under-constrained Circuit
9. Bulletproofs Paper: Frozen Heart
10. PlonK: Frozen Heart
11. Zcash: Trusted Setup Leak
12. MiMC Hash: Assigned but not Constrained
13. PSE & Scroll zkEVM: Missing Overflow Constraint
14. PSE & Scroll zkEVM: Missing Constraint

Common Vulnerabilities

1. Under-constrained Circuits
2. Nondeterministic Circuits
3. Arithmetic Over/Under Flows
4. Mismatching Bit Lengths
5. Unused Public Inputs Optimized Out
6. Frozen Heart: Forging of Zero Knowledge Proofs
7. Trusted Setup Leak
8. Assigned but not Constrained

This Work

- We describe **a Proof-of-Concept / Work-In-Progress tool for analysis of Halo2 circuits in Rust**

- Analyses for the following issues:
  - **Underconstrained circuits**
    - Assigned but unconstrained cells (abstract interpretation)
    - Multiple assignments to witnesses for a public input (SMT)
  - **Unused custom gates** (abstract interpretation)
  - **Unused columns** (abstract interpretation)

Download it here!

https://github.com/quantstamp/halo2-analyzer

Related Work

- Picus (https://github.com/chyanju/Picus)
  - Uses symbolic execution
  - Supports custom queries / property checking
  - Automated verification

… but for **R1CS**

- Ecne (https://github.com/franklynwang/EcneProject)
  - Fixed-point algorithm
  - Needs rules to be specified

… but *also* for **R1CS**

- QED^2 (Pailoor et al., 2023)
  - SMT-based approach
  - “uniqueness inference”

… but for Circom (**R1CS**)

Picus

Picus is a symbolic virtual machine for automated verification tasks on R1CS.

Ecne (R1CSConstraintSolver.jl)

Introduction

zk-SNARKs are a method for generating zero-knowledge proofs of arbitrary functions, as long as these functions can be expressed as the result of a R1CS (a rank-one constraint system). However, one still needs to convert functions into R1CS form. As this is a laborious process (though still far easier than starting from scratch), Ecne,

Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs

Authors: Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, Işıl Dillig Authors Info & Claims

Plan

1. **Introduction:** Zero-Knowledge Proofs, Halo2, and Related Work
2. **Abstract Interpretation Approach:** Introduction & Use
3. **SMT Approach:** Use
4. **Conclusion:** Summary & Future Work

Abstract Interpretation & Halo2

Uses **Abstract Interpretation** (Cousot & Cousot, 1976)

Approximation of programs via "**partial execution**": some calculations are performed, but others are not.

For Halo2: partially execute the polynomials, using abstract values.

- Try to determine if some polynomials are always non-zero; then they would not vanish!

Abstract Interpretation & Halo2

Create a new enum that represents a polynomial's value which is either:

- Something (probably depending on the witness)
- Definitely not zero (for any witness)
- Definitely zero (for any witness)

Then "partially execute": add, multiply, subtract values and get some inference (e.g. 0+0=0). Example of adding values below.

```
Expression::Sum(left, right) => {
    let res1 = eval_abstract(left, selectors);
    let res2 = eval_abstract(right, selectors);
    match (res1, res2) {
        (AbsResult::Variable, _) => AbsResult::Variable, // could be anything
        (_, AbsResult::Variable) => AbsResult::Variable, // could be anything
        (AbsResult::NonZero, AbsResult::NonZero) => AbsResult::Variable, // could be zero or non-zero
        (AbsResult::Zero, AbsResult::Zero) => AbsResult::Zero,
        (AbsResult::Zero, AbsResult::NonZero) => AbsResult::NonZero,
        (AbsResult::NonZero, AbsResult::Zero) => AbsResult::NonZero,
    }
}
```

Abstract Interpretation & Halo2

No witness is provided; we can't evaluate the gate polynomials, but we can evaluate polynomials in regions for concrete values of selector variables and constant variables

So we can get checks for:

- **Unused Gates**: for every gate there exists a region in which it is not always zero
- **Unconstrained Cells**: for every assigned cell in the region, it occurs in a polynomial which is not identically zero over this region
- **Unused Column**: every column occurs in some polynomial

May yield false negatives: may return that a polynomial is not identically zero, when in fact it is

Plan

1. **Introduction:** Zero-Knowledge Proofs, Halo2, and Related Work
2. **Abstract Interpretation Approach:** Introduction & Use
3. **SMT Approach**: Use
4. **Conclusion**: Summary & Future Work

Under-Constrained Circuits

A Plonkish circuit C is **under-constrained** if there exists an assignment x to Instance columns of C, and two set of assignments w and w' for its Advice columns, where both $\{x, w\}$ and $\{x, w'\}$ satisfy constraints of C.

Over-Constrained Circuits

A Plonkish circuit *C* is **over-constrained** if for some assignment *x* to instance columns of *C*, no assignments to the advice columns of *C* enable the system to have a solution, but the developer expects there to be one.

Example. Consider a circuit that states that for any positive integer *x* as input, there are two (distinct) advice columns entries that are positive integer and add up to *x*.

- for $x \geq 2$ ✅
- for $x = 1$ ❌

  it would not be meaningful to call the circuit over-constrained for this input value.

Halo2 to SMT

We convert from Rust to **SMTLIB** and add constraints as a **conjunction.**

- For **gate constraints**, we add a constraint that the polynomial is equal to zero

$$(\text{add}(x_{a,b,c}, y_{a',b',c'}) = 0)$$

- For **copy constraints**, we add a constraint that the variables are equal

$$(x_{a,b,c} = y_{a',b',c'})$$

- For **lookup constraints,** we add a constraint that a disjunction enforcing that a variable is equal to one of the legal values

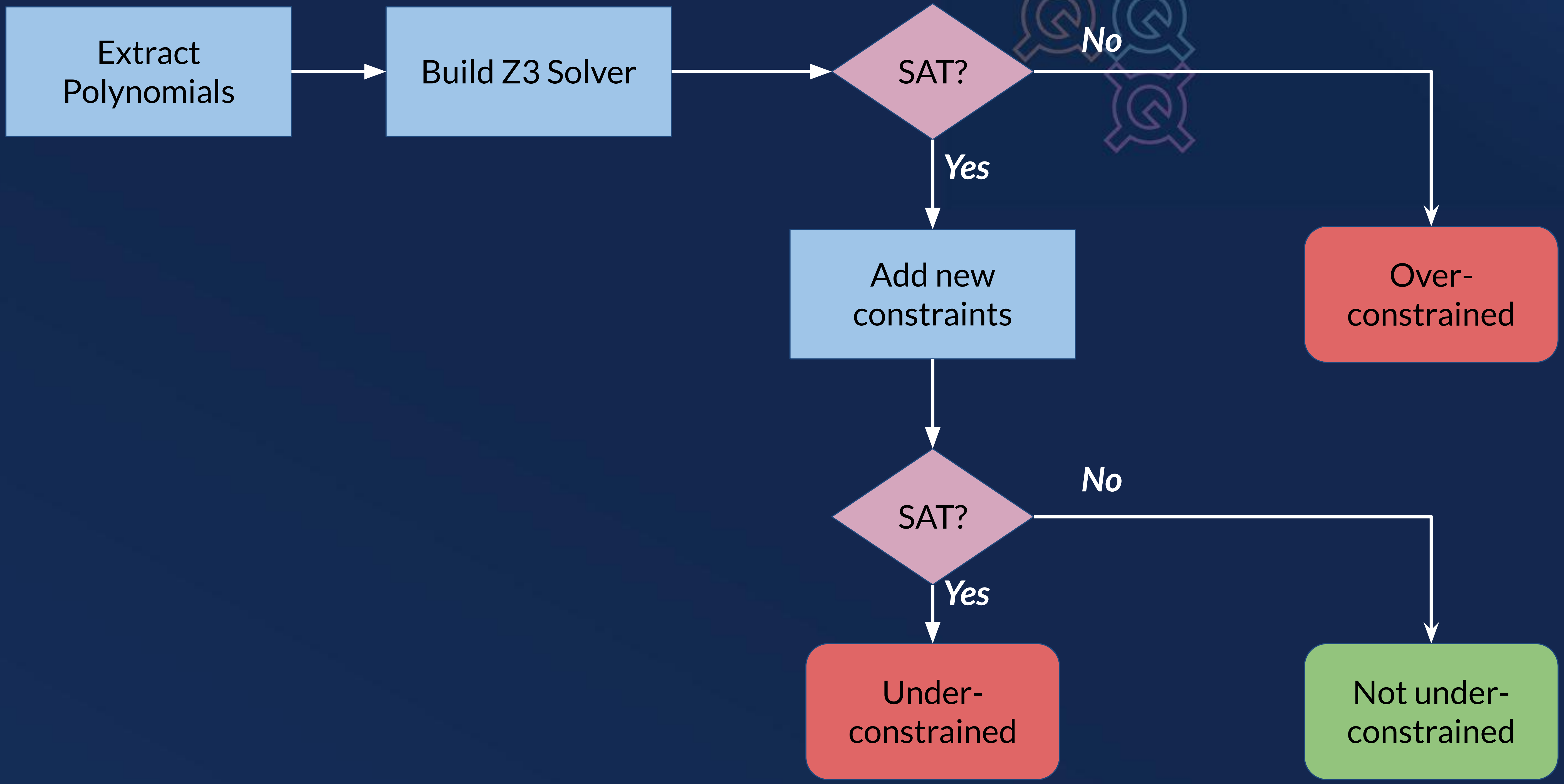
$$(x_{a,b,c} = v_1 \lor x_{a,b,c} = v_2 \lor \ldots \lor x_{a,b,c} = v_k)$$

Halo2 to SMT

We use **CVC5** (Barbosa et al. 2022) since there is a **finite field** solver for it (Ozdemir et al. 2023).

```
(set-logic QF_FF)
(declare-fun A-1-1-1 () (_ FiniteField 307))
(assert ( = A-1-1-1 (as ff0 (_ FiniteField 307)))
```

Analysis Logic



Under-Constrained Circuits Example

Motivating Example

Gate for x in [0,3]

Constraints:

Advice variables

$$b_0 \cdot (b_0 - 1) = 0$$

$$b_1 \cdot (b_1 - 1) = 0$$

Ensures binary values

$$b_0 + 2 \cdot b_1 = x$$

x is in the desired range

Instance Variable

Motivating Example


```
meta.create_gate("b1_binary_check", |meta| {
   let a = meta.query_advice(b1, Rotation::cur());
   let dummy = meta.query_selector(s);
   vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
   // a * (1-a)
});
```

Motivating Example


```
meta.create_gate("b1_binary_check", |meta| {
    let a = meta.query_advice(b1, Rotation::cur());
    let dummy = meta.query_selector(s);
    vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
    // a * (1-a)
});

meta.create_gate("b0_binary_check", |meta| {
    let a = meta.query_advice(b1, Rotation::cur());
    let dummy = meta.query_selector(s);
    vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
    // a * (1-a)
});
```

Motivating Example


```
meta.create_gate("b1_binary_check", |meta| {
   let a = meta.query_advice(b1, Rotation::cur());
   let dummy = meta.query_selector(s);
   vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
   // a * (1-a)
});

meta.create_gate("b0_binary_check", |meta| {
   let a = meta.query_advice(b1, Rotation::cur());
   let dummy = meta.query_selector(s);
   vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
   // a * (1-a)
});

meta.create_gate("equality", |meta| {
   let a = meta.query_advice(b0, Rotation::cur());
   let b = meta.query_advice(b1, Rotation::cur());
   let c = meta.query_advice(x, Rotation::cur());
            // we'll copy public instance here later using constrain_instance
   let dummy = meta.query_selector(s);
   vec![dummy * (a + Expression::Constant(Fr::from(2)) * b - c)]
});
```

Motivating Example Results


```
b0  -> 1
b1  -> 1
x   -> 3

equivalent model with same public input:
b0  -> 3
b1  -> 0
x   -> 3

Result:
The circuit is underConstrained.
```

- Takes < 1s to run on this example (no surprise)
- Push-button -- no additional property description necessary to write; but you *could* add more

Motivating Example


```
meta.create_gate("b1_binary_check", |meta| {
   let a = meta.query_advice(b1, Rotation::cur());
   let dummy = meta.query_selector(s);
   vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
   // a * (1-a)
});
```

Copy and paste error!

```
meta.create_gate("b0_binary_check", |meta| {
   let a = meta.query_advice(b1, Rotation::cur());
   let dummy = meta.query_selector(s);
   vec![dummy * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
   // a * (1-a)
});
```

```
meta.create_gate("equality", |meta| {
   let a = meta.query_advice(b0, Rotation::cur());
   let b = meta.query_advice(b1, Rotation::cur());
   let c = meta.query_advice(x, Rotation::cur());
         // we'll copy public instance here later using constrain_instance
   let dummy = meta.query_selector(s);
   vec![dummy * (a + Expression::Constant(Fr::from(2)) * b - c)]
});
```

Plan

1. **Introduction:** Zero-Knowledge Proofs, Halo2, and Related Work
2. **Abstract Interpretation Approach:** Introduction & Use
3. **SMT Approach:** Use
4. **Conclusion:** Summary & Future Work

Conclusion

- We have shown an approach to use abstract interpretation to find **assigned but unconstrained cells**, **unused custom gates**, and **unused columns** in Halo2
- We have shown how SMT solvers can be used to find **under-** and **over-constrained** Halo2 circuits

Download it here!

https://github.com/quantstamp/halo2-analyzer

Future Work

Future work is needed!

- **Limitations not yet known** - no readily available corpus of circuits to test scaling on; conversion, curation, or building necessary
- More analyses for **other** types of **bugs** and **issues** within Halo2 circuits; best practices?
- **Comparison** with, **combination** of, or **inspiration** from other approaches

k-bits	time (ms)	k-bits	time (ms)	k-bits	time (ms)
2	18.940541	16	71.075625	64	730.086875
4	28.063583	32	161.558958	128	4478.517416
8	36.888				

Table 4
Run times for the analysis in Section 3.2 on generalized circuits of Example 3.1.

https://github.com/quantstamp/halo2-analyzer

Thank you for listening!

@jgorzny

@jgorzny

jan@quantstamp.com

@quantstamp

Download it here!

https://github.com/quantstamp/halo2-analyzer