Automated Analysis of Halo2 Circuits

Fatemeh Heidari Soureshjani[1,2], Mathias Hall-Andersen[3], MohammadMahdi Jahanara[2], Jeffrey Kam[2], Jan Gorzny[2] and Mohsen Ahmadvand[2]

[1]*Computer Engineering and Software Engineering Department, Polytechnique Montréal, Montreal, Quebec, Canada*
[2]*Quantstamp, Inc., USA*
[3]*Department of Computer Science, Aarhus University, Denmark*

Abstract
Zero-knowledge proof systems are becoming increasingly prevalent and being widely used to secure decentralized financial systems and protect the privacy of users. Given the sensitivity of these applications, zero-knowledge proof systems are a natural target for formal verification methods. We describe methods for checking one such proof system: Halo2. We use abstract interpretation and an SMT solver to check various properties of Halo2 circuits. Using abstract interpretation, we can detect unused gates, unconstrained cells, and unused columns. Using an SMT solver, we can detect under-constrained circuits (in the sense that for the same public input they have two efficiently computable satisfying assignments). This is the first work we are aware of that applies lightweight formal methods to PLONKish arithmetization and Halo2 circuits.

Keywords
Zero-Knowledge Proof Systems, Halo2, Formal Methods, Abstract Interpretation, SMT

1. Introduction

A *zero-knowledge* proof system is a protocol that allows a *prover* to convince a *verifier* that certain statement is true without revealing any other information [1]. Recently, a specific type of these proof systems which are classed as a so-called *zk-SNARK* (defined below) have gained popularity in part due to their applications in scaling blockchain systems and building privacy-preserving applications.

This popularity comes from three desirable properties of zk-SNARK systems: that they are *zero-knowledge*, that their proofs are *succinct*, and that these protocols are *non-interactive*. A proof system is *zero-knowledge* if the verifier gains no knowledge beyond the correctness of the statement being proved, and it is *succinct* if verifying the proof takes time poly-logarithmic in the computation. The proof system is *non-interactive* if the prover can generate the proof in one round and any verifier can check this proof and be convinced of the outcome.

21st International Workshop on Satisfiability Modulo Theories, July 5-6, 2023, Rome, Italy
fatemeh.heidari-soureshjani@polymtl.ca (F. H. Soureshjani); mathias@hall-andersen.dk (M. Hall-Andersen); mohammad@quantstamp.com (M. Jahanara); jeffrey@quantstamp.com (J. Kam); jan@quantstamp.com (J. Gorzny); mohsen@quantstamp.com (M. Ahmadvand)
0000-0003-2197-8078 (F. H. Soureshjani); 0000-0002-0195-6659 (M. Hall-Andersen); 0009-0009-4483-406X (M. Jahanara); 0000-0002-4332-0026 (J. Kam); 0000-0003-1435-8508 (J. Gorzny); 0009-0006-2125-2225 (M. Ahmadvand)

A zero-knowledge proof system that has all three properties is often called a *zero-knowledge Succinct Non-interactive ARguments of Knowledge* or *zk-SNARK* for short. Example zk-SNARKs include Groth16 [2], Halo [3], and PLONK [4].

The properties of zk-SNARKs described above make zk-SNARKs attractive and useful for blockchain applications. First, zero-knowledge proof systems may be used to enable private and anonymous transactions on these trustless ledgers. For example, Tornado Cash [5] is a zero-knowledge application to anonymize transactions on the Ethereum blockchain [6]. Second, the non-interactive and succinct properties of such a system may be used to scale these systems even when the zero-knowledge property is not desired. That is, succinctness is valuable even when all inputs to the proof system are assumed to be public. Succinct proofs enable scaling of blockchains like Ethereum as computation can be performed off-chain, but verified on-chain. So-called *rollups* (also called *validating bridges* [7] or *commit-chains* [8]) are protocols that conform to this idea of separating execution and verification. Rollups which utilize zero-knowledge proof systems are called *zero-knowledge rollups*[1]. Example of zero-knowledge rollups include the Polygon zkEVM, StarkNet, and Scroll, among others.

zk-SNARKs are often used to prove statements of the format "*For function f and public input x, I know a private witness w such that $f(x, w) = y$*". A suitable artihmetization of the function is necessary as the SNARK proves a NP relation, often over a finite field. This can be done directly for simple functions, or using a compiler from a Domain-Specific Language (DSL) for more complex functions like those that represent the execution of virtual machines. Examples of zero-knowledge DSLs include Circom [9] and ZoKrates [10]. The use of these DSLs enable developers to build zero-knowledge applications by abstracting away the explicit notion of computing the function f described above.

Unsurprisingly, zero-knowledge applications may contain bugs. First, developers writing such zero-knowledge applications in a DSL may introduce bugs directly. Second, there may be errors in the resulting arithmetization introduced by the DSL compiler (e.g., bugs in Leo[2] and Noir[3]). To make matters worse, for complicated applications, these proof systems translate computations to arithmetizations too large to be manually checked.

To overcome this challenge, the zero-knowledge application community has turned to tooling. Automated tools like Ecne[4], Picus[5], and QED^2 [11] exist for checking *Rank-1 Constraint System* (R1CS) arithmetizations. These tools often check various notions of correctness, but pay special attention to whether or not the resulting system is *under-constrained*: if it has multiple valid efficiently computed assignments to satisfy the constraints (this is defined formally in Section 3.2). Under-constrained circuits could have allowed user funds to be stolen in Tornado Cash [12] or allowed repeated spending of tokens in the Aztec network [13], though both instances have been corrected.

Other representations exist and often build on the PLONK [4] proof system; these are said to use *PLONKish* arithmetization. Oversimplifying, this representation transforms a compu-

[1]We note that the term *zero-knowledge rollup* is a misnomer: the *zero-knowledge* part is often not important; more accurately these systems only value the computational integrity proofs generated by the zero-knowledge framework.
[2]https://github.com/AleoHQ/leo/issues/2042
[3]https://github.com/noir-lang/noir/issues/358
[4]https://github.com/franklynwang/EcneProject
[5]https://github.com/Veridise/Picus

tation to a circuit, consisting of a set of gates. Each circuit is represented as a table of values and polynomials constraining the table's rows. A more detailed description is provided in Section 2. To the best of our knowledge, no automated tools exists to check applications built using PLONKish arithmetizations. This work describes a proof of concept[6] for automatically checking such applications for correctness for the Halo2[7] proof system, which uses PLONKish arithmetization.

We describe approaches for push button analysis and outline how our tool can be adopted to check custom properties. We use a form of abstract interpretation [14] to check PLONKish arithmetizations. We search for unused gates and columns, as well as table entries which are assigned but not constrained. Unused gates and columns may be bugs as the developer may have intended to reference them. Entries which are assigned but not constrained may be a bug, and are instances of under-constrained circuits. Due to the severity of under-constrained circuits and the fact that our abstract interpretation approach may return false negatives, we provide another method to check if a circuit is under-constrained. We employ a Satisfiability Modulo Theories (SMT) solver (see e.g., [15]) to try to find multiple satisfying models (i.e., private inputs) for a circuit and a (fixed) public input in order to determine if it is under-constrained. Finally, we outline how to check custom properties by making the necessary modifications to calls to the SMT solver.

The paper is outlined as follows. Section 2 details the Halo2 proof system. Section 3 outlines methods for analyzing Halo2 circuits using abstract interpretation (Section 3.1) and SMT solvers (Section 3.2). Section 4 discusses related work and limitations of our work. Section 5 concludes the paper.

2. Halo2

Halo2 is a popular zero-knowledge proof system implemented in Rust. It uses PLONKish arithmetization to express functions or applications as circuits, as originally introduced by PLONK [4]. PLONK has various extensions (e.g., HyperPLONK [16], TurboPLONK [17], and UltraPLONK [18]) which adds more expressive custom gates to the proof system, combines PLONK with so-called *lookup* tables [19], and provides more efficient proving mechanisms for the same type of arithmetization.

PLONKish circuits are represented as tables of values from a finite field. Entries of the table are called *cells*. Circuits are made up of gates, which are constraints applied to sub-tables (a set of cells) across the table. Table columns have three main types: *instance*, *advice*, and *fixed*. *Instance* columns store public input, *advice* columns store private witness (private inputs and intermediate values used by the prover in the circuit), and *fixed* columns store fixed values that are part of the description of the circuit. The values of the table are constrained by three types of constraints: *gate constraints* and *equality constraints*, which we now define, and those from *lookups*, which we describe in Section 3.2.1. Gate constraints are generic polynomial constraints defined by the circuit designer. We will use $g(\langle w \rangle)$ to denote a gate constraint on $\langle w \rangle$, which represents an encoding of the witness w along with (unnamed) public inputs. Each

[6] Available at https://github.com/quantstamp/halo2-analyzer
[7] https://github.com/zcash/halo2

Selector (s)	Instance (b)	Advice (a)	Fixed (C)
1	15	5	3
Established at circuit configuration.	*Public input; known to prover and verifier.*	*Private witness; known only to prover*	*Public constant; known to prover. and verifier.*

Table 1
A very small Halo2 circuit's table, filled with real values. The corresponding polynomial constraint is equation (1). The table is filled to demonstrate that the prover knows an assignment $a = 5$ such that $C \cdot a = 3 \cdot 5 = 15 = b$ (which implies $C \cdot a - b = 0$, as required). That is, the prover knows the necessary factor a such that $C \cdot a = b$. Since $s = 1$, the gate is on.

gate constraint applies to a number of adjacent rows of the gate. Therefore a circuit whose table representation has several rows may have several polynomial constraints, all of which should be simultaneously satisfied. Equality constraints (a.k.a. *permutation constraints* or *copy constraints*) can enforce specific group of cells to have equal values across rows and columns. We refer to the table and its corresponding constraints as the *constraint system.*

With Halo2, gates are enabled by *selector* variables (or simply *selectors*). A *selector* is a binary variable that is multiplied to the polynomial constraint for each row: it is 0 if the gate is off or 1 if it is on. We will use s to denote a selector variable, and as a result, a polynomial constraint g on a gate is more accurately[8] represented as $f(\langle w \rangle, s) = s \cdot g(\langle w \rangle)$ for a witness w.

We say that 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. This is done either by setting the irrelevant selector variables to 0, or by providing (possibly secret) assignments to the table cells that result in the polynomial's evaluation to 0.

We illustrate the computation of a Halo2 gate in the following example. Suppose we have the following polynomial gate constraint, whose table is shown in Table 1:

$$f(\langle a, b, C \rangle, s) = s \cdot (C \cdot a - b). \tag{1}$$

The variables have the following interpretations: s is the selector (if it is 0 then the polynomial evaluates to 0), a is the input of the gate, b is the output of the gate, and C is a parameter used to configure the gate (a constant in the circuit). The gate computes the following relation: $b = C \cdot a$ (so that the polynomial is 0 even when $s = 1$). Thus the gate in this example is a multiplication by a constant gate.

A *region* in Halo2 is a logical collection of cells with a set of enabled gates (as indicated by the selectors). This work operates by collecting all the gates and regions using a custom implementation of the Halo2 `Layouter` trait, which normally is responsible for the place-and-route behaviour of compiling the circuit. This functionality is responsible for mapping the regions to rows to minimize the number of required rows to enforce the constraints; it essentially solves a packing problem. This is done because there is a limit to the number of rows used, though we can increase the number of columns at will by adding some complexity to the proof system. Our implementation of this trait, `AnalyticalLayouter`, will simply record the

[8]Recall that a gate is technically applied to all rows, and so this is actually the Hadamard product.

details of every region. Since we never need to actually generate a proof, an optimal packing is not necessary.

Finally, we note that the Halo2 Rust library offers a `Circuit` trait, the standard interface that should be implemented by any circuit object. We are also interested in one method of the `Circuit` trait, `synthesize`. The `synthesize` function is responsible both for creating the constraint system and witness generation. This is relevant for Section 3.2.

3. Analyses

In this section, we describe various analyses that can be performed on Halo2 circuits.

3.1. Unused Gates, Unused Columns, and Assigned but Unconstrained Cells

We first describe how to find unused gates and columns, as well as assigned but unconstrained cells, for a PLONKish circuit using a form of abstract interpretation (see e.g. [14]).

For a given computation, we have the set of enabled selectors, which determine which gates are to be used. We can assume that we have dummy values within cells (for reasons that will become clear later). For a set of selector variables set to 1, we have a set of corresponding polynomials; thus we have a set of polynomials for each gate that is turned on. Recall that if the circuit is satisfied, then every polynomial of every gate should vanish over every region according to the Halo2 proof system. Note that every gate which is turned off automatically vanishes, as it is multiplied by its zero-valued selector.

At evaluation time, we do not have a witness w and therefore we can't evaluate the gate polynomials (and doing so would always yield 0 anyway). What we do have is the selector values and constants which enables us to abstractly evaluate every polynomial of every gate in every region for concrete values of selector and constants. Thus, we can replace the polynomial's variables with abstract ones (and this is why the cells may be assumed to have dummy values). This approach abstracts away the exact values of variables so that each is either zero or non-zero, and consider their abstract evaluation within polynomials. In turn, some polynomials can be determined to be identically zero.

The first check we can implement via the abstract interpretation defined above is for unused custom gates. After running synthesis with the `AnalyticalLayouter` we have assignments for selector variables s and any constants C (but not for any witnesses, like a and b) in a gate polynomial like equation (1). As such we cannot fully evaluate a gate polynomial but we can determine if it is identically zero, i.e. evaluates to zero for every witness variable assignment. In order to do this, we replace every variable in a polynomial with an abstract variable type that is one of {`Variable`, `NonZero`, `Zero`}. Then, we can abstractly evaluate every polynomial and possibly determine if it is identically zero. Figure 1 shows an example of evaluating a sum in a polynomial using abstract interpretation with the variable types described above in Rust. Multiplication is defined similarly. For evaluating selector variables within the library, we can use `Zero` if the selector is set to zero and `NonZero` otherwise; similarly we can check if a constant is zero or not and use the appropriate abstract variable type. Negation has no effect on whether a variable is in fact `Zero` or `NonZero`.

```
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,
}
}
```

Figure 1: An example of evaluating a sum in a polynomial using abstract interpretation.

The use of abstract interpretation is fast but comes at the cost of accuracy. Our system may return that a polynomial is not identically zero, when in fact it is (i.e., a *false negative*).

Finally, as the set of gates and their polynomials are provided by the appropriate Rust trait for Halo2, we can use abstract interpretation as defined above to check for:

- **Unused (Custom) Gates**: check that for every gate's polynomials there exists a region in which it is not identically zero. Unused gates are likely bugs because the developer likely intended to reference each gate that they created.
- **Unconstrained Cells**: check that for every assigned cell in the region, it occurs in a polynomial which is not identically zero over this region. Unconstrained cells are almost certainly a bug, as the system may be under-constrained (see also Section 3.2). For example, the designer may have forgotten to enable a selector.
- **Unused Columns**: check that every column occurs in some (not identically zero) polynomial. Unused columns may be a bug (perhaps the columns were meant to be referenced), or enable an optimization by removing unused columns.

3.2. Under- and Over-constrained Circuits

In the context of PLONKish circuits, we define the properties of being under- and over-constrained, before showing how to detect such circuits using an SMT solver.

Let C be a PLONKish circuit. We call C *under-constrained* if there exist 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.

Being under-constrained is a usually a sign of error by circuit designers; most likely, the arithmetization is not faithful to the intended function. One common example is calculating value of a cell in the table as part of witness generation in `synthesize` but failing to select (or activate) the right gates to enforce a constraint over this cell. This may enable a malicious prover to create valid proofs that will be accepted by the verifier but should not be. Recall that under-constrained circuits have been a source of errors for real-world protocols. While some circuits may be under-constrained intentionally, we assume that users of this tool expect their circuit to have a single assignment for every valid input.

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. Note that the last part of the definition is necessary, as otherwise some circuits would always be over-constrained. As a trivial 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 $x \geq 2$, this circuit can be satisfied (in multiple ways), but for $x = 1$ this is not the case, and it would not be meaningful to call the circuit over-constrained for this input value.

Example 3.1. We provide an example where the aim is to check if a public input x is in the set $\{0, 1, 2, 3\}$. The code aims to implement the following constraints over a suitable Halo2 table:

$$b_0 \cdot (b_0 - 1) = 0 \tag{2}$$
$$b_1 \cdot (b_1 - 1) = 0 \tag{3}$$
$$b_0 + 2 \cdot b_1 - x = 0 \tag{4}$$

This set of constraints is uniquely satisfied if $x \in \{0, 1, 2, 3\}$, but is not satisfiable otherwise.

We provide an example of an under-constrained implementation of the previous system in Figure 2. The `meta` object is an instance of a `ConstraintSystem`, which creates Halo2 gates. In this example, lines 1-5 create the gate implementing constraint (2), lines 6-10 create the gate implementing constraint (3), and lines 11-17 create the gate implementing constraint (4). Line 2 gets a variable `b1` to be used in the gate from the advice column in the current row[9] because of the function `cur` called on the `Rotation` object. Line 3 gets the selector value from the configuration, that is, if this gate will be used. Finally line 4 encodes the expression $\texttt{d} \cdot \texttt{a} \cdot (1 - \texttt{a})$, making new copies of `a` as required by Rust. The expression on line 4 will be assumed to be equal to zero in the Halo2 library.

Lines 1-5 and 6-10 are similar; so similar, that the developer could copy-and-paste lines 1-5 to get lines 6-10 with small changes. However, in this example, the necessary modifications are not completed: the gate's name was updated on line 6, but the variable name `b1` was not updated on line 7. As a result, lines 6-10 actually encode the same constraint as lines 1-5.

Line 16 encodes the expression $\texttt{d} \cdot (\texttt{a} + 2\texttt{b} - \texttt{c})$ which represents $b_0 + 2b_1 = x$ as the proof system will enforce that the expression is equal to zero, along with the selector factor `d`. Note that although line 14 uses `x` from an advice column, it can be enforced to have the same value as the public input elsewhere in the code, e.g., by using the `constrain_instance` function and the appropriate public instance column, elsewhere in the code.

Throughout this example, we can ignore the selector values `d` as we will assume all gates are used and this is the only configuration of interest. With that in mind, we can see that the system will be under-constrained: when we set $x = 3$, there are two ways to assign values to the advice columns so that the system is satisfied: $\{b_0 = 1, b_1 = 1\}$ and $\{b_0 = 3, b_1 = 0\}$. In short, the hasty developer did not constraint b_0 to be in $\{0, 1\}$.

[9]If the rotation referenced `prev` or `next` or something else, we would change the index of the corresponding variable in the generated output to reference the concrete cell, rather than the relative one. For example, if `B-7` is the variable for the 7th row of the B column but the rotation was `next`, we add a constraint that `B-7=B-8`.

```
1  meta.create_gate("b1_binary_check", |meta| {
2      let a = meta.query_advice(b1, Rotation::cur());
3      let d = meta.query_selector(s);
4      vec![d * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
5  });
6  meta.create_gate("b0_binary_check", |meta| {
7      let a = meta.query_advice(b1, Rotation::cur());
8      let d = meta.query_selector(s);
9      vec![d * a.clone() * (Expression::Constant(Fr::from(1)) - a.clone())]
10 });
11 meta.create_gate("equality", |meta| {
12     let a = meta.query_advice(b0, Rotation::cur());
13     let b = meta.query_advice(b1, Rotation::cur());
14     let c = meta.query_advice(x, Rotation::cur());
15     let d = meta.query_selector(s);
16     vec![d * (a + Expression::Constant(Fr::from(2)) * b - c)]
17 });
```

Figure 2: An erroneously under-constrained set of gates in Halo2. The variable `b1` on line 7 should in fact be `b0`.

We now describe how to identify an under-constrained or over-constrained system in Halo2 using cvc5 [20]. We use the cvc5 solver as there is a finite field solver incorporated into it [21], facilitating a more direct conversion of polynomials to solver constraints. We focus on detecting under-constrained systems, but are able to check over-constrained systems for free. The idea is to extract the polynomial constraints, convert them to a model that can be checked for satisfiability in an SMT solver, and then determine if there are multiple satisfying models.

3.2.1. From Halo2 to SMT

We outline the conversion of Halo2 constraints into the format required by an SMT solver.

First, we build constraints based on the polynomials built in Halo2. Within our configuration, we can find all gates $\mathcal{G}_i$ within a region. Let $\mathcal{G}$ be the set of all gates $\mathcal{G}_i$ (that is, $\mathcal{G} = \{\mathcal{G}_i \mid \mathcal{G}_i \text{ is a gate}\}$). For each gate $\mathcal{G}_i$, we can collect each polynomial $\mathcal{P}_{i,j}$ defined within it: let $\mathcal{P}_i = \{\mathcal{P}_{i,j} \mid \mathcal{P}_{i,j} \text{ is a polynomial in } \mathcal{G}_i\}$; finally, let $\mathcal{P} = \cup_i \mathcal{P}_i$.

The polynomials are created in two major steps within the proof system: circuit configuration and when circuits are synthesized. The result provides a set of polynomials based on selectors set during these steps, as well as constraints that enforce values and the relationship between values in the table. Specifically, we enforce values given from a particular input and any witnesses.

Next, we build SMT constraints based on the polynomials in the set $\mathcal{P}$. For all polynomials $\mathcal{P}_{i,j} \in \mathcal{P}$ collected from constraint system, we add an `assert` statement, where $\mathcal{P}_{i,j}$ is converted to an expression accepted by the SMT input language (SMT-LIB [22]). An example of an equality constraint is shown in Figure 3. A variable `A-i-j-k` is the value of column A in region i, row j, and column k; `A-1-1-1` is the first entry of the first row and column in the first region.

We create an instance `S` of an SMT solver. To this solver, we add each of the assertions created

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

Figure 3: Example SMT output generated by our tool in the format of SMT-LIB [22]. The finite field logic (Quantifier Free Finite Field) is declared on line 1. In this example, the field is over the prime 307. Line 2 declares a variable for the model, and line 3 declares a sample equality constraint that `A-1-1-1` $= 0$, as `ff0` represents zero in the finite field (not shown).

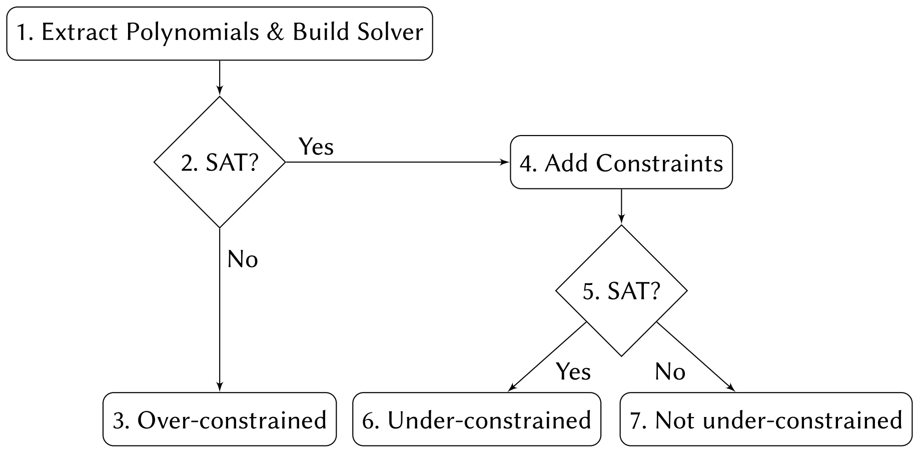

Figure 4: The process used to detect under- and over-constrained systems in our tool.

from the set $\mathcal{P}_{i,j}$, and thus the initial set of constraints in the solver is

$$\mathcal{C} = \bigwedge_{i,j} (\mathcal{P}_{i,j} = 0),$$

along with all copy constraints, constraints necessary to enforce equality due to rotations, and constraints related to lookups.

Halo2 handles lookup tables by adding a fixed column for each column in the lookup table. Within the constraint system, there is a `lookups` component which has an `argument` field, which comprises two vectors: `input_expressions` and `table_expressions`. The `input_expressions` vector determines which advice columns are constrained to occur in the lookup table. The `table_expressions` vector includes the column indexes of the fixed columns, which should equal the advice columns extracted from `input_expressions`. Using these vectors, we can determine which advice entry should occur in which fixed column index. Then, we implement the search in the lookup table as a constraint. For that, we need to have the actual values for the lookup table columns. We can extract this by running the Halo2 mock prover, which gives us a two-dimensional vector (`Fixes`) that includes each fixed column's values. The lookup can then be modelled as a constraint as a disjunction of conjunctions. Each conjunction is the constraint required by the fixed columns, mapped to entries in the advice columns. Since any values satisfying such a constraint is valid, we take the disjunction over

Advice A_0	Fixed F_0
$\vdots$	`c89efdaa`
$\vdots$	`7f8b6b08`
a_i	`2a80e1ef`
$\vdots$	`13600b29`
$\vdots$	`ceebf77a`

Table 2
A small range check modelled as a lookup in a Halo2 circuit table (only the relevant columns are shown). In this case, we want the i row of the advice column A_0, a_i to be one of the values in the only fixed column F_0. We add the following constraint to the solver: $(a_i = c89efdaa) \vee (a_i = 7f8b6b08) \vee ... \vee (a_i = ceebf77a)$.

Advice A_0	Advice A_1	Advice A_2	Fixed F_0	Fixed F_1	Fixed F_2
$\vdots$	$\vdots$	$\vdots$	000	000	000
$\vdots$	$\vdots$	$\vdots$	000	001	001
$a_{0,i}$	$a_{1,i}$	$a_{2,i}$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	111	111	000

Table 3
A lookup in a Halo2 circuit table with constraints on the fixed columns (only the relevant columns are shown). In this case, the entry of F_2 at row i is the bit-wise exclusive or function (XOR) applied to the values of row i in F_0 and F_1. In this case, we want the i row of the advice columns A_0, A_1, and A_2 to be a valid XOR for entries which have three bits. We add the following constraint to the solver: $(a_{0,i} = 000 \wedge a_{1,i} = 000 \wedge a_{2,i} = 000) \vee (a_{0,i} = 000 \wedge a_{1,i} = 001 \wedge a_{2,i} = 001) \vee ... \vee (a_{0,i} = 111 \wedge a_{1,i} = 111 \wedge a_{2,i} = 000)$.

all possible valid conjunctions. We illustrate via examples. A simple range look up is shown in Table 2; in this case, the inner conjunction does not exist; it has only one term. A more complicated example is shown in Table 3, which illustrates the need for the internal conjunction.

We now describe the process used to determine if a system is under- or over-constrained. This process is illustrated in Figure 4. The solver S is queried to see if it can find a satisfying assignment, and if so, a model that satisfies S (i.e., an assignment to the variables of the polynomials) is generated. If no such model is found, the system is *over-constrained* (and hence not under-constrained). If a model has been obtained, the system may not be under-constrained. We seek to find another, different model to also satisfy the system. Therefore, we add a set of constraints that ensure that at least one variable in the model must be different.

Let $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ be variables that are constraining public inputs, and let $\{\beta_1, \beta_2, ..., \beta_m\}$ be the variables that do not constrain public inputs. Let $\{\hat{\alpha}_1, \hat{\alpha}_2, ..., \hat{\alpha}_n\}$ (respectively $\{\hat{\beta}_1, \hat{\beta}_2, ..., \hat{\beta}_m\}$) be the set of values assigned by the satisfying model to the constraining (respectively non-constraining) variables. Then we have that $\alpha_i = \hat{\alpha}_i$ for $1 \leq i \leq n$ and $\beta_j = \hat{\beta}_j$ for

$1 \leq j \leq m$ in the model. We create a new solver `S'` whose constraint set is

$$\mathcal{C}' = \{\mathcal{C} \wedge (\beta_1 \neq \hat{\beta}_1 \vee \beta_2 \neq \hat{\beta}_2 \vee ... \vee \beta_m \neq \hat{\beta}_m)\} \quad (5)$$

If `S'` is satisfiable, we have proved that the circuit is under-constrained.

3.2.2. Fuzzing

Our system can be configured to also generate multiple public inputs. We ask for a satisfying assignment to the circuit, and if one is found, we ask if another satisfying assignment is possible for this particular public input assignment. If it is, we stop, as we have shown that the circuit is under-constrained. If not, we check if the circuit can be satisfied by another public input, and repeat until no more assignments can be found or we run out of time. This way the system acts as a sort of "fuzzer," allowing developers to explore assignments. We outline this process now.

Let $\mathcal{C}$ be the constraints for the circuit, as defined in Section 3.2. Then, we iterate over possible assignments to $\mathcal{C}$ to try to find a public input for which the circuit would be under-constrained.

Let $\mathcal{C}_0 = \mathcal{C}$. Then, at iteration $i \geq 0$, we ask a solver `S` to determine if $\mathcal{C}_i$ is satisfiable; if not, the circuit is over-constrained or we have exhausted all valid public inputs. The loop stops if $\mathcal{C}_i$ is not satisfiable.

We may therefore assume $\mathcal{C}_i$ is satisfiable with a public input assignment set $\{\hat{\alpha}_1^i, \hat{\alpha}_2^i, ..., \hat{\alpha}_n^i\}$ and witness assignment set $\{\hat{\beta}_1^i, \hat{\beta}_2^i, ..., \hat{\beta}_m^i\}$. Then we ask a solver to determine if

$$\mathcal{C}_i' = \{\mathcal{C}_i \wedge (\beta_1 \neq \hat{\beta}_1^i \vee \beta_2 \neq \hat{\beta}_2^i \vee \ldots \vee \beta_m \neq \hat{\beta}_m^i) \wedge (\alpha_1 = \hat{\alpha}_1^i \wedge \alpha_2 = \hat{\alpha}_2^i \wedge \ldots \wedge \alpha_n = \hat{\alpha}_n^i)\} \quad (6)$$

is satisfiable. If $\mathcal{C}_i'$ is also satisfiable, we have shown that the circuit is under-constrained as we have two assignments to private witness for a single assignment to the variables of $\{\hat{\alpha}_1^i, \hat{\alpha}_2^i, ..., \hat{\alpha}_n^i\}$.

Otherwise, we determine a new public input to try for constraint set $\mathcal{C}_{i+1}$ by preventing any future solver calls from using the public input assignment $\{\hat{\alpha}_1^i, \hat{\alpha}_2^i, ..., \hat{\alpha}_n^i\}$. Explicitly, we have

$$\mathcal{C}_{i+1} = \{\mathcal{C}_i \wedge (\alpha_1 \neq \hat{\alpha}_1^i \vee \alpha_2 \neq \hat{\alpha}_2^i \vee \ldots \vee \alpha_n \neq \hat{\alpha}_n^i)\}. \quad (7)$$

Now the process repeats with $\mathcal{C}_{i+1}$.

3.2.3. General Property Checking

The use of an SMT solver to detect under- and over-constrained circuits allows Rust-savy users to check other properties in some situations.

As an example, we illustrate another way we could find that the previous example is under-constrained. Suppose that instead of asking for two satisfying models, we are conscious of the need for b_0 and b_1 to be binary variables and want to see if it is ever the case that a satisfying model would violate these constraints. In this case, we could ask the SMT solver to take the constraint set $\mathcal{C}$ and also add a constraint $\mathcal{B}$ which says $b_0 \neq 1 \wedge b_0 \neq 0 \wedge b_1 \neq 0 \wedge b_1 \neq 1$. Assuming the solver does not time out, the result is either *satisfiable* or *unsatisfiable*. If $\mathcal{C} \wedge \mathcal{B}$ is satisfiable, it is not the case that both b_0 and b_1 are always binary variables. If the solver finds that these sets of constraints are unsatisfiable, this means that it is always the case that b_0 and b_1 are binary variables (assuming $\mathcal{C}$ is satisfiable on its own). If the SMT solver is unable to find a satisfiable example, the answer may not be known.

4. Related Work and Discussion

In this section, we comment on related work and discuss future directions of our work.

4.1. Related Work

As mentioned in Section 1, Ecne and Picus are tools for checking intermediate representations of proof systems that use a so-called Rank-1 Constraint System (R1CS) intermediate representation. These are not compatible with the Rust files used in the Halo2 proof system without additional effort. For instance, R1CS only supports quadratic polynomials while Halo2 polynomials may have arbitrary degree. Vella and Alt [23] presents a similar tool to check R1CS circuits, which uses SMT solvers to reach conclusions over large finite fields. More recently, Pailoor et al. [11] describe the QED^2 system for automated detection of under-constrained systems in R1CS using the same cvc5 solver. Their approach performs so called *semantic reasoning* over R1CS equations to determine if there is a unique solution, and may be more general then our approach. Their approach may apply to Halo2 circuits with some modification, though they point out that it may not benefit from their uniqueness constraint propagation.

Thomas [24] provides a specification language, Orbis, for zk-SNARK programming. This work is complementary as it provides a way to write Halo2 circuits, but does not comment on how to check the correctness of Halo2 circuits written via any means. The tooling described in this work may also be used for circuits generated from Orbis in the event that the Orbis compiler is not trustworthy.

Chin et al. [25] describe LEO, a language for formally verifying zero-knowledge applications. LEO aims to provide stronger guarantees for applications written in a custom zero-knowledge language, rather than relatively easy analysis of applications using the Halo2 library. As a result, the LEO language aims to solve problems beyond the scope of this paper.

4.2. Discussion

We now discuss the limitations and future directions for this work.

First, the use of an SMT solver likely means that this approach will not scale. Future work is required to generate example circuits or find real-world circuits that can be analyzed, in order to determine an approximate number of constraints and variables that can be checked by the solver before timing out. This may still be of use, as our approach can be applied to sub-circuits of an arithmetized function. Repositories of circuits like Circom[10] (used to create the ZKBENCH circuits of [11]) or those used to evaluate Ecne and Picus, do not exist for Halo2 to provide a corpus of circuits to evaluate our tool on. In fact, once a corpus of Halo2 circuits are collected,

[10] https://github.com/iden3/circomlib

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.

one could compare the approach of [11] to our approach. Another option is to apply this tool to the open-source collection of Halo2 circuits in the Privacy-Scaling Exploration zkEVM[11].

Pilspector[12] uses SMT solvers to analyse state machines written in the Polynomial Identity Language (PIL) language. It looks for nondeterminism in PIL descirptions which are in turn used by the eSTARK proof system [26]. Another interesting direction to consider is to compare their approach to encoding lookups with the approach taken by this tool.

As a starting point for the evaluation, we ran our tool on generalizations of Example 3.1. We analysed circuits for decomposing numbers into k bits for $k \in \{2, 4, 8, 16, 32, 64, 128\}$. Our evaluation chooses a random assignment to the public input and looks for two witness assignments (using the fuzzing approach, but limited to a single iteration). The run times are reported in Table 4; they were computed using a Apple MacBook Pro with an M1 Pro processor with 10 cores and 16GB of RAM. The time increases with the number of bits in the decomposition, which corresponds (roughly) to the number of variables in the model.

Second, there are likely additional issues for Halo2 that can be detected automatically. A collection of bugs related to zero-knowledge bugs is being maintained[13], though not all may be applicable to Halo2 application developers or the Halo2 proof system itself. We could also check that best practices are used in Halo2 programming, adding functionality akin to a linter, in future work.

5. Conclusion

We have presented methods to detect several issues in the Halo2 codebase using techniques and tooling which integrate directly into the Rust implementation. Our tool can find instances of under- and over-constrained circuits, as well unused gates, unconstrained cells, and unused columns. All of these issues may indicate an error with the circuits implemented by a Halo2 circuit developer. Our work is a proof-of-concept and incomplete: additional work is necessary to determine how well it scales, and if further analyses can be included. However, it demonstrates that Halo2 circuits are not beyond the realm of automated analysis.

Acknowledgements. The authors would like to thank Hanifa Boucheneb, Igor Braga, Martin Derka, and the anonymous SMT 2023 reviewers for their helpful comments on the codebase, future directions of this work, and drafts of this paper.

[11]https://github.com/privacy-scaling-explorations/zkevm-circuits
[12]https://github.com/Schaeff/pilspector/
[13]https://github.com/0xPARC/zk-bug-tracker

References

[1] S. Goldwasser, S. Micali, C. Rackoff, The knowledge complexity of interactive proof systems, SIAM J. Comput. 18 (1989) 186–208. URL: https://doi.org/10.1137/0218012.

[2] J. Groth, On the size of pairing-based non-interactive arguments, in: M. Fischlin, J. Coron (Eds.), Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II, volume 9666 of *Lecture Notes in Computer Science*, Springer, 2016, pp. 305–326. URL: https://doi.org/10.1007/978-3-662-49896-5_11.

[3] S. Bowe, J. Grigg, D. Hopwood, Halo: Recursive proof composition without a trusted setup, IACR Cryptol. ePrint Arch. (2019) 1021. URL: https://eprint.iacr.org/2019/1021.

[4] A. Gabizon, Z. J. Williamson, O. Ciobotaru, PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge, IACR Cryptol. ePrint Arch. (2019) 953. URL: https://eprint.iacr.org/2019/953.

[5] A. Pertsev, R. Semenov, R. Storm, Tornado cash privacy solution version 1.4, 2019. URL: https://berkeley-defi.github.io/assets/material/Tornado%20Cash%20Whitepaper.pdf.

[6] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[7] P. McCorry, C. Buckland, B. Yee, D. Song, SoK: Validating bridges as a scaling solution for blockchains, IACR Cryptol. ePrint Arch. (2021) 1589. URL: https://eprint.iacr.org/2021/1589.

[8] R. Khalil, A. Zamyatin, G. Felley, P. Moreno-Sanchez, A. Gervais, Commit-chains: Secure, scalable off-chain payments, IACR Cryptol. ePrint Arch. (2018) 642. URL: https://eprint.iacr.org/2018/642.

[9] M. Bellés-Muñoz, J. B. Melé, V. Daza, J. L. Muñoz-Tapia, New privacy practices for blockchain software, IEEE Softw. 39 (2022) 43–49. URL: https://doi.org/10.1109/MS.2021.3086718.

[10] J. Eberhardt, S. Tai, ZoKrates - scalable privacy-preserving off-chain computations, in: IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax, NS, Canada, July 30 - August 3, 2018, IEEE, 2018, pp. 1084–1091. URL: https://doi.org/10.1109/Cybermatics_2018.2018.00199.

[11] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez, J. V. Gaffen, J. Morton, M. Chu, B. Gu, Y. Feng, I. Dillig, Automated detection of underconstrained circuits for zero-knowledge proofs, Cryptology ePrint Archive, Paper 2023/512, 2023. URL: https://eprint.iacr.org/2023/512.

[12] Tornado Cash, Tornado.cash got hacked. By us., 2019. URL: https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8.

[13] M. Connor, Disclosure of recent vulnerabilities, 2021. URL: https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities.

[14] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: R. M. Graham, M. A. Harrison, R. Sethi (Eds.), Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, ACM, 1977, pp.

238–252. URL: https://doi.org/10.1145/512950.512973.

[15] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability modulo theories, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009, pp. 825–885. URL: https://doi.org/10.3233/978-1-58603-929-5-825.

[16] B. Chen, B. Bünz, D. Boneh, Z. Zhang, HyperPlonk: Plonk with linear-time prover and high-degree custom gates, IACR Cryptol. ePrint Arch. (2022) 1355. URL: https://eprint.iacr.org/2022/1355.

[17] A. Gabizon, Z. J. Williamson, The turbo-PLONK program syntax for specifying snark programs, 2019. URL: https://docs.zkproof.org/pages/standards/accepted-workshop3/proposal-turbo_plonk.pdf.

[18] Zac, Aztec's ZK-ZK-Rollup, looking behind the cryptocurtain, 2021. URL: https://medium.com/aztec-protocol/aztecs-zk-zk-rollup-looking-behind-the-cryptocurtain-2b8af1fca619.

[19] A. Gabizon, Z. J. Williamson, plookup: A simplified polynomial protocol for lookup tables, IACR Cryptol. ePrint Arch. (2020) 315. URL: https://eprint.iacr.org/2020/315.

[20] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, Y. Zohar, cvc5: A versatile and industrial-strength SMT solver, in: D. Fisman, G. Rosu (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I, volume 13243 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 415–442. URL: https://doi.org/10.1007/978-3-030-99524-9_24.

[21] A. Ozdemir, G. Kremer, C. Tinelli, C. W. Barrett, Satisfiability modulo finite fields, IACR Cryptol. ePrint Arch. (2023) 91. URL: https://eprint.iacr.org/2023/091.

[22] C. Barrett, P. Fontaine, C. Tinelli, The Satisfiability Modulo Theories Library (SMT-LIB), www.SMT-LIB.org, 2016.

[23] L. C. Vella, L. Alt, On satisfiability of polynomial equations over large prime fields, in: D. Déharbe, A. E. J. Hyvärinen (Eds.), Proceedings of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022, volume 3185 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2022, pp. 114–127. URL: http://ceur-ws.org/Vol-3185/extended9913.pdf.

[24] M. Thomas, Orbis specification language: a type theory for zk-SNARK programming, IACR Cryptol. ePrint Arch. (2022) 1003. URL: https://eprint.iacr.org/2022/1003.

[25] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, E. Smith, Leo: A programming language for formally verified, zero-knowledge applications, IACR Cryptol. ePrint Arch. (2021) 651. URL: https://eprint.iacr.org/2021/651.

[26] H. M. Ardevol, M. Guzmán-Albiol, J. B. Melé, J. L. Muñoz-Tapia, eSTARK: Extending starks with arguments, IACR Cryptol. ePrint Arch. (2023) 474. URL: https://eprint.iacr.org/2023/474.