Towards using Horn Clauses in Zero-Knowledge Protocols

Elvira Albert[1,2], Miguel Isabel[2], Clara Rodríguez-Núñez[2], and Albert Rubio[1,2]

[1] Institute of Knowledge Technology, Spain
[2] Complutense University of Madrid, Spain

Zero-knowledge (ZK) protocols [3,6,2,4] enable one party, called prover, to convince another one, called verifier, that a "statement" is true without revealing any information beyond the veracity of the statement. Most ZK systems operate on the model of *arithmetic circuits*, meaning that the language used to come up with the statement is that of satisfiable arithmetic circuits. The gates of an arithmetic circuit consist of additions and multiplications modulo p, where p is typically a large prime number of approximately 254 bits [1]. The wires of an arithmetic circuit are called signals, and can carry any value from the prime finite field $\mathbb{F}_p$. In such context, there is a set of public inputs known both to prover and verifier, and the prover proves that, with that public information, she knows a valid assignment to the rest of signals of the circuit that makes the circuit satisfiable (i.e., the statement is true). Most ZK systems use R1CS (Rank-1 Constraint System) for encoding circuits and wire assignment. An R1CS encodes a program as a set of quadratic constraints over its variables, so that a correct execution of a circuit is equivalent to finding a satisfying variable assignment (i.e., a solution of its R1CS representation). `circom` [5] is a low-level language where the programmer has to describe the statement as an R1CS and also provides an efficient way to obtain a satisfying assignment for given inputs.

Example 1. Let us consider the next `circom` program which defines an arithmetic circuit (as a *template*), which decides whether the input value is zero.

```
1    template IsZero() {
2      signal input in;
3      signal output out;
4      signal inv;
5      inv <-- in!=0 ? 1/in : 0;
6      out <== - in * inv + 1;
7      in * out === 0;
8    }
```

This template declares an input signal, `in`, an output signal, `out`, and an intermediate signal, `inv`. In Line 5, we set `inv` to be the inverse of `in` if `in` is not 0, and 0, otherwise. The use of `<--` does not any new constraint to the R1CS, since this computation is only used to obtain the satisfying assignment. Line 6 uses `<==` to add (a) the constraint `out === -in * inv + 1` to the R1CS and (b) the computation `out <-- -in * inv + 1` to obtain the satisfying assignment. Finally, Line 7 adds the constraint `in * out === 0` to the R1CS.

Let us consider $\mathbb{F}_p$, with $p = 11$. Two valid assignments for this circuit are $\{$`in` $\mapsto$ `2, inv` $\mapsto$ `6, out` $\mapsto$ `0`$\}$ and $\{$`in` $\mapsto$ `0, inv` $\mapsto$ `0, out` $\mapsto$ `1`$\}$.

Describing complex statements with conjunctions of quadratic constraints is not easy and programmers tend to make mistakes and write code whose semantics is not the intended. The interest of extending the language with HC arises in such complex cases, e.g., when we need to express conditional statements which are not easily encoded as conjunctions of quadratic constraints. In `circom`, in order to enhance generic descriptions, we can use "if-then-else" instructions to say which constraints belong to the circuit depending on a condition, but its evaluation should be known at compilation time, and therefore they cannot be used to encode a conditional relation among constraints. Still, programmers used to work with high-level languages like C++ and Java often use the "if-then-else" instruction erroneously as shown in the following example.

Example 2. Suppose that our statement requires to establish a relation between four input signals: `in`, `x`, `y`, and `z`, where `x` must be equal to the sum of `y` and `z` whenever `in` is equal to one. To this end, a non-experienced `circom` programmer (with a background on high-level languages) may easily write a code like this:

```
template WrongProgram() {
  signal input in, x, y, z;
  if( in === 1 )
       x === y  +  z;
}
```

This code is not accepted by `circom` as it is not asserting the conditional statement but saying that the constraint is in the circuit or not depending on a signal of the circuit itself. Thus, we would have a circuit that changes depending on the values of the wires.

In order to encode such conditional relation in `circom`, we need to do it in a much lower-level fashion (using only quadratic constraints). The following code is a correct program that expresses the statement.

Example 3. This example shows the template `ConditionalStatement`, which is a correct implementation of the previous example.

```
template ConditionalStatement() {
  signal input in, x, y, z;
  signal aux;
  component isz = IsZero();
  isz.in <== in  -  1;
  aux <== isz.out;
  aux * (x - y - z) === 0;
}
```

After declaring the four input signals and an intermediate signal `aux`, we declare the component `isz`, as an instance of template `IsZero`. Next, we set the input signal of `isz`, `isz.in`, to `in - 1`. Then, the intermediate signal `aux` is set to the output signal of `isz`, `isz.out`, which values 1 only if `in - 1 === 0` is satisfied, and 0, otherwise. Thus, we assert the constraint `aux * (x - y - z) === 0;` that guarantees that either `aux === 0` is satisfied (and then, `in - 1 === 0` is not satisfied), or `x - y - z === 0` is satisfied. All introduced constraints (using `===` and `<==`) are quadratic and describe the conditional example as a whole.

From the previous example, we realise that stating a Horn clause using only conjunctions of quadratic constraints is involved and furthermore the resulting constraints do not hint on the real meaning they have. We argue that adding "Horn clauses" to the language would add the expressiveness we need to declare such conditional statements without confusing the programmers in declaring forbidden conditionals as it happens in languages including "`if` statements".

Example 4. We use now the new `Horn block` instruction (and Prolog-like syntax for the implication) on the example. If the constraint `in === 1` is satisfied, then the constraint `x === y + z` must be also satisfied when generating the ZK proof.

```
template ConditionalStatement() {
  signal input in, x, y, z;
  Horn block{
      x === y + z :- in === 1;
  };
}
```

An additional benefit of having Horn blocks is that they will allow simplifications, what is key in the generation of the final R1CS that is used in the ZK-proof.

Regarding other clauses (non-Horn), where we have, for instance, $C_1 = 0 \vee C_2 = 0$, there is less interest in including them in the new format as, on one hand, are simpler to express with multiplication (e.g. with $C_1 \times C_2 = 0$), and on the other hand, and more importantly, they cannot be easily used for simplification purposes (even if C_1 and C_2 are linear). However, for efficiency purposes, it is interesting to consider a compact way to express conditional statements that include a condition and its complementary. Our idea is to use the Horn block instruction also to implement the negation in the else branch by setting a priority of evaluation of each Horn clause in the block according to order of appearance, i.e., if the constraints in the premise of the first Horn clause are satisfied, then the constraints in its conclusion must also be satisfied. However, if the premise is not satisfied, then the following Horn clauses must be satisfied.

Example 5. Suppose that if constraint `in === 1` is not satisfied, we want constraint `in === 2 * z` to be satisfied. We will implement these conditions in `circom` thanks to the new Horn blocks, as we can see in the next piece of code:

```
template IfThenElseStatement() {
  signal input in, x, y, z;
  Horn block{
     x === y + z :- in === 1;
     x === 2 * z :- _;
  };
}
```

If the constraint `in === 1` is satisfied, then the constraint `x === y + z` must be also satisfied when generating the ZK proof. However, if the constraint, `in === 1` is not satisfied, then the second Horn clause in the block is evaluated. Now, `_` is syntactic sugar for a tautology like `0 === 0`, which means that no constraint needs to be satisfied in the premise of the Horn clause. Thus, if constraint `in === 1` is not satisfied, constraint `x === 2 * z` must be satisfied.

References

1. Marta Bellés-Muñoz, Barry Whitehat, Jordi Baylina, Vanesa Daza, and Jose Luis Muñoz Tapia. Twisted edwards elliptic curves for zero-knowledge circuits. *Mathematics*, 9(23), 2021.
2. Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.
3. Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. Association for Computing Machinery.
4. Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
5. Iden3. Circom: Circuit compiler for zero-knowledge proofs. GitHub, 2020. Available online: `https://github.com/iden3/circom` (accessed on 15 December 2021).
6. Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252, 2013.