J-R1CS – a JSON Lines format for R1CS

Guillaume Drevon
gd@sikoba.com

Sikoba Research *

6 April 2019 - version 0.6

Abstract

R1CS (rank-1 constraint systems) define a set of bi-linear equations which serve as constraints suitable for ZK proofs. They describe the execution of statements written in higher-level programming languages and are used by many ZK proof applications, but there is as yet no standard way of representing them. In this paper, we present J-R1CS, a simple and lightweight *JSON Lines* format dedicated to R1CS.

This document was written in response to the call for Community Standards of the 2nd ZKProof Standards Workshop, under the Implementation topic.

1 Background and Motivation

R1CS is defined in [BCG+13] as a simpler reformulation of QAP (Quadratic Arithmetic Program) that were used previously. R1CS has de-facto become the low-level language for representing statements to be processed by ZK proof systems.

The zkproof community has started to work on a file format for R1CS in the implementation track of the first workshop [Imp]. However this binary file format is not practical for prototyping and quick developments.

Using the definitions introduced in the above mentioned workshop, we propose a *JSON Lines* format which has advantages in term of usability, readability and interopability, making it a good choice for pre-production usages. Indeed, JSON parsers/deserializers exist in many programming languages and JSON has become a widely used format for describing data for both humans and machines, due to its simplicity and compactness.

*Supported by Fantom Foundation

2 Model

R1CS can be easily described with a *JSON Lines* format [Jso]. Since a R1CS can contain a very large number of equations, it would not be practical to have all them included in one JSON object because then the parsing would load everything in memory at once – this is the drawback of a format proposed in [whi]. Of course one could use a specific parser, but that would defeat the advantages of using JSON.

The first line is a header describing the properties of the system. Listing 1 provides an example of the R1CS header in JSON format. For readability it is shown with line breaks, but in an J-R1CS file it should be all in one line.

```
{
   "r1cs":{
      "version":"1.0",
      "field_characteristic":"133581199851807797997178235848527563401",
      "extension_degree":1,
      "instances":3,
      "witnesses":5,
      "constraints":2000,
      "optimized":true
   }
}
```

Listing 1: JSON R1CS Header

Then we simply list every R1C line by line. As before, line breaks in Listing 2 are for readability only, the actual R1C line should not have any line breaks.

```
{
   {
      "A": [[0,"2"],[-1,"6"],[5,"4"],....],
      "B": [[0,"5"],[-6,"3"],[3,"4"],....],
      "C": [[0,"3"],[-1,"2"],[-2,"7"],[3,"4"],[4,"1"]...]
   }
}
```

Listing 2: JSON R1C

The corresponding R1C is $\langle A, X\rangle \cdot \langle B, X\rangle = \langle C, X\rangle$, where A, B and C are vectors of coefficients and X is the vector of variables (constant, instances and witnesses). In the JSON object, instead of putting all the coefficients for A,B and C, we add their index which allows to omit zero coefficients in the representation. Indeed in practice, there will be many variables in the system but only a few within one constraint. We follow the convention in [Imp] for the indexes; zero represents a constant, a negative index indicates an instance variable and positive index, a witness variable.

$$\begin{cases} index = 0 & \text{constant input} \\ index < 0 & \text{index of the instance inputs} \\ index > 0 & \text{index of the witness inputs} \end{cases} \tag{1}$$

3 Assignments

When using this format while working on the isekai project, I realized that R1CS are strongly linked to assignments, i.e a solution of the system. Indeed, assignments are required for generating a ZK proof, but in practice the assignments are computed along with the constraint systems. If you do not compute them at the same time, you will need to solve the constraint system which is a much harder task. The inputs and witnesses values can easily be stored as JSON arrays.


```
{
  "inputs":["0","1","0","1","3","8","5","4","2","1","9","1","8","1","0",
  "1","3","8","5","4","2","1","14757395258967641292 9","9"],

  "witnesses": ["1","1","1","0","0","0","1","0","0","0","0","0","0","0",
  "0","0","0","0","0","0","0","0","0","0","0","0","0","0","0","0",
  "0","0","9","0","0","0","0","0","0","0","3","0","4","0","0","4","0",
  "4","0","4","0","4","9","0","0","1","24320269857599194691384895272508 0
  834317204048893511492707757824295286761062 4","0","1","0","1","27360303
  589799094027808007181571593860685455500520042929622755233219760619 52",
  "0","0","0","1","18761351033005093047639776353077664361612883771785172
  29459846073135069299624 3","0","14757395258967641292 9","0"]
}
```


Listing 3: Assignments in Json Format

Such a JSON object can be easily added to an J-R1CS file because this format is already a list of JSON objects. Note that one should make sure not to reveal the witnesses values, else you do not have zero-knowledge anymore. So these parameters (inputs and/or witnesses) are optional and should be present only when creating the R1CS.

4 Examples

It can be interesting to see some examples. We start with a simple XOR circuit coming from a unit test of str4d's rust implementation of the ZKProof binary format [str]. This will allow us to compare with the binary format.


```
      {"r1cs":{"constraint_nb":3,
               "extension_degree":1,
               "field_characteristic":64513,
               "instance_nb":1,
               "version":"1.0",
               "witness_nb":2 }}
      {"A":[[0,"1"],[1,"64512"]],"B":[[1,"1"]], "C":[]}
      {"A":[[0,"1"],[2,"64512"]],"B":[[2,"1"]], "C":[]}
      {"A":[[1,"2"]],"B":[[2,"1"]],"C":[[-1,"64512"],[1,"1"],[2,"1"]]}
```


Listing 4: XOR Circuit in JSON R1C

```
R1CS XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Listing 5: XOR Circuit in Binary R1CS

The binary data cannot be rendered, you can find the base 64 encoding below. The J-R1CS file is 293 bytes size while the binary format is 48 bytes. However we cannot really extrapolate from such a small example.

```
UjFDUwAEgvAHAgIEAwIAAgIBAQICAQAAAgACBAEBBAIBAAABAgQBBAIDAQECAgQC)
```

Listing 6: XOR Circuit in Binary R1C Base 64

I have also generated with [ise] the Pinocchio's auction circuit [PGHR13] in J-R1CS format and it gives a 428 kb file. As expected this file can be compressed very well. 7zip with default options (compression level: normal) compressed it into a 14.8 kbytes file.

```
{"A":[[2464,"1"]],"B":[[2399,"1"],[2400,"1"]],"C":[[2530,"1"]]}

{"A":[[12,"1"],[2464,"21888242871839275222246405745257275088548364400416034
34343698204186575808495616"]],"B":[[2397,"1"],[2398,"1"]],
"C":[,[2531,"1"]]}

{"A":[[2464,"1"]],"B":[[11,"1"]],"C":[[2532,"1"]]}

{"A":[[12,"1"],[2464,"21888242871839275222246405745257275088548364400416034
34343698204186575808495616"]],"B":[[2399,"1"],[2400,"1"]],
"C":[[2533,"1"]]}

{"A":[[88,"1"]],"B":[[12,"0"]],"C":[[2534,"1"]]}

{"A":[[12,"1"],[88,"21888242871839275222246405745257275088548364400416034
34369820418657580849561 6"]],"B":[[12,"0"]],"C":[[2535,"1"]]}

{"A":[[352,"1"]],"B":[[12,"1"]],"C":[[2536,"1"]]}

{"A":[[12,"1"],[352,"21888242871839275222246405745257275088548364400416034
34369820418657580849561 6"]],"B":[[2534,"1"],[2535,"1"]],
"C":[,[2537,"1"]]}

{"A":[[616,"1"]],"B":[[12,"2"]],"C":[[2538,"1"]]}

{"A":[[12,"1"],[616,"21888242871839275222246405745257275088548364400416034
34369820418657580849561 6"]],"B":[[2536,"1"],[2537,"1"]],
"C":[[2539,"1"]]}
```

Listing 7: Auction Circuit in J-R1CS format (small extract)

5 Limitations and Improvements

5.1 Big integers

Although in theory JSON does not limit numbers, in practice number size will be limited by the JSON parser implementations, see [rfc] for more information. This is why we are using strings for

the coefficients which will need to be handled as big integers in the ZK proof application itself.

5.2 Order of R1C vectors

By default, J-R1CS does not require the elements of the R1C vectors to be ordered by index, as suggested in [Imp]. However, ordering these elements is definitely a good practice so it can be enforced by setting the "optimized" property to true.

5.3 Variable names

In the future, there may be DSLs which will allow to identify variables by name. In some situations, it might be useful to easily identify where in the R1CS these variables appear. We could therefore add the (optional) possibility of adding names for variables.

```
{
   "names": [[-1,"IN_A"],[-2,"IN_B"],.... [1,"W_A"],[2,"W_B"],.... ],
}
```

Listing 8: Optional variables names

5.4 Modularization

The proposed file format describes fully a standalone constraint system but it is not easy to combine several of them. This could be an interesting property if you have several frontend modules producing pieces of R1CS. In that case, using variable names could be a way to integrate several R1CS.

6 Conclusion

The format described in this document benefits from all the advantages of JSON, namely;

Extensibility - It is very easy to add parameters in JSON. For instance one could specify the base to use as a new parameter in the header file if you need to represent the coefficient in a different base. Indeed, libSnark and Pinocchio require hexadecimal input values.

Implementation - JSON is a very popular format and JSON parsers exists for almost any programming language. In Annex A we provide a sample implementation using the JSON for Modern C++ class.

Interoperability - JSON is widely used for modern communications between IT systems, such as Web API or Web services.

Readability - The format is readable for a human making it efficient for prototyping and debugging

Lightweight - Contrary to XML, JSON is not a verbose format. We have also shown that it compresses very well in practice. One should also note that most http layers provide transparent compression.

Scalability - The JSON line format allows us to use one JSON object for each constraint making it straightforward to load the constraints one-by-one and avoiding to load the entire system in memory. This means this format can scale very easily.

In conclusion, we believe it is a simple but powerful format for representing R1CS. This data being at the core of ZK proof systems, it is intended to be produced by many frontends and consumed by many different backends so interoperability and readability are crucial characteristics. At the same time, ZK proofs are resource intensive and we must not have negative impact on performance, so scalability and small size are also important aspects.

References

[BCG+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for C: verifying program executions succinctly and in zero knowledge. In *CRYPTO (2)*, volume 8043 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2013.

[Imp] Implementation track. `https://zkproof.org/documents.html`. zkproof-implementation-20180801.pdf.

[ise] isekai. `https://github.com/sikoba/isekai`.

[Jso] Json lines. `http://jsonlines.org/`.

[lib] libsnark. `https://github.com/SCIPR-Lab/libsnark`.

[PGHR13] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *IACR Cryptology ePrint Archive*, 2013:279, 2013.

[rfc] Rfc7159. `https://tools.ietf.org/html/rfc7159#section-6`.

[str] R1cs binary format. `https://github.com/str4d/zk/tree/r1cs-file`.

[whi] libsnark-tutorial. `https://github.com/barryWhiteHat/libsnark-tutorial/blob/master/zksnark_element/r1cs.json`.

Annex A - Implementation

Below is a C++ sample implemenation loading and saving the J-R1CS format from/to LibSnark constraint system objects [lib].

```
////////////////////////////////////
//Helper to convert coefficient to string
std::string FieldToString(FieldT cc)
{
    mpz_t t;
    mpz_init(t);
       cc.as_bigint().to_mpz(t);
    mpz_class big_coeff(t);    // As recommended by GMP library;
                               // cf. https://gmplib.org/manual/Converting-Integers.html
    return big_coeff.get_str();
}

json LinearCombination2Json(linear_combination<FieldT> vec)
{
    json jc;
    json jlt;

    for (linear_term<FieldT> const & lt:vec)
    {
        //TODO: Check the coefficient is not null
        json jlt;
        jlt.push_back(lt.index);                //TODO handle neg.idx
        //Convert the coefficient to string
        jlt.push_back(FieldToString(lt.coeff));

        jc.push_back(jlt);
    }
    return jc;
}

linear_combination<FieldT> parseLinearCombJson(json &jlc)
{
    linear_combination<FieldT> lc;
    for (auto const& term : jlc)
    {
        variable<FieldT> var(term[0]);
        std::string str_coeff = term[1];
        FieldT cc(str_coeff.c_str());

        lc.add_term(var, cc);

    }
    return lc;
}
```

Listing 9: Sample C++ implementation (part 1)

```
bool ToJsonl(r1cs_constraint_system<FieldT>  &in_cs, const std::string &out_fname)
{
    //convert r1cs to jsonl file
    json r1cs_header;
    r1cs_header["version"] = "1.0";
    r1cs_header["extension_degree"] = 1;
    r1cs_header["instance_nb"] = in_cs.primary_input_size;
    r1cs_header["witness_nb"] = in_cs.auxiliary_input_size;
    r1cs_header["constraint_nb"] = in_cs.num_constraints();
    r1cs_header["field_characteristic"] = 1;
    //write to file
    std::ofstream o(out_fname);
    if (!o.good())
        return false;
    json j;
    j["r1cs"] = r1cs_header;
    o << j << std::endl;
    //write constraints
    for (r1cs_constraint<FieldT>& constraint : in_cs.constraints)
    {
        json jc;
        jc["A"] = LinearCombination2Json(constraint.a);
        jc["B"] = LinearCombination2Json(constraint.b);
        jc["C"] = LinearCombination2Json(constraint.c);
        o << jc << std::endl;
    }
    o.close();
    return true;
}

bool FromJsonl(const std::string jsonFile, r1cs_constraint_system<FieldT> &out_cs)
{
    //read from file
    std::ifstream r1cs_file(jsonFile);
    if (!r1cs_file.good())
        return false;

    std::string line;
    json header;
    int i = 0;
    while (std::getline(r1cs_file, line))
    {
        json jc = json::parse(line);

        if (jc.count("r1cs") > 0)
        {
              // header
            header = jc["r1cs"];
        }
        else
        {
            //constraint
            linear_combination<FieldT> A = parseLinearCombJson(jc["A"]);
            linear_combination<FieldT> B = parseLinearCombJson(jc["B"]);
            linear_combination<FieldT> C = parseLinearCombJson(jc["C"]);
            r1cs_constraint<FieldT> constraint(A,B,C);
            out_cs.add_constraint(constraint);
        }
      }
    out_cs.primary_input_size = header["instance_nb"];
    out_cs.auxiliary_input_size = header["witness_nb"];

    return true;
}
```

Listing 10: Sample C++ implementation (part 2)

Annex B - Definitions

R1CS Rank 1 Constraint Systems. It is a NP-complete language for specifying relations, as a system of bilinear constraints.

version The version number of the R1CS JSON format. It is a string, its typical format will be "Major.minor".

field_characteristic The characteristic of the underlying field in which the coefficients belong to.

extension_degree Optional property, it is the degree of the field extension. Current ZK proof systems all use 1. Default value is 1.

instances The number of instance variables, corresponding to the public inputs.

witnesses The number of witness variables, corresponding to the private inputs and auxiliary variables.

constraints The number of constraints (of rank 1).

optimized Optional property which indicates if the vectors of the constraints are ordered by index and contain no 0 coefficients. The default value is "false". When set to "true", all indices inside a vector must be unique and ordered, and the coefficients cannot be 0. The ordering must be as follows: first index 0, then the negative indices (in decreasing order: -1, -2, ...), finally the positive indices (in increasing order).