zkp-toolkit Gadget list

> This article mainly lists the gadgets that the zkp-toolkit library has implemented in the first stage. In the implementation process, the code of bellman, libsnark and ethsnarkgadget is referenced. The code of each gadget is located in the `zkp-toolkit/src/gadget` folder. This article mainly briefly analyzes the implementation of each gadget.

gadget list

√ rangeproof

√ isnonzero

√ lookup_1bit

√ lookup_2bit

√ lookup_3bit

√ merkletree

√ boolean

√ mimc

gadget detailed design

zkp-toolkit gadget writing

In zkp-toolkit, there is a trait called ConstraintSynthesizer. To complete the verification in groth16, it is necessary to inherit this trait. Under this trait, there is a function `generate_constraints`, which implements the structure of the circuit, with or without substitution The witness construction circuit is implemented by this function, so it will be used twice, once when the setup construction circuit, the witness is set to None, and once when the proof is generated and substituted into the witness calculation.

```
/// Computations are expressed in terms of rank-1 constraint systems (R1CS).
/// The `generate_constraints` method is called to generate constraints for
/// both CRS generation and for proving.
pub trait ConstraintSynthesizer<F: Field> {
    /// Drives generation of new constraints inside `CS`.
    fn generate_constraints<CS: ConstraintSystem<F>>(
        self,
        cs: &mut CS,
    ) -> Result<(), SynthesisError>;
}
```

After implementing the circuit in the `generate_constraints` function implemented above, you need to test the existing circuit. The general test steps are as follows:

1. Generate pvk from the generated random value and empty circuit.

```
let mut rng = &mut test_rng();
    let n = 10u64; // range 0 ~ 2^10

    println!("Creating parameters...");
    let params = {
        let c = RangeProof::<Fr> {
            lhs: None,
            rhs: None,
            n: n,
        };

        generate_random_parameters::<Bn_256, _, _>(c, &mut rng).unwrap()
    };

    let pvk = prepare_verifying_key(&params.vk);
```

2. Bring withness into the circuit instance to generate proof.

```
    println!("Creating proofs...");

    let c1 = RangeProof::<Fr> {
        lhs: Some(Fr::from(24u32)),
        rhs: Some(Fr::from(25u32)),
        n: n,
    };

    let proof = create_random_proof(c1, &params, &mut rng).unwrap();
```

3. Use vk, proof, and public input to verify that the proof is correct.

```
    println!("Proofs ok, start verify...");

    assert!(verify_proof(&pvk, &proof, &[Fr::from(2u32).pow(&[n])]).unwrap());
```

The rangeproof test sample is as follows:

```
#[test]
fn test_rangeproof() {
    use curve::bn_256::{Bn_256, Fr};
    use math::fields::Field;
    use math::test_rng;
    use scheme::groth16::{
        create_random_proof, generate_random_parameters, prepare_verifying_key,
verify_proof,
    };

    let mut rng = &mut test_rng();
    let n = 10u64; // range 0 ~ 2^10

    println!("Creating parameters...");
    let params = {
        let c = RangeProof::<Fr> {
            lhs: None,
            rhs: None,
            n: n,
        };
```

```
        generate_random_parameters::<Bn_256, _, _>(c, &mut rng).unwrap()
    };

    let pvk = prepare_verifying_key(&params.vk);

    println!("Creating proofs...");

    let c1 = RangeProof::<Fr> {
        lhs: Some(Fr::from(24u32)),
        rhs: Some(Fr::from(25u32)),
        n: n,
    };

    let proof = create_random_proof(c1, &params, &mut rng).unwrap();
    println!("Proofs ok, start verify...");

    assert!(verify_proof(&pvk, &proof, &[Fr::from(2u32).pow(&[n])]).unwrap());
}
```

rangeproof

Features

Compare the size of two variables (lhs, rhs)

Constraint

- Calculate alpha_packed = 2 ^ n + B-A, in order to verify that the two are equal, add a circuit

  `1 * (2 ^ n + B-A) = alpha_packed`

- The alpha array represents the binary representation of alpha_packed, in order to verify that the two are equal, add a circuit

  `1 * sum (bits) = alpha_packed`

- In order to verify that each bit of the alpha array is binary

  `(1-bits_i) * bits_i = 0`

  Calculate sum = ∑ bits_i (i = 0..n-1)

  When sum = 0, it means that lhs = rhs, let output = 0, inv = 0

  When sum! = 0, it means lhs! = rhs, let output = 1, inv = 1 / sum, output (that is, not_all_zeros) indicates whether two numbers are equal.

- To ensure that the output is a binary number, verify

  `(1-output) * output = 0`

- Constrain the relationship between output and sum, there are two equality constraints, when sum is not 0, output must be 1

  `(1-output) * sum = 0`

- When sum = 0, output must also be 0

  `inv * sum = output`

- less_or_eq is alpha [n], so the relationship between less_or_eq and less can be expressed as

  `less_or_eq * output = less`

- Verify whether the relationship is established: `less * 1 = 1`

Code

rangeproof

isnonezero

Features

Determine if variable value is non-zero

Constraint

There are two constraints in the gadget in ethsnark. X is the input value and Y is the output value. The two constraints are:

X * (1-Y) = 0

X * (1 / X) = 0

There is no result output variable Y in zkp-toolkit, so the constraints are:

$X * (1/X) = 0$

Code

isnonzero

lookup_1bit

Features

The binary value of one bit of b is used as the value in the subscript range array C, and the result value is assigned to r.

Constraint

Input bit b, variable array c, result value r

$(c[0] + b * c[1] - (b * c[0])) * 1 = r$

When b is 0, c [0] = r

When b is 1, c [1] = r

Code

lookup_1bit

lookup_2bit

Features

The binary value of the two bits of b is used as the value in the subscript range array C, and the resulting value is assigned to r.

Constraint

Input 2bit b, variable array c, result value r

$(c[1]-c[0]+(b[1]*(c[3]-c[2]-c[1]+c[0])))*b[0]=-c[0]+r+(b[1]*(-c[2]+c[0]))$

The above constraints refer to the wording in ethsnark. If the above constraint equation holds, it must be satisfied:

b: 00 r = c [0]

b: 01 r = c [1]

b: 10 r = c [2]

b: 11 r = c [3]

Code

lookup_2bit

lookup_3bit

Features

The binary value of the three bits of b is used as the value in the subscript range array C, and the result value is assigned to r.

Constraint

$$
\begin{aligned}
&(c[0]+\\
&(b[0]*-c[0])+\\
&(b[0]*c[1])+\\
&(b[1]*-c[0])+\\
&(b[1]*c[2])+\\
&(b[0]*b[1]*(-c[1]+-c[2]+c[0]+c[3]))+\\
&(b[2]*(-c[0]+c[4]))+\\
&(b[0]*b[2]*(c[0]-c[1]-c[4]+c[5]))+\\
&(b[1]*b[2]*(c[0]-c[2]-c[4]+c[6]))+\\
&(b[0]*b[1]*b[2]*(-c[0]+c[1]+c[2]-c[3]+c[4]-c[5]-c[6]+c[7])))*1=r
\end{aligned}
$$

The above constraints refer to the implementation of ethsnark. If the above constraint equation holds, it must be satisfied:

b: 000 r = c [0]

b: 001 r = c [1]

b: 010 r = c [2]

b: 011 r = c [3]

b: 100 r = c [4]

b: 101 r = c [5]

b: 110 r = c [6]

b: 111 r = c [7]

Code

lookup_3bit.rs

merkletree

Features

Given a verification path of a merkletree, and a leaf node and root node. Verify that the leaf node's calculation result on the verification path is the same as the expected root node.

Constraint

There are 3 types of constraints in merkletree

- merkletree calculates the constraints of each left_digests and right_digests on the path. Every bit on the hash needs to satisfy $digests[i] * (1 - digests[i]) = 0$ so the total number of constraints here is $(2 * digest\_size - 1) * tree\_depth$.
- A hash constraint from the lowest level of merkletree to the root node tree_depth (where tree_depth is the actual tree height minus 1). Assuming each hash constraint x, the total constraint is $tree\ _depth * x$
- Determine the left and right positions of the leaf hash node and the internal hash node by the bit of address_bit. The constraint of each hash bit is: $is\_right * (right.bits[i] - left.bits[i]) = (input.bits[i] - left.bits[i])$ digest_size Constraint, looping tree_depth altogether. Numer of constraints: $digest\_size * tree\_depth$.
- Finally, the hash result calculated by the circuit is compared with the expected hash result. The circuit constraints are: $root\_digest[i] * 1 = computed\_root[i]$ digests_size constraints.

Code

merkletree.rs

boolean

Features

boolean logic gadget (this part mainly refers to bellman / gadget / boolean.rs), there are various gadget operations on boolean variables, Boolean is the encapsulation of AllocatedBit.

Constraint

A total of gadgets for bool operations are implemented:

```
AllocatedBit:
xor: Performs an XOR operation over the two operands
and: Performs an AND operation over the two operands
and_not Calculates: a AND (NOT b)
nor: Calculates (NOT a) AND (NOT b)
u64_into_boolean_vec_le: u64 to Vec<Boolean>
field_into_boolean_vec_le: Field to Vec<Boolean>
field_into_allocated_bits_le: Vec<AllocatedBit>
```

Code

boolean.rs

mimc

Features

mimc hash function

Constraint

xL, xR: = xR + (xL + Ci) ^ 3, xL

tmp = (xL + Ci) ^ 2

new_xL = xR + (xL + Ci) ^ 3

new_xL = xR + tmp * (xL + Ci)

new_xL-xR = tmp * (xL + Ci)

Constraint: MIMC_ROUNDS round `new_xL = xR + (xL + Ci) ^ 3` constraints

Code

mimc.rs