Introduction of zk-SNARK

1 Introduction

In cryptography, a zero-knowledge proof or zero-knowledge protocol is a method by which one party (the prover) can prove to another party (the verifier) that they know a statement, without reveal any information apart from the statement itself. The essence of zero-knowledge proofs is that it is trivial to prove that one possesses knowledge of certain statement by simply revealing it; the challenge is to prove such possession without revealing the knowledge or any additional knowledge which can be deduced from the provided information.

There are two types of zero-knowledge proof system, one is interactive zero-knowledge proofs and the other is non-interactive zero-knowledge proofs.

Interactive zero-knowledge proofs require interaction between the individual (or computer system) proving their statement using their knowledge and the another individual validating the proof. A protocol implementing zero-knowledge proofs of knowledge must necessarily require interactive input is usually in the form of one or more challenges such that the responses from the prover will convince the verifier if and only if the statement is true, i.e., if the prover does possess the claimed knowledge. If this were not the case, the verifier could record the execution of the protocol and replay it to convince someone else that they possess the secret information. The new party's acceptance is either justified since the replayer does possess the information (which implies that the protocol leaked information, and thus, is not proved in zero-knowledge), or the acceptance is spurious, i.e., was accepted from someone who does not actually possess the information.

The other type of zero-knowledge proofs is Non-interactive zero-knowledge proofs (NIZK). This paper will be concentrate on this type of zero-knowledge proofs, especially zk-SNARK (Zero-Knowledge Succinct Non-interactive Arguments of Knowledge).

Zero-knowledge proofs are advantageous in a myriad of application, including:

- Proving statement on private data, i.e., DNA matching without revealing full DNA
- Anonymous authorization
- Anonymous payments, i.e., paying taxes without revealing one's earning

- Outsourcing computation, i.e., blockchains guarantee that transactions are valid without revealing the information about recipient

For any zero-knowledge proof system, they should satisfy three properties:

- Completeness - if the statement is true then a prover can convince a verifier
- Soundness - a cheating prover can not convince a verifier of a false statement
- Zero-knowledge - the interaction only reveals if a statement is true and nothing else

By the way, there also are more detailed definition about the properties like computational soundness and perfect soundness, this paper would not talk about that. Nowadays, most implementation of zk-SNARK protocols are based on [Groth16].

2 Relative Definition

2.1 Definition 1 (Quadratic Span Program)

A quadratic span program Q over field F contains two sets of polynomials:
$V = \{v_k(x) : k \in \{0, ..., m\}\}$ and $W = \{w_k(x) : k \in \{0, ..., m\}\}$
and a target polynomials $t(x)$, all from $F[x]$. Q also contains a partition of the indices $I = \{1, ..., m\}$ into two sets $I_{labeled}$ and I_{free}, and a further partition of $\bigcup_{i\in[n],j\in\{0,1\}} I_{ij}$.

For input $u \in \{0,1\}^n$, let $I_u = I_{free} \bigcup_i I_{i,u_i}$ be the set of indices that "belong" to input u. Q accepts an input u iff there exist tuples $(a_1, ..., a_m)$ and $(b_1, ..., b_m)$ from field F^m, with $a_k = b_k = 0$ for all $k \notin I_u$, such that

$$t(x) \text{ divides } (v_0(x) + \sum_{k=1}^{m} a_k \cdot v_k(x)) \cdot (w_0(x) + \sum_{k=1}^{m} w_k(x)) \quad (1)$$

Q "computes" a Boolean function $f : \{0,1\}^n \to \{0,1\}$ if it accepts exactly those inputs $u \in \{0,1\}$ where $f(u) = 1$. The size of Q is m. The degree of Q is $deg(t(x))$.

2.2 Definition 2 (Strong homomorphic Encryption)

A homomorphic encryption using modular arithmetic can be represented as $E(v) = g^v(modn)$, where v is the data plaintext and need to be encrypted. Then we can get the definition of the computation:

- Plus: $E(s_0) \bigoplus E(s_1) = E(s_0 + s_1) = g^{s_0} + g^{s_1} = g^{s_0+s_1}$
- Multiply: $E(s)^t = E(ts) = (g^s)^t = g^{ts}$

Remark: Even we can multiply two encrypted value by two unencrypted value, we cannot multiply or divided two encrypted value directly, also means we cannot exponentiate an encrypted value.

2.3 Definition 3 (Knowledge-of-Exponent Assumption)

The knowledge of exponent (KEA) assumption introduced by Damgard [Dam91] says that given g, g^α it is infeasible to create to create c, c', so $c' = c^\alpha$ without knowing α so $c = g^\alpha$ and $c' = (g^\alpha)^\alpha$.

In other words, assume g is the generator of group G, which the order $|G|$ of group G is a prime number. The security parameter is $k = log(|G|)$. For all probabilistic polynomial time (PPT) algorithm A with input g and g^α, it would generate a pair of parameter $(x, y), x \in G$. Also, there is another PPT executor E with input g and g^α, and then it will also get the output (x, y). Apart from that, it will generate a g^r such that for a sufficiently large k, we have

$$Pr[y = x^a \text{ and } g^r \neq x] < \frac{1}{Q(k)} \text{for any polynomials Q} \tag{2}$$

In zk-SNARK, KEA is mainly used to generate a pair of α-shifts like (g, g^α) to prove that the prover performed a "shift operation" on the same encryption result without tampering. E.g. for a simple polynomial $f(x) = c \cdot x$, verifier want to ensure that prover would only "multiply c" on ciphertext by giving a pair of $(g^s, g^{s\alpha})$ with a random field element s to prover. Then prover should provide $((g^s)^c, (g^{\alpha s})^c)$ to prove that it has done the multiply operation on the same encryption result.

2.4 Definition 4 (Bilinear Groups)

Bilinear groups consists of $(G_1, G_2, P, G_T, g, e, h)$, with the following properties:

- G_1, G_2, G_T are groups of prime order p
- The pairing $e : G_1 \times G_2 \to G_T$ is a bilinear map
- g is a generator for G_1, h is a generator for G_2, and $e(g, h)$ is a generator for G_T
- There are efficient algorithms for computing group operations, evaluating the bilinear map, deciding membership of the groups, deciding equality of group elements and sampling generators of the groups. We refer to these as the generic group operations.

2.5 Definition 5 (NIZK Proof System)

A NIZK proof system for input x in language L, with witness w, which also means knowledge behind the x, is a set of efficient PPT algorithms (S, P, V) such that:

1. Setup: $\sigma \leftarrow S(1^k)$ generates the common reference string (CRS), where k is a security parameter.
2. Prover: $\pi \leftarrow P(\sigma, x, \omega)$ produces the proof
3. Verifier: $V(\sigma, x, \pi)$ outputs $\{0, 1\}$ to accept/reject the proof

3 Proof Construction of QSP

3.1 Getting Started

If a prover claims to know the factorization of a polynomials, how does verifier checks that? To prove the correctness of a single operation, we must enforce the correctness of the output for the operands provided. It's easy to abstract an operation as

$$\textit{left operand} \;\; \textbf{operator} \;\; \textit{right operand} \;=\; \textit{output} \tag{3}$$

The same can be represented as an operation polynomial:

$$l(x) \;\textbf{operator}\; r(x) = o(x) \tag{4}$$

where for some chosen a:

- l(x) - at a represents the value of the left operand
- r(x) - at a represents the value of the right operand
- o(x) - at a represents the result of the operation

Therefore if all operands and operator are represented correctly, then the equation $l(a)\textbf{operator}r(a) = o(a)$ should hold. After moving $o(a)$ to left, then we can get that a can be the root of the function $l(x)\textbf{operator}r(x) - o(x) = 0$. Because we know one of the root of the function, we can reconstruct the function to another form and it must contain cofactor $(x - a)$, which is named as *target polynomial* $t(x)$.

For example, let $l(x) = 3x, r(x) = 2x, o(x) = 6x$, so we get the operation $\times$ polynomial $3x \times 2x = 6x \rightarrow 6x^2 - 6x = 0$. Of course, 1 can be a root of this polynomial, which means the target polynomial $t(x) = x-1$, then we can reform the polynomial as $t(x) \cdot h(x) = (x - 1) \cdot 6x = 6x^2 - 6x = l(x) \times r(x) - o(x)$.

So, if the prover provides such polynomials to verifier, and verifier would accept it as valid, since $t(x)$ can be divided by $l(x) \times r(x) - o(x)$. On the contrary, if the prover is not honest and distort the $o(x)$ to $o(x) = 4x$, then the operation polynomial would be $6x^2 - 4x$ and it cannot be divisible by $t(x)$, so the verifier would not accept it.

3.2 Basic Model

3.2.1 Trusted Party Setup

Let us assume that we trust a single honest third party to setup the proof system, which means it will generate secrets s and α. As soon as α and all necessary powers of s with corresponding α-shifts are encrypted $(g^{\alpha}, g^{s^i}, g^{\alpha s^i}$ for i in (0, 1, ..., d), where $d = deg(t(x))$. Finally, the raw value (also called toxic waste) s, α must be deleted to avoid leakage.

These parameters are usually referred to as CRS. After CRS is generated any prover and any verifier can use it to conduct NIZK protocol. By the way, the optimized version of CRS will include encrypted evaluation of the target polynomial $g^{t(s)}$.

Moreover CRS is divided into two groups (Proving Key, Verification Key).

- Setup
  - sample random s and α
  - compute g^{α} and g^{s^i}, $g^{\alpha s^i}$, $g^t(s)$ for $i \in (0, ..., d), d = deg(t(x))$
  - Proving Key: $(g^{s^i}, g^{\alpha s^i})$
  - Verification Key: $(g^{t(s)}, g^{\alpha})$

3.2.2 Proof and Verification of Operation

First, we can easily find that prover can simply send the proof of $p(x) = l(x)\textbf{operator}r(x) - o(x)$ rather than sending three polynomials. However, there are two counterargument. On the one hand, in our protocol, the multiplication of encrypted values (i.e. $l(s) \times r(s)$) is not possible in the proving stage, since pairings can only be used once and it is required in the verification stage for polynomial restrictions check. On the other hand, this would leave an opportunity for the prover to modify the structure of polynomial at will but still maintain a valid cofactor $t(x)$, e.g., $p(x) = l(x)$ or $p(x) = l(x) - r(x)$ or even $p(x) = l(x) \times r(x) + o(x)$, as long as $p(x)$ has root α. Such modification effectively means that the proof is about a different statement, which is certainly not desired.

This is the reason that the evaluation of three polynomials have to be provided separately by the prover. From the verifier's view, it must check in encrypted space that $l(s) \times r(s) - o(s) = t(s)h(s)$. While a verifier can perform multiplication in encrypted space using bilinear maps, the subtraction $(-o(s))$ is an expensive operation, so we move $o(x)$ to the right side of the equation, which means the verifier need to check the equation:

$$l(x)r(x) = t(x)h(x) + o(x) \tag{5}$$

Then we can get the basic model of proof system.

- Prover
  - assign corresponding coefficients to the $l(x), r(x), o(x)$
  - compute polynomial $h(x) = \frac{l(x)r(x)-o(x)}{t(x)}$
  - evaluate encrypted polynomials $g^{l(s)}, g^{r(s)}, g^{o(s), g^{h(s)}}$ using $\left\{g^{s^i}\right\}_{i\in(0,\dots,d)}$
  - evaluate encrypted polynomials $g^{\alpha l(s)}, g^{\alpha r(s)}, g^{\alpha o(s)}$ using $\left\{g^{\alpha s^i}\right\}_{i\in(0,\dots,d)}$
  - set proof
    $\pi = (g^{l(s)}, g^{r(s)}, g^{o(s)}, g^{\alpha l(s)}, g^{\alpha r(s)}, g^{\alpha o(s)}, g^{h(s)})$
- Verifier
  - parse proof $\pi as(g^l, g^r, g^o, g^{l'}, g^{r'}, g^{o'}, g^h)$
  - polynomials restriction check:
    $e(g^l, g^\alpha) = e(g^{l'}, g)$
    $e(g^r, g^\alpha) = e(g^{r'}, g)$
    $e(g^o, g^\alpha) = e(g^{o'}, g)$
  - valid operation check: $e(g^l, g^r) = e(g^{t(s)}, g^h) \cdot e(g^o, g)$

Note: at this stage, this protocol still not have the property of zero-knowledge

3.3 Expanded Proof System - General Computation

According to the definition of α-shifts and the definition of bilinear maps, we can derive that α-shift could not only be usable to a single operation polynomial, but also be usable to the combination of multiple polynomials, called *variable polynomial*, e.g., the polynomial $L(s) = al_a(s) + bl_b(s)$ consists of two polynomials $l_a(s)$ and $l_b(s)$. While at the verification stage, we can use the pairing to check the structure of the $L(s)$ from the equation $e(g^{L(s)}, g^\alpha) = e(g^{\alpha L(x)}, g) = g^{\alpha al_a(s)+\alpha bl_b(s)}$. That is to say, α-shift can preserve a polynomial's structure after encryption.

For the above polynomial $L(s)$, coefficients (a, b) are its input. When it comes into general situation, we can do expansion to polynomials $l(x), r(x), o(x)$ while input variable values are $v = v_i, i \in (1, ..., n)$. Then we have $L(s) = \prod_{i=1}^{n} v_i l_i(s)$, same with $R(s), O(s)$. Therefore, we can update our basic proof system, and it will be suitable to multiple input. The question that how to get input would be talk later in this paper.

Now we should think about how to generate each $l_i(s)$ according to the input variable values (coefficients) $\{v_i\}_{i\in(0,\dots,n)}$. First, let's introduce a new term - *Polynomial interpolation*.

Definition 5 Polynomial interpolation: Given a set of n + 1 discrete data points $(x_k, y_k), k \in (0, ..., n)$, where no two x_k are same. A polynomial $p : R \to R$ is said to interpolate the data if $p(x_k) = y_k$ for each $k \in (0, ..., n)$.
Corollary:If f is a polynomial of degree at most n, then the interpolating polynomial of f at $n + 1$ distinct points is f itself.

Then we could update our proof system as following:

- Setup
  - Construct *variable polynomials* for left operand $\{l_i(x)\}_{i\in(1,...,n)}$ such that for all operations $j \in (1, ..., d)$ they evaluate to corresponding coefficients, and similarly for right operand and output
  - sample random s and α
  - calculate target polynomial $t(x) = (x - t_1)(x - t_2)...(x - t_d), d \in (1, ..., d)$, where $\{t_i\}_{i\in(1,...,d)}$ are roots of the polynomial $l(x) \times r(x) - o(x)$, and its evaluation $g^{t(s)}$
  - Compute proving Key:
    $(\left\{g^{s^k}\right\}_{k\in(1,...,d)}, \{g^{l_i(s)}, g^{r_i(s)}, g^{o_i(s)}, g^{\alpha l_i(s)}, g^{\alpha r_i(s)}, g^{\alpha o_i(s)}\}_{i\in(1,...,n)})$
  - Compute verification Key: $(g^t(s), g^\alpha)$
- Prover
  - calculate $h(x) = \frac{l(x)r(x)-o(x)}{t(x)}$, where $L(x) = \sum_{i=1}^{n} v_i \cdot l_i(x)$, and similarly $R(x), O(x)$
  - assign variable values and sum up to get operand polynomials $g^{L(s)} = \prod_{i=1}^{n} (g^{l_i(s)})^{v_i}$, and similarly $g^{R(s)}, g^{O(s)}$
  - assign variable values to the shifted polynomials $g^{\alpha L(s)} = \prod_{i=1}^{n} (g^{\alpha l_i(s)})^{v_i}$, and similarly $g^{\alpha R(s)}, g^{\alpha O(s)}$
  - calculate encrypted evaluation $g^{h(s)}$ using provided powers of s: $\{g^{s_k}\}_{k\in(1,...,d)}$
  - set proof:
    $\pi = (g^{L(s)}, g^{R(s)}, g^{O(s)}, g^{\alpha L(s)}, g^{\alpha R(s)}, g^{\alpha O(s)}, g^{h(s)})$
- Verifier
  - parse proof as $\pi = (g^L, g^R, g^O, g^{L'}, g^{R'}, g^{O'}, g^h)$
  - variable polynomial restriction check:
    $e(g^L, g^\alpha) = e(g^{L'}, g)$
    $e(g^R, g^\alpha) = e(g^{R'}, g)$
    $e(g^O, g^\alpha) = e(g^{O'}, g)$
  - valid operation check: $e(g^L, g^R) = e(g^{t(s)}, g^h) \cdot e(g^O, g)$

Note: The set of all the variable polynomials $\{l_i(x), r_i(x), o_i(x)\}_{i\in(1,...,n)}$ and the target polynomial $t(x)$ is called a *quadratic arithmetic program* (QAP).

3.4 Optimization Protocol - Non-Interchangeability of Operands and Output

Because we use the same α for all the operands of variable polynomials restriction, there is nothing that prevents prover from

1. using variable polynomials from other operands, e.g., $L'(x) = o_1(x) + r_1(x) + r_5(s) + ...$
2. swapping operand polynomials completely, e.g. $O(s)$ with $L(s)$ will result in operation $O(s) \times R(s) = L(s)$
3. re-using same operand polynomials, e.g., $L(s) \times L(s) = O(s)$

This interchangeability means that the prover can alter the execution and effectively prove some other computation. The obvious way to prevent such behavior is to use different α-shifts for the different operands, concretely we modify our protocol:

- Setup
  - ...
  - sample random $\alpha_l, \alpha_r, \alpha_o$ instead of α
  - calculate corresponding "$\alpha-$shifts" $g^{\alpha_l l_i(s)}, g^{\alpha_r r_i(s)}, g^{\alpha_o o_i(s)}$, $g^t(s)_{i\in(0,...,d)}, d = deg(t(x))$
  - Proving Key:
    $(\{g^{s_k}\}_{k\in(1,...,d)}, g^{l_i(s)}, g^{r_i(s)}, g^{o_i(s)}, g^{\alpha_l l_i(s)}, g^{\alpha_r r_i(s)}, g^{\alpha_o o_i(s)})$
  - Verification Key: $(g^t(s), g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o})$
- Prover
  - ...
  - assign variables to the "shifted polynomials $g^{\alpha_l L(s)} = \prod_{i=1}^{n} (g^{\alpha_l l_i(s)})^{v_i}$, and similarly $g^{\alpha_r R(s)}, g^{\alpha_o O(s)}$
  - set proof:
    $\pi = (g^{L(s)}, g^{R(s)}, g^{O(s)}, g^{\alpha_l L(s)}, g^{\alpha_r R(s)}, g^{\alpha_o O(s)}, g^{h(s)})$
- Verifier
  - ...
  - Variable polynomials restriction check:
    $e(g^L, g^{\alpha_l}) = e(g^{L'}, g)$
    $e(g^R, g^{\alpha_r}) = e(g^{R'}, g)$
    $e(g^O, g^{\alpha_o}) = e(g^{O'}, g)$

It is now not possible to use variable polynomials from other operands since $\alpha_l, \alpha_r, \alpha_o$ are not known to the prover.

3.5 Optimization Protocol - Variable Consistency Across Operands

For any variable v_i we have to assign its value to a variable polynomials for each corresponding operand, i.e., $(g^{l_i(s)})^{v_i}, (g^{r_i(s)})^{v_i}, (g^{o_i(s)})^{v_i}$. Because the validity of each of the operand polynomials is checked separately, no enforcement requires to use same variable values in the corresponding variable polynomials. This means that the value of variable v_1 in left operand can differ from variable v_1 in the right operand or the output.

To avoid different variable values, we can enforce equality of a variable value across operands through the approach of restricting a polynomials (as we did with variable polynomials). If we can create a α-shift across all operands, that would restrain prover such that he can assign only same value. A verifier can check the combination of three polynomials by sample a random value β:

$$e(g^{v_l \cdot l_i(s)} \cdot g^{v_r \cdot r_i(s)} \cdot g^{v_o \cdot o_i(s)}, g^{\beta}) = e(g^{v_{\beta,i} \cdot \beta(l_i(s)+r_i(s)+o_i(s))}, g) \tag{6}$$

Of course, the *beta* is encrypted and added to the verification key g^{β}. Now, if the values of all v_i were the same (i.e. $v_{L,i} = v_{R,i} = v_{O,i} = v_{\beta,i}, i \in (1, ..., n)$), the equation shall hold:

$$(v_{l,i} \cdot l_i(s) + v_{r,i} \cdot r_i(s) + v_{o,i} \cdot o_i(s)) \cdot \beta = v_{\beta,i} \cdot \beta \cdot (l_i(s) + r_i(s) + o_i(s)) \tag{7}$$

However, there is a non-negligible probability that at least two of $l(s), r(s), o(s)$ could either have same evaluation value or one polynomial is divisible by another. This would allow the prover to factor values $v_{L,i}, v_{R,i}, v_{O,i}, v_{\beta,i}$ such that at least two of them are non-equal but the equation holds, rendering the check ineffective. For example, let us consider a single operation, where $l(x) = r(x)$. We will denote evaluation of those two as $w = l(s) = r(s)$ and $y = o(s)$. The equation will look as:

$$\beta(v_l w + v_r w + v_o y) = v_\beta \cdot \beta(w + w + y) \tag{8}$$

Such form allows, for some arbitrary v_R and v_O, to set $v_\beta = v_o, v_l = 2v_o - v_r$, which will change the equation into:

$$\beta(2v_o w - v_r w + v_r w + v_o y) = v_o \cdot \beta(2w + y) \tag{9}$$

Obviously, equation (7) is still holds, since we use the same β to shift three polynomials. Therefore, we can use three different random β to ensure that operand's variable polynomials will have unpredictable values. Following are the protocol modifications:

- Setup
  - ...
  - sample random $\beta_l, \beta_r, \beta_o$

  - calculate, encrypt and add to the proving key the variable consistency polynomials:
    $\left\{ g^{\beta_l l_i(s)+\beta_r r_i(s)+\beta_o o_i(s)} \right\}_{i \in 1,\ldots,n}$
  - encrypt β-s and add to the verification key:$(g^{\beta_l}, g^{\beta_r}, g^{\beta_o})$

- Prover
  - ...
  - assign variable values to the variable consistency polynomials: $g^{z_i(s)} = (g^{\beta_l l_i(s)+\beta_r r_i(s)+\beta_o o_i(s)})^{v_i}$ for $i \in 1, \ldots, n$
  - add assigned polynomials in encrypted space: $g^{Z(s)} = \prod_{i=1}^{n} g^{z_i(s)} = g^{\beta_l L(s)+\beta_r R(s)+\beta_o O(s)}$
  - add to the proof: $g^{Z(s)}$

- Verifier
  - ...
  - check the consistency between provided operand polynomials and the "checksum" polynomial:
    $e(g^L, g^{\beta_l}) \cdot e(g^R, g^{\beta_r}) \cdot e(g^O, g^{\beta_o}) = e(g^Z, g)$, which is equivalent to:
    $e(g,g)^{\beta_l L+\beta_r R+\beta_o O} = e(g,g)^Z$

Same variable values tempering technique will fail in such construction because different β-s makes the same polynomials incompatible for manipulation. However, the terms $g^{\beta_l}, g^{\beta_r}, g^{\beta_o}$ are publicly available an adversary can modify the zero-index coefficient (constant variable) of any of the variable polynomials since it does not rely on s, i.e., $g^{\beta_l s^0} = g^{\beta_l}$

3.6 Optimization Protocol - Non-malleability of Variable and Variable Consistency Polynomials

The previous protocol in 3.5 still has two flaws:

1. Non-Interchangeability of Operands and Output: α-shifts preserve the polynomials' computational structure. However, since the prover knows g^{α_l}, he can transfer $L(x)$ into $L(x) = a \cdot l_a(x) + 1$ while the proof system can still holds.
2. Variable Consistency Across Operand: β-shifts also preserve the consistency between three operand polynomials, but it cannot restrict a dishonest prover to offset the proof data of different operand polynomials at same time.

One way to address malleability is to make $(g^{\beta_l}, g^{\beta_r}, g^{\beta_o})$ from verification key incompatible with $g^{Z(s)}$ by multiplying them in encrypted space by a random secret γ during setup stage: $(g^{\beta_l \gamma}, g^{\beta_r \gamma}, g^{\beta_o \gamma})$. Consecutively such masked encryption does not allow feasibility to modify $g^{Z(s)}$ in a meaningful way since Z(s) is not a multiple of γ, e.g., $g^{Z(s)} \cdot g^{v' \cdot \beta_l \gamma} = g^{\beta_l(L(s)+v'\gamma)+\beta_r R(s)+\beta_o O(s)}$. Because a prover dose not know the γ the alteration will be random. The modification requires us to balance the variable values consistency check equation in the protocol multiplying Z(s) by γ.

Following is update protocol:

- Setup
  - ...
  - sample random $\beta_l, \beta_r, \beta_o, \gamma$
  - add new terms to verification key: $(g^{\beta_l \gamma}, g^{\beta_r \gamma}, g^{\beta_o \gamma}, g^{\gamma})$
- Prover
  - ...
- Verifier
  - ...
  - variable values consistency check should hold:
    $e(g^L, g^{\beta_l \gamma}) \cdot e(g^R, g^{\beta_r \gamma}) \cdot e(g^O, g^{\beta_o \gamma}) = e(g^Z, g^{\gamma})$

3.7 Optimization of Variable Values Consistency Check

The variable values consistency check is effective now, but it adds 4 expensive pairing operations and 4 new terms to the verification key. The Pinocchio protocol uses a clever selection of the generators g for each operand ingraining the "shifts":

- Setup
  - ...
  - sample random $\beta, \gamma, \rho_l, \rho_r$ and set $\rho_o = \rho_l \cdot \rho_r$
  - set generators $g_l = g^{\rho_l}, g_r = g^{\rho_r}, g_o = g^{\rho_o}$
  - set proving key:
    $(\left\{g^{s^k}\right\}_{k\in[d]},$
    $\left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}, g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)}, g_r^{\beta r_i(s)}, g_o^{\beta o_i(s)}\right\})$
  - set verification key:
    $(g_o^{t(s)}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\beta\gamma}, g^{\gamma})$
- Prover
  - ...
  - assign variable values: $g^{Z(s)} = \prod_{i=1}^{n} (g_l^{\beta l_i(s)} \cdot g_r^{\beta r_i(s)} \cdot g_o^{\beta o_i(s)})^{v_i}$

- Verifier
  - ...
  - variable polynomials restriction check:
    $e(g_l^{L'}, g) = e(g_l^L, g^{\alpha_l})$, and similarly for g_r^R, g_o^O
  - variable values consistency check:
    $e(g_l^L \cdot g_r^R \cdot g_o^O, g^{\beta\gamma}) = e(g^Z, g^\gamma)$
  - valid operations check:
    $e(g_l^L, g_r^R) = e(g_o^t, g^h) \cdot e(g_o^O, g) \Rightarrow$
    $e(g,g)^{\rho_l \rho_r LR} = e(g,g)^{\rho_l \rho_r (th+O)}$

Such randomization of the generators further adds to the security making variable polynomials malleability, ineffective because for intended change it must be a multiple of either $\rho_l, \rho_r, or \rho_o$, raw or encrypted versions of which are not available.

3.8 Constraints - R1CS(Rank 1 Constraint System)

Our analysis has been primarily focusing on the notion of operation. However, the protocol is not actually "computing" but rather is checking that the output value is the correct result of an operation for the operand' s values. That is why it is called a constraint, i.e., a verifier is constraining a prover to provide valid values for the predefined "program" no matter what are they. A multitude of constraints is called a constraint system (in our case it is a rank 1 constraint system or *R1CS*).

Therefore we can also use constraints to ensure other relationships. For example, if we want to make sure that the value of the variable α can only be 0 or 1 (i.e., binary), we can do it with the simple constraint:$\alpha_l \times \alpha_r = \alpha_o$.

We can also constrain α to only be 2: $(\alpha - 2) \times 1 = 0$

A more complex example is ensuring that number α is a 4-bit number. Therefore, there would not only have one constraint.

1. $\alpha \times 1 = 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + 1 \cdot b_0 \quad (*)$
2. $b_0 \times b_0 = b_0$
3. $b_1 \times b_0 = b_1$
4. $b_2 \times b_0 = b_2$
5. $b_3 \times b_0 = b_3$

Quite sophisticated constraints can be applied this way, ensuring that the values used are complying with the rules. It is important to note that the above constraint 1 is not possible in the current operation' s construction:

$$\sum_{i=1}^{n} c_{l,i} \cdot v_i \times \sum_{i=1}^{n} c_{r,i} \cdot v_i = \sum_{i=1}^{n} c_{o,i} \cdot v_i \tag{10}$$

Because the value 1 (and 2 from the previous constraint) has to be expressed through $c \cdot v_{one}$, where c can be ingrained into the proving key, but the v_{one} may have any value because the prover supplies it. While we can enforce the $c \cdot v$ to be 0 by setting $c = 0$, it is hard to find a constraint to enforce v_{one} to be 1 in the construction we are limited by. Therefore there should be a way for a verifier to set the value of v_{one}.

We would talk about R1CS later in this paper.

3.9 Public Inputs and One

The proofs would have limited usability if it were not possible to check them against the verifier' s inputs, e.g., knowing that the prover has multiplied two values without knowing what was the result and/or values. While it is possible to "hardwire" the values to check against (e.g., the result of multiplication must always be 12) in the proving key, this would require to generate separate pair of keys for each desired "verifier' s input."

Therefore it would be universal if the verifier could specify some of the values (inputs or/and outputs) for the computation.

Fist, let us consider the proof values $g^{L(s)}, g^{R(s)}, g^{O(s)}$. Because we are using the homomorphic encryption it is possible to augment these values, which means that the verifier could add other variable polynomials to the already provided ones. Therefore if we could exclude necessary variable polynomials from the ones available to the prover, the verifier would be able to set his values on those variables, while the computation check should still match.

The necessary protocol update:

- Setup
  - ...
  - separate all n variable polynomials into two groups:
    - Verifier's m+1 $(0, ..., m)$:
      $L_v(s) = \sum_{i=1}^{m} l_i(s)$, and similarly for $R_v(s), O_v(s)$ where index 0 is reserved for the value $v_one = 1$
    - Prover's n-m $(m+1, ..., n)$:
      $L_p(s) = \sum_{i=m+1}^{n}$, and similarly for $R_p(s), O_p(S)$
  - set proving key:
    $(\left\{g^{s^k}\right\}_{k\in[d]},$
    $\left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}, g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)}, g_r^{\beta r_i(s)}, g_o^{\beta o_i(s)}\right\}_{i\in(m+1,...,n)})$
  - set verification key:
    $(g_o^{t(s)}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\beta\gamma}, g^{\gamma}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\right\}_{i\in(0,...,m)})$
- Prover

  - ...
  - calculate $h(x)$ accounting for the verifier's polynomials: $h(x) = \frac{L(x) \cdot R(x) - O(x)}{t(x)}$, where $L(x) = L_v(x) + L_p(x)$, and similarly for $R(x), O(x)$
  - provide proof:
    $\pi = (g_l^{L_p(s)}, g_r^{R_p(s)}, g_o^{O_p(s)}, g_l^{\alpha_l L_p(s)}, g_r^{\alpha_r R_p(s)}, g_o^{\alpha_o O_p(s)}, g^{Z(s)}, g^{h(s)})$

- Verifier
  - assign verifier's variable polynomial values and add to 1:
    $g_l^{L_v(s)} = g_l^{l_0(s)} \cdot \prod_{i=1}^{m} (g_l^{l_i(s)})^{v_i}$, and similarly for $R_v(s), O_v(s)$
  - variable polynomials restriction check:
    $e(g_l^{L'_p}, g) = e(g_l^{L_p}, g^{\alpha_l})$, and similarly for $g_r^{R_p}, g_o^{O_p}$
  - variable values consistency check:
    $e(g_l^{L_p} \cdot g_r^{R_p} \cdot g_o^{O_p}, g^{\beta\gamma}) = e(g^Z, g^\gamma)$
  - valid operation check:
    $e(g_l^{L_p} \cdot g_l^{L_v(s)}, g_r^{R_p} \cdot g_r^{R_v(s)}) = e(g_o^t, g^h) \cdot e(g_o^{O_p} \cdot g_o^{O_v(s)}, g)$

Effectively this is taking some variables from the prover into the hands of verifier while still preserving the balance of the equation. Therefore the valid operations check should still hold, but only if the prover has used the same values that the verifier used for his input.

3.10 Zero-Knowledge Proof Computation

Now we should consider make a proof of polynomial $p(x) = t(x)h(x)$ zero-knowledge. Then we have to use the random δ-shift, which makes the proof $\delta p(x) = t(x) \cdot \delta h(x)$ indistinguishable from random.

With the computation, we are proving instead that $L(s) \cdot R(s) - O(s) = t(s)h(s)$. While we could just adapt this approach to the multiple polynomials using same δ-shift to generate $\delta L(s), \delta R(s), \delta^2 O(s), \delta^2 h(s)$, which would satisfy the valid operations check through pairings:

$$e(g, g)^{\delta^2 L(s) R(s)} = e(g, g)^{\delta^2 (t(s)h(s) + O(s))} \tag{11}$$

The issue is that having same δ hinders security, because we provide those values separately in the proof:

- if $L(s) = R(s)$, then we could find that $g^{\delta L(s)} = g^{\delta R(s)}$
- if $L(s) = 5R(s)$, then verifier could use brute force to get this relation by doing iteration in 5 steps. It can also apply to the encrypted value of plus operation, e.g., $g^{L(s)} = (g^{R(s)})^i, i \in (1, ..., N)$
- other correlations between elements of the proof may be discovered, e.g., $e(g^{\delta L(s)}, g^{\delta R(s)}) = e(g^{\delta^2 O(s)}, g)$

Consequently, we need to have different randomness ($\delta - s$) for each polynomial evaluation, e.g.,

$$\delta_l L(s) \cdot \delta_r R(s) - \delta_o O(s) = t(s) \cdot (\triangle \ ? \ h(s)) \tag{12}$$

To resolve inequality on the right side of equation (10), we can only modify the proof's value $h(s)$, without alteration of the protocol which would be preferable. Then we need to talk about different situation of different ? operators.

- $? = \times$ we can get that:
  $$\triangle = \frac{\delta_l L(s) \cdot \delta_r R(s) - \delta_o O(s)}{t(s)h(s)}$$
  set $\delta_o = \delta_r \cdot \delta_l$, and then
  $$\triangle = \frac{\delta_l L(s) \cdot \delta_r R(s) - \delta_o O(s)}{t(s)h(s)} = \delta_l \delta_r$$

  However, as noted previously this hinders the zero-knowledge property, and even more importantly such construction will not accommodate the verifier' s input polynomials since they must be multiples of the corresponding $\delta - s$, which would require an interaction. We can try adding randomness to the evaluations:
  $$(L(s) + \delta_l) \cdot (R(s) + \delta_r) - (O(s) + \delta_o) = t(s) \cdot (\triangle \times h(S))$$
  $$\triangle = 1 + \frac{\delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_o}{t(s)h(s)}$$
  However due to randomness it is non-divisible. It is difficult to compute $g^{\triangle h(s)}$. Likewise computation is not possible through encrypted evaluation of $\triangle h(s)$ using encrypted powers $\left\{g^{s^i}\right\}_{i \in [d]}$, because the degree of $h(x)$, $\triangle$ is d, hence the degree of $\triangle h(x)$ is up to $2d$.

- $? = +$ Try applying $\triangle$ through addition:
  $$(L(s) + \delta_l) \cdot (R(s) + \delta_r) - (O(s) + \delta_o) = t(s) \cdot (\triangle + h(S))$$
  $$\triangle = \frac{\delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_o}{t(s)}$$
  Every term in the numerator is a multiple of a δ, therefore we can make it divisible by multiplying each δ with t(s):
  $$(L(s) + \delta_l t(s)) \cdot (R(s) + \delta_r t(s)) - (O(s) + \delta_o t(s)) = t(s) \cdot (\triangle + h(S)) \Rightarrow$$
  $$\triangle = \delta_r L(s) + \delta_l R(s) + \delta_l \delta_r - \delta_o$$
  Which we can efficiently compute in the encrypted space:
  $$g^{L(s) + \delta_l t(s)} = g^{L(s)} \cdot (g^{t(s)})^{\delta_l}, \text{ and similarly } R(s), O(s)$$
  $$g^{\triangle} = (g^{L(s)})^{\delta_r} \cdot (g^{R(s)})^{\delta_l} \cdot (g^{t(s)})^{\delta_l \delta_r} \cdot g^{-\delta_o}$$

Above all, using addition to get $\delta t(s)$-shift can effectively conceal the encrypted data while still pass the valid operations check:

$$L \cdot R - O + t(s)(\delta_r L + \delta_l R + \delta_l \delta_r - \delta_o) = t(s)h(s) + t(s)(\delta_r L + \delta_l R + \delta_l \delta_r - \delta_o) \tag{13}$$

3.11 zk-SNARK Protocol in QSP

Considering all the gradual improvements the final zero-knowledge succinct non-interactive arguments of knowledge protocol is (the zero-knowledge components are optional and highlighted with a different color):

- Setup
  - select a generator g and a cryptographic pairing e
  - for a function $f(u) = y$ with n total variables of which m are input/output variables, convert into the polynomial form $(\{l_i(x), r_i(x), o_i(x)\}_{i \in (0,\ldots,n)}, t(x))$ of degree d (equal to the number of operations) and size $n+1$
  - sample random $s, \rho_l, \rho_r, \alpha_l, \alpha_r, \alpha_o, \beta, \gamma$
  - set $\rho_o = \rho_l \cdot \rho_r$ and the operand generators $g_l = g^{\rho_l}, g_r = g^{\rho_r}, g_o = g^{\rho_o}$
  - set the proving key: $\left(\left\{g^{s^k}\right\}_{k \in [d]}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\right\}_{i \in (0,\ldots,n)},\right.$ $\left\{g_l^{\alpha_l l_i(s)}, g_r^{\alpha_r r_i(s)}, g_o^{\alpha_o o_i(s)}, g_l^{\beta l_i(s)}, g_r^{\beta r_i(s)}, g_o^{\beta o_i(s)}\right\}_{i \in (m+1,\ldots,n)},$ $\left. g_l^{t(s)}, g_r^{t(s)}, g_o^{t(s)}, g_l^{\alpha_l t(s)}, g_r^{\alpha_r t(s)}, g_o^{\alpha_o t(s)}, g_l^{\beta t(s)}, g_r^{\beta t(s)}, g_o^{\beta t(s)}\right)$
  - set the verification key: $\left(g, g_o^{t(s)}, g^{\alpha_l}, g^{\alpha_r}, g^{\alpha_o}, g^{\beta\gamma}, g^{\gamma}, \left\{g_l^{l_i(s)}, g_r^{r_i(s)}, g_o^{o_i(s)}\right\}_{i \in (0,\ldots,m)}\right)$

- Prover
  - for the input u, execute the computation of $f(u)$ obtaining values $(v_i)_{i \in (m+1,\ldots,n)}$ for all the intermediary variables
  - assign all values to the unencrypted variable polynomials $L(x) = l_0(x) + \sum_{i=1}^{n} v_i \cdot l_i(x)$, and similarly $R(x), O(x)$
  - sample random $\delta_l, \delta_r, \delta_o$
  - find $h(x) = \frac{L(x) \cdot R(x) - O(x)}{t(x)} + \delta_r L(x) + \delta_l R(x) + \delta_l \delta_r t(x) - \delta_o$
  - assign the prover's variable values to the encrypted variable polynomials and apply zero-knowledge δ-shift: $g_l^{L_p(s)} = (g_l^{t(s)})^{\delta_l} \cdot \prod_{i=m+1}^{n} (g_l^{l_i(s)})^{v_i}$, and similarly $g_r^{R_p(s)}, g_o^{O_p(s)}$
  - assign its α-shifted pairs: $g_l^{L'_p(s)} = (g_l^{\alpha_l t(s)})^{\delta_l} \cdot \prod_{i=m+1}^{n} (g_l^{\alpha_l l_i(s)})^{v_i}$, and similarly $g_r^{R'_p(s)}, g_o^{O'_p(s)}$
  - assign the variable values consistency polynomials: $g^{Z(s)} = (g_l^{\beta t(s)})^{\delta_l} \cdot (g_r^{\beta t(s)})^{\delta_r} \cdot (g_o^{\beta t(s)})^{\delta_o} \cdot \prod_{i=m+1}^{n} (g_l^{\beta l_i(s)} g_r^{\beta r_i(s)} g_o^{\beta o_i(s)})^{v_i}$

  - provide proof:$\pi = (g_l^{L_p(s)}, g_r^{R_p(s)}, g_o^{O_p(s)}, g_l^{L'_p(s)}, g_r^{R'_p(s)}, g_o^{O'_p(s)}, g^{Z(s)}, g^{h(s)})$

- Verifier
  - parse a provided proof as $(g_l^{L_p}, g_r^{R_p}, g_o^{O_p}, g_l^{L'_p}, g_r^{R'_p}, g_o^{O'_p}, g^Z, g^h)$
  - assign input/output values to verifier's encrypted polynomials and add to 1: $g_l^{L_v(s)} = g_l^{l_0(s)} \cdot \prod_{i=1}^{m} (g_l^{l_i(s)})^{v_i}$, and similarly$R_v(s), O_v(s)$
  - variable polynomials restriction check: $e(g_l^{L'_p}, g) = e(g_l^{L_p}, g^{\alpha_l})$, and similarly $g_r^{R_p}, g_o^{O_p}$
  - variable values consistency check: $e(g_l^{L_p} \cdot g_r^{R_p} \cdot g_o^{O_p}, g^{\beta\gamma}) = e(g^Z, g^\gamma)$
  - valid operations check: $e(g_l^{L_p} \cdot g_l^{L_v(s)}, g_r^{R_p} \cdot g_r^{R_v(s)}) = e(g_o^t, g^h) \cdot e(g_o^{O_p} \cdot g_o^{O_v(s)}, g)$

4 QAP 与 zkSnark

In real life, most problem can be flatten to a circuit, while the circuit could be further transferred into an R1CS. Then interpolation could change R1CS into a quadratic arithmetic problem (QAP). Therefore, in the implementation of zkSNARK is mainly based on QAP.

However, a zkSNARK based on a QAP is similar to a not optimized zk-SNARK based on a QSP, so the main focus is on how to convert the arithmetic circuit into a QAP expression. Finally, the optimized version of the zkSNARK protocol (Groth16) based on the QAP will be introduced.

4.1 QAP Definition

Definition 3.1 QAP (Quadratic Arithmetic Program) A quadratic arithmetic program Q over field F contains three sets of polynomials: $V = \{v_k(x) : k \in (0, ..., m)\}, W = \{w_k(x) : k \in (0, ..., m)\}, Y = \{y_k(x) : k \in (0, ..., m)\}$, and a target polynomial t(x), all from $F[x]$.

Let f be a function having input variables with labels 1,...,n and output variables with labels $m - n' + 1, ..., m$. We say that Q is a QAP that computes f if the following is true: $\alpha_1, ..., \alpha_n, \alpha_{m-n'+1}, ..., \alpha_m \in F^{n+n'}$ is a valid assignment to the input/output variables of f iff there exist $\alpha_{n+1}, .., \alpha_{m-n'} \in F^{m-n-n'}$ such that $t(x)$ could divides the following polynomial:um

$$(v_0(x) + \sum_{k=1}^{m} \alpha_k \cdot v_k(x)) \cdot (w_0(x) + \sum_{k=1}^{m} \alpha_k \cdot w_k(x)) - (y_0(x) + \sum_{k=1}^{m} \alpha_k \cdot y_k(x)) \quad (14)$$

4.2 R1CS 到 QAP

To describe an arithmetic circuit as a QAP problem is actually to describe each gate in the circuit, which is the so-called R1CS.

4.2.1 Description of R1CS

The description of R1CS is actually very simple. Its essence is a language that describes the relationship between variables, namely $A * B = C$ (A/B/C are linear combinations of input variables). The R1CS of an arithmetic circuit is essentially the description process of the multiplication gates in the circuit, and each multiplication gate corresponds to the constraint in R1CS.

Given m variables and 1 constant value with n constraints, all R1CS can be described as following: $V \times W = Y$

$$\begin{bmatrix} V_{00} & \cdots & V_{0M} \\ V_{10} & \cdots & V_{1M} \\ \cdots & & \\ V_{N-1,0} & \cdots & V_{N-1,M} \end{bmatrix} \cdot \begin{bmatrix} W_{00} & \cdots & W_{0M} \\ W_{10} & \cdots & W_{1M} \\ \cdots & & \\ W_{N-1,0} & \cdots & W_{N-1,M} \end{bmatrix} = \begin{bmatrix} Y_{00} & \cdots & Y_{0M} \\ Y_{10} & \cdots & Y_{1M} \\ \cdots & & \\ Y_{N-1,0} & \cdots & Y_{N-1,M} \end{bmatrix} \tag{15}$$

Each row is a constraint, e.g., the constraint in first row shows that:

$$(\sum_{i=0}^{M} \alpha_i V_{0i}) \cdot (\sum_{i=0}^{M} \alpha_i W_{0i}) = (\sum_{i=0}^{M} \alpha_i Y_{0i}) \tag{16}$$

For example, an arithmetic circuit as following, and then we can get a QAP of this circuit. $f : F_{11} \times F_{11} \times F_{11} \to F_{11} : (x_1, x_2, x_3) \to (x_1 \cdot x_2)$:

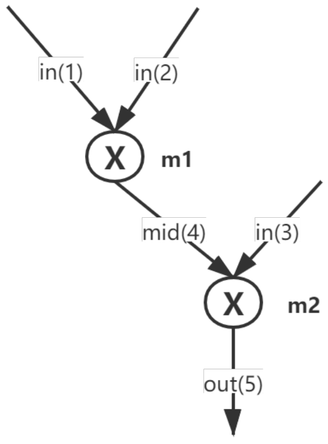

Figure 1: Example

We can find that m_1, m_2 are multiplication gates, and the coefficient set is $I = (in_1, in_2, in_3, mid_4, out_5)$. In the following paper, the coefficient set will be simplified to $I = (1, 2, 3, 4, 5)$. According to the R1CS description defined above, each multiplication gate (constraint) has 2 inputs and one output, corresponding to the form of $V \times W = Y$, and the entire circuit can also be regarded as a multiplication gate (or as A black box is easy to understand), the left input is multiplied by the input to get the output.

Because R1CS is a description of the multiplication gate, first define multiplication gate as: $\{v_k(x)\}_{k \in I}$ is the left input of multiplication gate x and v is the left input matrix of the entire circuit, and similarly to get W, Y. Then we have the value of $\{v_k(x)\}_{k \in I}$. If $v_k(x)$ is the left input of the multiplication gate x, then $v_k(x) = 1$, otherwise $v_k(x) = 0$. Similarly, W is related to the right input of a multiplication gate and Y is related to the output of a multiplication gate.

Therefore we can give a R1CS description of multiplication gates $m1, m2$ in Fig.1:

$$m1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}, m2 = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{17}$$

Then make reasonable assignments $\{\alpha_k\}_{k \in I}$ to the coefficients on the circuit, the R1CS of each multiplication gate (constraint) can be expressed in the form of equation (16).

4.2.2 From R1CS to QAP

Combine all left input to a left input matrix V, e.g., the left input matrix of example 1:

$$V = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{18}$$

Also, we can get the right input matrix W and output matrix Y of all multiplication gates in an entire circuit. Then we treat each column as a point-value pair evaluated by a polynomial, e.g., for left input, we have (assume we set $m_1 = 5, m_2 = 7$):

$$\begin{cases} v_1(m1) = v_1(5) & = 1 \\ v_1(m2) = v_1(7) & = 0 \end{cases} \Rightarrow v_1(x) = 5x + 9 \tag{19}$$

The expression of polynomials $\{v_k(x)\}_{k \in I}, \{w_k(x)\}_{k \in I}, \{y_k(x)\}_{k \in I}$ can be get by same ways. When the number of multiplication gates is large, Lagrangian interpolation and fast discrete Fourier transform (FFT) can be used to speed up the process of calculating coefficients.

After the circuit is converted to QAP, the corresponding target polynomial $t(x)$ needs to be generated. The target polynomial is defined as $t(x) = (x - m_1) \cdot ... \cdot (x - m_{k'}), k' < deg(t(x))$, while $deg(t(x))$ means the number of multiplication gates in the circuit, e.g., the target polynomial in the example 1 is:

$$t(x) = (x - m_1)(x - m_2) = (x - 5)(x - 7) \tag{20}$$

Above all, we can get the representation of a circuit.

$$QAP(Cir) = \{t(x), (v_k(x), w_k(x), y_k(x))_{k \in I}\} \tag{21}$$

In the example 1, the QAP representation is (over field F_{11}):

$$QAP(Cir) = \left\{ x^2 + 10x + 2, \left\{ \begin{array}{c} v = \{5x + 9, 0, 0, 6x + 3, 0\} \\ w = \{0, 5x + 9, 6x + 3, 0, 0\} \\ y = \{0, 0, 0, 5x + 9, 6x + 3\} \end{array} \right\} \right\} \tag{22}$$

Only if the coefficients assignment is valid then the QAP could be valid, which means equation (12) holds. Assume we assign $I = (2, 3, 4, 6, 2)$ to the coefficients in the example 1, then:

$$\begin{aligned}
&(\sum_{i=0}^{M} \alpha_i v_i(x)) \cdot (\sum_{i=0}^{M} \alpha_i w_i(x)) - (\sum_{i=0}^{M} \alpha_i y_i(x)) \\
&= (2(5x + 9) + 6(6x + 3)) \cdot (3(5x + 9) + 4(6x + 3)) - (6(5x + 9) + 2(6x + 3)) \\
&= x^2 + 10x + 2 \\
&= t(x) \cdot h(x), h(x) = 1
\end{aligned} \tag{23}$$

If the assignment is invalid, e.g., $I = (2, 3, 4, 6, 2)$, the left part would be $9x^2 + 7x + 5$. Then $h(x) = 9 + \frac{5x+4}{x^2+10x+2}$ cannot be divided by $t(x)$.

4.3 Groth16 Algorithm

In order to explain the Groth16 algorithm more conveniently, first give the definition of QAP in the Groth16 algorithm: Over a finite field F, we have statements $(\alpha_1, ..., \alpha_l) \in Z_p^l$, and witness $(\alpha_{l+1}, ..., \alpha_m) \in Z_p^{m-l}$. The following equation would holds if $\alpha_0 = 1$ $(deg(t(x)) = n)$:

$$(\sum_{i=0}^{M} \alpha_i v_i(x)) \cdot (\sum_{i=0}^{M} \alpha_i w_i(x)) = (\sum_{i=0}^{M} \alpha_i y_i(x)) + t(x)h(x) \tag{24}$$

4.3.1 zkSNARK in Groth16

Given three finite group G_1, G_2, G_T, and the generator $g, h, e(g, h)$ of each group. We have a bilinear map e such that $e : G_1 \times G_2 \to G_T$, for convenient, set $[y]_1 = g^y, [y]_2 = h^y$.

- Setup
  - pick $\alpha, \beta, \gamma, \delta, x \leftarrow Z_p^*$ randomly
  - Define $\tau = (\alpha, \beta, \gamma, \delta, x)$,
    and compute $\sigma = ([\sigma_1]_1, [\sigma_2]_2)$,
    $\sigma_1 =$

$(\alpha, \beta, (x^i)_{i=0}^{n-1}, \left\{\frac{\beta v_i(x)+\alpha w_i(x)+y_i(x)}{\gamma}\right\}_{i=0}^{l},$
$\left\{\frac{\beta v_i(x)+\alpha w_i(x)+y_i(x)}{\delta}\right\}_{i=l+1}^{m}, \left\{\frac{x^i t(x)}{\delta}\right\}_{i=0}^{n-2})$
$\sigma_2 = (\beta, \gamma, \delta, \{x^i\}_{i=0}^{n-1})$

  - set proving key: $(\alpha, \left\{\frac{\beta v_i(x)+\alpha w_i(x)+y_i(x)}{\gamma}\right\}_{i=0}^{l})_1$
    和 $(\beta, \gamma, \delta)_2$

- Prover
  - sample random r, s and computes $\pi = \Pi\sigma = ([A]_1, [B]_2, [C]_1)$, where
    $A = \alpha + \sum_{i=0}^{m} a_i v_i(x) + r\delta,\ B = \beta + \sum_{i=0}^{m} a_i w_i(x) + s\delta$
    $C = \frac{\sum_{i=l+1}^{m} a_i(\beta v_i(x)+\alpha w_i(x)+y_i(x))+h(x)t(x)}{\delta} + As + rB - rs\delta$

- Verifier
  - Accept the proof if and only if
    $[A]_1 \cdot [B]_2 = [\alpha]_1 \cdot [\beta]_2 + \sum_{i=0}^{l} a_i [\frac{\beta v_i(x)+\alpha w_i(x)+y_i(x)}{\gamma}]_1 \cdot [\gamma]_2 + [C]_1 \cdot [\delta]_2$
  - Verification only use four pairing:
    $e([A]_1, [B]_2) = e([\alpha]_1, [\beta]_2) \cdot e(\sum_{i=0}^{l} a_i [\frac{\beta v_i(x)+\alpha w_i(x)+y_i(x)}{\gamma}]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2)$

Because the calculation of the bilinear mapping has a high cost in the implementation process, the verification efficiency can be effectively improved by reducing the use of the pairing function.