Lesson 5

More cryptography theory

COMMITMENT SCHEMES

Definition A commitment scheme is defined by algorithms **Commit** and **Open** as follows:

Given a message m and randomness r, compute as output a value c

`c = Commit(m,r).`

that, informally, hides message m and r such that it is hard to invert function Commit to find m or r.

Given a commitment c, a message m and randomness r
`b = Open(c, m, r).`
the algorithm returns true if and only if
c = Commit(m, r).
A commitment scheme has 2 properties:

1. Binding. Given a commitment c, it is hard to compute a different pair of message and randomness whose commitment is c. This property guarantees that there is no ambiguity in the commitment scheme, and thus after c is published it is hard to open it to a different value.
2. Hiding. It is hard to compute any information about m given c.

PEDERSEN COMMITMENTS

- Given group $\mathbb{Z}_p^*$, of prime order p, where the discrete logarithm problem is infeasible, the commitment is computed for message m and randomness r as follows:
  `c = Commit(m,r)`
  In order to open this commitment, given message m and randomness r, we simply recompute it and compare with c.
  An interesting property is that the Pedersen commitment is homomorphic.

```
C(BF1, data1) + C(BF2, data2) == C(BF1 + BF2, data1 + data2)
```

Pedersen commitments are information-theoretically private: for any commitment you see, there exists some blinding factor which would make any amount match the commitment.

Polynomial commitments

From Vitalik's Blog

A polynomial commitment is best viewed as a special way to "hash" a polynomial, where the hash has the additional property that you can check equations between polynomials by checking equations between their hashes. Different polynomial commitment schemes have different properties in terms of exactly what kinds of equations you can check.

Here are some common examples of things you can do with various polynomial commitment schemes (we use com(P) to mean "the commitment to the polynomial P"):

- Add them: given com(P), com(Q) and com(R) check if P+Q=R
- Multiply them: given com(P), com(Q) and com(R) check if P∗Q=R
- Evaluate at a point: given com(P), w, z verify that P(w)=z

Fiat–Shamir heuristic

The Fiat–Shamir heuristic is a technique in cryptography for taking an interactive proof of knowledge and creating a digital signature based on it.

Here is an interactive proof of knowledge of a discrete logarithm.

1. Peggy wants to prove to Victor the verifier that she knows x: the discrete logarithm of $y = g^x$ to the base g
2. She picks a random $v \in \mathbb{Z}_q^*$ computes $t = g^v$ and sends t to Victor.
3. Victor picks a random $c \in \mathbb{Z}_q^*$ and sends it to Peggy.
4. Peggy computes $r = v - cx$ and returns r to Victor.

He checks whether $t \equiv g^r y^c$.
This holds because $g^r y^c = g^{v-cx} g^{xc} = g^v = t$

The Fiat–Shamir heuristic allows us to replace the interactive step 3 with a non-interactive random oracle access. In practice, we can use a cryptographic hash function instead.

1. Peggy wants to prove to Victor the verifier that she knows x: the discrete logarithm of $y = g^x$ to the base g
2. She picks a random $v \in \mathbb{Z}_q^*$ computes $t = g^v$
3. Peggy computes $c = H(g, y, t)$ where $H()$ is a cryptographic hash function.
4. She computes $r = v - cx$. The resulting proof is the pair (t, r). As r is an exponent of g, it is calculated modulo $q - 1$, not modulo q.
5. Anyone can check whether $t \equiv g^r y^c$.

SIGMA PROTOCOLS

For a cyclic group G_q of order q with generator g.
In order to prove knowledge of $x = \log_g y$, the prover interacts with the verifier as follows:

1. In the first round the prover commits himself to randomness r therefore the first message $t = g^r$ is also called commitment.
2. The verifier replies with a challenge c chosen at random.
3. After receiving c, the prover sends the third and last message (the response) $s = r + cx$

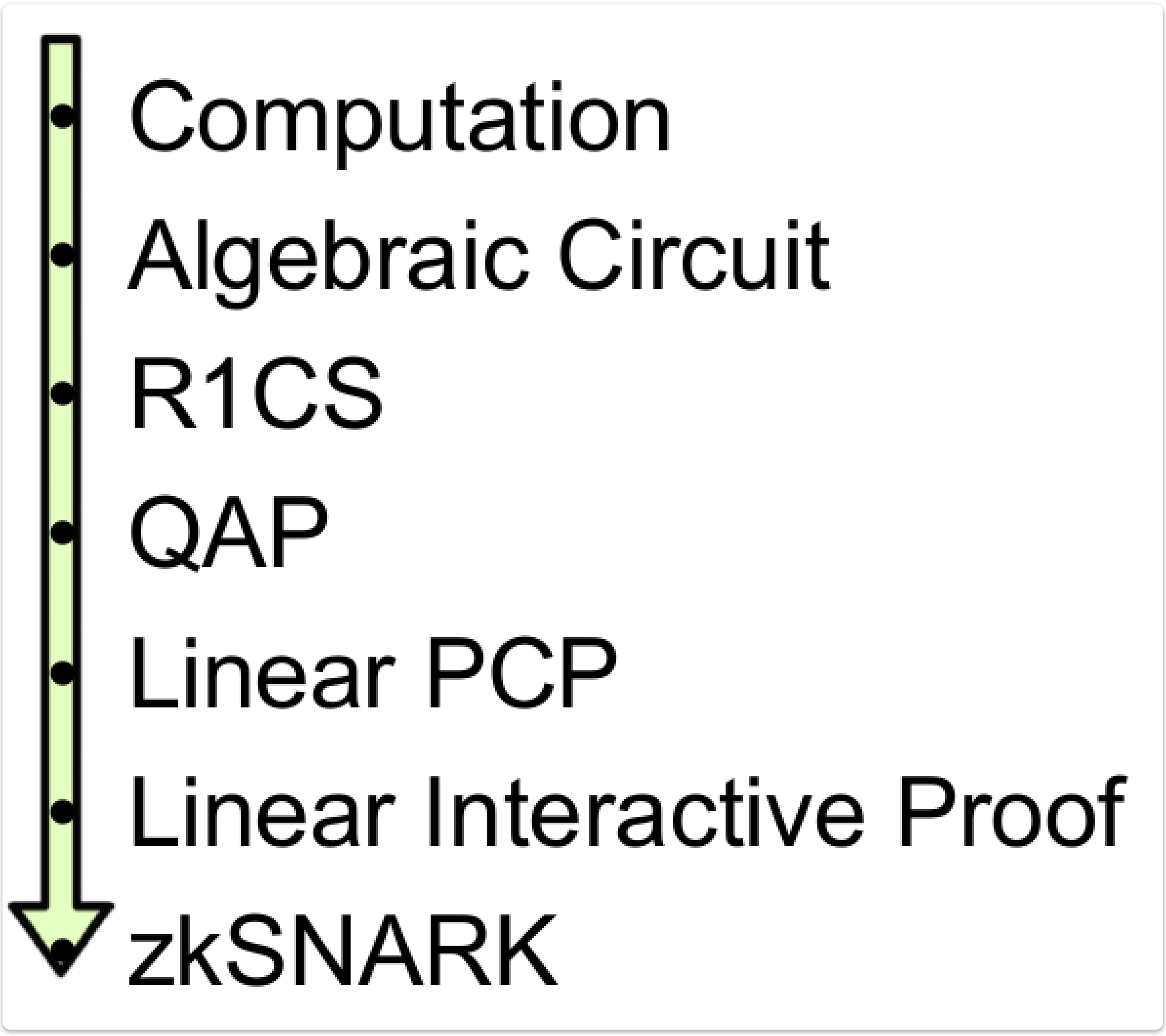
The verifier accepts, if $g^s = ty^c$
Protocols which have the above three-move structure (commitment, challenge and response) are called sigma protocols

zkSNARKS

Currently zkSNARKS are the most common proof system being used, they form the basis for the privacy provided in ZCash.

zkSNARK stands for zero knowledge Succint Non interative Argument of Knowledge. The features of succintness (they are small in size and the amount of computation required) and Non Interaction (a single step is sufficient to complete the proof) has helped their adoption in applications.

Overall Process

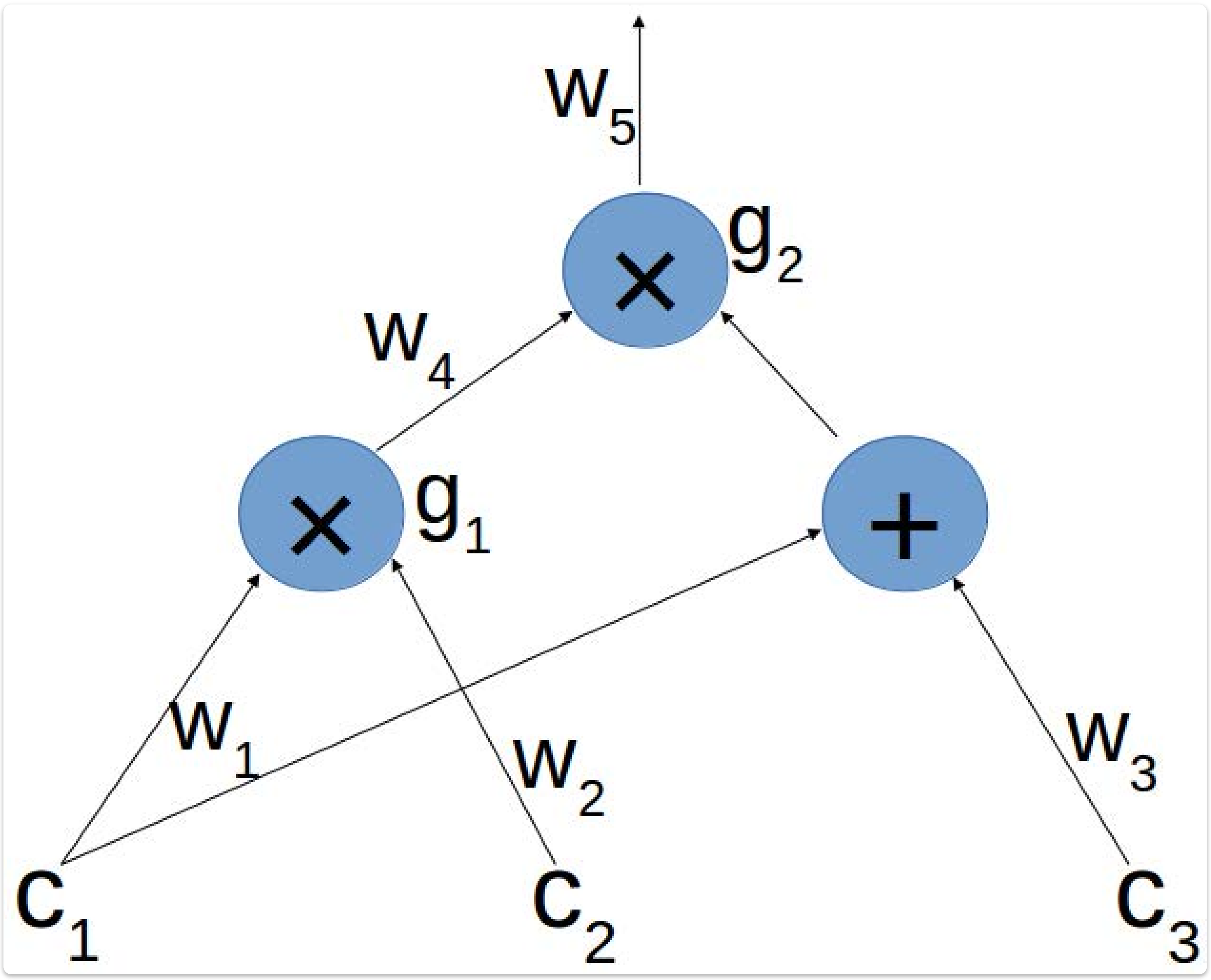


zkSNARKS process part 1 from program to QAP

Process overview

1. Trusted Setup
   ZKSNarks require a one off set up step to produce prover and verifier keys. This step is generally seen as a drawback to zkSNARKS, it requires an amount of trust, if details of the setup are later leaked it would be possible to create false proofs.
2. A High Level description is turned into an arithmetic circuit
   The creator of the zkSNARK uses a high level language to spcify the algorithm that constitutes and tests the proof.

This high level specification is compiled into an arithmetic circuit.
An arithmetic circuit can be thought of as similar to a physical electrical circuit consisting of logical gates and wires. This circuit contrains the allowed inputs that will lead to a correct proof.



3. Further Mathematical refinement
   The circuit is then turned into a an R1CS, and then a series of formulae called a Quadratic Arithmetic Program (QAP).
   The QAP is then further refined to ensure the privacy aspect of the process.
   The end result is a proof in the form of series of bytes that is given to the verifier. The verifier can pass this proof through a verifier function to receive a true or false result.
   There is no information in the proof that the verifier can use to learn any further information about the prover or their witness.

Trusted Setups

From ZCash explanation :

"SNARKs require something called “the public parameters”. The SNARK public parameters are numbers with a specific cryptographic structure that are known to all of the participants in the system. They are baked into the protocol and the software from the beginning.

The obvious way to construct SNARK public parameters is just to have someone generate a public/private keypair, similar to an ECDSA keypair *(https://electriccoin.co/blog/snark-parameters/#id2), and then destroy the private key.

The problem is that private key. Anybody who gets a copy of it can use it to counterfeit money. (However, it cannot violate any user's privacy — the privacy of transactions is not at risk from this.)"

We've devised a *secure multiparty computation* in which multiple people each generate a "shard" of the public/private keypair, then they each destroy their shard of the toxic waste private key, and then they all bring together their shards of the public key to to form the SNARK public parameters. If that process works — i.e. if *at least one of the participants* successfully destroys their private key shard — then the toxic waste byproduct never comes into existence at all.

They have recently introduced Halo2 which eliminates the need for a trusted setup

They use the curves Pallas and Vesta (collectively Pasta) which are also used by Mina

Pallas: y^2 = x^3 + 5y2=x3+5 over GF(0x40000000000000000000000000000000224698fc094cf91b992d30ed00000001)

Vesta: y^2 = x^3 + 5y2=x3+5 over GF(0x40000000000000000000000000000000224698fc0994a8dd8c46eb2100000001)

The use "nested amortization"— repeatedly collapsing multiple instances of hard problems together over cycles of elliptic curves so that computational proofs can be used to reason about themselves efficiently, which eliminates the need for a trusted setup.

Transforming our problem into a QAP

Lets look first at transforming the problem into a QAP, there are
3 steps :

- code flattening,
- conversion to a rank-1 constraint system (R1CS)
- formulation of the QAP.

Code Flattening

We are aiming to create arithmetic and / or boolean circuits from our code, so we change the high level language into a sequence of statements that are of two forms

x = y (where y can be a variable or a number)
and
x = y (op) z
(where op can be +, -, $*$, / and y and z can be variables, numbers or themselves sub-expressions).

For example we go from

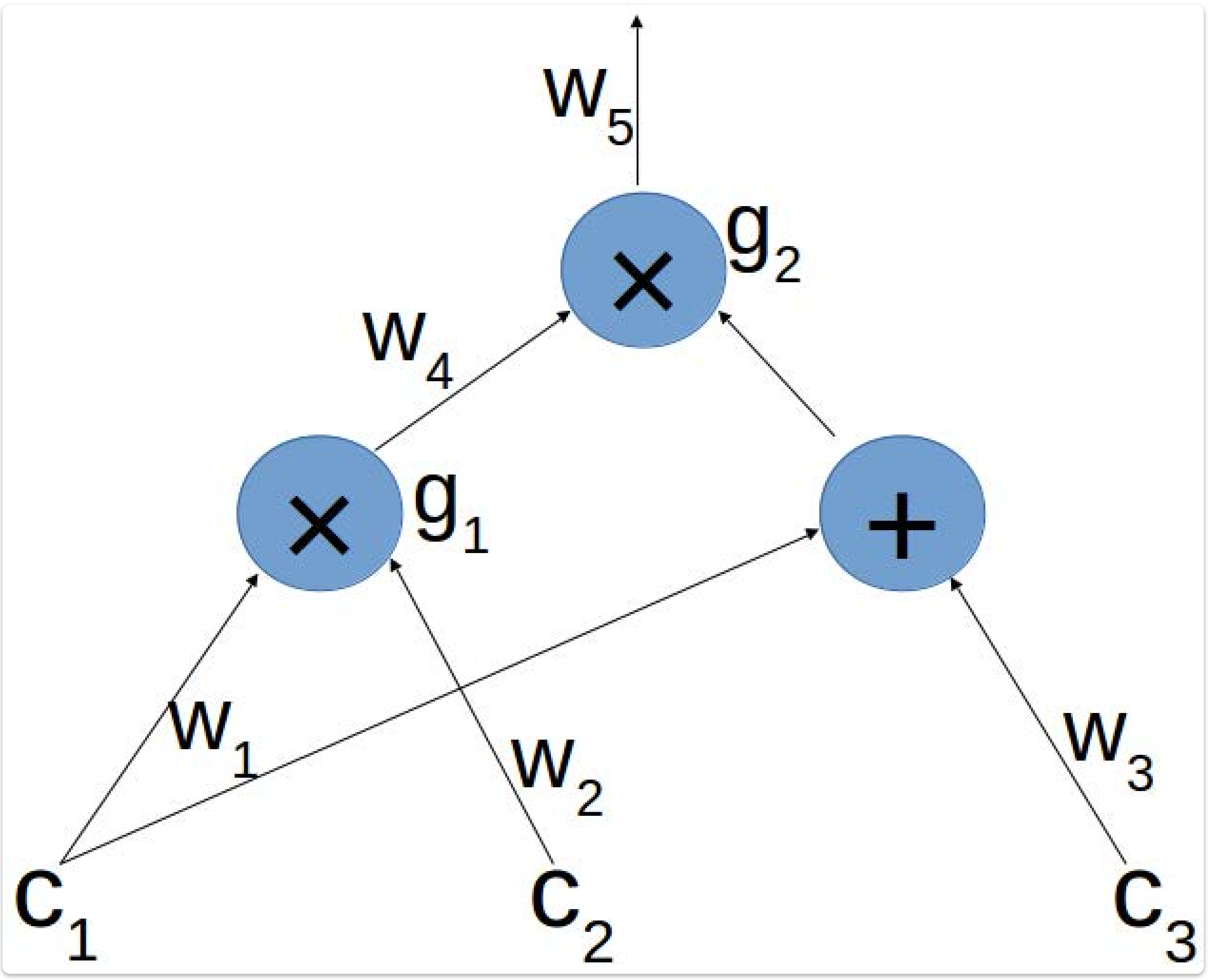
```
def qeval(x):
    y = x**3
    return x + y + 5
```

to

```
sym_1 = x * x
y = sym_1 * x
sym_2 = y + x
~out = sym_2 + 5
```

ARITHMETIC CIRCUIT

This is a collection of multiplication and addition gates



Rank 1 Constraint Systems

Constraint languages can be viewed as a generalization of functional languages:

- everything is referentially transparent and side-effect free
- there is no ordering of constraints
- composing two R1CS programs just means that their constraints are simultaneously satisfied.

(From http://coders-errand.com/constraint-systems-for-zk-snarks/)

The important thing to understand is that a R1CS is not a computer program, you are not asking it to produce a value from certain inputs. Instead, a R1CS is a verifier, it shows that an already complete computation is correct .

The arithmetc circuit is a compositon of multplicatve sub-circuits (a single multiplcation gate and mutiple addition gates)

A rank 1 constraint system is a set of these sub-circuits expressed as constraints, each of the form:
$AXB = C$
where A, B, C are each linear combinations c1· v1+ c2· v2+ ...
The c_i are constant field elements, and the v_i are instance or witness variables (or 1).

- $AXB = C$ doesn't mean C is computed from A and B just that A, B, C are consistent.

More generally, an implementation of $x = f(a, b)$ doesn’t mean that x is computed from a and b, just that x, a, and b are consistent.

Thus our R1CS contains :

- the constant 1
- all public inputs
- outputs of the function
- private inputs
- auxilliary variables

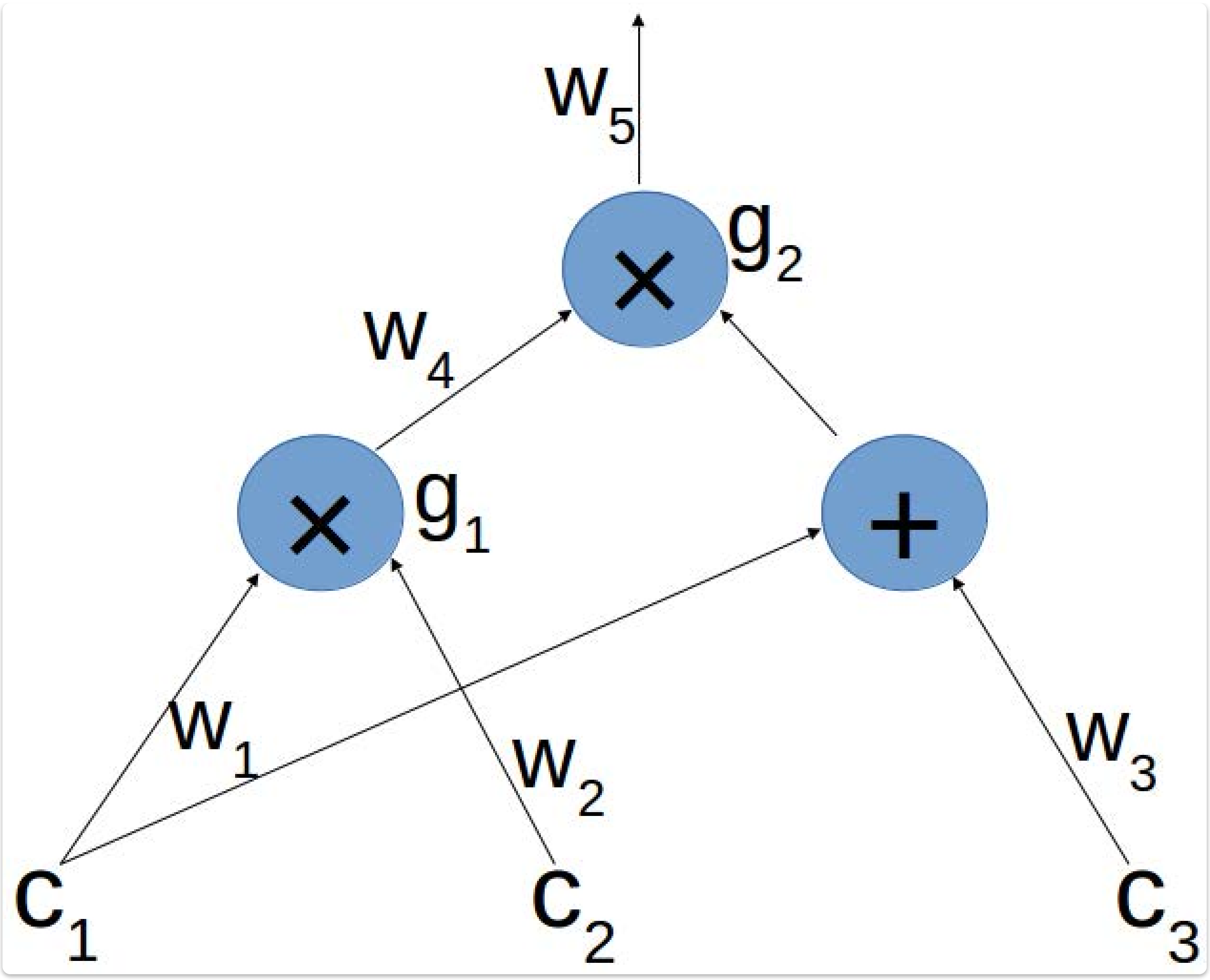
The R1CS has

- one constraint per gate;
- one constraint per circuit output.

EXAMPLE

Assume Peggy wants to prove to Victor that she knows
$c1, c2, c3$ such that
$(c1 \cdot c2) \cdot (c1 + c3) = 7$

We transform the expression above into an arithmetic circuit as depicted below



A legal assignment for the circuit is of the form:

(c1, . . . , c5), where c4 = c1 · c2 and c5 = c4 · (c1 + c3).

From R1CS to QAP

QUADRATIC ARITHMETIC PROGRAM

A Quadratic Arithmetic Program (QAP) is derived from an R1CS.
From [Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture] (https://eprint.iacr.org/2013/879.pdf)

Definition 2.2.

A quadratic arithmetic program of size m and degree d over $\mathbb{F}$
is a tuple (A B, C, Z), where A, ~ B, ~ C~ are three vectors, each of m + 1 polynomials in $\mathbb{F}\leq d-1[z]$, and $Z \in F[z]$ has degree exactly d.
Like a circuit, a QAP induces a satisfaction problem:

Our QAP itself is made up of 3 polynomial vectors, A B, C, corresponding to the linear combinations of the same name in the R1CS; plus one polynomial Z.

See (http://coders-errand.com/how-to-build-a-quadratic-arithmetic-program/)

Gennaro et al.showed that circuit satisfiability can be efficiently reduced to QAP satisfiability so we will use a Quadratic Arithmetic Program to verify circuit computation

Our goal is to devise a set of polynomials that simultaneously encode all of the constraints, so that we can verify the satisfiability thereof with a single check on the polynomials instead of a check over each constraint. The clever trick is to build the polynomials in a way that they can generate all of the constraints.

The R1CS is initially encoded as vectors and from these we define poynomials

We are aiming to create a left, right, and output polynomial of degree m for m constraints and a target polynomial of degree d

Given fixed values
$(c_1, \ldots, c_5)$
we use them as coefficients to define a left, right, and output polynomials.
We want to show that our created QAP has a legal assignment,

See ZCash explanation
From our example above each multiplication gate X has an associated target point in the field $\mathbb{F}_p$
.
In our example, they will be 1 and 2.

- Each wire (in our example, there are 5) has an associated left, right and output polynomials: .
- Each polynomial evaluates to either 0 or 1 on the target points
  If the polynomial is "related" to the multiplication gate associated with a target point, then it evaluates to 1. Otherwise to 0.

Our QAP Q of degree d and size m consists of polynomials

$L_1, \ldots, L_m, R_1, \ldots, R_m, O_1, \ldots, O_m$
and a target polynomial T of degree d.

An assignment (c1,...,cm) satisfies Q if,

defining

$$L := \sum_{i=1}^{m} ci \cdot Li, R := \sum_{i=1}^{m} ci \cdot Ri, O := \sum_{i=1}^{m} ci \cdot Oi$$

and we define the polynomial P
$P := L \cdot R - O$
Defining the target polynomial $T(X) := (X - 1) \cdot (X - 2) T(X) := (X - 1) \cdot (X - 2)$, we thus have that T divides P if and only if (c1,...,c5)(c1,...,c5) is a legal assignment.

We moved from a situation where we had d groups of 3 * d vectors of m + 1 coefficients to one where we have 3 vectors each with m+1 polynomials of d-1 degree.

We converted a set of vectors into polynomials that generate them when evaluated at certain fixed points. We used these fixed points to generate a vanishing polynomial that divides any polynomial that evaluates to 0 at least on all those points. We created a new polynomial that summarizes all constraints and a particular assignment, and the consequence is that we can verify all constraints at once if we can divide that polynomial by the vanishing one without remainder. This division is complicated, but there are methods (the Fast Fourier Transform) that can perform if efficiently.