A semi-technical overview of zk-Snarks

Paul de Lange*

February 7, 2021

Abstract

draft

* *E-mail:* p.delange@uky.edu

Contents

1 Introduction

This document aims at explaining how Sonic works, what is does and how it can be implemented. We intend to be pragmatic over academic and will try to only introduce formal concepts when they are needed.
Some common concepts and buzzwords are briefly explained in info boxes.

2 zk-SNARKS

zk-SNARKS offer a way (a "proof") for a prover **P** to convince a verifier **V** that it knows a set of constants $\{c_i\}_{i=1}^{M-1}$ that are subject to some relation $f(c_1, \ldots, c_{N-1}) = c_N$, where the relation f is a combination of the elementary arithmetic operations $\times$ and $+$ only. The constants c_i are elements of a field $c_i \in \mathbb{F}_q$, as is c_N, $c_N \in \mathbb{F}_q$, and the goal for **P** is to convince **V** that it indeed knows these constants subject to the constraint, without actually revealing what the constants are. A couple of remarks are in order.

First, all that we say here has a straightforward generalization to a system of multiple constraints and we will give an example of this later because this will be the typical scenario in the context of blockchain applications. The generalization simply means allowing for multiple outcoming wires at the end of the circuit.

Secondly note that although this set-up may look specific, it is actually rather general and many problems can be recast in this form; note that – having in mind the generalization to multiple constraints – this phrasing does include generic linear algebraic problems, as well as problems in optimization (e.g. TPS) and graph theory (e.g. coloring).

Finally, note that the problem of solving a constraint equation (or set of equations) like this is in general in NP). This means that, again generally, there will be no known algorithm that solves such an equation in polynomial time; however, given a solution, one *can* verify its validity in polynomial time. Although some redundant or edge cases of the formulation may be in P, this will not be the case generically. What is more, the particular formulation of our problem will be mapped to the so-called Quadratic Span Program, which is NP-complete (CITE Quadratic Span Programs and Succinct NIZKs without PCPs), so we can use this zk-SNARK for any NP problem.

NP

NP stands for *Non-deterministic Polynomial*. A problem sits in NP if there *is* a polynomial-time algorithm to check if a given solution indeed solves that problem. Think of a sudoku. Problems for which there is a polynomial algorithm to solve in its generality are in P (for Polynomial).

The notion NP pops up a lot in crypto and from its definition we can see why: we typically want to hide a message within some problem that is hard to solve for a malevolent adversary, but easy to check a given solution to (easy to "unlock") by the intended receiver. That is to say: we'd like the problem to be in NP, but not in P (sometimes called NP-intermediate). Note that "being in NP" depends on mankind: when someone finds a polynomial time algorithm for an NP-intermediate problem, it moves to P.

For crypto, however, just being in NP-intermediate is not good enough. There are problems that are not in P in its general formulation, but have too many individual instances of the problem that *are* actually easy to solve. In cryptography, we can't risk such a scenario. Rather, we need a problem that is in NP-intermediate, and is not in P for almost all instances (sometimes stated as "hard on average"). The discrete-log problem is a good example of being hard on average. To solve $x = g^y$ for y is not hard only when $y = 1$ or $y = 0$.

You may also have hear of "NP-hard". A problem L is in NP-hard if for every problem x in NP there is a polynomial time algorithm that reduces x to L. Note that from the definition, NP-hard problems need not be in NP. The subset of NP-hard problems that are is called NP-complete.

The phrasing as stated is relevant to the blockchain, because there a block has to carry the transaction information form one sending party to a receiving party, and we want to make it possible for any external verifier **V** to check if indeed the transaction was valid (according to the blockchain's protocol and rules), without disclosing information – sums, number of transactions, id. So for every transaction we want to introduce a prover **P** that any verifier van check the validity of the transaction. For many reasons this verification has to be able to be done quickly, without exchanging a high volume of data and also we want no two-way interaction between **P** and **V** and lastly of course it has to be in "zero knowledge", which means that **V** can not extract any more data from the proof than **P** intends to make public. All this is summed up in

the acronym zk-SNARK:

zero-knowledge Succinct Non-Interactive Argument of Knowledge.

We note here that any zero-knowledge proof π needs to satisfy the following three requirements:

- **Completeness**: **V** will accept an honestly generated proof π
- **Soundness**: a malicious **P** will convince **V** of with a false π with negligible probability only
- **Zero Knowledge**: **V** can not extract information from π other than the public data **P** provides

We will first introduce a generic zk-SNARK protocol (based on REF) and will in the section thereafter introduce the specific protocol Sonic.

The steps of a zk-SNARK protocol – Sonic included – are typically:

1. Map the arithmetic constraint problem to a (variation of an) arithemtic circuit
2. From this arithemtic circuit, construct a specific polynomial $p(x, y, \ldots)$
3. Rephrase the satisfaction of the constraints in terms of the fact the p is divisable by some known and public polynomial, called the *target polynomial* t: $t \,|\, p$.
4. Now this is the statement **P** will prove to **V** : I know p such that $t \,|\, p$ (which then directly implies / translates to: I know c_i that satisfy $f(c_i)$)
5. Introduce some non-interactive scheme that allows for that proof to be checked fast etc.

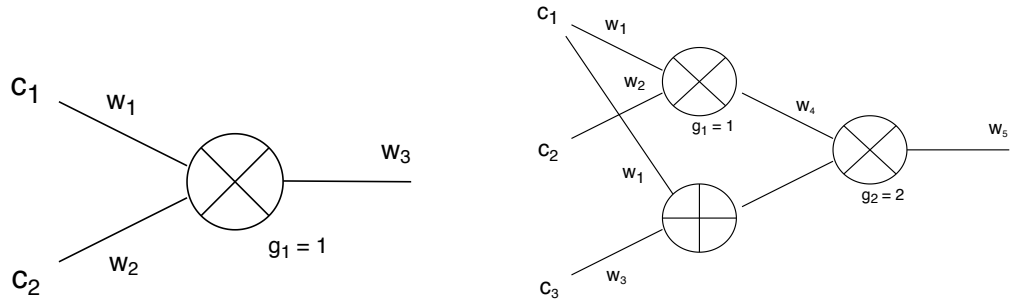
The subsections to come will follow these steps, beginning with the introduction of arithmetic circuits

2.1 Arithmetic Circuits

The first step in the snark will be to deconstruct our set of relations and draw them up into a diagram called an *arithmetic circuit*. An arithmetic circuit is, loosely speaking – an image of how we would let a computer execute such a relation in its most elementary form, identifying every single addition and multiplication. Every multiplication and addition is drawn as a *node* in a directed graph, with two wires – one *left* and one *right* – coming in to the node (representing the incoming free variables of the operation), and one *out coming* wire (representing the resulting free of the operation). Multiplication with a fixed scalar is done at the wire itself.
Consider the most basic example of the most simple multiplicative relation:

$$c_1 \cdot c_2 = c_3.$$

Its arithmetic circuit is drawn in figure 1a In this circuit, we identify one multiplication node



(a) Basic multiplication of two scalars (b) Combined addition and multiplication

Figure 1: Arithmetic circuits

(or gate) g_1, that has one left wire, w_1, one right wire w_2 and one output wire w_3.
The w_is are constants that we will be trying to solve with the circuit. The first rule is that the wires after incoming constants are assigned those constants, so in this case:

$$c_1 \to w_1,\ c_2 \to w_2,$$

Now we say that the circuit is *satisfied* of *valid* or *legal* if w_3 is assigned the constant c_3, and we have $c_3 = c_1 \cdot c_2$. In general, for a circuit with W wires, an array of constants

$$(c_1, \ldots, c_W)$$

is a valid assignment for that circuit if every c_i is assigned to w_i in a way that for every output wire, the assigned constant equals the result of the arithmetic operation on the assigned constants of the incoming wires.
This may sound extremely trivial, to the point that it feels like nothing is going on, which admittedly is almost the case. But we will go on and make this a little bit more complicated.

For later purposes, we are going to label the wires of a general circuit in a specific way, in a way that may sound random, but will make sense later when we get closer to the polynomial business.
In our arithmetic circuit, multiplicative gates are special: we only label multiplicative gates with g_i, and only wires coming out of a multiplicative gate will be labeled, where we think of the very incoming constants as being multiplication gates themselves (the result of $c_i \cdot 1$). Next, we

introduce the rule that, when from a node[1] more than one wire springs into another node, we will still think of this (bundle of) wire as one wire with one label. Again, the very incoming variables of the circuit will for this rule be thought of as nodes.

2.2 Making Polynomials (Quadratic Arithmetic Program)

With these rules in hand, we are ready to start cooking up the functions we need. First, we introduce the *target polynomial* $t(X)$ as follows:

Definition 2.1. For an arithmetic circuit over $\mathbb{F}$ with $\mathcal{M}$ the set of multiplicative gates, $|\mathcal{M}| = M$, the *target polynomial* $t(X)$ is defined as

$$t(X) = \prod_{i=1}^{M} (X - s_i), \tag{2.1}$$

for distinct $s_i \in \mathbb{F}$.

That is with each multiplicative gate g_i we associate a field element s_i where all s_i's are distinct. Note that for this definition to make sense we need $M \leq |\,\mathbb{F}\,|$.
Next, we are going to cook up three more polynomials: $\ell(X)$, $r(X)$ and $o(X)$, that will be associated to the left, right and output wires of the circuit respectively.
First, recall that only output wires of a multiplicative gate gets a label. When we identify all the *left* incoming wires of a gate g_i, we will however count all wires that go into it either directly (coming directly from a multiplicative gate) or *indirectly* (passing first through one or several additive gates before reaching g_i.) We will call the set of all these incoming left wires of gate g_i: $\mathcal{I}_{i,L}$, and similarly the set of such wire for the right incoming wires is written as $\mathcal{I}_{i,R}$.
Take for example the circuit in figure 1b. There we have:

$$\mathcal{I}_{2,L} = \{w_4\},\ \mathcal{I}_{2,R} = \{w_1, w_3\}.$$

Finally, let $c_{i,L,k}$ denote the fixed scalar that is multiplied with over a wire $w_k \in \mathcal{I}_{i,L}$, so that for every output w_g of a gate g_i we can write

$$w_i^{out} = \left(\sum_{k \in \mathcal{I}_{i,L}} c_{i,L,k} \cdot w_k \right) \cdot \left(\sum_{k \in \mathcal{I}_{i,R}} c_{i,R,k} \cdot w_k \right) \tag{2.2}$$

[1] we use the name node and gate interchangeably

Still in circuit 1b, this translates to

$$w_5 = (w_4) \cdot (w_1 + w_3)$$

For a gate g_i we will now set the output polynomial $o_i(X)$ by demanding that

$$o_i(s_j) = \delta_{ij} \tag{2.3}$$

that is: $o_i(s_j) = 1$ iff $i = j$ and $o_i(s_j) = 0$ otherwise. One solution[2] is:

$$o_i(X) = \prod_{m \neq i} \frac{X - s_m}{s_i - s_m}. \tag{2.4}$$

Also, for all wires in $\mathcal{I}_{g,L/R}$ we define polynomials by demanding[3]

$$\ell_i(s_j) = c_{j,L,i} \quad \text{for } k \text{ in } \mathcal{I}_{j,L}, \;\; 0 \text{ otherwise} \tag{2.5}$$

$$r_i(s_j) = c_{j,R,i} \quad \text{for } k \text{ in } \mathcal{I}_{j,R}, \;\; 0 \text{ otherwise} \tag{2.6}$$

The solution to these is straightforward although slightly more involved. We write down a solution for the case of $M = 2$:

$$\ell_1(X) = \quad \frac{c_{2,L,1} - c_{1,L,1}}{s_2 - s_1} X + \frac{s_1 c_{2,L,1} - s_2 c_{1,L,1}}{s_1 - s_2} \tag{2.7}$$

$$\ell_3(X) = \quad c_{2,L,3}(X - s_1) \tag{2.8}$$

and similar for $r_i(X)$.

Now we define the following sums:

$$o(X) = \quad \sum_{i=1}^{W} c_i^{out} o_i(X) \tag{2.9}$$

$$\ell(X) = \quad \sum_{i=1}^{W} c_i \ell_i(X) \tag{2.10}$$

$$r(X) = \quad \sum_{i=1}^{W} c_i r_i(X), \tag{2.11}$$

[2]This solution is by no means unique, but it is unique in this degree $d = M - 1$. Any positive power of $o_i(X)$ solves the condition also, or we could multiply with any polynomial $f(X/s_i)$ that has no constant term.

[3]Note that by abuse of notation, when we write $k \in \mathcal{I}_{j,L/R}$, we do not mean that k is an actual wire w_q, but k is the index q of that wire.

where W is the number of wires in the circuit, $(c_1, \ldots, c_W)$ is an assignment to the circuit, and c_i^{out} is the assignment to the wire coming out of gate g_i. Finally, we define

$$p(X) = \ell(X) \cdot r(X) - o(X). \tag{2.12}$$

This polynomial $p(X)$ will play the central role in the protocol: it's the function **P** will claim to know. This is due to the following observation: The construction of $o(X)$, $\ell(X)$ and $r(X)$ is such that, when evaluated at any of the s_i, we can readily check that, iff $(c_1, \ldots, c_W)$ is a valid assignment, then:

$$p(s_i) = \ell(s_i) \cdot r(s_i) - o(s_i) = 0, \tag{2.13}$$

or in other words $p(X)$ has at least all the roots of the *target polynomial* $t(X)$, or: $t(X)$ *divides* $p(X)$: To set up the protocol, we need the following theorem

Theorem 2.2. *An assignment* $(c_1, \ldots, c_W)$ *to a circuit with* W *wires is valid if and only if its* target polynomial $t(X)$ *divides* $p(X)$*:*

$$t(X) | \, p(X) \tag{2.14}$$

Hence, it is enough for **P** to prove to **V** that it knows $p(X)$ such that $t(X)|p(X)$.

We can think of the components $\ell_i(X)$ as spanning up a vector $\vec{\ell}$ (as similar for $r_i(X)$ and $o_i(X)$). Then, for an arithmetic circuit C, we sometimes refer to the collection

$$Q(C) = \left(\vec{\ell}, \vec{r}, \vec{o}, t(X)\right) \tag{2.15}$$

as the *Quadratic Arithmetic Program* of the circuit C.

2.3 Naive approach

The task for **P** is now to convince **V** that it knows $p(X)$ s.t. $t(X)|p(X)$. The input for the protocol will be an arithmetic circuit C. Before going further with a proper zk-SNARK protocol, we will first introduce a naive attempt, and see what's wrong with that.

The first thing that comes to mind when trying to come up with a (zero-knowledge) proof of knowledge for **P** to convince **V** that is knows a $p(X)$ s.t. $t(X)|p(X)$ for some given $t(X)$ is to just cook up a version of the Schnorr protocol, but in a version that would allow the secure communication of a polynomial. For an integer n, we know how to do a secure transfer to a verifier of the wire: we take a finite cyclic group G that is generated by g and we send $y = g^n$ to **V** . The verifier can then manipulate y using homomorphicity and check it has certain

properties, without learning what n is – solving the discrete logarithm is in NP.

For a polynomial $p(X)$ we could do something very simlilar. Here's what the equivalent of the Schnorr Sigma protocol for a proof the knowledge of a polynomial $p(X) = \sum_{i=0}^{d} c_i X^i$ of degree d, with coefficients in $\mathbb{F}$ and such that $t(X)|p(X)$, would look like.

1. Set-up:
$$bp = (p, G, g) \tag{2.16}$$
This is the agreed upon data that will be used throughout the protocol.
2. $\mathbf{V}$: takes $s \leftarrow \mathbb{F}$; sends to $\mathbf{P}$: $\{g^{s^i}\}_{i=0}^{d}$
3. $\mathbf{P}$: computes $h(X) = \frac{p(X)}{t(X)}$; computes $g^{p(s)} = \prod_{i=0}^{d} \left(g^{s^i}\right)^{c_i}$ and similarly computes $g^{h(s)}$; sends $(g^{p(s)}, g^{h(s)})$ to $\mathbf{V}$
4. $\mathbf{V}$: checks that $g^{p(s)} = \left(g^{h(s)}\right)^{t(s)}$. If true, it accepts the claim; if false rejects.

There are multiple ways in which this protocol is not satisfactory.

First, the prover's claim isn't very impressive. To know a degree d polynomial such that a given $t(X)$ of degree $d' < d$ divides it, isn't hard. We can just take $t(X)$ and multiply by any degree $d - d'$ polynomial, so we need more properties on $p(X)$ to make this hard, as we will later – or already have in the previous section. Still, this example is instructive, as it does contain the gist of the eventual protocols.

Second, although this protocol is *complete*, it is not *sound*, that is: there are ways for a malicious $\mathbf{P}$ to convince $\mathbf{V}$ of a false proof. For example, without knowing any polynomial at all, $\mathbf{P}$ could just take a random $r \leftarrow \mathbb{F}$ and send the pair $(g^{t(s)\cdot r}, g^r)$ to $\mathbf{V}$. This would be accepted. We need to cook up a way to force $\mathbf{P}$ into correctly following the protocol, or at least having a check to find out if it has.

Thirdly, the protocol is not zero-knowledge, and there are ways for $\mathbf{V}$ to extract information about $p(X)$: given $g^{p(s)}$, it can brute-force try c_i's to land at that result, with $g^{\sum_i c_i s^i}$.

The first problem is in a way addressed in the previous section, as we will eventually not be interested in just the naked polynomial $p(X)$ but in it's composition in terms of $\ell(X)$, $r(X)$ and $o(X)$, in a way that its properties will be hard enough be be non-trivial for the prover.

The second and third problem need a bit more attention. We'll introduce techniques to address these two issues right away.

Schnorr Protocol and the Fiat-Shamir Heuristic

The Schnorr protocol is probably the simplest and most instructive example of a *proof of knowledge*. The set-up is: given a finite cyclic group G of order p, generated by g, a prover $\mathbf{P}$ wants to prove to $\mathbf{V}$ that, given $y \in G$, it knows x such that $y = g^x$. The proof goes in three steps:

1. $\mathbf{P}$ takes a random $r \leftarrow G$ and sends $t = g^r$ to $\mathbf{V}$
2. $\mathbf{V}$ takes a random $c \leftarrow G$ and sends it to $\mathbf{P}$
3. $\mathbf{P}$ computes $s = r + cx$ and sends s to $\mathbf{V}$

After this communication, $\mathbf{V}$ accepts $\mathbf{P}$'s statement if $g^s = t \cdot y^c$. Such protocols are sometimes called Sigma protocols (drawing the 3-steps, it resembles the Greek capital Sigma). Note, however, that $\mathbf{P}$ can also just take a random x' that does not in fact solve the discrete log problem at hand, and it still has some chance of being accepted by $\mathbf{V}$. This probability is one over the size of the challenge space where c is taken from. We'll just state here that this protocol only zero-knowledge when the verifier's challenge space (where c is taken from) is small, e.g. when $c \in \{0, 1\}$. In this scenario, however, for the verifier to be properly convinced of $\mathbf{P}$'s statement (it is zero-knowledge, but not "sound"), and the protocol needs to be run many times to satisfy the soundness requirement – not a practical situation.
This protocol has another disadvantage: it is "interactive" – both $\mathbf{P}$ and $\mathbf{V}$ need to be on-line and need to exchange data. We can render this protocol non-interactive as follows: we replace step 2. by:

2. $\mathbf{P}$ computes $c = H(g, y, t)$ and sends it to $\mathbf{V}$, where H is a random oracle, or cryptographic hash function, e.g. $H =$ sha256 as of writing.

Now the Sigma protocol is turned into a non-interactive proof of knowledge. This is a very common procedure, and is sometimes called the Fiat-Shamir Heuristic.

2.3.1 Soundness by shifting

The gist of the soundness problem of our naive protocol is this (where we use a number case first and discuss polynomials later):

$\mathbf{V}$ wants to pick a g, and then have $\mathbf{P}$ take a value c – that's only known to $\mathbf{P}$ –, encrypt it

with g (i.e. compute g^c), and send g^c back to **V**. In the process, **P** wants to keep c hidden from **V**.

Now, how does **V** check that indeed **P** followed the instructions, and not just sent some bogus value? The simplest way would of course be for **P** to just supply c itself, but **P** will not agree on this as it is trying to achieve zero-knowledge. A better solution is this little protocol:

First, **V** takes a random α, and sends (g, g^α) to **P** . Then, it asks **P** to encrypt its secret c using both g and g^α, and send both encryptions $(z, z') = (g^c, (g^\alpha)^c)$ back. Now, **V** can check if indeed $z^\alpha = z'$. If that is the case, **V** can conclude that indeed **P** followed the instructions.

That **V** may conclude so is based on the "Knowledge of Coefficient Assumption" (KCA). This assumption basically states that there is no efficient way for a prover **P** to, given a pair (x, x^α), send back a pair (z, z') that also obeys $z^\alpha = z'$, except from – rather trivially – returning $(x^c, (x^\alpha)^c)$ for some choice of c. That is: when **P** gives back that pair it *knows* the coefficient c.

In making the zk-SNARK, we're not exchanging numbers but rather polynomials, so we need to generalize this protocol to allow for this case also. This is straightforward, and we give here the updated protocol that includes the shift:

1. **V** : takes an $s \leftarrow \mathbb{F}$ and an $\alpha \leftarrow \mathbb{F}$; sends to **P** : $\{(g^{s^i}, g^{\alpha s^i})\}_{i=0}^{d}$
2. **P** : computes $h(X) = \frac{p(X)}{t(X)}$; computes $g^{p(s)} = \prod_{i=0}^{d} \left(g^{s^i}\right)^{c_i}$ and similarly computes $g^{h(s)}$; it also computes $g^{\alpha p(s)} = \prod_{i=0}^{d} \left(g^{\alpha s^i}\right)^{c_i}$; sends $(z_p, z_{\alpha p}, z_h) = (g^{p(s)}, g^{\alpha p(s)}, g^{h(s)})$ to **V**
3. **V** : checks that $z_p = z_h^{t(s)}$ and that also $z_p^\alpha = z_{\alpha p}$. If true, it accepts; if false rejects.

2.3.2 Zero-knowledge by shifting

The protocol is not yet zero-knowledge: the verifier receives $g^{p(X)}$ and although it is hard to obtain $p(X)$, it is possible, given enough computation time and strength.

With the previous section in mind, this problem is actually pretty easy to address. In that protocol, instead of sending $(z_p, z_{\alpha p}, z_h)$, **P** could just first sample a random $\delta \leftarrow \mathbb{F}$, and send to **V** the triplet $(z_p^\delta, z_{\alpha p}^\delta, z_h^\delta)$. That way, the protocol is still sound and complete, but now it's also zero-knowledge.

2.3.3 Non-interactive with (nearly) trusted set-up

That our protocol is interactive is still problematic, and there is not yet a straightforward application of the Fiat-Shamir heuristic. A first attempt is to introduce a third, central party that

everybody – both **V** and bP – trust. From a cryptocurrency point of view this is of course not an interesting definitive solution, but we'll touch upon it here as it is an instructive example to build on.

2.4 Pinocchio Protocol

Recall that we started with building a circuit C from some *relation* R over $\mathbb{F}^N$, where N is the total number of constants that partake in the relation. Now, let $R = \{(\vec{u}, \vec{w})\}$ denote this relation, where we have split $N = N_1 + N_2$, and $\vec{u} \in \mathbb{F}^{N_1}, \vec{w} \in \mathbb{F}^{N_2}$. We refer to $\vec{u}$ as the *statement* and $\vec{w}$ as the *witness*. We will assume that solving the relation R is in NP, that is, given some statement $\vec{u}$, it is hard to find a $\vec{w}$ such that $(\vec{u}, \vec{w}) \in R$. We do assume, however, that given an $(\vec{u}, \vec{w})$, there exists a polynomial-time algorithm f

$$f : \mathbb{F}^{N_1} \times \mathbb{F}^{N_2} \to \{0, 1\}, \quad f((\vec{u}, \vec{w})) = 1 \text{ iff } (\vec{u}, \vec{w}) \in R, \;\; f((\vec{u}, \vec{w})) = 0 \;\text{ else} \tag{2.17}$$

The verifier will use this f to check provers claims. The prover will make public the statement, and will keep the witness to itself. We will lay out the protocol (CITE) in two steps:

2.4.1 Set-up

The zk-SNARK protocol starts by setting a list of data, denoted *bp*:

$$bp = (\, p, G, G_T, g_\star, e \,) \tag{2.18}$$

We will use this notation more throughout this text. In the list *bp*: p is a (large) prime number, G is a cyclic groups of order p, that is generated by $g_\star$: $\langle g_\star \rangle = G$; $e = e(g, h)$ is a *non-degenerate, bi-linear* map $e : G \times G \to G_T$, such that $e(g_\star, g_\star)$ generates G_T: $\langle s(g_\star, g_\star) \rangle = G_T$.
For G we will typically have an elliptic curve group in mind, that are is additively. We will write G_T multiplicatively. Then, the bi-linearity of e means that:

$$e(n_1 g, n_2 h) = e(g, h)^{n_1 n_2} \tag{2.19}$$

for all $g, h \in G$ and $n_1, n_2 \in \mathbb{Z}$.

KEY GENERATION

Input: Arithmetic circuit C with W wires

Thrusted third party generates: QAP $Q(C)$

1. Sample $\tau, \rho_\ell, \rho_r \leftarrow \mathbb{F}$. Set $\rho_o = \rho_\ell \rho_r$
2. Define the proving key: $\mathrm{pk} = (\mathrm{pk}_\ell, \mathrm{pk}_r, \mathrm{pk}_o, \mathrm{pk}_H)$,

$$\mathrm{pk}_\ell = (\ell_i(\tau)\rho_\ell g)_{i=0}^{W} = \left(\mathrm{pk}_{\ell,i}\right)_{i=0}^{m}$$

$$\mathrm{pk}_r = (r_i(\tau)\rho_r g)_{i=0}^{W} = \left(\mathrm{pk}_{r,i}\right)_{i=0}^{m}$$

$$\mathrm{pk}_o = (o_i(\tau)\rho_o g)_{i=0}^{W} = \left(\mathrm{pk}_{o,i}\right)_{i=0}^{m}$$

$$\mathrm{pk}_H = \left(\tau^i g\right)_{i=0}^{W} = \left(\mathrm{pk}_{H,i}\right)_{i=0}^{m}$$

3. Define the verification key: $\mathrm{vk} = (\mathrm{vk}_{IC}, \mathrm{vk}_Z)$,

$$\mathrm{vk}_{IC} = (\ell_i(\tau)\rho_\ell g)_{i=0}^{N} = (\mathrm{vk}_{IC,i})_{i=0}^{m}$$

$$\mathrm{vk}_Z = Z(\tau)\rho_o g$$

3 Sonic

Commitment

A commitment $\text{Com}(m)$ is a way for an actor $\mathbf{P}$ to take a secret message m, and seal it in a little protocol in a way that $\mathbf{P}$ is committed to m until the end of the protocol. It is the digital equivalent of putting your message in a sealed envelope, such that before the end of the ceremony,m stays hidden but can not be altered by $\mathbf{P}$, as in the final step, the envelope will be opened to check if indeed $\mathbf{P}$ stayed committed to m. It should work such that, once $\mathbf{P}$ has filed the commitment, it can not change m without someone noticing, while a the same time it should perfectly hide m for anyone else involved.
The first thing you might think of is: use a random oracle/hash function. Indeed, this checks to stated requirements. It has the problem however of being one-time useful only. Suppose a ceremony includes $\mathbf{P}$ making a commitment to some small integer n, committed to with $\text{Com}(n) = \text{sha}(n)$. At the end, $\mathbf{P}$ reveals n and anyone can check that indeed the hash function checks out. However, any time in future this same integer pops up, anyone will recognize the hash function, and the commitment is not hiding the message any more.
We can solve this by first raking a random string r, joining/concatenating it with m as $r||m$; applying the hash, $\text{Com}(m) = \text{sha}(r||m)$, and at the end releasing both r and m. If r's sampling space is large enough this will do. The practical problem is that this scheme is not *homomorpihc*: it doesn't obey $\text{Com}(m_1 \cdot m_2) = \text{Com}(m_1) \cdot \text{Com}(m_2)$ in any nice way. A homomorphic, hiding commitment scheme is given by *Pedersen's commitment scheme*:

1. Set-up: The committer $\mathbf{P}$ picks a large prime p, and a $q|p-1$ and considers the group $\mathbb{Z}_p^*$ generated by g, with $\mathbb{Z}_q \subset \mathbb{Z}_p^*$. $\mathbf{P}$ takes $a \leftarrow \mathbb{Z}_q$ and sets $h = g^a \mod p$. $(p, q, \mathbb{Z}_p^*, \mathbb{Z}_q, g, h)$ are all public: only a is secret.
2. $\mathbf{P}$ now *commits* to an $m \in \mathbb{Z}_q$ by taking $r \leftarrow \mathbb{Z}_q$ and making public:

$$c = g^m h^r \mod p \tag{3.1}$$

3. To, at the end of a ceremony, *open* of *reveal* the commitment, $\mathbf{P}$ publishes (m, r).

4 Beyond Sonic