A semi-technical overview of zk-Snarks and Sonic

Paul de Lange*

February 11, 2021

Abstract

draft

* *email:* 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 My first Sigma protocol

Valery is having an existential crisis. She's color blind, but she is starting to believe that maybe she really isn't, and that all her friends and family are just conspiring to trick her into believing she is.
Along comes Peter, her uncle, and Alice starts ventilating her conspiracy theory to him.

V : I trust you all no longer; I don't believe you can distinguish between what you call 'blue' and 'red' any better than I can. Unless you can offer me some proof.
P : Got it.
Peter takes out of his pocket two balls: one that's red, the other blue. Valery however, being color blind, sees two identical balls.
P : You see these two balls, and you can confirm they are identical?
V : Yes, indeed.
P : Let me convince you that I *can* distinguish between them. I will hand over the balls to you now. I want you to take one ball in each hand, and then hold the balls behind your back, so I can not see them. Then, I need you to decide yourself whether you swap the balls or not. After that, you present the balls to me again. Now, since I claim I can distinguish the balls, I should easily be able to tell whether you swapped the balls or not. However, in case I too see two identical balls, and can not see color as you suspect, I will not be able to tell whether you made the swap, and I won't be able to see.
Peter hand over the balls, blue in Valery's left hand; red in her right. Valery, per instruction, places the balls behind her back and decides to swap them. She presents the balls back to Peter.
P : You swapped them.
V : I did! OK, maybe you are right after all!

We have just witnessed our very first interactive proof of knowledge[1]. A proof of knowledge is a way for a prover **P** to convince a verifier **V** that it knows something, in this case that it knows how to distinguish between red and blue. In this instance, the proof (or protocol) was *interactive*[2]: Valery and Peter had to do a lot of talking and exchanging of balls and what not. What was nice about this proof, was that Peter only convinced Valery of the mere fact that it *does* know how to distinguish between red and blue. He did not explain *how* he did this. He did not disclose the mechanism or trick. This feature (convincing someone of your knowledge without

[1]Such a protocol is more abstractly referred to as a Sigma protocol, as the exchange $\mathbf{P} \to \mathbf{V} \to \mathbf{P}$ can be drawn to look like the Greek capital letter Σ.
[2]We refer to the dictionary where some terms and symbols are explained for the novice

spilling any other information whatsoever, is commonly referred to as proving something *in zero knowledge*, or just a *zero knowledge proof*.

There are some obvious problems in the proof. The first and most important one is that Valery is convinced a little to easily. Suppose, namely, that Peter *is* lying and in fact can not distinguish between red and blue at all. That means that at the final step of the protocol above, all he can do is guess whether Valery has made the swap or not. And in this scenario, that still leaves him with a 50% chance of guessing right! So overall, if Peter is lying and Valery's conspiracy theory is actually correct, Valery has a 50% chance of falsely buying Peter's nonsense. That's not acceptable.
This problem can readily be addressed however. We simply repeat the protocol! If we run this for a second time, then a lying Peter has to guess right in both instances of the protocol, and the chances of him guessing correctly are reduced to 25%. More generally, Valery can decide to ask for N instances of the protocol, hence reducing the chance of a lying Peter to pass this test to a slim:

$$\text{Chance of Peter guessing correctly at all } N \text{ instances of the verification protocol} = \left(\frac{1}{2}\right)^N$$

His chance reduces exponentially in N, and now Valery can just pick a high enough N to be convinced after all N runs.
As it happens, solving this problem did introduce a new one, one of practicality: the protocol becomes cumbersome, lengthy and boring, and it will be hard to enthuse Perter to partake.[3]
Another problem is that this protocol is rather specific, and it is not easy to see how we can take it and use it for other purposes that convincing the color blind they are, really, color blind.

This is a blog on cryptographic methods for cryptocurrency – Veil in particualar – and this blog is not intended for only the color blind, so why should we care about this example?
Veil is a cryptocurrency that offers privacy. That's very nice, but it introduces a puzzle: How can two parties that exchange a sum of Veil convince the rest of the Veil community that their transaction was bona fide (this is their "secret", like seeing color in the example), without disclosing their identity, the sum exchanged, or in fact without disclosing any type of (meta-)data about this transaction? To address this problem, we will use exactly the type of zero knowledge

[3]You could overcome this by using a large number of pairs of red and blue balls. This reduces the time it takes, but does increase the 'weight' – or bit size – of the balls to be exchanged in the protocol, and exchanging heavy information is problematic also.

protocols as above, and this document will delve into this technique a little deeper.

3 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 out coming 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) arithmetic circuit
2. From this arithmetic circuit, construct a specific polynomial $p(x, y, \ldots)$
3. Rephrase the satisfaction of the constraints in terms of the fact the p is divisible 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.

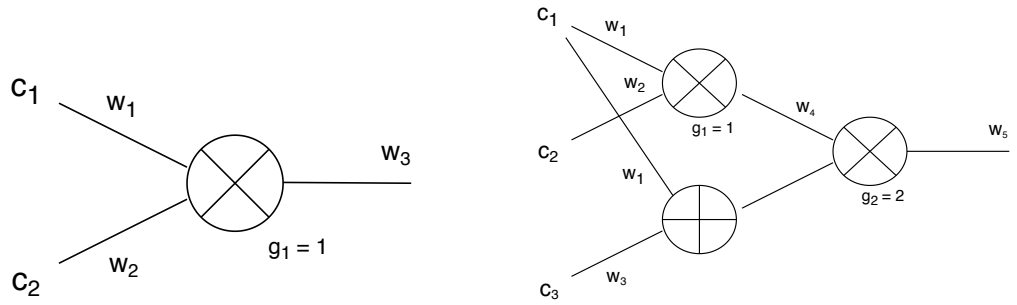
The subsections to come will follow these steps, beginning with the introduction of arithmetic circuits

3.1 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



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

Figure 1: Arithmetic circuits

In this circuit, we identify one multiplication node (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[4] 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.

3.2 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 3.1. 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

$$t(X) = \prod_{i=1}^{M} (X - s_i), \tag{3.1}$$

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

$$w_i^{out} = \left(\sum_{k \in \mathcal{I}_{i,L}} c_{i,L,k} \cdot w_k \right) \cdot \left(\sum_{k \in \mathcal{I}_{i,R}} c_{i,R,k} \cdot w_k \right) \tag{3.2}$$

[4] 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

$$o_i(s_j) = \delta_{ij} \tag{3.3}$$

that is: $o_i(s_j) = 1$ iff $i = j$ and $o_i(s_j) = 0$ otherwise. One solution[5] is:

$$o_i(X) = \prod_{m \neq i} \frac{X - s_m}{s_i - s_m}. \tag{3.4}$$

Also, for all wires in $\mathcal{I}_{g,L/R}$ we define polynomials by demanding[6]

$$\ell_i(s_j) = c_{j,L,i} \quad \text{for } k \text{ in } \mathcal{I}_{j,L}, \;\; 0 \text{ otherwise} \tag{3.5}$$

$$r_i(s_j) = c_{j,R,i} \quad \text{for } k \text{ in } \mathcal{I}_{j,R}, \;\; 0 \text{ otherwise} \tag{3.6}$$

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

$$\ell_1(X) = \quad \frac{c_{2,L,1} - c_{1,L,1}}{s_2 - s_1} X + \frac{s_1 c_{2,L,1} - s_2 c_{1,L,1}}{s_1 - s_2} \tag{3.7}$$

$$\ell_3(X) = \quad c_{2,L,3}(X - s_1) \tag{3.8}$$

and similar for $r_i(X)$.

Now we define the following sums:

[5] 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.

[6] 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.

$$o(X) = \sum_{i=1}^{W} c_i^{out} o_i(X) \tag{3.9}$$

$$\ell(X) = \sum_{i=1}^{W} c_i \ell_i(X) \tag{3.10}$$

$$r(X) = \sum_{i=1}^{W} c_i r_i(X), \tag{3.11}$$

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

$$p(X) = \ell(X) \cdot r(X) - o(X). \tag{3.12}$$

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:

$$p(s_i) = \ell(s_i) \cdot r(s_i) - o(s_i) = 0, \tag{3.13}$$

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 3.2. *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* $p(X)$*:*

$$t(X) | \, p(X) \tag{3.14}$$

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

$$Q(C) = \left(\vec{\ell}, \vec{r}, \vec{o}, t(X)\right) \tag{3.15}$$

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

3.3 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 similar. 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:
$$bp = (p, G, g) \tag{3.16}$$
This is the agreed upon data that will be used throughout the protocol.
2. **V** : takes $s \leftarrow \mathbb{F}$; sends to **P** : $\{g^{s^i}\}_{i=0}^{d}$
3. **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 **V**
4. **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 **P** to convince **V** of a false proof. For example, without knowing any polynomial at

all, **P** could just take a random $r \leftarrow \mathbb{F}$ and send the pair $(g^{t(s)\cdot r}, g^r)$ to **V** . This would be accepted. We need to cook up a way to force **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 **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.

3.3.1 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.

3.3.2 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.

3.3.3 Non-interactive by pairing

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 **P** – 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, and also immediately introducing the trusted set-up with a "nearly-trustless" set-up.

First, lets call the trusted party **T** . This is what the first part of our non-interactive protocol with trusted set-up, for proving **P** knows a degree-d polynomial $g(X)$ with $t(X)|g(X)$ for public $t(X)$, looks like:

1. Set-up: $bp = (p, G, g)$
2. **T** takes $s, \alpha \leftarrow \mathbb{F}$; computes $(g^{\alpha}, g^{s^i}, g^{\alpha s^i}, g^{t(X)})$; makes this public as follows:

$$\text{pk} = (g^{s^i}, g^{\alpha s^i}) \tag{3.17}$$

$$\text{vk} = (g^{t(s)}, g^{\alpha}) \tag{3.18}$$

   where we call pk and vk the proving key and verification key respectively

3. **T** deletes (s, α)

With this set-up, we aim at a scenario where **V** needs to do nothing but check something, and needs no communication with **P** . The idea is that, with the help of **T**'s proving key, **P** computes $g^{p(s)}$, $g^{\alpha p(s)}$, $g^{h(s)}$, and then **V** gets to verify just like before, now using the verification key. **V** needs one more instrument to facilitate this verification: a bilinear pairing.

Definition 3.3. A map $e : G_1, G_2 \to H$, with G_1, G_2, H finite groups, is called bilinear if

$$e(g_1^n, g_2^m) = e(g_1, g_2)^{mn}.$$

We call e a *pairing* if it is non-degenerate (there's no $g \in G_1$ such that $e(g, g') = 1$ for all $g' \in G_2$) and if it furthermore can be efficiently computed.

It is first of all not obvious such a pairing should exist for every triplet of groups (G_1, G_2, H). We will however, in cryptography, be interested in the scenario where we have explicit construction of such a pairing in hand. This is the case when we restrict the groups to descend from certain elliptic curves, where there is such a construction called the Tate pairing. It is beyond the scope of this text to delve deeper into elliptic curves and their pairings, also because there are great texts out there, so throughout we will think of the groups and the pairings as given. With this pairing, that will be a given, we are ready to complete the verification step of our zk-snark:

1. Set-up: $bp = (p, G, g, e)$ with $e : G \times G \to G$
2. After the **T** has generated the proving key and verification key as above, and **P** used pk to compute $\pi = (g^{\delta p(s)}, g^{\delta \alpha p(s)}, g^{\delta h(s)})$, a triplet that **V** will parse as $\pi = (g^p, g^{\alpha p}, g^h)$.
3. **V** checks that $e(g^p, 1) = e(g^t, g^h)$ and $e(g^{p'}, 1) = e(g^p, g^\alpha)$

With this last step, we solved a lot of our problems. There are two more problem. First, is the trusted set-up. From a crypto coin point of view, this is serious and one might say we have in the process threw out the baby with the bathwater. One way to solve this is to not introduce on trusted party, but introduce a lot of them, and start the protocol with a ceremony, where all trusted party members participate in generating the proving key and verification key. There is a neat way that makes sure the keys will be corrupt only is *all* participants of the ceremony all malevolent, that is trusting one out of the entire community would be enough (REF HERE). Although this is definitely an improved scenario, it still is not ideal. This process can be lengthy or messy, and it makes it hard to refresh the keys if needed, so we will look for another approach. Secondly, we still have the problem of triviality: the prover is still not proving anything fascinating. We will address this issue now, introducing the Pinocchio protocol.

3.4 Pinocchio Protocol

We now have everything in place to start introducing some actual useful protocols, starting with the Pinocchio protocol [PHGR16]. The Pinocchio protocol is the first practical zk-Snark protocol that was published. It contains the ingredients that we introduced in the previous sections, but also address the last standing open challenge. Whereas in the previous section, the prover was not very impressive in its claim; the Pinocchio protocol upgrades that scenario, really making the protocol about the original problem: proving knowledge of some constants in a relation. This protocol will be interactive and can be made non-interactive by the trusted party method as described in the previous section. We will follow the notation of [PHGR16] quite closely, so that the reader can easily switch to that reference. Like any public verifiable computation scheme, the protocol will have three steps: Key generation $((pk; vk) \leftarrow KeyGen(F))$, Compute $((y; \pi_y) \leftarrow Compute(pk, u))$ and Verify $((0; 1) \leftarrow Verify(vk, u, y, \pi_y))$.
The set-up will be that we have some elementary relation F with a total of N input/output variables in the field $\mathbb{F}$.

Pinocchio protocol

Input: Relation F with N input/output variables c_i.

1. Set $bp = (p, G, G_T, g, e)$, where p is a prime number, G, G_T are finite groups, $|G| = p$ and g generates G: $\langle g \rangle = G$, and e a non-trivial bilinear map $e : G \times G \to G_T$.
2. Convert F into an arithmetic circuit C and construct the corresponding degree-d, size-m^7 QAP $Q(C) = \left(t(X), \vec{\ell}, \vec{r}, \vec{o}\right)$. Set $I_{mid} = \{N+1, \ldots, m\}$
3. **Key Generation** $SRS = (pk, vk) \leftarrow \text{KeyGen}(F)$
   Sample $s, r_\ell, r_o, \beta, \alpha_\ell, \alpha_r, \alpha_o, \gamma \leftarrow \mathbb{F}$. Set $r_o = r_\ell\, r_r$ and define $g_\ell = g^{r_\ell}, g_r = g^{r_r}, g_o = g^{r_o}$. Now define the proving key pk as:

$$pk = \big(\{g_\ell^{\ell_k(s)}\}, \{g_r^{r_k(s)}\}, \{g_o^{o_k(s)}\}, \{g_\ell^{\alpha_\ell \ell_k(s)}\}, \{g_r^{\alpha_r r_k(s)}\},$$
$$\{g_o^{\alpha_o o_k(s)}\}, \{g^{s^i}\}, \{g_\ell^{\beta \ell_k(s)} g_r^{\beta r_k(s)} g_o^{\beta o_k(s)}\}\big)_{k \in I_{mid},\, i \in [0,\ldots,d]}$$

   and the verification key vk as:

$$vk = \left(g, g^{\alpha_\ell}, g^{\alpha_r}, g^{\alpha_o}, g^{\gamma}, g^{\beta\gamma}, g_o^{t(s)}, \{g_\ell^{\ell_k(s)} g_r^{r_k(s)} g_o^{o_k(s)}\}_{k \in [0,\ldots,N]}\right)$$

4. **Compute** $(y, \pi_y) \leftarrow \text{Compute}(pk, u)$ Given some input u for the relation F, the prover evaluates the arithmetic circuit to obtain $y \leftarrow F(u)$ and thus learns the values $\{c_i\}_{i \in [0,\ldots,m]}$ that are assigned to the wires.
   Prover computes $h(X)$ such that $p(X) = h(X)t(X)$ and cooks up the proof π_y:

$$\pi_y = \big(g_\ell^{\ell_{mid}(s)}, g_r^{r_{mid}(s)}, g_o^{o_{mid}(s)}, h^{h(s)}, g_\ell^{\alpha_\ell \ell_{mid}}, g_r^{\alpha_r r_{mid}(s)},$$
$$g_o^{\alpha_o o_{mid}(s)}, g_\ell^{\beta \ell_{mid}(s)} g_r^{\beta r_{mid}(s)} g_o^{\beta o_{mid}(s)}\big)$$

   where $\xi_{mid}(x) = \sum_{k \in I_{mid}} c_k\, \xi_k(x)$ for $\xi \in \{\ell, r, o\}$
5. **Verify** $\{0, 1\} \leftarrow \text{Verify}(vk, u, y, \pi_y)$
   The verifier parses the proof π_y as

$$\pi = (g^L, g^R, g^O, g^H, g^{L'}, g^{R'}, g^{O'}, g^Z).$$

   Computes: $g_\ell^{\ell_{I/O}(s)} = \prod_{k=0}^{N} \left(g_\ell^{\ell_k(s)}\right)^{c_k}$ $i, c_0 = 1$ (and similar for r, o). Now checks:
   - Divisibility:

$$e(g_\ell^{\ell_{I/O}(s)} g^L, g_r^{r_{I/O}(s)} g^R) = e(g_o^{t(s)}, g^H) e(g_o^{o_{I/O}(s)} g^O, g)$$

   - Shifting:

$$e(g^{L'}, g) = e(g^L, g^{\alpha_\ell}) \quad \text{(and same for R and O)}$$

   - Coefficients:

$$e(g^Z, e^\gamma) = e(g^L g^R g^O, g^{\beta\gamma})$$

This is the Pinocchio protocol. It is the first bona-fide zk-SNARK. It does now prove something NP-hard, but it still has some problems.
The set-up is still trusted. Now, we can still upgrade this trusted set-up to one where we distribute the trust over some network of clients, up to the point where we only need to trust one out of many to trust the entire set-up. One thing we would however like to introduce is some manner of being able to update the trusted set-up. Currently, we first have to start some ceremony, where one group of clients first has to generate the set of keys

$$SRS = (pk, vk).^{8}$$

Once this SRS is constructed, however, later in the process it cannot be altered. This is, from a security/crytpo point of view somewhat counter-intuitive and we would like a scenario where this SRS can be updated at any time without hassle. Sonic will do this, based on the ideas of Groth et all [REF]. We will now introduce Sonic.

4 Sonic

Sonic is an improvement of the Pinocchio protocol. Its methods aren't radically different but there are some new ingredients that we will list. We will not go into detail about this, but the protocol we will introduce here has one shortcoming that will be fixed in the next section. This has to do with computation times. We did not talk about this too much, but for zk-SNARKs to be practical they need to be fast. SuperSonic will basically be a speed-improved version of Sonic.
The new ingredients, that are referred to as Building Blocks in [REF], are:

- **Bilinear Groups** The form of bp is: $bp = (p, G_1, G_2, G_T, e, g, h)$, where $G_1 = \langle g \rangle$, $G_2 = \langle h \rangle$, $G_T = \langle e(g,h) \rangle$, and G_1 and G_2 are such that there is no
- **SRS** The general form of the SRS is

  $$SRS = \left\{ \{g^{x^i}\}_{i=-d}^{d}, \{g^{\alpha x^i}\}_{i=-d, i\neq 0}^{d}, \{h^{x^i}, h^{\alpha x^i}\}_{i=-d}^{d}, e(g, h^{\alpha}) \right\}$$

  Note the negative powers, and also that g^{α} is not part of the SRS.
- **Polynomial Commitment Scheme** It uses a *polynomial commitment scheme*. We explain this scheme in subsection (4.3).

[8] This set is commonly referred to as the SRS: the structured reference string

The way the constraints are mapped to a polynomial is also different in Sonic. We will now discuss these new ingredients in some more detail.

4.1 System of Constraints

The way that the constraint relation F is packed, is a little different in Sonic.

We start with three vectors, $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{F}^n$, such that

$$\mathbf{a} \circ \mathbf{b} = \mathbf{c}, \tag{4.1}$$

where $\circ$ means component-wise multiplication. To accommodate the linear constraints, we further introduce Q linear constraints

$$\mathbf{a} \cdot \mathbf{u}_q + \mathbf{b} \cdot \mathbf{v}_q + \mathbf{c} \cdot \mathbf{w}_q = k_q \tag{4.2}$$

where $q = 1, \ldots, Q$ and $\mathbf{u}_q, \mathbf{v}_q, \mathbf{w}_q \in \mathbb{F}^n$, $k_q \in \mathbb{F}_p$. We will quickly discuss a couple of examples to illustrate how this works.

- Even though it looks slightly patronizing, let's start with the very, very most simple relation. A linear relation in one variable x_1 that is known to the prover:

  $$x_1 = r$$

  for some $r \in \mathbb{F}_p$. Now of course, this example is trivial, as the solution is equal to the relation itself! It is however instructive to translate this to the language of Sonic's system of constraints. We should think of this as already having one "multiplicative gate", and $n = 1$. The multiplicative relation is $a_1 \cdot b_1 = c_1$, so our relation should really be seen as $1 \cdot b_1 = r$. So there's a left input variable a_1 that's constrained to $a_1 = 1$, there's a right input b_1 (that we labeled x_1 in our formulation) that has no constraint, and finally there's an output c_1 constrained to $c_1 = r$. So, $Q = 2$, and $(u_1, v_1, w_1; k_1) = (1, 0, 0; 1)$ and $(u_2, v_2, w_2; k_2) = (0, 0, 1; r)$. Note that the system is not unique. A system that describes the same relation is $(u'_1, v'_1, w'_1; k'_1) = (1, 0, 0; 1)$, $(u'_2, v'_2, w'_2; k'_2) = (0, 1, 0; r)$ and $(u'_3, v'_3, w'_3; k'_3) = (0, 0, 1; 0)$. Also, we have the freedom of picking what we call left and right inputs. We will call all such systems that describe the same relation equivalent.
- The next simplest case is arguably that of a system of two variables x_1, x_2 that obey a

linear constraint $x_1 + x_2 = r$. In this new system, we can identify the vectors as

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}, \mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

Without loss of generality, we will identify our variables $x_{1,2}$ with $b_{1,2}$ respectively. Will will then consider out system consisting of having two multiplicative gates that our not connected, with some additional constraints on top.

The first constraint is $a_1 = 1$. So: $\mathbf{u}_1 = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$ and $\mathbf{v}_1 = \mathbf{w}_1 = 0$, $k_1 = 1$. The second is $a_2 = 1$: $\mathbf{u}_2 = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$ and $\mathbf{v}_2 = \mathbf{w}_2 = 0$, $k_2 = 1$. Then we have $b_1 - c_1 = 0$ and $b_2 - c_2 = 0$: $u_3 = 0$ and $\mathbf{v}_3 = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right)$, $\mathbf{w}_3 = \left(\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right)$, $k_3 = 0$ and similar for $q = 4$. Then finally we have the actual additive constraint: $b_1 + b_2 - r = 0$: $u_5 = 0$, $\mathbf{v}_5 = \left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right)$, $w_5 = 0$ and $k_5 = r$.

These examples, that touch upon just the most trivial of cases, show that these arithmetic circuits really pull apart their elementary relations to their most primitive roots, and in the process make them look quite intricate cumbersome.
We will now move on tho show how this, in Sonic, is translated into polynomials.

4.2 Multivariate Polynomials

From the relations (4.1) and (4.2) we will now proceed to construct polynomials that will actually be used in the zero knowledge protocol.

4.3 Polynomial Commitment Scheme

Commitment

A commitment $\text{Com}(m)$ is a way for an actor **P** to take a secret message m, and seal it in a little protocol in a way that **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 **P** , as in the final step, the envelope will be opened to check if indeed **P** stayed committed to m. It should work such that, once **P** has filed the commitment, it can not change m without someone noticing, while at 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 **P** making a commitment to some small integer n, committed to with $\text{Com}(n) = \text{sha}(n)$. At the end, **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 $\text{Com}(m_1 \cdot m_2) = \text{Com}(m_1) \cdot \text{Com}(m_2)$. This is needed if we want to be able to update out commitment. A homomorphic, hiding commitment scheme is given by *Pedersen's commitment scheme*:

1. Set-up: The committer **P** picks a large prime p, and a $q|p-1$ and considers the group $\mathbb{Z}_p^*$ generated by g, with $\mathbb{Z}_q \subset \mathbb{Z}_p^*$. **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. Commit: **P** now *commits* to an $m \in \mathbb{Z}_q$ by taking $r \leftarrow \mathbb{Z}_q$ and making public:

$$c = g^m h^r \mod p \tag{4.3}$$

3. Open: To, at the end of a ceremony, *open* of *reveal* the commitment, **P** publishes (m, r).

5 Beyond Sonic

References

[PHGR16] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, February 2016. Research Highlight.