zkSNARKs in a Nutshell

Christian Reitwießner
chris@ethereum.org

Abstract

The possibilities of zkSNARKs are impressive; you can verify the correctness of computations without having to execute them, and you will not even learn what was executed – just that it was done correctly. Unfortunately, most explanations of zkSNARKs resort to hand-waving at some point, and thus they remain something "magical", suggesting that only the most enlightened actually understand how and why (and if?) they work. The reality is that zkSNARKs can be reduced to four simple techniques, and this article aims to explain them. Anyone who can understand how the RSA cryptosystem works, should also get a pretty good understanding of currently employed zkSNARKs.

As a very short summary, zkSNARKs as currently implemented, have 4 main ingredients (don't worry, we will explain all the terms in later sections):

A) Encoding as a polynomial problem

The program that is to be checked is compiled into a quadratic equation of polynomials: $t(x)h(x) = w(x)v(x)$, where the equality holds if and only if the program is computed correctly. The prover wants to convince the verifier that this equality holds.

B) Succinctness by random sampling

The verifier chooses a secret evaluation point s to reduce the problem from multiplying polynomials and verifying polynomial function equality to simple multiplication and equality check on numbers: t(s)h(s) = w(s)v(s)

This reduces both the proof size and the verification time tremendously.

C) Homomorphic encoding / encryption

An encoding/encryption function E is used that has some homomorphic properties (but is not fully homomorphic, something that is not yet practical). This allows the prover to compute $E(t(s))$, $E(h(s))$, $E(w(s))$, $E(v(s))$ without knowing s, she only knows $E(s)$ and some other helpful encrypted values.

D) Zero Knowledge

The prover obfuscates the values $E(t(s))$, $E(h(s))$, $E(w(s))$, $E(v(s))$ by multiplying with a number so that the verifier can still check their correct structure without knowing the actual encoded values.

The very rough idea is that checking $t(s)h(s) = w(s)v(s)$ is identical to checking $t(s)h(s)k = w(s)v(s)k$ for a random secret number k (which is not zero), with the difference that if you are sent only the numbers $(t(s)h(s)k)$ and $(w(s)v(s)k)$, it is impossible to derive $t(s)h(s)$ or $w(s)v(s)$.

This was the hand-waving part so that you can understand the essence of zkSNARKs, and now we get into the details.

1 RSA and Zero-Knowledge Proofs

Let us start with a quick reminder of how RSA works, leaving out some nit-picky details. Remember that we often work with numbers modulo some other number instead of full integers. The notation here is "$a + b \equiv c \pmod{n}$", which means "$(a + b) \,\%\, n = c \,\%\, n$". Note that the "(mod n)" part does not apply to the right hand side "c" but actually to the "$\equiv$"" and all other "$\equiv$" in the same equation. This makes it quite hard to read, but I promise to use it sparingly. Now back to RSA:

The prover comes up with the following numbers:

- p, q: two random secret primes
- $n := pq$
- d: random number such that $1 < d < n - 1$
- e: a number such that $de \equiv 1 \pmod{(p-1)(q-1)}$.

The public key is (e, n) and the private key is d. The primes p and q can be discarded but should not be revealed.

The message m is encrypted via

$$E(m) := m^e \,\%\, n$$

and $c = E(m)$ is decrypted via

$$D(c) := c^d \,\%\, n.$$

Because of the fact that $c^d \equiv (m^e \,\%\, n)^d \equiv m^{ed} \pmod{n}$ and multiplication in the exponent of m behaves like multiplication in the group modulo $(p-1)(q-1)$, we get $m^{ed} \equiv m \pmod{n}$. Furthermore, the security of RSA relies on the assumption that n cannot be factored efficiently and thus d cannot be computed from e (if we knew p and q, this would be easy).

One of the remarkable feature of RSA is that it is *multiplicatively homomorphic*. In general, two operations are homomorphic if you can exchange their order without affecting the result. In the case of homomorphic encryption, this is the property that you can perform computations on encrypted data. *Fully homomorphic encryption*, something that exists, but is not practical yet, would allow to evaluate arbitrary programs on encrypted data. Here, for RSA, we are only talking about

group multiplication. More formally: $E(x)E(y) \equiv x^e y^e \equiv (xy)^e \equiv E(xy) \pmod{n}$, or in words: The product of the encryption of two messages is equal to the encryption of the product of the messages.

This homomorphicity already allows some kind of zero-knowledge proof of multiplication: The prover knows some secret numbers x and y and computes their product, but sends only the encrypted versions $a = E(x), b = E(y)$ and $c = E(xy)$ to the verifier. The verifier now checks that $(ab) \,\%\, n \equiv c \,\%\, n$ and the only thing the verifier learns is the encrypted version of the product and that the product was correctly computed, but she neither knows the two factors nor the actual product. If you replace the product by addition, this already goes into the direction of a blockchain where the main operation is to add balances.

1.1 Interactive Verification

Having touched a bit on the zero-knowledge aspect, let us now focus on the other main feature of zkSNARKs, the succinctness. As you will see later, the succinctness is the much more remarkable part of zkSNARKs, because the zero-knowledge part will be given "for free" due to a certain encoding that allows for a limited form of homomorphic encoding.

SNARKs are short for *succinct non-interactive arguments of knowledge*. In this general setting of so-called interactive protocols, there is a *prover* and a *verifier* and the prover wants to convince the verifier about a statement (e.g. that $f(x) = y$) by exchanging messages. The generally desired properties are that no prover can convince the verifier about a wrong statement (*soundness*) and there is a certain strategy for the prover to convince the verifier about any true statement (*completeness*). The individual parts of the acronym have the following meaning:

- Succinct: the sizes of the messages are tiny in comparison to the length of the actual computation
- Non-interactive: there is no or only little interaction. For zkSNARKs, there is usually a setup phase and after that a single message from the prover to the verifier. Furthermore, SNARKs often have the so-called "public verifier" property meaning that anyone can verify without interacting anew, which is important for blockchains.
- ARguments: the verifier is only protected against computationally limited provers. Provers with enough computational power can create proofs/arguments about wrong statements (Note that with enough computational power, any public-key encryption can be broken). This is also called "computational soundness", as opposed to "perfect soundness".
- of Knowledge: it is not possible for the prover to construct a proof/argument without knowing a certain so-called *witness* (for example the address she wants to spend from, the preimage of a hash function or the path to a certain Merkle-tree node).

If you add the *zero-knowledge* prefix, you also require the property (roughly speaking) that during the interaction, the verifier learns nothing apart from the validity of the statement. The verifier especially does not learn the *witness string* – we will see later what that is exactly.

As an example, let us consider the following transaction validation computation:

$f(\sigma_1, \sigma_2, s, r, v, p_s, p_r, v) = 1$ if and only if σ_1 and σ_2 are the root hashes of account Merkle-trees (the pre- and the post-state), s and r are sender and receiver accounts and p_s, p_r are Merkle-tree proofs that testify that the balance of s is at least v in σ_1 and they hash to σ_2 instead of σ_1 if v is moved from the balance of s to the balance of r.

It is relatively easy to verify the computation of f if all inputs are known. Because of that, we can turn f into a zkSNARK where only σ_1 and σ_2 are publicly known and (s, r, v, p_s, p_r, v) is the witness string. The zero-knowledge property now causes the verifier to be able to check that the prover knows some witness that turns the root hash from σ_1 to σ_2 in a way that does not violate any requirement on correct transactions, but she has no idea who sent how much money to whom.

The formal definition (still leaving out some details) of zero-knowledge is that there is a *simulator* that, having also produced the setup string, but does not know the secret witness, can interact with the verifier – but an outside observer is not able to distinguish this interaction from the interaction with the real prover.

2 NP and Complexity-Theoretic Reductions

In order to see which problems and computations zkSNARKs can be used for, we have to define some notions from complexity theory. If you do not care about what a "witness" is, what you will *not* know after "reading" a zero-knowledge proof or why it is fine to have zkSNARKs only for a specific problem about polynomials, you can skip this section.

2.1 P and NP

First, let us restrict ourselves to functions that only output 0 or 1 and call such functions *problems*. Because you can query each bit of a longer result individually, this is not a real restriction, but it makes the theory a lot easier. Now we want to measure how "complicated" it is to solve a given problem (compute the function). For a specific machine implementation M of a mathematical function f, we can always count the number of steps it takes to compute f on a specific input x – this is called the *runtime* of M on x. What exactly a "step" is, is not too important in this context. Since the program usually takes longer for larger inputs, this runtime is always measured in the size or length (in number of bits) of the input. This is where the notion of e.g. an "n^2 algorithm" comes from – it is an algorithm that takes at most n^2 steps on inputs of size n. The notions "algorithm" and "program" are largely equivalent here.

Programs whose runtime is at most n^k for some k are also called "polynomial-time programs".

Two of the main classes of problems in complexity theory are P and NP:

> P is the class of problems L that have polynomial-time programs.

Even though the exponent k can be quite large for some problems, P is considered the class of "feasible" problems and indeed, for non-artificial problems, k is usually not larger than 4. Verifying a bitcoin transaction is a problem in P, as is evaluating a polynomial (and restricting the value to 0 or 1). Roughly speaking, if you only have to compute some value and not "search" for something, the problem is almost always in P. If you have to search for something, you mostly end up in a class called NP.

2.2 The Class NP

There are zkSNARKs for all problems in the class NP and actually, the practical zkSNARKs that exist today can be applied to all problems in NP in a generic fashion. It is unknown whether there are zkSNARKs for any problem outside of NP.

All problems in NP always have a certain structure, stemming from the definition of NP:

> NP is the class of problems L that have a polynomial-time program V that can be used to verify a fact given a polynomially-sized so-called witness for that fact. More formally:
>
> $L(x) = 1$ if and only if there is some polynomially-sized string w (called the *witness* such that $V(x, w) = 1$

As an example for a problem in NP, let us consider the problem of boolean formula satisfiability (SAT). For that, we define a boolean formula using an inductive definition:

- any variable x_1, x_2, x_3, ... is a boolean formula (we also use any other character to denote a variable)
- if f is a boolean formula, then $\neg f$ is a boolean formula (negation)
- if f and g are boolean formulas, then $(f \wedge g)$ and $(f \vee g)$ are boolean formulas (conjunction / and, disjunction / or).

The string "$((x_1 \wedge x_2) \wedge \neg x_2)$" would be a boolean formula.

A boolean formula is *satisfiable* if there is a way to assign truth values to the variables so that the formula evaluates to true (where $\neg$true is false, $\neg$false is true, true $\wedge$ false is false and so on, the regular rules). The satisfiability problem SAT is the set of all satisfiable boolean formulas.

> SAT(f) := 1 if f is a satisfiable boolean formula and 0 otherwise

The example above, "$((x_1 \wedge x_2) \wedge \neg x_2)$", is not satisfiable and thus does not lie in SAT. The witness for a given formula is its satisfying assignment and verifying that a variable assignment is satisfying is a task that can be solved in polynomial time.

2.3 P = NP?

If you restrict the definition of NP to witness strings of length zero, you capture the same problems as those in P. Because of that, every problem in P also lies in NP. One of the main tasks in complexity theory research is showing that those two classes are actually different – that there is a problem in NP that does not lie in P. It might seem obvious that this is the case, but if you can prove it formally, you can win US$ 1 million. Oh and just as a side note, if you can prove the converse, that P and NP are equal, apart from also winning that amount, there is a big chance that cryptocurrencies will cease to exist from one day to the next. The reason is that it will be much easier to find a solution to a proof of work puzzle, a collision in a hash function or the private key corresponding to an address. Those are all problems in NP and since you just proved that P = NP, there must be a polynomial-time program for them. But this article is not to scare you, most researchers believe that P and NP are not equal.

2.4 NP-Completeness

Let us get back to SAT. The interesting property of this seemingly simple problem is that it does not only lie in NP, it is also NP-complete. The word "complete" here is the same complete as in "Turing-complete". It means that it is one of the hardest problems in NP, but more importantly – and that is the definition of NP-complete – an input to any problem in NP can be transformed to an equivalent input for SAT in the following sense:

For any NP-problem L there is a so-called *reduction function* f, which is computable in polynomial time such that:

$$L(x) = SAT(f(x))$$

Such a reduction function can be seen as a compiler: It takes source code written in some programming language and transforms in into an equivalent program in another programming language, typically a machine language, which has the some semantic behaviour. Since SAT is NP-complete, such a reduction exists for any possible problem in NP, including the problem of checking whether e.g. a bitcoin transaction is valid given an appropriate block hash. There is a reduction function that translates a transaction into a boolean formula, such that the formula is satisfiable if and only if the transaction is valid.

2.5 Reduction Example

In order to see such a reduction, let us consider the problem of evaluating polynomials. First, let us define a polynomial (similar to a boolean formula) as an expression consisting of integer constants, variables, addition, subtraction, multiplication and (correctly balanced) parentheses. Now the problem we want to consider is

> PolyZero(f) := 1 if f is a polynomial which has a zero where its variables are taken from the set $\{0, 1\}$

We will now construct a reduction from SAT to PolyZero and thus show that PolyZero is also NP-complete (checking that it lies in NP is left as an exercise).

It suffices to define the reduction function r on the structural elements of a boolean formula. The idea is that for any boolean formula f, the value $r(f)$ is a polynomial with the same number of variables and $f(a_1, .., a_k)$ is true if and only if $r(f)(a_1, .., a_k)$ is zero, where true corresponds to 1 and false corresponds to 0, and $r(f)$ only assumes the value 0 or 1 on variables from $\{0, 1\}$:

- $r(x_i) := (1 - x_i)$
- $r(\neg f) := (1 - r(f))$
- $r((f \wedge g)) := (1 - (1 - r(f))(1 - r(g)))$
- $r((f \vee g)) := r(f)r(g)$

One might have assumed that $r((f \wedge g))$ would be defined as $r(f) + r(g)$, but that will take the value of the polynomial out of the $\{0, 1\}$ set.

Using r, the formula $((x \wedge y) \vee \neg x)$ is translated to $(1 - (1 - (1 - x))(1 - (1 - y))(1 - (1 - x))$.

Note that each of the replacement rules for r satisfies the goal stated above and thus r correctly performs the reduction:

$$\text{SAT}(f) = \text{PolyZero}(r(f)) \text{ or } f \text{ is satisfiable if and only if } r(f) \text{ has a zero in } \{0, 1\}$$

2.6 Witness Preservation

From this example, you can see that the reduction function only defines how to translate the input, but when you look at it more closely (or read the proof that it performs a valid reduction), you also see a way to transform a valid witness together with the input. In our example, we only defined how to translate the formula to a polynomial, but with the proof we explained how to transform the witness, the satisfying assignment. This simultaneous transformation of the witness is not required for a transaction, but it is usually also done. This is quite important for zkSNARKs, because the the only task for the prover is to convince the verifier that such a witness exists, without revealing information about the witness.

3 Quadratic Span Programs

In the previous section, we saw how computational problems inside NP can be reduced to each other and especially that there are NP-complete problems that are basically only reformulations of all other problems in NP - including transaction validation problems. This makes it easy for us to find a generic zkSNARK for all problems in NP: We just choose a suitable NP-complete problem.

So if we want to show how to validate transactions with zkSNARKs, it is sufficient to show how to do it for a certain problem that is NP-complete and perhaps much easier to work with theoretically.

This and the following section is based on the paper "Quadratic Span Programs and Succinct NIZKs without PCPs" by Gennaro, Gentry, Parno and Raykova (the technical report at `https://eprint.iacr.org/2012/215.pdf` has much more information than the journal paper), where the authors found that the problem called Quadratic Span Programs (QSP) is particularly well suited for zkSNARKs. A Quadratic Span Program consists of a set of polynomials and the task is to find a linear combination of those that is a multiple of another given polynomial. Furthermore, the individual bits of the input string restrict the polynomials you are allowed to use. In detail (the general QSPs are a bit more relaxed, but we already define the *strong* version because that will be used later):

A QSP over a field F for inputs of length n consists of

- polynomials $v_0, \ldots, v_m$, $w_0, \ldots, w_m$ over this field F,
- a polynomial t over F (the target polynomial),
- an injective function $f\colon \{(i,j) \mid 1 \leq i \leq n, j \in \{0,1\}\} \to \{1,\ldots,m\}$

The task here is roughly, to multiply the polynomials by factors and add them so that the sum (which is called a *linear combination*) is a multiple of t. For each binary input string u, the function f restricts the polynomials that can be used, or more specific, their factors in the linear combinations. For formally:

An input u is *accepted* (verified) by the QSP if and only if there are tuples $a = (a_1, \ldots, a_m)$, $b = (b_1, \ldots, b_m)$ from the field F such that

- $a_k, b_k = 1$ if $k = f(i, u[i])$ for some i, ($u[i]$ is the ith bit of u)
- $a_k, b_k = 0$ if $k = f(i, 1 - u[i])$ for some i and
- the target polynomial t divides $v_a w_b$ where $v_a = v_0 + a_1 v_1 + \cdots + a_m v_m$ and $w_b = w_0 + b_1 w_1 + \cdots + b_m w_m$.

Note that there is still some freedom in choosing the tuples a and b if $2n$ is smaller than m. This means QSP only makes sense for inputs up to a certain size – this problem is removed by using non-uniform complexity, a topic we will not dive into now, let us just note that it works well for cryptography where inputs are generally small.

As an analogy to satisfiability of boolean formulas, you can see the factors $a_1, \ldots, a_m$, $b_1, \ldots, b_m$ as the assignments to the variables, or in general, the NP witness. To see that QSP lies in NP, note that all the verifier has to do (once she knows the factors) is checking that the polynomial t divides $v_a w_b$, which is a polynomial-time problem.

We will not talk about the reduction from generic computations or circuits to QSP here, as it does not contribute to the understanding of the general concept, so you have to believe me that QSP is

NP-complete (or rather complete for some non-uniform analogue like NP/poly). In practice, the reduction is the actual "engineering" part – it has to be done in a clever way such that the resulting QSP will be as small as possible and also has some other nice features.

One thing about QSPs that we can already see is how to verify them much more efficiently: The verification task consists of checking whether one polynomial divides another polynomial. This can be facilitated by the prover in providing another polynomial h such that $th = v_a w_b$ which turns the task into checking a polynomial identity or put differently, into checking that $th - v_a w_b = 0$, i.e. checking that a certain polynomial is the zero polynomial. This looks rather easy, but the polynomials we will use later are quite large (the degree is roughly 100 times the number of gates in the original circuit) so that multiplying two polynomials is not an easy task.

So instead of actually computing v_a, w_b and their product, the verifier chooses a secret random point s (this point is part of the "toxic waste" of zCash), computes the numbers $t(s)$, $v_k(s)$ and $w_k(s)$ for all k and from them, $v_a(s)$ and $w_b(s)$ and only checks that $t(s)h(s) = v_a(s)w_b(s)$. So a bunch of polynomial additions, multiplications with a scalar and a polynomial product is simplified to field multiplications and additions.

Checking a polynomial identity only at a single point instead of at all points of course reduces the security, but the only way the prover can cheat in case $th - v_a w_b$ is not the zero polynomial is if she manages to hit a zero of that polynomial, but since she does not know s and the number of zeros is tiny (the degree of the polynomials) when compared to the possibilities for s (the number of field elements), this is very safe in practice.

4 The zkSNARK in Detail

We now describe the zkSNARK for QSP in detail. It starts with a setup phase that has to be performed for every single QSP. In zCash, the circuit (the transaction verifier) is fixed, and thus the polynomials for the QSP are fixed which allows the setup to be performed only once and re-used for all transactions, which only vary the input u. For the setup, which generates the *common reference string* (CRS), the verifier chooses a random and secret field element s and encrypts the values of the polynomials at that point. The verifier uses some specific encryption E and publishes $E(v_k(s))$ and $E(w_k(s))$ in the CRS. The CRS also contains several other values which makes the verification more efficient and also adds the zero-knowledge property. The encryption E used there has a certain homomorphic property, which allows the prover to compute $E(v(s))$ without actually knowing $v_k(s)$.

4.1 How to Evaluate a Polynomial Succinctly and with Zero-Knowledge

Let us first look at a simpler case, namely just the encrypted evaluation of a polynomial at a secret point, and not the full QSP problem.

For this, we fix a group (an elliptic curve is usually chosen here) and a generator g. Remember that a group element is called *generator* if there is a number n (the *group order*) such that the

list g^0, g^1, g^2, …, g^{n-1} contains all elements in the group exactly once. The encryption is simply $E(x) := g^x$. Now the verifier chooses a secret field element s and publishes (as part of the CRS)

$E(s^0)$, $E(s^1)$, …, $E(s^d)$ – d is the maximum degree of all polynomials.

After that, s can be (and has to be) forgotten. This is exactly what zCash calls toxic waste, because if someone can recover this and the other secret values chosen later, they can arbitrarily spoof proofs by finding zeros in the polynomials.

Using these values, the prover can compute $E(f(s))$ for arbitrary polynomials f without knowing s: Assume our polynomial is $f(x) = 4x^2 + 2x + 4$ and we want to compute $E(f(s))$, then we get $E(f(s)) = E(4s^2 + 2s + 4) = g^{4s^2+2s+4} = E(s^2)^4 E(s^1)^2 E(s^0)^4$, which can be computed from the published CRS without knowing s.

The only problem here is that, because s was destroyed, the verifier cannot check that the prover evaluated the polynomial correctly. For that, we also choose another secret field element, α, and publish the following "shifted" values:

$E(\alpha s^0)$, $E(\alpha s^1)$, …, $E(\alpha s^d)$

As with s, the value α is also destroyed after the setup phase and neither known to the prover nor the verifier. Using these encrypted values, the prover can similarly compute $E(\alpha f(s))$, in our example this is $E(4\alpha s^2 + 2\alpha s + 4\alpha) = E(\alpha s^2)^4 E(\alpha s^1)^2 E(\alpha s^0)^4$. So the prover publishes $A := E(f(s))$ and $B := E(\alpha f(s)))$ and the verifier has to check that these values match. She does this by using another main ingredient: A so-called *pairing function* e. The elliptic curve and the pairing function have to be chosen together, so that the following property holds for all x, y:

$$e(g^x, g^y) = e(g, g)^{xy}$$

Using this pairing function, the verifier checks that $e(A, g^\alpha) = e(B, g)$ – note that g^α is known to the verifier because it is part of the CRS as $E(\alpha s^0)$. In order to see that this check is valid if the prover does not cheat, let us look at the following equalities:

$$\begin{aligned} e(A, g^\alpha) &= e(g^{f(s)}, g^\alpha) = e(g, g)^{\alpha f(s)} \\ e(B, g) &= e(g^{\alpha f(s)}, g) = e(g, g)^{\alpha f(s)} \end{aligned}$$

The more important part, though, is the question whether the prover can somehow come up with values A, B that fulfill the check $e(A, g^\alpha) = e(B, g)$ but are not $E(f(s))$ and $E(\alpha f(s)))$, respectively. The answer to this question is "we hope not". Seriously, this is called the "d-power knowledge of exponent assumption" and it is unknown whether a cheating prover can do such a thing or not. This assumption is an extension of similar assumptions that are made for proving the security of other public-key encryption schemes and which are similarly unknown to be true or not.

Actually, the above protocol does not really allow the verifier to check that the prover evaluated the polynomial $f(x) = 4x^2 + 2x + 4$, the verifier can only check that the prover evaluated *some* polynomial at the point s. The zkSNARK for QSP will contain another value that allows the verifier to check that the prover did indeed evaluate the correct polynomial.

What this example does show is that the verifier does not need to evaluate the full polynomial to confirm this, it suffices to evaluate the pairing function. In the next step, we will add the zero-knowledge part so that the verifier cannot reconstruct anything about $f(s)$, not even $E(f(s))$ – the encrypted value.

For that, the prover picks a random δ and instead of $A := E(f(s))$ and $B := E(\alpha f(s)))$, she sends over $A' := E(\delta + f(s))$ and $B := E(\alpha(\delta + f(s))))$. If we assume that the encryption cannot be broken, the zero-knowledge property is quite obvious. We now have to check two things: 1. the prover can actually compute these values and 2. the check by the verifier is still true.

For 1., note that $A' = E(\delta + f(s)) = g^{\delta+f(s)} = g^{\delta}g^{f(s)} = E(\delta)E(f(s)) = E(\delta)A$ and similarly, $B' = E(\alpha(\delta + f(s)))) = E(\alpha\delta + \alpha f(s))) = g^{\alpha\delta+\alpha f(s)} = g^{\alpha\delta}g^{\alpha f(s)} = E(\alpha)^{\delta}E(\alpha f(s)) = E(\alpha)^{\delta}B$.

For 2., note that the only thing the verifier checks is that the values A and B she receives satisfy the equation $A = E(a)$ und $B = E(\alpha a)$ for some value a, which is obviously the case for $a = \delta + f(s)$ as it is the case for $a = f(s)$.

Ok, so we now know a bit about how the prover can compute the encrypted value of a polynomial at an encrypted secret point without the verifier learning anything about that value. Let us now apply that to the QSP problem.

4.2 A SNARK for the QSP Problem

Remember that in the QSP we are given polynomials $v_0, \ldots, v_m$, $w_0, \ldots, w_m$, a target polynomial t (of degree at most d) and a binary input string u. The prover finds $a_1, \ldots, a_m$, $b_1, \ldots, b_m$ (that are somewhat restricted depending on u) and a polynomial h such that

$$th = (v_0 + a_1v_1 + \cdots + a_mv_m)(w_0 + b_1w_1 + \cdots + b_mw_m).$$

In the previous section, we already explained how the common reference string (CRS) is set up. We choose secret numbers s and α and publish

$$E(s^0), E(s^1), \ldots, E(s^d) \quad \text{and} \quad E(\alpha s^0), E(\alpha s^1), \ldots, E(\alpha s^d)$$

Because we do not have a single polynomial, but sets of polynomials that are fixed for the problem, we also publish the evaluated polynomials right away:

- $E(t(s))$, $E(\alpha t(s))$,
- $E(v_0(s)), \ldots, E(v_m(s))$, $E(\alpha v_0(s)), \ldots, E(\alpha v_m(s))$,

- $E(w_0(s))$, …, $E(w_m(s))$, $E(\alpha w_0(s))$, …, $E(\alpha w_m(s))$,

and we need further secret numbers β_v, β_w, γ (they will be used to verify that those polynomials were evaluated and not some arbitrary polynomials) and publish

- $E(\gamma)$, $E(\beta_v\gamma)$, $E(\beta_w\gamma)$,
- $E(\beta_v v_1(s))$, …, $E(\beta_v v_m(s))$
- $E(\beta_w w_1(s))$, …, $E(\beta_w w_m(s))$
- $E(\beta_v t(s))$, $E(\beta_w t(s))$

This is the full common reference string. In practical implementations, some elements of the CRS are not needed, but that would complicate the presentation.

Now what does the prover do? She uses the reduction explained above to find the polynomial h and the values $a_1, \ldots, a_m, b_1, \ldots, b_m$. Here it is important to use a witness-preserving reduction (see above) because only then, the values $a_1, \ldots, a_m, b_1, \ldots, b_m$ can be computed together with the reduction and would be very hard to find otherwise. In order to describe what the prover sends to the verifier as proof, we have to go back to the definition of the QSP.

There was an injective function $f\colon \{(i,j) \mid 1 \le i \le n, j \in \{0,1\}\} \to \{1,\ldots,m\}$ which restricts the values of $a_1, \ldots, a_m, b_1, \ldots, b_m$. Since m is relatively large, there are numbers which do not appear in the output of f for any input. These indices are not restricted, so let us call them I_{free} and define $v_{\text{free}}(x) = \sum_k a_k v_k(x)$ where the k ranges over all indices in I_{free}. For $w(x) = b_1 w_1(x) + \cdots + b_m w_m(x)$, the proof now consists of

- $V_{\text{free}} := E(v_{\text{free}}(s)), \quad W := E(w(s)), \quad H := E(h(s)),$
- $V'_{\text{free}} := E(\alpha v_{\text{free}}(s)), \quad W' := E(\alpha w(s)), \quad H' := E(\alpha h(s)),$
- $Y := E(\beta_v v_{\text{free}}(s) + \beta_w w(s)))$

where the last part is used to check that the correct polynomials were used (this is the part we did not cover yet in the other example). Note that all these encrypted values can be generated by the prover knowing only the CRS.

The task of the verifier is now the following:

Since the values of a_k, where k is not a "free" index can be computed directly from the input u (which is also known to the verifier, this is what is to be verified), the verifier can compute the missing part of the full sum for v:

$E(v_{in}(s)) = E(\sum_k a_k v_k(s))$ where the k ranges over all indices *not* in I_{free}.

With that, the verifier now confirms the following equalities using the pairing function e (don't be scared):

1. $e(V'_{\text{free}}, g) = e(V_{\text{free}}, g^\alpha), \quad e(W', E(1)) = e(W, E(\alpha)), \quad e(H', E(1)) = e(H, E(\alpha))$
2. $e(E(\gamma), Y) = e(E(\beta_v \gamma), V_{\text{free}}) e(E(\beta_w \gamma), W)$
3. $e\Big(E(v_0(s))E(v_{in}(s))V_{\text{free}},\ E(w_0(s))W\Big) = e(H, E(t(s)))$

To grasp the general concept here, you have to understand that the pairing function allows us to do some limited computation on encrypted values: We can do arbitrary additions but just a single multiplication. The addition comes from the fact that the encryption itself is already additively homomorphic and the single multiplication is realized by the two arguments the pairing function has. So $e(W', E(1)) = e(W, E(\alpha))$ basically multiplies W' by 1 in the encrypted space and compares that to W multiplied by α in the encrypted space. If you look up the value W and W' are supposed to have – $E(w(s))$ and $E(\alpha w(s))$ – this checks out if the prover supplied a correct proof.

If you remember from the section about evaluating polynomials at secret points, these three first checks basically verify that the prover did evaluate some polynomial built up from the parts in the CRS. The second item is used to verify that the prover used the correct polynomials v and w and not just some arbitrary ones. The idea behind is that the prover has no way to compute the encrypted combination $E(\beta_v v_{\text{free}}(s) + \beta_w w(s)))$ by some other way than from the exact values of $E(v_{\text{free}}(s))$ and $E(w(s))$. The reason is that the values β_v are not part of the CRS in isolation, but only in combination with the values $v_k(s)$ and β_w is only known in combination with the polynomials $w_k(s)$. The only way to "mix" them is via the equally encrypted γ.

Assuming the prover provided a correct proof, let us check that the equality works out. The left and right hand sides are, respectively

$$\begin{aligned}
e(E(\gamma), Y) &= e(E(\gamma), E(\beta_v v_{\text{free}}(s) + \beta_w w(s))) \\
&= e(g, g)^{\gamma(\beta_v v_{\text{free}}(s) + \beta_w w(s))} \\
e(E(\beta_v \gamma), V_{\text{free}}) e(E(\beta_w \gamma), W) &= e(E(\beta_v \gamma), E(v_{\text{free}}(s))) e(E(\beta_w \gamma), E(w(s))) \\
&= e(g, g)^{(\beta_v \gamma) v_{\text{free}}(s)} e(g, g)^{(\beta_w \gamma) w(s)} \\
&= e(g, g)^{\gamma(\beta_v v_{\text{free}}(s) + \beta_w w(s))}
\end{aligned}$$

The third item essentially checks that $(v_0(s) + a_1 v_1(s) + \cdots + a_m v_m(s))(w_0(s) + b_1 w_1(s) + \cdots + b_m w_m(s)) = h(s)t(s)$, the main condition for the QSP problem. Note that multiplication on the encrypted values translates to addition on the unencrypted values because $E(x)E(y) = g^x g^y = g^{x+y} = E(x + y)$.

4.3 Adding Zero-Knowledge

As I said in the beginning, the remarkable feature about zkSNARKS is rather the succinctness than the zero-knowledge part. We will see now how to add zero-knowledge and the next section will be touch a bit more on the succinctness.

The idea is that the prover "shifts" some values by a random secret amount and balances the shift on the other side of the equation. The prover chooses random δ_{free}, δ_w and performs the following replacements in the proof

- $v_{\text{free}}(s)$ is replaced by $v_{\text{free}}(s) + \delta_{\text{free}} t(s)$
- $w(s)$ is replaced by $w(s) + \delta_w t(s)$.

By these replacements, the values V_{free} and W, which contain an encoding of the witness factors, basically become indistinguishable from randomness and thus it is impossible to extract the witness. Most of the equality checks are "immune" to the modifications, the only value we still have to correct is H or $h(s)$. We have to ensure that

$$(v_0(s) + a_1 v_1(s) + \cdots + a_m v_m(s))(w_0(s) + b_1 w_1(s) + \cdots + b_m w_m(s)) = h(s)t(s)$$

or in other words

$$(v_0(s) + v_{in}(s) + v_{\text{free}}(s))(w_0(s) + w(s)) = h(s)t(s).$$

still holds. With the modifications, we get

$$(v_0(s) + v_{in}(s) + v_{\text{free}}(s) + \delta_{\text{free}} t(s))(w_0(s) + w(s) + \delta_w t(s))$$

and by expanding the product, we see that replacing $h(s)$ by

$$h(s) + \delta_{\text{free}}(w_0(s) + w(s)) + \delta_w(v_0(s) + v_{in}(s) + v_{\text{free}}(s)) + (\delta_{\text{free}}\delta_w)t(s)$$

will do the trick.

5 Tradeoff between Input and Witness Size

As you have seen in the preceding sections, the proof consists only of 7 elements of a group (typically an elliptic curve). Furthermore, the work the verifier has to do is checking some equalities involving pairing functions and computing $E(v_{in}(s))$, a task that is linear in the input size. Remarkably, neither the size of the witness string nor the computational effort required to verify the QSP (without SNARKs) play any role in verification. This means that SNARK-verifying extremely complex problems and very simple problems all take the same effort. The main reason for that is because we only check the polynomial identity for a single point, and not the full polynomial. Polynomials can get more and more complex, but a point is always a point. The only parameters that influence the verification effort is the level of security (i.e. the size of the group) and the maximum size for the inputs.

It is possible to reduce the second parameter, the input size, by shifting some of it into the witness:

Instead of verifying the function $f(u, w)$, where u is the input and w is the witness, we take a hash function h and verify

$$f'(H, (u, w)) := f(u, w) \wedge h(u) = H.$$

This means we replace the input u by a hash of the input $h(u)$ (which is supposed to be much shorter) and verify that there is some value x that hashes to $H(u)$ (and thus is very likely equal to u) in addition to checking $f(x, w)$. This basically moves the original input u into the witness string and thus increases the witness size but decreases the input size to a constant.

This is remarkable, because it allows us to verify arbitrarily complex statements in constant time.

6 How is this Relevant to Ethereum

Since verifying arbitrary computations is at the core of the Ethereum blockchain, zkSNARKs are of course very relevant to Ethereum. With zkSNARKs, it becomes possible to not only perform secret arbitrary computations that are verifiable by anyone, but also to do this efficiently.

Although Ethereum uses a Turing-complete virtual machine, it is currently not yet possible to implement a zkSNARK verifier in Ethereum. The verifier tasks might seem simple conceptually, but a pairing function is actually very hard to compute and thus it would use more gas than is currently available in a single block. Elliptic curve multiplication is already relatively complex and pairings take that to another level.

Existing zkSNARK systems like zCash use the same problem / circuit / computation for every task. In the case of zCash, it is the transaction verifier. On Ethereum, zkSNARKs would not be limited to a single computational problem, but instead, everyone could set up a zkSNARK system for their specialized computational problem without having to launch a new blockchain. Every new zkSNARK system that is added to Ethereum requires a new secret trusted setup phase (some parts can be re-used, but not all), i.e. a new CRS has to be generated. It is also possible to do things like adding a zkSNARK system for a "generic virtual machine". This would not require a new setup for a new use-case in much the same way as you do not need to bootstrap a new blockchain for a new smart contract on Ethereum.

6.1 Getting zkSNARKs to Ethereum

There are multiple ways to enable zkSNARKs for Ethereum. All of them reduce the actual costs for the pairing functions and elliptic curve operations (the other required operations are already cheap enough) and thus allows also the gas costs to be reduced for these operations.

1. improve the (guaranteed) performance of the EVM
2. improve the performance of the EVM only for certain pairing functions and elliptic curve multiplications

The first option is of course the one that pays off better in the long run, but is harder to achieve. We are currently working on adding features and restrictions to the EVM which would allow better just-in-time compilation and also interpretation without too many required changes in the existing implementations. The other possibility is to swap out the EVM completely and use something like eWASM.

The second option can be realized by forcing all Ethereum clients to implement a certain pairing function and multiplication on a certain elliptic curve as a so-called precompiled contract. The benefit is that this is probably much easier and faster to achieve. On the other hand, the drawback is that we are fixed on a certain pairing function and a certain elliptic curve. Any new client for Ethereum would have to re-implement these precompiled contracts. Furthermore, if there are advancements and someone finds better zkSNARKs, better pairing functions or better elliptic curves, or if a flaw is found in the elliptic curve, pairing function or zkSNARK, we would have to add new precompiled contracts.