# An Introduction to the Use of zk-SNARKs in Blockchains


**Alexandre Miranda Pinto**

**Abstract** The advent of blockchain brings a wide horizon of opportunities to the world. The first such example, Bitcoin, strongly advocates a principle of total openness, which is reflected in the fact that all transactions are public and the history of each account can easily be reconstructed. Although an account cannot immediately be linked to a real-world identity, this does not grant strong guarantees of anonymity, and such a feature of Bitcoin and similar blockchains prevents it from reaching wide acceptance for financial use-cases, where users often desire strong confidentiality of their balances and financial history. As a consequence, there has recently been a growing interest in privacy-enhancing technologies that ensure public permissionless blockchains can keep the details of transactions private according to particular use cases. One of the most promising technologies in this area is Zero-Knowledge Proofs, and in particular zk-SNARKs, due to their very short proofs and verification times. This makes them well suited to be used as transaction data, hiding all the private details at the same time they guarantee the integrity and accuracy of the transaction, and to be verified on-chain by a smart contract. This paper is an introductory presentation of this topic, what advantages zk-SNARKs bring to the blockchain ecosystem and how they can be tailored to specific applications.

**Keywords** Zero-knowledge proofs · zk-SNARKs · Privacy in blockchains

## 1 Introduction and Paper Layout

Since the introduction of Bitcoin in 2009 [28], Distributed-Ledger Technology (DLT), more often known as blockchain, has steadily grown and been recognized as a new tool with potentially revolutionary use-cases. Some of them will be mainly technical, and will harness the strong guarantees of distributed consensus and the maintenance of a single source of truth shared by many independent (collaborative,

A. M. Pinto (✉)
Artos Systems, 160 Fleet Street, London EC4A 2DQ, UK
e-mail: alex.miranda.pinto@gmail.com



P. Pardalos et al. (eds.), *Mathematical Research for Blockchain Economy*,
Springer Proceedings in Business and Economics,
https://doi.org/10.1007/978-3-030-37110-4_16

or possibly competing) parties; but others may be rooted on innovative ways of thinking social and economical relationships, such as the intrinsic value of money, whom we must trust to manage its creation and whether users should be allowed to transfer it outside any controls of central authorities.

One of the radical innovations of Bitcoin was the total openness of its ledger. There may be different reasons for this, for example a desire to remove all trust necessary in central authorities, who may be viewed as potentially corruptible entities bent on limiting individual freedoms; or instead a belief in total transparency as a virtue of advanced societies, where those who have nothing to hide should not fear scrutiny; or it may even have been a simple decision to solve a hard technical problem, Byzantine Agreement (see [25]), in a network with millions of participants.

Whatever the case, such absolute transparency is not always desired, especially if Bitcoin is intended as a replacement for fiat currencies as a means of exchange. Typical users value their privacy in this domain, and would rather prefer to keep their transactions history private. This is why it is commonly advised to use each Bitcoin address only once,[1] to avoid linking transactions to one same identity. A better way is to use privacy-enhanced blockchains, either where confidentiality has been designed in by default (for example ZCash,[2] Monero,[3] Grin,[4] Beam,[5] Dash[6]) or when it has been added a posteriori by some other mechanism. See for example Zether [11], Mimblewimble [24, 30] or Coinjoin [17].

Zero-Knowledge Proofs (ZKP) are among the most popular technologies, and have turned from a quite specialized cryptographic technique into an everyday term for blockchain developers. This paper introduces the notion of ZKP for audiences without knowledge of cryptography (Sect. 2). It explores the notion of zk-SNARKs and compares them to recent alternatives, framing them in the context of blockchain (Sect. 4). In Sect. 3, I compare a few variants, with the focus on zk-SNARKs. The rest of the paper goes into more technical details, explaining how zk-SNARKs achieve their flexibility (Sect. 5) and referencing tools currently available to implement them (Sect. 6).

## 2 Zero-Knowledge Proofs

Zero-Knowledge Proofs were introduced in 1985 by Goldwasser, Micali and Rackoff [21]. These are a cryptographic technique in which two parties, the Prover and the Verifier, participate in a protocol. Following normal naming in the literature, I will call the Prover Peggy, and the Verifier Victor. Peggy and Victor both know a predicate

[1] See for example the recommendations in https://bitcoin.org/en/protect-your-privacy.

[2] https://z.cash.

[3] https://www.getmonero.org/.

[4] https://grin-tech.org/.

[5] https://www.beam.mw/.

[6] https://www.dash.org/.

$S$ and a public instance $x$. The aim of the protocol is for Peggy to prove to Victor that $S(x)$ is a true statement without revealing anything else about why that is true. An example might be a statement about a specific graph $G$: "The graph $G$ is 3-colourable". If this is true about $G$, Peggy can prove that is so, without revealing why it is so. That means Victor will not be able to learn any valid 3-colouring of $G$ from Peggy's proof.

A stronger notion of Zero-Knowledge Proof is the *Zero-Knowledge Proof of Knowledge (ZK-POK)*. With such a proof, Peggy can convince Victor not only of the truth of $S(x)$ but also that Peggy knows a witness that demonstrates it. Typically, this means Peggy knows why or how the statement can be true, and this knowledge is represented by some private witness $w$ known to Peggy but not to Victor. In this case, both Peggy and Victor know another predicate, a relation $R$ such that $S(x)$ is true if and only if $R(x, w)$ is true as well. Continuing the example above, in a ZK-POK Peggy convinces Victor not only that $S(x)$ is true, but also that she knows $w$ such that $R(x, w) = 1$.

## 2.1 *Zero-Knowledge and* NP

The class of predicates that can be proven in Zero-Knowledge is well defined: it coincides with the class **NP**, under the mild assumption that encryption functions exist. This class is made up of exactly those languages which can be *verified* in polynomial time, that is:

**Definition 1** Language $L$ is in **NP** if there is a relationship $R_L(x, w)$ that runs in time polynomial on the size of its input $x$ and can verify membership in the language: if $x$ belongs in the language, then there is a witness $w$ related to $x$. If $x$ is not in the language, then there is no witness that can be related to it. Formally,

$$\forall x \in L,\ \exists\, w\ s.t.\ R_L(x, w) = 1$$
$$\forall x \notin L,\ \forall\, w\ R_L(x, w) = 0.$$

This result was proved in [20].

An example of such a language is the non-primality of integers. Define the language *NONPRIMES* to be composed of all integers which are not prime. The relationship for this language is

$$R_{NONPRIMES}(x, s) = x \bmod s == 0 \wedge s \neq x \wedge s \neq 1,$$

and for each member $x$, we can provide a witness by showing a non-trivial factor of $x$.

## 2.2 Interactive and Non-interactive Proofs of Knowledge

The original Zero-Knowledge proofs were all examples of interactive proofs, where Peggy and Victor send messages to each other until Victor is satisfied. At each step, Victor sends a random query to Peggy that she can only answer successfully with some low probability in case the statement is false or she does not actually know a proof. If at any point Peggy is unable to successfully answer the query, then Victor knows she is cheating and rejects the proof. By repeating this questioning enough times, Victor reduces the probability that Peggy succeeds. For example, if Peggy's chance of success to any query is $\frac{1}{2}$, then the probability of succeeding ten times in a row is only $\left(\frac{1}{2}\right)^{10} = \frac{1}{2^{10}}$. Therefore, by issuing just 10 questions, Victor can have a very good assurance that Peggy is not lying.

There is a general approach to make a proof of knowledge non-interactive, called the Fiat-Shamir heuristic [16], as long as Victor's randomness is public to all parties. This is called a heuristic because it provides security in the random-oracle model only, that is, assuming that the hash function behaves as a good random function. By using this technique, Peggy can simulate the random queries Victor would pose her by replacing them with a hash of the previous message in the protocol. The first message is usually sent by Peggy, committing to a blinded version of her private input in order to guarantee that Victor does not learn anything about it (and maintain the Zero-Knowledge property), but also ensuring she cannot fool Victor by using a different value. Using this heuristic, Peggy can compute all of "Victor's messages", which she then can send in a single transcript of the whole session. Victor's work then reduces to verifying this single transcript and output whether he thinks that is a correct proof.

A different way to make a proof non-interactive is the use of a Common Reference String (CRS), proposed in [10] and expanded in [9]. This model differs from the Fiat-Shamir heuristic in that it uses true randomness, and not a simulation thereof. This is created in a setup phase and given to all participants. Some measure of trust is needed in this phase, as all participants must be assured that the string has been honestly generated and is correctly shared by all parties.

A special instance of non-interactive proofs of knowledge is known as zk-SNARKs. In fact, they're not proofs, but rather arguments of knowledge. A proof of knowledge guarantees that a malicious prover cannot prove any false statement. On the other hand, an argument of knowledge only gives such guarantees with respect to *computationally bounded provers*. The defining properties of zk-SNARKs is that they are succinct, that is, the proofs are very small (in fact, of constant size, no matter the complexity of the proof statement) and the verification is very fast. But they are constructed in the *common reference string model*, which has the drawback of requiring a trusted setup phase. zk-SNARKs were first defined in [8], initiating a very fruitful line of research. Practical zk-SNARKs based on Arithmetic Circuits and Quadratic Arithmetic Programs were introduced in [19].

Other more recent variants of Zero-knowledge non-interactive constructions are bulletproofs [12] and zk-STARKs [4]. Bulletproofs are especially suited to a specific

| | TIME | | SIZE | | SECURITY | | |
|---|---|---|---|---|---|---|---|
| | Proof | Verification | Key | Proof | Post- Quantum Resistant | Untrusted Setup | Assumptions |
| zk-SNARKs | n log n | O(1) | n | O(1) | No | No | Strong Number Theoretical |
| zk-STARKs | n polylog n | polylog n | 0 | log^2 n | Yes | Yes | Weak: CHRF |
| Bulletproofs | n log n | n log n | n | log n | No | Yes | Weak Number Theoretical |

**Fig. 1** Comparison between zk-SNARKs, zk-STARKs and bulletproofs

kind of proofs, demonstrating that the secret value falls within a certain range, but they can also be used for general **NP** circuits. zk-STARKs, on the other hand, are generic constructions and have a number of advantages, but the technology is still not mature enough to be usable in practice.

## 3 Why zk-SNARKs?

In practice, Bulletproofs, zk-SNARKs, and zk-STARKs are all interesting technologies to use. In this section, I compare them according to some relevant parameters and discuss why at the moment zk-SNARKs seem to be the most popular choice for use with blockchain technologies. These can be divided in two groups: performance and security. It will be noted that zk-STARKs are the preferred choice in terms of security, where they beat or equal the other two options. On the contrary, zk-SNARKs excel in performance when compared with the others, and zk-STARKs in particular are still eminently not practical due to their large proof sizes. As a consequence, zk-SNARKs are the preferred general-use choice at the moment, but can be overtaken by zk-STARKs if ongoing research can make them more effective. Bulletproofs hold the middle ground. They are more secure than zk-SNARKs and notably don't require a trusted setup. At the same time, they can also be performant for simple circuits, and gain from not requiring pairing technology. Still their proofs and verification size grow with the complexity of the proof statement. A summary of these aspects follows below (Fig. 1).

### *3.1 Performance*

*Proof Size* The big advantage of zk-SNARKs is the proof size, that is always a constant independently of the circuit's complexity. On the other hand, the size of a bulletproof grows logarithmically with the size of the circuit. For simple statements, this is short enough to be practical, but no technology can currently beat zk-SNARKs

in the general case. zk-STARKs are particularly bad, generating proofs in the order of tens or hundreds of kB while zk-SNARKs generate proofs in the hundreds of bytes.
*Verification Time* Verification time is directly related to the proof size and the number of public inputs. zk-SNARKs are again very efficient in this respect, with the time growing linearly with the number of public inputs. However, the most expensive operations are a constant number of elliptic-curve pairings, that may still dominate if the public inputs are few. Expect a proof to be verified in a few milliseconds. Bulletproofs in turn can be verified in time proportional to their proof size, which is fast for simple circuits. zk-STARKs also have fast verification, but still not as fast as zk-SNARKs.

The zk-STARK whitepaper claims zk-STARKs verify in a constant time, while zk-SNARKs's time would grow linearly. However, they include the setup time in this (which does not exists for zk-STARKs). They also show a comparison when the setup is not included, which shows zk-SNARKs about 10 times faster than zk-STARKs.

I believe they make an unfair comparison. This is because the SNARK setup is performed *only once per circuit*, whereas a single proof can be verified several times. These are clearly two separate processes, and their time should not be brought together as if the setup would be needed every time we make a verification, as that is plainly not the case. Therefore, the valid comparison for me shows zk-SNARKs are faster than zk-STARKs.
*Key Size* zk-STARKs and Bulletproofs get the upper hand here as neither require any keys, whereas zk-SNARKs do. In particular, proving keys can be extremely large for complex circuits (in the order of megabytes of even gigabytes, depending on the circuit complexity), since they essentially encode the whole computation.

## *3.2 Security*

*Trusted Setup* zk-SNARKs are constructed in the Common Reference String Model. This means they require a setup phase which creates a Common Reference String (CRS) made up of a proving key and a verification key that can be then distributed to appropriate users. This setup is highly sensitive. As part of the key generation, some randomness is created that must be destroyed at the end of the setup. Otherwise, an attacker who learns of this would be able to create false proofs. This is a difficult problem when the CRS has to be generated for use in a large network of untrusted participants.

In comparison, Bulletproofs and zk-STARKs do not require such a setup, which gives them an important advantage.
*Hardness Assumptions* Proofs of security usually depend on some hardness assumption: they reduce any possible attack (within a certain model) to breaking a known problem that is considered to be very hard. The weaker an assumption is, the more likely it is thought to be true and the less things it requires, the stronger the proof of security is. zk-SNARKs require very strong assumptions of number-theoretical nature, namely the relatively recent and still insufficiently understood Knowledge-

of-Exponent Assumption [3]. Bulletproofs are also based on a number-theoretical assumption, the hardness of the discrete logarithm, but this is a standard assumption and much weaker than that needed for zk-SNARKs. zk-STARKs require the weakest assumptions of all three, merely the existence of a collision-resistant hash function.
*Post-Quantum Resistance* Both zk-SNARKs and Bulletproofs require number-theoretical assumptions of a nature that will be easily broken in case quantum-computers become practical. On the other hand, zk-STARKs' assumption is not number-theoretical and is currently not known to be broken by quantum-computers. Therefore, zk-STARKs are considered to be post-quantum resistant, whereas the other two types of construction are not.

### *3.3 Existing Snarks*

Although the literature in zk-SNARKs is quite extensive, only a few of them have been implemented in practical cryptographic tools. The most popular ones are [6, 22], due to their proofs of constant size and verification time linear only in the input. Both are based on the same QAP front-end. The former is an update of the original [29], and has been available for a longer time. Although an attack was found on its definition this year [18], this has been fixed and the scheme is considered secure again. The Groth scheme is currently the most efficient one available, with shorter proofs. There is another alternative, [23], that has a proof as short as [22], but gives stronger guarantees, since it is actually a *signature of knowledge* and not just a proof of knowledge. It is, however, less efficient in both proof generation and verification.

Other earlier constructions can be found in [2, 5, 15, 19, 29].

## 4 Use in Blockchains

zk-SNARKs are particularly well suited to work in blockchains for two main reasons. First, they are non-interactive, which means verification can happen independently from the prover. This allows several verifiers to check the proof without collaboration and at their convenience. Secondly, the proof is concise, which means it can be conveniently given to a smart-contract without incurring a heavy gas cost and making the verification fast as well. But the main reason they are interesting is because of the functionality they bring, which is crucial for blockchain use-cases: privacy and scalability.

The case for privacy has already been argued at the beginning of this paper: I believe it is crucial for high mass-adoption in use cases where users require privacy, be it for their financial or commercial data. zk-SNARKs provide the ability to hide all of these from the public. It is possible to make the blockchain store only summaries of masked versions of a state, and enforce the consistency of updates by Zero-

Knowledge proofs. As an example, consider the case of a transaction in the UTXO model: this will include a list of all input notes, which together represent the money spent; and a list of output notes, which represent the money sent to the recipient(s). In Bitcoin, these notes are represented in the clear, but a private alternative could send just the hashes of these notes. This would totally obfuscate the balances spent and received and would even make possible the creation and destruction of value. In order to enforce consistency, we can add a Zero-Knowledge proof that the sum of inputs and outputs is the same (or differs only by the network fees) without revealing them. This is the approach taken by ZCash [31].

Privacy in this way immediately gives rise to scalability improvements. Since a single transaction can now be summarized by a short state update and a short proof, and because this proof can be automatically verified, then we can also do a single proof that validates other proofs. Instead of submitting a zero-knowledge proof for each transaction, we can effectively bundle a group of them, providing proofs for each, and then add a single proof that checks all of them. Only this proof is submitted to the chain, together with an update of state that reflects all of the verified transactions. An example of a project that is exploring this approach is CODA (see [27]), which builds on the academic work of [7].

## 5 QAP-Based Snarks

QAP-based zk-SNARKs have become popular for practical use-cases because of their flexibility. Once the predicate statement is codified in terms of an arithmetic circuit, a zk-SNARK can automatically be built by appropriate tools that turn the circuit into a QAP and then use that to setup the system. The implementer's job is mostly focused on specifying the R1CS that defines the problem, which can be done more or less trivially once the circuit is defined (the non-trivial aspect is that there may be thousands of gates and wires in the circuit).

Compare, for example, with $\Sigma-$protocols, which can still be used for a large variety of proofs but where the designer has to be much more careful in deciding the content of each message. Some results show how we can mix and match basic protocols to prove more complex statements (see for example [1, 13, 14, 26]), but as far as I know there is not a simple compact way to encode an arbitrary **NP** statement into a single proof.

For that reason, in this section I focus on the progression from arithmetic circuits to QAPs, explaining how these effectively encode the whole computation and so make the proof convincing.

## *5.1 Arithmetic Circuits*

Circuits have long been used as a computation model. A circuit is composed of wires, which carry values, and gates, which perform an operation on their input values and return one or more outputs. In complexity theory, it is common to consider logical circuits, where the value of each wire is either 0 or 1, and gates perform logical operations like **AND**, **OR**, **NOT** and their variants. For zk-SNARKs, we use instead arithmetic circuits. The only difference to logical circuits is the operations computed by their gates and the values that each wire can represent.

The gates of a circuit must be organized in a directed acyclic graph, so that computation always flows in a single direction from inputs to outputs. Unlike a programming language, circuits do not have loops or functions where the computation can return multiple times to the same place: computations are flattened and laid out in a way that each gate is evaluated only once and the wires, once set, never change value. Analogously to simple electric circuits, where you can determine the voltage and current of every wire once you turn the power on, you can 'instantaneously' determine the value of all wires, including the output, of a logical circuit once you set the input values.

QAP-Based Snarks are based on pairings over elliptic curves, which are ultimately used to encode all the steps of the computation. The arithmetic circuit used for a zk-SNARK is tied to the specific finite field underlying the elliptic curve used, and so each wire can represent a single field element. The circuit is the verifier of a computation, and so returns a binary value. For a valid proof, the circuit should return 1 if the private input (the witness) provided by the prover matches the public input known to both parties.

Gates can perform modular addition and multiplication. Note that the modulus used is the order of the curve, and not that of the field. This is because of how the circuit is encoded to produce the zk-SNARK: the values of each wire are multiplied with curve points combined in several linear combinations. In other words, they are the scalars resulting from a series of multiplications in the elliptic curve's group. Therefore, they will never be larger than the curve's order.

Figure 2 demonstrates a simple arithmetic circuit, and how the computation proceeds from the input wires to produce values to the outputs. With this circuit, the prover demonstrates knowledge of two private values, $a$ and $b$, that satisfy a certain relationship with a public input, $n$, namely $a^2 + ab - b = n$.

Each wire is thus assigned a single value, and the list of all these assignments constitutes an instance of the circuit's computation, which corresponds to a single input. Conventionally, in an assignment public wires are listed first, followed by private inputs and then the internal wires corresponding to intermediate computations. Circuit outputs are always public. The whole is preceded by a constant 1 that is added to enable constant values. The example in the figure corresponds to the following assignment, with one public input, one (public) output, two private inputs and four internal wires:

$$[1, 9, 1, 2, 5, 4, 10, 14, 14]. \quad (1)$$

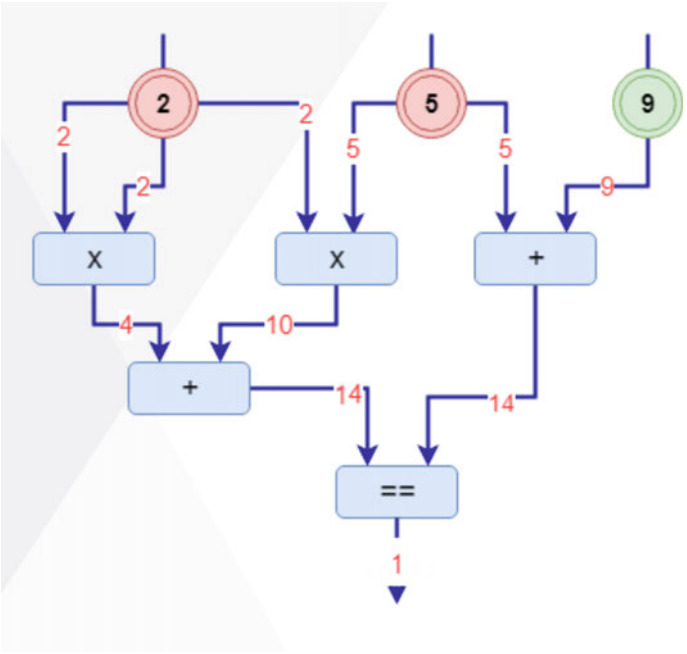


**Fig. 2** A simple arithmetic circuit proving knowledge of $a$, $b$ such that $a^2 + ab - b = n$

If we give a different input to the circuit, say $a = 4, b = 3, n = 10$, the result will be a failing computation, and the full assignment will be

$$[1, 10, 0, 4, 3, 16, 12, 28, 13].$$

## *5.2 Rank-1 Constraint Systems*

An assignment of values to all the wires in the circuit describes a single computation for a given set of public and private inputs. The next step in the construction of a zk-SNARK is to produce a set of constraints that assert this computation has been correctly performed, and that the assignment is internally consistent. In other words, the constraints check that each non-input wire follows correctly from the application of a gate operation to the input's gates. Therefore, we create a rank-1 constraint for each gate, and call Rank-1 Constraint System (R1CS) to the set of all constraints. Consequently, if a cheating prover does not know the correct private witness for the public input and tries to fool the verifier by showing an output that does not follow from its inputs, then at some point a gate's output must have been miscalculated and the corresponding constraint will not be valid.

Constraints are encoded as a simple multiplicative form:

$$a \times b = c,$$

where $a$, $b$ and $c$ are numbers resulting from evaluating linear combinations of wires ($A$, $B$ and $C$), as will be seen below. Given the circuit above, and a well-defined wire ordering, we can encode the first multiplication gate, which computes $a^2$, by

$$\begin{aligned} A &= [0, 0, 0, 1, 0, 0, 0, 0, 0] \text{ representing the first private input} \\ B &= [0, 0, 0, 1, 0, 0, 0, 0, 0] \text{ representing the same wire} \\ C &= [0, 0, 0, 0, 0, 1, 0, 0, 0] \text{ representing the first internal wire.} \end{aligned}$$

An addition gate must be represented by adding wires within the same linear combination. Consequently, the other input linear combination must simply represent 1. This is the representation of the first addition gate:

$$\begin{aligned} A &= [0, 0, 0, 0, 0, 1, 1, 0, 0] \text{ representing the addition of two internal wires} \quad (2) \\ B &= [1, 0, 0, 0, 0, 0, 0, 0, 0] \text{ representing the constant 1 wire} \\ C &= [0, 0, 0, 0, 0, 0, 0, 1, 0] \text{ representing the third internal wire.} \end{aligned}$$

The constraint system may include more constraints than just those implied by the gates. As an example, equality constraints can be added. An equality constraint on two wires can be written as a multiplication: one of the inputs is set to 1, and the other two encode the wires that are being compared.

More complex assertions can be composed in this form. For example, the constraint wire 3 can carry only a binary value, which can be described by the equation $w_3 \cdot (w_3 - 1) = 0$ or equivalently $w_3^2 = w_3$. Such a constraint would not ordinarily be represented in the circuit, since it does not have an impact in the computation, but should be added to the R1CS.

All of the matrices above simply encode an abstract verification. Their concrete meaning is given by mixing them with a wire-assignment corresponding to a computation. Let such assignment be a vector $\vec{s}$. Then, a constraint ($A$, $B$, $C$) is satisfied for the computation represented by $\vec{s}$ if and only if

$$\langle A \cdot \vec{s} \rangle \times \langle B \cdot \vec{s} \rangle = \langle C \cdot \vec{s} \rangle.$$

For example, taking the constraint in (2) and the assignment in (1), we have

$$\begin{aligned} a &= \langle [0, 0, 0, 0, 0, 1, 1, 0, 0] \cdot [1, 9, 1, 2, 5, 4, 10, 14, 14] \rangle = 14 \\ b &= \langle [1, 0, 0, 0, 0, 0, 0, 0, 0] \cdot [1, 9, 1, 2, 5, 4, 10, 14, 14] \rangle = 1 \\ c &= \langle [0, 0, 0, 0, 0, 0, 0, 1, 0] \cdot [1, 9, 1, 2, 5, 4, 10, 14, 14] \rangle = 14. \end{aligned}$$

Finally, it is worth noting that constraints can be more involved than the simple examples here, and that coefficients are often larger than 1.

### 5.3 Polynomial-Encoding

In practical circuits, there can be thousands or millions of wires and constraints, so that verifying them all individually would be too expensive. Instead, zk-SNARKs proceed by encoding all the constraints into three polynomial vectors, which allows for the simultaneous verification of the whole constraint set.

Let $n$ represent the number of wires in an assignment, and $k$ the number of constraints. Denote these by

$$\begin{aligned} &\mathbb{C}_1 = (\mathbf{A}_1, \mathbf{B}_1, \mathbf{C}_1) \\ &\cdots\cdots \\ &\mathbb{C}_k = (\mathbf{A}_k, \mathbf{B}_k, \mathbf{C}_n) \end{aligned}$$

Let

$$[\mathbf{A}] = \begin{bmatrix} \mathbf{A}_1[1]\ \mathbf{A}_1[2] \cdots \mathbf{A}_1[n] \\ \cdots \\ \mathbf{A}_k[1]\ \mathbf{A}_k[2] \cdots \mathbf{A}_k[n] \end{bmatrix}$$

be the matrix of all $A$ linear combinations.

We can devise a polynomial vector $\mathcal{A}$ with $n$ elements that describes $[\mathbf{A}]$. Each member $\mathcal{A}_i$ of $\vec{\mathcal{A}}$ is a polynomial that encodes the $i$th column of $[\mathbf{A}]$. We assign to each constraint $\mathbb{C}_j$ a fixed scalar value, $\sigma_j$. The pairs $(\sigma_1, \mathbf{A}_1[i]), \ldots, (\sigma_k, \mathbf{A}_k[i])$ represent the $i$th coordinate of all the constraints as points on a plane. Now define $\mathcal{A}_i$ as a polynomial that passes through these $k$ points, for example as the result of the Lagrange interpolation. Define analogously $\vec{\mathcal{B}}$ and $\vec{\mathcal{C}}$ for the other linear combinations. Next, we gather the terms of all polynomials evaluated at the coordinate for constraint $j$ under the following notation:

$$\vec{\mathcal{A}}\,(\sigma_j) = [\mathcal{A}_1(\sigma_j), \mathcal{A}_2(\sigma_j), \ldots, \mathcal{A}_n(\sigma_j)] \tag{3}$$

and observe that by construction this is

$$\vec{\mathcal{A}}\,(\sigma_j) = [\mathbf{A}][j] = \mathbf{A}_j.$$

It can now be easily checked that, for an assignment $\vec{s}$ as above and an arbitrary constraint $\mathbb{C}_j$, with $j \in \{1, \ldots, k\}$, and for all $i \in \{1, \ldots, n\}$:

$$\langle \vec{\mathcal{A}}\,(\sigma_j) \cdot \vec{s} \rangle \cdot \langle \vec{\mathcal{B}}\,(\sigma_j) \cdot \vec{s} \rangle = \langle \vec{\mathcal{C}}\,(\sigma_j) \cdot \vec{s} \rangle \Leftrightarrow$$
$$\langle \vec{\mathcal{A}}\,(\sigma_j) \cdot \vec{s} \rangle \cdot \langle \vec{\mathcal{B}}\,(\sigma_j) \cdot \vec{s} \rangle - \langle \vec{\mathcal{C}}\,(\sigma_j) \cdot \vec{s} \rangle = 0 \Leftrightarrow$$
$$\langle \mathbf{A}_j \cdot \vec{s} \rangle \cdot \langle \mathbf{B}_j \cdot \vec{s} \rangle - \langle \mathbf{C}_j \cdot \vec{s} \rangle = 0.$$

is satisfied if and only if the computation satisfies the $j$th constraint.

The above expression can be further developed, to show that it represents a simple polynomial expression of the kind $\mathcal{P}(\sigma) = 0$:

$$\mathcal{P}(\sigma) = \left(\sum_{i=1}^{n} \mathcal{A}_i(\sigma) \cdot s_i\right) \cdot \left(\sum_{i=1}^{n} \mathcal{B}_i(\sigma) \cdot s_i\right) - \left(\sum_{i=1}^{n} \mathcal{C}_i(\sigma) \cdot s_i\right) = 0 \qquad (4)$$

## 5.4 Proof Construction

The computation verified by a zk-SNARK should be known by both the Prover and the Verifier, and therefore the set of constraints and the corresponding polynomial vector will also be known by both. These set the rules of the computation. Recall that the proof asserts knowledge of a witness $\vec{s}$ that passes those constraints. This section details how the Prover can convince the Verifier of that.

I focus on the polynomial (4) developed in the last section. If the computation satisfies all constraints, then by construction $\mathcal{P}(\sigma) = 0$ at least when $\sigma \in S = \{\sigma_1, \ldots, \sigma_k\}$ and possibly at other points. It is important to notice here that $\mathcal{P}(\sigma)$ is defined for a specific witness $\vec{s}$ and so encodes a specific computation.

By a consequence of the fundamental theorem of algebra, $\mathcal{P}(\sigma)$ must be a multiple of a polynomial $\mathcal{Z}(\sigma)$ that vanishes exactly in set $S$, that is, $\mathcal{Z}(\sigma) = 0 \Leftrightarrow \sigma \in S$. This polynomial is defined as:

$$\mathcal{Z}(\sigma) = (\sigma - \sigma_1) \cdot (\sigma - \sigma_2) \cdot \ldots \cdot (\sigma - \sigma_k).$$

The reverse is also true, that is, if $\mathcal{Z}(\sigma)$ evenly divides $\mathcal{P}(\sigma)$, then $\mathcal{P}(\sigma)$ must vanish on set $S$ and so all constraints are satisfied by the witness $\vec{s}$. Therefore, to prove the correctness of a computation, it is enough to demonstrate that $\mathcal{P}(\sigma)$ is a multiple of $\mathcal{Z}(\sigma)$ by showing $\mathcal{H}(\sigma)$ such that $\mathcal{P}(\sigma) = \mathcal{H}(\sigma) \cdot \mathcal{Z}(\sigma)$.

Notice that the Verifier can compute $\mathcal{Z}(\sigma)$, and that this determines what a Prover needs to compute in order to produce a convincing proof. Thus, the whole computation can be specified in a Quadratic Arithmetic Program composed of

$$\text{QAP} = (\vec{\mathcal{A}}, \vec{\mathcal{B}}, \vec{\mathcal{C}}, \mathcal{Z}). \qquad (5)$$

The polynomial $\mathcal{H}(\sigma)$ and the witness $\vec{s}$ are the heart of the proof. Although the verifier can not compute $\mathcal{P}(\sigma)$, because it does not know $\vec{s}$, it knows it must follow from QAP and that this is embedded in the proving and verification keys in a way that only a valid $\mathcal{H}$ will make the proof valid.

This encoding is the basis for QAP-based zk-SNARKs. They share the same 'front-end', by encoding the computation into a QAP in the same way. It is in the subsequent constructions, how that computation is encoded and how the proof is created, that they differ. That is not in the scope for this paper, and I encourage the reader to consult the references in Sect. 3.3.

## 6 Tools

Currently, there are a limited number of tools supporting the development of zk-SNARK applications. All of those I know focus mainly on QAP-based zk-SNARKs. These tools can be divided in 2 layers: Domain Specific Languages (DSL) that allow describing the proof predicate in a high-level language that is easy to learn and use; and support for the construction of the zk-SNARK algorithms, including (i) the representation of the statement in some technical intermediate language; (ii) support for the zk-SNARK specific algorithms (CRS generation, proof creation and verification); (iii) all the necessary mathematical support, for fast calculation in very large fields, elliptic curves and bilinear pairings. Most of these use R1CS as the intermediate language, which is then compiled into a QAP. R1CS is close to the language of arithmetic circuit satisfiability and therefore is **NP**-complete. For this reason, it has become very popular and is used by all the libraries reviewed here.

### *6.1 Zk-SNARK Support Libraries*

The most complete library to the date is also one of the first: libsnark.[7] It is written in C++ and offers wide flexibility. It mainly uses R1CS as the language for representing the proof predicate, but can support other more efficient (but possibly less flexible) languages, such as BACS, USCS, TBCS. libsnark offers also different kinds of elliptic curves, via its dependency libff,[8] supporting BN, Edwards and MNT curves. It also supports different kinds of Snarks, including BCTV14, Groth16 and GM17.

DIZK[9] is another library published by SCIPR Labs, and is in some aspects like a reduced port of libsnark to Java. Instead of relying on external libraries for calculation of FFT and fast arithmetic, it integrates its own implementations for these tasks, but in some cases (eg FFT) with a reduced algorithm. DIZK's main selling point is the

[7] https://github.com/scipr-lab/libsnark.

[8] https://github.com/scipr-lab/libff.

[9] https://github.com/scipr-lab/dizk.

support for parallelization to speed up the setup and proof generation. It supports only 2 BN curves, and offers only one kind of zk-SNARK, Groth16, which is QAP-based and described in the R1CS language.

Snarkjs[10] is a library for implementing zk-SNARKs in Javascript. Again it is limited to only one specific type of curve (BN128), and implements two Snarks, the original BCTV14 and Groth16. Due to the narrow elliptic curve focus, the library is smaller and probably easier to understand than either DIZK or libsnark.

Bellman[11] is a compact library, being developed for ZCash, that supports the creation of Groth16 Snarks. It is developed in Rust, and supports only Groth16, again based on an R1CS representation. Unlike the previous systems, it seems to use only the BLS12-381 curve, as promoted by ZCash since the Sapling version.

## *6.2 DSL Tools*

Typically, zk-SNARK libraries are difficult to use without a way to encode an arbitrary proof predicate. This niche is covered by some dedicated libraries.

ZoKrates[12] is a tool written in Rust and C++ that offers a very simple language to encode arithmetic circuits and R1CS. It interfaces with libsnark and Bellman, and allows the creation of three types of zk-SNARK: BCTV14, GM17 and since recently Groth16. It is very actively developed, and a good choice for beginners.

jSnark[13]/xjSnark[14] are a pair of libraries in Java that simplify the specification of zk-SNARKs at a high-level. Its approach is quite different of ZoKrates, in that the language is more low-level, centered around the definition of gadgets. But this makes it more flexible than ZoKrates, since the programmer can fine tune the construction of the circuit. Despite being written primarily in Java, its default backend is libsnark.

Circom[15] is the complement of Snarkjs. Also written in JavaScript, it provides a language that has similarities to ZoKrates, but in a more C-like way. Its intended backend is Snarkjs.

Finally, Snarky[16] is an OCaml front-end for creating R1CS-based Snarks. It uses the libsnark backend by default, and differs from the other DSL in that its language has a functional approach.

[10] https://github.com/iden3/snarkjs.

[11] https://github.com/zcash/librustzcash/tree/master/bellman.

[12] https://github.com/Zokrates/ZoKrates.

[13] https://github.com/akosba/jsnark.

[14] https://github.com/akosba/xjsnark.

[15] https://github.com/iden3/circom.

[16] https://github.com/o1-labs/snarky.

## References


1. Agrawal, S., Ganesh, C., Mohassel, P.: Non-interactive zero-knowledge proofs for composite statements. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology - CRYPTO 2018, pp. 643–673. Springer International Publishing, Cham (2018)
2. Backes, M., Barbosa, M., Fiore, D., et al.: Adsnark: nearly-practical privacy-preserving proofs on authenticated data. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P), May 2015
3. Bellare, M., Palacio, A.: The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols. In: Proceedings of Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, 15–19 August 2004. Lecture Notes in Computer Science, vol. 3152, pp. 273–289. Springer (2004)
4. Ben-Sasson, E., Bentov, I., Horesh, Y., et al.: Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptol. ePrint Archi. **2018**, 46 (2018)
5. Ben-Sasson, E., Chiesa, A., Genkin, D., et al.: Snarks for c: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO (2). Lecture Notes in Computer Science, vol. 8043, pp. 90–108. Springer (2013)
6. Ben-Sasson, E., Chiesa, A., Tromer, E., et al.: Succinct non-interactive zero knowledge for a von neumann architecture. In: Proceedings of the 23rd USENIX Conference on Security Symposium. SEC'14, pp. 781–796 (2014)
7. Ben-Sasson, E., Chiesa, A., Tromer, E., et al.: Scalable zero knowledge via cycles of elliptic curves. Algorithmica **79**(4), 1102–1160 (2017). https://doi.org/10.1007/s00453-016-0221-0
8. Bitansky, N., Canetti, R., Chiesa, A., et al.: From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. ITCS '12, pp. 326–349. ACM, New York (2012)
9. Blum, M., De Santis, A., Micali, S., et al.: Noninteractive zero-knowledge. SIAM J. Comput. **20**(6), 1084–1118 (1991)
10. Blum, M., Feldman, P., Micali, S.: Non-interactive zero-knowledge and its applications. In: Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing. STOC '88, pp. 103–112. ACM, New York (1988)
11. Bünz, B., Agrawal, S., Zamani, M., et al.: Zether: towards privacy in a smart contract world. IACR Cryptol. ePrint Arch. **2019**, 191 (2019). https://eprint.iacr.org/2019/191
12. Bünz, B., Bootle, J., Boneh, D., et al.: Bulletproofs: short proofs for confidential transactions and more. In: Proceedings of 2018 IEEE Symposium on Security and Privacy, SP 2018, San Francisco, California, USA, 21–23 May 2018, pp. 315–334 (2018)
13. Ciampi, M., Persiano, G., Scafuro, A., et al.: Improved or-composition of sigma-protocols. In: Proceedings of Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, Part II, 10–13 January 2016, pp. 112–141 (2016). https://doi.org/10.1007/978-3-662-49099-0_5,
14. Cramer, R., Damgård, I., Schoenmakers, B.: Proofs of partial knowledge and simplified design of witness hiding protocols. In: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '94, pp. 174–187. Springer, London (1994). http://dl.acm.org/citation.cfm?id=646759.705842
15. Danezis, G., Fournet, C., Groth, J., et al.: Square span programs with applications to succinct NIZK arguments. In: ASIACRYPT (1). Lecture Notes in Computer Science, vol. 8873, pp. 532–550. Springer (2014)
16. Fiat, A., Shamir, A.: How to prove yourself: practical solutions to identification and signature problems. In: Proceedings on Advances in Cryptology—CRYPTO '86, pp. 186–194. Springer, London (1987)
17. Frankenfield, J.: Coinjoin, July 2018. https://www.investopedia.com/terms/c/coinjoin.asp. Accessed 27 May 2019
18. Gabizon, A.: On the security of the BCTV pinocchio zk-snark variant. IACR Cryptol. ePrint Arch. **2019**, 119 (2019)

19. Gennaro, R., Gentry, C., Parno, B., et al.: Quadratic span programs and succinct nizks without pcps. In: Proceedings of Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, 26–30 May 2013, pp. 626–645 (2013)
20. Goldreich, O., Micali, S., Wigderson, A.: Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. J. ACM **38**(3), 690–728 (1991)
21. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof-systems. In: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing. STOC '85, pp. 291–304. ACM, New York (1985)
22. Groth, J.: On the size of pairing-based non-interactive arguments. In: EUROCRYPT (2). Lecture Notes in Computer Science, vol. 9666, pp. 305–326. Springer (2016)
23. Groth, J., Maller, M.: Snarky signatures: minimal signatures of knowledge from simulation-extractable snarks. In: CRYPTO (2). Lecture Notes in Computer Science, vol. 10402, pp. 581–612. Springer (2017)
24. Jedusor, T.E.: Mimblewimble (2016). https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt. Accessed 27 May 2019
25. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982)
26. Maurer, U.: Zero-knowledge proofs of knowledge for group homomorphisms. Des. Codes Cryptogr. **77**(2–3), 663–676 (2015). https://doi.org/10.1007/s10623-015-0103-5
27. Meckler, I., Shapiro, E.: Coda: decentralized cryptocurrency at scale (2018). https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf. Accessed 30 May 2019
28. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008). https://bitcoin.org/bitcoin.pdf. Accessed 27 May 2019
29. Parno, B., Howell, J., Gentry, C., et al.: Pinocchio: nearly practical verifiable computation. In: IEEE Symposium on Security and Privacy, pp. 238–252. IEEE Computer Society (2013)
30. Poelstra, A.: Mimblewimble (2016). https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf. Accessed 27 May 2019
31. Sasson, E.B., Chiesa, A., Garman, C., et al.: Zerocash: decentralized anonymous payments from bitcoin. In: 2014 IEEE Symposium on Security and Privacy, May 2014, pp. 459–474