LEARNING ZK-SNARKS FROM THE BEGINNING

Joshua Jones
Advised by Dr. Stephen Graves
from The University of Texas at Tyler
jjones105@patriots.uttyler.edu

December 8, 2022

ABSTRACT

In this paper, we wish to explore how a succinct, non-interactive, zero-knowledge proof (zk-SNARK) can be constructed. The zk-SNARK, a type of cryptographic proof, requires the following components: the NP problem (a decision problem that has polynomial-time verification) to be verified it has been solved, a program representing the verification, a quick way to check the verification, an encryption process to make the information being verified unknown, and an asynchronous verification process.

The components will utilize taking a program, defining the program as a satisfiable arithmetic circuit, taking the arithmetic circuit and converting it to a Rank-1 Constraint System (R1CS), and transforming the R1CS to a Quadratic Arithmetic Problem (QAP). Once we have a sufficient QAP, we have the desired proof structure. To incorporate the zero-knowledge part of the proof, we will shift the components in the QAP in a specific way to prevent knowledge from being divulged during the verification process. Lastly, we will utilize the Knowledge of Exponent Assumption (KEA) to prevent any fake proofs to be accepted.

***K*eywords** zero-knowledge proof · verifiable computation · polynomial transforming · encryption

1 Introduction

In this article, we wish to explore some implementations of cryptographic proof systems known as zero-knowledge succinct non-interactive argument of knowledge abbreviated to zk-SNARK. In around the 1980s, [3] developed the fundamentals of zk-SNARKs. The components of zk-SNARKs can be defined as follows:

1. Succinct: The time the proof system takes is short.
2. Non-interactive: The proof can be written and stored to be read and verified at a later time.
3. ARguments: The proof system can only generate fake proofs if enough computational power is provided. This is also known as "computational soundness."
4. of Knowledge: It is impossible for the proof system to generate a proof without knowing valid inputs.

Let us dive more into what a zero-knowledge proof system is.

1.1 What is Zero-knowledge

We first start with a given scenario. Suppose Bob has a password x that he uses to access his online bank account. Bob first rewrites x by using the *one-way function*[1] $h(x) = z$ and sends z to his online bank. Later, Alice, who works for the bank, wants to verify Bob is the same person that created z. That is, Bob needs to show Alice that he knows the preimage of h. By knowing the preimage of h Alice is convinced that either Bob is the same person that created z or Bob was able to computationally find the preimage of h; the latter of which is assumed to be infeasible for Bob. While Bob could show x to Alice and allow her to perform h, this would compromise Bob's password. Thus Bob needs a way to convince Alice without revealing x.

In the above scenario, Bob is known as a prover and Alice is known as a verifier. A prover wishes to convince the verifier that a particular statement is true, while the verifier makes sure that the prover doesn't try to falsify the validity of the statement when proving it. The condition that Bob needs to convince Alice without revealing x is known as zero-knowledge. The way Bob convinces Alice is known as a proof system.

1.2 Defining a Proof System

For brevity, we will informally define what a proof system is. A proof is something that convinces another person that a given statement is true. A *proof system* is the step-by-step process taken that determines what is considered a legitimate proof for the statement. A proof system will return that a proof either does or doesn't validate a statement. Let P_1 be a prover and V_1 be a verifier. A proof system's properties must include

- Completeness: P_1 must be able to convince V_1 of any true statement since a statement is true if a convincing proof exists.
- Soundness: If P_1 tries to convince V_1 of a false statement, then V_1 will reject the proof given by P_1 with very high probability.

Additionally, we would like the process of verifying and proving of the proof to be quick. That is, V_1 should be able to verify the proof given by the P_1 quickly, and P_1 should be able to generate a proof quickly, provided P_1 has valid inputs for the statement to be true. The process of verifying can be done by checking the *computation* of the statement. The computation of the statement is the equivalent of encoding the problem on a computer and verifying if the inputs can match the outputs.

One question that should arise to the reader is, "What statements can be checked using a zero-knowledge proof system?" To answer this, we must define some computational complexity theory.

2 What is Considered a Valid Statement

To determine what kind of statements can be implemented using zk-SNARKs, we define the scope of the feasibility of executing the statement on a computer machine, M. For the statement to run on M, the statement in compiled[2] into

[1] A one-way function is a function f such that requires an infeasible amount of computations to obtain the preimage of f, but where f can be quickly performed

[2] When a statement is compiled, the algorithm for executing the statement is written as code by the user and is translated into machine code which is a string of only zeros and ones.

some function f which takes an input (variable or vector of variables) denoted x. In particular, we want to relate how long it takes to execute, defined as *run-time*, f with the size of x and the amount of resources a computer takes.

2.1 Classes P and NP

For the purpose of zk-SNARKs, we can define a statement to be equivalent to what is known as a *decision problem*. A decision problem is a question that asks if a given input fulfills a desired property. For the case of zk-SNARKs, we want to verify if a given proof is valid. For example, $(f(x) == y) \in \{0, 1\}$ where 0 represents "false" and 1 represent "true" to checks if x fulfills the property that $f(x) = y$. Formally, the definition of the corresponding language to a decision problem S is.

$$\mathcal{L}_d = \{w : w \text{ is the compiled instance of } S \text{ given inputs } y \text{ such that } S(y) = 1\}$$

In general, a language $\mathcal{L}$ is a set of strings over an alphabet. For computers, the alphabet is just $\{0, 1\}$. So the language of computers is any length of binary strings $a_0 a_1 \dots a_n$ formed by using $a_i \in \{0, 1\}$. Notice a computation $w \in \mathcal{L}_d$ if w provides a valid execution of some algorithm representing S with given inputs y. Expanding from just decision problems, we can classify any problem[3] S based on how long a valid execution of the encoded (and compiled) problem takes on M. The two classifications are as follows:

Definition 2.1. We define the set **P** to be all decision problems that can be solved by a deterministic Turing machine[4], M, in polynomial time. That is, it takes at most $O(n^k)$ steps to execute given an input of size n. Extending the definition to a decision language $\mathcal{L}_d$, we have $\mathcal{L}_d \in$**P** if and only if there exist an deterministic M such that

- M runs in polynomial time on all inputs x of size n.
- For all $w \in \mathcal{L}_d$, M outputs 1.
- For all $w \notin \mathcal{L}_d$ M outputs 0.

Definition 2.2. We define the set **NP** to be all decision problems that can be solved by a non-deterministic Turing machine and can be verified in polynomial time. Notice the solution is checked in $O(n^k)$; however, obtaining the solution is not necessarily in polynomial time. Also, $\mathcal{L}_d \in$**NP** if and only if there exist a M such that

- M verifies a solution in polynomial time on all inputs x of size n.
- For all $w \in \mathcal{L}_d$, M outputs 1.
- For all $w \notin \mathcal{L}_d$ M outputs 0.

The main difference between the two classes is that if a problem $S \in$**P** problems it is easy to create a solution; however, a problem $S \in$**NP** can only be verified quickly. The speed of finding or verifying a solution determines whether the machine M is deterministic or non-deterministic.

Remark 2.3. At the time of writing this paper, it is conjectured that **P**$\neq$**NP**. The functionality of zk-SNARKs only works if this holds to be true, otherwise, there would exist a polynomial time algorithm to compute a solution to any hard problem (including encryption). These hard problems are the foundation of cryptography; for this article, we assume this to be true.

The types of statements zk-SNARKs use correspond to NP problems since we want some statement that is quick to verify but is hard to solve[5]. The solution to the hard-to-solve problem is called a *witness*, denoted w. Since the problem is hard to solve for V_1, it is not feasible for V_1 to calculate w but is feasible to verify $w \in \mathcal{L}_d$. Before we continue, let us see an example of a problem in **NP** and why it is easy to verify, but hard to solve. let us consider a decision version of a knapsack problem.

Example. Suppose a store has a deal that allows you to fill a bag with different games and pay v dollars for the bag. You want to maximize the worth of the contents of the bag without exceeding the weight the bag can hold, denoted b. Given n games weighing $b_1, b_2, \dots, b_n$, and priced at values $v_1, v_2, \dots, v_n$, is there a way to fill the bag with the n games such that the bag doesn't break due to exceeding the weight and the value of the games inside the bag is greater than v? More formally, is there a $B \subseteq \{1, 2, \dots, n\}$ such that $\sum_{i \in B} b_i \leq b$ and $\sum_{i \in B} v_i \geq v$.

[3] A problem is any abstract question to answer

[4] A Turning machine is a model that describes how a computer system evaluates a provided algorithm

[5] A problem is hard to solve if the only known algorithms to find a solution are through brute-force (trying every possible combination of inputs in M) to see if there exist a solution.

Notice, it is quick to check whether the games corresponding with indices in B exceed b and not exceed v. Simply add the weights and values. This would take $O(n)$ steps, where n is the number of games chosen. If we encoded this process onto a machine, M, we have the decision Knapsack problem is in **NP**. Also note the worst case is that every possible combination of chosen games doesn't exceed v. For a computer, the time to check every possible combination to find a solution is $O(2^n)$, which is not in polynomial time.

We want to use zk-SNARKs for these types of problems in **NP**, however, we also want to find a way to represent any problem in **NP** as a single NP problem allowing us to have a single proof system construction. By finding an NP problem with this characteristic, we can create an algorithm to verify any type of NP problem using a zk-SNARK. In other words, we wish to find a problem that we can implement that is **NP-Complete**.

Definition 2.4. A problem S is NP-Complete if the following hold:

1. It is a problem for which the correctness of each solution can be verified quickly (namely, in polynomial time) and a brute-force search algorithm can find a solution by trying all possible solutions.
2. The problem can be used to simulate every other problem for which we can verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly, but not solved quickly. Note that if we could find solutions to some NP-complete problem quickly, we could quickly find the solutions to every other problem to which a given solution can be easily verified.

To find an NP-Complete problem we have to determine if there is a way to map every instance of a language $\mathcal{L}$, to the language of the NP-Complete problem. We can define this mapping as follows:

Definition 2.5. A language $\mathcal{L}_d$ is polynomial time reducible to a language $\mathcal{L}_{d_2}$ if there exist a polynomial runtime function f such that for all w, $w \in L_d$ if and only if $f(w) \in \mathcal{L}_{d_2}$. The function f is known as a *reduction function*.

Fortunately, Cook Levin proved such a problem known as the *Boolean satisfiability problem* (SAT) defined as

Definition 2.6. (Boolean Satisfiability Problem)(SAT) Given a Boolean formula B, we claim it possible to assign truth values of 0 being false and 1 being true to B's variables w such that $B(w) = 1$.

Theorem 2.7. *The Cook-Levin Theorem SAT is NP-Complete*

Proof. Given in [8] The proof contains two parts: we must show $SAT \in$**NP** and that any other NP problem is polynomial reducible to SAT. First notice that for any Boolean formula, to check if an input is satisfiable takes $n \in O(n)$ Boolean operations that are encoded on a nondeterministic Turing machine. Hence SAT is in **NP**. Let us now define a problem $A \in$**NP** that is encoded on a nondeterministic Turing machine M that decides if $w \in A$ or $w \notin A$ in $O(n^k)$ time. We define a reduction formula f to be

$$f(w) = b(M, W)$$

such that $w \in A$ if and only if $b(M, W$ is satisfiable. To describe $b(M, w)$, we will use the following notation: define a tableau, T for M on w is a $n^k \times n^k$ table which rows describe the configuration history[6] of M given the input w, seen in Figure 1.

[6]The configuration history of M is the recorded configurations that M goes through, where a configuration includes each state M is in and the current order of inputs. A more formal introduction can be found in [8] on page 31

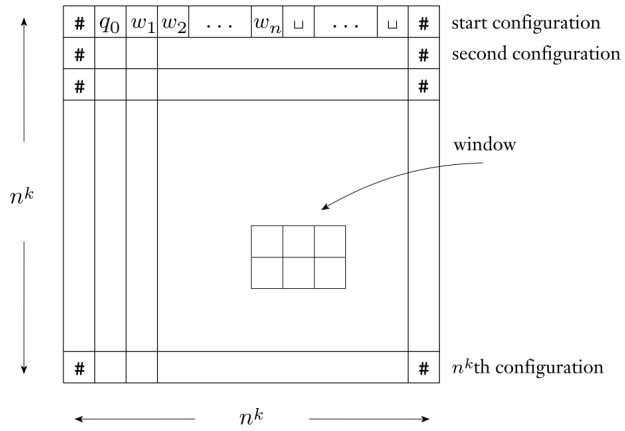

Figure 1. A Tableau representing M's configuration states

The notation in Figure 1 is defined as follows:

- $q_i \in Q$ where Q is the set of all states in M.
- $w_i \in w$ where w is the list of all input variables.
- # represents the beginning and ending of each state configuration.
- $\sqcup$ represents padding to match the length of all configurations to the largest configuration history of M of size n.

It suffices to show if there exists a T for M on an input list w, then M return w is a satisfiable argument for A. Notice for T to exist, every row configuration must be a valid configuration. We can now define the formula $B(M, w)$ produced by the reduction function f. Let $B(M, w)$ be the formula

$$B(M, w) = B_{\text{cell}} \wedge B_{\text{start}} \wedge B_{\text{move}} \wedge B_{\text{accept}}$$

Let us begin defining the variables of $B(M, w)$. Suppose $C = Q \cup \{w_1, w_2, \ldots, w_n, \sqcup\} \cup \{\#\}$ where C are all possible inputs in a entry of T, called a *cell*. We can represent if a cell is full by letting $x_{i,j,c} = 1$ if a cell in T contains the variable $c \in C$ where i, j represent the i^{th} row and the j^{th} column of T. The first thing we have to check on M is that each cell is assigned exactly one variable. We can represent this as the Boolean formula[7].

$$B_{\text{cell}} = \bigwedge_{1 \le i,j \le n^k} \left[\left(\bigvee_{c \in C} x_{i,j,c} \right) \wedge \left(\bigwedge_{\substack{c_1, c_2 \in C \\ c_1 \neq 1 c_2}} (\overline{x_{i,j,c_1}} \vee \overline{x_{i,j,c_2}}) \right) \right]$$

The next part of determining an accepting T is verifying that the first row of T contains the initial state of the computer, all of the input variables, and any remaining cells filled with the $\sqcup$ variable. We can check this using the following formula

$$B_{\text{start}} = x_{1,1,\#}, \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge x_{1,4,w_2} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \ldots \wedge x_{1,n^k-1,\sqcup} \wedge x_{x_1,n^k,\#}$$

where q_0 is the initial state of M. Similarly, we can check to see if M on w is accepted; that is the state when M returns an answer to the encoded decision problem is present in the last row of T.

$$B_{accept} = \bigvee_{1 \le i,j \le n^k} x_{i,j,q_{accept}}$$

[7]Note that the symbol $\bigvee_{i \le n}$ represents $a_1 \vee a_2 \vee \ldots \vee a_n$. The case is similar for the symbol $\bigwedge$

Lastly, we have to check that each configuration corresponds to the preceding row's configuration according to M's rules[8]. We want to check each 2×3 window located in the tableau does not violate M's transition function[9] Define W_l to be the set of legal windows. We then check if each window in T is in the set W_l. Therefore we can define

$$B_{\text{move}} = \bigwedge_{\substack{1 \le i < n^k \\ 1 < j < n^k}} \bigvee_{w_c \in W_l} (x_{i,j-1,w_{c_{1,1}}} \wedge x_{i,j,w_{c_{1,2}}} \wedge x_{i,j+1,w_{c_{1,3}}} \wedge x_{i+1,j-1,w_{c_{2,1}}} \wedge x_{i+1,j,w_{c_{2,2}}} \wedge x_{i+1,j+1,w_{c_{2,3}}})$$

Notice the inner **or** statement will evaluate to 1 whenever the current window being checked matches some valid window w_c. Also, all possible windows are predetermined by the rules of M.

Now that we have defined $B(M, w)$ we need to verify that the reduction of f is in polynomial time. To do so, we will examine the size of $B(M, w)$. Notice the size of $B(M, w)$ corresponds as follows.

1. The size of T is n^{2k} with each cell having l possible values, where l is the size of C. Thus the total number of variables is $O(n^{2k})$.
2. Each cell has exactly one symbol from the set of input variables and states of the machine; executing B_{cell} takes $O(n^{2k})$ steps.
3. The first row of T must start with the beginning state of M, contain all the inputs of w and fills the remaining cells with blanks, $\sqcup$; executing B_{start} takes $O(n^k)$ steps.
4. The last row contains the state that shows M accepts w as an acceptable configuration; executing B_{accept} takes $O(n^k)$ steps.
5. Each 2×3 window within T to confirm T corresponds to a legal computation history; executing B_{move} takes $O(n^{2(k-2)})$ steps.

Notice the reduction of A using f that takes $O(n^{2k}) + O(n^k) + O(n^k) + O(n^{2(k-2)}) = O(n^{2k})$ steps. Since f is a polynomial time reduction of A to SAT, SAT is NP-Complete. ■

Less formally, the Cook-Levin Theorem allows us to validate a given input to a problem on a machine that returns if the input to the problem is valid or not. Since computers are based on Boolean logic, any computation can be verified using Boolean SAT. This is almost exactly what we want for our purpose of zk-SNARKs. The only negative consequence of Boolean satisfiability is the sheer number of variables that have to be verified. Fortunately, we can take a higher-level approach by using arithmetic satisfiability which can be seen in [2]. We now have that any instance of a decision problem, given an input, can be checked using satisfiability. Next, we will define a method of constructing a zk-SNARK that implements a way to check this satisfiability.

3 Constructing a zk-SNARK

Before proceeding forward, we first need to define some characteristics of the arithmetic done in zk-SNARKs. First, all of the zk-SNARK construction is done in finite field arithmetic. For any prime number p, we define $\mathbb{F}_p$ to be a set of integers modulus p. For the remainder of the paper, we will assume all operations are done in F_p. The construction of zk-SNARK also relies on writing our computation in the form of univariate polynomials, which will be in the ring of $\mathbb{F}[z]$. Though not we will not explain why now, the reason will become clear later on.

While there are multiple construction variants of zk-SNARKs we will focus on the Pinocchio protocol created by [3] with the added feature of zero-knowledge as the primary focus of the Pinocchio protocol was to prevent the prover from providing false proofs being accepted from the verifier.

The process of creating a zk-SNARK can be generalized into six steps.

1. Computation
2. Arithmetic Circuits
3. Rank 1 Constraint System

[8]For brevity, we reference [8] to explain the rules of M

[9]A *transition function defines a legal window by specifying three things: the symbol to be written in the next configuration, the direction the next state will be, and the next window.*

4. Quadratic Arithmetic Program (QAP)
5. Encryption for Zero Knowledge and Verification
6. zk-SNARK system

As a little foreshadowing, QAPs provide a succinct (fast) way to verify the computation was done; however, the steps to building that desired speed require getting steps 1-3.

3.1 Computation

We know we have an NP-Complete problem using arithmetic satisfiability; that is we, can write any decision problem in the form of this arithmetic problem. Our objective is to express the verification process of a proof sent by the prover to the verifier in the form of some computation. Since the verification process of a proof is realized by some programming language[10], we can represent the verification of the proof using an encoded version. Since we operate in a finite field, there are some restrictions on the properties of the encoded version which include:

1. All variable assignments are immutable
2. No type of loops (recursive or iterative)
3. Only use addition and multiplication operators

Notice the 3rd restriction isn't necessary restricting as any negative number by repeating $-a \equiv p - a \pmod{p}$ until $p - a$ returns a positive value. Division can also be done by utilizing the extended euclidean algorithm or Fermat's Little Theorem as both take $O(\log n)$ where n is the value being divided.

Theorem 3.1. *Fermat's Little Theorem Let p be a prime number and let a be any number with $a \neq 0$ (mod p). Then $a^{p-1} \equiv 1$ (mod p).*

Applying Fermat's Little Theorem we can expand $a^{p-1} = a \cdot a^{p-2}$. Therefore we have that any fraction $\frac{1}{b} \pmod{p} \equiv b^{p-2} \pmod{p}$. We then can apply a series of multiplications to obtain $\frac{a}{b} \pmod{p}$. Exponentiation of the form a^k for some $k \in \mathbb{Z}$ can be done similarly with a series of multiplications. Lastly, we can represent if statements with clever multiplication. Let's consider the python code below.

```
def ifExample(c, x,y):
    if c:
        return x
    else:
        return y
```

We can represent this as the function $f(c, y, x) = c \cdot x + (1 - c) \cdot y$ for some $c \in \{0, 1\}$ and $x, y \in \mathbb{F}_p$. For the construction of inequalities, there are better zk-SNARK implementation methods[11] that can be used. Equality operation, however, can be implemented as their own equation that needs to be verified. Remember, we want to verify the execution of the code, not directly execute it. Once we have our encoded version of our problem we want to represent it as an arithmetic circuit, which we will see in the next section.

3.2 Arithmetic Circuits and RICS

Definition 3.2. An arithmetic circuit Φ over a field $\mathbb{F}$ and a set of variables $X = \{x_1, x_2, \ldots, x_n\}$ is a directed acyclic graph with the following properties:

1. The nodes of Φ are called gates.
2. The edges of Φ are called wires.
3. A gate is of in-degree l if it has l incoming wires
4. A gate is of out-degree 0 is called an output gate.
5. Every gate of in-degree 0 on Φ are labeled by a variable $x_i \in X$ or a $y \in \mathbb{F}$

[10] The high-level code written in a computer

[11] Inequalities are done utilizing bit-decomposition and are not supported with this particular proof system

6. All other gates are labeled by the addition, $+$, and multiplication, $\times$, operators.

7. (Optional) The labeled edges correspond to any generated variables $d_1, \ldots, d_m$.

We call Φ a formula if it is a directed tree whose edges are directed from the leaves to the root.

For zk-SNARK our primary focus is on bilinear gates, that is gates with in-degree 2 and our-degree 1. The two types of gates mentioned above can be seen below for $x_1, x_2 \in \mathbb{F}$ and a generated variable d_1.
Note that either gate has a left input, right input, and an output; in this case labeled x_1, x_2, and d_2 respectively.

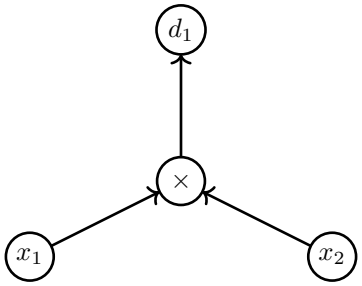

Figure 2. Multiplication Gate

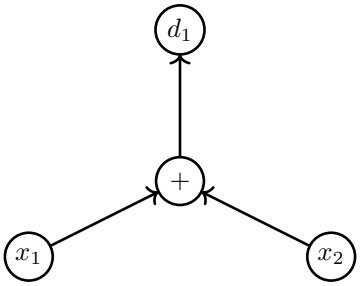

Figure 3. Addition Gate

Additionally, we have $x_1 \times x_2 - d_1 = 0$ or $x_1 + x_2 - d_1 = 0$. In general, these equations are known as constraints. For verification purposes in zk-SNARKs, we want to check to make sure these constraints are valid for every gate in the arithmetic circuit. Since zk-SNARK utilizes gates of in-degree 2, any computation that we wish to verify must be broken down into a system of equations with only one operation in each equation. There are three possible variations that can occur in the system of equations. For a, b, c we have

1. $a = b$ where, b is a value in $\mathbb{F}_p$ or a variable and a is a variable.
2. $a = b \circ c$ where $\circ$ is either an addition or multiplication operator.

The way of obtaining this equation is through a process called flattening. Let us return to our example code represented as the function $f(c, x, y) = c \cdot x + (1 - c) \cdot y$. Notice it is equivalent to the following system and arithmetic circuit. As

$$
\begin{aligned}
c \cdot x &= d_1 \\
(-1) \cdot c &= d_2 \\
1 + d_2 &= d_3 \\
d_3 \cdot y &= d_4 \\
d_1 + d_4 &= r
\end{aligned}
$$

where r is the output of $f(c, x, y)$ and d_1, d_2, d_3 and d_4 are generated variables.

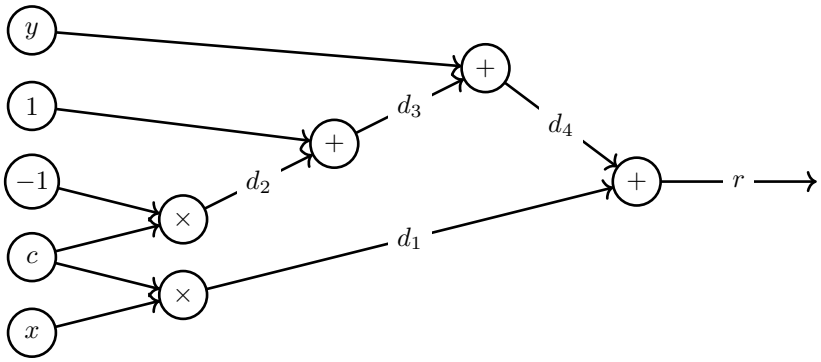

Figure 4. Example Arithmetic Circuit

we can see, an arithmetic circuit provides all the constraints the verifier needs to check to make sure the prover isn't trying to falsify doing a computation; that is an instance[12] of the arithmetic circuit is satisfiable. Given an input, the verifier can take the claimed output and match it to each input with respect to the corresponding inputs. However, with zk-SNARKs, the verifier is not allowed to see the input variables as that would break the zero-knowledge attribute of the proof system. We want to write the system of equations in a way where we can check each operation to its corresponding inputs. In other words, we will convert an arithmetic circuit into a Rank-1 Constraint system (R1CS).

[12]Given inputs of the arithmetic circuit are substituted into the variables

Definition 3.3. A R1CS is a set of three matrices A_l, A_r, A_o and a solution (a.k.a. witness), $\vec{w}$, whose entries correspond to values in $\mathbb{F}_p$. We say a R1CS is *satisfiable* if and only if $\vec{z}$ with $z_1 = 1$ such that

$$A_l\vec{z} \circ A_r\vec{z} - A_o\vec{z} = \vec{0}$$

where $\circ$ denotes the Hadamard product[13].

Utilizing R1CS allows us to verify equality operations in our arithmetic circuit along with any equality operations the verifier needs to check. Consider a generalized instance of an arithmetic circuit Φ with inputs $\vec{x}$ and output r, where the prover wants to convince the verifier that there exists a $\vec{w}$ with entries in $\mathbb{F}_p$ such that $\vec{w}$ satisfies Φ using the equation of R1CS. We can describe the setup as seen in [9] Define N to be the number of arithmetic gates in Φ plus the length of $\vec{x}$ plus 1 (for output). Define $M = N - |\vec{w}|$. Then the R1CS will have three $M \times N + 1$ matrices denoted A_l, A_r, A_o. Then we define $\vec{z}$ such that $z_1 = 1$ and any other entries as the entry from $\vec{x}$, the output r, or an auxiliary (generated) variable from a gate of Φ.

We want the i^{th} row to show the inputs of x hold if and only if z_i equals x_i. To do this, we set the i^{th} of A_l to be the standard basis vector $e_1 \in \mathbb{F}^{N+1}$, the i^{th} of A_r to be the standard basis vector $e_i \in \mathbb{F}^{N+1}$, and the i^{th} of A_o to be $x_i \cdot e_i$. This setup asserts that for the i^{th} row in the system we have $z_i - x_i = 0$ and $z_j - r = 0$ where j corresponds to the value of z checked to be equal to the output.

We also have to show the constraints hold for multiplication and addition gates. For each entry i of z corresponding to an addition gate C, we set the i^{th} row A_l to be the standard basis vector $e_1 \in \mathbb{F}^{N+1}$ and the i^{th} row of A_r to be $e_{j_1} + e_{j_2} \in \mathbb{F}^{N+1}$, and i^{th} row of A_o to be e_i. Note that j_1, j_2 correspond to the index of two wires coming into the addition gate. Notice for each row asserts we have $z_{j_1} + z_{j_2} - z_j = 0$. For multiplication, we set the i^{th} row of A_l to be the standard basis vector $e_{j_1} \in \mathbb{F}^{N+1}$, the i^{th} row A_r to be the standard basis vector $e_{j_1} \in \mathbb{F}^{N+1}$, and the i^{th} row of A_o to be $e_j \in \mathbb{F}^{N+1}$. Each row in the matrices will now assert $(z_{j_1} \cdot z_{j_2}) - z_j = 0$.

Not that we may add rows to A_l, A_r, A_o to provide equality checks. Keep in mind the size of the matrices and z will change by however many equality checks are added. Let us suppose we want to check $x_i == x_j$. This is equivalent to stating $x_i - x_j = 0$, which corresponds to an addition gate and a multiplication gate as seen in our previous code example. With the structure of an R1CS defined, we now have a proof system that verifies any problem converted to an arithmetic circuit was executed by a prover properly. Notice that a prover will not satisfy the equation of an R1CS if a witness $\vec{w}$ doesn't provide valid inputs into $\vec{z}$.

We now have a simple proof system; however, the process is not very succinct. To speed up the process of evaluating the R1CS we will convert it into a Quadratic Arithmetic Program that utilized the properties of polynomials.

3.3 Quadratic Arithmetic Programs (QAP)

Before we define what a QAP is, the readers must know the properties of univariate polynomials in a ring[14], $\mathbb{F}[x]$. For the purpose of this paper, we assume the reader has knowledge on this subject including the operations that can be done in a polynomial ring.

The definition of Quadratic arithmetic programs is given by [2], and is as follows:

Definition 3.4. (Quadratic Arithmetic Programs) QAP A quadratic arithmetic program (QAP) Q over field $\mathbb{F}$ contains three sets of polynomials $\mathcal{A} = \{l_k(x) : k \in \{0, \ldots, m\}\}, \mathcal{B} = \{r_k(x) : k \in \{0, \ldots, m\}\}, \mathcal{C} = \{o_k(x) : k \in \{0, \ldots, m\}\}$, and a divisor polynomial $D(x)$, all from $F[x]$. Let $f : \mathbb{F}^n \longrightarrow \mathbb{F}^{n'}$ be a function having input variables with indices $1, \ldots n$ and output variables labeled $m - n' + 1 \ldots m$. We say that Q is a QAP that computes f if the following is true. $a_1, \ldots a_n, a_{m-n'-1}, \ldots, a_m \in \mathbb{F}^{n+n'}$ is a valid assignment to the input/output variables of f if and only if there exists $\vec{a} = (a_{n+1}, \ldots a_{m-n'} \in \mathbb{F}^{m-n-n'}$ such that $D(x)$ divides:

$$Q_e(x) = \underbrace{((l_0(x) + \sum_{k=1}^{m} a_k \cdot l_k(x))}_{A(x)} \cdot \underbrace{(r_0(x) + \sum_{k=1}^{m} a_k \cdot r_k(x))}_{B(x)} - \underbrace{(o_0(x) + \sum_{k=1}^{m} a_k \cdot o_k(x))}_{C(x)} \tag{3.1}$$

Note that the size of Q is m and the degree of Q is equivalent to the degree of $D(x)$. Note also stating $D(x)$ divide $Q_e(x)$ is equivalent to stating there exist an $H(x)$ such that $Q_e(X) = D(x)H(x)$

[13] entry wise multiplication between vectors

[14] A type of algebraic structure with polynomials of the form $c_0 + c_1x + c_2x^2 + \ldots + c_{m-1}x^{m-1} + c_mx^m$ where $\{c_0, \ldots, c_m\}$ is the set of coefficients and m represents the highest degree of the polynomial. Not also, $c_m \neq 0$

The polynomial $D(x)$ is known as a target polynomial and is commonly a zero polynomial in $\mathbb{F}[x]$. The objective of QAPs are to create where the equation 3.1 provides a linear combination of the sets of polynomials that is a multiple of the target polynomial. That way whenever a prover inputs their witness vector values $\vec{w}$ to $\vec{a}$, $D(x)$ will only divide the linear combination if $\vec{w}$ is a satisfying argument.

Now that we have the structure of a QAP defined, we need to convert our R1CS into polynomials that can be evaluated at a given x value that correspond to the rows represented in A_l, A_r, and A_o. We can do this utilizing Lagrange interpolation[15].

Definition 3.5. A Lagrange interpolating polynomial is a polynomial $G(x)$ of degree $d < n$ that passes through n points $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ defined as

$$G(X) = \sum_{i=1}^{n} y_i \Pi_{j=1}^{n} \frac{x - x_j}{x_i - x_j}$$

In other words, for each row in A_l, A_r, A_o we take the corresponding index, i, and entry a_i and create the set of points $\{(i, a_i) : i \in \{0, \ldots N + 1\}$ interpolating them into a polynomial $l(x), r(x)$, or $o(X)$ respectively. Applications for generating these polynomials use fast Fourier Transforms as they provide an efficient algorithm for multiplying polynomials. An implementation of this can be seen in [6]. Once we transform all the rows of the listed three matrices, we also provide the same $\vec{z}$ as the values for $\vec{a}$. With this, we have an equivalent mapping of the RICS to a QAP.

Up to this point, the evaluation isn't much of an improvement since we are still manipulating the vector of polynomials[16]. In this case, we have the degree of each polynomial is equivalent to the number of rows in a given A_i matrix. Similarly, the number of polynomials corresponds to the number of entries in a given matrix row. The trick is that we want to evaluate all of these values at once. That is the verifier will pick a random point s, plug it into x and solve. When we do this, we are simply evaluating $Q(s) \in \mathbb{F}_p$ and dividing by $D(d) \in \mathbb{F}_p$ which we can use methods mentioned previously to evaluate. If we get $Q(s)$ is divisible $D(s)$, there is a sufficiently small probability that the prover didn't cheat. Notice with this method, we no longer guarantee that the prover doesn't cheat, rather, we take a probabilistic approach that finding a different $Q(s)$ that is divisible by $D(s)$ is sufficiently small. This sufficiently small probability follows from the Schwartz-Zippel Lemma.

Theorem 3.6. *(Schwartz Zippel Lemma) Let $P \in \mathbb{F}[x]$ be a non-zero polynomial of total degree $d \geq 0$ over a field $\mathbb{F}$. Let S be a finite subset of $\mathbb{F}$ and let $r_1, r_2, \ldots, r_n$ be selected at random independently and uniformly from S. Then the probability $P(r_1, r_2, \ldots, r_n) = 0 \leq \frac{d}{|S|}$, where $|S|$ is the size of S.*

Less formally, two polynomials of degree d will differ from at most d points. By letting our finite field S be the integers $\pmod p$ for p that is a really large prime. We have the probability of the prover cheating by evaluating using a different polynomial with the same random value s is d/p. Since p is really large, the probability of the prover guessing a valid solution is negligible. Thus, the knowledge that the prover is now trying to prove is that they know a $Q_e(x) = D(x) \cdot H(x)$, rather than the witness to satisfy the encoded problem.

3.4 Obtaining Non-interaction Constraint

At the current point in the paper, we have all the tools provided for the construction of zk-SNARK except the non-interactive and zero knowledge constraints. We will now discuss the non-interactive constraint by utilizing a *Common Reference String* (CRS).

Definition 3.7. A Common Reference String is a collection of parameters that both the prover and verifier rely on to operate in a proof system. The development of a CRS occurs during a trusted setup[17].

We will discuss what type of parameters are held in the CRS later in the paper. If we put all the parameters needed to execute the proof, any prover and verifier can utilize the CRS to operate the proof protocol. The parameters can be split into two groups: a proving key and a verification key. While we will only discuss the scenario of a single prover and single verifier there are techniques that can be implemented to allow a prover to prove to multiple verifiers as seen in [5][18]. We now have most of the construction for zero-knowledge proofs. Currently, we have a succinct way for a verifier to verify a prover's solution, provided the prover doesn't cheat by not using the described arithmetic circuit

[15]Note that interpolation in finite fields may *not* give a unique divisor polynomial however we will see later how to make sure this polynomial is enforced

[16]For implementation purposes, we represent the polynomial as a vector, where the entries correspond to the coefficients of the polynomial, namely $[c_0, c_1, \ldots, c_m]$

[17]A trusted setup is the process of creating random variables used for shifting in the elliptic curve space. Each party creating the CRS must prevent their random variables from being found

[18]See section 3.6 for details

mapped to a QAP. Additionally, we have not completely hidden the witness vector $\vec{w}$ or the polynomials the prover uses. To provide zero knowledge and verification, we use elliptic pairings as defined in [1].

Definition 3.8. Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be two cyclic groups of order r. Let g_1 be a generator[19] of $\mathbb{G}_1$ and g_2 be a generator of $\mathbb{G}_2$ both written in additive notation. A pairing is an efficient map $e : \mathbb{G}_1 \times \mathbb{G}_2 \longrightarrow \mathbb{G}_T$, where $\mathbb{G}_T$ is also a cyclic group of order r satisfying the following properties:

- bilinearity: For every nonzero element $\alpha, \beta \in \mathbb{F}_r$, we have $e(\alpha g_1, \beta g_1) = e(g_1, g_2)^{\alpha\beta}$
- non-degeneracy: $e(g_1, g_2)$ is not the identity in $\mathbb{G}_r$

Remark 3.9. In [1] it is noted that a clever choice of a pairing function, a pairing-friendly elliptic curve and instance of $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ can optimize implementations. For the construction of zk-SNARK using the Pinocchio protocol defines $\mathbb{G}_1 = \mathbb{G}_2$.

3.5 Evaluating Honestly with Zero Knowledge

Let us first look at how we would encrypt a polynomial $f(x)$ of degree d at a secret point s chosen by the verifier. First, we will choose a friendly-pairing elliptic curve function e and a generator g. Then, we have the verifier find all the exponents of s and encrypt them with the generator; that is the verifier finds

$$\{g^{s^i}\} \text{ for } i \in \{0, \ldots, d\}$$

These values allow a prover to compute the polynomial without actually knowing the value of s. For example $2s^2+3s+2$ is encrypted into

$$g^{2s^2+3s+2} = (g^{s^2})^2(g^{s^1})^3(g^{s^0})^2$$

However, the verifier will want to make sure the prover evaluates the polynomial function correctly. Thus the verifier will implement a shift α to each corresponding exponent of s as

$$\{\alpha g^{s^i}\} \text{ for } i \in \{0, \ldots, d\}$$

The verifier will then use the pairing function e to verify the prover's work. In particular, the verifier will use the property that

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

Hence the verifier will check the prover by verifying the following two equations are equivalent for some coefficients $c_0 \ldots c_d$ of f.

$$e(g^f(s), g^\alpha) = e((g^{s^d})^{c_d}(g^{s^{d-1}})^{c_{d-1}} \ldots (g^{s^0})^{c_0}, g^\alpha) = e(g, g)^{(c_d s^d + c_{d-1} \ldots s^0 c_0)\alpha}$$
$$e(g^{\alpha f(s)}, g) = e((g^{\alpha s^d})^{c_d}(g^{\alpha s^{d-1}})^{c_{d-1}} \ldots (g^{\alpha s^0})^{c_0}, g) = e(g, g)^{(c_d s^d + c_{d-1} \ldots s^0 c_0)\alpha}$$

The above relies on what is known as the Knowledge-of-Exponent Assumption (KEA) described in [7] which relies on the discrete logarithm problem[20] being really hard to solve for prime cyclic groups, which we have chosen.

Now that we have seen how a verifier can use a shifting technique to verify polynomial evaluation using KEA, we need to make sure the polynomials being evaluated are the ones that are represented in the QAP. The three cases the verifier has to check the prover on is

1. The polynomials in the sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$ in the QAP remain consistent.
2. The variables are not changed during the evaluation of the QAP mid-process.
3. The evaluation of the QAP is valid.

To obtain the needed shifts, we will implement the shift with the generator g ahead of time with random variables ρ_1, ρ_2 defining $g_A = g^{\rho_1}$, $g_B = g^{\rho_2}$, $g_C = g^{\rho_1\rho_2}$ which helps with verification efficiency as described in [4].

First, we will replace the single α shift with random variables $\alpha_1, \alpha_2, \alpha_3$ in our polynomials since rather than proving the evaluation on one polynomial, we want to prove the evaluation of each $l_k(s), r_k(s), o_k(s)$ from $\mathcal{A}, \mathcal{B}, \mathcal{C}$ respectively. We can encrypt them as

$$\{g_A^{l_k(s)}\}_{k\in\{0,\ldots n\}} \qquad \{g_B^{r_k(s)}\}_{k\in\{0,\ldots n\}} \qquad \{g_c^{o_k(s)}\}_{k\in\{0,\ldots n\}}$$

[19] A $g \in \mathbb{G}$ is called generator if $\mathbb{G} = \{\alpha g : \alpha \in \mathbb{F}_p\}$ Since α is an integer, αg is well defined.

[20] The discrete logarithm problem is if $g^x = h$, is there a computationally feasible way to find what x was given a generator g and result in the group generated by g denoted h

and enforce the check to validate these polynomials with

$$e(g_A^{l_k(s)}, g_A^{\alpha_1}) = e(g_A, g_A)^{\alpha_1 l_k(s)} \qquad e(g_A^{\alpha_1 l_k(s)}, g_A) = e(g_A, g_A)^{\alpha_1 l_k(s)}$$
$$e(g_B^{r_k(s)}, g_B^{\alpha_2}) = e(g_B, g_B)^{\alpha_2 r_k(s)} \qquad e(g_B^{\alpha_2 r_k(s)}, g_B) = e(g_B, g_B)^{\alpha_2 r_k(s)}$$
$$e(g_C^{o_k(s)}, g_C^{\alpha_2}) = e(g_C, g_C)^{\alpha_3 o_k(s)} \qquad e(g_C^{\alpha_3 r_k(s)}, g_C) = e(g_C, g_C)^{\alpha_3 o_k(s)}$$

With these shifts, the polynomials given in the QAP must remain in the same order in the QAP when multiplied by the witness vector $\vec{z}$. For example, the prover cannot replace $B(x)$ with $A(x)$ resulting in the proof of $A(x) * A(x) - C(x)$ rather than the original $Q_e(x)$ equation.

Now, we have to utilize a shifting approach along with a newly defined polynomial $Z(x) = \sum_{i=1}^{n} \beta(\rho_1 l_i(x) + \rho_2 r_i(x) + \rho_3 o_i(x)$ known as a consistency polynomial where β is a random shift variable. This polynomial implies $A(x), B(x)$ and $C(x)$ as defined in the QAP must be linear combinations of the corresponding polynomials in the sets $\mathcal{A}, \mathcal{B}, \mathcal{C}$ while being in the . Notice that when we encrypt $Z(s)$ using g_A, g_B, g_C with their corresponding in polynomials

$$e(g_A^{A(s)} \cdot g_B^{B(s)} \cdot g_C^{C(s)}, g^{\beta}) = e(g, g)^{Z(s)} \qquad e(g^{Z(s)}, g) = e(g, g)^{Z(s)}$$

Currently, the β shift almost prevents the ρ shifts from being able to be solved. However if g^{β} is provided to the prover, (in this case it is) the prover can multiply the coefficients of $A(s), B(s)$, and $C(s)$ to falsify the KEA assumption. Thus we add yet another shift γ that prevents β from being seen. Then the verifier will provide $g^{\beta\gamma}$ and g^{γ} will now check

$$e(g_A^{A(s)} \cdot g_B^{B(s)} \cdot g_C^{C(s)}, g^{\beta}\gamma) = e(g, g)^{Z(s)} \qquad e(g^{Z(s)}, g^{\gamma}) = e(g, g)^{Z(s)\gamma}$$

Applying these shifts allows the verifier to check the prover is providing a valid QAP and following the construction of the zk-SNARK. While the verifier can now check the prover's work, we now need to implement zero-knowledge[21] property so the verifier doesn't know any of the polynomials of the QAP; instead, only knows shifts of the polynomials.

The prover can apply a clever shift using random variables $\delta_1, \delta_2, \delta_3 \in \mathbb{F}_p$ to $A(x)B(x) - C(x) = D(x)h(x)$ in the following way:

$$(A(x) + \delta_1 D(x)) \cdot (B(x) + \delta_2 D(x)) - (C(x) + \delta_3 D(x)) = D(x)(\delta_2 A(x) + \delta_1 B(x) + \delta_1\delta_2 D(x) - \delta_3 + H(x)$$
$$A(x)B(x) + \cancel{\delta_1 D(x)B(x)} + \cancel{\delta_2 D(x)A(x)} + \cancel{\delta_1\delta_2 (D(x))^2} - C(x) - \cancel{\delta_3 D(x)} = \cancel{\delta_2 D(x)A(x)} + \cancel{\delta_1 D(x)B(x)} + \cancel{\delta_1\delta_2 (D(x))^2} - \cancel{\delta_3 D(x)} + D(x)H(x)$$
$$A(x)B(x) - C(x) = D(x)H(x)$$

Notice the following allows the prover to provide to the verifier $A'(X)B'(x) - C'(X) = D(X)H'(X)$, where the new polynomials represent the shifts mentioned above. Notice also when we encrypt $A'(X)$ we obtain

$$g^{A(s)+\delta_1 D(s)} = g^{L(s)} + (g^{D(s)})^{\delta_1}$$

The case is similar for all the other encryption processes of $B'(x), C'(X)$ and $H'(x)$. Note for the prover to do this, the following encrypted values must be provided to support the KEA checks the verifier does:

$$g_A^{D(s)}, g_B^{D(s)}, g_C^{D(s)}, g_A^{\alpha_1 D(s)}, g_B^{\alpha_2 D(s)}, g_C^{\alpha_3 D(s)}, g_A^{\beta D(s)}, g_B^{\beta D(s)}, g_C^{\beta D(s)} \tag{3.2}$$

While the verifier can now verify the prover and the prover can keep their argument of knowledge a secret. We need to add one more characteristic to our encryption technique. Notice that for $A(x)$ and a polynomial $l_i \in \mathcal{A}$ from our QAP, we have that

$$g^{A(s)} \cdot g^{l_i(s)} = g^{A(s)+l_i(s)}$$

holds in our encrypted space. Therefore we can separate the $\vec{z}$ variables of size n into $\vec{v}$ and $\vec{w} + \vec{a} = \vec{p}$ where $\vec{v}$ is the vector of public variables (including $z_1 = 1$ as the first entry of $\vec{v}$) of size m and the remaining variables $\vec{p}$ to be what the prover has to keep confidential from the verifier. This extension allows different public variables to be implemented, thus allowing the same problem with different initial variables to be evaluated. We also no longer need to provide verification checks on all polynomials in $\mathcal{A}, \mathcal{B}, \mathcal{C}$ but only the ones multiplied by a value in$\vec{p}$ during the construction of the QAP. Notice this also doesn't affect our statistical zero-knowledge property as

$$g_A^{A_p(x)+\delta_1 D(X)} \cdot g_A^{A_v(x)} = g_A^{A_p(x)+\delta_1 D(X)+A_v(x)}$$

where $A_p(X)$ denotes the polynomial corresponding to the values $\vec{w} + \vec{a}$ and $A(x)_v$ denotes the polynomial corresponding to the public values in the QAP. One can think of this approach as splitting our QAP into parts we can compute ahead of time before a verifier checks the proof.

[21]The method that will be used provides a statistical zero-knowledge property and is defined in theorem 13 of [2]

3.6 Writing a zk-SNARK system

We now have everything needed to develop a generalized zk-SNARK protocol. We will give a recap of the construction of a zk-SNARK. Suppose we have a prover and verifier that wants to prove they know a solution to an instance of a problem S. The verifier needs to answer the decision problem "Does the prover know a valid answer to the instance of S?" First, both the prover and verifier enter a setup phase where they decide on the following

- A large prime p
- A cyclic group $\mathbb{G}$ order p with generators g respectively.
- An elliptic pairing function $e : \mathbb{G} \times \mathbb{G} \longrightarrow \mathbb{G}_2$ where $\mathbb{G}_2$ has order r.
- Encode the instance of S, flatten, and write it as a system of equations corresponding to an arithmetic circuit Φ.
- Convert the arithmetic circuit to a R1CS (Note, the prover does **not** input a witness vector $\vec{W}$ yet.
- Convert the R1CS to a QAP containing and agree on a target polynomial $D(x)$
  1. $\mathcal{A} = \{l_k(x) : k \in \{0, \ldots, n\}\}$
  2. $\mathcal{B} = \{r_k(x) : k \in \{0, \ldots, n\}\}$
  3. $\mathcal{C} = \{o_k(x) : k \in \{0, \ldots, n\}\}$

  such that $A(x), B(x), C(x)$ correspond to the result of $Q_e(x)$ and a target polynomial $D(x)$ of degree d (number of operations to check in Φ) and size $n + 1$
- Create the proving key

$$\left(\left\{ g^{s^j} \right\}_{j \in \{0,\ldots,d\}}, \left\{ g_A^{l_i(s)}, g_B^{r_i(s)}, g_C^{o_i(s)} \right\}_{i \in \{0,\ldots,n\}} \right.$$
$$\left\{ g_A^{\alpha_1 l_i(s)}, g_B^{\alpha_2 r_i(s)}, g_C^{\alpha_3 o_i(s)}, g_A^{\beta l_i(s)}, g_B^{\beta r_i(s)}, g_C^{\beta o_i(s)} \right\}_{i \in \{m+1,\ldots,n\}}$$
$$\left. \left\{ g_A^{D(s)}, g_B^{D(s)}, g_C^{D(s)}, g_A^{\alpha_1 D(s)}, g_B^{\alpha_2 D(s)}, g_C^{\alpha_3 D(s)}, g_A^{\beta D(s)}, g_B^{\beta D(s)}, g_C^{\beta D(s)} \right\} \right)$$

  using the sampled random $s, \rho_1, \rho_2, \alpha_1, \alpha_2, \alpha_3 \in \mathbb{F}_p$ from the verifier where $g_A = g^{\rho_1}, g_B = g^{\rho_2}$, and $g_C = g^{\rho_1 \rho_2}$.
- Create the verification key

$$\left(g, g_C^{D(x)}, \left\{ g_A^{l_i(s)}, g_B^{r_i(s)}, g_C^{o_i(s)} \right\}_{i \in \{0,\ldots,m\}}, g^{\alpha_1}, g^{\alpha_2}, g^{\alpha_3, g^{\gamma}, g^{\beta\gamma}} \right)$$

  using sampled random $\gamma \in \mathbb{F}_p$ from the verifier.
- Verifier must discard[22] the generated random variables.

Notice the current CRS, which has all of the parameters needed for the verifier to check the proof anytime after the prover has finished creating a proof of the satisfying QAP. Remember the purpose of using a QAP is because when evaluating at a point s, it is probabilistically infeasible for the prover to cheat, provided the appropriate shifts and KEA checks are present.

Once the setup phase is complete, the prover will then create their proof by doing the following:

- Given a claimed solution with input $\vec{w}$, the prover executes encoded instance of S and records all $n - m + 1$ private input variables and auxiliary variables d_i for $i \in m + 1, \ldots n$ in the $\vec{p}$. Note that $\vec{p}$ is the witness vector for the QAP.
- Simplify for $A(x), B(x), C(x)$ as given in the definition of QAP. For example $A(X) = l_0(x) + \sum_{i=1}^{n} a_i \cdot l_i(x)$.
- Sample random variables for shifts denoted $\delta_1, \delta_2, \delta_3 \in \mathbb{F}_{\text{।}}$.
- Solve for $H(x) = \frac{A(x)B(x) - C(x)}{D(x)} + \delta_2 A(x) + \delta_1 B(x) + \delta_1 \delta_2 D(x) - \delta_3$. Note that the degree of $H(x)$ is equivalent to the degree of the target polynomial, hence the prover can provide $H(s)$ as

$$\prod_{i=1}^{d} \left(g^{s^i} \right)^{h_i}$$

  where h_i represents the coefficients of $H(s)$.

[22] These random variables cannot be known by anyone otherwise the proof can be falsified, hence should be permanently destroyed

- Solve for $A_p(x), B_p(x)$, and $C_p(x)$ in the encrypted space; that is

$$g_A^{A_p(x)} = \left(g_A^{D(s)}\right)^{\delta_1} \cdot \prod_{i=m+1}^{n} \left(g_A^{\alpha_1 l_i(s)}\right)^{d_i}$$
$$g_B^{B_p(x)} = \left(g_B^{D(s)}\right)^{\delta_2} \cdot \prod_{i=m+1}^{n} \left(g_B^{\alpha_2 r_i(s)}\right)^{d_i}$$
$$g_C^{C_p(x)} = \left(g_C^{D(s)}\right)^{\delta_3} \cdot \prod_{i=m+1}^{n} \left(g_C^{\alpha_3 o_i(s)}\right)^{d_i}$$

- Provide α shifted pairs for KEA check; that is

$$g_A^{A'_p(x)} = \left(g_A^{\alpha_1 D(s)}\right)^{\delta_1} \cdot \prod_{i=m+1}^{n} \left(g_A^{l_i(s)}\right)^{d_i}$$
$$g_B^{B'_p(x)} = \left(g_B^{\alpha_2 D(s)}\right)^{\delta_2} \cdot \prod_{i=m+1}^{n} \left(g_B^{r_i(s)}\right)^{d_i}$$
$$g_C^{C'_p(x)} = \left(g_C^{\alpha_3 D(s)}\right)^{\delta_3} \cdot \prod_{i=m+1}^{n} \left(g_C^{o_i(s)}\right)^{d_i}$$

  where the apostrophe denotes the polynomials with the shifts.

- Define the consistency polynomial

$$g^{Z(s)} = \left(g_A^{\beta D(s)}\right)^{\delta_1} \left(g_B^{\beta D(s)}\right)^{\delta_2} \left(g_C^{\beta D(s)}\right)^{\delta_3} \cdot \prod_{i=m+1}^{n} \left(g_A^{\beta l_i(s)} g_B^{\beta r_i(s)} g_C^{\beta o_i(s)}\right)^{d_i}$$

- Post to the CRS the following proof

$$\left(g_A^{A_p(x)}, g_B^{B_p(x)}, g_C^{C_p(x)}, g^{H(s)} g_A^{A'_p(x)}, g_B^{B'_p(x)}, g_C^{C'_p(x)}, g^{Z(s)}\right)$$

At this point, all the information is present in the CRS for the verifier to verify the proof given by the prover. Since the verifier can do this at any time without having the interact with the prover (aside from the initial trust setup), we have the non-interactive property we desired. Additionally, we have the zero-knowledge property as the verifier won't (unless the prover didn't keep them private) know the δ shifts.

Lastly, the verifier will do the following to complete the zk-SNARK:

- Define $\vec{v}$ as the vector of public inputs in the vector $\vec{z}$ in the QAP where $\vec{v}$ is of size m.
- Extract the proof from the CRS

$$\left(g_A^{A_p(x)}, g_B^{B_p(x)}, g_C^{C_p(x)}, g^{H(s)} g_A^{A'_p(x)}, g_B^{B'_p(x)}, g_C^{C'_p(x)}, g^{Z(s)}\right)$$

- Solve for the public input variable polynomials $A_v(s), B_v(s), C_v(s)$ where

$$g_A^{A_v(s)} = g_A^{l_0(s)} \cdot \prod_{i=1}^{m} \left(g_A^{l_i(s)}\right)^{v_i}$$
$$g_B^{B_v(s)} = g_B^{l_0(s)} \cdot \prod_{i=1}^{m} \left(g_B^{r_i(s)}\right)^{v_i}$$
$$g_C^{C_v(s)} = g_C^{l_0(s)} \cdot \prod_{i=1}^{m} \left(g_C^{o_i(s)}\right)^{v_i}$$

  where v_i is the i^{th} public input in $\vec{v}$.

- Check the prover has the correct polynomials by verifying

$$e\left(g_A^{A_p(x)}, g^{\alpha_1}\right) = e\left(g_A^{A'_p(x)}, g\right)$$
$$e\left(g_B^{B_p(x)}, g^{\alpha_2}\right) = e\left(g_A^{B'_p(x)}, g\right)$$
$$e\left(g_C^{C_p(x)}, g^{\alpha_3}\right) = e\left(g_A^{A'_p(x)}, g\right)$$

- Check the prover keeps the values of auxiliary and private variables consistent by verifying

$$e\left(g_A^{A_p(x)} g_B^{B_p(x)} g_C^{C_p(x)}, g^{\beta\gamma}\right) = e\left(g^{Z(s)}, g^{\gamma}\right)$$

- Finally check the QAP provided holds by verifying the QAP $(A_p(s)+A_v(s))(B_p(s)+B_v(s)) = D(s)H(s)+(C_p(s)+C_v(s))$ encrypted as

$$e\left(g_A^{A_p(x)} g_A^{A_v(x)}, g_B^{B_p(x)} g_B^{B_v(x)}\right) = e\left(g_C^{D(s)}, g^{H(s)}\right)\left(g_C^{C_p(x)} g_C^{C_v(x)}\right)$$

If all verification steps pass, then the verifier knows the prover has the knowledge to answer the instance of the problem S.

4 Conclusion

While the protocol described in this paper does implement a zero-knowledge proof system, there have been even more variations and optimizations to make the execution of zk-SNARK even faster as seen in [1]. As we can see, zk-SNARKs are an excellent tool for verifying integrity and confidentiality. In fact, zk-SNARKs are often used for discerning the integrity of a computation outsourced by a third party that knows sensitive data that cannot be shared, even with the party desiring the computation. Other examples include anonymous authorization and identification for access without the authorizer having to know any sensitive information of the party desiring access.

Lastly, we described zk-SNARKs in great detail; however, we never presented any concrete/non-trivial examples. Currently, there is a personal repository[23] dedicated to describing variations of zk-SNARK and is planned to include nontrivial examples like a Constrained Traveling Salesman Problem. At the time of writing this paper though, there are a few more constraints I, the author, need to define before writing it.

[23] `https://github.com/CaliburnSlash/zk-SNARKS`

References

[1] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. 2019.

[2] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. Cryptology ePrint Archive, Paper 2012/215, 2012.

[3] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[4] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Paper 2013/279, 2013.

[5] Maksym Petkus. Why and how zk-snark works. 2019.

[6] R Rateman. The (finite field) fast fourier transform. 2000.

[7] Christian Reitweissner. zksnarks in a nutshell. 2016.

[8] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, third edition, 2013.

[9] Justin Thaler. *Proofs, Arguments, and Zero-Knowledge*. 2022.

5 Acknowledgements

Many thanks to Dr. Stephen Graves for allowing me to ask questions and gather my thoughts and Benjamin Burns for helping me proofread and assisting in converting my thoughts into comprehensive English words.