Radboud University Nijmegen

Faculty of science

zk-SNARKs

A literature study

Bachelor thesis Mathematics

Author:
Cleo Gerards
s1001586

Supervisor:
dr. W. Bosma

Second reader:
dr. S. A. Terwijn

July 2022

Abstract

In this literature study, we study the notion of zk-SNARKs. These are statistical zero-knowledge proofs that are complete, knowledge sound, and succinct. To define these terms, we also introduce the big O. To build a SNARK, we will need a private information retrieval scheme, a probabilistically checkable proof, and an extractable hash function. We also build a Merkle tree, so that we can compute a Merkle tree commitment. With these concepts, we are ready to build a SNARK. In the end, we know what it means for a cryptographic protocol to be a zk-SNARK and we know exactly how to build one.

Acknowledgements

Firstly, I would like to thank my thesis supervisor dr. Wieb Bosma, who has supported me and my work since last year. He introduced me to the concept of zk-SNARKs and presented me with the articles *zkSNARKs in a nutshell* [1] and *Halo Infinite: Proof-Carrying Data from Additive Polynomial Commitments* [2], so that I could get a better understanding of this concept. These articles were the starting point of this thesis. He also supported me during the process of writing this thesis, which I always appreciated.

Moreover, I would like to thank Anca Nitulescu. She wrote a very clear paper about zk-SNARKs [3], where for the first time, I really understood zk-SNARKs. This article was the core of my thesis and truly helped me further appreciate this topic.

Lastly, I would like to thank Nir Bitansky et al, who wrote an extensive article about zk-SNARKs [4]. This was the article that Anca Nitulescu considerably used and it was also very helpful for me. I owe a very large part of this thesis to their research.

Contents

1 Introduction

1.1 Outline

In this literature study, we explain the concept of zk-SNARKs, which is a concept in cryptography and uses the concept of zero-knowledge proofs. This means we have notions of mathematics, but also some of computer science. Since zk-SNARKs are a fairly new subject manner, there is still little mathematical writing about this subject. In fact, the idea behind zero-knowledge proofs was only introduced in the 1980s and the abbreviation zk-SNARK was just introduced by Bitansky et al in 2012. The goal of this thesis is thus to translate the notions of computer science, to a more mathematical one. Therefore, this thesis is set up so that a common mathematics student without extra knowledge of computer science will understand everything that is written.

Since zk-SNARKs can be a hard concept to grasp, we will explain everything in layers. We will begin small and fill in more and more details as the thesis processes. We will first look at the idea behind zk-SNARKs in Section 2 and remain informal. In Section 4 we will fill in the details of the definition of zk-SNARKs and we will define them formally. After we exactly know what it means for a cryptographic scheme to be a zk-SNARK, we will see an example of a SNARK in Section 5. Note that we left out the 'zk'-part here. As this is again a hard concept, we will also begin small and fill in the details one by one.
Moreover, in the last section, we will find an appendix. Here, all the minor definitions that were needed during this thesis are written down.
In the end, we will have a full understanding of zk-SNARKs; we know the exact systems and requirements behind them and we will know how we can set up a SNARK ourselves.

1.2 Situation

zk-SNARKs are a form of non-interactive zero-knowledge proofs, where the zk in zk-SNARK refers to the zero-knowledge part. A zero-knowledge proof is a protocol where a prover wants to prove to the verifier that his statement is true. The zero-knowledge part refers to the notion that no information will leak, except for whether the statement of the prover is true or not.
Whereas the first zero-knowledge proofs depended on some interaction between the verifier and the prover, a zk-SNARK is non-interactive. This means that explicit communication between the prover and the verifier is not allowed.

In our study, we will look at NP-problems, so at statements that can be verified in polynomial time. In Section 3.1 we will learn what polynomial time means, but for now, assume that the verifier can verify the statement in a relatively short amount of time. Moreover, a function is called a problem if it will only output a 0 or a 1. In our case, if the verify finds that a statement is true, it will output a 1, and else a 0.
It is also known that we can build a SNARK for every NP-problem. For example, see [5]. Outside of the zero-knowledge part, this succinctness is a remarkable part of zk-SNARKs. With succinctness, we mean the notion that a verifier can verify the statement in a relatively short amount of time, even if the making of the proof took a long time to do.

In some literature, you will also find zk-SNARGs or zk-STARKs. These are very similar concepts, but there is a slight difference. In [3] we can find the definition of a zk-SNARG where the requirement of knowledge soundness is just replaced by *computational soundness*. We will explain the meaning of knowledge soundness in Section 4.

As we will later see, zk-SNARKs require a trusted set-up phase, which we will call the generation of a common reference string. This string ensures us that indeed a trusted set-up has taken place. zk-STARKs do not require this phase, they use publicly verifiable randomness instead.

Of course, there are also practical applications of zk-SNARKs. We will not dive into these applications, but it is good to know that zk-SNARKs are for example used in cryptocurrency, for instance, Zcash and Monero use the notion of zk-SNARKs.

2 Idea behind zk-SNARKs

We begin by looking at the idea behind zk-SNARKs.
The situation is as follows: we have two people, Alice and Bob, and Bob wants to prove to Alice that a statement is true, without revealing any information about the proof. In other words, he wants to prove to Alice that he has certain knowledge. Alice has to verify that the proof of Bob is valid. Due to their respective roles in this process, Alice is often called the verifier and Bob is often called the prover. In this process, they will use three algorithms: the generation of a common reference string, the proof algorithm, and the verify algorithm. We will also have one additional algorithm called the simulator.

2.1 Algorithms

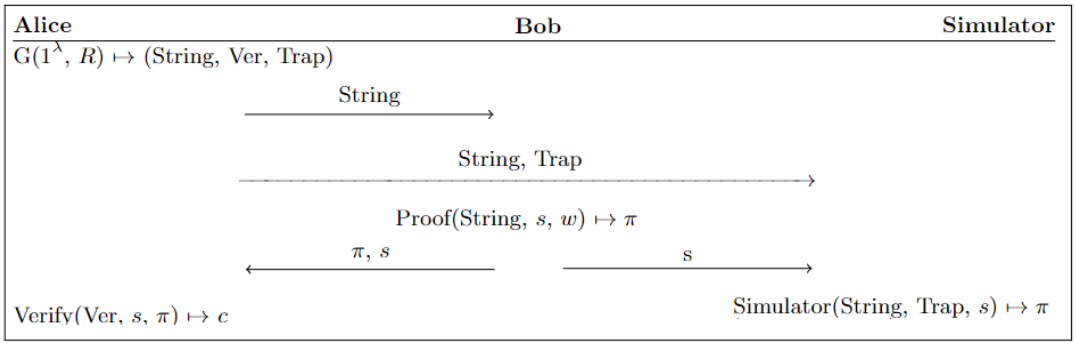
Generation common reference string: Before the whole process begins, Alice has to generate a common reference string, which is a random string. To do this, she needs a decidable binary relation R (9.1) and a security parameter $\lambda \in \mathbb{N}$, so λ determines how difficult it is to break the process for an outsider. She inputs 1^λ, which is the concatenation of λ times 1. She gets a common reference string, a verification status, and a trapdoor (9.2) as output. She sends this common reference string to Bob, which he needs for his proof algorithm. Later on, she needs the verification status, which she does not send to anyone. The trapdoor is later used in the simulator algorithm.
This part was just the set-up phase for the whole process. The generation of the common reference string is done so that both Alice and Bob can assume that a trusted set-up has taken place. Moreover, at this stage, Alice does not know what Bob wants to prove, but she has generated everything so that Bob can prove some statement and so that Alice can check this proof.

Proof algorithm: Bob takes as input the common reference string, generated by Alice, a statement, and a witness (9.3). He chooses this statement and its witness such that the pair is in the relation R. His algorithm will then output some proof. He sends an encrypted version of this proof, together with the statement, back to Alice.
He chooses this statement himself, in this stage of the process. The statement, and therefore its witness, was not known beforehand. After he has sent his statement and his proof to Alice, he can not change them. Furthermore, when we will later build a SNARK in Section 5.2, Bob will use Merkle trees to encrypt his proof. He could encrypt his proof in other ways, but we will only study this case.

Verify algorithm: Alice uses her generated verification status, together with the encrypted proof of Bob, to check whether the statement of Bob is true or false. If she accepts the proof, the verify algorithm will output a 1 and otherwise it will output a 0.

Simulator algorithm: The simulator is used by outsiders, so people outside of Alice and Bob. It uses the trapdoor and the common reference string, created by Alice, and the statement, created by Bob, to also output a valid proof. People who use the simulator to interact with Alice will likewise get an accepted proof if Bob's proof is accepted, and the proof will not be accepted if Bob's proof is not accepted. And so, these outsiders can learn what Bob learns from Alice; they learn if the statement of Bob is true or false.

In the end, we have the following scheme:



Here, G is the algorithm of Alice to generate a common reference string, String is the common reference string itself, Ver is the verification status, and Trap is the trapdoor. Furthermore, Proof is the proof algorithm of Bob, s is his statement, w is the witness of s, and π is his proof. As said before, Bob has to choose s and w such that $(s,\ w) \in R$; w is not a valid witness for statement s if $(s,\ w) \notin R$.
Moreover, Verify is the verify algorithm of Alice, and c is defined as:

$$c = \begin{cases} 1 & \text{if Alice accepts } \pi \\ 0 & \text{otherwise} \end{cases}$$

Lastly, Simulator is the simulator algorithm, which also generates a proof π.

We will later see that Alice will verify the encryption of proof π step by step. She will request only parts of the encryption of π so that the verification process will take less time. This means that Bob can not change his proof over time; he has to commit to π and its encryption, and as a result to the divided parts of the encryption of π. However, he does not have to reveal the values of the divided parts beforehand. The values will only be revealed when Alice requests them. However, Alice will not request every value, so some values remain hidden.
The encryption will be done using a private information retrieval scheme, which will be explained in Section 6.1.

2.2 Informal requirements

In the case of zk-SNARKs, there are several requirements for the algorithms mentioned above. These requirements are informally implemented in the name zk-SNARK since it is an abbreviation. zk-SNARK stands for:

- **Zero-knowledge:** No information leaks in the process
- **Succinct:** Proofs are short, it is easy to verify that a given argument proves the statement
- **Non-interactive:** There is no back-and-forth communication between Alice and Bob
- **Arguments:** Alice is only protected against adversaries without a huge amount of computer power

- **of Knowledge:** Bob has actual proof of his arguments, he is the one with knowledge

We will later see how these requirements are defined formally, and how this is implemented in the scheme from Section (2.1).

3 Functions

To define the requirements for zk-SNARKs formally, we first need to know what it means for a function to be *negligible*, where we will use the so-called big O notation.

3.1 Big O

To measure the runtime of an algorithm or to compare the runtimes of multiple algorithms, we use the big O notation. The runtime of an algorithm is sometimes also called the execution time of an algorithm.
Let f and g be two functions. Informally, $f \in O(g)$, if f does not asymptotically grow much faster than g. This is sometimes also denoted as $f = O(g)$. Below, $O(g)$ is defined formally.

Definition 3.1. Let $g : \mathbb{N} \to \mathbb{R}^+$. Then $\boldsymbol{\mathcal{O}(g)} = \{f : \mathbb{N} \to \mathbb{R}^+ | \exists\, c \in \mathbb{R}^+$ and $\exists\, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$

For example, $3(n)(n+4) \in \mathcal{O}(n^2)$. Indeed, take $c = 6$ and $n_0 = 4$, then $\forall n \geq n_0 = 4 :\ 3(n)(n+4) = 3n^2 + 12n \leq c \cdot n^2 = 6n^2$.

For a polynomial expression, we can often quickly see the asymptotic upper bound, as $a_d x^d + a_{d-1}x^{d-1} + ... + a_1 x + z_0 \in \mathcal{O}(x^d)$. We see that the coefficients a_d, ..., a_1, z_0 do not influence the asymptotic upper bound, and the same is true for the lower variables x^{d-1}, ..., x.

The most simple runtime is **constant time**, which is expressed as $f(n) = O(1)$. This simply means that the runtime is not dependent on the length of the input of the algorithm.

A common runtime is **exponential time**, which is expressed as $f(n) = 2^{O(n)}$. So, in other words: $\exists\, c \in \mathbb{R}^+$ and $\exists\, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 : f(n) \leq 2^{c \cdot n}$. Note that we can now also write: $f(n) \leq m^n$ for some $m > 0$. This is because $2^{c \cdot n} = \left(2^c\right)^n$ and if we set $m = 2^c$ we get our desired expression.
In conclusion, $f(n) = 2^{O(n)}$ is equivalent to $f(n) \in O(m^n)$, with $m > 0$.

A different common runtime is **polynomial time**, which is expressed as $f(n) = \text{poly}(n) = 2^{O(\log(n))}$. This simply means that $f(n)$ is bounded by some polynomial expression with variable n.
This is because $f(n) = \text{poly}(n)$ means that $\exists\ c \in \mathbb{R}^+$ and $\exists\ n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 : f(n) \leq 2^{c \cdot \log(n)}$. Here, we can use any logarithmic base greater than 1.
Note that we can write n as $2^{\log_2(n)}$. And so, $n^c = \left(2^{\log_2(n)}\right)^c = 2^{c \cdot \log_2(n)}$. Substituting this in our equation, we get that $f(n) = \text{poly}(n)$ means that $\exists\, c \in \mathbb{R}^+$ and $\exists\, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 : f(n) \leq n^c$.

Another common runtime is **polylogarithmic time**, which is expressed as $f(n) = \text{poly}(\log(n))$. This is sometimes shortened as $f(n) = \text{polylog}(n)$.
Using our knowledge of polynomial time, we see that $f(n) = \text{polylog}(n)$ means that $\exists\, c \in \mathbb{R}^+$ and $\exists\, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 : f(n) \leq (\log(n))^c = \log^c(n)$.

As we see, the big O defines an asymptotic upper bound, but we can also define an asymptotic lower bound. This is known as the big Ω.

Definition 3.2. Let $g : \mathbb{N} \to \mathbb{R}^+$. Then $\boldsymbol{\Omega(g)} = \{f : \mathbb{N} \to \mathbb{R}^+ | \exists\, c \in \mathbb{R}^+$ and $\exists\, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 : c \cdot g(n) \leq f(n)\}$.

3.2 Negligible functions

Now that we are familiar with the big O notation, we can define what it means for a function to be negligible.

Definition 3.3. A function $f : \mathbb{N} \to \mathbb{R}^+$ is **negligible** in $\lambda \in \mathbb{N}$, if $\exists\ c \in \mathbb{N}$ such that $f(\lambda) \in \mathcal{O}(\lambda^{-c})$, where λ is the security parameter.
This is denoted by $f = \text{negl}(\lambda)$.
Furthermore, a function f is **overwhelming** in $\lambda \in \mathbb{N}$ if $1 - f(\lambda)$ is negligible, so if $\exists\ c \in \mathbb{N}$ such that $(1 - f(\lambda)) \in \mathcal{O}(\lambda^{-c})$
This is denoted by $f = 1 - \text{negl}(\lambda)$

If we combine this with our knowledge of the big O, we get the following:
A function f: $\mathbb{N} \to \mathbb{R}^+$ is negligible if $\forall c \in \mathbb{N}\ \exists\ N \in \mathbb{Z}$ such that $\forall x > N$: $|f(x)| < \frac{1}{x^c}$

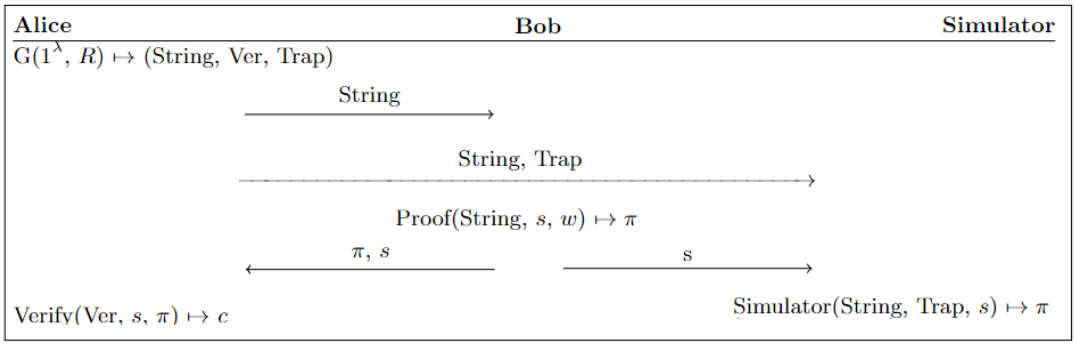
4 Details of zk-SNARKs

4.1 Formal requirements

To define a zk-SNARK on the system outlined in Section 2.1, there are some requirements. These requirements are:

- Completeness: A verifier will accept a prover's proof of a statement overwhelmingly when the prover has a valid witness.
- Knowledge soundness: When an argument is valid, there is an extractor (9.4) that can compute a witness successfully.
- Succinctness: The verification algorithm and the proof algorithm only need polynomial time.
- Statistical Zero-knowledge: Only whether a statement is true or false will leak in the process.

We will now define these requirements formally. We have the same situation as in Section 2. We have a generation algorithm G, a verification algorithm, and a proof algorithm. This means that we still have the same scheme as before:



The requirements are formally defined as follows:

Completeness: $\forall\lambda \in \mathbb{N},\ \forall R$ and $\forall(s,\ w) \in R$:

$$\mathbb{P}[\text{Verify}(\text{Ver},\ s,\ \pi) = 1 \text{ and } (s,\ w) \in R \mid G(1^\lambda,\ R) \mapsto (\text{String, Ver, Trap}) \text{ and } \text{Proof}(\text{String},\ s,\ w) \mapsto \pi] = 1 - \text{negl}(\lambda)$$

As we can see, in this requirement we can just follow the scheme; the generator algorithm of Alice needs a security parameter λ and relation R and outputs a common reference string String, a verification status Ver and a trapdoor Trap, however, this trapdoor is not further used in this requirement. Now, Bob uses the common reference string, and his own statement s and its witness w to make a proof π, where $(s,\ w) \in R$. In other words, w is a valid witness for s. Since $(s,\ w) \in R$, we want Alice to accept the statement and the proof.
So, the requirement is that Alice overwhelmingly accepts the statement s with its proof π, using her verification status, when the above circumstances are present.

Knowledge soundness: $\forall A$, an adversary (9.5), $\exists\ \epsilon_A$, an extractor, where A and ϵ_A are both probabilistic Turing machines (9.6), we have:

$$\mathbb{P}[\text{Verify}(\text{Ver},\ s,\ \pi) = 1 \text{ and } (s,\ w) \notin R \mid G(1^\lambda,\ R) \mapsto (\text{String, Ver, Trap}) \text{ and } A\|\epsilon_A(\text{String}) \mapsto ((s,\ \pi),\ w))] = \text{negl}(\lambda)$$

Again, we follow the scheme. Alice uses the security parameter λ and the relation R to generate the common reference string String, the verification status Ver and the trapdoor Trap. Now, there is an adversary that makes a valid proof π for a statement s, so Alice accepts this proof. Moreover, we have an extractor that will compute a witness w for this statement s. Since proof π for the statement s will be accepted, we require that this witness is valid. Or in other words, the probability that $(s,\ w) \notin R$ is negligible. Summarised, this means that the probability that an adversary has a valid proof for a statement, but the statement and its witness are not in the language R, is negligible. Following the above description, we can also write:

$$\mathbb{P}[\text{Verify(Ver, } s,\ \pi) = 1 \text{ and } (s,\ w) \in R \mid G(1^\lambda,\ R) \mapsto \\ \text{(String, Ver, Trap) and } A\|\epsilon_A(\text{String}) \mapsto ((s,\ \pi),\ w))] = 1 - \text{negl}(\lambda)$$

Succinctness: Verify runs in $\text{poly}(\lambda + |s|)$ time and Proof runs in $\text{poly}(\lambda)$ time.

Statistical Zero-knowledge: $\forall\lambda \in \mathbb{N},\ \forall R,\ \forall(s,\ w) \in R$ and $\forall A$, where A is again an adversary that is a probabilistic Turing machine, the following distributions are statistically close (9.7):

$$\text{F}_1 = [\text{Proof(String, } s,\ w) \mapsto \pi_1 \mid G(1^\lambda,\ R) \mapsto \text{(String, Ver, Trap)}] \\ \text{F}_2 = [\text{Proof(String, Trap, } s) \mapsto \pi_2 \mid G(1^\lambda,\ R) \mapsto \text{(String, Ver, Trap)}]$$

So, once more Alice uses the security parameter λ and the relation R to generate a common reference string String, a verification status Ver and a trapdoor Trap. In F_1, Bob makes a proof π_1 for his statement s and its witness w. In F_2 the simulator makes a proof π_2 for the same statement s, but now using the trapdoor. In this proof algorithm, the trapdoor is used as a statement and s is used as the witness of the trapdoor.
Now, we have statistical zero-knowledge if the distribution F_1, so the distribution where Bob makes a proof, is statistically close to the distribution F_2, so the distribution where the simulator makes a proof. Looking at the definition of statistically close, this means that the variation distance of these two distributions is negligible. So, the difference between the values of the probabilities of F_1 and F_2 is negligible, for every event in the distributions.
And so, only whether a statement is true or false can leak in this process and no other information will be leaked.

5 Building a SNARK

These algorithms are still quite vague, so we will now look at a more detailed description of our four main algorithms. We will show an implementation of these algorithms that uses Merkle trees. These Merkle trees will allow us to compute a succinct commitment for π so that the verification and the proof algorithms will be succinct. We will ignore the zero-knowledge part.
Note that this is just one example of how to build a SNARK, there are also other ways to build SNARKs. For example, Anca Nitulescu used lattices to build a SNARK, which is based on *Learning With Errors*, see [3].

5.1 Hashes

In computer science, hashes are often used to encrypt data. The definition of a hash is the following:

Definition 5.1. A **cryptographic hash function** h takes an input M and returns a hash value $h(M)$ of a fixed length n.

Usually, this input M exists of bits. In this case for h we have: h: $\{0,1\}^* \to \{0,1\}^n$. Here, $\{0,1\}^*$ means that the input is of arbitrary length, and the input only consists of strings of the numbers 0 and 1, $\{0,1\}^n$ means the output is of length n, and the output only consists of the numbers 0 and 1.
Simply put, we can see a hash function as a function that will encrypt its input to an output with a fixed length.

For a hash function to be safe, we often demand the hash to be collision-resistant. Informally, this means that it is hard to find multiple inputs where the output of the hash will be the same. This is formally defined as:

Definition 5.2. A hash function h is **collision-resistant** if for all A, an algorithm of polynomial size, and for all $k \in \mathbb{N}$ we have:

$$\mathbb{P}[x \neq y \text{ and } h(x) = h(y) | A(h) \mapsto (x, y)] \leq \text{negl}(k)$$

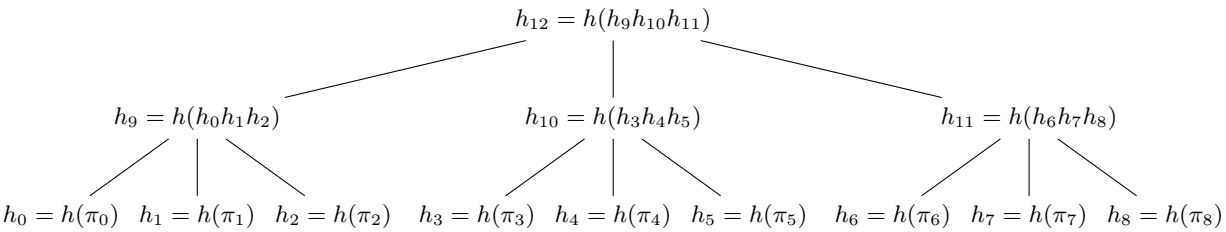
Note that we need an algorithm of polynomial size. This means that the size of A grows as the input h becomes bigger, but it can not asymptotically grow faster than some polynomial expression with variable h.

5.2 Merkle Trees

To build a Merkle tree we will need a proof π, this is the proof we want to encrypt. For the encryption, we use a collision-resistant hash h: $\{0,1\}^{g(n)} \to \{0,1\}^n$, with $n \in \mathbb{N}$. Note that this means that g is a function in n. We will divide π into $\frac{|\pi|}{g(n)}$ pieces, here $|\pi|$ is the size of π. This means that every piece is of length $g(n)$, as the sum of the length of the pieces should be equal to the length of π. Because, if l is the length of a piece, we get: $\frac{|\pi|}{g(n)} \cdot l = |\pi|$, and so $l = |\pi| \setminus (\frac{|\pi|}{g(n)}) = g(n)$.
Since we have pieces of length $g(n)$, we can apply our hash function h on each of the pieces, since h takes inputs of length $g(n)$. This will give us the encryption of the pieces, where the encrypted pieces are of length n. Now, we take some encrypted pieces together, by just concatenating the hash values such that we get a string of length $g(n)$. We again apply our hash function h and so forth. We iterate this until we reach one single string of length n.
This will give us a tree of encrypted pieces, which is called the Merkle tree of π.

If we need to, we will pad with zeroes. For example, if we want to split 111 into two parts, we get 11 and 10, where we used an extra 0 at the end so that we still have two strings of the same length.

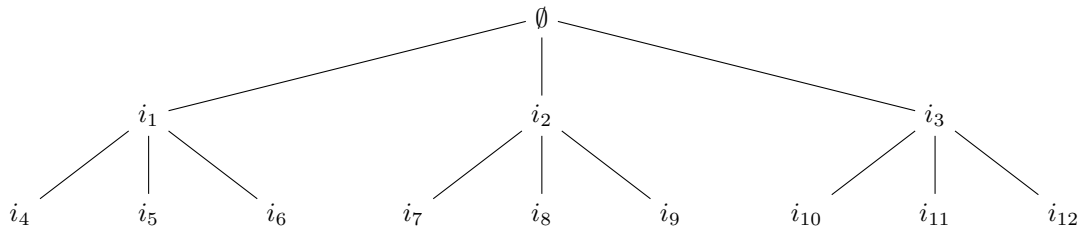
As an example, we split π into 9 pieces and we take $g(n) = 3$. We then get the following picture:



Here, π_0 are the first $g(n)$ bits of π, π_1 are the second $g(n)$ bits of π, et cetera. We apply the hash on these pieces and get the hash values $h_0,\ h_1,\ ...,\ h_8$. Since $g(n) = 3$, we take the first three encrypted pieces together and apply our hash again, resulting in $h_9 = h(h_0h_1h_2)$. We do the same for the middle three and last three encrypted pieces. Now we get the hash values $h_9,\ h_{10}$ and h_{11}, we combine these to get $h_{12} = h(h_9h_{10}h_{11})$. Since we end up with a single string (of length n), we end the process.

Now, if $|\pi| = (\frac{g(n)}{n})^{d+1}$, for some $d \in \mathbb{N}$, we will get a tree of depth d.
Often, $g(n) = 2n$. In that case, we will get a binary tree, since every vertex in the three is of length n and we take a hash over a concatenation of length $g(n) = 2n$, so we should concatenate 2 vertices each time. As a result, in the case of $g(n) = 2n$, the depth of the Merkle tree is $d = \log_2(\pi)$.

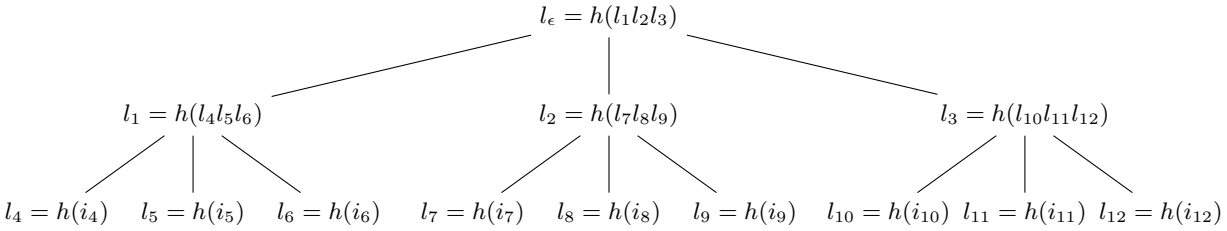
To build our SNARK, we use a specific labelling strategy.
We associate a vertex in the tree with the path from the root to this vertex. So, for a vertex of distance j from the root we have: $i = i_1...i_j$, where i_l is the l-th vertex in the tree and the root is labelled with the empty string $\emptyset$. We will label the tree as below, but we can choose whatever labelling system we like.



Let $d \in \mathbb{N}$ and let $k \in \mathbb{N}$. Then, for a Merkle tree of depth d and a hash function h: $\{0,1\}^{k^2} \to \{0,1\}^k$, we label the vertices as follows:

- Let $i = i_1...i_d$ be a leaf of the tree, so this leaf is the i-th k-bit block of proof π. We say a block is of k-bit if the block consists of k bits. We hash the value of this k-bit block and denote it by l_i.
- Let $i = i_1...i_n$ be a vertex that is not a leaf. We will label it by $h(i = l_{i1}...i_{ik})$, so as the hash of the concatenation of the values of its k children.
- The root is labelled by $h(l_1l_2...l_k)$, so again as the hash of the values of its k children. We also denote this as l_ϵ and it is called the root hash, top hash, or master hash.

Following this system, we get the following labelling:



Now that all the vertices are labelled, we define a Merkle tree commitment as the pair $(d,\ l_\epsilon)$. We also define an opening open_i as the set of all the labels of all the vertices and their siblings of the path from the root to the vertex i. So, in the example above, $\text{open}_5 = \{l_\epsilon,\ l_1,\ l_2,\ l_3,\ l_4,\ l_5,\ l_6\}$.

5.3 Our algorithms

We will now show how to build a SNARK, where more details like specific parameters and requirements will be filled in later. We have the same four algorithms as before, but we will leave out the simulator since the simulator can copy the algorithms of Bob.

Generation common reference string: As before, Alice has an algorithm G. In our example, this algorithm consists of two main steps:

1. the verification status
2. the common reference string, linked to the verification status

For the first step, Alice first generates a vector $r = (r_1, r_2, ..., r_q)$, consisting of strings of coins needed for the verification process, where a coin is a random bit. We can see the random generation as a coin toss, so for example if the coin lands on heads, $r_{il} = 1$, and if the coin lands on tails, $r_{il} = 0$, for an $i \in [q] = \{1,\ 2,\ ...,\ q\} = \{k \in \mathbb{N} \mid k \leq q\}$, $l \in \mathbb{N}$, and r_{il} is the l-th bit in the string r_i.
Then, she again generates a string of coins K, where K will be later used for encryption. In this section, we will not see how this K is implemented, as this is a detail we will fill in in Section 7.1. Lastly, she generates a hash function h.
Now, the verification status is set as the triple $\text{Ver} := (h,\ r,\ K)$.

For the second step, Alice first computes $r^{(j)} = (r_1^{(j)},\ r_2^{(j)}, ..., r_q^{(j)})$, where $j \in \mathbb{N}$ and $\forall i \in [q]$: $\text{r}_i^{(j)}$ is defined as first j bits of r_i. She defines the set $r^* = \{r^{(j)} |\ j\ \in \mathbb{N}\}$. She then computes $C_{r^*} = \{C_{r^{(j)}} \mid C_{r^{(j)}}$ is an encryption of $r^{(j)},\ j \in \mathbb{N}\}$.
Now, the common reference string is set as the pair $\text{String} := (h,\ C_{r^*})$.

This concludes the generation of the verification status and the common reference string. In a small overview, we get the following:

1. Generate $r = (r_1, r_2, ..., r_q)$ for verifying, generate K for encrypting, and generate hash h.
   $\text{Ver} := (h,\ r,\ K)$
2. Compute $r^{(j)} = (r_1^{(j)}, r_2^{(j)}, ..., r_q^{(j)})$, set $r^* = \{r^{(j)} |\ j\ \in \mathbb{N}\}$, and compute $C_{r^*} = \{C_{r^{(j)}} \mid C_{r^{(j)}}$ is an encryption of $r^{(j)},\ j \in \mathbb{N}\}$.
   $\text{String} := (h,\ C_{r^*})$.

Proof algorithm: Now that Alice has generated a common reference string, Bob can make his proof. This is again done in two main steps:

1. Generating a simple proof π
2. Transforming this proof using a Merkle tree

For the first step, Bob chooses a statement s with witness w and makes a proof π for this statement. How he does this exactly, we will see in Section 7.2. He also computes a Merkle tree commitment $(d,\ l_\epsilon)$ for this proof.

For the second step, Bob computes a database and stores every opening for every point where the proof π is needed for the verification process. This collection of points is defined as the set open_a. He then encrypts open_a as C_{open_a}. In this process, he will use a private information retrieval scheme, which we will explain in Section 6.1.
The final proof of Bob is now set as the triple $\Pi = (d, l_\epsilon, C_{\text{open}_a})$.

In a small overview, we get the following:

1. Make proof π for statement s and witness w. Compute $(d,\ l_\epsilon)$
2. Compute $\text{open}_a = \{\text{Opening of } \pi\}$, compute encryption C_{open_a}.
   $\Pi := (d,\ l_\epsilon, C_{\text{open}_a})$

Verify algorithm: Now that Bob has generated his proof Π, Alice has to verify this proof. She first has to verify the specific parameters, which will be explained in Section 7.3. She also decrypts C_{open_a} and verifies its openings. To verify the openings, she checks if her verify algorithm accepts the values of points of π in the database. If the verify algorithm accepts these values, she also accepts the proof Π of Bob. Otherwise, she will reject his proof.

Again, in a small overview, we get the following:

1. Verify the parameters
2. Decrypt C_{open_a}
3. Verify the openings

This concludes the general idea of building a SNARK using Merkle Trees. We will now focus on filling in the details of this process.

6 Details building a SNARK

To meet the requirements for a SNARK in our construction, we use specific algorithms. We will explain these algorithms in this section, so we can fill in the details of Section 5.

6.1 Private information retrieval

For our construction, we first need a computational polylogarithmic private information retrieval scheme. This is a scheme where someone can access data from a database, without revealing what data is accessed.

A private information retrieval scheme is a triple of algorithms (PEnc, PEval, PDec). Here, PEnc is an encryption algorithm, PEval is an evaluation algorithm and PDec is a decryption algorithm.

Let DB be a database with 2^n data points, with $n \in \mathbb{N}$, and where each data point is an element of $\{0,1\}^j$, with $j \in \mathbb{N}$. Let $i \in \{0,1\}^n$ be a query (9.8) to database DB, and let λ be the security parameter. Then, we have the following:

- The encryption algorithm will encrypt i, using the security parameter λ and randomness W. So: $\text{PEnc}_W(1^\lambda,\ i) \mapsto\ C$, where C is the encryption of query i.
- The evaluation algorithm will generate a string x, where DB$[i]$ is in some way incorporated in x, so PEval(DB, C) $\mapsto\ x$.
  Moreover, PEval(DB, C) = poly(λ, 2^n, j).
- The decryption algorithm will decrypt string x so that DB$[i]$ will be the output. Again, randomness W is used. So, we get: $\text{PDec}_W(x) \mapsto \text{DB}[i]$.

For this triple to be a private information retrieval scheme, we have the following three requirements:

Correctness: If we encrypt some query i with our encryption algorithm, then use our evaluation algorithm, and lastly use our decryption algorithm, we will get the original query:

$$\mathbb{P}[\text{PDec}_W(x) = \text{DB}[i] \mid \text{PEnc}_W(1^\lambda, i) \mapsto\ C \text{ and PEval(DB, } C) \mapsto\ x] = 1.$$

Succinctness: $\text{PEnc}_W(1^\lambda,\ i) = \text{poly}(\lambda,\ n,\ j)$ and $\text{PDec}_\text{R}(x) = \text{poly}(\lambda,\ n,\ j)$. In addition, the size of C and the size of x are bounded.

Semantic security: The encryption is semantically secure for multiple queries.
This is defined as follows: for all algorithms A of polynomial size, for all security parameters $\lambda \in \mathbb{N}$, and for all pairs of queries $i = (i_1, ..., i_q) \in \{0,1\}^{\text{poly}(\lambda)}$, $\hat{i} = (\hat{i}_1, ..., \hat{i}_q) \in \{0,1\}^{\text{poly}(\lambda)}$, with $q \in \mathbb{N}$, we have:

$$\mathbb{P}[A(\text{PEnc}_W(1^\lambda,\ i)) = 1] - \mathbb{P}[A(\text{PEnc}_W(1^\lambda,\ \hat{i})) = 1] \leq \text{negl}(\lambda)$$

Since i is a vector, we have that $\text{PEnc}_W(1^\lambda,\ i) = (\text{PEnc}_{W_1}(1^\lambda,\ i_1),\ \ldots\ , \text{PEnc}_{W_q}(1^\lambda,\ i_q))$, and the same is true for $\hat{i}$.
In other words, only negligible information can be retrieved from the encryption of i and $\hat{i}$.

For our construction, we also required this private information retrieval scheme to be polylogarithmic, so this whole scheme should be done in polylog(n) time.
This concludes the definition of a private information retrieval scheme.

6.2 Probabilistically checkable proofs

Secondly, we need a probabilistically checkable proof, where a universal relation R_U is used.

Definition 6.1. A **universal relation** is defined as the set $R_U = \{(s,\ w)|\ w \text{ is a witness for statement } s\}$. Here, s is defined as $s = (M,\ x,\ t)$, with $|w| \leq t$ for every witness w, M is a Turing machine, and M will accept $(x,\ w)$ in t steps or less.

We can also define a subset $R_c \subseteq R_U$:

Definition 6.2. A **subset R_c** of R_U is defined as the set $R_c = \{(s,\ w)|\ w \text{ is a witness for statement } s, \text{ such that } t \leq |x|^c\}$.

Again, s is defined as $s = (M,\ x,\ t)$, just as in the definition of a universal relation.

A probabilistically checkable proof for a universal relation R_U is now defined as a triple of algorithms ($\mathrm{P_{pcp}}$, $\mathrm{V_{pcp}}$, $\mathrm{E_{pcp}}$). Here, $\mathrm{P_{pcp}}$ is the proof algorithm, $\mathrm{V_{pcp}}$ is the verify algorithm, and $\mathrm{E_{pcp}}$ is an extractor.

Let $(s,\ w) \in R_U$, with s a statement and w a witness for s. Then, we have the following:

- The proof algorithm will generate a proof π for input $(s,\ w)$, so: $P_{\mathrm{pcp}}(s,\ w) \mapsto \pi$. We have that $|\pi| = \mathrm{poly}(t)$, where $|\pi|$ is the size of π.
  Furthermore, $P_{\mathrm{pcp}}(s,\ w) = \mathrm{poly}(|s|,\ t)$.
- The verify algorithm will verify proof π for the pair $(s,\ w)$, where π is made by P_{pcp}. For this, the verify algorithm needs a pair $(s,\ r)$, where $r \in \{0,1\}^{O(\log(t))}$. So, we get that:
  $$V^{\pi}_{\mathrm{pcp}}(s,\ r) = \begin{cases} 1 & \text{if proof } \pi \text{ is accepted} \\ 0 & \text{otherwise} \end{cases}$$
  Here, we have $V^{\pi}_{\mathrm{pcp}}(s,\ r) = \mathrm{poly}(|s|)$, and $V^{\pi}_{\mathrm{pcp}}(s,\ r)$ uses $O(\log(t))$ random bits and $\mathrm{polylog}(t)$ locations in proof π.
- The extractor will compute a witness w for the pair $(s,\ \pi)$, such that $(s,\ w) \in R_U$. So, $E_{\mathrm{pcp}}(s,\ \pi) \mapsto w$. We also have $E_{\mathrm{pcp}}(s,\ \pi) = \mathrm{poly}(|s|,\ t)$.

For this triple to be a probabilistically checkable proof, we have the following two requirements:

Completeness: A valid proof for a statement with a valid witness will be accepted overwhelmingly:

$$\forall (s,\ w) \in R_U, \forall \lambda \in \mathbb{N}\text{: } \mathbb{P}[V^{\pi}_{\mathrm{pcp}}(\mathrm{s, r}) = 1] = 1 - \mathrm{negl}(\lambda)$$

Soundness: A pair $(\hat{s}, \hat{w}) \notin R_U$ will be accepted negligibly:

$$\text{If } \exists\ \epsilon > 0, \text{ such that } \forall \mathrm{s}\text{: } \mathbb{P}[V^{\pi}_{\mathrm{pcp}}(s,\ r) = 1] \geq 1 - \epsilon,$$

then the extractor will compute a witness w for the pair $(s,\ \pi)$, such that $(s,\ w) \in R_U$. So, $E_{\mathrm{pcp}}(s,\ \pi) \mapsto w$. We also have $E_{\mathrm{pcp}}(s,\ \pi) = \mathrm{poly}(|s|,\ t)$.

This concludes the definition of a probabilistically checkable proof.

6.3 Extractable collision-resistant hashes

Lastly, we need an extractable collision-resistant hash function. We have already seen the definition of a hash function and what it means for a hash function to be collision-resistant in Section 5.1, so now we only need to define what it means for a function to be extractable.

Informally, a function $f : X \to Y$ is extractable if $\forall x \in X$ such that we can compute $y = f(x)$, there is an extractor that can compute $x = f^{-1}(y)$.
More precisely, a function $f : X \to Y$ is extractable if for any adversary A that can produce a valid evaluation of f, meaning it can find the value $f(x) = y$, for a given x, there is an extractor ϵ_A that can produce a preimage of y, so ϵ_A can find an $\hat{x}$, such that $f(\hat{x}) = y$. Note that this ϵ_A did not know the original pre-image x.
This is formally defined as:

Definition 6.3. Let $H = \{H_k\}_{k\in\mathbb{N}}$ be a set of functions, with $\forall k \in \mathbb{N} : H_k : \{0,1\}^{l(k)} \to \{0,1\}^k$. Then, H is **extractable** if for all algorithms A of polynomial size, and for all polynomials m, there exists an extractor ϵ_A of polynomial size, such that for all security parameters $\lambda \in \mathbb{N}$, and for all $z \in \{0,1\}^{m(k)}$, we have:

$$\mathbb{P}[A(h,\ z) \mapsto y,\ \exists\ x \text{ such that } h(x) = y,\ \epsilon_A \mapsto \hat{x},\ h(\hat{x}) \neq y] \leq \text{negl}(\lambda)$$

Here, $h \in H = \{H_k\}_k$.

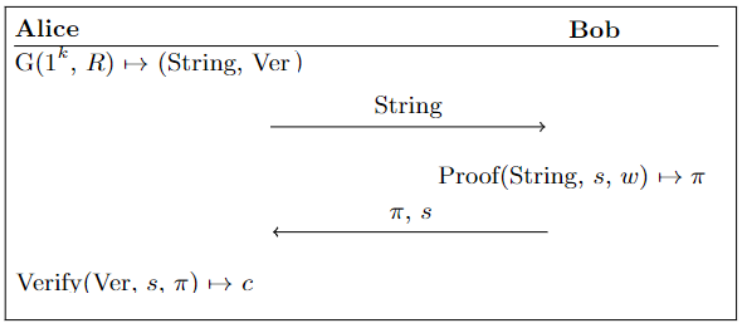
In other words, if we have $y = f(x)$, the probability that an extractor finds an $\hat{x}$ such that $y \neq f(\hat{x})$ is negligible, where z is any auxiliary input.

7 Back to building

Now that we know what a private information retrieval system, a probabilistically checkable proof, and an extractable collision-resistant hash function are, we can fill in the details of Section 5.3, where we began building our SNARK.

Let R_U be a universal relation, as defined in Section 6.2. For this universal relation, we have a probabilistically checkable proof. Remember that this is a triple of algorithms $(P_{\text{pcp}}, V_{\text{pcp}}, E_{\text{pcp}})$.
Let $(s,\ w) \in R_U$, where again $s = (M,\ x,\ t)$. We require that for all proofs π: $|\pi| \leq t^2$.
We also require that the verifier V_{pcp} uses $O(1)$ queries and $f(t) = O(\log(t))$ coins.
We also require our private information retrieval system (PEnc, PEval, PDec) to be succinct. Lastly, we need a set $\{H_k\}_{k\in\mathbb{N}}$ of extractable collision-resistant hash functions, where $\forall k \in \mathbb{N}:\ H_k\colon \{0,\ 1\}^{k^2} \to \{0,1\}^k$.

Remember our scheme from before, but leaving out the simulator part:



Note that we now have $k \in \mathbb{N}$ as our security parameter, instead of the $\lambda \in \mathbb{N}$ we used before.

7.1 Generation common reference string

As we can see, the first thing that happens in the scheme is the generation algorithm of Alice. This will be done in two steps:

1. The generation of the verification status
2. The generation of the common reference string

Generation verification status: Let $q \in \Omega(\log\ (k))$. Remember that $\Omega(\log(k))$ defines an asymptotic lower bound for q. Alice generates $r = (r_1,\ ...,\ r_q)$, where $r_1,\ ...,\ r_q$ are strings of coins and $\forall i \in [q]:\ r_i \in \{0,\ 1\}^{\log^2(k)}$. r will later be used in the verification algorithm V_{pcp}.
After, Alice also generates a string of coins $K \in \{0,\ 1\}^{\text{poly}(k)}$. K will later be used as the randomness when encrypting $q \cdot \log^2(k)$ queries in the private information retrieval scheme. She also randomly chooses an extractable collision-resistant hash function $h \in \{H_k\}_{k\in\mathbb{N}}$.
The verification status is now set as the triple $\text{Ver} := (h,\ r,\ K)$.

Generation common reference string: Let $j \in [\log^2(k)]$. Then $r^{(j)} = (r_1^{(j)},\ ..., r_q^{(j)})$, where r is the vector of strings of coins Alice just generated and $\forall i \in [q]:\ r_i^{(j)}$ are the first j bits of r_i. Alice computes the set $r^* = \{r^{(j)} \mid j \in [\log^2(k)]\}$.
Then, we need an encryption of $r^{(j)}$ using the private information retrieval scheme. So,

$\text{PEnc}_K(1^k,\ r^{(j)}) \mapsto C_{r^{(j)}}$. Alice then computes $C_{r^*} = \{C_{r^{(j)}} \mid j \in [\log^2(k)]\}$.
Lastly, the common reference string is set as the pair String $:= (h,\ C_{r^*})$.

So, in a small overview we get:

1. Generate strings of coins $r = (r_1,\ ...,\ r_q)$, for $q \in \Omega(\log(k)), r_i \in \{0,\ 1\}^{\log^2(k)}$. Generate a string of coins $K \in \{0,\ 1\}^{\text{poly}(k)}$ and a hash h.
   Ver $:= (h,\ r,\ K)$
2. Compute $(r_1^{(j)},\ ..., r_q^{(j)})$, for $j \in [\log^2(k)]$. Compute $r^* = \{r^{(j)} \mid j \in [log^2(k)]\}$ and $C_{r^*} = \{C_{r^{(j)}} \mid j \in [\log^2(k)]\}$, where $C_{r^{(j)}}$ is an encryption of r^j using the private information retrieval scheme.
   String $:= (h,\ C_{r^*})$

This concludes the first part of our scheme; the generation algorithm of Alice. The common reference string String is sent to Bob.

7.2 Proof algorithm

Now that Alice is finished with her generation algorithm, we move on to the proof algorithm of Bob.

Just as in Section 5.3, we do this in two steps:

1. Generating a proof π
2. Transforming this proof using a Merkle tree

Generating a proof π: First, Bob needs to choose his statement s and its witness w, for which $(s,\ w) \in R_c$. Recall that $R_c \subseteq R_U$ and that R_U is a universal relation. In Section 6.2 we specified what this means:
A **universal relation** is defined as the set $R_U = \{(s,\ w) |\ w \text{ is a witness for statement } s\}$. Here, s is defined as $s = (M,\ x,\ t)$, with $|w| \leq t$ for every witness w, M is a Turing machine, and M will accept $(x,\ w)$ in t steps or less.
And so, here we have that the statement s is defined as $s = (M,\ x,\ t)$. Since $R_c \subseteq R_U$, we also require that $t \leq |x|^c$. Moreover, we require that $f(t) = O(\log(t)) \leq \log^2(k)$.

Now Bob makes a probabilistically checkable proof for the pair $(s,\ w)$, so $\text{P}_{\text{pcp}}(s,\ w) \mapsto \pi$. The size of this proof π should be $|\pi| = k^{d+1}$. Remember that we also required that $|\pi| \leq t^2$, and so we should have that $k^{d+1} \leq t^2$.
Now that Bob has a proof π, he can build a Merkle tree for this proof. He builds a Merkle tree of depth d, using the extractable collision-resistant hash h that Alice also used, and computes the root hash l_ϵ. Now he has the Merkle tree commitment $(d,\ l_\epsilon)$ for his proof π.

Transforming Bob's proof using a Merkle tree: Firstly, Bob needs to compute a database DB. This database consists of $2^{f(t)} = 2^{O(\log(t))}$ data points, where for every data point $p^{f(t)}$ in the database we have $p^{(f)} \in \{0,\ 1\}^f$. Again, $p^{(f)}$ are the first f bits of p.
Later, the verification algorithm $\text{V}_{\text{pcp}}^{\pi}$ of Alice will query this database, so it will request information from the database. Naturally, Bob wants to make sure that this verification algorithm will accept his proof. And so, at every point that $\text{V}_{\text{pcp}}^{\pi}(s,\ p^{(f)})$ will request, he stores the opening $\text{open}_{p^{(f)}}$ for the proof π. Remember that the opening of a point is the set of vertices in the path from the root to this point in the Merkle tree of π and all the siblings of these vertices.

Using the encryption algorithm and the evaluation algorithm of the private information retrieval scheme, Bob will now compute the encryption of these openings. So, $\text{PEnc}_K(1^k,\ p^{(f)}) \mapsto C_{p^{(f)}}$ and $\text{PEval}(\text{DB},\ C_{p^{(f)}}) \mapsto C_{\text{open}_{p^{(f)}}}$.
The final proof of Bob is now set as the triple $\Pi := (d,\ l_\epsilon,\ C_{\text{open}_{p^{(f)}}})$.

So, in a small overview we get:

1. Choose $(s,\ w) \in R_c$. Make a probabilistically checkable proof π for $(s,\ w)$. Make Merkle tree of π of depth d and compute the root hash l_ϵ.
2. Compute database DB of 2^f points $p^{(f)}$. For every point that $\text{V}^{\pi}_{\text{pcp}}(s,\ p^{(f)})$ will request, store $\text{open}_{p^{(f)}}$. Encrypt $p^{(f)}$ as $C_{p^{(f)}}$ using the private information retrieval scheme and evaluate $C_{p^{(f)}}$ as $C_{\text{open}_{p^{(f)}}}$, also using the private information retrieval scheme.
   $\Pi := (d,\ l_\epsilon,\ C_{\text{open}_{p^{(f)}}})$

This concludes the second part of the scheme; the proof algorithm of Bob. Statement s and its proof Π are now sent to Alice.

7.3 Verify algorithm

Now that Bob has made his proof, we move on to the last part of our scheme; the verification algorithm of Alice. This is done in two small steps:

1. Verify the parameters
2. Decrypt the openings and verify them

Verify the parameters: Remember that we required that $t \leq |x|^c$, since $(s,\ w) \in R_c \subseteq R_U$, where R_U is a universal relation. Moreover, before the whole process began, we required that $|\pi| \leq t^2$. So first, Alice has to check that $k^{d+1} \leq t^2 \leq |x|^{2c}$. If this is not the case, she immediately rejects proof Π of Bob. On the other hand, if the parameters are correct, she moves on to the next step.

Decrypt the openings and verify them: Using the private information retrieval scheme, Alice decrypts $C_{\text{open}_{p^{(f)}}}$, so $\text{PDec}_K(C_{\text{open}_{p^{(f)}}}) \mapsto \text{open}_{p^{(f)}}$. Now she checks all openings $\text{open}_{p^{(f)}}$. So, for every point in the path from $p^{(f)}$ to the root in the Merkle tree, she has to check that the hash of the concatenation of $p^{(f)}$ and his siblings gives the value of the parent of $p^{(f)}$. If this is not the case, she rejects proof Π of Bob.
Alice also has to check the opened values. So, if $\pi\big|_{f^{(p)}}$ is the value in the location $f^{(p)}$, then she has to check if $\text{V}^{\pi\big|_{f^{(p)}}}_{\text{pcp}}(s,\ f^{(p)}) = 1$, because then $\text{V}^{\pi\big|_{f^{(p)}}}_{\text{pcp}}$ has accepted $\pi\big|_{f^{(p)}}$. If this is not the case, she also rejects proof Π of Bob.

So, in a small overview we get:

1. Verify that $k^{d+1} \leq t^2 \leq |x|^{2c}$, if this is not the case reject s and Π
2. Decrypt $C_{\text{open}_{p^{(f)}}}$, verify opened paths and check $\text{V}^{\pi\big|_{f^{(p)}}}_{\text{pcp}}$, if one or more are incorrect, reject s and Π
3. Accept s and Π

Now we are at the end of our scheme. If at this point Alice has not rejected proof Π she will accept Π and Bob has proven statement s.

8 Afterthoughts

I really enjoyed working on this thesis. Beforehand, I also enjoyed my Mathematics bachelor's, but I was especially interested in mathematics that could explain the theoretical side of computer science. Moreover, cryptography always intrigued me, so as you can imagine, I truly liked learning about the cryptographic protocol that is a zk-SNARK. I was very motivated to work on this thesis and I still am. I would love to learn more about cryptography and I would have loved to do more research on zk-SNARKs. Unfortunately, time was a constraint. I would have liked to maybe look at more ways to build a SNARK or to look at the applications, but unfortunately, I did not have the time.
I do, however, feel like I met the goal of this thesis. My goal was to be able to explain zk-SNARKs generally, but also in detail, to a fellow mathematics student. The process of translating computer science to mathematics was at times quite challenging. A lot of articles assumed prior knowledge and used terms that I as a mathematics student did not know. And so, during the writing of this thesis, I had to do a lot of searching and really had to think about what was written in my used sources. However, now that we are at the end, I feel like I can explain zk-SNARKs to my fellow students, as I am confident they will understand this thesis.

Overall, I am happy with the end result and I am curious as to what I will learn in my future academic career.

9 Appendix

In this appendix, we find all the minor definitions that were needed in this thesis.

9.1 Definitions in Section 2.1 (Algorithms)

In Section 2.1 we needed the definition of a binary relation, the definition of a trapdoor function, and the definition of a witness:

Definition 9.1. R is a **decidable binary relation** if it is

1. decidable: there is an efficient algorithm to see if a statement holds
2. a binary relation: it is a relation in the mathematical sense, and it is a relation between two elements

Moreover, an algorithm A is **efficient** if:

1. $A(n) = \text{poly}(n)$ and A consists of finite procedures
2. the output of A is always correct
3. anyone can do the steps of A themselves, and if they follow the steps of A strictly, the output will be correct

Definition 9.2. A function f is a **trapdoor function** when:

1. it is easy to compute $y = f(x)$
2. it is hard to compute $x = f^{-1}(y)$

For example, the multiplication of the factorisation of large prime numbers is a trapdoor function. It is easy to choose some prime numbers and to multiply them, but it is hard to factorise the product without knowing the factorisation beforehand.

Definition 9.3. A **witness** w of a statement s is some information, that makes the statement s easy to verify.

Let's for example have as a statement: '2349 is not a prime number'. Then a witness w could be that 3 is a factor of 2349. Now, it is easy to check that 2349 is not a prime number since you just have to compute $2349 \div 3 = 783$.

9.2 Definitions in Section 4.1 (Formal requirements)

In Section 4.1 we needed the definition of an extractor, the definition of an adversary, the definition of a probabilistic Turing machine, and the definition of statistically close.

Definition 9.4. An **extractor** is a function ϵ that can compute a witness for a given input x.

Definition 9.5. An **adversary** A is a (malicious) entity that tries to break the cryptographic system.

Definition 9.6. An algorithm, or function, A is a **probabilistic Turing machine** if:

1. A is a Turing machine
2. $A \in \text{poly}(\lambda)$

Where $\lambda \in \mathbb{R}$ is the security parameter.

We will not dive into Turing machines in this thesis. For now, it is good to know that they are some kind of computational device. If one wants to learn more about Turing machines, one can look in the Stanford Encyclopedia of Philosophy [8].

Definition 9.7. Let F_1 and F_2 be distributions over the same finite domain D. The **variation distance** over these distributions is defined as:

$$\Delta(F_1, F_2) = \tfrac{1}{2} \sum_{d \in D} |\mathbb{P}[F_1 = d] - \mathbb{P}[F_2 = d]|.$$

Now, two distribution families $\{F_{1_i}\}_{i \in \mathbb{N}}$ and $\{F_{2_i}\}_{i \in \mathbb{N}}$ are **statistically close** if

$$\Delta(F_{1_i}, F_{2_i}) = \text{negl}(i).$$

9.3 Definitions in Section 6.1 (Private information retrieval)

In Section 6.1 we only needed one definition, which was the definition of a query.

Definition 9.8. A **query** is an order given to a database, such that a specific action is taken. This action can retrieve information stored in the database if needed.

References

[1] C. Reitwießner, "zkSNARKs in a Nutshell," 2016. [Online]. Available: https://blog.ethereum.org/2016/12/05/zksnarks-in-a-nutshell/

[2] D. Boneh, J. Drake, B. Fisch, and A. Gabizon, "Halo infinite: Proof-carrying data from additive polynomial commitments," in *Advances in Cryptology – CRYPTO 2021*, T. Malkin and C. Peikert, Eds. Cham: Springer International Publishing, 2021, pp. 649 – 680.

[3] A. Nitulescu, "Lattice-based Zero-knowledge SNARGs for Arithmetic Circuits," in *Cryptology ePrint Archive, Paper 2019/1251*, 2019, https://eprint.iacr.org/2019/1251. [Online]. Available: https://eprint.iacr.org/2019/1251

[4] N. Bitansky, R. Canetti, and C. Alessandro, "The Hunting of the SNARK," in *Journal of Cryptology*, 2017, vol. 30, pp. 989 – 1066.

[5] O. Goldreich, S. Micali, and A. Widgerson, "Proofs that Yield Nothing But Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems," in *Journal of the ACM*, 1991, vol. 38, pp. 690 – 728.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to Algorithms." Massachusetts Institute of Technology, 2009, pp. 43 – 64.

[7] M. Sipser, "Introduction to the Theory of Computation." Thomson Course Technology, 2006, pp. 247 – 253.

[8] D. Barker-Plummer, "Turing Machines," in *The Stanford Encyclopedia of Philosophy*, Winter 2017 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2017. [Online]. Available: https://plato.stanford.edu/archives/win2017/entries/turing-machine/