Summary of "Proofs of Space"

[Dziembowski, Faust, Kolmogorov, Pietrzak]

Motivation

The motivation for Proofs of Space (PoS) comes from the motivation for the broader concept of Proofs of Work (PoW). A PoW is a means of showing that some non-trivial amount of computational work was done in relation to some statement. PoWs were orginally proposed to counteract email spamming. A spammer has the goal of sending out large quantities of emails over a short period of time. This is in contrast to a regular email user, who may only need to send out relatively few emails per day. An email service provider such wishes to limit the ability of spammers to send out large quantities of emails but does not wish to inconvenience regular users. One way to accomplish this is to have a computational cost associated with each email sent. This cost would be in the form of a value that is moderately hard to compute, but easy for the email provider to verify. The example provided in the paper is as follows: The PoW is a value σ such that the hash $\mathcal{H}(\text{email}, \sigma)$ begins with t zeros. The sender would have to compute an expected 2^t hashes to find σ. t is chosen to be large enough so that it is computationally expensive for a spammer to send out many emails, but small enough so that a regular user sending out a few emails per day is not burdened. On the provider's side, it is easy to verify whether or not $\mathcal{H}(\text{email}, \sigma)$ begins with t zeros. Some other applications of PoWs are:

- Metering website access
- Countering denial-of-service attacks
- Block generation for cryptocurrencies, such as Bitcoin
- The CAPTCHA system, which uses human attention as the PoW

The idea of PoSs follows from the idea of a PoW, but whereas the latter uses computational power as the investment, a PoS uses disk space. Consider the following variant of the email example: Instead of limiting the number of emails a user can send out, the provider limits the number of email accounts, to prevent spammers from registering many email addresses. For each registered email address, a user would be required to dedicate a non-trivial amount of disk space (the paper suggests 100GB). Occasionally the verifier, the email service provider, would ask the user for a PoS (i.e prove that they have actually dedicated the space). A simple but impractical solution would be to for the verifier to send a random, and therefore incompressible 100GB file to the user. Later, the verifier would request bits from the file at random positions. To avoid the verifier having to store the entire file, a PRF could be used to generate the file and then the verifier would only have to send the file once to the user. Unfortunately this is impractical for a service such as email where the provider might have many new registrations per day, and would have to send out many 100GB files.

So far the motivation for PoSs has been established. The desired scheme is one where the computational, storage, and commmuncation complexity of the verifier is much smaller than that of the prover, in this case polylogarithmic in the storage requirement of the prover. In order to acheive this, the prover must generate the file, and then later the verifier will

run the PoS, which will be cheap for both the verifier and prover, assuming the prover has stored the file. Unlike the simple scheme mentioned before, there is no way to ensure that a cheating prover might not store the file, and would instead generate it whenever the PoS is run. This fact is addressed in the security definition of the scheme.

PoS Definition + Security Definition

A PoS is an interactive protocol between two parties, a prover P and a verifier V. There are two kinds of protocols involved in the PoS, an initialization and an execution. The PoS has perfect completeness, so for any honest prover that stores the entire file, the verifier will accept with probability 1. The protocol does not acheive perfect soundness. However the verifier will only accept the output of a cheating prover with negligible probability.

The security and efficiency of the PoS is analyzed in the random oracle model, in which all parties have access to the same random function (collision-resistant hash) $\mathcal{H}(\{0,1\}^* \to \{0,1\}^L), L \in \mathbb{N}$. In the security for this scheme, both the time and storage complexity of both parties are measured in relation to $\mathcal{H}$. The running time of the parties is measured as the number of queries made to $\mathcal{H}$, with longer queries taking time proportional to the length of the query. The storage of both parties is measured as the number of outputs of $\mathcal{H}$ that are stored.

The security of the protocol (N_0, N_1, T) is stated in terms of bounds on a cheating prover, $\widetilde{P}$. In the initialization phase, $\widetilde{P}$ will make queries $\mathcal{A} = (a_1, \ldots, a_q)$ to $\mathcal{H}$. Then P' will store a file S of $N_0 L$ (each output of $\mathcal{H}$ is of length L) bits of these values, which will be denoted by $S_{\mathcal{H}}$. Because $\mathcal{H}$ is a random oracle, its outputs are incompressible, and the assumption is made that $\widetilde{P}$ cannot encode more than N_0 outputs of $\mathcal{H}$ in $N_0 L$ bits.

In the execution phase, V asks $\widetilde{P}$ to output $\mathcal{H}(b)$ for b in a subset of $\mathcal{A}$. With the above assumption, $S_{\mathcal{H}}$ is the set of outputs that $\widetilde{P}$ can produce without additional computation. N_1 is defined to be the amount of storage that P' is allowed to use at any time during the execution, which includes the originally stored N_0 hashes. T is the number of queries that P' is allowed to make during the execution. Security is defined such that a PoS is $(N_0, N_1, T)-$secure if every $(N_0, N_1, T)-$adversary will make V accept with only negligible probability.

Graph Pebbling Game

Before discussing the construction of the PoS, we look at another interesting concept in complexity theory, called the graph pebbling game, that will be useful for proving the security of the PoS. The game consists of a DAG $G = (V, E)$, and some number of pebbles M. The rules for pebbling a vertex are as follows:

- At each step, a player may pebble a single vertex of the graph
- A vertex may only be pebbled if each of the vertex's predecessors has a pebble.

The goal of the game is to be able to pebble every vertex with M pebbles. Note that it is trivial to pebble a graph with N pebbles. The pebbling game can be used to model the time-space tradeoffs of the PoS. The bounds on the number of pebbles required to pebble different families of graphs is well researched, and the security of the PoS will be shown by reducing it to a graph pebbling game.

Construction

Consider a DAG $G = (V, E)$ with $N = |V|$ vertices. Every vertex $v \in V$ has an associated value $w(v) \in \{0,1\}^L$. Let $\pi(v)$ be the set of predecessors of v, and let

$$w(v) = \mathcal{H}(v, w(\pi(v))).$$

If v has no predecessors, then let $w(v)$ be the hash of its index. Let both parties be given this graph, and the prover is asked to compute the hash for every vertex in the graph. To make the verifier efficient in checking that a computed hash is consistent, a hash tree is used. As an example, for $N = 8$, the hash tree would be as follows:

$$\phi = \mathcal{H}(\mathcal{H}(\mathcal{H}(x_1, x_2), \mathcal{H}(x_3, x_4)), \mathcal{H}(\mathcal{H}(x_5, x_6), \mathcal{H}(x_y, x_8)))$$

It is sufficient for the prover to compute the root of the hash tree and send it to the verifier during the initialization phase. Then, the verifier requests a challenge set C of random vertices. The prover sends can send $\log N$ values to the verifier for each $c \in C$. These values correspond to the siblings of the nodes on the from c to ϕ, the root of the hash tree. For example to open x_3, the prover would send (x_{12}, x_4, x_{5678}). Then the verifier can check that:

$$\mathcal{H}(x_{12}, \mathcal{H}(x_3, x_4)), x_{5678}) = \phi$$

The binding properties of the hash tree mean that it is hard for a cheating prover to commit ϕ and later send different values for any x_i. Once the verfifier checks that the hash tree is consistent, the initialization phase ends.

During the execution phase, the verifier sends another challenge set to the prover, and asks that the prover provide the $\log N$ hash values of the siblings of every node in the challenge set. If they are still consistent, the verifier accepts. It is easy to see how one can reduce the problem of generating the hash value for a given vertex in the graph to pebbling that vertex in the graph pebbling game.

Security

The security of the PoS comes from the bounds on the pebbling game mentioned earlier. The paper shows two different families of graphs can be used to generate different securities for the PoS. Using these graphs with high pebbling complexity, two different securities for PoSs are shown (all constants $c_i > 0$):

$$(c_1(N/\log N), c_2(N/\log N), \infty)\text{-secure PoS}$$
$$\text{and}$$
$$(c_3 N, \infty, c_4 N)\text{-secure PoS}$$

The first result means that any cheating prover storing a file of size (measured in L bit blocks) $O(N/\log N)$ during the initialization phase must use $\Omega(N/\log N)$ storage during the execution phase. There is no bound on the time that this prover is allowed to use. The second result means that any cheating prover msut either use $\Omega(N)$ storage initially or make $\Omega(N)$ calls to the random oracle. The efficiencies of all parties are given in the table below (γ is the security parameter):

	communication	computation P	computation V
PoS 1 Init.	$O(\gamma \log^2 N)$	$4N$	$O(\gamma \log^2 N)$
PoS 1 Exec.	$O(\gamma \log N)$	0	$O(\gamma \log N)$
PoS 2 Init.	$O(\gamma \log N \log\log N)$	$N \log\log N$	$O(\gamma \log N \log\log N)$
PoS 2 Exec.	$O(\gamma \log N)$	0	$O(\gamma \log N)$