The Theory Behind Blockchains (Spring 2019)
Recitation 4

Nir Bitansky

1 Proofs of Work

Definition (Dwork-Naor 92): A proof of work consists of three algorithms:

- $Gen(d, n)$: takes a difficulty parameter d and and additional security parameter n and outputs a puzzle p.
- $Solve(p)$: takes a puzzle p and outputs a solution s.
- $Ver(p, s)$: takes a puzzle p and an alleged solution s and verifies it.

Efficiency:

- A puzzle is generated fast in some fixed number of steps $\approx n$, independently of d.
- Solving a puzzle takes d steps (typically $d \gg n$).
- Verifying can be done fast with $\approx n$ steps.

Non-amortization:

- An algorithm that runs for $\ll d \times t$ steps fails to solve a randomly generate puzzles $p_1, \dots, p_t$, except with negligible probability $negl(n)$. (Here we could of course define a more precise notion of amortization parameterized by a function $f(d, t, n)$, which may allow for some amortization. E.g., $f = \sqrt{t} \times d$.

How to think about the parameters. A puzzle is meant to capture moderate hardness. You should think of both d and t as a very large polynomials in n, e.g. n^{10}.

1.1 Is this enough for blockchain consensus?

The idea in Bitcoin is to use puzzles in order to distribute block creation, where the aim to divide the right to create block proportionally to computational power. As defined above, it's not clear that proofs of work can necessarily achieve this.

In particular, imagine that we have one malicious party A that controls a 3/7 of the computational power and two other honest parties B and C who each control 2/7 of the computational power. If either B or C compete against A in solving a puzzle p, they will naturally lose. For them to win, they should be able to divide the work between them. That is, solving the puzzle should be something that can be parallelized.

In fact, things seem even more tricky since the honest parties are not really aware of each other and cannot coordinate jointly solving the puzzle. Rather, the honest parties B and C should independently invest work so that the probability that one of them ends up with the solution before the malicious A becomes $\approx 4/7$.

Indeed, this can be achieved as long as the best way to solve a puzzle is by repeatedly trying solutions at random, which is the type of POW that bitcoin presumably has.

2 Preprocessing and Time-Space Tradeoffs

In general, we don't have a deep understanding of the concept of moderate hardness. Typically, we consider either adhoc candidates or constructions in idealized models, like the random oracle model.

We'll now discuss one of the basic rules for choosing a moderately hard function, which is to be aware of preprocessing attacks. Specifically in a preprocessing attack, we perform a certain long precomputation. This computation, which we think about as being *offline*, later allows to solve puzzles *online* much faster. To ensure, appropriate non-amortization this should be taken into account.

Claim (trivial): Consider any puzzle generators, with puzzles and solutions of size n each. Then we can always perform a $O(2^{2n})$ long preprocessing that would later allow us to solve any puzzle in time $O(n)$.

Simply keep the truth table. Of course that a major caveat is that this preprocessing attack requires huge storage $\approx 2^n \times n$, and storage is typically even more costly than computation. So we should be aware of the tradeoff.

Indeed, in some cases, there may be less trivial preprocessing attacks.

Example: inverting a permutation assume you're given access to a random permutation $p : \{0,1\}^n \to \{0,1\}^n$. Then inverting $p(x)$ for a random x would naturally take $\approx N = 2^n$ steps.

Claim: Any permutation p can be preprocessed so that with storage $\sqrt{N}$ we can invert it on any input in time $\approx \sqrt{N}$.

We use the fact that any permutation can be decomposed to distinct cycles. Now we can take any such cycle of length ℓ and divide to $\ell/\sqrt{N}$ parts of length $\sqrt{N}$ and store the end points. Then for each cycle we'll keep an ordered list of these endpoints. The endpoint themselves are stored in a sorted array.

To invert $p(x)$ we start applying p from $p(x)$ until we either hit an element in these lists, in which case we return to the element before and walk towards the preimage, or we get to the preimage.