Ahoy-hoy Ethereum Researchers,

For the past few months I've been exploring the potential of compressing random data and I'm requesting feedback on my current approach. I will describe how my algorithm works and where my research is at towards the end (the code for all of this is around 95% complete.)

I want to state that I don't believe random data can be compressed directly – HOWEVER: and this is important – it might be possible to approximate random data and compress that approximation.

Here is how my compression algorithm works:

1. A 1024 byte buffer is split into 17 bit chunks.
2. Each chunk is 'stored' in a golomb-coded set (GCS) and prefixed with an offset id (e.g. chunk 1, chunk 2 …) . Space required = 706 bytes.
3. For every offset in the GCS a list of candidates are generated by combining the offset ID with every possible 17 bit value.
4. Within a candidate list the chunk offset is recorded, creating a list of 482 chunk offsets. The first bit of every chunk is also used as a way to make the candidate list smaller (total size is 61 bytes for the bit filter.)
5. The chunk offsets are grouped into pairs [x, y], and the pairs are grouped into sets of four [[x, y], …], … There are 61 such sets.
6. A modified proof-of-work algorithm is done for every quad-set:
7. For every nonce in a 4 byte range calculate standard proof-of-work for each pair.
8. Increment total cumulative PoW.
9. Keep a list of nonces sorted by total cumulative PoW.
10. For every nonce in a 4 byte range calculate standard proof-of-work for each pair.
11. Increment total cumulative PoW.
12. Keep a list of nonces sorted by total cumulative PoW.

The nonces require 241 bytes to store and 61 bytes have been used for the bit filter. That only leaves 13 bytes but as I'll show later – it may be enough.

Decompression

To decompress the data involves reconstructing the quad-sets and using the offsets therein to dereference the original data inside the GCS. The decompression algorithm is based on the following observations:

1. There is a certain number of zero-prefix bits associated with the nonces that have the best possible cumulative PoW.
2. We don't know ahead of time where these prefix bits may fall among the pair hashes. But since there are multiple results it's possible to impose a rule-based filter. E.g. all hashes must have at least 4 zero-prefix bits.
3. This allows for a heuristic filter to be used for brute forcing the quad-sets using the 4 byte nonce and the number of candidates for each chunk.

The filtering approach will return a list of possibilities for each quad-set. The offset among these possibilities will be saved in meta data, along with the nonce. There is still 13 bytes remaining, so the last challenge is to be able to encode every offset in less than 13 bytes. At first inspection this doesn't seem possible because if we assume a worse cause scenario of 11 bits per offset – that totals 83 bytes. But the nonce list contains patterns and that may be key to solving this problem.

On average it requires a certain number of attempts to find high-value PoW for the nonces. So most of the nonces tend to be high value. A simple scheme that divides all of the nonces by the lowest nonce, saves the quotient, remainder, and min nonce saves an additional 60 bytes. By the way: all of the parameters chosen so far are to prevent exponential growth of result sets – allowing the code to finish – or for space-saving / CPU efficiency reasons. They are not arbitrary.

Here is the problem

This is where my research stops and I'm left with a bunch of unanswered questions. What range of offsets can we expect

for all the quad-set candidates after filtering them? Can they be compressed like the nonces can? Can the bit filter be compressed? Is filtering every quad-set even possible? Can the bit filter be eliminated? What else am I missing?

Due to the large amount of compute resources required for filtering I have only confirmed successful recovery of a single quad-set.

My private compute cluster is only 96 cores at the moment which isn't enough to continue this research (it would probably take a month to finish one run of the algorithm.) I believe that I'm close to a working algorithm

but I lack the resources to prove this. If any of you think this will work or have access to a ton of CPU cores (around 512 if not more) – I think I can prove my approach will work.

All feedback welcome.

PS:

Here is the current code just to give you a more precise idea:https://github.com/robertsdotpm/rand

My code is already mostly complete and shows how to marshal cryptographically random data into a GCS in less space and use a series of nonce puzzles to recover it.

The code points to my Spark cluster and isn't finished yet though so don't bother trying to run it.