Seeking a Better Proof of Work Function for Cryptocurrency

Samuel Erb
Graduate Student, WPI
srerb@wpi.edu / samrerb@erbbysam.com

Abstract

Designing an ideal proof of work function has proved to be a challenging task for the designers of crypto currencies. There have been many attempts to solve various deficiencies presented by existing proof of work functions, but many problems still remain. This paper attempts to define what would make a proof of work function ideal in current computing world. This paper will also define an improvement to the Cuckoo Cycle algorithm in order to form a more ideal proof of work function.

Bitcoin

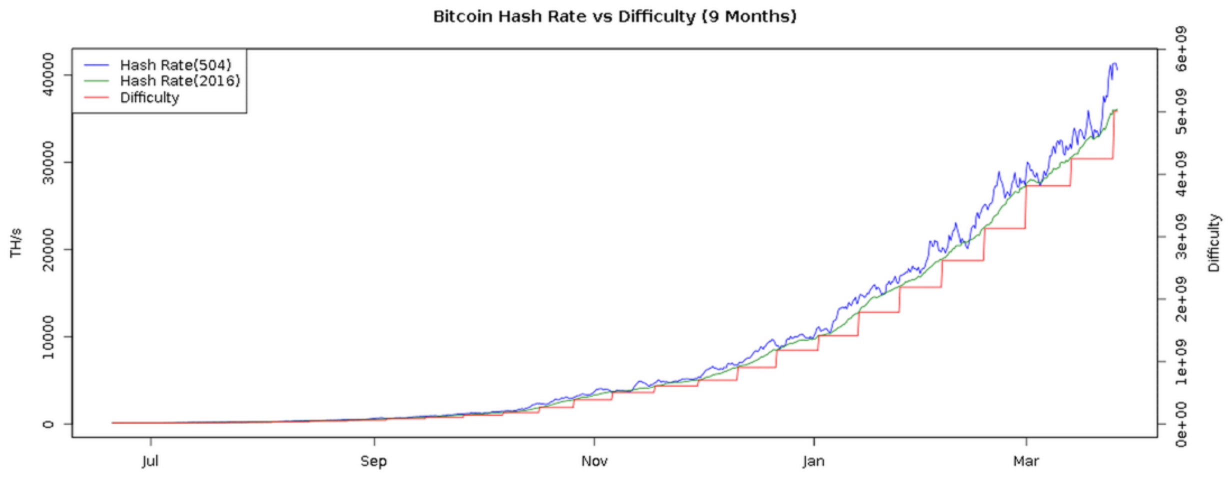
Before it is possible to define an ideal proof of work function we must first define how we're going to use it. **Bitcoin**[0] is cryptographically strong decentralized digital currency. There is no central control or authority and the network as a whole controls transactions.

In order to issue a new **transaction**, all a bitcoin owner needs is to broadcast that transaction to the network. The transaction defines an input and a smaller output value which are not even, the difference is the transaction cost. If this was all that happened, it would be trivial to beat that network propagation in order to double spend any bitcoins that you had.

In order to work around this double spending problem, there is something called the blockchain. The **blockchain** is a record of all transactions that have occurred in the network. The longest blockchain that exists from a hardcoded 0th block is considered the accurate record of all transactions. The goal for bitcoin miners is to find the next block in the chain before anybody else does.

When a transaction is propagated to the entire network, **miners** pick that up and add it to the next block (if there's a reward for doing so in form of that transaction cost). They then try and prove that they "own" this block. Any node on the network can prove ownership by completing a proof of work function first on the network. If the completes the proof of work first, they broadcast out the block, which they embed their node ID in and that adds the block value to their address. Each block's value is agreed on by the network to a set value which goes down over time as well

as all of the transaction costs accumulated in the block.
For bitcoin, the proof of work function is to find a double sha256 that is less than a certain value (where that value is based on the globally agreed network **difficulty** which is scaled to make the proof of work solution average 10 minutes on the network) & each block awards at least 25 bitcoins, or more than $12500 at the time of writing. This award is often split between multiple miners who team up together (a **mining pool**) to solve the proof of work function first (and prove they participate by sharing a simpler version of the proof of work function (ie. less zeros)).
Quickly searching the internet for “Bitcoin asic” turns up a large amount of custom built hardware for mining as quickly and efficiently as possible. This has lead to an ever increasing arms-race to build & faster custom hardware.

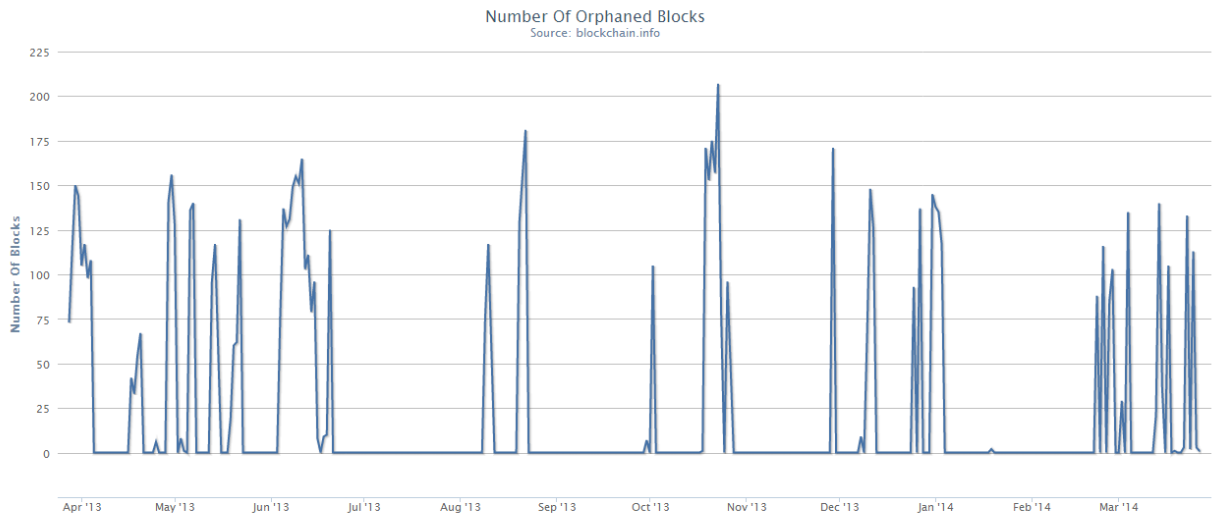


dates are 2013-2014 [1]

Blockchain splits

Occasionally, multiple miners discover solutions to the next block at exactly the same time. When they do they both propagate their solution out to the network. Whichever solution reaches a node first is accepted as their next block. When the miner who finds the next block after the split chained from the first block that they saw will cause the first block they saw to be part of the new longest chain in the network and will therefore be accepted by the network completely. The nodes that saw the other block first previously will discard it and pick up the new chain.
This can become a problem if the propagation on the network is slow[3]. Slow propagation speeds can be caused by nodes not being interconnected enough, a slow block verification function and a non-optimized algorithm. Bitcoin was helped most by increasing node

interconnectivity in order to reduce the number of orphaned blocks propagated to the network[3]. However, as we will show below, a slow block verification function can be significantly harmful to propagation times and therefore increase the likelihood of orphaned blocks.



number of orphaned Bitcoin blocks [2]

Blockchain verification speed

When a new user connects using a Bitcoin Client, they must download every other block from the other nodes in the network. This presents a challenge as every block downloaded must be verified using the Bitcoin verification function. At the time of writing there are 292815 bitcoin blocks[5] that would have to be verified for a new client connecting. This would mean 2x292815 SHA256 hashes computed.

While the test below shows that computing this many hashes is trivial, a more complex & time consuming verification function would become a factor in how long it would take to sync the blockchain on a local computer.

Proof of work functions

While bitcoin uses a double sha256 hash as it’s proof of work function, that is not the only option. We also do not believe it is the optimal choice for a proof of work function. Many other crypto currencies use different hash functions, we will detail two below.

Litecoin

Litecoin is identical to Bitcoin expect for the fact that it issues a new block every 2.5 minutes and

uses the scrypt hash function for verification and is preconfigured to issue 4 times as many coins as Bitcoin[6].
The scrypt hash function was originally built as a replacement to existing password hashing functions such as PBKDF2[7]. It works by requiring a very large amount of memory which will then be randomly accessed in sequential order to form a hash of the input data. While the memory blob being accessed is deterministic and could therefore be computed only when needed, it would be significantly slower to do so than to simply generated & store this blob in RAM once, then accessing it as needed. In order to verify a Litecoin hash, the memory blob must be recreated.
Unfortunately, due to the rise in popularity in Litecoin and the initial memory requirement set rather low ($N=1024$, $r=1$, $p=1 \rightarrow$ 128k memory required per hash[8]), GPU accelerated software and custom ASIC hardware has been developed speed up Litecoin mining, which is something that it's implementers hoped to avoid.

Vertcoin

Vertcoin is identical to Litecoin expect is uses a variable "N" parameter in the scrypt verification function to attempt to keep up with increases in computer hardware[9]. At predetermined dates, the N factor in the scrypt hash function will rise.
While adjusting parameters or the hashing function itself is something that could be done if a majority of a bitcoin-like network agrees to (such as through a client-code update), Vertcoin proactively works to avoid custom hardware from being built to mine it.

ASIC design & cost

It is outside the scope of this paper to develop any theory around ASIC design & costs. However if there is a way to exploit or solve proof of work functions faster by designing custom hardware, than there will eventually be a profit to be made by doing so.

Memory and the speed wall

Memory speed increases over the past decades has not kept up with the exponential increases in CPU speed[11]. A general purpose computer is already hitting the "memory wall".
A proof of work system can exploit this fact as memory latencies likely will not change significantly in the immediate future and any custom hardware that could be built to calculate the proof of work significantly faster than an existing computer would result in a breakthrough in the

“memory wall” [9].

Modern Botnets

Botnet have started to mine crypto currency for the Botnet owner and have made 1000’s of Bitcoins by doing so[4]. We must define that Botnets target the average consumer PC as there is no information to speak to the contrary. Before accelerated hardware such as GPU’s and ASIC’s are used to mine crypto currencies, they are generally mined by CPU only. This can make using botnets to mine profitable for the owners and provide economic incentive for them to grow them larger by infecting more computers.

Cuckoo Cycle

This leads us to a new algorithm, Cuckoo Cycle, which has been specially designed to be a more ideal proof of work function [10]. It has a large memory requirement for finding cycles within a bipartite graph and can be trivially verified (2 SHA256 hashes + 42 siphash hashes) without the memory requirement. This is a more ideal proof of work function as it does not have the downside of requiring a large amount of memory to verify the hash.
This algorithm works by defining a bipartite graph. By then adding edges based on a fast preimage resistant hash function (siphash) and the input to be hashed the algorithm attempts to form a cycle. Once a cycle has been found, it can verified in a trivial manner that it is valid by keeping track of the nonce’s used to generate it.
At this time there is still a need for research to be done to examine the effects of parallel CPU’s on this algorithm as an unexpected speedup was observed during testing.

Hash speed

As viewed below, we can see how long it takes to verify 200000 blocks for bitcoin, litecoin and a hypothetical cuckoo cycle based currency.

```
bitcoin:
total: 1.270000 sec time per hash: 0.000006 sec runs:200000
cuckoo cycle:
total: 0.640000 sec time per hash: 0.000003 sec runs:200000
Litecoin:
total: 92.639999 sec time per hash: 0.000463 sec runs:200000
```

Run on authors laptop (2013 core i5). see appendix A for code.

These numbers are gathered under unrealistic circumstance, in particular we did not use an actual block during hashing, but it is still worthwhile to note just how much slower Litecoin verification is when hashing the same value, even with the low scrypt parameters used.

Defining an Ideal Verification Function

We can use everything we have mentioned above to define an ideal crypto currency verification function. This is similar to the list defined in section 2 of [9]. The three properties we seek are:
Botnet Resistance - The verification function should have a predefined required amount of RAM and parallelization speedup which would optimally run on a mid-high end average PC. This should handicap the usability of such a PC such that the mining would be noticable[9].
ASIC Resistance - The amount of RAM required should be large enough such that the primary user of computation time is memory chip latency's. Parallelization or speedup of any CPU function should not create a significant speedup in the time to compute the proof of work.
Quick Verification - By having a trivial verification function, this works to lowers block chain fork % as well as allowing new users to sync up faster. This verification function likely should not be memory hard and parallelizable for future custom hardware implementations.

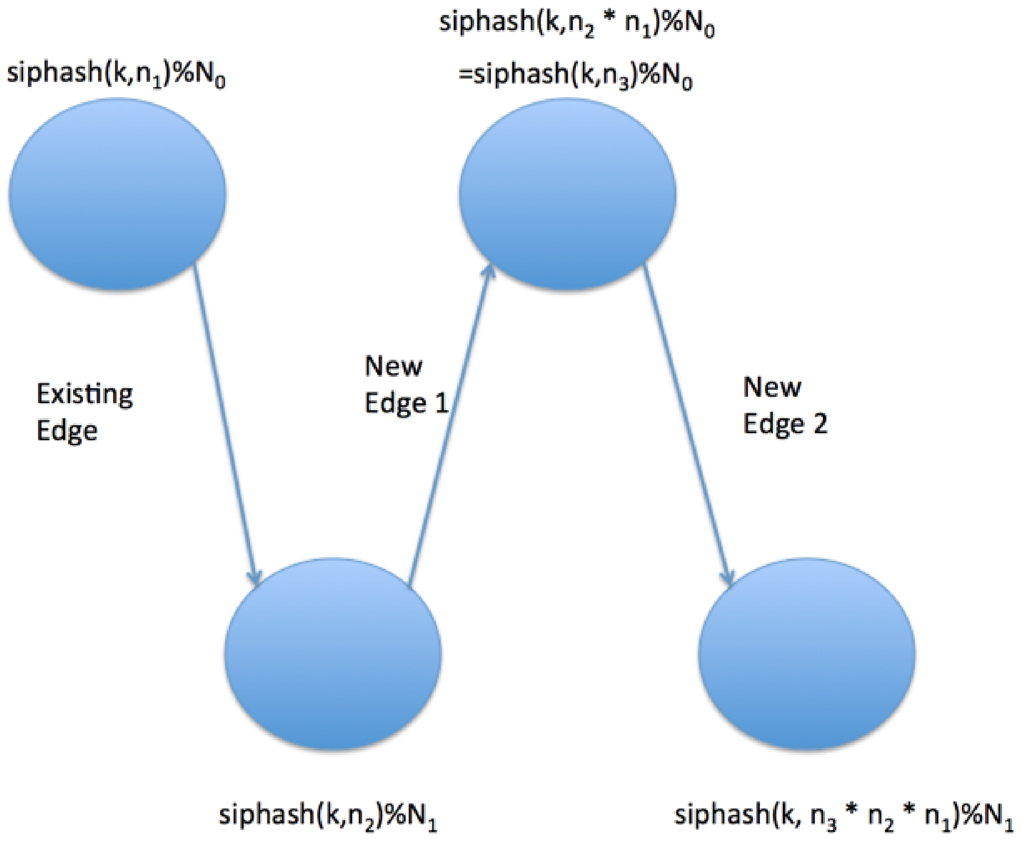
Cuckoo Cycle Parallelization

The Cuckoo Cycle function defined above has been observed to see unexpected enhancements by adding multiple parallel threads to compute edges and has been observed to be significantly speed up by precomputing the edges formed from the siphash hashes[9]. We would like to formalize an approach to a massively parallel computation to the Cuckoo Cycle that works by breaking down the problem into more manageable steps. We believe that this parallelization would decrease both ASIC and botnet resistance as step 1 & 3 below are massively parallelizable and steps 2 & 4 are trivial. This does not lower the memory hard requirements of the Cuckoo Cycle hash, although as step 1 is only dependant on RAM write speed and step 3 is only dependent on RAM read speed, we would not be able anymore to exploit complex read/write situations in the cache which would likely cause a further, desirable, cache slowdown.
Steps:
1) Precompute all possible siphash values, this can be trivially parallelized. Once computed, store them in the bipartite graph.
2) Broadcast the bipartite graph to n cores(likely multiple hosts as well), such that each host can

store this bipartite graph in memory.
3) At each core, start searching for possible Cuckoo Cycle solutions from position (1/n) * (# of vertices in half the bipartite graph). Each host will have to have the entire graph in their memory as a solution can include any vertex in the graph.
4) Once a solution has been found, return it to the host computer.

Cuckoo Cycle modification

As outlined above, the Cuckoo Cycle hash can be easily parallelized in order to speed up finding a solution. We see a simple solution to this problem and that is to make the computation of edges dependant on previous computations. In order to keep the computationally cheap verification function we must be careful not to make our edge computations based on too many previous computed edges. The computation is as follows:
When a new edge is added to the bipartite graph, if a vertex is going to be shared with another edge, the other vertex should be recomputed. The new value should be the siphash of the multiplication of all previous nonces in the partial loop with the new nonce. In picture form, using the notation from the original Cuckoo Cycle paper:


$\mathrm{siphash}(k,n_1)\%N_0$
$\mathrm{siphash}(k,n_2 * n_1)\%N_0$
$=\mathrm{siphash}(k,n_3)\%N_0$
Existing Edge
New Edge 1
New Edge 2
$\mathrm{siphash}(k,n_2)\%N_1$
$\mathrm{siphash}(k, n_3 * n_2 * n_1)\%N_1$

This algorithm means that the siphash values cannot be precomputed as the nonces used for the siphash values could range from 0 to N^{42} where N is the maximum nonce value. This drastically increase the amount of space required to precompute siphash values.
This presents algorithm modifications:
-When the recomputed vertex of a new edge matches the existing vertex of another edge, that edge could then have it's non-shared vertex recomputed as well, further lengthening the chain.
-if a recomputed edge causes a loop that is too short, the loop is no longer useful to the computation the edge that caused that loop could likely be discarded.
-The verification loop must include a starting vertex marked (this would mean that there would be an additional bit per vertex when communicating the solution. In the case of a 42 edge loop, this would be an additional 6 bytes to transmit and would only add 41 multiplications to the verification calculation).

Formal definition of our modification
Modifying the definition from [9]. Fix an even number of nodes N and an even cycle length L < N. Function cuckoo maps any 128-bit key k (the header digest) to a bipartite graph G = (V_0 U V_1 , E), where V_0 and V_1 are disjoint set of integers modulo N/2, and E has an edge between **siphash(k, 2*j*n) mod N/2** in V_0 and **siphash(k, j*n) mod N/2** in V_2 for every nonce n and sum j.
Starting from an initial vertex and j=1, each edge q that is added to form a cycle will have one vertex with a value of j=1 and another with the value for j equal to the multiplicative sum of a previous n values $\prod_{j=1}^{q} n_j$ from edge 1 to q. The vertex with the multiplicative sum must be the vertex which does not connect to the previous edge or must be the one that connects back to the initial vertex.
A proof for G is a subset of L nonces whose corresponding edges form an L-cycle in G.

A practical note on j
The value for j can be as large as q! in the definition above. This can quickly grow to large to fit into a 64 bit value. Therefore, we can take j mod (2^64 / max nonce value), or in C (ULLONG_MAX / NONCEMAX). In such a scheme, in order to create a lookup table for each J value with each possible nonce value less than 2^16 (where 2^ 16 is NONCEMAX) it would take: 2^48*2^16 possible positions, each needing (48+16) bits to represent it or 2^48*2^16*(48+16) bits

total = 147.6 EB (EB=exabytes by wolfram alpha) for each side of our graph.

Looking back at our Cuckoo Cycle Parallelization

We conjecture that the modification we made above would make the Cuckoo Cycle hash resistant to computational parallelization decomposition we defined above. This should increase botnet & ASIC resistance as we can no longer quickly precompute siphash values and now force a cache of a computer seeking a solution to have to handle both reads and writes while only adding 6 bytes and 41 multiplications to hash verification.

Restating the parallelization here after the algorithm modification:

1) Precompute all possible siphash values, this can be trivially parallelized. Once computed, store them in the bipartite graph. **This would no longer be easy as each edge past the first in a loop has a hash based on all previous edges, drastically increasing the number of hashes that would have to be precomputed to N^{42} as shown above.**

2) Broadcast the bipartite graph to n cores(likely multiple hosts as well), such that each host can store this bipartite graph in memory. **Each host would have to have enough space for N^{42} precomputed values for this to be viable. This is not realistic even if we are requiring just 100KB of RAM.**

3) At each core, start searching for possible Cuckoo Cycle solutions from position (1/n) * (# of vertices in half the bipartite graph). Each host will have to have the entire graph in their memory as a solution can include any vertex in the graph. **Each core as defined here would have to generate the cycle while searching for a solution effectively removing steps 1 & 2.**

4) Once a solution has been found, return it to the host computer.

Conclusion

In this paper we showed the current state of proof of work functions in crypto currencies and defined what would make an ideal proof of work function. We also proposed a change to the Cuckoo Cycle algorithm in order to enhance its resistance to parallelization. There are and will likely be more crypto currency breakthroughs (such as proof of stake[12]). Designing an ideal proof of work function for a crypto currency requires a knowledge of algorithm design, modern computing and cryptology. As the current market cap for Bitcoin hovers around $7.2 billion[13] this will likely be an area of research well into the future.

References

[0] Bitcoin: A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto
https://bitcoin.org/bitcoin.pdf
[1] website, “Bitcoin Difficulty”, accessed March 2014
https://bitcoinwisdom.com/bitcoin/difficulty
[2] website, “Number of Orphaned Blocks”, accessed March 2014
http://blockchain.info/charts/n-orphaned-blocks
[3] Information Propagation in the Bitcoin Network, Christian Decker, Roger Wattenhofery
http://www.tik.ee.ethz.ch/file/49318d3f56c1d525aabf7fda78b23fc0/P2P2013_041.pdf
[4] Botcoin: Monetizing Stolen Cycles, Danny Yuxing Huang etc.
https://cseweb.ucsd.edu/~smeiklejohn/files/ndss14.pdf
[5] website, no title, accessed March 2014
https://blockexplorer.com/q/getblockcount
[6] Technical Basis of Digital Currencies, Simon Sprankel
http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf
[7] Stronger Key Derivation via Sequential Memory-Hard Functions, Colin Percival
http://www.tarsnap.com/scrypt/scrypt.pdf
[8] website, “scrypt”, accessed March 2014
https://litecoin.info/Scrypt
[9] PDF, “What is Vertcoin?”, accessed March 2014
https://vertcoin.org/Vertcoin-DavidMuller.pdf
[10]Cuckoo Cycle: a graph-theoretic proof-of-work system, John Tromp
https://github.com/tromp/cuckoo/blob/master/cuckoo.pdf?raw=true
[11] Hitting the Memory Wall: Implications of the Obvious, Win. A. Wulf, Sally A. McKee
http://www.di.unisa.it/~vitsca/SC-2011/DesignPrinciplesMulticoreProcessors/Wulf1995.pdf
[12] website, “Proof of Stake - Bitcoin”, accessed March 2014
https://en.bitcoin.it/wiki/Proof_of_Stake
[13] website, “Bitcoin Tops $600, Up 60x Over the Last Year”, accessed March 2014
http://mashable.com/2013/11/18/bitcoin-600/

Appendix A - code used to compute hash speeds above:

```
#include "scrypt_platform.h"
#include "scryptenc.h"
```

```
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <openssl/sha.h>

//sha256 function from:
//http://stackoverflow.com/questions/7853156/calculate-sha256-of-a-file-using-openssl-lib
crypto-in-c
void sha256(char *string, char *outputBuffer)
{
  unsigned char hash[SHA256_DIGEST_LENGTH];
  SHA256_CTX sha256;
  SHA256_Init(&sha256);
  SHA256_Update(&sha256, string, strlen(string));
  SHA256_Final(hash, &sha256);
  int i = 0;
  for(i = 0; i < SHA256_DIGEST_LENGTH; i++) {
    sprintf(outputBuffer + (i * 2), "%02x", hash[i]);
  }
  outputBuffer[64] = 0;
}

int main (char *argv, int argc) {
  unsigned long long count = 200000;
  char to_sha[] = "example_to_sha\n";
  char *output;
  int i;
  clock_t ta, tb;

  output = calloc(1,65*sizeof(char));

  /* example bitcoin verification
   * -major work is going to be calculating the 2 SHA256 hashes
   * -uses openssl SHA256 hash function
   */

  ta = clock();
  for (i = 0;i<count;i++) {
    sha256(to_sha, output);
    sha256(to_sha, output);
  }
  tb = clock();
  printf("bitcoin:\n");
  printf("total: %f sec time per hash: %f sec runs:%llu\n",
         (((float)tb-ta)/CLOCKS_PER_SEC),
         (((float)tb-ta)/CLOCKS_PER_SEC/count),
         count);

  /* example cuckoo cycle verification
   * -major work is going to be calculating the SHA256 hash
   * -uses openssl SHA256 hash function
   */

  ta = clock();
  for (i = 0;i<count;i++) {
    sha256(to_sha, output);
  }
  tb = clock();
  printf("cuckoo cycle:\n");
  printf("total: %f sec time per hash: %f sec runs:%llu\n",
         (((float)tb-ta)/CLOCKS_PER_SEC),
         (((float)tb-ta)/CLOCKS_PER_SEC/count),
         count);
  /* example litecoin verification (N=1024, r=1, p=1)
   * -major work is going to be calculating the scrypt hash
   * -using scrypt hash provided via http://www.tarsnap.com/scrypt.html
   */

  ta = clock();
  for (i = 0;i<count;i++) {
    crypto_scrypt(to_sha, sizeof(to_sha), to_sha, sizeof(to_sha),
                 1024, 1, 1, output, sizeof(output));
  }
  tb = clock();
  printf("Litecoin:\n");
  printf("total: %f sec time per hash: %f sec runs:%llu\n",
         (((float)tb-ta)/CLOCKS_PER_SEC),
         (((float)tb-ta)/CLOCKS_PER_SEC/count),
         count);

}
```