Cuckoo Cycle
a memory-hard proof-of-work system

John Tromp

January 26, 2014

1 Introduction

A "proof of work" (PoW) system allows a verifier to check—with negligible effort—that a prover has expended a large amount of computational effort. Originally introduced as a spam fighting measure, where the effort is the price paid by an email sender for demanding the recipient's attention, they now form one of the cornerstones of crypto-currencies.

Bitcoin[3] uses hashcash[1] as proof of work for new blocks of transactions, which requires finding a nonce value such that twofold application of the cryptographic hash function SHA256 to this nonce (and the rest of the block header) results in a number with many leading 0s. The bitcoin protocol dynamically adjust this "difficulty" number so as to maintain a 10-minute average block interval. Starting out at 32 leading zeroes in 2009, the number has steadily climbed and is currently at 63, representing an incredible $2^{63}/10$ double-hashes per minute. This exponential growth of hashing power is enabled by the highly-parallellizable nature of the hashcash proof of work. which saw desktop cpus out-performed by graphics-cards, these in turn by programmable gate arrays, and finally by custom designed chips (ASICs).

Downsides of this development include high investment costs, rapid obsolesence, centralization of mining power, and "waste" of energy. This has led people to look for alternative proofs of work that lack parallelizability, aiming to keep commodity hardware competitive.

Litecoin replaces the sha256 hash function in hashcash by a single round, 128KB version of the *scrypt* key derivation function. Technically, this is no longer a proof of work system, as verification takes a nontrivial amount of computation. Even so, GPUs are a least an order of magnitude faster than CPUs for Litecoin mining, and ASICs are coming on to the market in 2014.

Primecoin [2] is an interesting design based on finding long Cunningham chains of prime numbers, using a two-step process of filtering candidates by *sieving*, and applying pseudo-primality tests to remaining candidates. The most efficient implementations are still CPU based. A downside to primecoin is that its use of memory is not constrained much.

2 Memory latency; the great equalizer

While cpu-speed and memory bandwidth are highly variable across time and architectures, main memory latencies have remained relatively stable. To level the mining playing field, a proof of work system should not merely be memory bound-rather than CPU bound; it should have the following additional properties:

scalable The amount of memory needed should be a parameter that can scale arbitrarily.

linear The number of memory accesses should be linear in the amount—on average, each memory cell is accessed a constant number of time

tmto-hard It should not allow any time-memory trade-off—using only half as much memory should incur several orders of magnitude slowdown.

no-locality The pattern of memory accesses should be sufficiently random as to make caches useless.

sequential Results should to a large extent depend on a proper ordering of memory accesses.

cpu-easy the total running time should be dominated by waiting for memory accesses

simple The algorithm should be sufficiently simple that one can be convinced of its optimality.

Combined, these properties ensure that the proof of work system is almost entirely constrained by main memory latency and scales appropriately for any application.

We introduce the first proof of work system satisfying these properties. Amazingly, it amounts to little more than enumerating nonces and storing them in a hashtable. All hashtables break down when trying to store more items than it was designed to handle. For one particular hash table design, this very break down is of a special form that can be turned into a concise and quickly verifiable proof. Enter the cuckoo hash table.

3 Cuckoo hashing

Introduced by Rasmus Pagh and Flemming Friche Rodler in 2001[4], a cuckoo hash table consists of two same sized hash tables with different hash functions, providing two possible locations for each key (and constant lookup time). Upon insertion of a new key, if both locations are already taken, then one is kicked out and inserted in its alternate location, possibly displacing yet another key, repeating the process until either a vacant location is found, or some maximum number of iterations, is reached. The latter can only happen once cycles have formed in the *Cuckoo graph*. This is a bipartite graph with a node for each location and an edge for every key, connecting the two locations it can reside at. This naturally suggests a proof of work problem, which we now formally define.

4 The proof of work function

Fix three parameters $L \leq E \leq N$ in the range $\{4, ..., 2^{32}\}$, which denote the cycle length, number of edges (also easyness, opposite of difficulty), and the number of nodes, resprectively. L and N must be even. Function cuckoo maps any binary string h (the header) to a bipartite graph $G = (V_0 \cup V_1, E)$, where V_0 is the set of integers modulo $N_0 = N/2 + 1$, V_1 is the set of integers modulo $N_1 = N/2 - 1$, and E has an edge between SHA256(h n) mod N_0 in V_0 and SHA256(h n) mod N_1 in V_1 for every nonce $0 \leq n < E$. A proof for G is a subset of L nonces whose corresponding edges form an L-cycle in G.

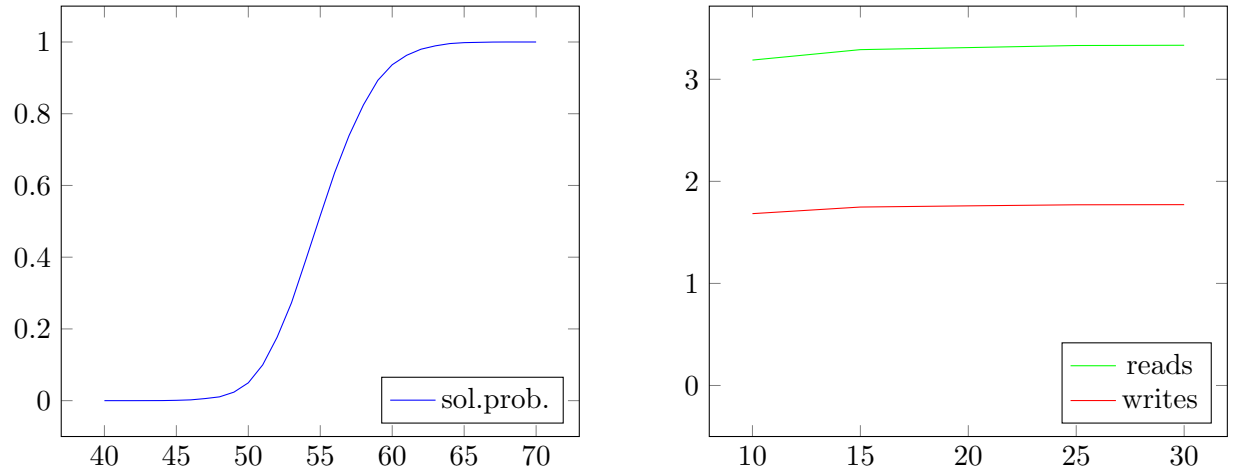
5 Cycle formation

A set of stored keys gives rise to a *directed* cuckoo graph, where each stored key is a directed edge from the location where it resides to its alternate location. The outdegree of every node in this graph is either 0 or 1, When there are no cycles yet, the graph is a *forest*, a disjoint union of trees. In each tree, all edges are directed, directly, or indirectly, to its root, the only node in the tree with outdegree 0. Addition of a new key causes a cycle if and only if its two locations are in the same tree, testable by following the path each locations to its root. In case of equal roots, we can compute the length of the resulting cycle as 1 plus the sum of the path-lengths to the node where the two paths first join. If the cycle length is not L, we keep the graph acyclic by not storing the new key. There is some probability

of overlooking other L-cycles that uses that key, but in the important low easyness case of having few cycles in the cuckoo graph to begin with, it does not significantly affect the rate of solution finding.

6 Implementation and performance

The C-program listed in the Appendix is also available online at `https://github.com/tromp/cuckoo` together with a Makefile, proof verifier and this paper. 'make test' tests everything. The main program uses 31 bits per node to represent the directed cuckoo graph, reserving the most significant bit for marking edges on a cycle, to simplify recovery of the proof nonces. On my 3.2GHz Intel Core i5, in case no solution is found, size 2^{20} takes 4MB and 0.25s, size 2^{25} takes 128MB and 10s, and size 2^{30} takes 4GB and 400s, a significant fraction of which is spend pointer chasing.



The left plot above shows the probability of finding a 42-cycle as a function of the percentage edges/nodes, as determined from 10000 runs at size 2^{20}. The right plot shows the average number of memory reads and writes per edge as function of log size, as determined from 1000 runs each. It is almost constant at 3.3 reads and 1.75 writes.

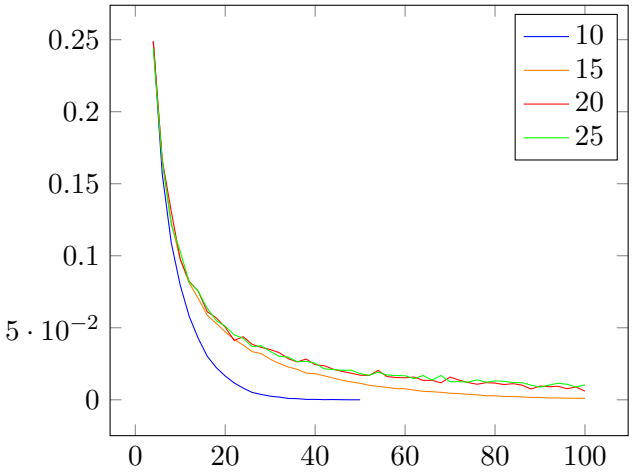
7 Memory-hardness

I conjecture that this problem doesn't allow for a time-memory trade-off. If one were to store only a fraction p of V_0 and V_1, then one would have to reject a fraction p^2 of generated edges, drastically reducing the odds of finding cycles for $p < 1/\sqrt{2}$ (the reduction being exponential in cycle length). There is one obvious trade-off in the other direction. By doubling the memory used, nonces can be stored alongside the directed edges, which would save the effort of recovering them in the current slow manner. The speedup falls far short of a factor 2 though, so a better use of that memory would be to run another copy in parallel.

8 Choice of cycle length

Extremely small cycle lengths risk the feasability of alternative datastructures that are more memory-efficient. For example, for $L = 2$ the problem reduces to finding a birthday collision, for which a Bloom filter would be very effective, as would Rainbow tables. It seems however that the Cuckoo representation might be optimal even for $L = 4$. Such small values still harm the TMTO resistance though as mentioned in the previous paragraph. In order to keep proof size manageable, the cycle

length should not be too large either. We consider 24-64 to be a healthy range, and 42 a nice number close to the middle of that range. The plot below shows the distribution of cycle lengths found for sizes $2^{10}, 2^{15}, 2^{20}, 2^{25}$, as determined from 100000,100000,10000, and 10000 runs respectively. The tails of the distributions beyond $L = 100$ are not shown. For reference, the longest cycle found was of length 1726.



9 Scaling memory beyond 16-32 GB

While the current algorithm can accomodate up to $N = 2^{33} - 2$ nodes by a simple change in implementation, a different idea is needed to scale beyond that. To that end, we propose to use K-partite graphs with edges only between partition k and partition $(k + 1)$ mod K, where k is fed into the hash function along with the header and nonce. With each partition consisting of at most $2^31 - 1$ nodes, the most significant bit is then available to distinguish edges to the two neighbouring partitions. The partition sizes should remain relatively prime, e.g. by picking the largest K primes under 2^{31}.

References

[1] Adam Back. Hashcash - a denial of service counter-measure. Technical report, August 2002.

[2] Sunny King. Primecoin: Cryptocurrency with prime number proof-of-work. July 2013.

[3] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[4] Rasmus Pagh, Ny Munkegade Bldg, Dk-Arhus C, and Flemming Friche Rodler. Cuckoo hashing, 2001.

10 Appendix A: cuckoo.c Source Code

```
// Cuckoo Cycle, a memory-hard proof-of-work
// Copyright (c) 2013-2014 John Tromp

#include "cuckoo.h"
// algorithm parameters
#define MAXPATHLEN 8192

// used to simplify nonce recovery
```

```
#define CYCLE 0x80000000
int cuckoo[1+SIZE]; // global; conveniently initialized to zero

int main(int argc, char **argv) {
  // 6 largest sizes 131 928 529 330 729 132 not implemented
  assert(SIZE < (unsigned)CYCLE);
  char *header = argc >= 2 ? argv[1] : "";
  printf("Looking for %d-cycle on cuckoo%d%d(\"%s\") with %d edges\n",
              PROOFSIZE, SIZEMULT, SIZESHIFT, header, EASYNESS);
  int us[MAXPATHLEN], nu, u, vs[MAXPATHLEN], nv, v;
  for (int nonce = 0; nonce < EASYNESS; nonce++) {
    sha256edge(header, nonce, us, vs);
    if ((u = cuckoo[*us]) == *vs || (v = cuckoo[*vs]) == *us)
      continue; // ignore duplicate edges
    for (nu = 0; u; u = cuckoo[u]) {
      assert(nu < MAXPATHLEN-1);
      us[++nu] = u;
    }
    for (nv = 0; v; v = cuckoo[v]) {
      assert(nv < MAXPATHLEN-1);
      vs[++nv] = v;
    }
#ifdef SHOW
    for (int j=1; j<=SIZE; j++)
      if (!cuckoo[j]) printf("%2d:   ",j);
      else            printf("%2d:%02d ",j,cuckoo[j]);
    printf(" %x (%d,%d)\n", nonce,*us,*vs);
#endif
    if (us[nu] == vs[nv]) {
      int min = nu < nv ? nu : nv;
      for (nu -= min, nv -= min; us[nu] != vs[nv]; nu++, nv++) ;
      int len = nu + nv + 1;
      printf("% 4d-cycle found at %d%%\n", len, (int)(nonce*100L/EASYNESS));
      if (len != PROOFSIZE)
        continue;
      while (nu--)
        cuckoo[us[nu]] = CYCLE | us[nu+1];
      while (nv--)
        cuckoo[vs[nv+1]] = CYCLE | vs[nv];
      for (cuckoo[*vs] = CYCLE | *us; len ; nonce--) {
        sha256edge(header, nonce, &u, &v);
        int c;
        if (cuckoo[c=u] == (CYCLE|v) || cuckoo[c=v] == (CYCLE|u)) {
          printf("%2d %08x (%d,%d)\n", --len, nonce, u, v);
          cuckoo[c] &= ~CYCLE;
        }
      }
      break;
    }
    while (nu--)
      cuckoo[us[nu+1]] = us[nu];
    cuckoo[*us] = *vs;
  }
  return 0;
}
```

11 Appendix B: cuckoo.h Header File

```
// Cuckoo Cycle, a memory-hard proof-of-work
```

```
// Copyright (c) 2013-2014 John Tromp

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <assert.h>
#include <openssl/sha.h>

// proof-of-work parameters
#ifndef SIZEMULT
#define SIZEMULT 1
#endif
#ifndef SIZESHIFT
#define SIZESHIFT 20
#endif
#ifndef EASYNESS
#define EASYNESS (SIZE/2)
#endif
#ifndef PROOFSIZE
#define PROOFSIZE 42
#endif

#define SIZE (SIZEMULT*(1<<SIZESHIFT))
// relatively prime partition sizes
#define PARTU (SIZE/2+1)
#define PARTV (SIZE/2-1)

// generate edge in cuckoo graph from hash(header++nonce)
void sha256edge(char *header, int nonce, int *pu, int *pv) {
  uint32_t hash[8];
  SHA256_CTX sha256;
  SHA256_Init(&sha256);
  SHA256_Update(&sha256, header, strlen(header));
  SHA256_Update(&sha256, &nonce, sizeof(nonce));
  SHA256_Final((unsigned char *)hash, &sha256);
  uint64_t u64 = 0, v64 = 0;
  for (int i = 8; i--; ) {
    u64 = ((u64<<32) + hash[i]) % PARTU;
    v64 = ((v64<<32) + hash[i]) % PARTV;
  }
  *pu = 1 +         (int)u64;
  *pv = 1 + PARTU + (int)v64;
}
```