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

John Tromp

January 11, 2014

1 Introduction

A "proof of work" (PoW) system allows a verifier, such as an email recipient, to check (with negligable effort) that a prover, like the sender of the email, has expended a large amount of computational effort. This effort is the price paid for demanding the recipient's attention.

In cryptocurrencies, a global peer-to-peer network of "miners" use proof of work to create concensus on the state of the public transaction ledger, for which they are awarded new coins (and transaction fees).

Bitcoin[1] uses hashcash-SHA256[2] 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. The current value of 62 thus implies that on the order of $2^{62}/600$ double-hashes are performed every second, enabled by the highly-parallellizable nature of the hashcash proof of work. This saw desktop cpus out-performed by graphics-cards, these in turn by programmable gate arrays, and finally by custom designed chips (ASICs).

We propose a new proof of work that is constrained by memory-latency, and thus inherently not parallellizable. The basic idea is to have the prover store all the nonces in a data structure known as a hashtable, which combines efficient memory usage with (near) constant lookup time. As the load (ratio of stored items to available slots) becomes too high, some condition should arise that can be used as a small and easily checkable proof. Enter the cuckoo hash table.

2 Cuckoo hashing

Introduced by Rasmus Pagh and Flemming Friche Rodler in 2001[2], 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.

3 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.

4 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.

5 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, size 2^{20} takes less than a second, size 2^{25} takes about 30s, and size 2^{30} about 1000s, a significant fraction of which is spend pointer chasing. Memory usage in these cases is 4MB, 128MB, and 4GB. Figure 1 shows the probability of finding a 42-cycle as empirically determined from 10000 runs at size $1 \cdot 2^{20}$.

6 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 $p < 1/\sqrt{2}$. 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.

References

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

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

7 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 65536

// 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);
      us[++nu] = u;
    }
    for (nv = 0; v; v = cuckoo[v]) {
      assert(nv < MAXPATHLEN);
      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);
        if (cuckoo[u] == (CYCLE|v) || cuckoo[v] == (CYCLE|u))
          printf("%2d %08x (%d,%d)\n", --len, nonce, u, v);
      }
      break;
    }
    while (nu--)
      cuckoo[us[nu+1]] = us[nu];
    cuckoo[*us] = *vs;
  }
  return 0;
}
```

8 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;
}
```

9 Appendix C: plots

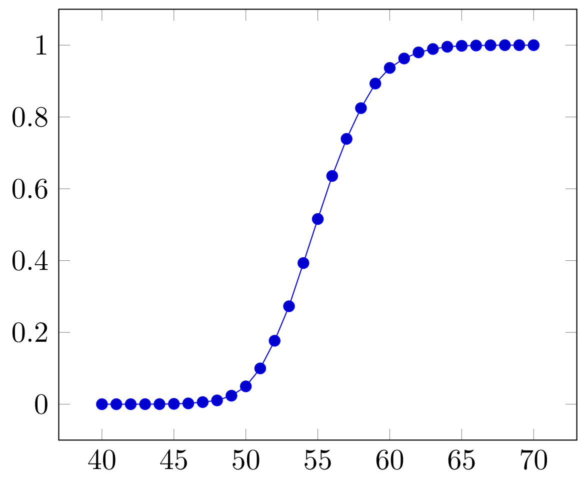

Figure 1: Solution probability as function of percentage edges/nodes