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

John Tromp

February 3, 2014

Abstract

We introduce the first trivially verifiable, scalable, memory-and-tmto-hard proof-of-work system.

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[1] uses hashcash[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. 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 (GPUs), these in turn by field-programmable gate arrays (FPGAs), and finally by custom designed chips (ASICs).

Downsides of this development include high investment costs, rapid obsolesence, centralization of mining power, and large power consumption. This has led people to look for alternative proof-of-work systems that resist parallelizability, aiming to keep commodity hardware competitive.

Litecoin replaces the SHA256 hash function in hashcash by a single round version of the *scrypt* key derivation function. Its memory requirement of 128KB is a compromise between computation-hardness for the prover and verification efficiency for the verifier. Although designed to be GPU-resistant, GPUs are now a least an order of magnitude faster than CPUs for Litecoin mining, and ASICs are have appeared on the market in early 2014.

Primecoin [3] 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.

Adam Back [4] has a good overview of proof-of-work papers past and present.

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. This suggests making the proof-of-work system latency-bound to level the mining playing field. Ideally, it should have the following properties:

verify-trivial A proof can be checked in microseconds rather than milliseconds.

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

linear The number of computational steps and the number of memory accesses is linear in the amount of memory.

tmto-hard There is no time-memory trade-off; using only half as much memory should incur several orders of magnitude slowdown.

random-access RAM is accessed randomly, making bandwidth and caches irrelevant.

parallel-hard Memory accesses cannot be effectively parallelized.

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

Combined, these properties ensure that a proof-of-work system is entirely constrained by main memory latency[1] and scales appropriately for any application. The single biggest advantage of a memory-and-latency-hard proof-of-work is *sustainability*—miner costs should be dominated by investment in low-power general purpose hardware rather than investment in single purpose hardware coupled with high operational costs.

We introduce the very first proof-of-work system that is both verify-trivial and tmto-hard. Furthermore, it satisfies all other properties, except for being parallel-resistant at best. Amazingly, it amounts to little more than enumerating nonces and storing them in a hashtable. While all hashtables break down when trying to store more items than it was designed to handle, in one hashtable design in particular this breakdown is of a special nature that can be turned into a concise and easily verified proof. Enter the cuckoo hashtable.

3 Cuckoo hashing

Introduced by Rasmus Pagh and Flemming Friche Rodler[5], a cuckoo hashtable consists of two same-sized tables each with its own hash function mapping a key to a table location, providing two possible locations for each key. Upon insertion of a new key, if both locations are already occupied by keys, 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.

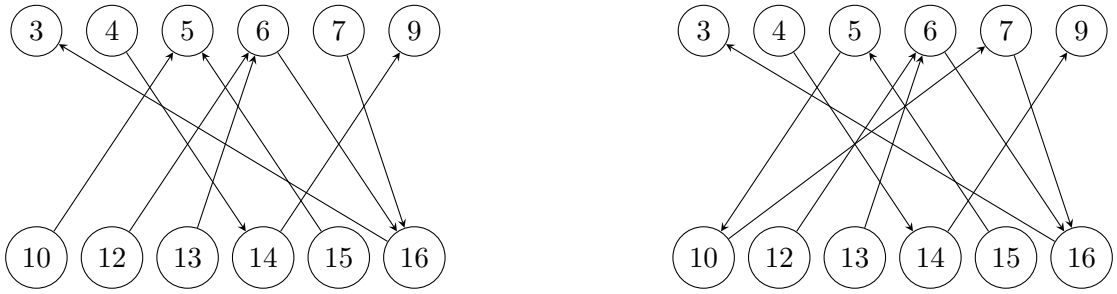
4 The proof-of-work function

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 $\text{hash}(h, n) \bmod N_0$ in V_0 and $\text{hash}(h, n) \bmod N_1$ 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.

[1] The memory requirement should exceed that of the largest available single-chip memory to enforce chip-to-chip latencies.

5 Solving a proof-of-work problem

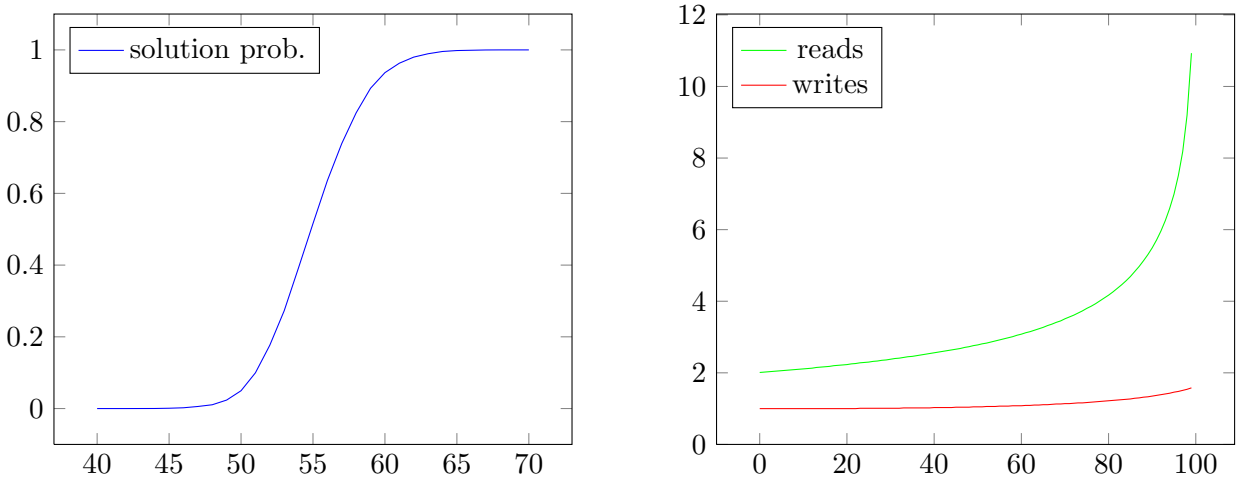
We enumerate the E nonces, but instead of storing the nonce itself as a key in the Cuckoo hashtable, we store the alternate key location at the key location, and forget about the nonce. We thus maintain the *directed* cuckoo graph, in which the edge for a key is directed from the location where it resides to its alternate location. Moving a key to its alternate location thus corresponds to reversing its edge. 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. Initially there are just N singleton trees consisting of individual nodes which are all roots. Addition of a new key causes a cycle if and only if its two endpoints are nodes in the same tree, which we can test by following the path from each endpoint to its root. In case of different roots, we reverse all edges on the shorter of the two paths, and finally create the edge for the new key itself, thereby joining the two trees into one. The left diagram below shows the directed cuckoo graph for header 'header' on $N = 9 + 7$ nodes after adding edges $(3, 16), (5, 15), (6, 16), (7, 16), (5, 10), (6, 13), (6, 12), (9, 14)$ and $(4, 14)$ (nodes with no incident edges are omitted for clarity). In order to add the 10th edge $(7, 10)$, we follow the paths $7 \to 16 \to 3$ and $10 \to 5$ to find different roots 3 and 5. Since the latter path is shorter, we reverse it to $5 \to 10$ so we can add the new edge as $(10 \to 7)$, resulting in the right diagram.



When adding the 11th edge $(3, 15)$, we find the singleton path 3 and the path $15 \to 5 \to 10 \to 7 \to 16 \to 3$ with equal roots. In this case, 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. In the diagram, the paths first join at the root, and the cycle length is computed as $1 + 0 + 5 = 6$. If the cycle length equals L, then we solved the problem, and recover the proof by enumerating nonces once more and checking which ones formed the cycle. If not, then we keep the graph acyclic by ignoring the edge. There is some probability of overlooking other L-cycles through that edge, but in the important low easiness case of having few cycles in the cuckoo graph to begin with, it hardly 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. 'make example' reproduces the example shown in the diagrams. 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 0.025s, size 2^{25} takes 128MB and 2.5s, and size 2^{30} takes 4GB and 136s, most of which is spent on memory latency.



The left plot above shows the probability of finding a 42-cycle as a function of the percentage edges/nodes, while the right plot shows the average number of memory reads and writes per edge as a function of the percentage nonce/easiness (progress through main loop). Both were determined from 10000 runs at size 2^{20}; results at size 2^{25} look almost identical. In total the program averages 3.3 reads and 1.75 writes per edge.

7 Additional difficulty control

At very low easiness settings, it's difficult to determine the solution probability with much precision. If desired, one can instead impose an additional constraint on the L-cycle, namely that the sum of its nonces modulo E be less than some difficulty threshold D. This reduces the success probability by exactly E/D.

8 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.

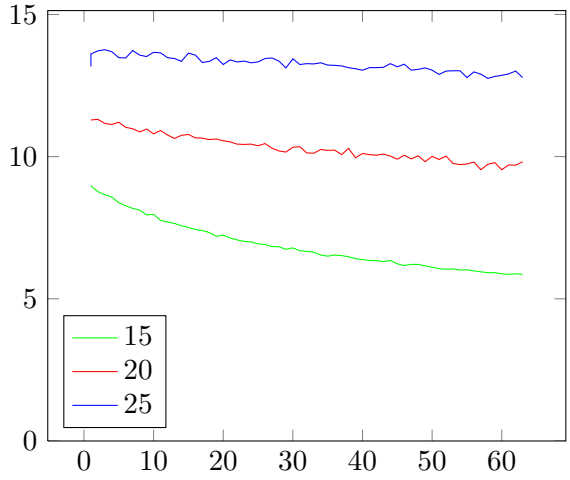
9 Parallelizability

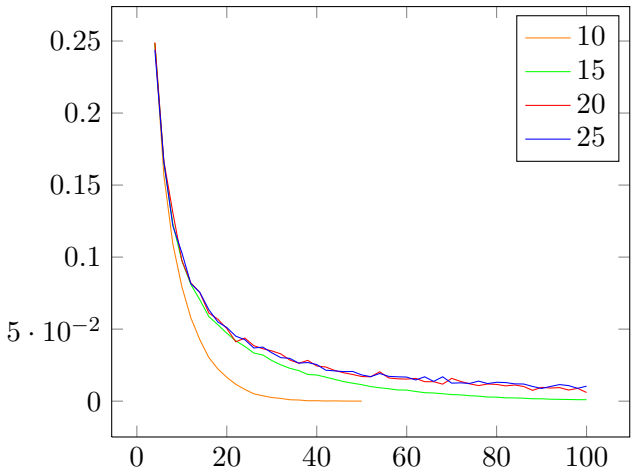
To parallellize the program one could run P processing elements (PE) connected to at least P memory elements (ME) by some multistage interconnection network. For $0 \leq p < P$, PE p processes all nonces $p \bmod P$.

Note that unless one uses at least P^2 MEs, memory routing conflicts will frequently arise. The possibility of development of custom hardware for improved parallel random access to main need not be a big concern, as this is likely to benefit many other applications as well.

Parallelization also presents algorithmic challenges. Paths from an edge's two endpoints are not well-defined when other edge additions and path reversals are still in progress. One example of such

a path conflict is the addition of edges $(7, 10)$ and $(3, 15)$ in the example diagrams. If these were to happen in parallel, then the path from 15 will likely end at 5 because edge $(10 \rightarrow 5)$ hasn't been reversed yet. As a result, $(3 \rightarrow 15)$ will be added without realizing it creates a cyle. Thus, in a parallel implementation, path following cannot be assumed to terminate, creating further complications, like the need for cycle detection algorithm such as [6]. Simply ignoring the formed cycle, by aborting paths whose length exceed some threshold, means that no cycles can ever be found containing nodes that lead to the cycle. The left plot below shows the average number of times that either of the two roots for a nonce occurred a given number of nonces ago, which suggests that there are potentially about $10P$ such conflicts. Analysing how these conflicts affect performance and the probabillity of finding cycles is a topic for further research.





10 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 right plot above 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.

11 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 $(k + 1) \bmod K$, 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}.

12 Conclusion

Cuckoo Cycle is an elegant proof-of-work design emphasizing memory latency. Its parallelizability may be limited to that of multiprocessor-memory interconnection networks in general, offering hope of leveling the mining playing field through use of commodity hardware. Future research is needed to quantify the behaviour of multiprocessor implementations.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," May 2009. [Online]. Available: http://www.bitcoin.org/bitcoin.pdf

[2] A. Back, "Hashcash - a denial of service counter-measure," Tech. Rep., Aug. 2002, (implementation released in mar 1997).

[3] S. King, "Primecoin: Cryptocurrency with prime number proof-of-work," Jul. 2013. [Online]. Available: http://primecoin.org/static/primecoin-paper.pdf

[4] A. Back, "Hashcash.org," Feb. 2014. [Online]. Available: http://www.hashcash.org/papers/

[5] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jalgor.2003.12.002

[6] R. P. Brent, "An improved Monte Carlo factorization algorithm," *BIT*, vol. 20, pp. 176–184, 1980.

13 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] : "";
  setheader(header);
  printf("Looking for %d-cycle on cuckoo%d%d(\"%s\") with %d edges\n",
              PROOFSIZE, SIZEMULT, SIZESHIFT, header, EASINESS);
  int us[MAXPATHLEN], nu, u, vs[MAXPATHLEN], nv, v;
  for (int nonce = 0; nonce < EASINESS; nonce++) {
    sipedge(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/EASINESS));
      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--) {
        sipedge(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;
    }
    if (nu < nv) {
      while (nu--)
        cuckoo[us[nu+1]] = us[nu];
      cuckoo[*us] = *vs;
    } else {
      while (nv--)
        cuckoo[vs[nv+1]] = vs[nv];
      cuckoo[*vs] = *us;
    }
  }
  return 0;
}
```

14 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 EASINESS
#define EASINESS (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)

typedef uint64_t u64;

#define ROTL(x,b) (u64)( ((x) << (b)) | ( (x) >> (64 - (b))) )

#define SIPROUND \
  do { \
    v0 += v1; v1=ROTL(v1,13); v1 ^= v0; v0=ROTL(v0,32); \
    v2 += v3; v3=ROTL(v3,16); v3 ^= v2; \
    v0 += v3; v3=ROTL(v3,21); v3 ^= v0; \
    v2 += v1; v1=ROTL(v1,17); v1 ^= v2; v2=ROTL(v2,32); \
  } while(0)

// SipHash-2-4 specialized to precomputed key and 4 byte nonces
u64 siphash24( int nonce, u64 v0, u64 v1, u64 v2, u64 v3) {
  u64 b = ( ( u64 )4 ) << 56 | nonce;
  v3 ^= b;
  SIPROUND; SIPROUND;
  v0 ^= b;
  v2 ^= 0xff;
  SIPROUND; SIPROUND; SIPROUND; SIPROUND;
  return v0 ^ v1 ^ v2  ^ v3;
}

u64 v0 = 0x736f6d6570736575ULL, v1 = 0x646f72616e646f6dULL,
    v2 = 0x6c7967656e657261ULL, v3 = 0x7465646279746573ULL;

#define U8TO64_LE(p) \
  (((u64)((p)[0])      ) | ((u64)((p)[1]) <<  8) | \
   ((u64)((p)[2]) << 16) | ((u64)((p)[3]) << 24) | \
   ((u64)((p)[4]) << 32) | ((u64)((p)[5]) << 40) | \
   ((u64)((p)[6]) << 48) | ((u64)((p)[7]) << 56))

// derive siphash key from header
void setheader(const char *header) {
  unsigned char hdrkey[32];
  SHA256((unsigned char *)header, strlen(header), hdrkey);
  u64 k0 = U8TO64_LE( hdrkey ); u64 k1 = U8TO64_LE( hdrkey + 8 );
  v3 ^= k1; v2 ^= k0; v1 ^= k1; v0 ^= k0;
}

// generate edge in cuckoo graph
void sipedge(int nonce, int *pu, int *pv) {
  u64 sip = siphash24(nonce, v0, v1, v2, v3);
  *pu = 1 +         (int)(sip % PARTU);
  *pv = 1 + PARTU + (int)(sip % PARTV);
}
```