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

John Tromp

March 19, 2014

Abstract

We introduce the first trivially verifiable, scalable and tmto[1]-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 64, representing an incredible $2^{64}/10$ 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 and require a nontrivial amount of memory, 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 at least an order of magnitude faster than CPUs for Litecoin mining, and ASICs 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. Downsides to this proof-of-work are its complexity and the scope for algorithmic improvements that may be kept private. Its memory requirements, while larger than Litecoin's, are still modest.

Momentum [4] proposes finding birthday collisions of hash outputs as proof-of-work, the simplest way to combine scalable memory usage with trivial verifiability. Its memory requirements are not very strict though. as Bloom filters or rainbow tables can identify collisions, and parallellizes well.

Adam Back [5] has a good overview of proof-of-work papers past and present.

---

[1]time-memory trade-off

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 Amount of computational steps and memory accesses are linear in the amount of memory.

tmto-hard 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 No speedup is possible for parallel implementations.

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[2] and scales appropriately for any application.

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 parallel-hardness (see the later section on parallelizability). 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[6], 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 an (even) number of nodes $8 \leq N < 2^{32}$, a number of edges $N/4 < E \leq N$, and an (even) cycle length $4 \leq L < 256$. Function cuckoo maps any 128-bit key k (the header digest) 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 $\mathrm{siphash}(k, n) \bmod N_0$ in V_0 and $\mathrm{siphash}(k, n) \bmod N_1$ 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.

---

[2]The memory requirement should exceed that of the largest available single-chip memory to enforce off-chip latencies. It should also be a significant fraction of the typical memory of a botnet computer, as sending the machine into swap-hell is likely to alert its owner.

5 Solving the 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 $7 \rightarrow 16 \rightarrow 3$ and $10 \rightarrow 5$ to find different roots 3 and 5. Since the latter path is shorter, we reverse it to $5 \rightarrow 10$ so we can add the new edge as $(10 \rightarrow 7)$, resulting in the right diagram.


When adding the 11th edge $(3, 15)$, we find the singleton path 3 and the path $15 \rightarrow 5 \rightarrow 10 \rightarrow 7 \rightarrow 16 \rightarrow 3$ 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 case of having few cycles in the cuckoo graph to begin with, it hardly affects 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 the latest version of this paper. 'make test' tests everything. 'make example' reproduces the example shown above. The main program uses 32 bits per node to represent the directed cuckoo graph, plus about 64 KB per thread for 2 auxiliary arrays that not only record traversed paths, but also double as a mini cuckoo table used for storing edges of a solution cycle, allowing easy recovery of the proof nonces. The left plot below shows both the total runtime in seconds and the runtime of just the hash computation, as a function of (log)size. The latter is purely linear, while the former is superlinear due to increasing memory latency as the nodes no longer fit in cache. The right plot show this more clearly as the percentage of hashing to total runtime, ending up around 5%.
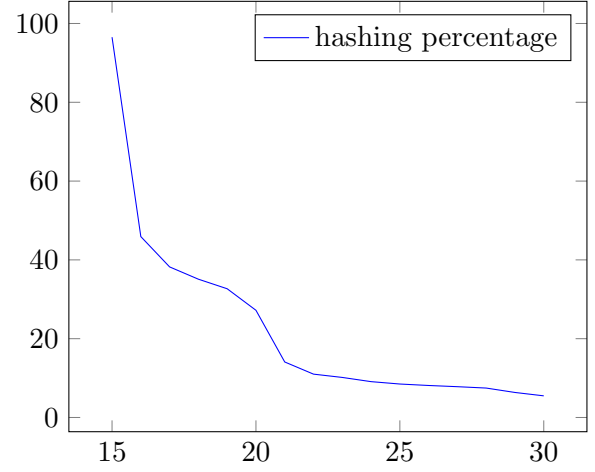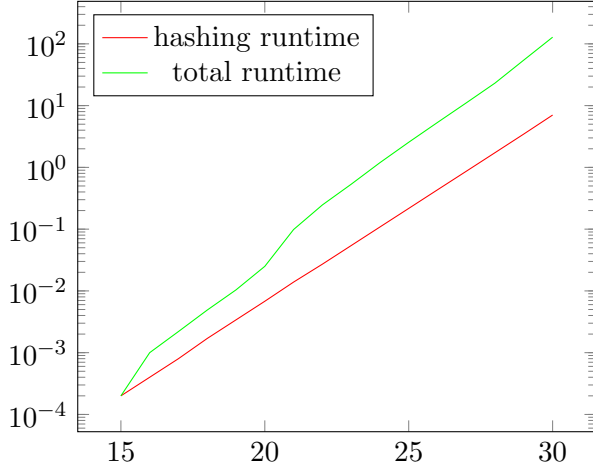
On a 3.2GHz Intel Core i5, size 2^{20} takes 4MB and 0.025s, size 2^{25} takes 128MB and 2.5s, and size 2^{30} takes 4GB and 128s, or roughly half a minute per GB. The left plot below shows the probability of finding a 42-cycle as a function of the percentage edges/nodes (relative easiness), while the 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 1.1 writes per edge.
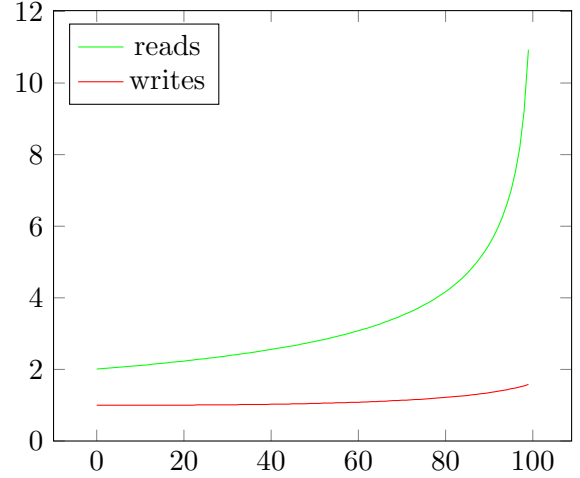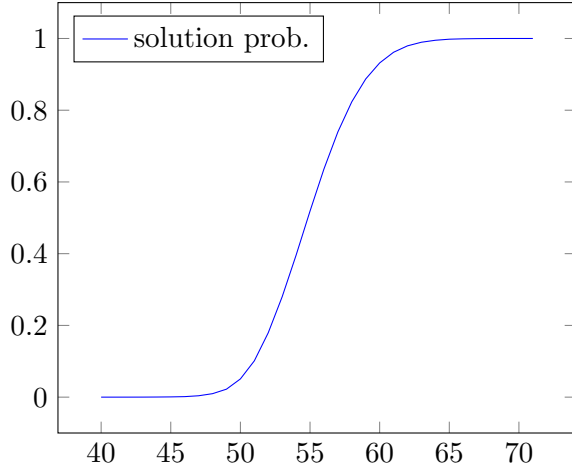

7 Difficulty control

Relative easiness (the ratio E/N) determines a base level of difficulty, which may suffice for applications where difficulty is to remain fixed. The ratio $E/N = 1$ is suitable when a practically guaranteed solution is desired, For crypto currencies, where difficulty must scale in precisely controlled manner across a huge range, adjusting easiness is not suitable. The implementation default $E/N = 1/2$ gives a solution probability of roughly 2.2%, while the average number of cycles found increases slowly with size; from 2 at 2^{20} to 3 at 2^{30}. For further control, a diffculty target $0 < T < 2^{256}$ is introduced, and we impose the additional constraint that the sha256 digest of the cycle nonces in ascending order be less than T, thus reducing the success probability by a factor $2^{256}/T$.

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 only applies to solution finding runs though, so a better use of that memory would be to run another copy in parallel.

9   Parallelization

The implementation allows the number of threads to be set with -DNTHREADS. For $0 \leq t < T$, thread t processes all nonces $t \bmod T$. Parallelization presents some 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 check for duplicate edges yielding a false negative, if in between checking the two endpoints, another thread reverses a path through those nodes. Another 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, not realizing it creates a cyle. Thus, in a parallel implementation, path following can no longer be assumed to terminate. Instead of using a cycle detection algorithm such as [7], our implementation notices when the path length exceeds MAXPATHLEN, and reports wether this is due to a path conflict. 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, showing that there are potentially about $10T$ such conflicts. Despite these conflicts, and the complete lack of synchronization between threads (apart from a solution recording mutex), the cuckoo array at any time represents some, possibly cyclic, directed cuckoo graph on a subset of proccessed nonces. The right plot shows in green the speedup achieved by up to 40 threads on a dual Intel Xeon E5-2690 at 3GHz with 10 hyperthreaded cores each. Each datapoint represents 10 runs on 2^{30} nodes with headers "0" through "9". Although the speedup is sublinear at 26 for 40 threads, there are still clear gains to be made by using more threads. Attempting to predict what happens with more than 40 threads, we instead made each thread generate memory requests faster by reducing the number of rounds in siphash. This produced some unexpected results which we're not sure how to explain, but show for completeness anyway. On 2^{20} nodes, aborts happened 0.03% of the time with 2 threads, 0.1% with 4 threads, 0.27% with 8 threars, and 0.2% with 12 threads. They overlooked 0.13%, 0.31%, 0.57%, and 0.17%, respectiverly, of the 42-cycles found by the single-threaded runs. Empirically then, path conflicts have negligable effect on multi-threaded performance. The percentage of hashing to total runtime shows the same behaviour as in the single-threaded case, with a dual Xeon X5670 machine spending only 5.5% of wallclock time on hashing with 2^{28} nodes.
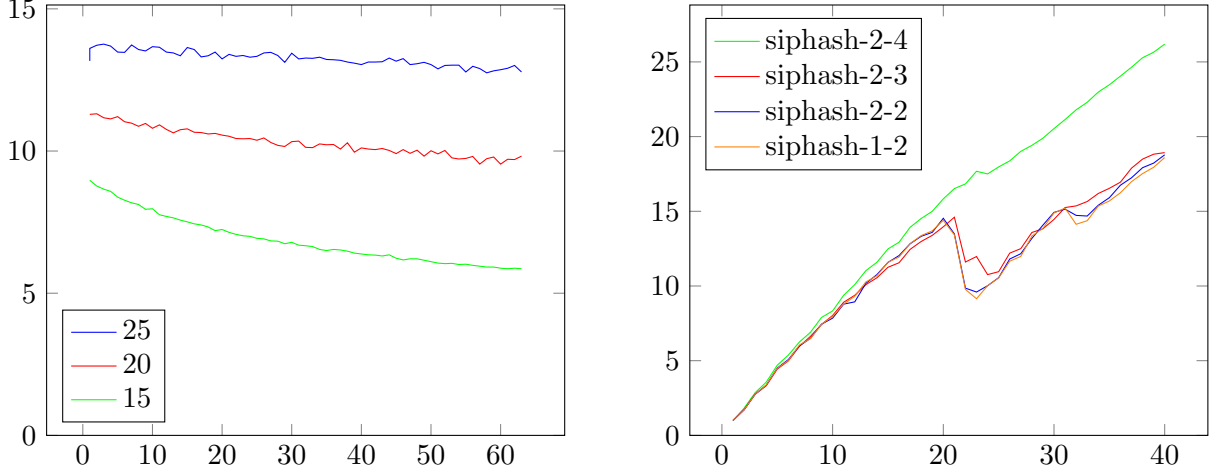
10 Choice of cycle length

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 as in the Momentum proof-of-work. It is conceivable however that the Cuckoo representation is already optimal for $L = 4$. Such small values still harm the TMTO resistance though, as mentioned in the previous paragraph, and may reduce parallelization resistance. In order to keep proof size manageable, the cycle length should not be too large either. We consider 20-64 to be a healthy range, which averages to 42. 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 2120.
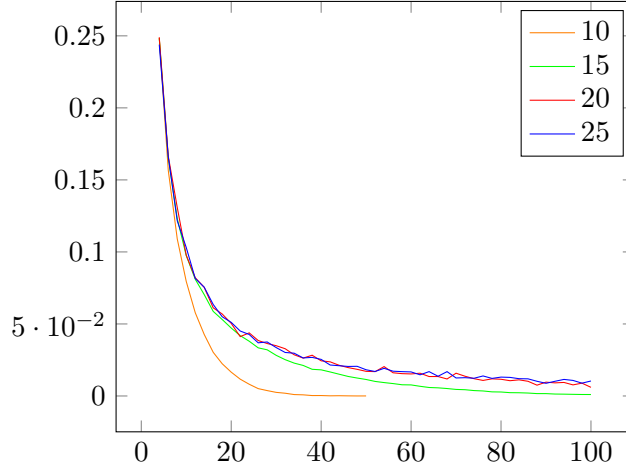
11 Scaling memory beyond 16-32 GB

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. Setting

the size of partition 0 to be relatively prime to $2(K-1)$, and making each successive partition smaller by 2, ensures that neighbouring partition sizes are relatively prime.

12   Conclusion

Cuckoo Cycle is an elegant proof-of-work design emphasizing memory latency over computation. This promotes investment in low-power general purpose hardware (RAM) rather than investment in single purpose hardware coupled with high operational costs, making mining more sustainable. More research is needed to determine the limits of parallelizability.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 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," Tech. Rep., Jul. 2013. [Online]. Available: http://primecoin.org/static/primecoin-paper.pdf

[4] D. Larimer, "Momentum - a memory-hard proof-of-work via finding birthday collisions," Tech. Rep., Oct. 2013. [Online]. Available: http://invictus-innovations.com/s/MomentumProofOfWork-hok9.pdf

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

[6] 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

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

13   Appendix A: cuckoo.h

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

#include <stdint.h>
#include <string.h>
#include <openssl/sha.h> // if openssl absent, use #include "sha256.c"

// proof-of-work parameters
#ifndef SIZEMULT
#define SIZEMULT 1
#endif
#ifndef SIZESHIFT
#define SIZESHIFT 20
#endif
#ifndef PROOFSIZE
#define PROOFSIZE 42
#endif

#define SIZE (SIZEMULT*((unsigned)1<<SIZESHIFT))
```

```c
// relatively prime partition sizes, assuming SIZESHIFT >= 2
#define PARTU (SIZE/2+1)
#define PARTV (SIZE/2−1)


typedef uint64_t u64;
typedef struct {
  u64 v[4];
} siphash_ctx;


#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(siphash_ctx *ctx, char *header) {
  unsigned char hdrkey[32];
  SHA256((unsigned char *)header, strlen(header), hdrkey);
  u64 k0 = U8TO64_LE(hdrkey);
  u64 k1 = U8TO64_LE(hdrkey+8);
  ctx->v[0] = k0 ^ 0x736f6d6570736575ULL;
  ctx->v[1] = k1 ^ 0x646f72616e646f6dULL;
  ctx->v[2] = k0 ^ 0x6c7967656e657261ULL;
  ctx->v[3] = k1 ^ 0x7465646279746573ULL;
}


#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(siphash_ctx *ctx, unsigned nonce) {
  u64 b = ( ( u64 )4 ) << 56 | nonce;
  u64 v0 = ctx->v[0], v1 = ctx->v[1], v2 = ctx->v[2], v3 = ctx->v[3] ^ b;
  SIPROUND; SIPROUND;
  v0 ^= b;
  v2 ^= 0xff;
  SIPROUND; SIPROUND; SIPROUND; SIPROUND;
  return v0 ^ v1 ^ v2 ^ v3;
}


// generate edge in cuckoo graph
void sipedge(siphash_ctx *ctx, unsigned nonce, unsigned *pu, unsigned *pv) {
  u64 sip = siphash24(ctx, nonce);
  *pu = 1 +            (unsigned)(sip % PARTU);
  *pv = 1 + PARTU + (unsigned)(sip % PARTV);
}


// verify that (ascending) nonces, all less than easiness, form a cycle in header−generated graph
int verify(unsigned nonces[PROOFSIZE], char *header, int easiness) {
  siphash_ctx ctx;
  setheader(&ctx, header);
  unsigned us[PROOFSIZE], vs[PROOFSIZE], i = 0, n;
  for (n = 0; n < PROOFSIZE; n++) {
```

```
      if (nonces[n] >= easiness || (n && nonces[n] <= nonces[n-1]))
        return 0;
      sipedge(&ctx, nonces[n], &us[n], &vs[n]);
  }
  do {  // follow cycle until we return to i==0; n edges left to visit
    int j = i;
    for (int k = 0; k < PROOFSIZE; k++) // find unique other j with same vs[j]
      if (k != i && vs[k] == vs[i]) {
        if (j != i)
          return 0;
        j = k;
      }
    if (j == i)
      return 0;
    i = j;
    for (int k = 0; k < PROOFSIZE; k++) // find unique other i with same us[i]
      if (k != j && us[k] == us[j]) {
        if (i != j)
          return 0;
        i = k;
      }
    if (i == j)
      return 0;
    n -= 2;
  } while (i);
  return n == 0;
}
```

14   Appendix B: cuckoo_miner.h

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

#include "cuckoo.h"
#include <stdio.h>
#include <pthread.h>

// algorithm parameters
#define MAXPATHLEN 8192

typedef struct {
  siphash_ctx sip_ctx;
  unsigned easiness;
  unsigned *cuckoo;
  unsigned (*sols)[PROOFSIZE];
  unsigned maxsols;
  unsigned nsols;
  int nthreads;
  pthread_mutex_t setsol;
} cuckoo_ctx;

typedef struct {
  int id;
  pthread_t thread;
  cuckoo_ctx *ctx;
} thread_ctx;

int path(unsigned *cuckoo, unsigned u, unsigned *us) {
  int nu;
  for (nu = 0; u; u = cuckoo[u]) {
```

```c
    if (++nu >= MAXPATHLEN) {
      while (nu-- && us[nu] != u) ;
      if (nu < 0)
        printf("maximum path length exceeded\n");
      else printf("illegal %4d-cycle\n", MAXPATHLEN-nu);
      pthread_exit(NULL);
    }
    us[nu] = u;
  }
  return nu;
}


// largest number of u64's that fit in MAXPATHLEN-PROOFSIZE unsigned's
#define SOLMODU ((MAXPATHLEN-PROOFSIZE)/2)
#define SOLMODV (SOLMODU-1)

void storedge(u64 uv, u64 *usck, u64 *vsck) {
  int j, i = uv % SOLMODU;
  u64 uvi = usck[i];
  if (uvi) {
    if (vsck[j = uv % SOLMODV]) {
      vsck[uvi % SOLMODV] = uvi;
    } else {
      vsck[j] = uv;
      return;
    }
  } else usck[i] = uv;
}

void solution(cuckoo_ctx *ctx, unsigned *us, int nu, unsigned *vs, int nv) {
  u64 *usck = (u64 *)&us[PROOFSIZE], *vsck = (u64 *)&vs[PROOFSIZE];
  unsigned u, v, n;
  for (int i=0; i<SOLMODU; i++)
    usck[i] = vsck[i] = 0L;
  storedge((u64)*us<<32 | *vs, usck, vsck);
  while (nu--)
    storedge((u64)us[(nu+1)&~1]<<32 | us[nu|1], usck, vsck); // u's in even position; v's in odd
  while (nv--)
    storedge((u64)vs[nv|1]<<32 | vs[(nv+1)&~1], usck, vsck); // u's in odd position; v's in even
  pthread_mutex_lock(&ctx->setsol);
  for (unsigned nonce = n = 0; nonce < ctx->easiness; nonce++) {
    sipedge(&ctx->sip_ctx, nonce, &u, &v);
    u64 *c, uv = (u64)u<<32 | v;
    if (*(c = &usck[uv % SOLMODU]) == uv || *(c = &vsck[uv % SOLMODV]) == uv) {
      ctx->sols[ctx->nsols][n++] = nonce;
      *c = 0;
    }
  }
  if (n == PROOFSIZE)
    ctx->nsols++;
  else printf("Only recovered %d nonces\n", n);
  pthread_mutex_unlock(&ctx->setsol);
}

void *worker(void *vp) {
  thread_ctx *tp = (thread_ctx *)vp;
  cuckoo_ctx *ctx = tp->ctx;
  unsigned *cuckoo = ctx->cuckoo;
  unsigned us[MAXPATHLEN], u, vs[MAXPATHLEN], v;
  int nu, nv;
```

```
      for (unsigned nonce = tp->id; nonce < ctx->easiness; nonce += ctx->nthreads) {
        sipedge(&ctx->sip_ctx, nonce, us, vs);
        if ((u = cuckoo[*us]) == *vs || (v = cuckoo[*vs]) == *us)
          continue; // ignore duplicate edges
#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 = path(cuckoo, u, us)] == vs[nv = path(cuckoo, v, vs)]) {
          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:%d%%\n", len, tp->id, (int)(nonce*100L/ctx->easiness));
          if (len == PROOFSIZE && ctx->nsols < ctx->maxsols)
            solution(ctx, us, nu, vs, nv);
          continue;
        }
        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;
        }
      }
    }
    pthread_exit(NULL);
}
```

15   Appendix C: cuckoo_miner.c

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

#include "cuckoo_miner.h"
#include <stdlib.h>
#include <unistd.h>
#include <assert.h>

int main(int argc, char **argv) {
  assert(SIZE < 1L<<32);
  int nthreads = 1;
  int maxsols = 8;
  char *header = "";
  int c, easipct = 50;
  while ((c = getopt(argc, argv, "e:h:m:t:")) != -1) {
    switch (c) {
      case 'e':
        easipct = atoi(optarg);
        break;
      case 'h':
        header = optarg;
        break;
      case 'm':
        maxsols = atoi(optarg);
        break;
      case 't':
```

```
        nthreads = atoi(optarg);
        break;
    }
  }
  assert(easipct >= 0 && easipct <= 100);
  printf("Looking for %d-cycle on cuckoo%d%d(\"%s\") with %d%% edges and %d threads\n",
            PROOFSIZE, SIZEMULT, SIZESHIFT, header, easipct, nthreads);

  cuckoo_ctx ctx;
  setheader(&ctx.sip_ctx, header);
  ctx.easiness = (unsigned)(easipct * (u64)SIZE / 100);
  assert(ctx.cuckoo = calloc(1+SIZE, sizeof(unsigned)));
  assert(ctx.sols = calloc(maxsols, PROOFSIZE*sizeof(unsigned)));
  ctx.maxsols = maxsols;
  ctx.nsols = 0;
  ctx.nthreads = nthreads;
  pthread_mutex_init(&ctx.setsol, NULL);

  thread_ctx *threads = calloc(nthreads, sizeof(thread_ctx));
  assert(threads);
  for (int t = 0; t < nthreads; t++) {
    threads[t].id = t;
    threads[t].ctx = &ctx;
    assert(pthread_create(&threads[t].thread, NULL, worker, (void *)&threads[t]) == 0);
  }
  for (int t = 0; t < nthreads; t++)
    assert(pthread_join(threads[t].thread, NULL) == 0);
  for (int s = 0; s < ctx.nsols; s++) {
    printf("Solution");
    for (int i = 0; i < PROOFSIZE; i++)
      printf(" %x", ctx.sols[s][i]);
    printf("\n");
  }
  return 0;
}
```