# Cuckoo Cycle:
# a graph-theoretic proof-of-work system

John Tromp

April 13, 2014

**Abstract**

We introduce the first graph-theoretic proof-of-work system, based on finding cycles in large random graphs. Such problems are arbitrarily scalable and trivially verifiable. Our implementation uses 1 bit per edge, and up to 1 bit per node. We hypothesize that using less causes superlinear slowdown.

## 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.

As proof-of-work for new blocks of transactions, Bitcoin [1] adopted Adam Back's hashcash [2] proof-of-work. This requires finding a nonce value such that application of a cryptographic hash function (twofold SHA256 in Bitcoin's case) to this nonce (and the rest of the block header) results in a number with many leading 0s. The number of leading 0s is dynamically adjusted by the protocol so as to maintain a certain average block interval (10-minutes for Bitcoin).

Since Bitcoin, many other crypto-currencies have adopted hashcash, with various choices of underlying hash function. the most well-known being *scrypt* as used in Litecoin.

Primecoin [3] introduced the notion of a number-theoretic proof-of-work, as the first alternative to hashcash among crypto-currencies. This requires finding long chains of nearly doubled prime numbers, with a certain relation to the block header. Current algorithms use a two-step process of filtering candidates by *sieving*, and applying pseudo-primality tests to remaining candidates. Unlike hashcash, current algorithms for Primecoin are somewhat complex and involve many trade-offs. Recently, another prime-number based crypto-currency, Riecoin, was introduced, based on finding clusters rather than chains of prime numbers.

## 2 Graph-theoretic proofs-of-work

We propose to base proofs-of-work on finding certain subgraphs in large random graphs. Formally, fix a keyed hash function $h : \{0,1\}^K \times \{0,1\}^W_i \to \{0,1\}^W_o$/footnotehash functions generally have arbitrary length inputs, but here we fix the input width at $W_i$ bits., and a small graph $H$ as a target subgraph. Now pick a large number $N \leq 2^W_o$ as the number of nodes, and $M \leq 2^{W_i-1}$ as the number of edges. Each key $k \in \{0,1\}^K$ generates a graph $G_k = (V, E)$ where $V = \{v_0, \ldots, v_{N-1}\}$, and

$$E = \{(v_{h(k,2i) \bmod N}, v_{h(k,2i+1) \bmod N}) | i \in [0, \ldots, M-1]\} \quad (1)$$

The inputs $i$ $in[0, \ldots, M-1]$ are also called *nonces*. The graph has a *solution* if $H$ occurs as a subgraph. Denote the number of edges in h as $|H|$. A proof of solution is an ordered list of $|H|$ nonces that generate the edges of $H$'s occurance in $G$. Such a proof is verifiable in constant time, independent of $N$ and $M$.

A simple variation generates random bipartite graphs: $G_k = (U \cup V, E)$ where $U = \{u_0, \ldots, u_{\frac{N}{2}-1}\}$, $V = \{v_0, \ldots, v_{\frac{N}{2}-1}\}$, and

$$E = \{(u_{h(k,2i) \bmod \frac{N}{2}}, v_{h(k,2i+1) \bmod \frac{N}{2}}) | i \in [0, \ldots, M-1]\} \tag{2}$$

The expected number of occurances of $H$ as a subgraph of $G$ is a function of both $N$ and $M$, and in many cases it is roughly a function of the ratio $\frac{M}{N}$. For fixed $N$, the function is monotonically increasing in $M$. To make the proof-of-work challenging, one chooses a value of $M$ that yields less than one expected solution (but not much less).

## 3 Cuckoo Cycle

Perhaps the simplest possible choice of subgraph is the *cycle*. The reason for calling the resulting proof-of-work Cuckoo Cycle is that inserting numbers in a Cuckoo hashtable naturally leads to forming cycles in random bipartite graphs!

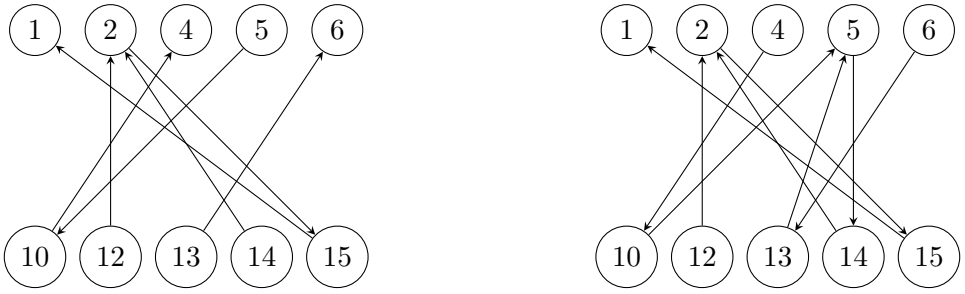
## 4 Cuckoo hashing

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 is bound to 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. It also matches the bipartite graph defined above if the cucko hashtable were based on function $h$. In fact, the insertion procedure suggest a simple algorithm for detecting cycles.

## 5 Cycle detection in Cuckoo hashing

We enumerate the $M$ 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 '39' on $N = 8+8$ nodes after adding edges $(1, 15), (2, 12), (4, 10), (2, 15), (6, 13), (5, 10)$ and $(2, 14)$ (nodes with no incident edges are omitted for clarity). In order to add the 8th edge $(5, 13)$, we follow the paths $5 \to 10 \to 4$ and $13 \to 6$ to find

different roots 4 and 6. Since the latter path is shorter, we reverse it to $6 \to 13$ so we can add the new edge as $(13 \to 5)$. In order to add to 9th edge $(5, 14)$ we now find the path from 5 to be the shorter one, so we reverse that and add the new edge as $(5 \to 14)$, resulting in the right diagram.



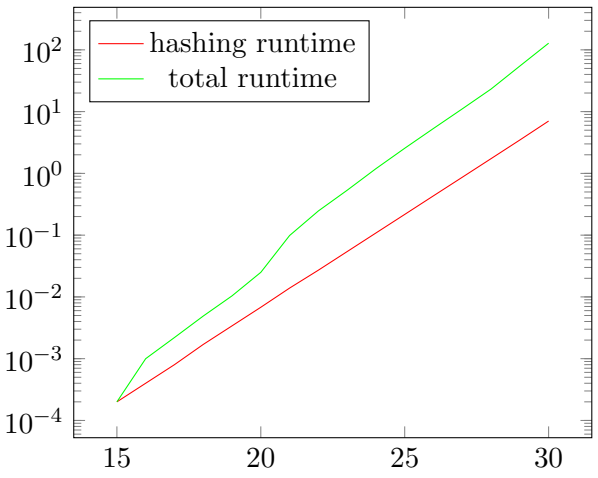

When adding the 10th edge $(4, 12)$, we find the paths $4 \to 10 \to 5 \to 14 \to 2 \to 15 \to 1$ and $12 \to 2 \to 15 \to 1$ with equal roots. In this case, we can compute the length of the resulting cycle as 1 plus the sum of the path-lengths to the node where the two paths first join. In the diagram, the paths first join at 2, and the cycle length is computed as $1 + 4 + 1 = 6$. 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.

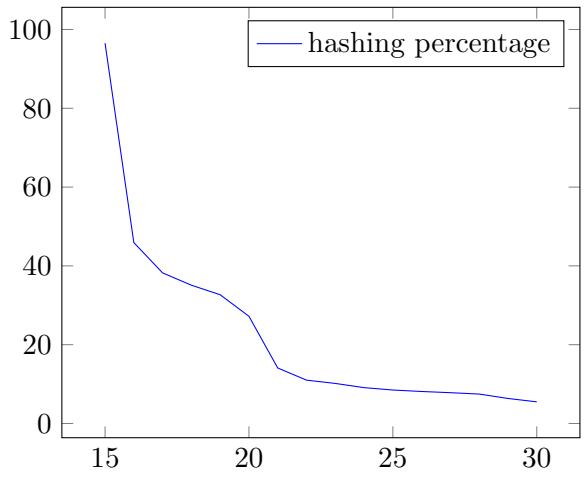
## 6 Union-find

The above representation of the directed cuckoo graph is an example of a *disjoint-set data structure* [7], and our algorithm is closely related to the well known union-find algorithm, where the find operation determines which subset an element is in, and the union operation joins two subsets into a single one. For each edge addition to the cuckoo graph we perform the equivalent of two find operaations and one union operation. The difference is that the union-find algorithm is free to add directed edges between arbitrary elements. Thus it can join two subsets by adding an edge from one root to another, with no need to reverse any edges. Conversely, our algorithm can be seen as the first one that solves the union-find problem by maintaining a direction on all union operations while keeping the maximum outdegree at 1.

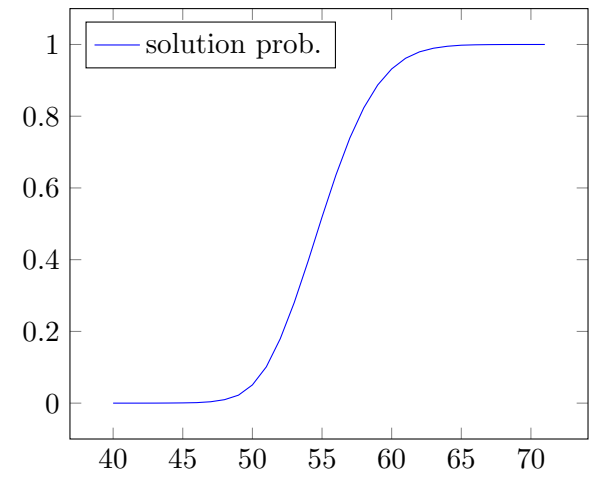
## 7 Implementation and performance

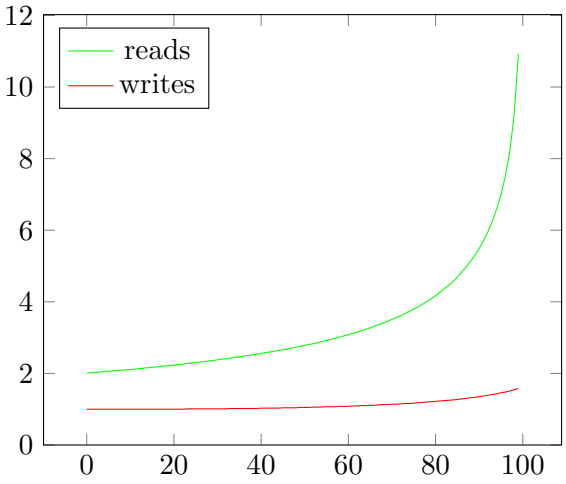
The C-program listed in the Appendix is also available online at `https://github.com/tromp/cuckoo` together with a Makefile, proof verifier and the latest version of this paper. 'make test' tests everything. 'make example' reproduces the example shown above. The main program uses 32 bits per node to represent the directed cuckoo graph, plus about 56KB per thread for 3 auxiliary arrays. The `us` and `vs` arrays 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 `uvpre` array precomputes edges in blocks of 1024, which was found (inexplicably) to save up to 30% runtime. 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.




## 8 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 $2^{256}/T$.

## 9 Memory-hardness

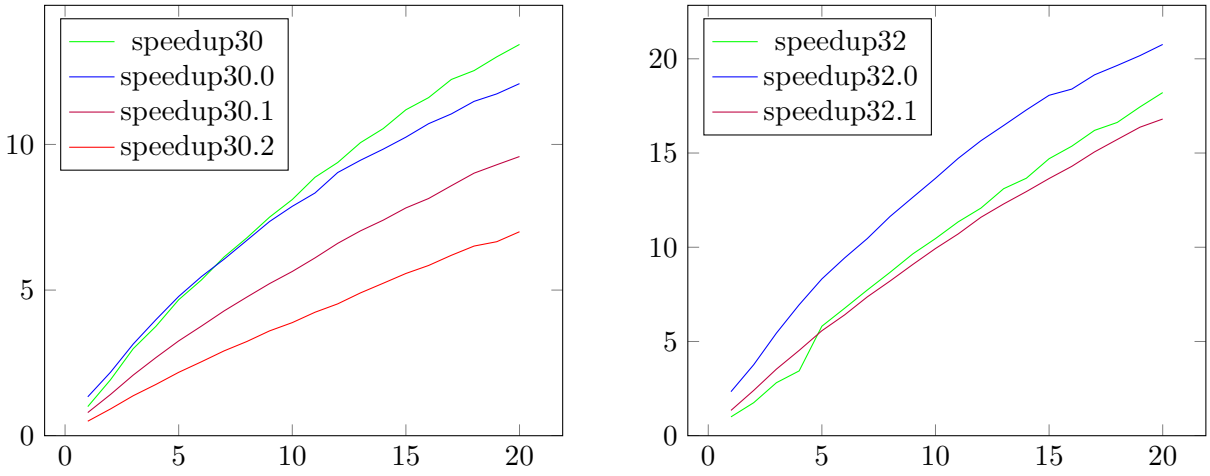
I conjecture that this problem doesn't allow for a time-memory trade-off. If one were to store only a fraction $p$ of $V_0$ and $V_1$, then one would have to reject a fraction $p^2$ of generated edges, drastically reducing the odds of finding cycles for $p < 1/\sqrt{2}$ (the reduction being exponential in cycle length).

## 10 Parallelization

The implementation allows the number of threads to be set with option `-t`. 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 [8], our implementation notices when the path length exceeds MAXPATHLEN (4096 by default), 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. 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.

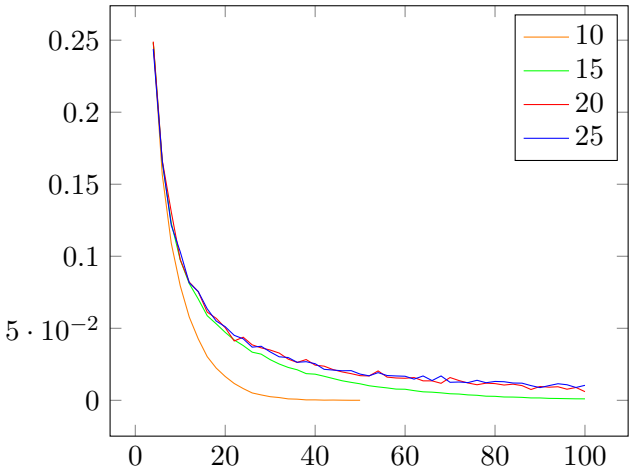
The right plot shows in green the speedup achieved by up to 40 threads on a system with two Intel Xeon E5-2690 cpus, each which 10 hyperthreaded cores running at 3GHz. Each datapoint represents 10 runs on $2^{30}$ nodes with headers "0" through "9". Although the speedup is sublinear at 26 for 40 threads, it doesn't look like the memory interface is saturated yet. Since we don't have access to more than 40 threads, we tried two alternative approaches to see happens when the memory access frequency increases further, In the second approach we precompute all the hashes, shown in orange. This shows some sort of breakdown of the memory interface starting from 21 threads, which is presumably where hyperthreading starts to kick in. It is perhaps odd that such a hyperthreading transition is completely absent from the green line. Since the hash precomputation showed a significant speedup, the implementation was changed to make it the default. Reproducing the other curves requires compilation with -DPRESIP=0 for C or setting CuckooSolve.PRESIP=1 for java.

UPDATE: We added 3 curves for the new implementation with 21x reduced memory, tested up to 20 threads. This shows some really weird boost at the thread maximum.




## 11 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.




## 12 Choice of memory size

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, With a significant risk of sending machines into swap-hell and likely alerting its owner, botnet operators will refrain from deploying Cuckoo Cycle, prefering other proof-of-work schemes whose smaller memory footprint allows them to stay under the radar. With memory sizes doubling roughly every 18 months, 16GB will become commonplace in a few years. While the current algorithm can accomodate up to

$N = 2^{33} - 2$ nodes (32GB) by separately keeping track of which partition a vertex is in, 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.

## 13 Memory latency; the great equalizer

While cpu-speed and (sequential) 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 in order to level the mining playing field. We introduce the very first graph-theoretic proof-of-work system. which aims to satisfy most of the above 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.

## 14 Related work

As proof-of-work for new blocks of transactions, Bitcoin [1] uses hashcash [2]. This 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 growth was enabled by the migration of hash computation from desktop processors (CPUs) to graphics-card procecssors (GPUs), to field-programmable gate arrays (FPGAs), and finally to custom designed chips (ASICs).

Downsides of this development include high investment costs, rapid obsolesence, centralization of mining power, and large power consumption. Although ASICs are the most energy-efficient way of computing hashes, the tiny amount of die-space needed for a single SHA256 circuit allows a huge number of them (e.g. 1440 on KnC's Neptune) to be crammed onto a single chip, consuming 100s of Watts and requiring ample cooling. Thus, energy costs dominate the economics of mining.

This has led people to look for alternative proof-of-work systems that, by requiring a nontrivial amount of memory, resist such massive parallelizability, and narrow the performance gap with commodity hardware. Memory chips, in the form of DRAM, have only a small portion of their circuitry active at any time[1] and thus require orders of magnitude less power.

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. ASICs first appeared on the market in early 2014 and are expected to dominate Litecoin mining by the fourth quarter.

Primecoin [3] is the first example of a number-theoretic proof-of-work. an interesting number-theoretdesign 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. GPU implementations that outperform CPU ones have recently appeared. Downsides to this proof-of-work

[1] this applies to each of the 8 or so *banks* that make up the memory in each chip, while operating in parallel

are its complexity and the scope for algorithmic improvements. 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.

## 15 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] Wikipedia, "Disjoint-set data structure — wikipedia, the free encyclopedia," 2014, [Online; accessed 23-March-2014]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Disjoint-set_data_structure&oldid=600366584

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


## 16 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 SIZESHIFT
#define SIZESHIFT 25
```

```
#endif
#ifndef PROOFSIZE
#define PROOFSIZE 42
#endif

#define SIZE (1UL<<SIZESHIFT)
#define HALFSIZE (SIZE/2)
#define NODEMASK (HALFSIZE-1)

typedef uint32_t u32;
typedef uint64_t u64;
typedef u64 nonce_t;
typedef u64 node_t;

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

// SipHash-2-4 specialized to precomputed key and 8 byte nonces
u64 siphash24(siphash_ctx *ctx, u64 nonce) {
  u64 v0 = ctx->v[0], v1 = ctx->v[1], v2 = ctx->v[2], v3 = ctx->v[3] ^ nonce;
  SIPROUND; SIPROUND;
  v0 ^= nonce;
  v2 ^= 0xff;
  SIPROUND; SIPROUND; SIPROUND; SIPROUND;
  return v0 ^ v1 ^ v2  ^ v3;
}

// generate edge endpoint in cuckoo graph
node_t sipnode(siphash_ctx *ctx, nonce_t nonce, int uorv) {
  return siphash24(ctx, 2*nonce + uorv) & NODEMASK;
}
```

```
void sipedge(siphash_ctx *ctx, nonce_t nonce, node_t *pu, node_t *pv) {
  *pu = sipnode(ctx, nonce, 0);
  *pv = sipnode(ctx, nonce, 1);
}

// verify that (ascending) nonces, all less than easiness, form a cycle in header-generated graph
int verify(nonce_t nonces[PROOFSIZE], const char *header, u64 easiness) {
  siphash_ctx ctx;
  setheader(&ctx, header);
  node_t us[PROOFSIZE], vs[PROOFSIZE];
  unsigned 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
    unsigned j = i;
    for (unsigned 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 (unsigned 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;
}
```

# 17 Appendix B: cuckoo_miner.h

```
// Cuckoo Cycle, a memory-hard proof-of-work
// Copyright (c) 2013-2014 John Tromp
// The edge=trimming time-memory trade-off is due to Dave Anderson:
// http://da-data.blogspot.com/2014/03/a-public-review-of-cuckoo-cycle.html

#include "cuckoo.h"
#ifdef __APPLE__
#include "osx_barrier.h"
#endif
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <assert.h>
#include <vector>
#include <atomic>
#include <set>
```

```
// algorithm parameters
#ifndef PART_BITS
// #bits used to partition edge set processing to save memory
// a value of 0 does no partitioning and is fastest
// a value of 1 partitions in two, making twice_set the
// same size as shrinkingset at about 33% slowdown
// higher values are not that interesting
#define PART_BITS 0
#endif
// L3 cache should exceed NBUCKETS buckets of BUCKETSIZE uint_64_t (0.5MB below)
#ifndef LOGNBUCKETS
#define LOGNBUCKETS      8
#endif
#ifndef BUCKETSIZE
#define BUCKETSIZE       256
#endif

#ifndef IDXSHIFT
// we want sizeof(cuckoo_hash) == sizeof(twice_set), so
// CUCKOO_SIZE * sizeof(u64) == TWICE_WORDS * sizeof(u32)
// CUCKOO_SIZE * 2 == TWICE_WORDS
// (SIZE >> IDXSHIFT) * 2 == 2 * ONCE_BITS / 32
// SIZE >> IDXSHIFT == HALFSIZE >> PART_BITS >> 5
// IDXSHIFT == 1 + PART_BITS + 5
#define IDXSHIFT (PART_BITS + 6)
#endif
#ifndef CLUMPSHIFT
// 2^CLUMPSHIFT should exceed maximum index drift (ui++) in cuckoo_hash
// SIZESHIFT-1 is limited to 64-KEYSHIFT
#define CLUMPSHIFT 9
#endif
// grow with cube root of size, hardly affected by trimming
#define MAXPATHLEN (8 << (SIZESHIFT/3))

// set that starts out full and gets reset by threads on disjoint words
class shrinkingset {
public:
  std::vector<u32> bits;
  std::vector<u64> cnt;

  shrinkingset(int nthreads) {
    nonce_t nwords = HALFSIZE/32;
    bits.resize(nwords);
    cnt.resize(nthreads);
    cnt[0] = HALFSIZE;
  }
  u64 count() const {
    u64 sum = 0L;
    for (unsigned i=0; i<cnt.size(); i++)
      sum += cnt[i];
    return sum;
  }
  void reset(nonce_t n, int thread) {
    bits[n/32] |= 1 << (n%32);
    cnt[thread]--;
  }
  bool test(node_t n) const {
    return !((bits[n/32] >> (n%32)) & 1);
  }
  u32 block(node_t n) const {
```

```
    return ~bits[n/32];
  }
};

#define PART_MASK ((1 << PART_BITS) - 1)
#define ONCE_BITS (HALFSIZE >> PART_BITS)
#define TWICE_WORDS ((2 * ONCE_BITS) / 32)

class twice_set {
public:
#ifdef ATOMIC
  typedef std::atomic<u32> au32;
  au32 *bits;

  twice_set() {
    assert(bits = (au32 *)calloc(TWICE_WORDS, sizeof(au32)));
  }
  void reset() {
    for (unsigned i=0; i<TWICE_WORDS; i++)
      bits[i].store(0, std::memory_order_relaxed);
  }
  void set(node_t u) {
    node_t idx = u/16;
    u32 bit = 1 << (2 * (u%16));
    u32 old = std::atomic_fetch_or_explicit(&bits[idx], bit    , std::memory_order_relaxed);
    if (old & bit) std::atomic_fetch_or_explicit(&bits[idx], bit<<1, std::memory_order_relaxed);
  }
  u32 test(node_t u) const {
    return (bits[u/16].load(std::memory_order_relaxed) >> (2 * (u%16))) & 2;
  }
#else
  u32 *bits;

  twice_set() {
    assert(bits = (u32 *)calloc(TWICE_WORDS, sizeof(u32)));
  }
  void reset() {
    for (unsigned i=0; i<TWICE_WORDS; i++)
      bits[i] = 0;
  }
  void set(node_t u) {
    node_t idx = u/16;
    u32 bit = 1 << (2 * (u%16));
    u32 old = bits[idx];
    bits[idx] = old | (bit + (old & bit));
  }
  u32 test(node_t u) const {
    return bits[u/16] >> (2 * (u%16)) & 2;
  }
#endif
  ~twice_set() {
    free(bits);
  }
};

#define CUCKOO_SIZE (SIZE >> IDXSHIFT)
#define CUCKOO_MASK (CUCKOO_SIZE - 1)
#define KEYSHIFT (IDXSHIFT + CLUMPSHIFT)
#define KEYMASK ((1 << KEYSHIFT) - 1)
```

```
class cuckoo_hash {
public:
  typedef std::atomic<u64> au64;
  au64 *cuckoo;

  cuckoo_hash() {
    assert(cuckoo = (au64 *)calloc(CUCKOO_SIZE, sizeof(au64)));
  }
  ~cuckoo_hash() {
    free(cuckoo);
  }
  void set(node_t u, node_t v) {
    u64 niew = (u64)v << KEYSHIFT | (u & KEYMASK);;
    for (node_t ui = u >> IDXSHIFT; ; ui = (ui+1) & CUCKOO_MASK) {
      u64 old = 0;
      if (cuckoo[ui].compare_exchange_strong(old, niew, std::memory_order_relaxed))
        return;
      if (((u^old) & KEYMASK) == 0) {
        cuckoo[ui].store(niew, std::memory_order_relaxed);
        return;
      }
    }
  }
  node_t operator[](node_t u) const {
    for (node_t ui = u >> IDXSHIFT; ; ui = (ui+1) & CUCKOO_MASK) {
      u64 cu = cuckoo[ui].load(std::memory_order_relaxed);
      if (!cu)
        return 0;
      if (((u^cu) & KEYMASK) == 0)
        return (node_t)(cu >> KEYSHIFT);
    }
  }
};

class cuckoo_ctx {
public:
  siphash_ctx sip_ctx;
  shrinkingset *alive;
  twice_set *nonleaf;
  cuckoo_hash *cuckoo;
  nonce_t (*sols)[PROOFSIZE];
  unsigned maxsols;
  std::atomic<unsigned> nsols;
  int nthreads;
  int ntrims;
  pthread_barrier_t barry;

  cuckoo_ctx(const char* header, int n_threads, int n_trims, int max_sols) {
    setheader(&sip_ctx, header);
    nthreads = n_threads;
    alive = new shrinkingset(nthreads);
    cuckoo = 0;
    nonleaf = new twice_set;
    ntrims = n_trims;
    assert(pthread_barrier_init(&barry, NULL, nthreads) == 0);
    assert(sols = (nonce_t (*)[PROOFSIZE])calloc(maxsols = max_sols, PROOFSIZE*sizeof(nonce_t)));
    nsols = 0;
  }
  ~cuckoo_ctx() {
    delete alive;
```

```
    if (nonleaf)
      delete nonleaf;
    if (cuckoo)
      delete cuckoo;
  }
};

typedef struct {
  int id;
  pthread_t thread;
  cuckoo_ctx *ctx;
  u64 (* buckets)[BUCKETSIZE];
} thread_ctx;

void barrier(pthread_barrier_t *barry) {
  int rc = pthread_barrier_wait(barry);
  if (rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD) {
    printf("Could not wait on barrier\n");
    pthread_exit(NULL);
  }
}

#define NBUCKETS          (1 << LOGNBUCKETS)
#define BUCKETSHIFT       (SIZESHIFT-1 - LOGNBUCKETS)
#define NONCESHIFT        (SIZESHIFT-1 - PART_BITS)
#define NODEPARTMASK      (NODEMASK >> PART_BITS)
#define NONCETRUNC        (1L << (64 - NONCESHIFT))

void trim_edges(thread_ctx *tp) {
  cuckoo_ctx *ctx = tp->ctx;
  u64 (* buckets)[BUCKETSIZE] = tp->buckets;
  shrinkingset *alive = ctx->alive;
  twice_set *nonleaf = ctx->nonleaf;
  u32 bucketsizes[NBUCKETS];

  for (unsigned part = 0; part <= PART_MASK; part++) {
    for (int uorv = 0; uorv < 2; uorv++) {
      if (tp->id == 0)
        nonleaf->reset();
      barrier(&ctx->barry);
      for (int qkill = 0; qkill < 2; qkill++) {
        for (int b=0; b < NBUCKETS; b++)
          bucketsizes[b] = 0;
        for (nonce_t block = tp->id*32; block < HALFSIZE; block += ctx->nthreads*32) {
          u32 alive32 = alive->block(block); // GLOBAL 1 SEQ
          for (nonce_t nonce = block; alive32; alive32>>=1, nonce++) {
            if (alive32 & 1) {
              node_t u = sipnode(&ctx->sip_ctx, nonce, uorv);
              if ((u & PART_MASK) == part) {
                u32 b = u >> BUCKETSHIFT;
                u32 *bsize = &bucketsizes[b];
                buckets[b][*bsize] = (nonce << NONCESHIFT) | (u >> PART_BITS);
                if (++*bsize == BUCKETSIZE) {
                  *bsize = 0;
                  for (int i=0; i<BUCKETSIZE; i++) {
                    u64 bi = buckets[b][i];
                    if (!qkill) {
                      nonleaf->set(bi & NODEPARTMASK); // GLOBAL 1 RND BUCKETSIZE-1 SEQ
                    } else {
                      if (!nonleaf->test(bi & NODEPARTMASK)) { // GLOBAL 1 RND BUCKETSIZE-1 SEQ
```

```
                        nonce_t n = (nonce & -NONCETRUNC) | (bi >> NONCESHIFT);
                        alive->reset(n <= nonce ? n : n-NONCETRUNC, tp->id); // GLOBAL SEQ
                      }
                    }
                  }
                }
              }
            }
          }
          for (int b=0; b < NBUCKETS; b++) {
            int ni = bucketsizes[b];
            for (int i=0; i<ni ; i++) {
              node_t bi = buckets[b][i];
              if (!qkill) {
                nonleaf->set(bi & NODEPARTMASK);
              } else {
                if (!nonleaf->test(bi & NODEPARTMASK)) {
                  nonce_t n = (HALFSIZE & -NONCETRUNC) | (bi >> NONCESHIFT);
                  alive->reset(n < HALFSIZE  ? n : n-NONCETRUNC, tp->id); // GLOBAL SEQ
                }
              }
            }
          }
          barrier(&ctx->barry);
        }
      }
    }
  }

int path(cuckoo_hash &cuckoo, node_t u, node_t *us) {
  int nu;
  for (nu = 0; u; u = cuckoo[u]) {
    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;
}

typedef std::pair<node_t,node_t> edge;

void solution(cuckoo_ctx *ctx, node_t *us, int nu, node_t *vs, int nv) {
  std::set<edge> cycle;
  unsigned n;
  cycle.insert(edge(*us, *vs));
  while (nu--)
    cycle.insert(edge(us[(nu+1)&~1], us[nu|1])); // u's in even position; v's in odd
  while (nv--)
    cycle.insert(edge(vs[nv|1], vs[(nv+1)&~1])); // u's in odd position; v's in even
  unsigned soli = std::atomic_fetch_add_explicit(&ctx->nsols, 1U, std::memory_order_relaxed);
  for (nonce_t nonce = n = 0; nonce < HALFSIZE; nonce++)
    if (ctx->alive->test(nonce)) {
      edge e(sipnode(&ctx->sip_ctx, nonce, 0), HALFSIZE+sipnode(&ctx->sip_ctx, nonce, 1));
      if (cycle.find(e) != cycle.end()) {
```

```
        ctx->sols[soli][n++] = nonce;
        cycle.erase(e);
      }
    }
}

void *worker(void *vp) {
  thread_ctx *tp = (thread_ctx *)vp;
  assert(tp->buckets = (u64 (*)[BUCKETSIZE])calloc(NBUCKETS * BUCKETSIZE, sizeof(u64)));
  cuckoo_ctx *ctx = tp->ctx;

  shrinkingset *alive = ctx->alive;
  int load = 100 * HALFSIZE / CUCKOO_SIZE;
  if (tp->id == 0)
    printf("initial load %d%%\n", load);
  for (int round=1; round <= ctx->ntrims; round++) {
    trim_edges(tp);
    if (tp->id == 0) {
      load = (int)(100 * alive->count() / CUCKOO_SIZE);
      printf("%d trims: load %d%%\n", round, load);
    }
  }
  if (tp->id == 0) {
    if (load >= 90) {
      printf("overloaded! exiting...");
      exit(0);
    }
    delete ctx->nonleaf;
    ctx->nonleaf = 0;
    ctx->cuckoo = new cuckoo_hash();
  }
  barrier(&ctx->barry);
  cuckoo_hash &cuckoo = *ctx->cuckoo;
  node_t us[MAXPATHLEN], vs[MAXPATHLEN];
  for (nonce_t block = tp->id*32; block < HALFSIZE; block += ctx->nthreads*32) {
    for (nonce_t nonce = block; nonce < block+32 && nonce < HALFSIZE; nonce++) {
      if (alive->test(nonce)) {
        node_t u0, v0;
        sipedge(&ctx->sip_ctx, nonce, &u0, &v0);
        v0 += HALFSIZE;  // make v's different from u's
        node_t u = cuckoo[u0], v = cuckoo[v0];
        if (u == v0 || v == u0)
          continue; // ignore duplicate edges
        us[0] = u0;
        vs[0] = v0;
        int nu = path(cuckoo, u, us), nv = path(cuckoo, v, vs);
        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:%d%%\n", len, tp->id, (int)(nonce*100L/HALFSIZE));
          if (len == PROOFSIZE && ctx->nsols < ctx->maxsols)
            solution(ctx, us, nu, vs, nv);
          continue;
        }
        if (nu < nv) {
          while (nu--)
            cuckoo.set(us[nu+1], us[nu]);
          cuckoo.set(u0, v0);
        } else {
```

```
          while (nv--)
            cuckoo.set(vs[nv+1], vs[nv]);
          cuckoo.set(v0, u0);
        }
      }
    }
  }
  free(tp->buckets);
  pthread_exit(NULL);
}
```

# 18 Appendix C: cuckoo_miner.cpp

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

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

int main(int argc, char **argv) {
  int nthreads = 1;
  int maxsols  = 8;
  int ntrims   = 8 << PART_BITS;
  const char *header = "";
  int c;
  while ((c = getopt (argc, argv, "h:m:n:t:")) != -1) {
    switch (c) {
      case 'h':
        header = optarg;
        break;
      case 'm':
        maxsols = atoi(optarg);
        break;
      case 'n':
        ntrims = atoi(optarg);
        break;
      case 't':
        nthreads = atoi(optarg);
        break;
    }
  }
  printf("Looking for %d-cycle on cuckoo%d(\"%s\") with 50%% edges, %d trims, %d threads\n",
              PROOFSIZE, SIZESHIFT, header, ntrims, nthreads);
  u64 edgeBytes = HALFSIZE/8, nodeBytes = TWICE_WORDS*4;
  int edgeUnit, nodeUnit;
  for (edgeUnit=0; edgeBytes >= 1024; edgeBytes>>=10,edgeUnit++) ;
  for (nodeUnit=0; nodeBytes >= 1024; nodeBytes>>=10,nodeUnit++) ;
  printf("Using %d%cB edge and %d%cB node memory.\n",
     (int)edgeBytes, " KMGT"[edgeUnit], (int)nodeBytes, " KMGT"[nodeUnit]);
  cuckoo_ctx ctx(header, nthreads, ntrims, maxsols);
  thread_ctx *threads = (thread_ctx *)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);
  free(threads);
```

```
  for (unsigned s = 0; s < ctx.nsols; s++) {
    printf("Solution");
    for (int i = 0; i < PROOFSIZE; i++)
      printf(" %lx", (long)ctx.sols[s][i]);
    printf("\n");
  }
  return 0;
}
```