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

John Tromp

September 15, 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 cycle finding algorithm uses one bit per edge, and up to one bit per node. Runtime is linear in graph size and dominated by random access latency. We exhibit two alternative algorithms that allow for a memory-time trade-off (TMTO)—decreased memory usage by a factor k while increasing runtime by a factor $\Omega(k)$. The implied constant gives a notion of (presumed) memory-hardness, which is shown to be dependent on cycle length, and thus guides its choice.

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 entails 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, thereby offering the first alternative to hashcash among crypto-currencies. Primecoin identifies long chains of nearly doubled prime numbers, constrained by a certain relation to the block header. Verification of these chains, while very slow compared to bitcoin's, is much faster than attempting to find one. This asymmetry between proof (attempt) and verification is typical in non-hashcash proofs of work. 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 pseudo-random graphs. In the Erdős-Rényi model, denoted $G(N, M)$, a graph is chosen uniformly at random from the collection of all graphs with N nodes and M edges. Instead, we choose edges deterministically from the output of a keyed hash function, whose key could be chosen uniformly at random. For a well-behaved hash function, these two classes of random graphs should have nearly identical properties. Formally, fix a keyed hash function $h : \{0,1\}^K \times \{0,1\}^{W_i} \to \{0,1\}^{W_o}$, and a small graph H as a target subgraph[1]. Now pick a large number $N \leq 2_o^W$ 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

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

The inputs $i \in [0, \ldots, M-1]$ are also called *nonces*[2] The graph has a *solution* if H occurs as a subgraph. Denote the number of edges in H as L. A proof of solution is an ordered list of L nonces that generate the edges of H's occurrence in G_k. Such a proof is verifiable in time $O(L)$, independent of N and M.

A simple variation generates random bipartite graphs: $G_k = (V_0 \cup V_1, E)$ where $V_i = \{v_0, v_2, \ldots, v_{N-2}\}$, $V_1 = \{v_1, v_3, \ldots, v_{N-1}\}$, and

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

The expected number of occurrences of H as a subgraph of G is a function of both N and M, and in many cases 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

The simplest possible choice of subgraph is a fully connected one, or a *clique*. While an interesting choice, akin to the number-theoretic notion of a prime-cluster, as used in Riecoin, we leave its consideration to a future paper. In this paper we focus on what is perhaps the next-simplest possible choice, the *cycle*. Specifically, we propose the hash function siphash with a $K = 128$ bit key, $W_i = W_o = 64$ input and output bits, $N \leq 2^{64}$ a 2-power, $M = N/2$, and H a 42-cycle (more on that in a later section). 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 [4], 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 matches the bipartite graph defined above if the cuckoo hashtable were based on function h. In fact, the insertion procedure suggests a simple algorithm for detecting cycles.

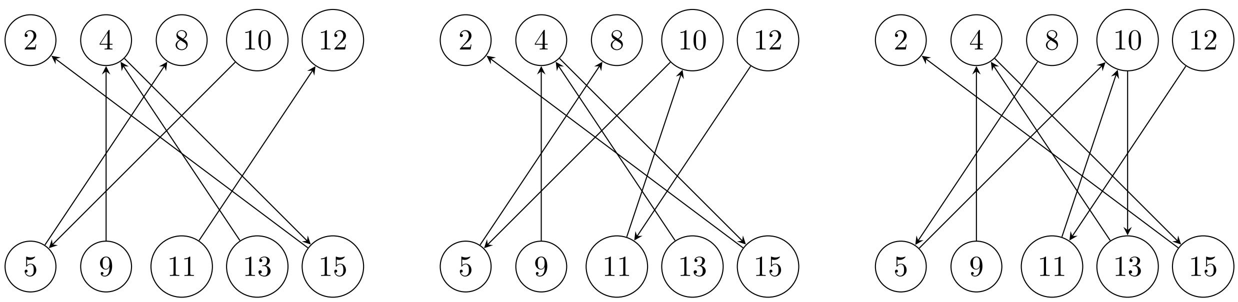
5 Cycle detection in Cuckoo hashing

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

[1] hash functions generally have arbitrary length inputs, but here we fix the input width at W_i bits.

[2] These *micro* nonces should be distinguished from the *macro* nonce used to generate key k.

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 $(2,15), (4,9), (8,5), (4,15), (12,11), (10,5)$ and $(4,13)$ (nodes with no incident edges are omitted for clarity). In order to add the 8th edge $(10,11)$, we follow the paths $10 \to 5 \to 8$ and $11 \to 12$ to find different roots 8 and 12. Since the latter path is shorter, we reverse it to $12 \to 11$ so we can add the new edge as $(11 \to 10)$, resulting in the middle diagram. In order to add to 9th edge $(10,13)$ we now find the path from 10 to be the shorter one, so we reverse that and add the new edge as $(10 \to 13)$, 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 $1+4+1=6$.

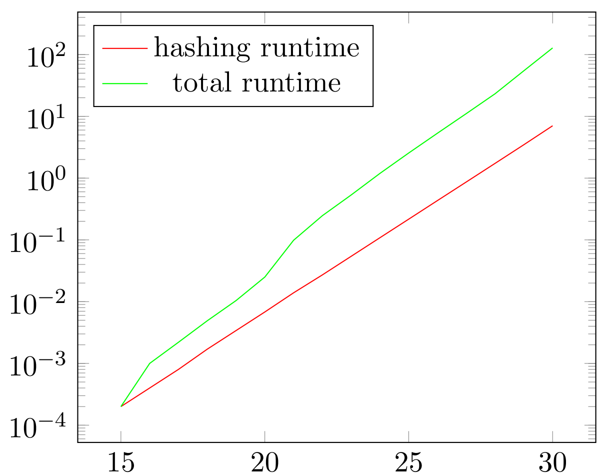
6 Union-find

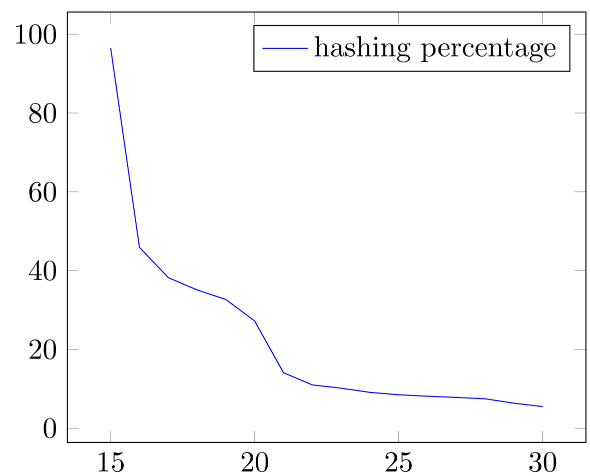
The above representation of the directed cuckoo graph is an example of a *disjoint-set data structure* [5], and our algorithm is closely related to the well-known union-find algorithm, where the find operation determines which subset an element is in, and the union operation joins two subsets into a single one. For each edge addition to the cuckoo graph we perform the equivalent of two find operations 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. Our algorithm on the other hand solves the union-find problem by maintaining a direction on all union operations while keeping the maximum outdegree at 1.

7 Cuckoo Cycle basic algorithm

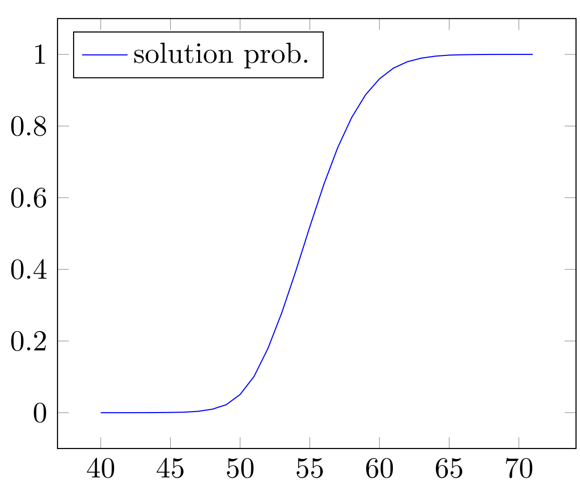
The above algorithm for inserting edges and detecting cycles forms the basis for our basic proof-of-work algorithm. If a cycle of length L is found, then we solved the problem, and recover the proof by storing the cycle edges in a set and enumerating nonces once more to see which ones generate edges in the set. If a cycle of a different length is found, then we keep the graph acyclic by ignoring the edge. There is some risk of overlooking other L-cycles through that edge, but when the expected number of cycles is low (which is what we design for), this ignoring of cycle forming edges hardly affects the rate of solution finding.

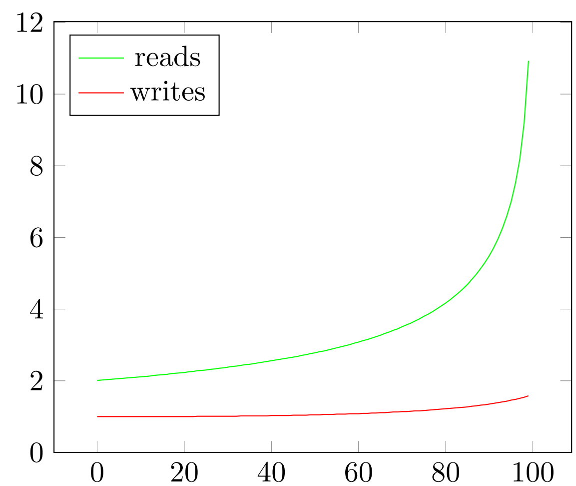
This algorithm is available online at `https://github.com/tromp/cuckoo` as either the C-program simple_miner.cpp or the Java program SimpleMiner.java. A proof verifier is available as cuckoo.c or Cuckoo.java, while the repository also has a Makefile, as well as the latest version of this paper. 'make example' reproduces the example shown above. The simple program uses 32 bits per node to represent the directed cuckoo graph, plus about 64KB per thread for 2 auxiliary arrays. 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%.





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 difficulty 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 Edge Trimming

Dave Andersen [6] suggested drastically reducing the number of edges our basic algorithm has to process, by repeatedly identifying nodes of degree one and eliminating its incident edge. Such *leaf edges* can never be part of a cycle. This works whenever the number of edges M, is at most half the number of nodes N, since the expected degree of a node is then at most 1.

This is implemented in our main algorithm in cuckoo_miner.h It maintains a set of *alive* edges as a bit vector. Initially all edges are alive. In each of a given number of trimming rounds, it shrinks this set as follows. A vector of 2-bit degree counters, one per u-node, is initialized to all zeroes. Next, for all alive edges, compute its u-endpoint and increase the corresponding counter, capping the value at 2. Next, for all alive edges, compute its u-endpoint and if the corresponding counter is less than 2, set the edge to be not-alive. These steps are repeated for the other partition, of v-nodes. Preprocessor symbol PART_BITS, whose value we'll denote as B, allows for trading-off node counter storage for time, by processing the nodes in multiple passes depending on the value of their least significant bits. The memory usage is M bits for the alive set and $N/2^B$ for the counters.

After all edge trimming rounds, the counter memory is freed, and allocated to a custom cuckoo_hashtable (based on [7]) that presents the same interface as the simple array in the basic algorithm, but gets by with much fewer locations, as long as its *load*, the ratio of remaining edges to number of locations, is significantly less than 1.

The number of trimming rounds, which can be set with option `-n`, defaults to $2^{1+(B+3)*(B+4)/2}$, which achieves a load close to 50%.

10 A Time-Memory Trade-Off algorithm

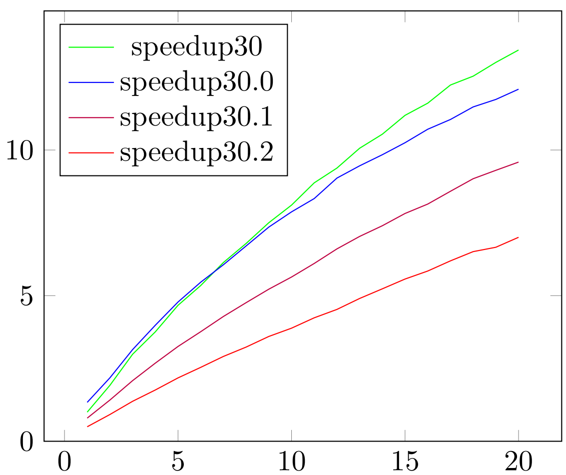
David Andersen suggested an alternative method of trimming that avoids storing a bit per edge. Expanding on that idea led to the algorithm implemented in tomato_miner.h It selects a suitably small subset Z of vertices from partition U, and builds a breadth-first-search (BFS) forest of depth equal to cycle length L. For each new BFS layer, it enumerates all edges to see which ones are incident to the previous layer, and maintains a directed forest on all BFS nodes like the base algorithm does. If the graph has an L-cycle, then the above procedure will find it as long as the distance from (any node in) Z to the cycle is at most $L/2$. Benchmarking experiments show this to be more efficient than searching only to depth $L/2$, which would suffice for finding the cycle as long as it intersects Z. This algorithm uses memory roughly $|Z| \cdot L \cdot 64$ bits.

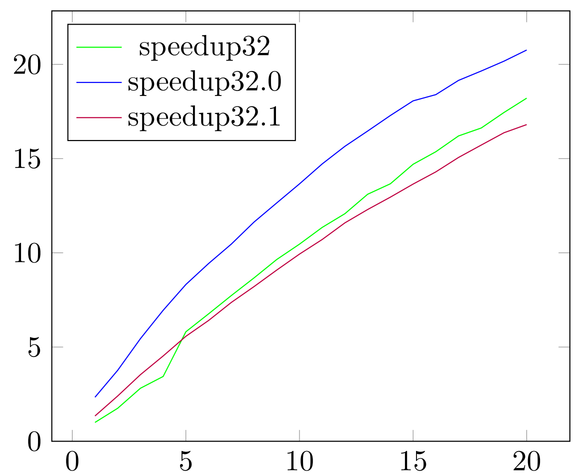
11 Parallelization

The implementation allows the number of threads to be set with option `-t`. For $0 \le t < T$, thread t processes all nonces $t \bmod T$. Parallelization in the basic algorithm presents some minor 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 inadvertent creation of cycles when a reversal in progress hampers another thread's path following causing it to overlook root equality. 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 (8192 by default), and reports whether this is due to a path conflict.

In the main algorithm, cycle detection only takes a small fraction of total runtime and the conflicts above could be avoided altogether by running the cycle detection single threaded. We therefore turn our attention to parallelization of edge trimming.

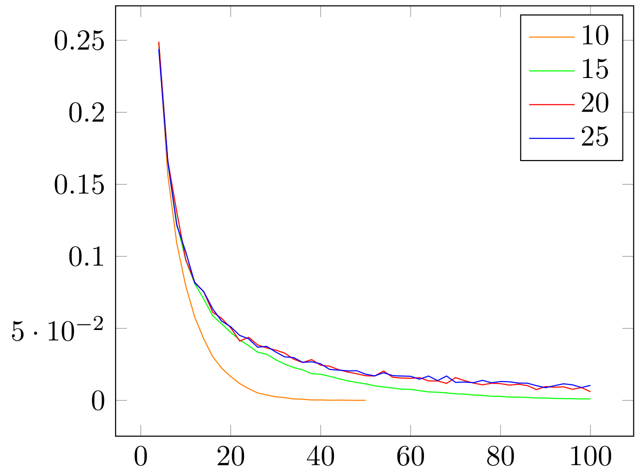
To be expanded...





12 Choice of cycle length

Extremely small cycle lengths risk the feasibility of alternative algorithms with better performance. 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 cycle finding approach is optimal for any $L = 4$ or $L = 6$. (Ongoing) analysis of the tmto miner shows that larger cycle length make the problem more memory-hard, which argues against such small cycle lengths. 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.



13 Choice of memory size

The chosen Cuckoo graph size affects what type of hardware can mine effectively at a given block interva time. To illustrate, suppose an average desktop machine needs 1 minute for a single proof attempt, and the block interval time is only 2 minutes. Then it will waste a large fraction (almost half) of its attempts as about half the time, someone else finds a proof in under 2 minutes. To reduce such waste to a small percentage, the time for a single proof attempt should be a similarly small fraction of the block interval time. Obviously this is achieved more easily with a small graph (and hence memory) size.

Larger memory sizes have two advantages though. First, they make it harder for botnets to mine without causing excessive swapping. Sending a computer into swap-hell will likely alert its owner and trigger a cleanup.

Second, they make it harder for custom FPGA or ASIC implementations to remain self-contained. Rather, to remain cost effective, they will likely need to make use of commodity DRAM chips, and incur a 50ns row activation delay for random access to each memory bank. If a future CPU with hardware siphash24 support can saturate DRAM, then

We expect these opposing goals to lead to sizes from $2^2$8 to $2^3$2, with the larger ones geared more toward longer block interval times and faster mining hardware.

An interesting option will be to take a Myriad-coin style approach: allow a range of k different sizes, but maintain a separate difficulty control for each one, adjusted so that each proof size accounts for roughly $1/k$ of all blocks.

14 Computation versus memory

Starting out at 32 leading zeroes in 2009, Bitcoin difficulty has steadily climbed and is currently at 66, representing an incredible $2^{66}/10$ double-hashes per minute. This growth was enabled by the migration of hash computation from desktop processors (CPUs) to graphics-card processors (GPUs), to field-programmable gate arrays (FPGAs), and finally to custom designed chips (ASICs).

Downsides of this development include high investment costs, rapid obsolescence, 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.

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.

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 have started to dominate Litecoin mining by the third quarter.

Momentum [9] 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 there is a straightforward linear time-memory trade-off.

Adam Back [10] has a good overview of proof-of-work papers past and present.

15 Conclusion

Cuckoo Cycle is a novel graph-theoretic proof-of-work design that combines scalable memory requirements with instant verifiability. It's also the first proof-of-work in which memory latency dominates the runtime. This could lead to mining costs being dominated by DRAM investments, changing the economics of mining in ways that require further study. More research is also needed to determine the effectiveness of GPUs and FPGAs at running Cuckoo Cycle.

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] 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

[5] 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

[6] D. Andersen, "A public review of cuckoo cycle," Apr. 2014. [Online]. Available: http://da-data.blogspot.com/2014/03/a-public-review-of-cuckoo-cycle.html

[7] J. Preshing, "The world's simplest lock-free hash table," Jun. 2013. [Online]. Available: http://preshing.com/20130605/the-worlds-simplest-lock-free-hash-table/

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

[9] 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

[10] A. Back, “Hashcash.org,” Feb. 2014. [Online]. Available: http://www.hashcash.org/papers/

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;
#if SIZESHIFT <= 32
typedef u32 nonce_t;
typedef u32 node_t;
#else
typedef u64 nonce_t;
typedef u64 node_t;
#endif

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))

#ifndef SHA256
#define SHA256(d, n, md) do { \
    SHA256_CTX c; \
    SHA256_Init(&c); \
    SHA256_Update(&c, d, n); \
    SHA256_Final(md, &c); \
  } while (0)
#endif

// 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, u32 uorv) {
  return (siphash24(ctx, 2*nonce + uorv) & NODEMASK) << 1 | uorv;
}

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 uvs[2*PROOFSIZE];
  for (u32 n = 0; n < PROOFSIZE; n++) {
    if (nonces[n] >= easiness || (n && nonces[n] <= nonces[n-1]))
      return 0;
    sipedge(&ctx, nonces[n], &uvs[2*n], &uvs[2*n+1]);
  }
  u32 i = 0;
  for (u32 n = PROOFSIZE; n; ) { // follow cycle for n more steps
    u32 j = i;
    for (u32 k = i&1; k < 2*PROOFSIZE; k += 2) // find unique other j with same parity and uvs[j]
      if (k != i && uvs[k] == uvs[i]) {
        if (j != i)
          return 0; // more than 2 occurences
        j = k;
    }
    if (j == i)
      return 0; // no other occurence
    i = j^1;
    if (--n && i == 0) // don't return to 0 too soon
      return 0;
  }
```

```
  return i == 0;
}
```