# Cuckoo Cycle:
# a memory bound graph-theoretic proof-of-work

John Tromp

April 2, 2019

**Abstract**

We introduce the first graph-theoretic proof-of-work system, based on finding small cycles or other structures in large random graphs. Such problems are trivially verifiable and arbitrarily scalable, and appear to require memory linear in graph size to solve efficiently. We show that in random bipartite graphs with average degree 1, length $L$ cycles occur with frequency approximately $1/L$. On these graphs, algorithm "lean" finds cycles using 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 time-memory trade-off (TMTO) algorithms decreasing memory usage by a factor $k$ while increasing runtime by a factor $c \cdot k$. The constant $c$ provides a measure of memory-hardness, which is shown to be dependent on cycle length, guiding the latter's choice. Trading in the other direction, algorithm "mean" uses a few dozen bits per edge but is 4x faster in practice, becoming memory bandwidth bound rather than latency bound. We present performance figures for optimized CPU and GPU implementations, and discuss possible ASIC implementations.

## 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]. Hashcash entails finding a nonce value such that application of a cryptographic hash function to this nonce and the rest of the block header, results in a number below a target threshold[1]. The threshold is dynamically adjusted by the protocol so as to maintain an average block interval of 10 minutes.

Bitcoin's choice of the simple and purely compute-bound SHA256 hash function allowed for an easy 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), with huge improvements in energy-efficiency at every step.

Since Bitcoin, many other crypto-currencies have adopted hashcash, with various choices of underlying hash function. the most well-known being *scrypt* as introduced by Tenebrix [3] (since faded into obscurity) and copied by Litecoin [4]. Scrypt, designed as a sequential memory-hard key derivation function, was specifically chosen to resist the migration away from CPUs and be "GPU-hostile". To adapt to the efficient verifiability requirement of proof-of-work, its memory footprint was limited to a mere 128 KB. Before long however, the large computing power and memory bandwidth of GPUs were harnessed to vastly outperform CPUs, and within a few years scrypt ASICs came to market to in turn obliterate GPU performance.

[1] or, less accurately, results in many leading zeroes

Primecoin [5] 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 characterstic in non-hashcash proofs of work. Recently, two other prime-number based crypto-currencies were introduced. Riecoin is based on finding clusters of prime numbers, and Gapcoin on finding large gaps between consecutive prime numbers.

Momentum [6] proposes finding birthday collisions of hash outputs, in what could well be the simplest possible asymetric proof-of-work, combining scalable memory usage with trivial verifiability. Unfortunately, as pointed out in [7], the "golden collision" search technique of [8] allows for reducing memory consumption $q$-fold while increasing runtime only $\sqrt{q}$, effectively breaking the scheme. Momentum is interesting though as both Cuckoo Cycle and Equihash generalize birthday collision in different ways.

Adam Back [9] has a good overview of proof-of-work papers past and present.

## 2 Motivation

Cuckoo Cycle aims to be the simplest possible proof-of-work. Verifying a cycle is a simple procedure, and edges are defined by a rather simple hash function. Meanwhile, the simplest kind of chip circuitry is SRAM memory. Its exceedingly regular nature jumps out in die photographs as big uniform rectangles. Cuckoo Cycle aims to be a proof of SRAM. As shown later, the most efficient known way of solving Cuckoo Cycles is by updating random 2-bit counters, a task for which SRAM is uniquely suited.

We will not strive for provable lower bounds on memory usage. Such bounds appear to be attainable only under the so-called *random oracle model*, where memory tends to be used merely as a store for chains of compute-intensive hash outputs. Instead, we present an efficient proof-finding algorithm along with our best attempts at memory-time trade-offs, and conjecture that these cannot be significantly improved upon. Lacking rigorous proofs of memory hard-ness, Cuckoo Cycle should be considered a heuristic scheme.

## 3 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 $h : \{0,1\}^K \times \{0,1\}^{h\text{in}} \rightarrow \{0,1\}^{h\text{out}}$, and a small graph $H$ as a target subgraph[2]. Now pick a large number $N \leq 2^{h\text{out}}$ as the number of nodes, and $M \leq 2^{h\text{in}-1}$ as the number of edges. Each key $k \in \{0,1\}^K$ generates a graph $G_k = (V, E)$ where $V = \{v_0, \ldots, v_{N-1}\}$, $E = \{e_0, \ldots, e_{M-1}\}$, and (with | denoting catenation of bit strings)

$$e_i = (v_{h_k(i|0) \bmod N}, v_{h_k(i|1) \bmod N}) \tag{1}$$

The inputs $i \in [0, \ldots, M-1]$ are called *edge indices* 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 $L$ edge indices that generate the edges of $H$'s occurrence in $G_k$. Such a proof is verifiable in time depending only on $H$ (typically linear in $L$), independent of $N$ and $M$.

[2] hash functions generally have arbitrary length inputs, but here we fix the input width at $h_{\text{in}}$ bits.

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

$$e_i = (u_{h_k(i|0) \bmod \frac{N}{2}}, v_{h_k(i|1) \bmod \frac{N}{2}}) \tag{2}$$

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 $\frac{M}{N}$ (half the average node degree). For fixed $N$, this 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.

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.

## 4 Cuckoo Cycle

In this paper we focus on what is perhaps the next-simplest possible choice, the *cycle*. Specifically, we propose to use bipartite graphs generated by the hash function siphash with a $K = 256$ bit key, $h_{\text{in}} = h_{\text{out}} = 64$ input and output bits, $N \leq 2^{64}$ a 2-power, $M = N/2$, and $H$ an $L$-cycle. The reason for calling the resulting proof-of-work Cuckoo Cycle is that inserting items in a Cuckoo hashtable naturally leads to cycle formation in random bipartite graphs.

## 5 Cuckoo hashing

Introduced by Rasmus Pagh and Flemming Friche Rodler [10], 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 inserted 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.

## 6 Cycle detection in Cuckoo Cycle

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, 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. Let us illustrate this process with an actual example.

The left diagram in Figure 1 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

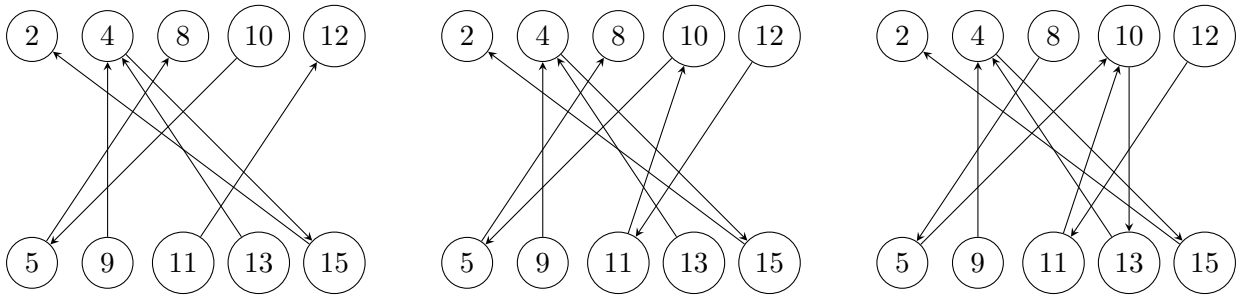


Figure 1: cycle formation and detection in a Cuckoo graph

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 $(8, 9)$, we find the paths $8 \to 5 \to 10 \to 13 \to 4 \to 15 \to 2$ and $9 \to 4 \to 15 \to 2$ 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 join. In the diagram, the paths join at node 4, and the cycle length is computed as $1 + 4 + 1 = 6$.

## 7 Union-find

The above representation of the directed cuckoo graph is an example of a *disjoint-set data structure* [11], 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.

## 8 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` in src/simple_miner.cpp, while a proof verifier is available in src/cuckoo.c. The github 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 two path-following arrays. The left plot in Figure 2 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

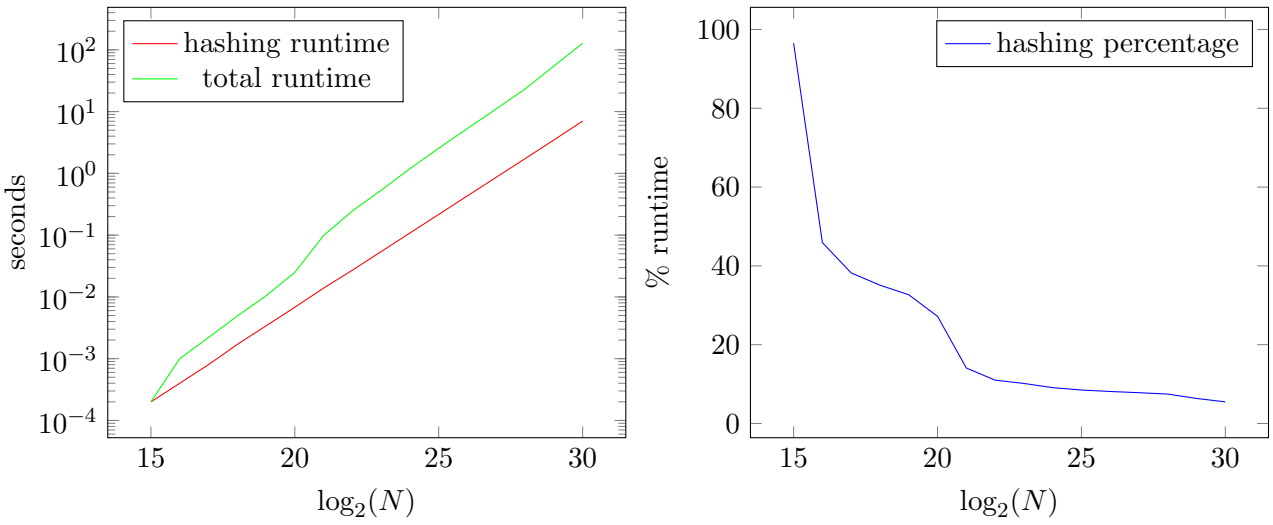


Figure 2: Runtime and compute intensity of the basic algorithm

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 in Figure 3 shows the probability of finding a 42-cycle as a function of the percentage edges/nodes, while the right plot shows the average number of memory reads and writes per edge as a function of the percentage of processed nonces (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 basic algorithm averages 3.3 reads and 1.1 writes per edge.

## 9 Theoretical cycle length distribution

## 10 Difficulty control

The ratio $\frac{M}{N}$ determines a base level of difficulty, which may suffice for applications where difficulty is to remain fixed. Ratios $\frac{M}{N} \geq 0.7$ are suitable when a practically guaranteed solution is desired.

For crypto currencies, where difficulty must scale in precisely controlled manner across a large range, adjusting the number of edges is not suitable. The implementation default $\frac{M}{N} = \frac{1}{2}$ 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 $\frac{2^{256}}{T}$.

## 11 Edge Trimming

David Andersen [12] suggested drastically reducing the number of edges our basic algorithm has to process, by repeatedly identifying nodes of degree one and eliminating their incident edge. Such *leaf edges* can never be part of a cycle. This works well when $\frac{M}{N} \leq \frac{1}{2}$ since the expected degree of a node is then at most 1, and a significant fraction of edges are expected to be leaf edges.

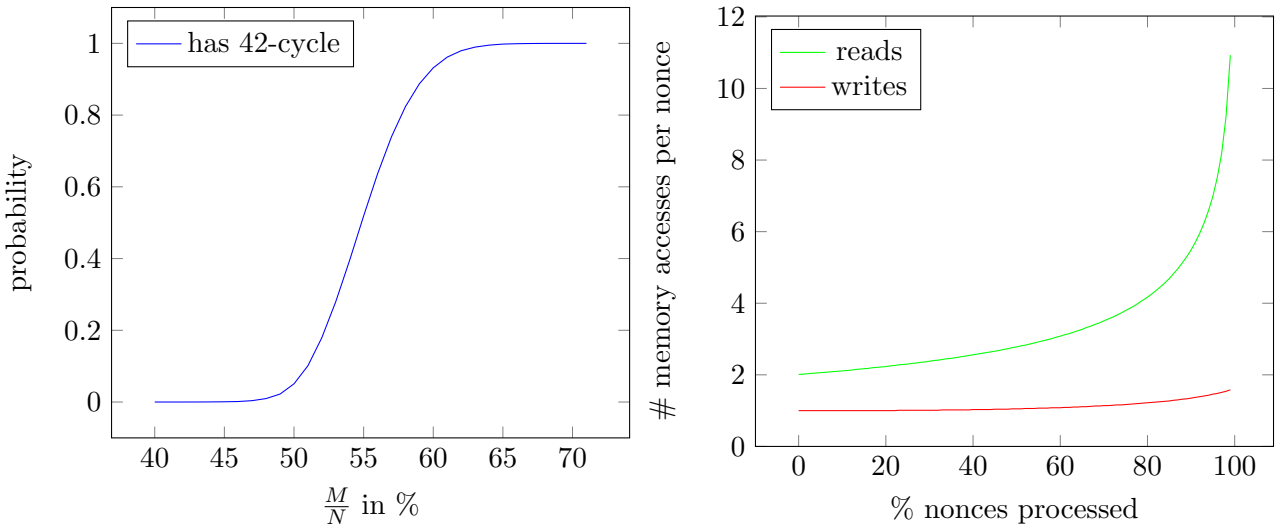


Figure 3: Threshold nature of solution, and increasing memory usage on threshold approach

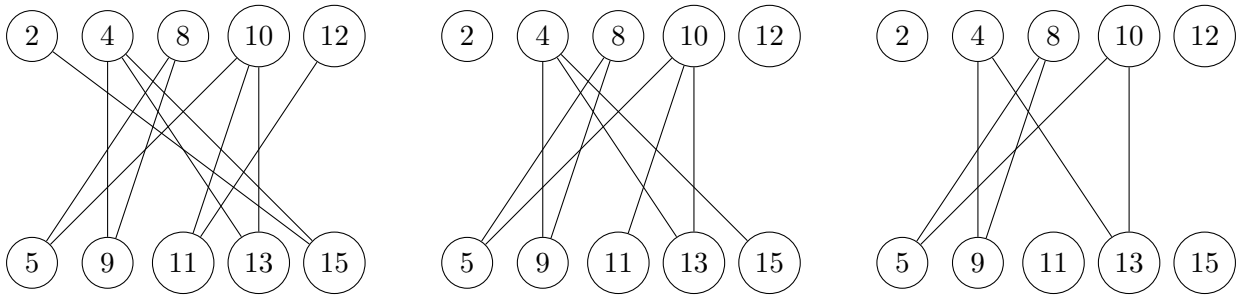


Figure 4: Trimming of edges which cannot be part of a cycle

Trimming is implemented in our main algorithm in cuckoo_miner and hcuckoo_miner.cpp. 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 even node, is initialized to all zeroes. Next, for all alive edges, compute its even endpoint and increase the corresponding counter, capping the value at 2. Next, for all alive edges, compute its even endpoint and if the corresponding counter is less than 2, set the edge to be not-alive. These steps, both of which cause the random accesses required in property MB2, are repeated for all odd endpoints.

Preprocessor symbol PART_BITS, whose value we'll denote as $B$, allows for *counter partitioning*, which trades off node counter storage for runtime, by processing nodes in multiple passes depending on the value of their $B$ least significant bits[3]. The memory usage is $M$ bits for the alive set and $N/2^B$ for the counters.

The diagrams in Figure 4 show two rounds of edge trimming on the earlier example. In round one even nodes 2 and 12 lose their single incident edge and in round two, odd nodes 11 and 15 lose their remaining single incident edge. At this point only the 6-cycle is left, so further trimming would be pointless.

After all edge trimming rounds, the counter memory is freed, and allocated to a custom cuckoo_hashtable

[3]excluding the very least significant bit distinguishing even from odd nodes.

(based on [13]) 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 bounded away from 1; e.g. under 90 percent.

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

## 12 Time-Memory Trade-Offs (TMTOs)

David Andersen also 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, which, unlike the main algorithm, can trade-off memory directly for runtime. On the downside, to even achieve memory parity with the main algorithm, it already incurs a big slowdown. To the extent that this slowdown is unavoidable, it can be called the *memory hardness* of the proof-of-work.

The TMTO algorithm selects a suitably small subset $Z$ of even vertices as a base layer, and on top of that builds a breadth-first-search (BFS) forest of depth $L/2$, i.e. half the cycle length. For each new BFS layer, it enumerates all edges to see which ones are incident to the previous layer, adding the other endpoint. It maintains a directed forest on all BFS nodes, like the base algorithm does on all nodes. For increased efficiency, the base layer $Z$ is filtered for nodes with multiple incident edges. If the graph has an $L$-cycle one of whose nodes is in $Z$, then the above procedure will find it. If one choice of $Z$ doesn't yield a solution, then the data structures are cleared and the next subset is tried.

A variation on the above algorithm omits the filtering of $Z$, and expands the BFS to a whole $L$ levels. This way, an $L$-cycle will be found as long as the distance from (any node in) $Z$ to the cycle is at most $L/2$. It thus has a much higher chance of finding a cycle, but requires more space to store the significantly bigger BFS forest.

For each each value of $L \in \{2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, 32, 40, 48, 56, 64\}$ we ran these 2 algorithms on 200 graphs of size $2^{25}$ that include an $L$-cycle, choosing subset size as a 2-power that results in a memory usage of 4MB, and analysed the distribution of number of subsets tried before finding a solution. Since there is possible overlap between the BFS forests of different initial subsets, especially with the second algorithm, the distributions are skewed toward lower numbers. To maximize solution finding rate then, it pays to give up on a graph when the first few subsets tried fail to provide a solution. For each algorithm and cycle length, we determined the minimum number of tries needed to guarantee solutions in at least 50 of the 200 graphs. In Figure 5 we plot the slowdown relative to the reference algorithm also using 4MB (2MB for edges and 2MB for nodes).

The zigzagging is caused by the current implementation being limited to 2-power sizes of both subsets and cuckoo tables while the load of the latter is kept between 45% and 90. The BFS($L$) algorithm exhibits at least one order of magnitude slowdown, that grows very slowly with cycle length, while the BFS($L/2$) algorithm exhibits roughly linear slowdown. Assuming that these algorithms cannot be significantly improved upon, this shows Cuckoo Cycle with larger cycle lengths satisfying property MB4,

## 13 Choice of cycle length

A cycle of length 2 means that two nonces produce identical edge endpoints—a *collision* in edge space. The Momentum proof-of-work looks for collisions on 50 bits of hash output among $2^{26}$ nonces. This is in essence Cuckoo Cycle with $N = 2^{25} + 2^{25}$ nodes and cycle length $L = 2$, with two differences.

First, edges are generated not by equation (2), but by splitting a SHA512 hash of $(k, \text{nonce}/8)$ into 8 64-bit words, taking the most significant 50 bits of the $(\text{nonce} \bmod 8)th$ one, and viewing that as a pair of two 25-bit edge endpoints, appending a bit to make them even and odd.

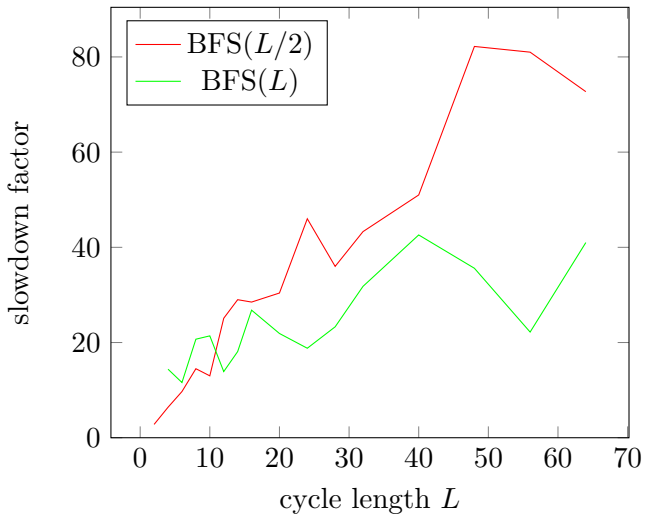


Figure 5: Reduction in solution finding rate for two TMTO algorithms

Second, the choice of $M = 2^{26}$ gives a ratio $\frac{M}{N}$ of 1 rather than $\frac{1}{2}$ and as such prohibits the use of edge trimming.

Since the extreme case of $L = 2$ is so special, there is likely to be a greater variety of algorithms that are more efficient than for the general case. While we haven't found (and don't know of) an improved main algorithm, we did find an improved BFS($L/2$) TMTO algorithm (implemented in momentomatum.cpp) that cuts the memory usage in half, resulting in a slowdown of only 1.75—a lack of memory-hardness.

The preceding analysis suggests that cycle length should be at least 20 to guard against the more efficient BFS($L/2$) algorithm, with an additional safety factor of 2.

In order to keep proof size manageable, the cycle length should not be too large either. We thus consider 20-64 to be a healthy range, and suggest the use of the average of 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.

## 14 Parallelization

All our implementations allow the number of threads to be set with option `-t`. For $0 \leq t < T$, 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 [14], 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.

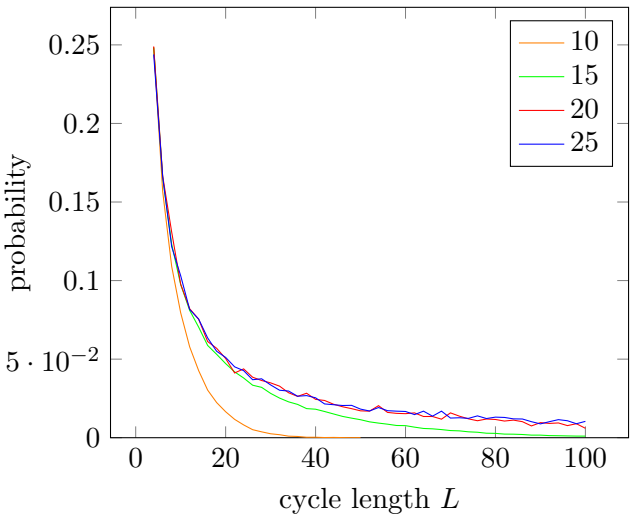


Figure 6: Distribution of cycle lengths in random graphs

In edge trimming, parallelization is achieved by partitioning the set of edges. To maintain efficient access to the bitmap of live edges, each thread handles words (of 32 edge-bits each) spaced $T$ apart.

Atomic access is used by default for accessing the 2-bit counters. Disabling this results in a small chance of removing multiple edges incident to a node that access the counter at the same time.

The implementation further benefits from bucketing the addresses of counters to be updated or tested, based on their most significant bits. Thus, when a bucket becomes full and is emptied by actually performing those updates/tests, the accesses are limited to a certain address range, which turns out to reduce memory access latencies.

The plots below show the speedup over single thread performance achieved by multithreading at various graph sizes and counter-partition levels.

## 15 Choice of graph size

For cryptocurrency purposes, the choice of Cuckoo graph size should be in accordance to its block interval 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. This desirable property is known as *progress-freeness*, and in our case is achieved more easily with a small graph (and hence memory) size.

Larger memory sizes have two advantages. Beyond satisfying property MB1, they also 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, so botnet operators can be expected to eschew memory bound PoWs in favor of low-memory ones.

We expect these opposing goals to lead to graph sizes from $2^{28}$ to $2^{32}$, with the larger ones geared more toward longer block interval times and faster mining hardware.

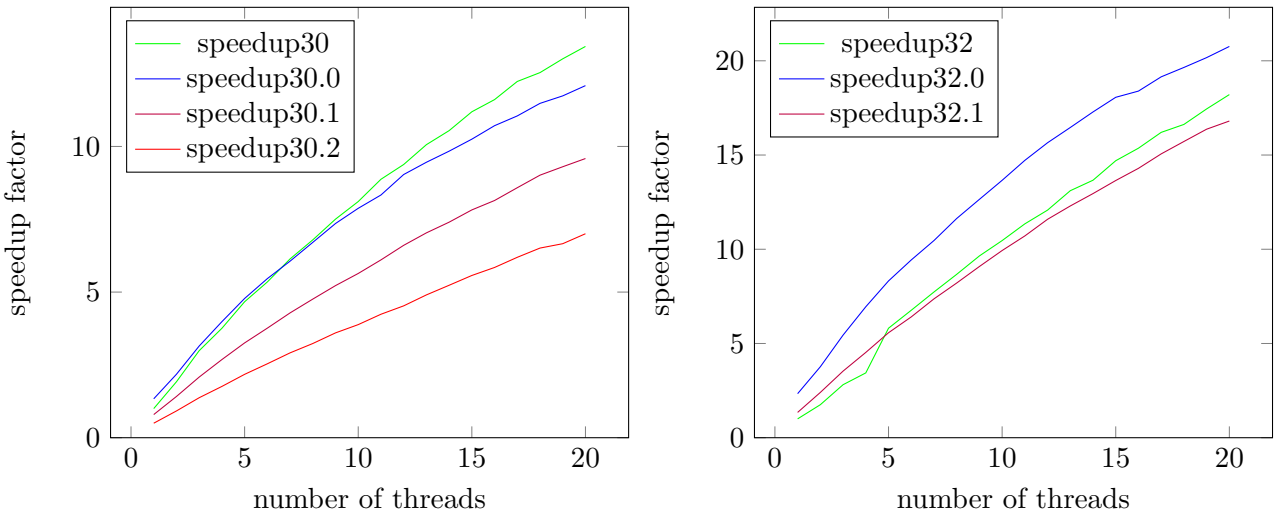


Figure 7: Multi-threading speedup

## 16 Dynamic Sizing

Ideally, graph size should grow with evolving memory chip capacities, so as to preserve property MB1. Although these have shown remarkable adherence to Moore's Law in the past, this cannot be relied on for the more distant future. We therefore propose to re-evaluate the graph size every so-many difficulty adjustments. If the difficulty target is sufficiently low, then the graph size is deemed to have become "too easy" for existing hardware, and gets doubled.

In order to make this transition smoother and avoid severe loss of proof-of-work power, we propose having a range of sizes allowed at any time, namely $k$ consecutive 2-powers for some small number $k \geq 2$. As with Myriad-coin, separate difficulty controls are maintained for each size, adjusted so that each size accounts for roughly $\frac{1}{k}$ of all blocks.

Doubling graph sizes is then equivalent to disabling the smallest 2-power, and enabling a new largest one, whose initial difficulty target is twice that of the previous largest. Even if none of the hardware that was working on the smallest 2-power is repurposed for a larger size, since this hardware only accounted for a fraction $\frac{1}{k}$ of the rewards, the loss of proof-of-work power should be acceptable.

It remains to decide what exact form the "difficulties too low" condition should take.

## 17 Conclusion

Cuckoo Cycle is a novel graph-theoretic proof-of-work design that combines scalable memory requirements with instant verifiability, and the first where memory latency dominates the runtime.

Barring any unforeseen memory-time trade-offs, it makes for a near-ideal memory bound proof-of-work whose cost effectiveness on commodity hardware could greatly benefit decentralization of mining.

## 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] Lolcust, "[announce] tenebrix, a cpu-friendly, gpu-hostile cryptocurrency," Sep. 2011. [Online]. Available: https://bitcointalk.org/index.php?topic=45667.0

[4] coblee, "[ann] litecoin - a lite version of bitcoin. launched!" Oct. 2011. [Online]. Available: https://bitcointalk.org/index.php?topic=47417.0

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

[6] D. Larimer, "Momentum - a memory-hard proof-of-work via finding birthday collisions," Tech. Rep., Oct. 2013. [Online]. Available: www.hashcash.org/papers/momentum.pdf

[7] A. Biryukov and D. Khovratovich, "Equihash: Asymmetric proof-of-work based on the generalized birthday problem," Cryptology ePrint Archive, Report 2015/946, 2015, https://eprint.iacr.org/2015/946.

[8] P. C. van Oorschot and M. J. Wiener, "Parallel collision search with cryptanalytic applications," *J. Cryptology*, vol. 12, no. 1, pp. 1–28, Jan. 1999.

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

[10] 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

[11] 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

[12] 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

[13] 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/

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

# 18 Appendix A: cuckoo.h

```
// Cuckoo Cycle, a memory-hard proof-of-work
// Copyright (c) 2013-2017 John Tromp

#include <stdint.h> // for types uint32_t,uint64_t
#include <string.h> // for functions strlen, memset
#include <stdarg.h>
#include <stdio.h>
#include <chrono>
#include <ctime>
#include "../crypto/blake2.h"
#include "../crypto/siphash.hpp"

#ifdef SIPHASH_COMPAT
#include <stdio.h>
```

```
#endif

// proof-of-work parameters
#ifndef EDGEBITS
// the main parameter is the 2-log of the graph size,
// which is the size in bits of the node identifiers
#define EDGEBITS 29
#endif
#ifndef PROOFSIZE
// the next most important parameter is the (even) length
// of the cycle to be found. a minimum of 12 is recommended
#define PROOFSIZE 42
#endif

// save some keystrokes since i'm a lazy typer
typedef uint32_t u32;
typedef uint64_t u64;

#ifndef MAX_SOLS
#define MAX_SOLS 4
#endif

#if EDGEBITS > 30
typedef uint64_t word_t;
#elif EDGEBITS > 14
typedef u32 word_t;
#else // if EDGEBITS <= 14
typedef uint16_t word_t;
#endif

// number of edges
#define NEDGES ((word_t)1 << EDGEBITS)
// used to mask siphash output
#define EDGEMASK ((word_t)NEDGES - 1)

// Common Solver parameters, to return to caller
struct SolverParams {
        u32 nthreads = 0;
        u32 ntrims = 0;
        bool showcycle;
        bool allrounds;
        bool mutate_nonce = 1;
        bool cpuload = 1;

        // Common cuda params
        u32 device = 0;

        // Cuda-lean specific params
        u32 blocks = 0;
        u32 tpb = 0;

        // Cuda-mean specific params
        u32 expand = 0;
        u32 genablocks = 0;
        u32 genatpb = 0;
        u32 genbtpb = 0;
        u32 trimtpb = 0;
        u32 tailtpb = 0;
        u32 recoverblocks = 0;
        u32 recovertpb = 0;
```

```
};

// Solutions result structs to be instantiated by caller,
// and filled by solver if desired
struct Solution {
 u64 nonce = 0;
 u64 proof[PROOFSIZE];
};

struct SolverSolutions {
 u32 edge_bits = 0;
 u32 num_sols = 0;
 Solution sols[MAX_SOLS];
};

#define MAX_NAME_LEN 256

// last error reason, to be picked up by stats
// to be returned to caller
char LAST_ERROR_REASON[MAX_NAME_LEN];

// Solver statistics, to be instantiated by caller
// and filled by solver if desired
struct SolverStats {
        u32 device_id = 0;
        u32 edge_bits = 0;
        char plugin_name[MAX_NAME_LEN]; // will be filled in caller-side
        char device_name[MAX_NAME_LEN];
        bool has_errored = false;
        char error_reason[MAX_NAME_LEN];
        u32 iterations = 0;
        u64 last_start_time = 0;
        u64 last_end_time = 0;
        u64 last_solution_time = 0;
};

// generate edge endpoint in cuckoo graph without partition bit
word_t sipnode(siphash_keys *keys, word_t edge, u32 uorv) {
  return keys->siphash24(2*edge + uorv) & EDGEMASK;
}

enum verify_code { POW_OK, POW_HEADER_LENGTH, POW_TOO_BIG, POW_TOO_SMALL, POW_NON_MATCHING, POW_BRA
const char *errstr[] = { "OK", "wrong header length", "edge too big", "edges not ascending", "endpo

// verify that edges are ascending and form a cycle in header-generated graph
int verify(word_t edges[PROOFSIZE], siphash_keys *keys) {
  word_t uvs[2*PROOFSIZE];
  word_t xor0 = 0, xor1 = 0;
  for (u32 n = 0; n < PROOFSIZE; n++) {
    if (edges[n] > EDGEMASK)
      return POW_TOO_BIG;
    if (n && edges[n] <= edges[n-1])
      return POW_TOO_SMALL;
    xor0 ^= uvs[2*n  ] = sipnode(keys, edges[n], 0);
    xor1 ^= uvs[2*n+1] = sipnode(keys, edges[n], 1);
  }
  if (xor0|xor1)              // optional check for obviously bad proofs
    return POW_NON_MATCHING;
  u32 n = 0, i = 0, j;
  do {                        // follow cycle
```

```
    for (u32 k = j = i; (k = (k+2) % (2*PROOFSIZE)) != i; ) {
      if (uvs[k] == uvs[i]) { // find other edge endpoint identical to one at i
        if (j != i)           // already found one before
          return POW_BRANCH;
        j = k;
      }
    }
    if (j == i) return POW_DEAD_END;  // no matching endpoint
    i = j^1;
    n++;
  } while (i != 0);           // must cycle back to start or we would have found branch
  return n == PROOFSIZE ? POW_OK : POW_SHORT_CYCLE;
}

// convenience function for extracting siphash keys from header
void setheader(const char *header, const u32 headerlen, siphash_keys *keys) {
  char hdrkey[32];
  // SHA256((unsigned char *)header, headerlen, (unsigned char *)hdrkey);
  blake2b((void *)hdrkey, sizeof(hdrkey), (const void *)header, headerlen, 0, 0);
#ifdef SIPHASH_COMPAT
  u64 *k = (u64 *)hdrkey;
  u64 k0 = k[0];
  u64 k1 = k[1];
  printf("k0 k1 %lx %lx\n", k0, k1);
  k[0] = k0 ^ 0x736f6d6570736575ULL;
  k[1] = k1 ^ 0x646f72616e646f6dULL;
  k[2] = k0 ^ 0x6c7967656e657261ULL;
  k[3] = k1 ^ 0x7465646279746573ULL;
#endif
  keys->setkeys(hdrkey);
}

// edge endpoint in cuckoo graph with partition bit
word_t sipnode_(siphash_keys *keys, word_t edge, u32 uorv) {
  return sipnode(keys, edge, uorv) << 1 | uorv;
}

u64 timestamp() {
        using namespace std::chrono;
        high_resolution_clock::time_point now = high_resolution_clock::now();
        auto dn = now.time_since_epoch();
        return dn.count();
}

////////////////////////////////////////////////////////////////
// Declarations to make it easier for callers to link as required
////////////////////////////////////////////////////////////////

#ifndef C_CALL_CONVENTION
#define C_CALL_CONVENTION 0
#endif

// convention to prepend to called functions
#if C_CALL_CONVENTION
#define CALL_CONVENTION extern "C"
#else
#define CALL_CONVENTION
#endif

// Ability to squash printf output at compile time, if desired
```

```
#ifndef SQUASH_OUTPUT
#define SQUASH_OUTPUT 0
#endif

void print_log(const char *fmt, ...) {
        if (SQUASH_OUTPUT) return;
        va_list args;
        va_start(args, fmt);
        vprintf(fmt, args);
        va_end(args);
}
////////////////////////////////////////////////////////////////
// END caller QOL
////////////////////////////////////////////////////////////////
```