Exploiting Time-Memory Tradeoffs in Cuckoo Cycle

David G. Andersen
Carnegie Mellon University

Preliminary working draft of 1 August 2014. This paper is a work-in-progress and is being released to stimulate discussion. It hasn't even been spellchecked, so proceed with caution!

Abstract

Cuckoo Cycle is a recently proposed "memory-hard" computational Proof-of-Work (PoW) function designed to require a minimum amount of memory to compute efficiently. This paper describes two algorithms that use the sparsity of the Cuckoo Cycle graph to use less memory than both the initially hypothesized and updated proposed bounds for this proof-of-work function. The second algorithm presents a near-linear time/memory tradeoff for the most computationally intensive part of solving the Cuckoo Cycle proof-of-work function.

1 Introduction

Proof-of-Work functions have been proposed for (and are used in) applications from cryptocurrencies, spam and Sybil attack prevention, and other scenarios where it is beneficial to combine anonymity with some form of rate-limiting [3, 2].

The core idea behind a Proof-of-Work (PoW) is that a sender (S) can prove to a recipient (R) that S has, in expectation, expended a certain amount of computational effort, and that R can verify this in fast, constant time. Using the example of spam prevention, the recipient of an email might ask the sender to prove that it spent 1 second of CPU time, ensuring that a normal human sender would be unaffected, but a spam sender could send no more than 60 messages per minute.

In several areas, "memory-hard" PoW functions are deemed particularly attractive, for reasons including: (1) A smaller performance gap between fast computers and slower mobile devices; (2) A smaller performance gap between a software implementation running on CPUs and specialized optimizations using graphics processors (GPUs) or custom silicon ASICs. This concern has become particularly germane for cryptocurrencies with the arrival of custom ASIC-based "miners" for two of the most popular proof-of-work functions.

The recently proposed Cuckoo Cycle [**?**] is designed to be a "memory-hard" PoW function. In this paper, we demonstrate both a high-performance, memory-efficient implementation of the baseline version of Cuckoo Cycle that uses an order of magnitude less memory than it was initially claimed to require, and demonstrate an effective time/memory tradeoff that could potentially lead to ASIC implementations.

2 Cuckoo Cycle Basics

The "proof" in the Cuckoo Cycle PoW is to find a cycle of length Lin a random sparse bipartite graph. At a glance, such a formulation has several attractive properties: It intuitively requires at least $O(N)$ steps to evaluate, and the random connections between vertices would seem to induce a requirement for either random accesses or sorting-based algorithms. In turn, verification can still be fast, as verifying that a cycle exists is $O(L)$.

The concrete formulation of the Cuckoo Cycle problem:

Given an input value X, create a hash $\mathbf{H} = \text{Hash}(X, nonce)$. $\mathbf{H}$ will serve as the generator for a bipartite graph G in which to find a cycle. A successful output from the Cuckoo Cycle problem is a pair $\langle \text{nonce}, \text{edgelist} \rangle$ such that the edges in edgelist form a cycle of length L in the graph G defined by $\mathbf{H}$. To be able to tune the difficulty of finding a solution, a further requirement is imposed: $\text{SHA256}(\text{sorted edgelist}) < T$, where T is a target difficulty. A smaller target difficulty will reject more otherwise-successful solutions, requiring more executions of the Cuckoo Cycle finder.

G is a bipartite graph with N nodes on each side and N edges between them (as suggested by the author). The edges are accessed by an oracle that, given an input index $1 <= i <= N$, outputs the numerical index of the left-hand and right-hand nodes in the graph. This oracle is $Hash(\mathbf{H}, 2*i+0/1)\%N$, where the suggested hash function is SIPhash [1], a fast but reasonably-secure short hash function. The left hand node is found by hashing $2*i+0$ and the right by hashing $2*i+1$.

Notably, the graph G can therefore be represented compactly using only N and $\mathbf{H}$, but identifying all edges incident upon a particular vertex requires iterating through the entire edge set at least once.

The initial proposal for Cuckoo Cycle provided a union-find-based algorithm for efficiently finding a cycle of length L in such a graph. It ran in roughly $O(N)$ time using $N*$

64 bits of memory to represent the graph. In the remainder of this paper, we assume the existence of such a solver, but perform extensive pre-processing, called "edge trimming", before invoking it, to achieve substantially faster runtime and reduced memory requirements.

3 A Compact Edge Trimming Algorithm

The first optimization for Cuckoo Cycle arises from the observation that the graph is very sparse: With N vertices on each side and only N edges in total, roughly $1/e$ (36.7%) of the bins on each side are empty. A vertex can only participate in cycle if it has at least two incident edges, and, thus, nearly $2/e$ of the bins cannot be in a cycle based only upon an immediate count of incident edges, by the Poisson approximation to the Binomial. As a consequence, a single iteration through the edges in $2N$ steps can reduce the number of active vertices on each side by nearly 73%.

From there, it is a simple recursive formulation to derive a fast and compact edge trimming algorithm: Begin by defining all nodes as live. Count all incident edges *from live nodes*. Mark as dead any node that does not have 2 or more such incident edges. Repeat until the desired degree of trimming has been achieved.

This can be implemented compactly and fast by using a bitvector to represent the liveness of the nodes (optionally one more to track edge liveness to allow rapidly skipping unnecessary edges), and a saturating two-bit counter bitvector for counting edges incident upon each vertex. The latter is helpful because a node remains live whether it has 2 or 200 edges incident upon it, so it is only necessary to count to two, and thus, only two bits are required.[1]

Pseudocode for this algorithm is shown in Algorithm 1.

Algorithm 1 Edge Trimming

```
1:  procedure EdgeTrim(keys)
2:      liveLeft ← BitVector(true, N)
3:      liveRight ← BitVector(true, N)
4:      for iter = 1 to MaxIters do
5:          count ← IntVector(0, N)
6:          for edge = 1 to N do
7:              if liveRight[edge.Right] then
8:                  count[edge.Left]++
9:              end if
10:         end for
11:         for node = 1 to N do
12:             if count[i] < 2 then
13:                 liveLeft[i] = False
14:             end if
15:         end for
16:     end for
        (Repeat equivalently for the right side)
17: end procedure
```

4 A Time-Memory Tradeoff

In prior memory-hard functions such as scrypt [**?**], time/memory tradeoffs have been used to enable efficient ASIC-based implementations, using fast custom circuitry to eliminate expensive storage and memory bandwidth requirements. A specific goal of Cuckoo Cycle was to prevent such tradeoffs by having superlinear scaling: If reducing the memory us from N to $\frac{N}{k}$ should increase the amount of computation by, e.g., k^2 or some other superlinear factor.

This section presents an algorithm for achieving a near-linear time/memory tradeoff for edge pruning in Cuckoo Cycle. Note that this speedup applies only to the edge-pruning step, and that the pruned graph must still be processed separately, with about 1-3% of the remaining live nodes that the original graph had. This result graph can be streamed as a sequential memory write, which makes it reasonably efficient for use on GPUs and ASICs that have an attached memory controller, but it is an important current limitation.

To begin, define a "node partition" P as a subset of the nodes on one side of the bipartite graph. The node partition is the unit of the TMTO: Picking $|P| = \sqrt{N}$, for example, will thus require $c * \frac{N}{\sqrt{N}} \log(N)$ bits of memory for edge partitioning, where c is a small constant.

The approach is straightforward.

1. Establish liveness for each node in P as in Algorithm 1: Iterate through the edge set, count edges incident to each node in P, and mark as "dead" those with fewer than 2 incident edges. This step operates in $O(N)$ time and requires $3|P|$ bits of memory for the counter and live bitvectors. At the end of this step, there are roughly $\frac{|P|}{e}$ live nodes remaining.
2. For every edge in G, if it is connected to a live node in P, add its other endpoint to a dictionary D. D tracks the identities of those nodes responsible for "keeping alive" nodes in P. This step requires $O(N)$ time and roughly $|P| \log N$ bits of memory. The logarithmic factor arises because the dictionary must store the full node ID of the nodes it tracks.
3. Perform liveness tracking for every node in D, as above. Remove from D any node with less than 2 incident edges. This eliminates roughly half of the nodes in D (each already had one incident edge from P, so this reduction is smaller than the initial pruning).
4. Re-calculate liveness for the nodes in P, only incrementing the count of incident edges if the other endpoint of the edge in D is still live.
5. Apply this idea recursively to P-D1-D2, P-D1-D2-D3,

[1] A "trit"-style succinct representation is likely to be unappealingly slow in practice, despite the waste of 0.5 bits per entry in this simple representation.

etc.

6. Once sufficient iterations have been performed on P, move on to the next subset, discarding the data structures associated with P and writing the few remaining live elements from P to a list.

At the end of this step, about $l = 1 - 3\%$ of the items in P are still live. They can be represented using $l\log(|P|)$ bits, because it is only necessary to represent their offset within the partition. The requirement to represent these remaining live nodes thus provides the remaining lower bound on the memory required to solve the cuckoo cycle problem, of roughly $0.02N\log(|P|)$ bits.

The memory requirements for the recursive dictionary are approximately constant (for a given $|P|$) because (1) at each successive iteration, the initial number of nodes is smaller than in the previous, thus requiring less per-hop dictionary space; and (2) the sparsity of the graph ensures that the expected number of nodes at further-out hops does not grow as the graph extends.[2]

Several optimizations remain possible after this initial optimization:

- Use the liveness established on prior partitions when computing the liveness of later partitions.
- Hierarchically combine partitions for further pruning in an efficient manner once their size is sufficiently small.

The former would likely require writing the remaining nodes using an efficient encoding such as an Elias-Fano sequence. The latter's effectiveness is an open question to explore.

A proof-of-concept implementation of this for a single partition can be found at http://www.cs.cmu.edu/~dga/crypto/cuckoo/partitioned.jl . While not performance-optimized (and written in Julia), it covers the basics of the algorithm.

5 Discussion

The algorithms shown thus far are preliminary and the analysis somewhat uncareful, but I believe they serve as a good starting point for understanding future TMTO attacks against sparse graph proof-of-work functions such as Cuckoo Cycle.

In a very preliminary sense, there should be few barriers to implementing the described algorithm in hardware: It requires only bitvectors and dictionaries mapping integers to small counts, the latter of which can be handled efficiently using content-addressable memories (CAMs), conventional data structures, or slightly overprovisioned associative caches.

[2] This analysis is too glib and stops being true as the graph is pruned asymptotically towards containing cycles and very-long paths. But it works well for the first few iterations. n.b. - a very careful version would bound the maximum size instead of dealing only with the expected, or show that the probability of needing to skip and move on to the next graph instance is low.

The largest barrier to overcome yet is the need to pass the pruned graph to a more complex solver. This is an interesting question for further exploration: it is very likely that the sparsity of the graph will save the day again here, because many of the small cycles that might keep nodes alive after edge pruning are unlikely to be involved in a successful 42-cycle and could possibly be detected and pruned with the addition of a small amount of extra bookkeeping. As an alternative, of course, an ASIC-based implementation might provide a small embedded general purpose ARM-style core to handle this final solution-finding without the need to go off-chip.

Based upon these results, I continue to believe that much more attention needs to be paid to this function before it is adopted as a proof-of-work based upon its time/memory tradeoff resistance claims. It remains an intriguing idea, but the very algorithmic issues that make it *interesting* also mean that without a more solid proof of its difficulty, it may still yield to further algorithmic improvements.

6 Conclusion

This paper demonstrates an efficient implementation strategy for Cuckoo Cycle that reduces the memory requirements by over an order of magnitude compared to the initially hypothesized bounds. It further presents a partial nearly-linear time/memory tradeoff attack that is effective against the edge trimming phase that constitutes the great majority of time spent solving a cuckoo cycle problem instance.

References

[1] J.-P. Aumasson and D. J. Bernstein. SipHash: a fast short-input PRF. In *Progress in Cryptology-INDOCRYPT*, pages 489–508, 2012.

[2] A. Back. Hashcash. http://www.cypherspace.org/hashcash/, 1997.

[3] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Advances in CryptologyCRYPTO92*, pages 139–147. Springer, 1993.