MOMENTUM - A MEMORY-HARD PROOF-OF-WORK VIA FINDING BIRTHDAY COLLISIONS

DANIEL LARIMER
dlarimer@invictus-innovations.com
Invictus Innovations, Inc

ABSTRACT. We introduce the concept of memory-hard proof-of-work algorithms and argue that in order for proof-of-work based systems to be secure against attacks using custom hardware, they should be constructed from memory-hard functions. Proof-of-work is an economic measure that is designed to be expensive to find but cheap to verify. Existing solutions used for proof-of-work are either trivial to parallelize or too slow to verify when tuned to use gigabytes of memory. We present a new family of proof-of-work algorithms that can be validated in less than a millisecond but can demand any amount of memory for finding solutions in a practical manner.

1. INTRODUCTION

Proof-of-work schemes are designed to be hard to solve but relatively easy to verify. Unfortunately, most approaches to proof-of-work achieve their fast verification by simply verifying one round of an embarrassingly parallel search algorithm. These embarrassingly parallel algorithms are quickly adapted to graphics cards, FPGAs, or even ASIC designs which would give an attacker several orders of magnitude advantage over the common computer.

In the case of crypto-currencies where the primary goal of the proof-of-work is decentralization of trust it becomes critical that the proof-of-work cannot be optimized and accelerated by FPGA or ASIC designs with any meaningful economic return on investment. One approach that has been adopted by many crypto-currencies is to use what is known as a Sequential Memory-Hard Function[1], such as Scrypt. These algorithms attempt to make the proof-of-work dependent upon sequential random access to a large array of data and thus be memory constrained which limits parallelization. The challenge with sequential memory-hard functions is that when they

[1] http://www.tarsnap.com/scrypt/scrypt.pdf

are tuned to use large amounts of memory they lose the property of being easy to verify. For example, simply populating 1 Gigabyte of memory with crypto-graphically secure pseudorandom data can take a second to perform. As a result the requirement to validate such a memory-hard proof-of-work would create an opportunity to perform a denial of service attack.

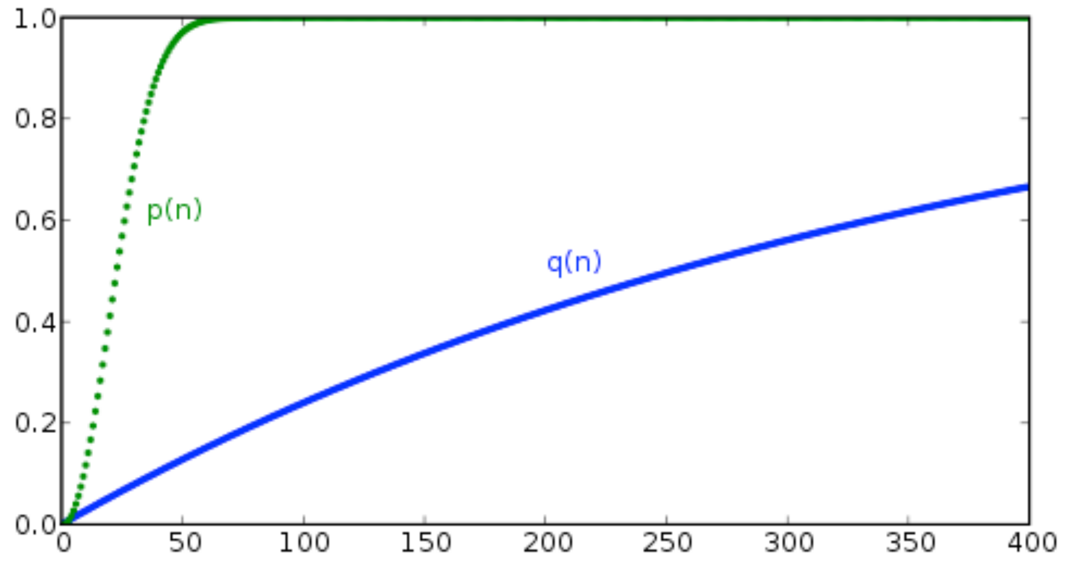
This paper introduces a family of memory-hard proof-of-work algorithms that can be validated in milliseconds while requiring gigabytes of memory to solve efficiently.

2. MEMORY-HARD PROOF-OF-WORK

To achieve the goal of being trivial to verify but memory intensive to solve, the proof-of-work must have asymmetry in the amount of memory required to validate the work. As a consequence, the individual steps of the proof-of-work must be embarrassingly parallel because they are the foundation of the validation step. Despite embarrassingly parallel steps that can be run in less than a millisecond, algorithms can be made memory-hard by requiring a solution that depends upon the relationship between any two or more of the parallel steps and thereby benefits from the storage of the result of every parallel step. The results can be quickly verified by performing just two or three parallel steps and checking the relationship between the results produced.

The most straightforward example is finding collisions based upon the Birthday Problem [3]. In probability theory, the birthday problem concerns the probability that in a set of *n* randomly chosen people, some pair of them will have the same birthday. By the pigeonhole principle, the probability reaches 100% when the number of people reaches 367. However, 99% probability is reached with just 57 people, and 50% probability with 23 people. These conclusions assume that each day of the year is equally probable for a birthday.

Figure 1 shows the significant benefit achieved by remembering all *n* solutions in the search for a matching pair of birthdays. If you were to expand the number of days in the year to 128 billion and selected birthdays in a cryptographically secure pseudo-random manner then the amount of memory required to efficiently find a matching pair would be on the order of 45 megabytes. However, if you increase the requirement to finding 3 people with the same birthday then the memory requirements exceed 1 gigabyte on average. Any attempt to replace memory with computation would force the algorithm to follow the *q(n)* curve which is at such an algorithmic disadvantage that massive parallelism cannot overcome the need for memory to efficiently solve this problem. The best one could hope for is to generate potential matches in parallel, but the results would have to be stored for most efficient solutions.



p(n) = probability of a match q(n) = probability of matching your birthday

Figure 1

The most straightforward and parallel solution would require an array of 128 billion items and after finding each potential result check to see if there is an item in memory at that location. This approach would require 500 Gigabytes of RAM and is thus impractical. An alternative is to use a hash table which would dramatically reduce the amount of memory required. Unfortunately for the potential attacker, such a hash table becomes a source of lock contention or slow atomic operations. The embarrassingly parallel birthday generation step is hobbled by the need to synchronize storage in the hash table. This last synchronization step places a limit on the amount of parallelism that can be employed.

While hash tables force some serialization to the process of finding birthday matches, there exist many means to minimize the overhead associated with such serialization. If you consider the potential of simply ignoring random data corruption and verifying your results when a potential match is found it becomes clear that the birthday problem combined with a hash table is better than current proof-of-work systems, but has room for improvement if a means can be found to define an additional constraint on the collision that forces all parallel operations through a common mutex.

3. THE ALGORITHM

Assuming a cryptographically secure hashing function Hash(x) and a Sequental-Memory-Hard hashing function BirthdayHash(x) such as scrypt the algorithm for this proof of work can be defined as follows:

Given a block of data D, calculate H = Hash(D).
Find nonce A and nonce B such that BirthdayHash(A +H) == BirthdayHash(B+H)
If Hash(H + A + B) < TargetDifficulty then a result has been found, otherwise keep searching.

4. SCALING DIFFICULTY

For crypto-currencies, it is not enough that the proof of work be memory hard, it must also be flexible enough to scale the difficulty of the work to finely tune the block production rate. For this reason the final step of the proof of work is to perform the hash of the data and both birthday nonces and then check to see if the resulting hash is below a target threshold. This final step behaves just like Bitcoin or Scrypt based proof of work systems.

We speculate that the best of all results is to combine the Birthday Search with a traditional sequential memory-hard function such as Scrypt that also leverages hardware accelerated algorithms like AES to ensure that even the most fundamental, embarrassingly parallel, step of the Birthday Search, the generation of birthdays, is non-trivial to implement on an ASIC in a manner more efficient than a CPU.

5. EMERGENT PROPERTIES

With traditional proof-of-work systems likes SHA256 or Scrypt it was possible to gain performance through parallelism alone. However, regardless of how efficiently an ASIC can run the individual birthday hash steps in parallel, your use of memory must scale to store the result of every parallel run or you will lose an algorithmic advantage that cannot be overcome by increasing levels of parallelism. If one were to create a Scrypt ASIC capable of 1 Giga-hash / second then one would require 10 terabytes of RAM to most efficiently find one solution at the target difficulty every 10 minutes on average.

The performance of the proof-of-work increases the longer it runs as it fills up memory with potential birthday matches. As a result the efficiency of the algorithm has some *momentum* to it which makes it expensive to restart the search with a new block of data. This property has some very useful side effects for block chain based systems where the miner could gain some advantage by adjusting the structure of the block they are mining every time new data is available. The most efficient mining strategy is to cache all transactions that you receive while mining the current block until someone finds the current block, and then create a new block with all of the cached transactions. This property means that a transaction broadcast 5 seconds after the last block was found has no advantage over a transaction broadcast 5 minutes after the last block was found because few miners will begin working on including either transaction until the next block is found. It is because of the *momentum* property that we have named this proof of work system Momentum.

6. CONCLUSIONS

We have introduced a new class of memory-hard proof-of-work algorithms that are asymmetric in the memory and time requirements for finding a solution in comparison to verifying the solution and which contain a significant amount of sequential operations. These algorithms are well suited for block-chain based proof-of-work systems or stronger key derivation functions.

REFERENCES

[1] Colin Percival. Stronger Key Derivation Via Sequential Memory-Hard Functions
[2] Fabien Coelho. Exponential Memory-Bound Functions for Proof of Work Protocols
[3] Roberts Matthews, Fiona Stones. Coincidences: the truth is out there.
[4] Adam Back. Hashcash - A Denial of Service Counter-Measure
[5] Sergio Demian Lerner - Mavepay, A new Lightweight payment Scheme for Peer to Peer Currency Networks
http://bitslog.files.wordpress.com/2012/04/mavepay1.pdf