Block Chain Based Proof-of-Work Hash
and Wild Keccak as a Reference Implementation

Boolberry Team

August 1, 2014

Introduction

The cornerstone of all Proof-of-Work (PoW) based cryptocurrencies is the hash function used to confirm that work was done. It is used within the currency to enable a *decentralized* group of mutually untrusting participants to agree on a consistent transaction history and protect against double-spending. To do so, the currency requires participants to prove that they have "wasted" a certain amount of computational power by presenting a proof to the PoW function. For this work, the participants receive a reward in the form of generated coins. Ultimately, the cost of coin generation becomes comparable to energy consumption, coupled with the depreciation of the equipment.

However, in the real world many hash functions were easily optimized to work not only on the classical CPU, but also the GPU, and later ASICs. This progression has led to certain problems, such as an unequal distribution of generated coins as well as the centralization of network hashrate, which in turn makes it possible for one entity to double-spend.

The problem

Currently, one of the main problems of Proof-of-Work projects is ASIC miners -- custom chips designed and manufactured specifically to solve the PoW as fast and efficiently as possible. Production of ASIC miners, which provide hash rates several orders of magnitude higher than CPUs and GPUs, appear to give a small group of individuals the ability to perform most of the work. This in turn casts doubt on the reliability and safety of the network as a payment system. Moreover, the introduction of ASIC-based mining creates a sudden drop in mining profitability for those with conventional equipment, which further exacerbates the problem of centralization and can significantly affect pricing. Bitcoin's hash function, SHA-256, was the first to succumb to the problem.

Existing solutions

Many hash algorithms have been created or used in an attempt to mitigate this problem. Litecoin is likely the most well-known Bitcoin fork created with the goal of resisting GPU and ASIC development. It used **SCrypt** in its PoW algorithm in its attempt to achieve this. SCrypt is a

sequential memory-hard function requiring asymptotically more memory (128KB) than the original bitcoin algorithm [1]. Despite its memory-hardness, it soon became clear that SCrypt was easier to implement in GPU than originally thought. ASIC implementations followed soon thereafter, but despite its original goal, the Litecoin development team rejected performing a “hard fork” to avoid the chaos that accompanies a mandatory upgrade by nearly all participants in a large decentralized system [2].

Another solution to the problem is to combine different algorithms as done by **Quark**. Quark used nine rounds of six different hash functions in an attempt to keep the GPU miners away. The Quark solution worked for less than three months.

The **X11** developer, Evan Duffield, also combined different algorithms but with a different goal. Evan has written on several occasions that X11 was integrated into Darkcoin not with the intention to prevent ASIC manufacturers from creating ASICs for X11 in the future, but rather to provide a similar migratory path that Bitcoin had (CPUs, GPUs, ASICs). He expects that eventually, as Darkcoin grows in market capitalization, and ASIC investments becomes profitable, ASICs will be developed [3].

Several functions were designed to solve specific computational problems. For example, the **Primecoin** algorithm requires miners to find prime numbers using the Cunningham chains method. Verification of this algo is fast and memory-cheap while the actual calculations appearing to be memory hard because to get effective work required a prepared data structure (about 100kb). As it was pointed in [4], this algorithm has two weaknesses:

*"'Time-memory tradeoff"' - an ASIC has the option of removing much of the memory required by sacrificing some computational efficiency; even with only 100 KB per thread a miner can be fairly efficient.

'*"All clear effect"' - as it turns out, it is possible for GPUs and ASICs to have multiple threads share the same memory. Since Primecoin mining only requires the sieve to be filled once, an ASIC can calculate a sieve first and then run thousands of threads through it.

The **Birthday Attack** algorithm looks for hash collisions. The ingenuity of the protocol comes from the fact that while efficiently computing the birthday attack requires storing the hashes in a data structure so that every new hash can be checked against all previous ones, verifying the solution only requires checking two hashes. The algorithm is resistant against the all-clear effect because memory must be flushed every 2^{32} rounds. However, there are several shortcuts and time-memory tradeoff attacks. First, one can optimize by storing only the first few bytes of each hash instead of the entire hash. Second, one can only store hashes with the first two bits being 00; this takes 75% of possible solutions out of consideration, reducing time efficiency by 4x, but it also increases space efficiency by 4x. Finally, there is potential for optimization by examining the various options between storing hashes in a large array, a binary or red-black tree, and other more complex structures; the optimal algorithm may be quite complicated.[4]

The Ethereum project was planning to use **Dagger**, a PoW algorithm that claimed to be memory-hard to compute and memory-easy to verify. Dagger creates a directed acyclic graph with a total of $2^{23} - 1$ nodes in sequence. If the miner finds a node between index 2^{22} and 2^{23} such that this resulting hash is below 2^{256} divided by the difficulty parameter, the result is a valid proof of work. The algorithm was analysed and found flawed: “Dagger seems to provide almost the best possible scenario for parallelization. In Dagger, a certain amount of RAM is filled by pseudo-random data derived from the header and the nonce. This data is produced in rounds. Each round, a number of elements from the previous round outputs are hashed together. An

optimized implementation for an ASIC (or FPGA) is evident for anyone with some discrete logic design background." [6]. Finally etherium developers rejected to use Dagger as PoW algorithm.

Cuckoo Cycle is a graph-theoretic PoW system, based on finding cycles in large random graphs [7]. Originally, it was expected the function would memory consume about 4GBof memory, but later David Andersen [8] made an optimization that reduced memory usage to 512MB-300MB.

The potential problem of functions such as Primecoin, Dagger or Cuckoo Cycle is that it assumes that for solving problem needed specific amount of memory, but there could be discovered some optimizations or new approaches that may do these Proof-of-Work functions with less memory.

Going back to classic hash functions, **CryptoNight** hash looks pretty strong. It uses a similar concept as SCrypt but has a huge 2MB scratchpad (taking advantage of the CPU's fast cache memory) that is modified on each step. It uses the AES instruction set that is built in to most modern CPUs. On the other hand, this hash function is extremely slow: hash operation takes about 20ms for modern CPU and about 150ms on laptops. The speed makes some DoS attacks possible and causes slow block chain synchronization, especially with laptops. The slow work time comes from big scratchpad. On one hand it takes advantage of CPU's cache memory, but on the other hand the algorithm has to prepare and then process whole scratchpad data, therefore it actually takes longer time. These constraints were supposed to protect hash from GPU and ASIC implementation, but a GPU miner appeared on the scene in 2 weeks after this technology got public attention.

Idea

After review of existing Proof-of-Work functions, we determined we wanted to have a function with the following properties:

1. **Memory hard** - undoubtedly this property is a key feature of being ASIC resistant.
2. **Fast validation** - it is very desirable to have a fast synchronization time to protect nodes from possible DOS attacks.

Classic SCrypt-like hash functions create a scratchpad for every hash calculation. The shortcoming to this is larger scratchpads take more time to prepare. 2MB is reasonable size based on limitations forced by speed requirements. However, memory hardness is desired only at mining; single hash calculation is not needed to be memory hard.

Boolberry took a different approach. The scratchpad is generated to provide an array of pseudo-random data to work on, and the generation causes a lot of memory access operations. But, the cryptocurrency already has pseudo-random data in the block chain; some hashes, keys and so on. **The idea is to use this block chain data as one solid incremental scratchpad for hashing. A portion of each new block is added to this scratchpad.**

Now it is possible to have a wide range of scratchpad sizes, since there are no performance restriction due to scratchpad generation, as in the SCrypt-like functions. But, after public discussion [9] it became clear that too large of a scratchpad would make it impossible to

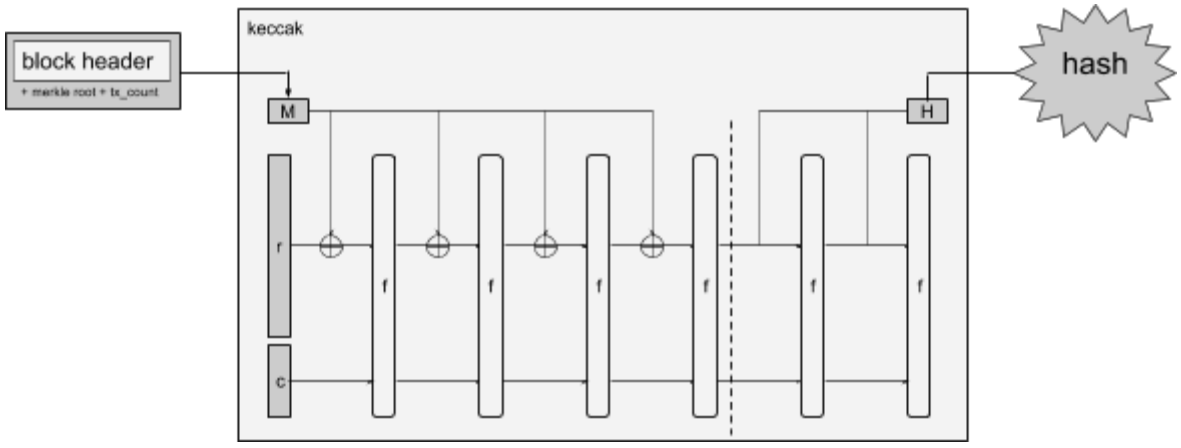
have SPV-client in the future, especially for mobile platforms. On the other hand, if the scratchpad is too small, it leaves room for a bigger advantage for GPU mining and possibility of ASICs. Considering both of these circumstances we decided to restrict scratchpad growth to about 90MB per year. For incremental scratchpad update we use following data:

* previous block id
* coin-base transaction's one-time public key
* coin-base transaction's merkle hash
* coin-base transaction's output keys

Implementation

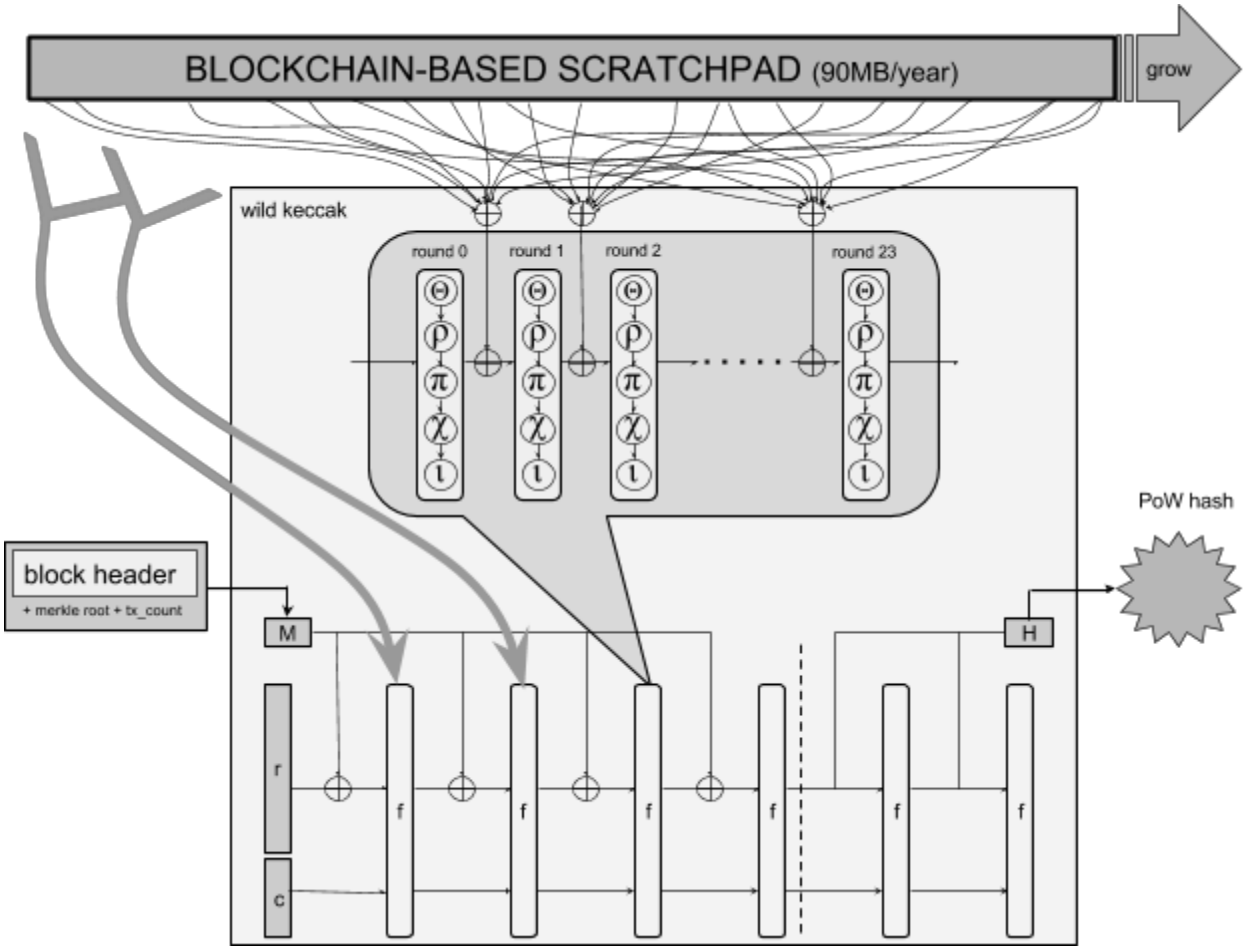
Now that there is a global scratchpad, a hash function is needed that's fast and cryptographically strong. The hash function will use this scratchpad with many reads from randomly-chosen addresses.

Keccak (SHA-3) was chosen as the base hash function, and made memory hard by injecting random memory reads in between the internal permutation rounds. To illustrate the difference, this is the original Keccak function[10]:



Original Keccak Schema

Each function f does 24 rounds of permutations, so we decided to inject read operations right between each round inside f:



Wild Keccak Schema

Wild Keccak has modified permutations function - we modified operations in “Theta” phase to use 64-bit multiplication instead of XOR, in order to give slight advantage to consumer level CPU over GPU and ASIC miners:

```
for (i = 0; i < 5; i++)
{
   bc[i] = st[i] ^ st[i + 5] ^ st[i + 10] * st[i + 15] * st[i + 20];
}
```

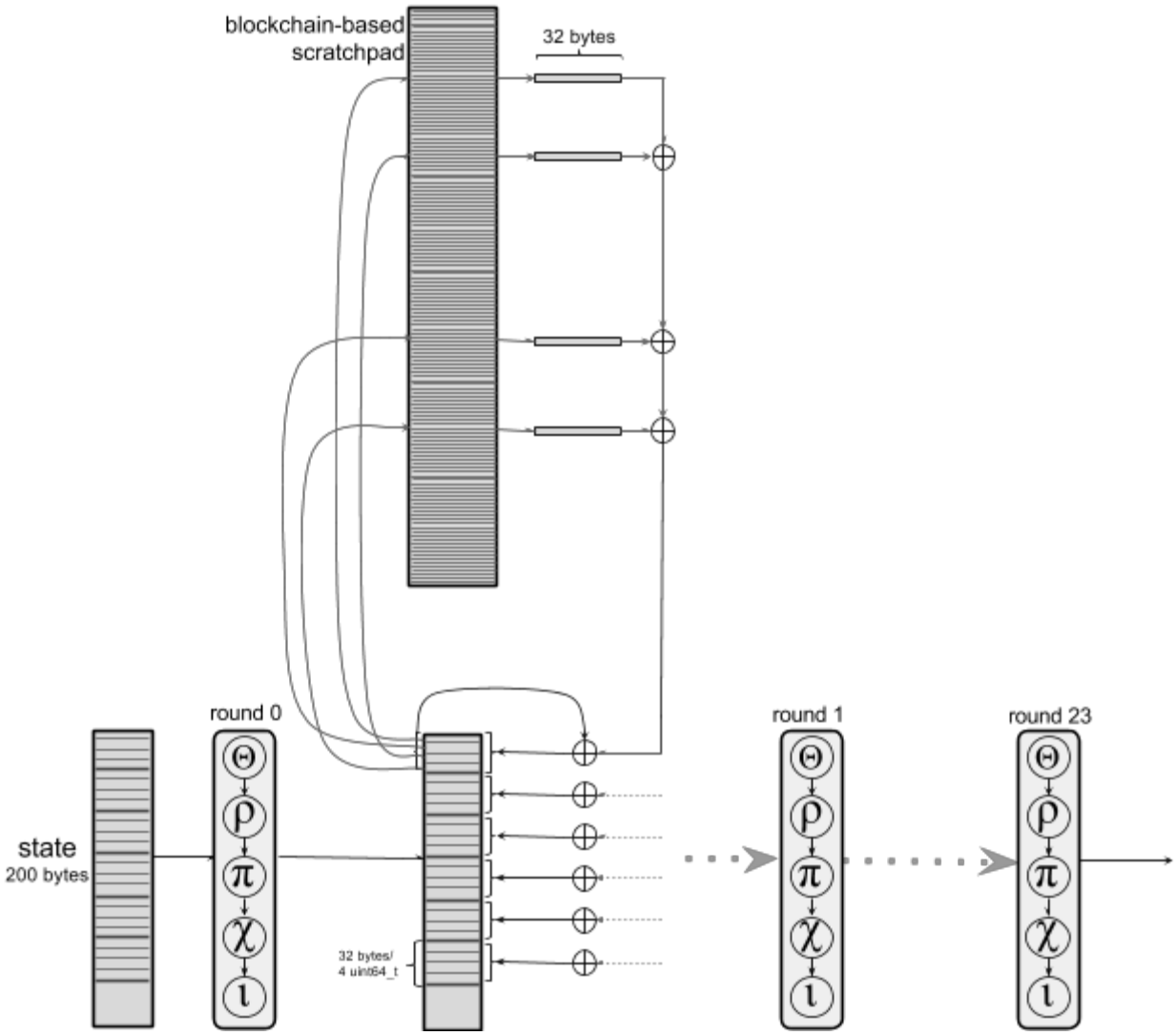
It is debatable if this modification will keep all cryptographic properties of hash function, but it should work fine as PoW function.

After each Keccak permutation round, its internal state array is filled with pseudo-random data as a result of computing the hash function, modified by xoring with arbitrarily selected data from global scratchpad.

Let:

S[] - global scratchpad represented as array of 32-byte blocks
B.a, B.b, B.c, B.d - 8 byte long parts of 32-byte long block
KS - keccak state, represented as array of 32-byte long blocks,
then modification function will be:

```
for(i = 0; i != 6; i++) /*for whole keccak state*/
   KS[i] = KS[i] ⊕ S[KS[i].a % S.size()] ⊕ S[KS[i].b % S.size()] ⊕
S[KS[i].c % S.size()] ⊕ S[KS[i].d % S.size()]
```



Wild Keccak State Modification

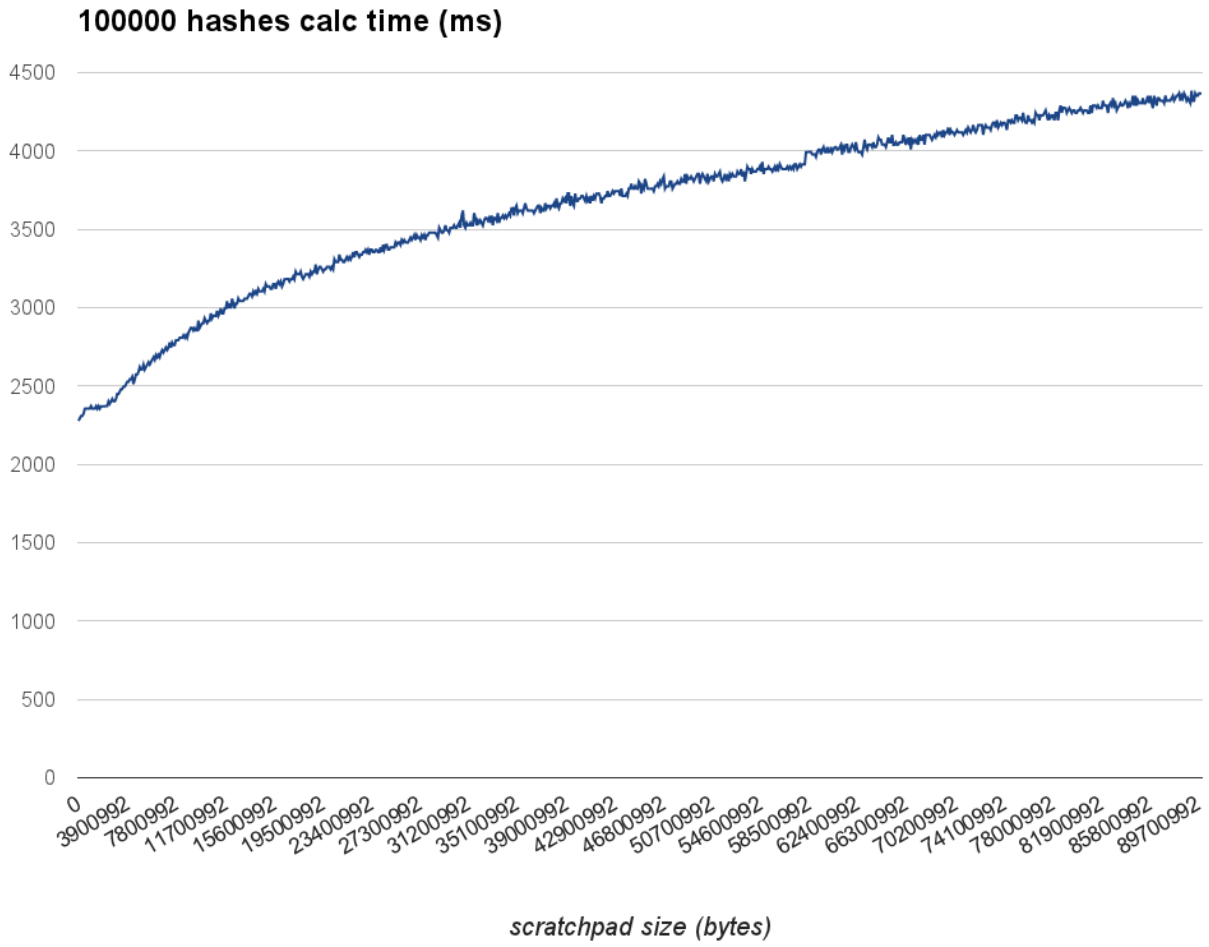
With that approach, for usual block header hashing operation Wild Keccak takes about 1100 operations of reading 32-byte blocks from global scratchpad.

Note: Current version of Wild Keccak has missing round constant change as an implementation error. The consequences of that error merits further analysis, but the error has been deemed non-critical at this juncture due to multiple state array modifications between rounds.

Results & Conclusion

To ensure that current implementation of the idea that block chain based Proof-of-Work hash function is really memory hard, we conducted performance tests using different scratchpad sizes, and for each size, 100000 hash calculations were measured. As there are significant differences between CPU cache memory reads and DRAM memory reads, memory hard function should slow down with increase of scratchpad size due to CPU cache memory satiation. Here are results measured on Windows 7 x64 with Intel Core i5 for scratchpad sizes from 0 till 90MB, which is about a year of currency life:

.



As seen, while scratchpad size is growing, hash calculation time is also growing significantly which confirms that resulting hash function is actually memory hard and at the same time single hash calculations are almost as fast as original keccak.

This diagram could be easily built by running performance_tests module and passing it's results into spreadsheet.

References:

[1] https://litecoin.info/scrypt
[2] https://litecointalk.org/index.php?topic=18166.0
[3] http://wiki.darkcoin.eu/wiki/X11
[4] https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-Dagger
[5] http://www.hashcash.org/papers/momentum.pdf
[6] http://bitslog.wordpress.com/2014/01/17/ethereum-dagger-pow-is-flawed/
[7] https://github.com/tromp/cuckoo/blob/master/cuckoo.pdf
[8] http://da-data.blogspot.de/search?q=cuckoo
[9] https://bitcointalk.org/index.php?topic=588421.0
[10] http://keccak.noekeon.org/Keccak-main-2.1.pdf