Specification of the FishHash algorithm

Lolliedieb

October 27, 2023

Abstract

In this document we will specify and explain the details about the new FishHash proof of work scheme created as a new proof of work for the IronFish blockchain.

1 Introduction

In a voting that ended on October 3rd 2023 the mining community of the IronFish project opted to change the proof of work scheme (PoW) used in the project to a proposal made by Lolliedieb on September 18th. The until then nameless proof of work scheme got its name "FishHash" through the discussions within the community in the days between the presentation and the final voting.

The goal behind the whole process that preceded the voting was the idea of democratizing the mining of IronFish. For most blockchain projects, a large number of miners in the initial mining community is beneficial because it increases the total number of people who interact with the blockchain and thus accelerates the initial spread.

However, during the first days of the IronFish project, it was observed that there was a high percentage of blocks of unknown origin. Due to the lack of reaction to price changes of this hash rate of unknown origin, it could at least be assumed that this hash rate had a not inconsiderable economic advantage over the widespread GPU miners and was thus contrary to the idea of a mining community that was as broadly based as possible.

The idea of fish hash - which is the result of the discussion process about re-diversification of the mining community - is to be memory bandwidth bound rather then the compute-bound blake3 algorithm. For memory chips there is only a limited diversity in their performance and characteristics, since all memory cells are optimized to record data as reliably and quickly as possible and to make it available to the user again. For this reason, the memories used in different device categories such as graphics cards (GPU), field programmable gate arrays (FPGA) and application-specific integrated circuit (ASIC) are also very similar and have only minor differences in their performance when identical chips are used in different devices.

Therefore, the idea is to create a PoW scheme that is mainly based on the existence and performance of these memory chips, so that the performance and efficiency difference between specialized and widely available hardware is as small as possible. This document describes the basic structure of this new scheme and which steps are necessary to compute a correct hash value with the new scheme.

2 Algorithm Outline

The basic concept of most memory-hard algorithms is based on three processing phases that will be executed in order. These are depicted in algorithm 1.

This scheme is quite commonly used by multiple PoW algorithms as Ethash, Etchash, KawPow, Octopus or with slight modification Autolykos just to name a few. To create a secure pow scheme one hereby always relies on established hash functions for computing the initial seed and the final result.

The further properties like the memory size requirement and so the minimum device specs required to run the pow scheme. For pow schemes that are memory limited, which is when the central loop of

Algorithm 1 Outline of memory hard PoW schemes

$StartValue \leftarrow HashFunction(BlockHeader, Nonce)$

$MixValue \leftarrow Expand(StartValue)$
for $0 \leq n < NumAccess$ **do**
  $LoadValue \leftarrow ReadDataSet(MixValue, StartValue, n)$
  $MixValue \leftarrow MixFunction(MixValue, LoadValue)$
end for
$CollapsedValue \leftarrow Collapse(MixValue)$
$FinalValue \leftarrow HashFunction(StartValue, CollapsedValue)$

operation dominates the overall run time of the algorithm, the performance that can be achieved on the algorithm is determined by the *ReadDataset* portion of the loop, so the total access with and the number of loop iterations.

Beside this for many miners properties like the power draw are important. This property mostly depends on the locality of the data set, e.g. if it can be fully cached or if the device memory requires to be active during the mining process, and the ratio between the core dependent operations like the $HashFunction$ and the $MixFunction$ and the memory access call.

In order to obtain an algorithm limited by memory bandwidth, it is important that the amount of memory segments requested in the central loop is large in relation to the hash function used as well as the overhead of the mix function.

3 FishHash Algorithm Details

3.1 Hash Functions and Expanding

The hash function used for the FishHash pow scheme is defined to be the Blake3 algorithm. We chose this algorithm because it is optimized to run on native 32 bit hardware as many GPUs are. Also its rounds are rather lightweight compared to other hash functions. Due to the IronFish block header having a size of 180 bytes the first iteration will require three rounds of Blake3, while the final hash call processing the 64 bytes of $StartValue$ plus 32 bytes of $MixValue$ needs to run two such iterations.

Due to the structure of the IronFish block header having a section part at the end of the header that can be modified by the miner, for the purpose of mining it is possible at the moment to reduce the number of iterations in the initial hash call to just one. This can change though in case the IronFish block header structure is modified in future developments of the blockchain.

The output of a Blake3 hash function call has up to 64 bytes. The MixValue used for our proof of work scheme will be 128 bytes. In order to pad the output of the initial call to this length the 64 output bytes will be duplicated. As for the final Blake3 call we will ask for a 32 Byte output of the hash function. This equals the output length the Blake3 function has when mining IronFish before the fork.

Therefore the inputs and outputs of FishHash as a whole equal the usage of Blake3 function for computing the block header hash value before the fork to FishHash and the new scheme can be used as a drop in replacement.

3.2 Fetching and Mixing values

The FishHash algorithm will always fetch three values of 128 bytes width and also aligned by 128 bytes from the data set to be processed into a new mixing value. This access size is wide enough to utilize the memory bandwidth a typical customer GPU offers by always filling full cache lines.

In order to fetch three different values of pseudo random position in every pass of the central loop, we will take different disjoint sections of the MixValue and perform a modular reduction modulo the number of items in the data set. In order to reduce the overhead we decided to use constant positions

in the data set, which makes getting the reduced value easier then e.g. for ethash. The three position will be offset by 32 bytes in between. This ensures that in case of multiple threads working on the same MixHash the calculation the address calculation can be spread and performed by different threads in parallel.

Let for $Var[a : b]$ denote the bytes with index $a \leq b$ of Var be read as an integer value of length $b - a + 1$ in little endian encoding. Also let $DataSet[n]$ denote the nth 128 byte wide entry of the pre-calculated data set. Then the function $ReadDataSet$ for FishHash is defined like follows.

Algorithm 2 ReadDataSet

$LoadValue[0 : 127] \leftarrow DataSet\,[MixValue[0 : 3]\ \%\ dataSetSize]$
$LoadValue[128 : 255] \leftarrow DataSet\,[MixValue[32 : 35]\ \%\ dataSetSize]$
$LoadValue[256 : 383] \leftarrow DataSet\,[MixValue[64 : 67]\ \%\ dataSetSize]$

For mixing the fetched data with the $MixValue$ we decided to first modify two of the three fetched values in different manners with the existing $MixValue$. This ensures that the old value in its full length is relevant for the next generated iteration of $MixValue$. Also this ensures that it is relevant if a fetched value is in first, second or third position. Afterwards we interpret the 128 bytes values as 16 integers of 64 bit or 8 bytes each and perform a single multiply and add operation on each of them to generate a new $MixValue$.

Algorithm 3 Mixing Function

$LoadValue[128 : 255] \leftarrow FNV\,(MixValue, LoadValue[128 : 255])$
$LoadValue[256 : 383] \leftarrow LoadValue[256 : 383] \oplus MixValue$
for $0 \leq i < 16$ **do**
  $MixValue[8i, 8i + 7] \leftarrow LoadValue[8i, 8i + 7] \cdot_{64} LoadValue[128 + 8i, 128 + 8i + 7]$
  $MixValue[8i, 8i + 7] \leftarrow MixValue[8i, 8i + 7] +_{64} LoadValue[256 + 8i, 256 + 8i + 7]$
end for

We define that in the description of algorithm 4 the symbol $\oplus$ denotes a logical bitwise xor, the function FVN is identical to the function of the same name known from the Ethash PoW, namely $FNV(a, b)$ is defined to be $(0x01000193 \cdot a) \oplus b$. Because this function is working on 32 bit values we define this function to operate on all 32 dword sized chunks of $LoadValue[128 : 255]$ and $MixValue$. Note that the two operations of the loop usually can be performed as a single multiply and add operation on most architectures, but for format reasons we decided to note them down as two separated operations.

3.3 Collapsing the Final Value

The 128 bytes output of the $MixValue$ variable will be collapsed down to a value of length 32 bytes to reduce the computation demand of the final call to the Blake3 hash function. In order to make all bytes of $MixValue$ important this can not happen by a simple truncation, but instead we use the same reduction function based on the above mentioned FNV function that is common to most Ethash siblings.

Algorithm 4 Reduction of MixValue

for $0 \leq i < 8$ **do**
  $CollapsedValue[4i, 4i + 3] \leftarrow FNV(MixValue[16i, 16i + 3], MixValue[16i + 4, 16i + 7])$
  $CollapsedValue[4i, 4i + 3] \leftarrow FNV(CollapsedValue[4i, 4i + 3], MixValue[16i + 8, 16i + 11])$
  $CollapsedValue[4i, 4i + 3] \leftarrow FNV(CollapsedValue[4i, 4i + 3], MixValue[16i + 12, 16i + 15])$
end for

4 Data Set Generation and Parameters

Different to siblings of Ethash the data set size in FishHash is a constant value and equals 4608 MBytes or exactly 37748717 data set elements, which equals the figures for Ethash epoch 448.

The algorithm for creating the dag is identical to the one for creating an Ethash DAG for epoch 448, but with two changes central changes. First of all the seed to generate the light cache is the 32 byte value received by

$$\begin{aligned} Blake3('FishHash') &= Blake3(0x4669736848617368) \\ &= 0xeb0163aef2ab1c5a66310c1c14d60f4255a9b39b0edf26539844f117ad672119. \end{aligned}$$

Secondly the iterations used for calculating a dag item is doubled from the original value of 256 to 512. Otherwise the generation of the data set remains identical to the Ethash algorithm, because the general approach has proven itself to be free of any backdoor reducing the calculation effort over the past years.

5 Static Analysis

5.1 Memory Demand

As mentioned in the previous section the memory size required to mine equals 4608 MBytes. On top of this most implementations will allocate additional 72 MBytes to fit the light cache. These 72 MBytes are also required for a light verify that is there for mostly verifying, but not mining computed hashes. Note that implementation dependent these 72 MBytes are optional once the data set has been build and the space can be used for other operations afterwards. Also it is possible to keep the light cache off device and just build the dag at an lower speed in case there is shortage of memory on the device itself. With this storage requirements the algorithm can be performed on all GPUs with at least 5 GByte released during the past 6 years as well as the specialized GPUs with 5 GBytes. In case of GPUs equiped with 6 GBytes this should be feasible independent of the operating system and independently of a screen to be connected or not.

Once the minimum memory capacity requirement is met, the actual performance of the algorithm depends on the memory bandwidth of the memory. With the access width of 128 sequential and aligned bytes for each access, the memory accesses are identically wide as, for example, with the Ethash algorithm, where over 90% of the theoretically available memory bandwidth can also be used practically on all modern architectures.

In each iteration of the main loop there are three such accesses. Furthermore, there are a total of 32 iterations of the main loop, so that the memory bandwidth requirement for each calculated hash value is 12288 bytes. This is 50% higher than with Ethash and it can be assumed that, unless there are other reasons to the contrary, the maximum performance of the mining process is about 2/3 of the performance of mining Ethash on the same hardware given the same settings.

5.2 Computational Demand

The computational demand of calculating a single hash value is highly dependent on the actual architecture of the device the PoW scheme is supposed to be run on. For the sake of simplicity we will assume that we have an underlying architecure that is able to process one 32 bit integer or logical operation at a time. Note that the actual clock cycles used on modern architectures will be less, because most architectures offer specialized instructions that can perform multiple operations in one instruction.

For the central loop of the algorithm calculating the three addresses takes less then 10 clock cycles each. This includes the operations required to perform a Barret reduction on the 32 bit value that needs to be reduced mod the data set size and to shift it to actually generate a valid memory address from it. The application of the FNV function and the $\oplus$ operation on the fetched values will take 64 and 32 operations. With regard to the two 64 bit operations we can assume, that a 64 bit multiply can be emulated with five 32 bit operations on a 32 bit architecture and the 64 bit addition can be emulated with two 32 bit additions with carry over. Therefore the total amount of operations can be upper bound by $16 \cdot (5 + 2) = 112$ instructions.

Overall the central loop can be estimated to take less then $32 \cdot (30 + 64 + 32 + 112) = 7616$ 32-bit operations. Additionally there are the currently required three blake3 loop iterations per calculated

hash. Each of them taking $12 \cdot 8$ calls to the Blake3 g-function, which consists of 14 operations with 32 bit values. Therefore the calculation of the needed Blake3 hashes will add about $3 \cdot 7 \cdot 8 \cdot 14 = 2352$ additional operations.

Note that with this figures the computational demand is relatively small compared with the memory demand, because already the latency for fetching uncached data from the data set will span multiple hundreds of clock cycles. Also the numbers presented per hash here are significantly less then e.g. for the Ethash algorithm, because alone the 24 iterations of the twice performed keccak algorithm need to be estimated with 250 cycles for each iteration or over 12000 operations in total using the same assumptions as for our algorithm.

Therefore comparing with other algorithms using a similar structure the computational demand of the algorithm is rather small and we can assume that on modern architectures the assumption to be memory bandwidth bound holds. This also applies for other device classes as FPGAs or ASICs, that should be capable of performing the algorithm once the minimum criteria of enough memory size is met, but the advantage they can gain over common hardware as GPUs is limited to improve the computational side of the algorithm, which is not performance critical. This limitation of potential performance and efficiency gains is what makes FishHash a GPU friendly algorithm in the end.