Binarium cryptocurrency whitepaper

http://binarium.money

Author: Rodion Karimov (RodionKarimov@yandex.ru , http://imagination-works.ru , https://vk.com/RodionKarimov), assistents: Anton Novozhilov (haron@dcyber.ru) and Vladislav Nitcak (wladislav_n@bk.ru).

Abstract

Binarium cryptocurrency is the first one, which is protected from **ASIC**s. It does so by configurable hashing algorithm (with use of different hashing and encryption functions), which configures itself once per week or blocks generation difficulty change. This makes it costly to implement and own ASICs for each sub-function and reconfigure them in accordance with current algorithm state. And it uses data amplification with **Salsa20**[9] fast stream cipher. This also makes hashing function dependent on **RAM** sizes and random accesses speed. Which, in turn, makes it even more costly to build ASICs for and reduces **GPU** efficiency in mining, because their **VRAM** is better suited for piplined transfers of large data volumes, instead of speed of random accesses, and because their cache is shared between cores and part of it is read-only. Also, each thread, calculating hashes for Binarium blocks should have its own copy of memory area for data amplification, because this process and its intermediate data depends on information of block in consideration. With the main anti-ASIC factor is the ability to change hashing functions after Binarium widespread adoption, while keeping its current **consensus**.

Binarium cryptocurrency is the first one, which supports hashes functions changes and other major network-wide changes without breaking a consensus. It makes so by implementing updates in 4 steps: introduction of unactivated updates into clients; "*soft*" clients enforcing each other to update the software; "*hard*" clients enforcing each other for updates; and actual activation of changes in clients, when network reports satisfying level of changes adoption in it. In this model clients send upgraded peers statistics information to main Binarium **RPC** server, this allows to monitor network state and activate changes, when enough amount of users upgraded to new software versions. This functionality will be implemented after launch of cryptocurrency, when we'll gather enough feedback on current algorithm and overall Binarium block-chain functioning.

It is based on **Dash**[2] cryptocurrency and supports **master nodes**, **InstantSend** and **PrivateSend** functions from it.

We plan to make its integration with online games in form of:

- Ingames currency, which player can buy and sell for real money.
- Created by game developers in-game and near-game quests, which players can complete and receive reward in Binariums.
- Player to player trade agreements, from which game developers and Binarium developers receive commission.
- Games, **DLC**s, ingame items, ingame money and other valuables exchange.
- Ability to organize tournaments by game developers and gamers themselves with prizes in Binarium, games, DLCs, ingame items and other valuables.

Later on we plan to add smart-contracts, electronic documents management and other functions, which we'll keep for now in secret. We'll provide further technical details on functions implementation in the following sections, also you can see it yourself in Binarium source code[14].

1. Algorithms diversity and reconfiguration

1.1. Hashing algorithms

Binarium hashing algorithm is based on **X11**[2], it uses its hashing sub-functions: **blake**, **bmw**, **groestl**, **skein, jh**, **keccak**, **luffa**, **cubehash**, **shavite**, **simd**, and **echo**. Also following functions were added: Russian **GOST 2012 Streebog**[3] and **Whirlpool**[4]. We considered adding **SwiFFT**[5] hashing function, which uses **Fast Fourier Transforms**[15] to generate hashes from input data and, probably, can be used in **Post-Quantum Cryptography**[13] era. But this function currently is rather raw and not widely used, so it is best to put it now into further development and consider its implementation in next network-wide updates.

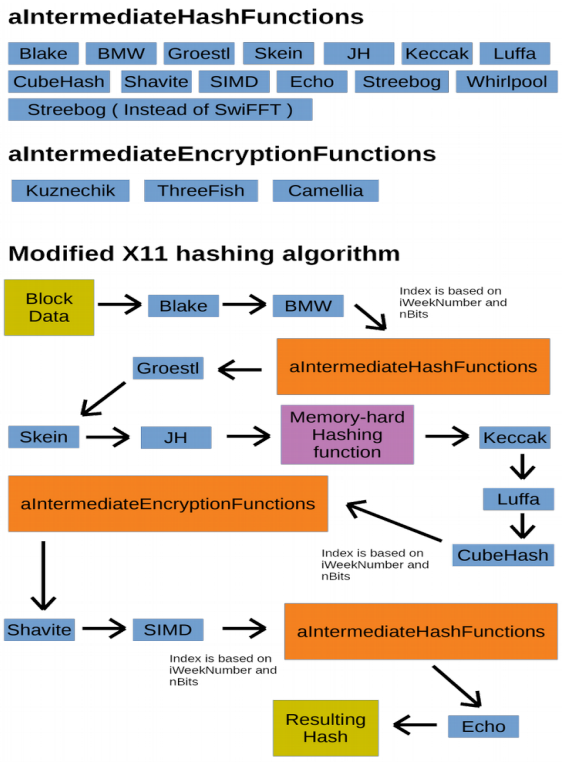

Figure 1: Binarium configurable hashing algorithm.

1.2. Encryption algorithms

Binarium also incorporates reliable cryptographic encryption functions in the hash obtaining process: **GOST 2015** block encryption function **Kuznechik**[6], **ThreeFish**[7] block encryption function and **Camellia**[8] block cipher.

1.3. Reconfiguration takes place each week or by blocks difficulty change

There are 3 configurable steps, introduced between X11 hashing sub-functions applications, they are defined by week number from genesis block and **nBits** field of current block. The first one after step two, between BMW and Groestl functions:

```
iWeekNumber = _iTimeFromGenesisBlock /
I_ALGORITHM_RECONFIGURATION_TIME_PERIOD_IN_SECONDS *
I_ALGORITHM_RECONFIGURATION_TIME_PERIOD_IN_SECONDS;
iIndex = ( iWeekNumber + nBits ) % I_AMOUNT_OF_INTERMEDIATE_HASH_FUNCTIONS;
aIntermediateHashFunctions [ iIndex ] ( uint512AdditionalHash.begin (), 64,
nullptr, static_cast<void*>(&hash[1]) );
```

It selects hashing function from array of available and applies it to intermediate hash. The second one after step 8 of X11 hashing, between applications of CubeHash and Shavite sub-functions:

```
iIndex = ( iWeekNumber + nBits ) %
I_AMOUNT_OF_INTERMEDIATE_ENCRYPTION_FUNCTIONS;
aIntermediateEncryptionFunctions [ iIndex ] ( static_cast<const
void*>(&hash[6]), 64, static_cast<const void*>(&hash[0]),
static_cast<void*>(&hash[7]) );
```

It selects encryption algorithm from array of available and applies it to intermediate hash. And 3-rd configurable step is introduced between 10th and 11th steps, between SIMD and Echo sub-hashes applications:

```
iIndex = ( iWeekNumber + nBits + 10 ) % I_AMOUNT_OF_INTERMEDIATE_HASH_FUNCTIONS;
aIntermediateHashFunctions [ iIndex ] ( uint512AdditionalHash.begin (), 64,
nullptr, static_cast<void*>(&hash[9]) );
```

It selects hashing function from array with offset from first reconfiguration index.

2. Memory hard pseudo-random writes and reads hashing function

2.1. Sequential Salsa20

First 32 KB memory block is zeroed for data amplification, then data is written to it in pseudo-random locations in form of 512 bits memory blocks. Salsa20 is applied sequentially: for each new encryption previous encryption result is applied to source data and there is no easy way to know where new data block will be written in memory. This way memory can be processed only sequentially and there is no way to parallelize this process on GPU and ASIC.

```
memset ( aMemoryArea, 0,
I_AMOUNT_OF_BYTES_FOR_MEMORY_HARD_FUNCTION );
```

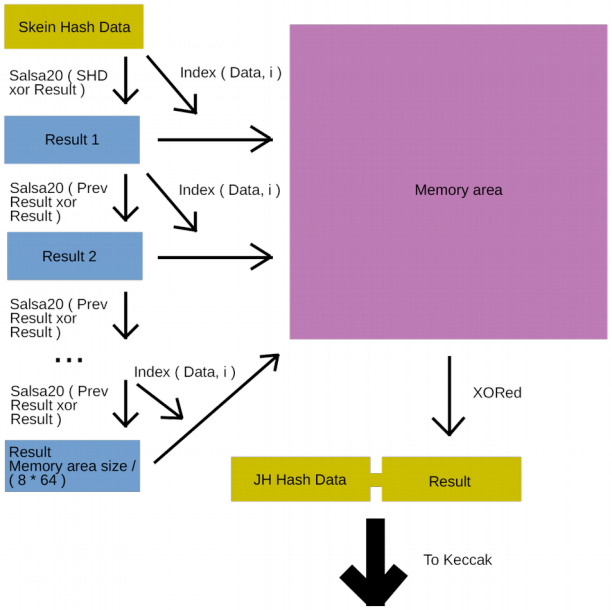

Figure 2: Binarium memory hard hashing function.

Then sequential encryption of intermediate Binarium block hash data (result of Skein sub-function) is performed:

```
for ( i = 0; i < I_AMOUNT_OF_BYTES_FOR_MEMORY_HARD_FUNCTION / ( 64 ) / 2; i ++ )
{
  iWriteIndex = (
    //  % I_AMOUNT_OF_BYTES_FOR_MEMORY_HARD_FUNCTION here is to prevent
    // integer overflow on subsequent addition operation.
    GetUint64IndexFrom512BitsKey ( uint1024CombinedHashes.begin () + 64, 0 ) %
I_AMOUNT_OF_BYTES_FOR_MEMORY_HARD_FUNCTION +
    i * I_PRIME_NUMBER_FOR_MEMORY_HARD_HASHING ) %
      ( I_AMOUNT_OF_BYTES_FOR_MEMORY_HARD_FUNCTION - 8 * ECRYPT_BLOCKLENGTH );

  // From previous encryption result in memory to next encryption result in
memory.
  ECRYPT_encrypt_blocks (
    & structECRYPT_ctx,
    uint1024CombinedHashes.begin () + 64,
    & ( aMemoryArea [ iWriteIndex ] ),
    8 );

  uint1024CombinedHashes.XOROperator ( 64, & ( aMemoryArea [ iWriteIndex ] ) );

} //-for
```

`Function GetUint64IndexFrom512BitsKey ()` takes 512 bits memory block (`uint1024CombinedHashes.begin () + 64`) and obtains from it `uint64_t` for pseudo-random seed for index into 32 KB `aMemoryArea`, which is combined with `i * I_PRIME_NUMBER_FOR_MEMORY_HARD_HASHING` , which performs iteration over this memory area. Then encryption process goes on : `ECRYPT_encrypt_blocks ()` and its result is combined with `uint1024CombinedHashes.begin () + 64` , from which on next step new index and source data will be formed. This makes new encryption steps dependent on previous steps and makes it necessary to perform encryptions sequentially. Also, this way it is hard to predict which data will be written to which locations and introduces large possibility of data parts overwrites, so it is hard to predict what will be placed in given memory bits. After this whole memory block is XOR'ed into `uint1024CombinedHashes.begin () + 64` memory area and it is combined with step 5 hash (JH result) of modified X11 algorithm.

```
for ( i = 0; i < I_AMOUNT_OF_BYTES_FOR_MEMORY_HARD_FUNCTION / 64; i ++ ) {
  uint1024CombinedHashes.XOROperator ( 64, & ( aMemoryArea [ i * 64 ] ) );

} //-for
```

This step happens after step 5 of X11 hashing algorithm, before Keccak hashing sub-function is applied.

However, each block **nNounce** guess can be processed in separate thread, which opens possibility for parallelization. But each thread requires its own 32 KB memory area to store amplified data. So, this limits parallelization on GPU and increases ASICs building cost. And overall algorithm is random memory writes and reads bound, so this limits its performance on GPU and ASICs and opens possibilities for **egalitarian computing**[10]: users with equipment with different performance have equal or close abilities in network. This makes it much harder to concentrate large computing powers in single hands and perform **51% attack**[11].

2.2. Performance considerations.

Our hashing algorithm has 15 hashing subfunctions + 1 memory-hard hashing function, from these 15 only 3 are configurable and overall hashing speed is bound to speed of random memory writes and reads. So, we expect, that hashing speed will not vary too much, when algorithm reconfigures itself.

3. Introduction of major cryptocurrency updates, while keeping current consensus

3.1. 4-steps major network-wide changes implementation in cryptocurrency network

a) Implementation of changes in code in inactive state with placing of hashing functions, which indicate to other peers whether clients have updated software. They do so via **Zero-knowledge proof**[12]-like protocol: clients ask their peers to send them results of applications of hashing functions with given index to random data and then check its correctness with their results. Also in code conditions are introduced, which indicate when changes should be activated and from which block or time network should switch to new functionality. Triggers for these conditions are polled from consensus of RPC-servers.

We can create, say, 100 RPC-servers and network will switch to new functionality, when all servers answer and report the same conditions for functionality switch. This way, we can easily control whether network will switch or not: we can disable upgrade process, in case of emergency, just by switching off at least 1 server. And this way malicious agents will not be able to easily gain control over all RPC-servers and hold it during all time after setting upgrade conditions and till update itself. We can set in clients mandatory pause in 1 week – a couple of days from update activation announcement till its adoption by network, to ensure that update is initiated by us.

b) Peers check whether other clients have updated software and "*softly*" enforce each other to update: send to them **GUI** notifications about this fact.

c) Peers "*hardly*" enforce each other to update: they ban for defined time other clients, if they have not updated their software. Network will move to this stage, when adoption of changes will reach 95% or more. Clients can gather statistics about update status of their peers and send it to main RPC server. This way we can activate new functions implementation stages, when network is ready for them. We can consider statistics only from clients, which have at least some amount of Binarium in their wallets, so that malicious agents will not be able to gather large amount of dummy empty wallets and distort statistics.

d) Update itself: we define on RPC servers consensus conditions, from which network will start to use new functionality (block index, time from genesis block, and so on). Clients receive this information and switch to it, when time comes.

3.2. **BIP**s[16]

We can consider on whether **Bitcoin Improvements Proposals**[16] are applicable to network updates, but our system is better in this respect, because BIPs require authors to gather by hand community consensus on changes and keep it while network reconfigures itself. While our system will do this automatically.

References

1. Bitcoin whitepaper : https://bitcoin.org/bitcoin.pdf .
2. Dash : https://www.dash.org/ .
3. GOST 2012 Streebog whitepaper : http://specremont.su/pdf/gost_34_11_2012.pdf .
4. Whirlpool hashing function : https://en.wikipedia.org/wiki/Whirlpool_(cryptography) .
5. SwiFFT hashing function : https://www.eecs.harvard.edu/~alon/PAPERS/lattices/swifft.pdf .
6. GOST 2015 block encryption function Kuznechik : http://www.tc26.ru/standard/gost/GOST_R_3412-2015.pdf .
7. ThreeFish block encryption function : http://www.skein-hash.info/sites/default/files/skein1.3.pdf .
8. Camellia block encryption function : https://info.isl.ntt.co.jp/crypt/eng/camellia/ .
9. Salsa20 stream cipher : https://cr.yp.to/snuffle.html .
10. Egalitarian computing : https://arxiv.org/pdf/1606.03588.pdf .
11. 51% attack : https://www.investopedia.com/terms/1/51-attack.asp .
12. Zero-knowledge proof : https://en.wikipedia.org/wiki/Zero-knowledge_proof .
13. Post-quantum cryptography : https://en.wikipedia.org/wiki/Post-quantum_cryptography .

14. Binarium sources : https://github.com/binariumpay/binarium .
15. Fast Fourier Transform : https://en.wikipedia.org/wiki/Fast_Fourier_transform .
16. Bitcoin Improvements Proposals : https://github.com/bitcoin/bips/blob/master/bip-0002.mediawiki .