The Solo Project

Whitepaper (v1.0)

Author: TyphooN (typhoon@minesolo.com)
www.minesolo.com

Introduction

The Solo Project's goal is to eliminate the economic incentives that lead to network centralization. Solo's Proof of Work algorithm, CN-IO, requires a fully synced copy of the blockchain on each mining node. As part of the planned future roadmap, contributors to the Solo Project plan to research and develop additional randomization that will adapt to changing mining solutions with the goal of maintaining solo CPU mining as the preferred method of distribution. Presently, the CN-IO Proof of Work Algorithm provides highly scalable performance on CPUs and ARM devices with AES-NI crypto acceleration. Due to the low scratchpad memory allocation (128KB per thread), performance scales very well on low to mid range devices and the PoW algorithm draws less total power from the CPU socket as measured in software vs similar PoW algorithms.

With advancements in the cryptocurrency landscape, Solo aims to achieve true scalability by pursuing a scalable network all while maintaining a secured and decentralized state. In the recent addition of the SECOR [3] tech layer, Solo has enabled faster transactions and preserved network security and stability through uncle mining and 20 second target block time. This provided Solo the opportunity to push itself one step closer towards a well balanced future.

The Solo Project contributors are adamant about keeping current with all privacy enabling and scalability impacting features that Cryptonote [2] based blockchains benefit from. At the time of writing this whitepaper the following technologies are enabled in mainnet: dynamic block sizing, Ring Signature Confidential Transactions (RingCT)[4], Bulletproofs v2 [5], LWMA1 DAA (Difficulty Adjustment Algorithm with negative timestamp vulnerability mitigation) [8] and SECOR (Uncle Block Mining) [3] support.

The Solo community feels that a large supply with a much longer emission period than competing blockchains is crucial for sustained growth in a decentralized blockchain network and for the use of a cryptocurrency in everyday transactions. Rather than the typical scenario of a rapid emission scheme within the first few months or years of a blockchain network coming online, all XSL (Solo) will be minted over the course of 30+ years with no disruptive reward halving events.

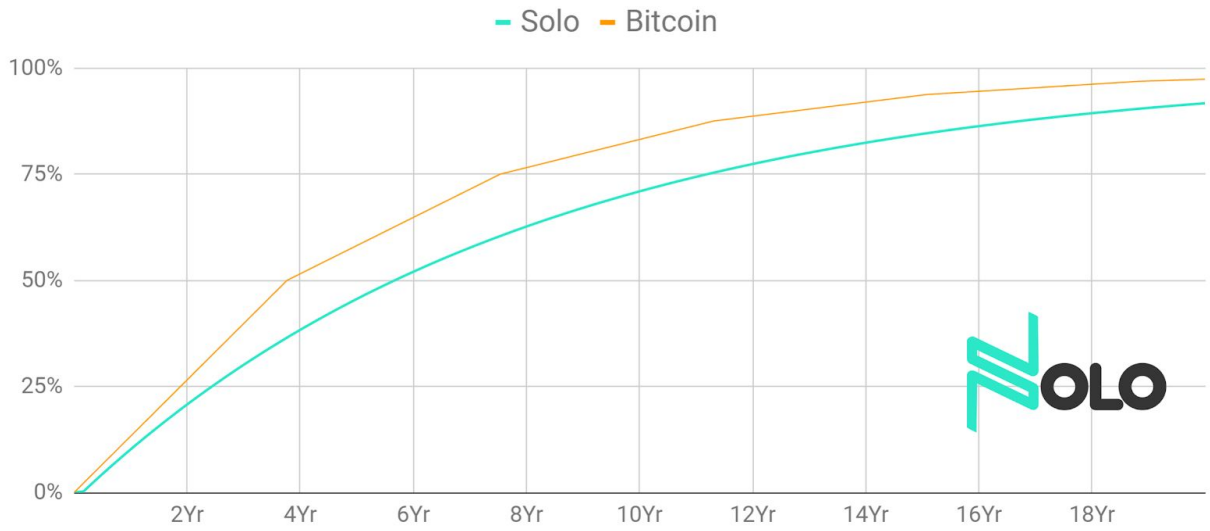
Tokenomics

Supply & Emission Curve

Solo's ~18.4B planned supply is possible due to the CRYPTONOTE_DISPLAY_DECIMAL_POINT value being 9 instead of the usual value of 12 in most other cryptonote coins. The Solo community feels that a large supply, scalability (fast block timing and SECOR support), low fees, and a long emission period is crucial for adoption in everyday transactions. Tail emission on the Solo network will kick in at 2095 AD and is setup to incentivize organic network growth over time as the block reward remains high for a long period of time instead of severely reduced after only a few years of mining found in most other privacy coins.

As Solo mining is the main method of distribution of XSL in the network, the reward must be high enough for mining nodes to continue to mine even as difficulty rises significantly. After the community voted prior to block 88,0001 the block reward was increased via a hard fork so that both high and low powered machines may participate in the network for the long run. By increasing the block reward, while still maintaining a very slow emission rate a satisfactory reward can be found by even low to mid range CPUs.

At the time of writing this document (August 20 2020) 18.59% of the supply has been minted in the 1 year, 3 months, and 18 days that the Solo network has been online. The block reward at block 2364154 was ~1193.56 XSL, the current Circulating Supply is ~2.46B XSL of the planned ~18.4B supply before tail emission kicks in. This is far slower than the emission schedule in both Bitcoin, Monero, and most other privacy blockchain projects in the space.




```
print_coinbase_tx_sum 0 2364154
Sum of coinbase transactions between block heights [0, 2364154) is
3428858746.884496694 consisting of 3428812341.709133404 in emissions,
and 46405.175363290 in fees
```

Fees and Transaction Speed

At the time of writing this document the current fee is 0.050823500 Solo per kB, which keeps fees for most transactions at a fraction of an XSL. With Bulletproofs v2 [5] support, low input transfers are 2-3KB which leaves the average TX fee at ~0.1-0.15 XSL. Even if the value of XSL is $1 USD or higher, the fee in fiat currency to send most transactions will remain low when compared to the competition.

Transferred funds have a 20 block unlock period. This translates to about 6 minutes 40 seconds for funds to unlock in the recipient wallet. After funds are unlocked, they can then be sent to another destination after that time. Compared to Monero's 2 minute block time coupled with a 10 block unlock period, this translates to roughly 20 minutes if blocks are found on schedule [7].

On the Bitcoin network, the average transaction time is 1 hour as long as the mempool is not congested. During times of heavy transactions on the Bitcoin blockchain, transfers with higher fees paid are prioritized while transfers with lesser fees remain in the mempool. If too low of a fee is paid on the Bitcoin network, a transfer can remain in the mempool for up to 3 days [9] before the transfer can be rebroadcast if the mempool remains congested due to Bitcoin's static 1MB block every 10 minutes [1]. This can be highly problematic for the end user. Thanks to dynamic block sizing, as the median block size increases on the Solo network, the block size will dynamically increase until all transfers are recorded on chain.

Adaptive Block Reward

The Solo Project has an adaptive reward based on historical data. The emission formula slightly decreases block reward each block as a function of all coinbase transaction amounts previously generated. Reward for blocks 1-88,000 based on a speed factor per minute of 22, resulting in a reward ranging from 549.755797504 to 548.315910839. After the community voted prior to block 88,0001 the block reward was increased via a hard fork so that both high and low powered machines may participate in the network for the long run. Reward for blocks 88,001 and greater based on a speed factor per minute of 25, resulting in a reward starting at 4,386.527155990 at block 88,001. Blocks 227001 and greater have a block time of 20 seconds, the reward was reduced to maintain a speed factor per minute of 25, resulting in a reward starting at 1414.513079344 at block 227001.

Uncle blocks receive a total reward of 155% of what would have been the mined block reward for the same block; Reward is split 50% to Uncle, 105% to Mined Block. See the latest block coinbase transaction amount for the current block reward. Solo's emission schedule is slower than bitcoin and does not include disruptive halving events.

Funding

All project related infrastructure costs, bounties, marketing costs and listing fees have been paid out of personal funds, with no ICO and no crowd funding.

Premine and Early Contributions

Solo is a continuation of a side project started by NERVA developer *angrywasp* and included a zero block (premine) reward of ~70,368, representing 0.00038% of the total supply. At block 66001 a hard fork was activated, after this time the founder no

longer participated in development. At the time of the fork ~36,318,207 coins had been minted representing ~0.196% of the total supply.

As of block 256000 (April 17, 2019) the current supply was ~689,220,912 representing 3.7% of the total supply. The current supply can be found using the command print_coinbase_tx_sum in the daemon command line.

SECOR (Simple Extended Consensus Resolution)

A key feature of Solo is the implementation of Masari's Simple Extended Consensus Resolution (SECOR, AKA Uncle Mining) [3]. SECOR further secures the network by reattaching otherwise orphaned blocks on top of the longest chain. By activating SECOR, faster block times can be achieved without the use of master nodes, DAG or some other centralized solution. Faster block times result in more transaction throughput making the blockchain more scalable.

Alternate chains are temporary disagreements about the longest chain, it happens in every cryptocurrency and with 20 second block times it happens frequently. This happens when there is a slight delay in sending signals to the entire network, the shorter the block time the larger effect the lag has.

Without alternate chains there can not be an uncle block mined. If two miners find blocks within the same time window instead of dropping one, the solution is added to the chain as an uncle block. The miner producing the uncle block is rewarded 50% of the calculated block reward they would have received for that block. Alternate chains with lower difficulty and/or outside of the window will stay in memory until solod is restarted but the only alternate chains that meet certain requirements are added to the main chain via an uncle mined.

More information about SECOR can be found in the Masari SECOR whitepaper [3].

The CN-IO Proof of Work Algorithm

In this section of the Solo whitepaper, changes of Cryptonight-IO vs NERVA's CN-Adaptive v1 [6] will be explained. A technical understanding of the Cryptonight hash

algorithm, NERVA's CN-Adaptive v1 hash algorithm, and C++ programming is assumed.

Basic Overview

The daemon uses block hashes from previous blocks to generate an unpredictable dataset which is then used to manipulate the data in the scratchpad. This data manipulation occurs after the scratchpad is first populated with data, before the AES mixing function is performed.

CN-IO Proof of Work Goals and Restrictions

CN-IO strengthens the solo CPU mining goals of Solo, by making it a requirement to have a local blockchain copy. Each of those possible combinations can be performed up to 64 different ways via the variable AES iteration count. CN-IO performs well across all CPUs with AES-NI instruction support due to the low L3 cache requirement per thread (128KB per thread). Compared to other similar algorithms this allows for more threads to be run on lower to mid range CPUs as a scratchpad size of 1-2MB+ can leave a lower end CPU unable to run as many threads as the available logical processors.

The blockchain must be synced one block at a time, so that the previous block hash can be read to generate the required information to randomize the scratchpad. The cost of this restraint is sync speed. For convenience, a verified blockchain snapshot is provided by the Solo Project core team for users that wish to reduce the initial sync time. A current link to a recently verified blockchain snapshot can always be found on our website at https://www.minesolo.com/ .

It is also important to distinguish that CN-IO is not exactly a modification to the CN Variant 1 as used by Monero. CN-IO creates a secondary algorithm which generates a random dataset that is applied to change the scratchpad buffer to change the resulting hash produced by Cryptonight. As a result, CN-IO could be applied to any cryptonight variant that may seek to promote Solo CPU mining as it's preferred method of distribution.

The most optimized version of CN-IO is always available to the public as soon as it has been tested. If optimizations may be made to the algorithm, constructive criticism

on the Solo Discord (https://discord.minesolo.com) and merge requests are welcome. The goal of CN-IO is to keep the barrier of entry low to all miners, including those with only a single low to mid range CPU. Compare this to Monero, which included a potentially crippled hashing algorithm to the public [10].

Detailed Overview

Solo's hash algorithm, Cryptonight IO (CN-IO) is based on Cryptonight-v1 with the following amendments:

- Scratchpad reduced from 2MB to 128KB
- Reduce AES mixing function iterations from 0x80000 to 0x40000
- Per block variation of the AES mixing function iteration count between 0x40000 - (0x40000 + 1024)

CN-IO makes the following changes to Cryptonight-v1:

- Reduce AES mixing function iteration variation from 1024 to 64

CN-IO maintains the basic functionality of v1 and provides a secondary algorithm to manipulate the data in the scratchpad, and resulting hash, further randomizing the algorithm each block. This data is calculated before the Cryptonight hash algorithm is run and applied to the scratchpad after it is initialized, directly before the variable iteration AES mixing function.

The scratchpad used in Solo's hashing algorithm is 128KB.
Of that, 32 bytes are manipulated via a non-predictable algorithm, which is calculated at the start of each block. A broad overview of of the steps looks like:

- Get hash of last block
- Bit shift arbitrary bytes in the hash to generate 4 uint8_t values
- Read the block hashes at these locations, reading back from the current height
- Convert the 4 block hashes into a single 128 byte buffer
- Generate arrays of scratchpad byte indices and randomization values from that buffer
- Apply that randomization data to the scratchpad buffer

Step 1: Get a list of old block hashes from the blockchain

Step 1 involves fetching the hashes of 4 previous blocks in the blockchain.
The process consists of the following steps:

- Retrieve a previous block hash. In this case we take block data from blocks height - CRYPTONOTE_MINED_MONEY_UNLOCK_WINDOW_V1 as an arbitrary value
- XOR 4 pairs or arbitrary bytes together to generate 4 new uint8_t values (designated b1 - b4)
- Retrieve the block hash at the height - bytes value. Example: height - b1, and be sure to use data from the alt_chain on spawned alt_chains.
- Do this for each 4 values

The last block is XOR'd together like so:

```text
text
01234567890123456789012345678901
|   |   |   |   |   |   |   |
b1  b2  b3  b4  b1  b2  b3  b4
```

Code:

```cpp
cpp
#define CRYPTONOTE_MINED_MONEY_UNLOCK_WINDOW_V1         5
uint64_t ht = height - CRYPTONOTE_MINED_MONEY_UNLOCK_WINDOW_V1;
crypto::hash h0 = bc->get_block_id_by_height(ht, alt_chain);

uint8_t b1 = (uint8_t)(h0.data[0] ^ h0.data[16]);
uint8_t b2 = (uint8_t)(h0.data[4] ^ h0.data[20]);
uint8_t b3 = (uint8_t)(h0.data[8] ^ h0.data[24]);
uint8_t b4 = (uint8_t)(h0.data[12] ^ h0.data[28]);

crypto::hash h1 = bc->get_block_id_by_height(ht - b1, alt_chain);
crypto::hash h2 = bc->get_block_id_by_height(ht - b2, alt_chain);
crypto::hash h3 = bc->get_block_id_by_height(ht - b3, alt_chain);
crypto::hash h4 = bc->get_block_id_by_height(ht - b4, alt_chain);
```

At the end of the first step, we then have 4 block hashes to be used in step 2. By using the last block hash, we have an unpredictable method of retrieving block hashes which results in a different set of hashes each block. Also, since the last block hash is not known until the new block begins mining, there is no way to precalculate the information and all miners get the last block hash at the same time.

There is one drawback to this method however and it is quite significant. Since each block needs the hash of the previous block to calculate the hash algorithm, blockchains must be synced one block at a time. When attempting to sync more than one block at a time, the previous block hash is not stored on disk and the hash to verify the block cannot be calculated. This results in longer sync times, and a higher load on nodes providing the blockchain data.

Step 2: Copy the block hashes to a single 128 byte buffer

In step 2 we convert the 4 32 byte hashes into a single 128 byte buffer. This is done simply by interleaving the 4 hashes together in 4 byte chunks, which can be visualized as

```
text
h1 0000            0000            0000
h2 |   1111        |   1111        |   1111
h3 |   |   2222    |   |   2222    |   |   2222
h4 |   |   |   3333|   |   |   3333|   |   |   3333
   |   |   |   |   |   |   |   |   |   |   |   |
   00001111222233330000111122223333000011112222333 3....

Code:
cpp
int j = 0;
for (int i = 0; i < 128; i += 16)
{
    std::memcpy(cn_bytes + i, h1.data + j, 4);
    std::memcpy(cn_bytes + i + 4, h2.data + j, 4);
    std::memcpy(cn_bytes + i + 8, h3.data + j, 4);
    std::memcpy(cn_bytes + i + 12, h4.data + j, 4);
    j += 4;
```

```
}
```

This buffer is then used to generate a struct `random_values` in step 3

Step 3: Generate the random values for the algorithm

Now we are at the step where we will generate the random values that will randomize the scratchpad data.
First the layout of the `random_values` struct (defined in src/crypto/hash-ops.h):

```
cpp
#define RANDOM_VALUES 32
enum {
  NOP = 0,
  ADD,
  SUB,
  XOR,
  OR,
  AND,
  COMP,
  EQ
};

typedef struct randomizer_values
{
  //list of bitwise functions to perform from the list above
  uint8_t operators[RANDOM_VALUES];
  //list of scratchpad locations to perform the bitwise operations on
  //this has to be uint32_t as the scratchpad is too long to store
every possible index in uint16_t
  uint32_t indices[RANDOM_VALUES];
  //a list of values to perform the bitwise operations with
  int8_t values[RANDOM_VALUES];
} random_values;
```

As you may deduce, each array is 32 values long. Each index in these arrays functions in a group

- Scratchpad index - `indices[n]`: This is the index in the scratchpad the byte manipulation will occur
- Operator - `operators[n]`: A value in the range of 0-7 corresponding to the enum to determine which function is performed
- Value - `values[n]`: The operand of the bitwise operation defined by `operators[n]`

The easiest way to describe what the operators are is with the code

```
cpp
void randomize_scratchpad(random_values *r, uint8_t *scratchpad)
{
    if (r == NULL)
        return;
    for (int i = 0; i < RANDOM_VALUES; i++)
    {
        switch (r->operators[i])
        {
            case ADD:
                scratchpad[r->indices[i]] += r->values[i];
                break;
            case SUB:
                scratchpad[r->indices[i]] -= r->values[i];
                break;
            case XOR:
                scratchpad[r->indices[i]] ^= r->values[i];
                break;
            case OR:
                scratchpad[r->indices[i]] |= r->values[i];
                break;
            case AND:
                scratchpad[r->indices[i]] &= r->values[i];
                break;
            case COMP:
                scratchpad[r->indices[i]] = ~r->values[i];
                break;
            case EQ:
                scratchpad[r->indices[i]] = r->values[i];
                break;
```

```
        }
    }
}
```

After this, 32 bytes of the scratchpad are altered, the salt is derived from the data of a random set of blocks, and the rest of the Cryptonight algorithm completes.

Conclusion

The Solo Project has a CPU favourable Proof of Work hashing algorithm that requires a fully synced copy of the blockchain on each mining node. As such it promotes decentralization as all mining nodes are full nodes, rather than clients to a decreasing number of full nodes as nethash rises (a common scenario seen in current large blockchain networks). As the network grows, the developers of the Solo project plan to bring block times down to 15 seconds. This will lower the difficulty of finding blocks for low hashrate miners.

The Solo Project aims to remain a fast, scalable, private cryptocurrency with a widely distributed blockchain for decades to come. The economy is designed with a block reward large enough to keep even the smallest miners interested in mining during the long run. As the number of miners online increases, the distribution of the full blockchain increases which is a tenet of decentralization. Fee levels will remain low, and the number of coins used in everyday transactions will be entire coins not fractions. Contributors to the Solo project feel that this is very important for mainstream adoption, as working with fractions of a coin appears to be a mental block for many people getting into larger blockchain projects as valuation has dramatically risen over the years. These issues are at the core of what the Solo Project aims to resolve.

References

[1] https://bitcoin.org/bitcoin.pdf
[2] https://cryptonote.org/whitepaper.pdf

[3]https://nbviewer.jupyter.org/github/masari-project/research-corner/blob/master/secor/secor.pdf
[4] https://web.getmonero.org/resources/research-lab/pubs/MRL-0005.pdf
[5] https://eprint.iacr.org/2017/1066.pdf
[6] https://bitbucket.org/snippets/nerva-project/keG5G8/the-cn-adaptive-v2-algorithm
[7]https://masteringmonero.com/book/Mastering%20Monero%20First%20Edition%20by%20SerHack%20and%20Monero%20Community.pdf
[8] https://github.com/zawy12/difficulty-algorithms/issues/3
[9] https://hackernoon.com/holy-cow-i-sent-a-bitcoin-transaction-with-too-low-fees-are-my-coins-lost-forever-7a865e2e45ba
[10] https://da-data.blogspot.com/2014/08/minting-money-with-monero-and-cpu.html