Autolykos: The Ergo Platform PoW Scheme

Alexander Chepurnoy[*], Vasily Kharin[†], Dmitry Meshkov[‡]

December, 04, 2020
v2.0

Abstract

This document describes family of Autolykos Proof-of-Work algorithms used in Ergo cryptocurrency. From block 1 until 414,720 Autolykos version 1 is used on Ergo mainnet. Since block 414,720 Autolykos version 2 is used.

1 Autolykos version 2

Autolykos version 2 is following Autolykos version 1 defined in Section 2, but with certain modifications made:

- non-outsourceability switched off. It turns out (based on more than 1 year of non-outsourceable PoW experience) that non-outsourceable PoW is not attractive to small miners.
- now algorithm is trying to bind an efficient solving procedure with a single table of 2 GB
- table size (memory requirements of a solving algorithm) grows with time
- now table is depending on block height only, so there is no penalization for recalculating block candidate for the same height

Solution verification and proving are given in Alg. 1 and Alg. 2.
Please note that:

1. $H()$ is now not $mod\ q$, but ordinary $blake2b256$ with 256-bit output
2. $takeRight(n, \cdot)$ is taking n least significant (right) bytes (from 32 bytes array)
3. sum is always about 256 bits (32 bytes)
4. h is height of a block (32 bits = 4 bytes)

[*] Autolykos v. 1 and v. 2
[†] Autolykos v. 1
[‡] Autolykos v. 1

5. N value is growing with time as follows. Until block $614,400$, $N = 2^{26} = 67,108,864$. From this block, and until block $9,216,000$, every 51,200 blocks N is increased by 5 percent. Since block $9,216,000$, value of N is fixed and equals to $2,147,387,550$. Test vectors for N values are provided in Table 1.
6. M is still equal to $8 * 1,024$ (bytes)

Algorithm 1 Solution verification and proving

Input: $m, nonce$
$i := takeRight(8, H(m||nonce))\ mod\ N$
$e := takeRight(31, H(i||h||M))$
$J := genIndexes(e||m||nonce)$
$f := \sum_{j \in J} takeRight(31, H(j||h||M))$
require $H(f) < b$

Algorithm 2 Block mining

Input: upcoming block header hash m, block height h
Calculate $r_{i \in [0,N)} = takeRight(31, H(j||h||M))$
while *true* **do**
    $nonce \leftarrow$ rand
    $i := takeRight(8, H(m||nonce))\ mod\ N$
    $e := takeRight(31, H(i||h||M))$
    $J := genIndexes(e||m||nonce)$
    $f := \sum_{j \in J} r_j$
    if $H(f) < b$ **then**
        return $(m, nonce)$
    end if
end while

Height	N value
500,000	67,108,864
600,000	67,108,864
614,400	70,464,240
700,000	73,987,410
788,400	81,571,035
1,051,200	104,107,290
9,216,000	2,147,387,550
29,216,000	2,147,387,550

Table 1: Test vectors for N values

2 Autolykos version 1

2.1 Introduction

Security of Proof-of-Work blockchains relies on multiple miners trying to produce new blocks by participating in PoW puzzle lottery, and the network is secure if the majority of them are honest. However, the reality becomes much more complicated than the original one-CPU-one-vote idea from the Bitcoin whitepaper[1].

The first threat to decentralization came from mining pools – miners tend to unite in mining pools. Regardless of the PoW algorithm number of pools controlling more then 50% of computational power is usually quite small: 4 pools in Bitcoin, 2 in Ethereum, 3 in ZCash, etc. This problem led to the notion of non-outsourceable puzzles [2, 3]. These are the puzzles constructed in such a way that if a mining pool outsources the puzzle to a miner, miner can recover pool's private key and steal the reward with a non-negligible probability. However the existing solutions either have too large solution size (kilobyte is already on the edge of acceptability for distributed ledgers) or very specific and can not be modified or extended in any way without breaking non-outsourceability.

The second threat to cryptocurrencies decentralization is that ASIC-equipped miners are able to find PoW solutions orders of magnitude faster and more efficiently than miners equipped with the commodity hardware. In order to reduce the disparity between the ASICs and regular hardware, memory-bound computations where proposed in [4]. The most interesting practical examples are two asymmetric memory-hard PoW schemes which require significantly less memory to verify a solution than to find it [5, 6]. Despite the fact that ASICs already exist for both of them [7, 8], they remain the only asymmetric memory-hard PoW algorithms in use.

In this paper we propose *Autolykos* — new asymmetric memory-hard non-outsourceable PoW puzzle. In Section 3 we provide a full specification of *Autolykos*, while in Section 3.1 we discuss its properties. Few auxiliary algorithms are placed in Appendix.

3 Autolykos v.1 puzzle

The proposed scheme requires following components:

1. Cyclic group $\mathbb{G}$ of prime order q with fixed generator g and identity element e. Secp256k1 elliptic curve is used for this purposes.
2. Number of elements k required in the solution. Value $k = 32$ is used in implementation.
3. Number N of elements in the list $R \subset \mathbb{Z}/q\mathbb{Z}$ to be stored in miner's memory. Value $N = 2^{26}$ is used in implementation.
4. Hash function H which returns the values in $\mathbb{Z}/q\mathbb{Z}$. Particular implementation is based on Blake2b256 and is described in Alg.5.
5. Hash function $genIndexes$ which returns a list of numbers from $0 \dots (N-1)$ of size k. It is based on Blake2b256 and is described in Alg.6.

6. Target interval parameter b, that is recalculated via difficulty adjustment rules.
7. Constant message $M = [0, \ldots, 1023].flatMap(i => Longs.toByteArray(i))$ that is used to enlarge message size and increase elements calculation time.

Autolykos is based on one list k-sum problem: miner should find k elements from the pre-defined list R of size N, such that $\sum_{j \in J} r_j - sk = d$ is in the interval $\{-b, \ldots, 0, \ldots, b \mod q\}$. In addition, we require set of element indexes J to be obtained by one-way pseudo-random function $genIndexes$. This prevents optimizations as soon as it is hard to find such a seed, that $genIndexes(seed)$ returns the desired indexes.

Thus we assume that the only option for miner is to use the simple brute-force algorithm 3 to create a valid block.

Algorithm 3 Block mining

Input: upcoming block header hash m, key pair $pk = g^{sk}$
Generate randomly a new key pair $w = g^x$
Calculate $r_{i \in [0,N)} = H(j||M||pk||m||w)$
while $true$ **do**
    $nonce \leftarrow \mathsf{rand}$
    $J := genIndexes(m||nonce)$
    $d := \sum_{j \in J} r_j \cdot x - sk \mod q$
    if $d < b$ **then**
        return $(m, pk, w, nonce, d)$
    end if
end while

Note that although the mining process utilizes private keys, solution itself only contains public keys. Solution verification can be performed by Alg. 4.

Algorithm 4 Solution verification

Input: $m, pk, w, nonce, d$
require $d < b$
require $pk, w \in \mathbb{G}$ and $pk, w \neq e$
$J := genIndexes(m||nonce)$
$f := \sum_{j \in J} H(j||M||pk||m||w)$
require $w^f = g^d pk$

3.1 Discussion

First, notice that in Algorithm 3 we refer to construction $f(m, nonce, w, pk) = \sum_{j \in genIndexes(m||nonce)} H(j||M||pk||w)$ as a hash function. Public key plays a role of commitment. Therefore, the pair (pk, d) is a Schnorr signature with a public key w over the message $(m, nonce)$ with a hash function f. If one denotes e the corresponding value of f, and pass to more common

notations: $e = f(m, nonce, w, w^e g^{-d})$. The puzzle consists in trying different nonces and keys in order for signature to satisfy $d \in \{-b, \ldots, 0, \ldots, b\}$. Security follows from the security of Schnor signatures, and outsourcing the puzzle is equivalent to outsourcing the signature (or parts of signature creation routine). The only difference from conventional setup is the design of function f. It must be constructed in such a way that efficient massive evaluations with different nonces require allocating large amount of memory (benefitting from data reuse), whereas single evaluation on verifier's side can be done "on fly".

To achieve this, algorithm 3 requires to keep the whole list R during the main loop. Every pre-calculated hash occupies 32 bytes, so the whole list of N elements occupies $N \cdot 32 = 2Gb$ of memory. For sure, a miner can recalculate these elements "on fly" during the main loop and thus reduce memory requirements. However in such a case the number of calls of H will significantly grow up (e.g. assuming GPU hashrate $G = 2^{30} H/s$ [9] and block interval $t = 120\ s$, every element will be used $(G/N) \cdot k \cdot t = 3 \cdot 10^4$ times on average.) reducing miner's efficiency and profit.

While list R is quite big, it's filling consumes quite a lot of time: our initial implementation [10] consumed 25 seconds on Nvidia GTX 1070 to fill list R. This part, however, may be sufficiently optimized if miner in addition stores a list of unfinalized hashes $u_{i \in [0,N)} = H(i||M||pk$ in memory, consuming 5 more Gigabytes of it. In such a case this work to calculate unfinalized hashes should be done only once during mining initialization while finalizing them and filling the list R for the new header will only consume few milliseconds (for about 50 ms on Nvidia GTX 1070).

The protocol is quite efficient in terms of solution size and verification time: it consists of 2 public keys of size 32 bytes, number d that is at most 32 bytes (but contains a lot of leading zeros in case of the small target b) and an 8-bytes long nonce. Header verification requires verifier to calculate 1 $genIndexes$ hash, k hashes H and perform two exponentiations in the group. Reference Scala implementation [11] allows verifying block header in 2 milliseconds on Intel Core i5-7200U, 2.5GHz.

References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] A. Miller, A. Kosba, J. Katz, and E. Shi, "Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 680–691.

[3] I. E. P. Daian, E. G. Sirer, and A. Juels, "Piecework: Generalized outsourcing control for proofs of work," in *BITCOIN Workshop*, 2017.

[4] C. Dwork, A. Goldberg, and M. Naor, "On memory-bound functions for fighting spam," in *Annual International Cryptology Conference*. Springer, 2003, pp. 426–444.

[5] A. Biryukov and D. Khovratovich, "Equihash: Asymmetric proof-of-work based on the generalized birthday problem," *Ledger*, vol. 2, pp. 1–30, 2017.

[6] Ethash. [Online]. Available: https://github.com/ethereum/wiki/wiki/Ethash/6e97c9cea49605264c6f4d1dc9e1939b1f89a5a3

[7] Bitmain confirms release of first ethereum asic miners. [Online]. Available: https://www.coindesk.com/bitmain-confirms-release-first-ever-ethereum-asic-miners

[8] Bitmains latest crypto asic can mine zcash. [Online]. Available: https://www.coindesk.com/bitmains-latest-crypto-asic-can-mine-zcash

[9] Non-specialized hardware comparison. [Online]. Available: https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison

[10] Autolykos gpu miner. [Online]. Available: https://github.com/ergoplatform/Autolykos-GPU-miner

[11] Autoleakus scala implementation. [Online]. Available: https://github.com/ergoplatform/ergo/tree/master/src/main/scala/org/ergoplatform/mining

Appendix

Implementation of hash function H which returns the values in $\mathbb{Z}/q\mathbb{Z}$:

Algorithm 5 Numeric hash

function H($input$)
  $validRange := (2^{256}/q) \cdot q$
  $hashed := Blake2b256(input)$
  if $hashed < validRange$ **then**
    return $hashed.mod(q)$
  else
    return $H(hashed)$
  end if
end function

Implementation of hash function $genIndexes$ which returns a list of size k with numbers in $0 \dots (N-1)$:

Algorithm 6 Index generator

function genIndexes($seed$)
  $hash := Blake2b256(seed)$
  $extendedHash := hash||hash$
  return $(0 \dots k-1).map(i => extendedHash.slice(i, i+4).mod(N))$
end function