Reducing Computational Waste: Space and Usefulness

Iskandar Pashayev
Stanford University

Abstract

Proof of Work mechanisms are essential to the security and value of cryptocurrencies. The most prominent cryptocurrency, Bitcoin, employs a Proof of Work that is asymmetric with respect to time, i.e. the prover incurs a significant time cost to determine a solution to a Bitcoin challenge but the verifier requires little time to verify a solution. However, the Proof of Work is symmetric with respect to space: both the prover and the verifier require little amounts of space. A Proof of Work that is asymmetric with respect to space could significantly reduce the waste generated by the time difficulty of the current Bitcoin Proof of Work. In this paper, we survey two Proof of Work proposals with asymmetric space requirements: Cuckoo Cycle and Equihash. We also survey a Proof of Work mechanism for which the solutions to challenges can be applied to problems of practical interest.

1 Introduction

The Bitcoin [6] electronic cash system rewards users that invest computational resources for solving a particular challenge with a coin. This coin serves as a certificate for the work performed. This interaction is commonly referred to as a Proof of Work (PoW) mechanism. The PoW mechanism is essential in giving the currency value and security from forgery and double-spending attacks. The current Bitcoin PoW mechanism requires participants to invest a significant amount of time in computing the solution to the current challenge. However, the mechanism requires very little utilization of another important computational resource: space.

The main motivation for the replacement of the current Bitcoin PoW mechanism is that the computation is wasteful. This notion of wastefulness has two unrelated concerns. The first has a physical interpretation: the computation consumes an unsustainable amount of electricity on modern computing hardware. The physical resource consumption to generate this electricity has been called an environmental disaster [5]. The second interpretation is one of *usefulness*. The solutions to the Bitcoin PoW challenges are not useful computations that have practical applications. We describe and detail a formalization of this notion in section 5 and how it can link to space-based PoWs.

First we survey the literature on Bitcoin and alternative cryptocurrencies. In particular, we describe PoW mechanisms and their motivations. We describe two proposed PoWs based on functions that are space-hard to compute but space-easy to verify: Cuckoo Cycle and Equihash. We also delve into further reducing computational waste through Proofs of Useful Work. We then describe how a new PoW mechanism can be incorporated in the current Bitcoin network and the challenges associated with the process. Lastly, we describe further work that can be done in this area.

2 Bitcoin

The main purpose of Bitcoin is to provide a secure electronic currency the exchange of which does not require a financial intermediary. A system that provides this currency needs to be resistant to double-spending attacks. To achieve such a currency, the Bitcoin system relies on a completely public transaction history for all coins stored in a blockchain. To add new blocks to the blockchain, participants in the network must solve a computational challenge, the solution of which grants the solver a valuable reward in the form of Bitcoins. At any given moment in the Bitcoin network, the longest blockchain is the one recognized by the nodes in the network. The difficulty of the computational problem makes forging a longer blockchain computationally infeasible.

2.1 Proof of Work

The Bitcoin PoW has two key importances to the Bitcoin network. First, it gives the currency *value*. By investing significant resources, solvers demand a return for the work put into mining Bitcoins. As a result, the currency gains value. Second, the Proof of Work makes it impossible for someone to forge a valid blockchain. The current Bitcoin PoW operates by scanning for an origin block such that its SHA-256 hash begins with some number of zeros, as specified by the Bitcoin network. The origin block contains a block header, a nonce, and the hash of the last block in the currently accepted chain. Assuming SHA-256 is a one-way function, solving instances of this problem is computationally intractable and requires a brute-force search. Thus, in computing a solution to an instance, one must have performed a great deal of work and invested a significant amount of computational resources. However, computational resources come in a variety of forms: time, space, randomness, etc. While the current Bitcoin PoW mechanism requires a significant investment in time, it requires very little space. In aggregate, a participant needs approximately 80 bytes [6] to find a solution to a challenge.

2.2 Altcoins

Bitcoin has spurred hundreds of alternative cryptocurrencies – also called *altcoins* – that seek to improve the original Bitcoin system. These improvements span a wide range of problems. However, we limit our discussion to those focus on utilizing space as a computational resource in their PoW mechanisms.

The scrypt hash function is the most popular basis for a memory-hard PoW mechanism [4]. Such PoW mechanisms have been implemented in the fairly popular Litecoin and Dogecoin. However, the scrypt hash function is designed to require a lot of memory regardless of its input. Thus, PoW mechanisms based on the scrypt hash function are not symmetric with respect to space. If the prover requires a lot of space to find a solution, the verifier must also use a lot of space to verify the solution. This opens the verifier to denial of service attacks as the verifier can be spammed with false solutions; it is critical for a verifier to run with little computational requirements. Thus, to satisfy this constraint, the PoW mechanisms for Litecoin and Dogecoin allow GPU and ASIC enabled provers to obtain a significant advantage over regular users. A PoW mechanism based on a function that is asymmetric with respect to space would enable a system in which provers cannot optimize costs of space without undermining the efficiency of the verifier.

3 Cuckoo Cycle

Cuckoo Cycle is the first PoW mechanism based on a graph theoretic problem [7]. Furthermore, the function that the PoW employs aims to achieve asymmetry with respect to space. In particular, the Cuckoo Cycle PoW is based on finding cycles in large random graphs. The prover then utilizes a Cuckoo hash table to find a cycle that satisfies the length requirement specified in the issued challenge.

3.1 Proof of Work

The challenge issuer fixes a hash function $h : \{0,1\}^K \times \{0,1\}^{W_i} \to \{0,1\}^{W_o}$ and a target subgraph H, which in our case is a cycle. He then samples $k \in \{0,1\}^K$, $N \in [2^{W_o}]$, and $M \in [2^{W_i-1}]$ uniformly at random. Here k is the key, N is the number of vertices in the random graph, and M is the number of edges in the random graph. With these parameters, we construct a bipartite graph $G_k = (V,E)$ as follows:

$$V = \{v_0, \ldots, v_{N-1}\},$$
$$E = \{\{v_{2(h(k,2i) \mod \frac{N}{2})}, v_{2(h(k,2i+1) \mod \frac{N}{2})+1}\} \mid 0 \le i < M)\}.$$

An input i is also called a *nonce*. To solve this challenge, a prover computes a list of nonces that identifies a subgraph of G_k that is isomorphic to H. To further fine-tune the difficulty of solving the challenge, the challenge issuer may also constrain the hash of the solution to be smaller than some specified T, i.e. the SHA-256 hash of the solution begins with a specific number of zeros.

3.2 Cycle Finding

To find the desired cycle in G_k, a prover utilizes a Cuckoo hash table. The Cuckoo hash table consists of two tables, each with N elements. Generally, each table has its own hash function. Thus, the Cuckoo hash table can be thought of as a directed bipartite graph where the out-degree for each vertex is one. We have two partitions in the Cuckoo graph, each labeling the vertex set of G_k, which we name A and B without loss of generality. For each edge $e \in E$, we fix an ordering $e = \overleftarrow{\{l,r\}}$.

Next, we iterate through the edge set of G_k. If $r \in A$ is not mapped to some $v \in B$, we add a directed edge from $r \in A$ to $l \in B$. Otherwise, if $l \in B$ is not mapped to some $v \in A$, we add a directed edge from $l \in B$ to $r \in A$. If neither case holds, we reverse the edges of the shorter directed path (that begins at $r \in A$ or $l \in B$). If the reversed path and the non-reversed path do not share the same root, we add the appropriate directed edge to the Cuckoo graph. Otherwise, we have detected

a cycle, from which it is easy to extract the cycle. If the cycle has the desired length, we return it. Otherwise we discard the last edge added to the Cuckoo graph, revert the reversed path to its original state, and continue iterating through the edge set.

3.3 Security

The sparsity of the graph, determined by the ratio of edges to vertices, approximates the level of difficulty for a given challenge in the Cuckoo Cycle PoW. Tromp conjectures that $\frac{M}{N} < 1$ satisfies the difficulty requirement for a practical PoW scheme. As the graph becomes more dense, the number of cycles (or target substructure) in the graph increases in expectation. Empirically, the probability that a graph on N vertices has a cycle of length l as a function of the ratio $\frac{M}{N}$ can be represented with a logistic function. As a result, the challenge issuer can tune the parameters M and N to achieve the desired difficulty. Furthermore, Tromp empirically shows that the number of memory accesses per nonce is superlinear with respect to the percentage of nonces processed, implying that the PoW scheme achieves the desired asymmetry with respect to space.

Edge Trimming. David Andersen describes an optimization to the cycle finding algorithm that significantly reduces the number of edges processed [1]. This optimization takes advantage of the fact that every vertex in a cycle has a degree of two. Thus, for all vertices $v \in V$ with degree one, we can eliminate edges from G_k adjacent to v. Edge trimming can also be parallelized safely for further time efficiency. As a result, the time-space tradeoff for this optimization is not very steep. Edge trimming works well if G_k is especially sparse since the prover can efficiently compute the degrees of vertices from the smaller edge set. Consequently, the challenge issuer loses his effectiveness in increasing the difficulty of a challenge by increasing the sparsity of the graph.

4 Equihash

4.1 Generalized Birthday Problem

Biryukov and Khovratovich propose an alternative asymmetric Proof of Work scheme based on the generalized birthday problem [3]. The main motivation for developing this PoW was to generate an asymmetric PoW that cannot be optimized as dramatically as the Cuckoo Cycle PoW.

The generalized birthday problem, or the k-XOR problem, is defined as follows: given a list L of n-bit strings $\{X_i\}$, determine a subset of unique $\{X_{i_j}\}$ such that the XOR of all its elements is equal to 0. In the case of Equihash, each X_i is the output of a non-keyed PRF, such as a hash function in counter mode, and so we wish to satisfy

$$H(i_1) \oplus H(i_2) \oplus \ldots \oplus H(i_{2^k}) = 0. \quad (1)$$

To solve an instance of the k-XOR problem, one can run Wagner's algorithm, which runs in time $(k+1)N$ using space $(2^{k-1}+n)N/8$ [3].

Algorithm 1 Wagner's Algorithm for solving the k-XOR problem. Given list L of n-bit strings, $N << 2^n$

1: Enumerate the list as $X_1, X_2, \ldots, X_N$ and store (X_i, i) in a table T.
2: Sort T on X_j. Find all unordered pairs (a,b) such that the first $\frac{n}{k+1}$ bits of $X_a \oplus X_b$ is equal to 0. Replace the table T with these pairs, where each i becomes a tuple (a,b).
3: Repeat step 2 for the next $\frac{n}{k+1}$ bits until $\frac{2n}{k+1}$ bits are non-zero.
4: Find a collision on the last $\frac{2n}{k+1}$ bits.
5: Return list $\{i_j\}$ satisfying Equation (1).

4.2 Security

Amortization. One issue with the k-XOR problem is that with sufficient memory, one can amortize solutions of instances by storing strings that generate subsets of colliding bits. However, despite this amortizability, Biryukov and Khovratovich notice that intermediate 2^l-XORs collide on some $\frac{nl}{k+1}$ bits. Thus, by requiring solutions to satisfy which bits these intermediate 2^l XORs agree on, we reduce the amortizability of the original k-XOR problem with high probability. The authors refer to this technique as *algorithm binding* since this constraint binds the solver to a specific algorithm flow.

Time-Space Tradeoffs. A solver can perform two types of optimizations for computing Wagner's algorithm. To reduce the size of the table T in memory, a prover might recompute intermediate XOR values from the indices. However, this is impractical for large k as the recomputations take too much time. Second, the prover might store only a fraction t of the bits for each index. Then after the last step, the prover recomputes the missing fraction of index bits for the solution. For large t it suffices to check the XOR of each possible pair of the solution. For smaller t, the prover essentially repeats the algorithm with 2^k different lists. An empirical result shows that a combination of these two optimizations yields the best results [3]. However, the optimized

Wagner algorithm still requires a non-trivial amount of space and time to run. More specifically, the optimized algorithm runs in $k2^{\frac{n}{k+1}+2}$ time and $2^{\frac{n}{k+1}}(2^k + \frac{n}{2k+2})$ bytes of memory. Thus, with proper choice of parameters to the problem, the challenge issuer can set difficulty requirements for solving challenges at little verification overhead.

Parallelism. There are two ways in which parallelism can be sought after during computation time. The first is in the choice of sorting algorithm and the second in the collision search. Biryukov and Khovratovich do not claim security against parallelization. However, both forms require a memory access bandwidth that restricts the effectiveness of modern GPUs and ASICs. In particular, with p processors, memory bandwidth grows by a factor of p. Since the best commercial products do not exceed 512 GB/s bandwidth and regular desktop computers can have a 17 GB/s bandwidth, solvers utilizing GPUs and ASICs are not at an overwhelming advantage over regular users.

4.3 Proof of Work

The Equihash PoW mechanism works as follow. First, the challenge issuer selects a cryptographic hash function H and integers n, k, d used to specify the following PoW constraints:

(a) Memory M is $2^{\frac{n}{k+1}+k}$ bytes

(b) Time T is $(k+1)2^{\frac{n}{k+1}+d}$ calls to the hash function H

(c) Solution size is $2^k(\frac{n}{k+1}+1)+160$ bits

(d) Verification cost is 2^k hashes and XORs.

Next, he selects a seed I and issues a challenge to find a 160-bit nonce V and $(\frac{n}{k+1}+1)$-bit $x_1, x_2, \ldots, x_{2^k}$ such that the nonce satisfies:

(a) $H(I\|V\|x_1) \oplus \ldots \oplus H(I\|V\|x_{2^k}) = 0$

(b) $H(I\|V\|x_1\|\ldots\|x_{2^k})$ has d leading zeros.

(c) $H(I\|V\|x_{w2^l+1}) \oplus \ldots \oplus H(I\|V\|x_{w2^l+2^l})$ has $\frac{nl}{k+1}$ leading zeros for all w, l

(d) $(x_{w2^l+1}\|\ldots\|x_{w2^l+2^{l-1}}) < (x_{w2^l+2^{l-1}+1}\|\ldots\|x_{w2^l+2^l})$

Here, (a) is the solution to the instance of the generalized birthday problem, (b) sets the difficulty requirement that the challenger imposes on the solver, and (c) is the algorithm binding to prevent amortization.

5 Proofs of Useful Work

A Proof of Useful Work (uPoW) mechanism must satisfy a *hardness* requirement similar to a regular PoW: providing correct solutions to challenges implies that the prover performed a meaningful amount of work. Furthermore, a uPoW must also satisfy a notion of *usefulness*, i.e. the solution to a network-issued challenge can be quickly and verifiably reconstructed from the solvers' response. Thus, a uPoW has an extra `Recon` algorithm which, given the solution to a challenge for the uPoW, reconstructs a solution to an instance of a problem that reduces to the problem used by the uPoW. A repository of solutions to instances of these hard problems can then be referred to for related practical tasks. Ball et al. show uPoW mechanisms for the Orthogonal Vectors, 3SUM, and All-Pairs Shortest Paths problems [2]. Furthermore, the network can issue challenges for problems that reduce to one of OV, 3SUM, and APSP. Thus, instances of such problems with practical interest can be delegated to the computing network as a means of faster and less wasteful computation.

5.1 Orthogonal Vectors

The k-Orthogonal Vectors (k-OV) problem is specified as follows. Given k sets of n vectors, $(U_1, \ldots, U_k)$, where each vector has dimension d. Decide if there exist $u^s \in U_s$, $s \in [k]$ such that the dot product is 0:

$$\sum_{l\in[d]} u_l^1 \cdots u_l^k = 0.$$

While k-OV is conjectured to be worst-case hard, there is no conjecture for its difficulty in the average case. As a result, Ball et al. define a related problem gOV which, assuming worst-case hardness for k-OV, implies average-case hardness for gOV. More specifically

$$g\mathrm{OV}^k_{n,d,p} : \mathbb{F}_p^{knd} \to \mathbb{F}_p$$

Here, p is prime and all other parameters are the same as in k-OV. gOV is then computed as

$$g\mathrm{OV}^k_{n,d,p}(U_1, \ldots, U_k) = \sum_{i_1,\ldots,i_k\in[n]} \prod_{l\in[d]} (1 - u^1_{i_1 l} \cdots u^k_{i_k l}).$$

In words, gOV is a polynomial of degree kd that computes the number of solutions to an instance of the k-OV problem.

5.2 Interactive Proof of gOV

For $l \in [d]$, let $\phi_l^s : \mathbb{F}_p \to \mathbb{F}_p$, where ϕ_l^s represents the l-th column of U_s. Each such ϕ_s has degree at most $(n-1)$.

Let $q(i_1,\ldots,i_k) = \prod_{l\in[d]}(1-\phi_l^1(i_1)\cdots\phi_l^k(i_k)$. So

$$g\mathrm{OV}^k_{n,d,p}(U_1,\ldots,U_k) = \sum_{i_1,\ldots,i_k\in[n]} q(i_1,\ldots,i_k).$$

In this case, we have that gOV has degree $(n-1)kd$. For any $s\in[k]$ and $\alpha_1,\ldots,\alpha_{s-1}\in\mathbb{F}_p$, define

$$q_{s,\alpha_1,\ldots,\alpha_{s-1}}(x) = \sum_{i_{s+1},\ldots,i_k\in[n]} q(\alpha_1,\ldots,\alpha_{s-1},x,i_{s+1},\ldots,i_k),$$

which we denote q_s in short-hand notation. Here q_s is a univariate polynomial with degree at most $(n-1)d$.

An interactive proof for gOV works as follows:

1. Prover sends the coefficients of q_1^*
2. Verifier checks that $\sum_{i_1\in[n]} q_1^*(i_1) = y$. Reject if not equal.
3. For $s = 1,\ldots,k-2$
   (a) Verifier sends random $a_s \leftarrow \mathbb{F}_p$
   (b) Prover sends the coefficients of $q^*_{s+1,\alpha_1,\ldots,\alpha_s}$
   (c) Verifier checks that $\sum_{i_{s+1}\in[n]} q^*_{s+1,\alpha_1,\ldots,\alpha_s}(i_{s+1}) = q^*_{s,\alpha_1,\ldots,\alpha_{s-1}}(\alpha_s)$. Reject if not equal.
4. Verifier samples $\alpha_{k-1} \leftarrow \mathbb{F}_p$ and checks that $q^*_{k-1,\alpha_1,\ldots,\alpha_{k-2}}(\alpha_{k-1}) = q_{k-1,\alpha_1,\ldots,\alpha_{k-2}}(\alpha_{k-1})$. Reject if not equal.
5. Verifier accepts.

The above protocol is an interactive proof for gOV with perfect completeness and soundness error at most $\frac{knd}{p}$ [2]. Completeness is straightforward to see from the definitions of q and q_s. We analyze soundness in more depth. So a cheating prover wishes to have the verifier accept a false solution. Then the prover's q_1^* must be different from q_1. Furthermore, since the degree of q_1^* is less than nd, the probability that $q^*(\alpha_1) = q_1(\alpha_1)$ for a uniformly random α_1 is less than $\frac{nd}{p}$. If the prover passes this check, then the verifier again sends an α_2 for which the prover must again send a q^*_{2,α_1} that is not equal to q_{2,α_1}. The probability of an equality in this case is again less than $\frac{nd}{p}$. Since the protocol has $k-1$ rounds, the probability that the prover passes all the checks and gets the verifier to accept is less than $\frac{knd}{p}$.

5.3 Useful Proof of Work Scheme

The following four algorithms define a uPoW mechanism based on the k-OV problem.

1. `Gen`(x): Given an instance $x\in\{0,1\}^{knd}$, interpret x as an element of $\mathbb{F}_p^{knd}$. Sample a random field element $r\in\mathbb{F}_p^{knd}$. Output a challenge as the set of vectors $c_x = \{y_t = x + tr \mid t\in[kd+1]\}$
2. `Solve`(c_x): Compute $z_t = g\mathrm{OV}^k_{n,d,p}(y_t)$ and output the set $s = \{z_t\}_{t\in[kd+1]}$. Run the interactive protocol described in section 5.2 in parallel with `Verify`.
3. `Verify`(c_x, s): Run the interactive protocol described in section 5.2 in parallel with `Solve`.
4. `Recon`(c_x, s): Interpret $s = \{z_t\}$ as the evaluations of a univariate polynomial $h(t)$ at $t = 1,\ldots,kd+1$. Interpolate to find the coefficients of h and compute $z = h(0)$. If $z\neq 0$, output 1 as the solution for k-OV, else output 0.

Any problem that can be efficiently reduced to k-OV can have its instances delegated as challenges in this scheme. In particular, Ball et al. mention any graph theoretic property that can be stated in first order logic can be reduced to k-OV.

6 Discussion

6.1 Integration of new PoW Mechanism

Updates to Bitcoin can be performed at three different levels: *hard forks*, *soft forks*, and *relay policy updates* [4]. The network requires a hard fork if the update invalidates transactions or blocks under previous rules. Since miners that would upgrade to solve this new computational puzzle would produce blocks that would be rejected by miners running an earlier version, the network would have to perform a hard fork. Consequently, hard-forks require near-unanimity amongst the miners to successfully update the network. Given the current investment of Bitcoin miners in ASICs designed to solve instances of the current SHA-256 hash problem, such near-unanimous acceptance poses a significant challenge to the PoW mechanism update.

6.2 Future Work

Further work can be done on the analysis of cycle finding with Cuckoo hash maps. It is possible that there are more efficient cycle finding algorithms. Furthermore, it is unknown if Wagner's algorithm is the most efficient algorithm for solving the generalized birthday problem. More efficient algorithms would break both PoW schemes. Tromp explores cycles in his PoW mechanism, but leaves cliques and independent sets for further review. The choice of substructure in the graph is arbitrary, which allows for a diverse set of graph-based PoW

mechanisms. Another area of research for PoW mechanisms is in the quantum computer setting; there may be algorithms that run on quantum computers which completely break modern PoW mechanisms. In this case, we will need new Proofs of Work based on harder problems that are hard even for a quantum computer. One such example could be Ramsey number calculation.

References

[1] ANDERSEN, D. A public review of cuckoo cycle, 2014.

[2] BALL, M., ROSEN, A., SABIN, M., AND VASUDEVAN, P. N. Proofs of useful work. Tech. rep., IACR Cryptology ePrint Archive, 2017.

[3] BIRYUKOV, A., AND KHOVRATOVICH, D. Equihash: asymmetric proof-of-work based on the generalized birthday problem. *Proceedings of NDSS 2016* (2016), 13.

[4] BONNEAU, J., MILLER, A., CLARK, J., NARAYANAN, A., KROLL, J. A., AND FELTEN, E. W. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 104–121.

[5] MCCOOK, H. An order-of-magnitude estimate of the relative sustainability of the bitcoin network, 2014.

[6] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system.

[7] TROMP, J. Cuckoo cycle: a memory-hard proof-of-work system. *IACR Cryptology ePrint Archive 2014* (2014), 59.