Certified Solutions on Blockchain to Computationally Difficult Optimization Problems

Alberto Leporati

University of Milan-Bicocca, Department of Informatics, Systems and Communication, Edificio U14 (ABACUS), Viale Sarca 336, 20126 Milano, Italy

Abstract

Blockchains allow to store in an unalterable and secure way different types of information. Because this information can be associated with the precise moment (timestamp) at which it is stored, it is sometimes referred to as notarization, or certification. In this paper we propose to use blockchains to store in a certified way solutions to computationally difficult optimization problems. Specifically, we show how it is possible to encode within a smart contract the verification of a proposed solution and its subsequent storage. After discussing how the proposed framework works, and the security assumptions made, we consider as use cases two famous NP-hard problems: Knapsack and Max-SAT, which are representative of a large class of optimization problems that arise in the real world. For each of the two cases we briefly discuss the peculiarities of the corresponding smart contract, showing how it fits in the general framework.

Keywords

Blockchain, Computationally Difficult Optimization Problems, Certified Storage of Solutions

1. Introduction

A blockchain is a decentralized ledger shared between nodes on a computer network. These nodes work together to maintain a secure, immutable and decentralized record of data.

Typical uses of blockchains include supply chain management, document notarization, decentralized finance (DeFi) applications, and tokenization of digital and physical assets. Every application needs a blockchain with certain characteristics, and in fact there are different types of blockchains. Thus, a blockchain can be *public* or *private*, depending upon whether its contents is publicly available or not. Furthermore, a blockchain can be either *permissionless* or *permissioned*.

Another common use of blockchains is the *notarization* (also *certification*) of documents. The idea is to demonstrate that a digital document existed in exactly that form at a given time. Since the block size does not allow documents to be saved directly in the blockchain, the document file is usually saved on a decentralized file system, such as IPFS[1], while the document's cryptographic hash fingerprint is saved on the blockchain. A natural next step is the certification of processes, or workflows. In these cases the blockchain is used to certify the correct execution of the processing phases, collecting information and documents both from the machinery and from the people involved in the process.

In this paper we assume that a research group, or a company, has to solve a computationally difficult optimization problem. This problem can be related for example to the optimization of transport lines or optimal delivery of goods, the search for a new medicine, optimal scheduling of processes, DNA alignment, etc. In any case, we assume that finding the optimal solution can bring great advantages to the people who proposed the problem, such as the release of additional research funds, competitiveness on the market, or other. Therefore, we assume that the team who proposes the problem is willing to pay a reward to the person who finds the optimal solution. Since often even a suboptimal solution can still be enough, the reward will be paid to whoever found the best solution among those proposed, i.e. the one that has the greatest value of the objective function to be maximized (or the lowest value, in case the objective function must be minimized).

DLT2025: 7th Distributed Ledger Technology Workshop, June, 12-14 2025 – Pizzo, Italy

alberto.leporati@unimib.it (A. Leporati)

0000-0002-8105-4371 (A. Leporati)

[1] https://ipfs.tech/

Thus, we propose to use a blockchain to store in a certified way the solutions of the problem. The proposer of the problem publishes a smart contract that checks the validity of submitted solutions, and stores valid solutions along with a timestamp. The contract accepts proposed solutions for a predefined submission time, after which the contract owner rewards the one who submitted the best solution. To clarify how this mechanism can be implemented, in the following we consider two examples of computationally hard optimization problems: Knapsack and Max-SAT, which are representative of a large class of real-world optimization problems. For each of the two cases we discuss how the corresponding smart contract is made, and how it fits into the general framework.

Using a blockchain, rather than a centralized application managed by the problem proposer, makes the entire process more transparent and robust against fraud. In fact, anyone will be able to examine the proposed solutions stored in the state of the smart contract, and this information cannot be modified. We will also discuss some security issues and assumptions made to mitigate possible attacks, the most important of which is *solution stealing*: when a proposed solution is sent to the public mining pool, a dishonest user can see it and propose the same solution with a higher fee for the miner/validator.

The rest of the paper is structured as follows. In Section 2 we describe the design and operation of the proposed framework, and we discuss some security concerns as well as possible mitigations. In Section 3 we present the smart contracts that deal with the Knapsack and the Max-SAT optimization problems. In Section 4, we discuss some related works found in the literature. Finally, Section 5 draws conclusions and offers some perspectives for future work.

2. The Proposed Framework

In this section we describe how the proposed framework is made and how it works. Let us clarify that the implementation of the framework is still in its early stages; therefore, albeit presenting a possible software architecture for the system, we will focus our attention on smart contracts, which are the most relevant parts of the system in the context of blockchains. We assume that an Ethereum-based blockchain is used, and that smart contracts are written in the Solidity language.

Assume that a research group, or a company, has to solve some real-world computationally difficult optimization problem, for which checking a proposed solution is much less computationally expensive than finding an optimal solution (this is the typical situation when considering NP-hard optimization problems). Since even suboptimal solutions can be very valuable to the research team, they are willing to reward anyone who provides the best among submitted solutions. As a first step, the research group formalizes the optimization problem, defining precisely how its instances are made and how the cost function that associates an integer or real value to each admissible solution is defined. Then, they write a smart contract that checks whether a submitted solution is admissible (i.e., valid), and calculates its cost. The smart contract also stores the submitted solution in its state if it has a better cost value than the solutions proposed so far. The smart contract will contain the data structures and variables needed to represent the problem instance and the submitted solutions. The variables containing the problem instance will be assigned during the creation of the smart contract via the `constructor()` function. The submitted solutions will be contained in the array `historyOfBestSolutions`, which will be populated over time during the expected submission period. This period can be closed (and never reopened) by the contract owner by calling the `closeSubmissions()` functions, that sets the value of the boolean variable `submissionsOpen` to `false`. Anyone can check whether we are in the submission period by calling the `areSubmissionsOpen()` function.

A solution can be submitted using the `submitSolution()` function, which is the core of the smart contract. The precise logical flow of this function is illustrated in Algorithm 1. It takes as input a proposed solution and returns `true` if it is valid and better than the solutions stored so far; otherwise, it returns `false`. Moreover, in the former case it appends the proposed solution at the end of the `historyOfBestSolutions` array, and emits the `SolutionFound()` event. In this way, the smart contract stores the sequence (history) of increasingly better solutions submitted to it over time. For each of these solutions, also the timestamp and the address of the proposer are stored. Note that

`checkValidity()` is an independent function, that can be called by anyone to check whether a proposed solution is valid; it returns a Boolean value, and does not modify the state of the contract.

Algorithm 1 Pseudo-code of the `submitSolution()` function

Input: A proposed solution
Output: true if solution is valid and better than currently known solutions, false otherwise

```
1:  function SUBMITSOLUTION(solution)
2:      if submissions are not open then return false
3:      if not checkValidity(solution) then return false
4:      currently_best ← last solution stored in historyOfBestSolutions array
5:      if value(solution) > value(currently_best) then
6:          append solution to the end of historyOfBestSolutions
7:          emit SolutionFound() event
8:          return true
9:      else
10:         return false
11:     end if
12: end function
```

Finally, the smart contract contains some helper functions. The `getNumberOfSolutions()`, `getStoredSolution()`, and `getBestSolution()` functions allow one to get the number of solutions stored in the contract, to get the details of a specified (through its index in the `historyOfBestSolutions` array) solution, and to get the details of the best among the stored solutions (that is, the last one in the `historyOfBestSolutions` array), respectively. If no such solutions exist, because the specified index in the `historyOfBestSolutions` array is out of range, or because the array is empty, a dummy empty solution is returned.

Let us now look at some potential security issues, and how we can try to mitigate them. If the smart contract is published on a public permissionless blockchain, such as Ethereum, it is subject to the *solution stealing* attack: since in this type of blockchains all transaction proposals end up in the public mining pool and are visible to all miners, it may happen that a dishonest miner tries to include in the block they are building their own transaction containing the solution to the optimization problem, instead of the transaction proposed by the user who found it. This attack can also be mounted by any user who is able to look at the content of the mining pool: when they see a solution, they propose a similar transaction but with a higher fee for the miner. While a complete solution to this problem is difficult and yet to be found, to mitigate this attack we suggest that the proposer of the problem rely on an existing permissioned blockchain, or create their own. In the former case, the team will pay the use of the blockchain to the consortium that operates it. In the latter case, the research team should ensure a good level of decentralization, installing a node at each of the partners participating in the project. This should not be a problem in the case of European projects, for example, where a distribution of competences between partners from several different countries is foreseen.

If the research group sets up its own permissioned blockchain, it will also run a certification authority, which allows those who want to participate in solving the problem to register and receive the credentials needed to submit the solutions to the smart contract. The blockchain will be used as a distributed ledger, using proof-of-authority (PoA) as a consensus algorithm. Usually, information is written to the blockchain very quickly, since it is not necessary to wait for miners or validators to reach consensus and authorize the transactions. This mitigates the possibility that other users look at the mining pool and steal the proposed solutions. However, a dishonest node might still be able to steal the solutions it is supposed to validate. We believe that there is no technical solution to this problem; instead, the nodes that make up the consortium that manages the permissioned blockchain could sign a legal contract stating that they cannot propose solutions, and that they commit to pay damages in case of incorrect behavior. The same consideration applies to dishonest uses of the certification authority, such as revoking a user's credentials so they can't submit solutions. Clearly, this mitigation is not possible in

the case of adopting a permissionless blockchain.

A well known problem of permissioned blockchains is that they are completely controlled by the consortium. This means that if the consortium members agree, they can modify the content of the blocks as they want. This problem is especially felt for small consortia. An often adopted solution to prevent this possibility is to save the cryptographic hash of the last block of the blockchain on a well-known public blockchain, such as Ethereum or Polygon[2], at established time intervals. The exact amount of time will be established by the consortium, considering that the more frequent the notarization on the public blockchain, the more trust is instilled in users (but the more expensive the operation). Clearly, in this way the solutions remain confined to the permissioned blockchain; the notarization on the public blockchain only serves to demonstrate that the data present on the permissioned blockchain has been altered, but does not allow for its recovery. A more expensive but still more transparent solution would be to publish a nearly identical copy of the smart contract on the permissionless blockchain. The differences between this copy and the original smart contract would be the following: (1) only the owner of the contract can save the proposed solutions, and (2) in doing so, they would also save the address of the user who proposed the solution. In this way, every time a new solution is saved in the smart contract on the permissioned blockchain, the owner of the contract would save a copy of the same solution on the permissionless blockchain. This would allow public scrutiny of the entire sequence of proposed solutions. Let us note, however, that running the `submitSolution()` and `checkValidity()` functions on a permissionless blockchain may be expensive in terms of gas used. This is another reason why we believe a permissioned blockchain – in which gas is not used – may be more appropriate.

Notice that the solution theft issue is similar to frontrunning and MEV (Maximally Extractable Value) in financial uses of blockchains [1]. To mitigate these problems (or, at least, to democratize them), infrastructures such as flashbots[3] have been proposed for Ethereum. In this case, proposed solutions do not go through the public pool but are sent directly to the block proposer. However, dishonest block proposers may still steal solutions; furthermore, there are numerous block proposers within the infrastructure, and this makes it more difficult to ensure that invocations to the `submitSolution()` function are executed rapidly and in the expected order.

To simplify the use of the framework by users, the research team could develop a web application that allows users to register and create an account. Inside the account it would be possible for the user to submit new solutions and see which solutions they have submitted, at what time, and what is the value of the corresponding cost function. For greater transparency, the account may also contain direct links to the blockchain transactions corresponding to the submitted solutions. These links may be used with any blockchain explorer to inspect the content of the transactions. The account could also contain contact information for the delivery of the rewards, and the possibility to revoke the credentials used to call the smart contract functions, and/or request the generation of new credentials, if the user believes their credentials have been compromised. The web application could adopt the typical architecture of decentralized applications, which consists of the following elements:

- The user interface (frontend), that communicates with the backend via REST APIs. It could be implemented using, for example, the React library[4].
- The backend, that contains the operating logic, manages the information stored in the centralized database, exposes the REST APIs to the frontend, and manages access to the blockchain and its smart contracts. It could be implemented using the Sails Javascript MVC framework[5]. Communication between the APIs and the blockchain could occur through the Web3.js Ethereum Javascript library[6].

[2] https://polygon.technology/
[3] https://www.flashbots.net/
[4] https://react.dev/
[5] https://sailsjs.com/
[6] https://web3js.org/

- A centralized database, such as MongoDB[7], in which all information that should not be recorded on the blockchain—including the above mentioned account information—is stored.
- A permissioned version of the Ethereum blockchain, such as Quorum[8].
- A public blockchain (such as Ethereum or Polygon, on which the hash fingerprint of the last current block of the permissioned blockchain will be notarized at regular time intervals.

We conclude this section by noting that a single permissioned blockchain can handle many different optimization problems, each one associated with the corresponding smart contract. It is therefore not necessary to build a different blockchain for each problem to be solved.

3. Two Examples

To give a more precise idea of how our proposed framework works, we consider two NP-hard optimization problems which are representative of a large class of real-world optimization problems: KNAPSACK, and MAX-SAT. For each of these problems we illustrate how instances and solutions can be represented within the smart contract, and how the validity of the proposed solutions can be verified.

The complete source code of the two smart contracts can be found on https://github.com/alepo42/CertifiedSolutions/. Both contracts have been implemented and tested using the Ethereum Remix IDE[9]; precisely, using the Remix Desktop application version 1.0.8-insiders for MacOS Sequoia 15.4.1, running on a MacBook Air M2 laptop with 16 GB of RAM. The Solidity compiler used is version 0.8.26.

In the KNAPSACK optimization problem, there are n items – each with its own value v_i and weight w_i – and a knapsack of maximum weight capacity W. Each item can be entirely put into the knapsack, or left out (indeed, this variant is sometimes called the 0-1 KNAPSACK problem). The goal is to choose the items to put in the knapsack, in such a way as to maximize their total value, without exceeding the capacity of the knapsack itself. The formal definition of the problem is therefore the following:

- Name: KNAPSACK
- Instance: a set of n items numbered from 1 up to n, each with a weight w_i and a value v_i, along with a maximum weight capacity W.
- Admissible solution: a subset of the n items such that $\sum_{i=1}^{n} w_i x_i \leq W$ and $x_i \in \{0, 1\}$.
- Goal: maximize $\sum_{i=1}^{n} v_i x_i$.

Here x_i represents the number of copies (either 0 or 1) of item i to include in the knapsack. A candidate solution can then be represented either as the list of indices (a subset of $\{1, 2, \ldots, n\}$) of the items to be inserted into the knapsack, or as the binary vector $(x_1, x_2, \ldots, x_n)$. In this paper, we adopt the former representation. In any case, the cost function to be maximized is $\sum_{i=1}^{n} v_i x_i$, subject to the constraint $\sum_{i=1}^{n} w_i x_i \leq W$.

The weight and value of each instance item are stored in a `struct Item`. The `items` array thus contains the weights and values of all the instance items. Knapsack capacity W, instead, is stored in the integer `maxWeight` variable, as shown in the following code snippet.

```
1  // Structure to represent an object
2  struct Item {
3    uint256 weight;
4    uint256 value;
5  }
6
7  // Array of available items
8  Item[] public items;
9
10 // Maximum knapsack capacity
11 uint256 public maxWeight;
```

[7] https://www.mongodb.com/
[8] https://github.com/ConsenSys/quorum
[9] https://remix.ethereum.org

The weights w_i and values v_i of the instance items, and the maximum capacity of the backpack W, are specified by the owner of the smart contract when it is created. The value of W is simply indicated as an integer, while the weights and values of the elements are specified as a tuple. For example, the tuple $[[10, 60], [20, 100], [30, 120]]$ indicates that the instance contains three elements, having weights $w_1 = 10$, $w_2 = 20$, $w_3 = 30$ and values $v_1 = 60$, $v_2 = 100$, and $v_3 = 120$. All these values are stored in the above mentioned smart contract variables by the `constructor()` function, which also sets the owner's address and the Boolean flag `submissionOpen` to `true`, indicating that candidate solutions can be accepted.

The solutions are stored in the smart contract state in the following format:

```
// Structure to represent a solution
struct BestSolution {
  uint256[] solutionItems;
  uint256 solutionValue;
  uint256 solutionWeight;
  uint256 timestamp;
  address senderAddress;
}

// History of best solutions
BestSolution[] public historyOfBestSolutions;
```

where `solutionItems` is an array containing the indices (a subset of $\{1, 2, \ldots, n\}$) of the items that compose the solution, `solutionValue` is the sum of the values of these items, `solutionWeight` is their total weight, `timestamp` is the block timestamp indicating the moment of time in which the solution has been submitted, and `senderAddress` is the address of the user who proposed the solution. All submitted solutions are stored in the `historyOfBestSolutions` array.

The `checkValidity()` function can be used to verify that a proposed solution is valid, i.e, that it satisfies the following conditions: (1) all indices of the proposed subset of $\{1, 2, ..., n\}$ are different, and (2) the total weight of the items does not exceed knapsack capacity. If the presented solution is valid, the function returns the triple `(true, totalValue, totalWeight)`, where `true` indicates validity, and `totalValue` and `totalWeight` are the sum of values and weights of the items, respectively. On the other hand, if the presented solution is not valid, the triple `(false, 0, 0)` is returned.

```
// Function to check the validity of the proposed solution
function checkValidity(uint256[] memory _solution) public view returns (bool, uint256, uint256) {
  uint256 totalWeight = 0;
  uint256 totalValue = 0;

  // All the indices of the proposed solution must be different
  // Otherwise, return false
  for (uint256 i = 0; i < _solution.length; i++) {
    for (uint256 j = i+1; j < _solution.length; j++) {
      if (_solution[i] == _solution[j]) {
        return (false, 0, 0);
      }
    }
  }
  // Computes the total weight and value of the proposed solution
  for (uint256 i = 0; i < _solution.length; i++) {
    // If one of the indices is not valid, return false
    if (_solution[i] >= items.length) {
      return (false, 0, 0);
    }
    totalWeight += items[_solution[i]].weight;
    totalValue += items[_solution[i]].value;
  }
  // If the total weight exceeds maximum capacity, return false
  if (totalWeight > maxWeight) {
    return (false, 0, 0);
  }
```

```
28    // At this point, we have a valid solution, and we return its total value
29    // and weight
30    return (true, totalValue, totalWeight);
31  }
```

The `submitSolution()` function is, together with the above `checkValidity()` function, the core of the smart contract, since it allows users to submit their proposals for solving the optimization problem. It takes as input a proposed solution, as usual in the form of a subset of $\{1, 2, \ldots, n\}$, and returns `true` if it is valid and better than the solutions stored so far; otherwise, it returns `false`. Due to space limitations, we do not report the code of the function here, which however implements the logic described in the pseudo code of Algorithm 1.

As another example of optimization problem we consider Max-SAT, in which an assignment to the Boolean variables must be found that maximizes the number of satisfied clauses. Formally, the Max-SAT problem is defined as follows:

- Name: Max-SAT
- Instance: a Boolean formula ϕ, built over a set of n Boolean variables $\{x_1, x_2, \ldots, x_n\}$, written in conjuntive normal form. That is, $\phi = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, where each $C_i = (X_{i_1} \vee X_{i_2} \vee \ldots \vee X_{i_{k_i}})$ is a clause and each X_{i_j} is a literal, i.e., a variable or the negation of a variable. Without loss of generality, we can assume that the number of literals in each clause is between 1 and n, since no clause should contain repetitions of the same variable, or a variable and its negation.
- Admissible solution: an assignment to the Boolean variables $x_1, x_2, \ldots, x_n$.
- Goal: maximize the number of clauses of ϕ that are satisfied, that is, that are made true by the assignment.

To store an instance, this time we simply list all the clauses of the formula, where each clause is a list of literals. Each literal is represented as a positive integer number in the range $[1, \ldots, n]$ if it is a variable, or as a negative number in the range $[-n, \ldots, -1]$ if it is a negated variable. So, for example, the Boolean formula $\phi = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (x_2 \vee \neg x_3)$ is represented as $[[1, 2], [-1, 3], [2, -3]]$. The integer variable `numberOfVariables` contains the number n of variables upon which the instance formula is defined, and is computed by the `constructor()` function while storing the list of clauses in the `formula` array.

The solutions are stored in an array whose elements are made up of a dedicated `struct`, that contains an assignment (represented as a list of 0 and 1), the number of clauses satisfied by such an assignment, the block timestamp, and the address of the user who proposed the solution. The `checkValidity()` function is used to verify that the proposed assignment is valid, i.e, it is a list containing n elements equal to 0 or 1. Also in this case, the `submitSolution()` function strictly follows the logic illustrated in Algorithm 1. Finally, the helper functions are exactly the same as the ones that compose the smart contract for the Knapsack problem.

Due to space limitations, we cannot enter into further details. We refer the interested reader to the above mentioned GitHub repository, where the source code of the smart contracts can be found.

4. Some Related Works

By conducting a literature search, we discovered that we are not the first to have had the idea of using a smart contract to store the solution of computationally difficult problems. In [2] (an unpublished short note) the authors propose a framework, named Crick, for incentivizing and verifying work on NP-complete problems in a fully decentralized manner. The main idea is to issue a token as a reward for the submission of valid improvements to existing solutions and updating the store of solutions accordingly. The framework uses the decentralized storage network Swarm[10], both to store the (publicly available) datasets related to the problems and the database of proposed solutions. In the Crick framework, each time a better solution is proposed than those found so far, a reward token is emitted. This makes

[10] http://swarm-gateways.net/bzz:/theswarm.eth/

it possible to perform the so-called Bubka attack[11], whereby a user could submit increasingly better solutions, specifically to maximize the number of reward tokens obtained. This attack is not possible in our framework, as a reward will be given only to the one user who has proposed the best solution when the submission period has ended.

The work just cited is the only one that comes close to the proposal made in this paper. Other works involve solving computationally hard problems by incorporating them into the work done by miners. So, for example, Primecoin[12] proposes a new type of PoW consensus protocol based on searching for prime numbers, Curecoin[13] asks clients to perform protein folding tasks, and Coinami [3] proposes a modification to PoW that includes DNA alignment problems. On a similar line [4] and [5] propose variants of the PoW consensus algorithm that reward blockchain miners for using computational resources to solve NP-complete puzzles and optimization problems of scientific interest. Finally, in [6] the authors present Hybrid Mining, a mining protocol that combines solving real-world instances of NP-complete problems with Hashcash mining. Any node in the P2P network can submit a problem, together with a reward for its solution; then, every other node of the network can add a new block to the blockchain by either solving a submitted problem or taking part in standard Hashcash PoW.

5. Conclusions and Directions for Future Work

We proposed a blockchain-based framework for the certification of solutions to computationally hard problems. Focusing in particular on smart contracts, we showed how to represent the instances and solutions of two NP-hard optimization problems: Knapsack and Max-SAT.

A clear direction for future work is to perform an extensive evaluation about the scalability of the proposed framework, in terms of gas used, when increasingly large instances of NP-hard optimization problems are considered. Moreover, a difficult and important issue to face is that of stealing solutions, and executing calls to the `submitSolution()` function in the expected order. Another direction for future work is to implement the entire web application, and test it with a set of users. It would also be interesting to write software that helps research teams design smart contracts associated with optimization problems. Such software would specify problem instances using a mathematical formalism, and output the Solidity code of the smart contract.

Declaration on Generative AI

The author did not use any Generative AI tools.

References

[1] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, A. Juels, Flash Boys 2.0: Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges, 2019. URL: https://arxiv.org/abs/1904.05234.

[2] C. Oliver, A. Ricottone, P. Philippopoulos, Proposal for NP-Complete Problem Refinement Smart Contract, https://pphili.github.io/misc/crick.html, 2017. Accessed: 2025-04-18.

[3] A. M. Ileri, H. I. Ozercan, A. Gundogdu, A. K. Senol, M. Y. Ozkaya, C. Alkan, Coinami: A Cryptocurrency with DNA Sequence Alignment as Proof-of-work, 2016. URL: https://arxiv.org/abs/1602.03031.

[4] C. G. Oliver, A. Ricottone, P. Philippopoulos, Proposal for a fully decentralized blockchain and proof-of-work algorithm for solving NP-complete problems, 2017. URL: https://arxiv.org/abs/1708.09419.

[5] P. Philippopoulos, A. Ricottone, C. G. Oliver, Difficulty Scaling in Proof of Work for Decentralized Problem Solving, Ledger 5 (2020). doi:10.5195/ledger.2020.194.

[11] This attack is named after Ukrainian pole vaulter Sergey Bubka, who repeatedly broke his own records by small margins to claim a new prize each time.

[12] https://primecoin.io/

[13] https://curecoin.net/

[6] K. Chatterjee, A. K. Goharshady, A. Pourdamghani, Hybrid mining: exploiting blockchain's computational power for distributed problem solving, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC '19, ACM, New York, NY, USA, 2019, p. 374–381. doi:10.1145/3297280.3297319.