Ofelimos: Combinatorial Optimization via Proof-of-Useful-Work
A Provably Secure Blockchain Protocol

Matthias Fitzi
IOHK
matthias.fitzi@iohk.io

Aggelos Kiayias
University of Edinburgh & IOHK
akiayias@inf.ed.ac.uk

Giorgos Panagiotakos
IOHK
giorgos.panagiotakos@iohk.io

Alexander Russell
University of Connecticut & IOHK
acr@cse.uconn.edu

October 14, 2021

Abstract

Minimizing the energy cost and carbon footprint of the Bitcoin blockchain and related protocols is one of the most widely identified open questions in the cryptocurrency space. Substituting the proof-of-work (PoW) primitive in Nakamoto's longest chain protocol with a *proof of useful work* (PoUW) has been long theorized as an ideal solution in many respects but, to this day, the concept still lacks a convincingly secure realization.

In this work we put forth *Ofelimos*, a novel PoUW-based blockchain protocol whose consensus mechanism simultaneously realizes a decentralized optimization-problem solver. Our protocol is built around a novel local search algorithm, which we call Doubly Parallel Local Search (DPLS), that is especially crafted to suit implementation as the PoUW component of our blockchain protocol. We provide a thorough security analysis of our protocol and additionally present metrics that reflect the usefulness of the system. As an illustrative example we show how DPLS can implement a variant of WalkSAT and experimentally demonstrate its competitiveness with respect to a vanilla WalkSAT implementation. In this way, our work paves the way for safely using blockchain systems as generic optimization engines for a variety of hard optimization problems for which a publicly verifiable solution is desired.

Contents

1 Introduction

Blockchain protocols based on Proof of Work (PoW) capitalize on the work performed by protocol participants, called miners, to ensure the security of the maintained transaction ledger. In the most prominent blockchain designs following Bitcoin's paradigm [46] the work performed serves no other purpose besides maintaining security. Such protocols are also permissionless and incentive-driven and hence offer rewards to prospective miners who decide to join the protocol and commit computational effort. This has lead to an ever increasing energy expenditure in systems like Bitcoin. At the time of this writing, Bitcoin has an annualized energy expenditure on par with many small to medium countries (see e.g., the Cambridge Bitcoin Electricity Consumption Index, `https://cbeci.org`).

The above trend has been identified early on as an important aspect in the Bitcoin ecosystem leading to two major avenues for potential improvement of the underlying blockchain protocol. The first is aimed at changing the PoW mechanism to another resource that would have potentially "greener" characteristics, e.g., proof of stake [17, 29, 37], proof of space [21, 50], proof of space-time [44], and similar mechanisms. A common concern to these approaches however, is the change of the underlying security primitive (from "work" to something else) and the inevitable impact of this change to the system's security guarantees. The second direction which ameliorates this issue, and is the focus of this work, is to repurpose the invested computational effort towards solving real-world problems. This direction thus highlights a proof-of-*useful*-work (PoUW) design approach for blockchain protocols.

Early designs and implementation attempts such as Noocoin [16] and Primecoin [38] highlighted the fundamental issue that would henceforth plague progress towards a robust PoUW system. If the work solved is truly useful then the attackers may direct the system towards solving problem instances that are easy for them (e.g., due to precomputation or other private advantage due to the underlying instance-space structure) and hence the security guarantees are dubious; at the same time, minimizing the attacker's ability to manipulate the system may render the system's computations useless in practice (e.g., Primecoin [38] and Gapcoin [23] compute sequences of Cunningham primes and gaps between primes respectively—both mathematical objects of dubious usefulness).

Our contributions. We propose the first PoUW-based blockchain protocol that is accompanied by a thorough security and usefulness analysis. Central to our construction is a novel general-purpose algorithm for *stochastic local search* called Doubly Parallel Local Search (DPLS). Our key technique for protocol design is to mold the whole blockchain protocol execution into a DPLS engine that demonstrably performs the steps of the algorithm in a publicly verifiable manner. The PoUW operation in our consensus protocol has the miners collectively run DPLS on instances contributed by interested clients. In more detail our results are the following.

(I) Doubly Parallel Local Search. We put forth a new stochastic algorithm for local search. With DPLS we achieve the following two-pronged objective: (i) The algorithm suitably reflects in its structure the stochastic properties of the underlying permissionless blockchain operation so that it will be feasible to view the blockchain execution as a virtual machine running the algorithm; (ii) Stochastic local search is a powerful and generic algorithmic paradigm for solving computationally hard optimization problems. Thus DPLS can be casted within the broad family of stochastic local search algorithm variants and have its usefulness assessed with respect to problems of high real-world value.

DPLS is a general purpose stochastic local-search algorithm that is based on an underlying algorithm M, called the exploration algorithm, which examines a given set of points in the solution space and produces another one, after some local exploration that requires a modicum of computational effort. Based on M, DPLS follows a "doubly" parallel search strategy where a number of paths are pursued in parallel, and in each path a number of exploration threads via M are executed of which the best one, according to a scoring function, is selected.

(II) Moderately hard DAG computations. To contemplate the possibility of using a DPLS solver within a proof-of-work setting, it is essential to be able to express the hardness conditions under which running the basic exploration sub-problem exhibits *moderate hardness* (MH). This property is the necessary requirement for a computational problem to be applicable in the blockchain setting. What makes the modeling more challenging compared to, say, the case of Bitcoin's PoW algorithm is that we cannot resort to an idealized model (such as the Random Oracle Model) and we have to express the moderate hardness property in a way that can be suitably utilized in the security arguments of the blockchain protocol.

To capture this and at the same time reflect the parallelizable nature of DPLS, we focus on the DAG-computation abstraction which has been widely used in the modeling of parallel computations (e.g., see [49, 2]). In the setting of DPLS the main computational unit is the exploration step M and we are interested to express the MH of arbitrary DAG computations over M. The delicate part of this modeling is to express the advantage $\hat{\epsilon}$ of the adversary over the honest parties as a function of its ability to "grind" the randomness of the DAG computation as well as capitalize on any advantage obtained from observing previously published steps in the computation.

(III) The PoUW-based blockchain protocol. At a high level, our protocol operates by having parties post instances for problems of interest in the ledger, while locking funds denominated in the ledger's native token to incentivize miners to work towards solving them. Maintaining the blockchain translates to performing steps of the DPLS algorithm for the instances in the ledger and being rewarded for that—with foresight, we stress that solving such instances directly (or posting pre-solved instances) will not help in extending the ledger. Problem setters can keep funding a particular DPLS computation whenever its funds are getting depleted.

The cornerstone of our protocol is its PoUW process that operates in three stages. In the "pre-hash" stage a random string is generated via hashing and testing whether a small hash value has been attained as in a standard PoW. This string will constitute the random seed for the DPLS exploration step M and will be essential for controlling grinding attacks. When the exploration stage terminates, a "post-hash" step determines with a single hash query whether the resulting value qualifies as a PoUW. This "sandwiching" of M between two small hashes is essential for security since it forces an adversarial miner to seed the computation with a randomly selected seed and learn that a successful block can be issued only after the exploration step M is complete. However, if we apply this idea naively, there are two major disadvantages: first, a number of useful exploration steps will go to waste, since they won't lead to a block, and second, adjusting the hardness of block production (which is needed for blockchain security) would impact usefulness (since miners will spend too much effort trying to find such small hashes).

We resolve these issues by two mechanisms. First, taking advantage of the scoring function g, we have the miners publish the best value they have produced based on all their post-hash attempts; in this way, progress in the DPLS computation is not lost. Note that deviating from this strategy may only impact usefulness—the security of the protocol is maintained against any Byzantine deviation. Second, by adapting the 2-for-1 PoW mechanism of [24], which allows to produce two types of blocks with a single hash attempt: either an "input-block", in which case it is inserted in the blockchain as a transaction, or a "ranking-block" which extends the blockchain and contains any number of input blocks. Using this decoupling mechanism, we can keep the steady progress of the DPLS computation and adjust the underlying (ranking) block hardness independently. The crucial emerging property here is that, as more miners join the protocol, the DPLS computation will be sped up proportionally, while the ranking block production can be maintained steady as required for the security of the underlying blockchain protocol. In this way, the more real-world useful problem instances are submitted to the system (as evidenced by the increased funding locked with each one and the platform's native token appreciation), the more computational power will be introduced to the DPLS engine to solve them.

We prove our protocol secure under a standard "honest majority" type of assumption reminiscent of the Bitcoin protocol analysis, where the distance from $1/2$ depends, among other parameters, also to the MH advantage $\hat{\epsilon}$ (we note that even if $\hat{\epsilon} = 1$, i.e., MH entirely collapses, our protocol remains secure with a bound close to $1/4$).

(IV) Usefulness metrics. In a nutshell, our blockchain protocol can be thought as a decentralized DPLS solver. This suggests the following two complementary metrics to measure its effective usefulness. The first one asks how good is the blockchain execution as a DPLS engine. This can be done by measuring the ratio per unit of time of the number of steps that the blockchain protocol spends in DPLS computations compared to its total number of steps. We call this metric U_{eng}, as it can be thought as the efficiency of the blockchain protocol as an "engine" that runs DPLS. The second metric asks how useful are DPLS computations themselves and we denote it by U_{alg}. For a given instance distribution we define this metric as the ratio between the expected number of steps of the best algorithm for that instance distribution divided by the expected number of steps that DPLS takes. Note that identifying the best algorithm for a problem could be infeasible based on current state of the art, so in this case the best algorithm can be simply substituted with the best *known* algorithm for the problem at hand. Combining the above two metrics, we can obtain, as an overall metric of usefulness, the product $\mathsf{U}_{\text{eng}} \cdot \mathsf{U}_{\text{alg}}$.

Given the above formalism we observe that for our protocol it holds that (i) $\mathsf{U}_{\text{eng}} \leq 1/2$, which stems from the fact that we balance the pre-hash probability of success to require the same effort as the worst-case time complexity of M—this enables us to prove security for any advantage $\hat{\epsilon}$ in the underlying MH assumption. (ii) U_{eng} will be close to $1/2$, if the advantage $\hat{\epsilon}$ shows little sensitivity to increased grinding. We note that the $1/2$ bound can be surpassed by taking into account the sensitivity of $\hat{\epsilon}$ and adaptively setting the pre-hash difficulty, however such a direction would be only feasible if we restrict the class of exploration algorithms M to those whose hardness is well understood. Estimating U_{alg} requires some real-world baseline. We explore this by implementing within the DPLS engine WalkSAT [55, 35], a popular local search algorithm for satisfiability problems, and compare how does the DPLS implementation fare with respect to running WalkSAT in isolation. The instance distribution is also an important consideration; for illustrative purposes, we focus on *Blocks World Planning*, a well known NP-hard problem in AI [32] for which there is an abundance of public data sets. Using WalkSAT as the baseline, we show that a single-thread implementation of DPLS performs reasonably well against WalkSAT, investing about twice as much computational steps, i.e., something that amounts to an estimation of $\mathsf{U}_{\text{alg}} \approx 1/2$. Similar results are obtained from additional experiments that reflect adversarial deviations and the effect of parallelization.

The above results are evidence for the non-negligible real-world usefulness of our PoUW-based blockchain protocol. We anticipate that investigating further the DPLS blockchain engine as an optimization solver will be an exciting research direction from an algorithmic perspective. There is yet another beneficial dimension of using our blockchain protocol as a DPLS solver: optimization is executed collaboratively in a publicly verifiable manner. Depending on the task, public verifiability has intrinsic usefulness and this can be seen as the price the system pays for the remaining ratio $1 - \mathsf{U}_{\text{eng}} \cdot \mathsf{U}_{\text{alg}}$. For instance, optimization tasks such as athletic-competition tournament scheduling or various matching problems (e.g., the allocation of residents to hospitals or radio frequency auctions) can benefit from public verifiability; see Section 6 for further discussion and references.

Related work. Beyond the early work mentioned in PoUW coins and designs [16, 38, 23], a number of other works investigated the concept. One line of work considered hybrid constructions where the miner can choose between applying either standard PoW *or* doing some potentially useful computation [48, 12, 60]. Further constructions for PoUW mining were given by Loe et al. [41], and Dotan et al. [20], and, closer to our work, Baldominos et al. [8], and Lihu et al. [39], suggested to base PoUW on stochastic search and machine-learning problems. In all these previous approaches the security of the system was not rigorously analyzed and in many cases concrete attacks by e.g., an adversary who directly plants easy instances to solve, are feasible.

In contrast to the above, a formal security approach was taken in [9] but the published version of the work retracted the "usefulness" dimension of the original paper. Also, their proof-of-work construction is not suited for permissionless ledgers as it does not introduce any variance in puzzle-completion time.

Finally, some alternative approaches to the problem at hand that are worth mentioning in our context are the concept of "merged mining", a technique employed in a number of cryptocurrencies where the mining effort for the blockchain has a dual use as mining Bitcoin and hence it is useful in this sense; Permacoin [43] where, via proofs of retrievability, the usefulness dimension is in maintaining a public file store; and useful work enforced via a trusted execution environment [59] where, in contrast to the the above solutions, full trust in a specific hardware manufacturer is required.

We note that, to the best of our knowledge, no prior fully decentralized, PoUW-based blockchain protocol has been published along with a thorough security (or usefulness) analysis.

Organization of the paper. In Section 2 we describe the computational model and some basic notation. DPLS and our notion of moderately hard computation are presented in Section 3. In Section 4 we present our blockchain protocol, whose security and usefulness we analyze in Section 5. Applications and experimental results are given in Section 6, while some of the code is presented in the Appendix.

2 Preliminaries

Notation. For $k \in \mathbb{N}^+$, $[k]$ denotes the set $\{1, \ldots, k\}$. We denote sequences by $(a_i)_{i \in I}$, where I is a countable index set. For a set X, $x \leftarrow X$ denotes sampling an element from X uniformly at random. For a distribution $\mathcal{U}$ over a set X, $x \leftarrow \mathcal{U}$ denotes sampling an element of X according to $\mathcal{U}$. By $\mathcal{U}_m$ we denote the uniform distribution over $\{0, 1\}^m$. We denote that some function f is negligible in λ by $f(\lambda) < negl(\lambda)$. We let λ denote the security parameter.

Security model. We adopt the computational model of [25], which is a variant of the model presented in [24]. There, the set of parties $\{P_1, \ldots, P_n\}$ running the protocol is fixed and the parties, the environment $\mathcal{Z}$, the adversary $\mathcal{A}$, and the control program $\mathcal{C}$ coordinating the execution are all modeled as IRAMs. The adversary $\mathcal{A}$ is active and can corrupt up to t parties in order to break security.

Communication model. We follow the communication model used by most previous works [51, 7] that analyze blockchain protocols in the cryptographic setting, where time is discrete and the network is (partially) synchronous. In more detail, the protocol advances in rounds and communication happens through a diffusion functionality. Honest parties can use it to send messages which may be adaptively delayed for up to Δ rounds by the adversary, but are guaranteed to be received by everyone in the network. Communication is not authenticated, in the sense that the functionality does not provide any guarantees regarding the origin of sent messages. Finally, the adversary is rushing and can additionally choose to send its own messages only to a subset of the parties.

Setup. All parties have access to a *common reference string* (CRS), sampled from a known efficiently samplable distribution, which is used to instantiate a succinct non-interactive argument (SNARG) system [30] SNARG = $(\mathsf{S}, \mathsf{P}, \mathsf{V})$. Note that there are several ways to securely establish a CRS for a SNARG in a permissionless blockchain environment. In particular, assuming the slightly stronger notion of an updatable *structured reference string* (SRS) [31, 42], the construction of [36] allows to obtain a common reference string.

Random Oracle. Parties have access to a random-oracle (RO) functionality [10]. We use both RO and non-RO based moderately hard problems and, in order to argue about security, we need to be able to compare their computational costs. We thus assume that a query to RO takes c_H computational steps both for the honest parties and the adversary.

Concrete modeling. $\mathcal{A}$ and $\mathcal{Z}$ have a concrete bound of $t \cdot c_H$ steps they can take per round as well as an upper bound θ on the number of messages they can send per round.

3 Doubly Parallel Local Search

One way to design a PoUW blockchain for optimization problems is to: (i) first pick your favorite optimization algorithm, and then (ii) try to design a blockchain protocol around it. The disadvantage of such an approach is that any change in the target optimization problem may result in vital changes to the blockchain and consensus system, requiring new security proofs. Here, instead, we adopt a modular approach where we first build a PoUW blockchain based on a generic optimization algorithm, and later, with minimal overhead, instantiate it with the problem-specific parameters. This allows for re-using our blockchain analysis for different instantiations of the optimization algorithm.

We start, in Section 3.1, by giving a high level overview of DPLS, the generic optimization algorithm that our blockchain protocol is implementing from a client's point of view, i.e., ignoring the internal details of the blockchain algorithm. In Section 3.2, we then expand on the notion of moderate hardness of useful computation, on which the security of our blockchain protocol is based.

3.1 Algorithm description

DPLS overview. Clients of our protocol publish on the blockchain the optimization problems that they want miners to solve. Miners, on the other hand, run the Doubly Parallel Local Search (DPLS) algorithm, which we introduce next, to solve these problems.

We first note that solving large optimization problems may require more work than what can be computed by a node during the mining of a single block. Thus, we chose DPLS to be a *distributed algorithm* where the computation result is obtained by multiple state updates, some of them possibly occurring concurrently. Concurrent updates is the first source of parallelism of our doubly parallel algorithm.

In its core, DPLS searches the solution space X by repeatedly exploring the neighborhood of a currently selected location/point, looking for a neighboring point that promises progress towards an optimal solution. More concretely, based on the description of a problem instance Λ, DPLS gradually builds a DAG G recording the already explored locations in X. A single exploration step then consists of invoking a generic exploration algorithm M on G, yielding a new location in X, with the goal of extending G by a node representing a new location of better quality (computed by a scoring algorithm g_Λ), thereby progressing the exploration.

Note that, in a strictly sequential execution, a 'linear' graph G may be sufficient. However, maintaining a DAG of explored locations allows for more general flavors of local search where multiple threads are concurrently explored by different parties. Such an execution is presented in Figure 1.

As the search algorithm is distributed, in an attempt to minimize communication and local precomputation, we cannot afford to publish every micro-update. For this reason, each party computes a large number of local exploration steps in batches, publishing only the best exploration result from the batch. To this end, the exploration algorithm M is parametrized by an *inner state* z that determines the common state of the execution batch, e.g., a common starting location in G to focus the batched search. Batched search is the second source of parallelism of our doubly parallel algorithm.

Given the above, DPLS is parametrized by the following sub-algorithms:

- *Initialization* algorithm $\mathsf{Init}(\Lambda)$: A probabilistic algorithm taking as input an instance description Λ and outputs a DAG G.
- *Focus* algorithm $\mathsf{F}(\Lambda, G)$: A probabilistic algorithm taking as input Λ, G and outputs an inner-state string z.
- *Exploration* algorithm $M_\Lambda(G, z, r)$: A deterministic algorithm taking as input a DAG G, an inner state z, and a seed r, and outputs a point $x \in X$.
- *Scoring* algorithm $g_\Lambda(x)$: A deterministic algorithm taking as input Λ and $x \in X$, and outputs the score $y \in \mathbb{R}$ of x.
- *Termination* algorithm $\mathsf{Finished}(\Lambda, G)$: A deterministic algorithm taking as input Λ, G and outputs 1 if the algorithm has finished, and 0 otherwise.

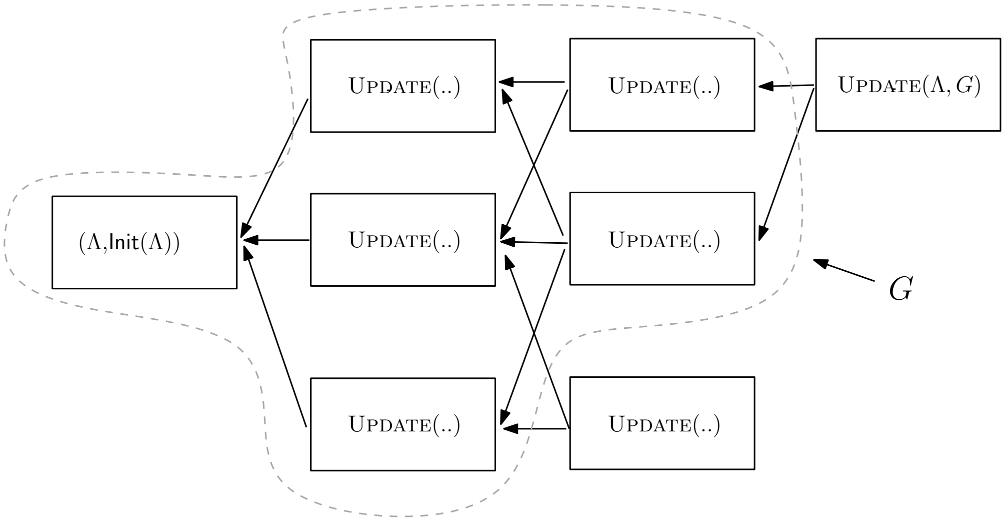


Figure 1: An execution of the DPLS algorithm. Due to desynchronization state updates may be computed on a partial view of the execution (view G).

DPLS modeled in a blockchain setting. Problem solving starts by the problem setter posting an instance description Λ together with the output of $\mathsf{Init}(\Lambda)$ in the blockchain, in the form of a special transaction.[1] Miners work on such an instance by running the UPDATE procedure (Algorithm 1),

[1] To further avoid adversarial manipulation, we can have parties run the initialization function themselves using coins generated by hashing the block that includes the posted problem instance.

which makes use of the sub-algorithms introduced above. The outputs produced are posted to the blockchain and are in turn used by other parties to produce additional updates. The search algorithm ends when predicate $\mathsf{Finished}(\Lambda, G)$ is equal to 1.

UPDATE takes as inputs the chosen instance description Λ and the party's current view of the DAG G. The inner state z, passed as a parameter in UPDATE, is generated using algorithm $\mathsf{F}(\Lambda, G)$, while the number k of different invocations of M is distributed according to the geometric distribution, with the exact parameters of the distribution set by the protocol designer. The sampling of k from the geometric distribution models its integration into the useful-work mining procedure where each computation of M qualifies for block production with probability p_2—to publish a state update, the miner must find a block. After k is fixed, that many seeds $(r_i)_{i\in[k]}$ are sampled at random, and algorithm $M(G, z, r_i)$ is invoked k times, with the best-scoring result (according to function g) being output by UPDATE.

Algorithm 1 The state update procedure.

```
function UPDATE($\Lambda, G$)
    $z \leftarrow \mathsf{F}(\Lambda, G)$                                          ▷ Compute the inner state
    $k \leftarrow Geom(p_2)$                                          ▷ Sample from geometric
    $(r_i)_{i\in[k]} \leftarrow \mathcal{U}_m^k$                                          ▷ Sample uniformly
    $S := \{(z, r_i, x_i) | x_i := M(G, z, r_i), i \in [k]\}$                  ▷ Invoke $M$
    $(z, r, x) := \arg\max_{(z,r,x)\in S} g(x)$                          ▷ Pick best
    return $(z, r, x)$
```

An example. We present a DPLS variant of the classical WalkSAT algorithm [55, 35] for the SAT problem. First, we describe the original WalkSAT algorithm. Starting from some initial configuration, at each step, WalkSAT picks a variable to flip (Algorithm 2) as follows: Given the current configuration, one of the unsatisfied clauses is chosen at random. For each of the variables involved in the clause, a grade is computed which is equal to the number of clauses that are going to be broken (i.e, turn from satisfied to unsatisfied) if the chosen variable is flipped. If there exist variables that have grade 0, then one of them is selected at random and flipped. Otherwise, a variable is selected (and flipped) at random, with probability wp coming from the selected clause, and with probability $1 - wp$ coming from the variables with the best grade. The walk continues until a solution is found (Algorithm 3), or some other condition is met, e.g., an upper bound on the total number of flips is reached. If no solution is found, the algorithm can be restarted from some other point in the solution space.

In the DPLS variant, the instance description Λ encodes the description of the SAT instance, i.e., the number of variables and the different clauses, with the solution space X being equal to the possible configurations of the SAT variables. To take advantage of the first level of parallelization, $\mathsf{Init}(\Lambda)$ outputs a number of different initial configurations in X; in each invocation of UPDATE, miners pick at random which location/configuration in G to work on and encode this information in z. Given this configuration, exploration algorithm $M(G, z, r)$ amounts to running WalkSAT for a fixed number of flips. Note, that the starting configuration is the same for the different runs of M in a single UPDATE invocation, allowing miners to focus their search. On the other hand, the randomness used by the different WalkSAT invocations come from the respective seeds (r), leading to the exploration of different points in the solution space. To choose the best among these points, g counts the number of satisfied clauses in the respective ending configurations that are at max depth in the DAG. Hence, UPDATE outputs the configuration that maximizes g, which is then possibly going to be used by another miner as the starting point of another run of UPDATE. The algorithm terminates after a predefined number of updates have been posted. We point the reader to Section 6 for the experimental evaluation of the performance of this algorithm.

Algorithm 2 The variable selection function of WalkSAT. It is parametrized by probability wp, the set of clauses C, and the grade function $grade_{A,C}(x)$, which counts the number of clauses in C that will be broken if variable x is flipped in configuration A.

```
function PickVariable($A, C$)
    $c \leftarrow \{c|c$ is an unsatisfied clause in $C$ $\}$
    $s := \min\{grade_{A,C}(x)|x \in \mathrm{Var}(c)\}$            ▷ Var outputs the variables in $c$.
    if s = 0 then
        $V := \{x \in \mathrm{Var}(c)|grade_{A,C}(x) = s\}$
    else
        with probability $wp$ do: $V := \mathrm{Var}(c)$
        otherwise do: $V := \{x \in \mathrm{Var}(c)|grade_{A,C}(x) = s\}$
    $x \leftarrow V$            ▷ $x$ is the variable selected to be flipped.
    return $x$
```

Algorithm 3 The WalkSAT algorithm. It is parameterized by configuration A, the set of clauses C, and a bound m on the number of trials. The randomness to be used by WalkSAT is passed in parameter r.

```
function WalkSAT($A, C, m; r$)
    $i := 0, f :=$ false
    while $((f =$ false$) \wedge (i < m))$ do
        $x \leftarrow$ PickVariable(A,C)
        $A := flip(A, x)$            ▷ Flip variable $x$ in $A$.
        $f := (A$ satisfies all clauses in $C)$
        $i = i + 1$
    if $f =$ false then return $\perp$
    else return $A$
```

Next, we give a detailed description of the DAG computation involved in our algorithm, as well as of the rest of the functions involved in the DPLS algorithm: Init, F, Finished (Algorithm 4). For the DAG computation, sets X, Z represent the set of possible configurations of the SAT instance, while $g(x)$ is equal to the number of clauses satisfied by configuration x. The Init function generates multiple initial configurations. Each of them is used as the starting point of a different walk. A party now picks one starting point A_z at random using the F function, and then M selects the best ending point that extends A_z, and is at maximum depth, to use as the starting point in the WalkSAT procedure invoked by M. The WalkSAT algorithm is only run for a bounded number of flips (parameter m). Following the schema presented in Section 3, the Update process invokes M multiple times, with only the best configuration among these invocations being output. The algorithm terminates when a configuration is found that satisfies all clauses.

3.2 Moderately-hard DAG computations

In DPLS, most of the work is spent running the exploration algorithm M. Hence it is natural to base security on the moderate hardness of this computation. We now describe in detail the syntax of M and its relevant security properties required for its use in a PoUW protocol.

As explained earlier, an important aspect is that state updates in DPLS are performed in a distributed way, and without much coordination. Moreover, the parameters of the computations performed will be possibly influenced by the adversary, in the sense that he may try to post a client

Algorithm 4 The Modified WalkSAT algorithms. Function $depth_G(v)$ outputs the depth of v on DAG G.

```
function Init($\Lambda, G$)
    $(A_i)_i \leftarrow X^l$                                   ▷ Sample $l$ points from solution space
    return $\{(i, \perp, A_i)\}_i$                            ▷ Return the DAG with the initial points

function F($\Lambda, G$)
    $z \leftarrow [l]$                                        ▷ Next point selected is going to extend $A_z$
    return $z$

function M($\Lambda, G, z, r$)
    $C, m := \Lambda$                                         ▷ Read the instance description
    $S := ...$
      $\{x' | \exists r' : \nexists r'', x'' : (z, r', x'), (z, r'', x'') \in V(G)...$
        $... \wedge\ depth_G(z, r', x') < depth_G(z, r'', x'')\}$
                                                              ▷ Select max depth points extending $A_z$
    $x := \arg\max_{x \in S} g(x)$                            ▷ Select best
    $A' \leftarrow \textsc{WalkSAT}(x, C, m; r)$              ▷ Run for $m$ flips
    return $A'$

function Finished($\Lambda, G$)
    $C, m := \Lambda$
    $f := (\exists (z, r, x) \in V(G) : x$ satisfies all clauses in $C)$
    return $f$
```

problem to be solved, only with the purpose of subverting the underlying blockchain protocol. As the security of the blockchain depends on the hardness of individual computations of M, we must guarantee that they remain moderately hard even when parameters are chosen maliciously.

Based on these restrictions, we adopt a DAG structure for computations of M, where each computation corresponds to a vertex on the DAG and depends on multiple previous vertices. Our notion can be seen a generalization of the iterated computation paradigm [11, 26], where each computation depends on a single vertex. New vertices are generated based on the current view of the DAG, an inner-state string, and, an unpredictable seed. As explained earlier, the inner state allows parties to focus their work in the context of DPLS, while the seed randomizes the computation to force the adversary to do work of average-case complexity—in contrast to possibly selecting "cheap" instances to gain an advantage in block production. Next, we formally introduce the notion of a DAG computation.

Definition 1. (DAG computation/transcript.) A DAG computation specifies a sequence of instance descriptions $\mathcal{I} = (\Lambda_\lambda)_\lambda$. For every value of the security parameter $\lambda \in \mathbb{N}$, an instance description Λ specifies:

1. a finite, non-empty set Z (inner state);
2. a finite, non-empty set X (output);
3. a relation R described below.

A *transcript* of a DAG computation Λ corresponds to a labeled DAG G where each vertex $u \in V(G)$ is labeled with a tuple $(z^{(u)}, r^{(u)}, x^{(u)}) \in Z \times \{0,1\}^\lambda \times X$ (edges have no labels). R relates transcripts to triplets in $Z \times \{0,1\}^\lambda \times X$. Given a transcript G, a vertex $u \in V(G)$ is *R-compliant*[2] if it either has no incoming edges or it holds that $(((z^{(u_i)}, r^{(u_i)}, x^{(u_i)}))_i, (z^{(u)}, r^{(u)}, x^{(u)})) \subseteq R$, where $\{u_i\}_i$ is the set of starting vertices of the incoming edges. We say that G is R-compliant if all of its vertices are. We write $\Lambda[Z, X, R]$ to indicate that Λ specifies Z, X, R as above.

R essentially describes how a set of vertices, corresponding to computations of M, can be extended by a new computation. By recursive application, R is sufficient to formalize the notion of an R-compliant transcript of a DAG computation. A DAG computation also provides two algorithms. For this purpose, we require that the instance descriptions, as well as the elements of the sets Z, X, can be uniquely encoded as bit strings of length polynomial in λ. All algorithms are parametrized by $\Lambda[Z, X, R]$:

- *Verification* algorithm $V_\Lambda(G)$: A deterministic algorithm taking as input a transcript G and outputs 1 if it is R-compliant and 0 otherwise; and
- *Exploration* algorithm $M_\Lambda(G, z, r)$: A deterministic algorithm taking as input a DAG G, an inner state z, and a seed r, and outputs a point $x \in X$.

For simplicity, we will omit writing Λ as a parameter of V, M when it is clear from the context. Moreover, we assume that M is correct, i.e., for $x \leftarrow M(G, z, r)$ where G is R-compliant, it holds that if we add a vertex u on G with label (z, r, x) that is connected to all other vertices, then the resulting transcript is R-compliant. We denote extending a labeled DAG G with a new node labeled with (z, r, x) and connected to all other nodes by $G \oplus (z, r, x)$, and merging two transcripts G, G' by $G \cup G'$.

Moderate hardness. Next, we introduce a moderate-hardness (MH) notion for DAG computations. Our notion builds on ideas found in [25, 26]. As we want to build a protocol that can accommodate solving multiple optimization problems, MH is expressed w.r.t. a family of DAG computations (per security parameter level), each corresponding to a different instantiation of the DPLS algorithm. While our protocol allows for exploration algorithms with different time complexities, for simplicity, we assume that they all have approximately the same *worst-case complexity*, i.e., we define $\hat{t}$ to be equal to $\max_{\Lambda, G, z, r}\{\mathsf{Steps}_{M_\Lambda}(G, z, r)\}$.

On a high level, we require that the time it takes to generate a given number of new vertexes in the DAG, is proportional to their number as well as $\hat{t}$, the worst case complexity of M. In more detail, in the security experiment, the adversary has access to three oracles $\mathcal{O}, \mathcal{M}, \mathcal{V}$. Its goal is to compute m new vertices for seeds generated at random from oracle $\mathcal{O}$ in less than $(1 - \hat{\epsilon}) \cdot m\hat{t}$ steps, where $\hat{\epsilon}$ reflects the advantage of the adversary compared to M. The adversary is allowed to query oracle $\mathcal{O}$ more than m times, and possibly use oracles $\mathcal{M}$ and $\mathcal{V}$ to simulate new honestly computed vertexes and verify whether a DAG computation is R-compliant, respectively. $\hat{\epsilon}$ is parameterized by the respective rates of queries $q_\mathcal{O}/m, q_\mathcal{M}/m, q_\mathcal{V}/m$ to reflect the possible adversarial advantage. We note, that oracles $\mathcal{M}$ and $\mathcal{V}$ are provided to aid composition;[3] Finally, we require that the property holds with overwhelming probability and for m greater than some parameter $\hat{k}$.

Definition 2. Let $\mathcal{I} = ((\Lambda_{\lambda,i})_i)_\lambda$ be a family of DAG computations. $\mathcal{I}$ is $(\hat{t}, \hat{\epsilon}, \hat{k})$-*Moderately Hard* (MH) if for any PPT RAM $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, $\lambda \in \mathbb{N}$, and all polynomially large $m \geq \hat{k}$, it holds that the adversary wins with probability $negl(\lambda)$ in $\mathsf{Exp}^{\mathrm{MH}}_{\mathcal{A},\mathcal{I},\hat{\epsilon},\hat{t}}(1^\lambda, m)$

[2] We adopt a terminology similar to the proof-carrying data (PCD) paradigm [13].

[3] In the blockchain setting, the adversary sees blocks generated by other parties, simulated by oracle $\mathcal{M}$, and sends out blocks that other parties may drop or adopt depending on whether they are valid, simulated by oracle $\mathcal{V}$.

$$\left\{\begin{array}{l} st \leftarrow \mathcal{A}_1(1^\lambda); ((\Lambda_i, G_i, z_i, r_i, x_i))_{i\in[m]} \leftarrow \mathcal{A}_2^{\mathcal{O},\mathcal{V},\mathcal{M}}(st); \\ b_1 := \mathsf{Steps}_{\mathcal{A}_2^{\mathcal{O},\mathcal{M},\mathcal{V}}}(1^\lambda, st) < (1-\hat{\epsilon}(\frac{q_{\mathcal{O}}}{m}, \frac{q_{\mathcal{V}}}{m}, \frac{q_{\mathcal{M}}}{m}))m \cdot \hat{t}; \\ b_2 := \bigwedge_{i=1}^{m}((G_i, z_i, r_i) \in Q_{\mathcal{O}} \wedge V_{\Lambda_i}(G_i \oplus (z_i, r_i, x_i)) = 1); \\ \text{ret } b_1 \wedge b_2 \end{array}\right\}$$

where $q_{\mathcal{O}}$ queries are made to oracle

$$\mathcal{O}(\Lambda, G, z) = \left\{ r \leftarrow \{0,1\}^\lambda; \text{ return } r \right\},$$

$q_{\mathcal{V}}$ queries are made to oracle

$$\mathcal{V}(\Lambda, G) = \left\{ \text{if } V_\Lambda(G) = 0, \text{ then return } 0, \text{ else } \text{ return } 1 \quad \right\},$$

and $q_{\mathcal{M}}$ queries are made to oracle

$$\mathcal{M}(\Lambda, G, z) = \left\{\begin{array}{l} \text{if } V_\Lambda(G) = 0, \text{ then return } \perp \\ \text{else, } r \leftarrow \{0,1\}^\lambda; \text{ return } (r, M_\Lambda(G, z, r)) \end{array}\right\}.$$

As hinted above, we require that the MH computation is also robust against grinding attacks that take advantage of the seed oracle $\mathcal{O}$. The intuition behind this is that the adversary can possibly sample many seeds from $\mathcal{O}$, and choose to do the computation only for the easier ones among them. Security against such attacks boils down to upper-bounding the speed-up the adversary obtains by making extra queries to $\mathcal{O}$.

Since the adversary can always perform a DAG computation for a seed generated by $\mathcal{O}$ in $\hat{t}$ steps, extra queries to $\mathcal{O}$ can only speed-up $\mathcal{A}$ by at most $\hat{t}$ steps, i.e., for $a \geq 0$

$$\begin{aligned} &(1-\hat{\epsilon}(1,b,c))m\hat{t} \leq (1-\hat{\epsilon}(1+a,b,c))m\hat{t} + ma\hat{t} \\ &\Leftrightarrow \hat{\epsilon}(1+a,b,c) \leq \hat{\epsilon}(1,b,c) + a. \end{aligned} \tag{1}$$

This implies that any DAG computation enjoys some resistance to grinding attacks related to the worst-case runtime of M. We formally state this in the following lemma.

Lemma 3. *Let $\mathcal{I}$ be a family of DAG computations that is $(\hat{t}, \hat{\epsilon}, \hat{k})$-MH. Then, for any function $\bar{\epsilon}$ where for any $a \geq 0$ it holds that $\bar{\epsilon}(1+a,b,c) \leq \hat{\epsilon}(1,b,c) + a$, $\mathcal{I}$ is $(\hat{t}, \bar{\epsilon}, \hat{k})$-MH.*

Proof. For the sake of contradiction, assume that $\mathcal{I}$ is not $(\hat{t}, \bar{\epsilon}, \hat{k})$-MH. This implies that there exist an adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, $m \geq \hat{k}$ and $a \geq 0$, am polynomially large in λ, such that $\mathcal{A}$ wins in $\mathsf{Exp}^{\mathrm{MH}}_{\mathcal{A},\mathcal{I},\bar{\epsilon},\hat{t}}(1^\lambda, m)$ with non-negligible probability. Since the total number of queries to oracle $\mathcal{O}$ is bounded by a polynomial, by an averaging argument it holds that there exists some $\overline{a} \geq 0$ such that the event that $\mathcal{A}$ wins in $\mathsf{Exp}^{\mathrm{MH}}_{\mathcal{A},\mathcal{I},\bar{\epsilon},\hat{t}}(1^\lambda, m)$ and the number of queries to $\mathcal{O}$ is exactly $(1+\overline{a})m$ occurs with non-negligible probability. Let $\overline{a}$ be the smallest such value. If $\overline{a} = 0$, then $\mathcal{A}$ can be used to directly break the $(\hat{t}, \hat{\epsilon}, \hat{k})$-MH property of $\mathcal{I}$. Otherwise, we use $\mathcal{A}$ to construct an adversary $\mathcal{A}' = (\mathcal{A}'_1, \mathcal{A}'_2)$ that breaks the $(\hat{t}, \hat{\epsilon}, \hat{k})$-MH property, as described in the following paragraph.

$\mathcal{A}'_1$ behaves exactly as $\mathcal{A}_1$ in the first phase of the experiment, and passes the generated state st to $\mathcal{A}'_2$. $\mathcal{A}'_2$ runs $\mathcal{A}_2$ internally. Queries to oracles $\mathcal{M}, \mathcal{V}$ made by $\mathcal{A}_2$ are answered by $\mathcal{A}'_2$ using the appropriate oracles. On the other hand, only the first m queries to oracle $\mathcal{O}$ made by $\mathcal{A}_2$ are answered using oracle $\mathcal{O}$ whereas all further responses for queries to $\mathcal{O}$ are simulated by $\mathcal{A}'_2$. In case that $\mathcal{A}_2$ is successful and produces valid witnesses for m of the queries asked, $\mathcal{A}'_2$ collects the queries that were only simulated, and solves their respective problems itself using the standard solver M. Finally, it outputs the witnesses for the m queries made to $\mathcal{O}$. Otherwise, it outputs $\perp$.

We now analyze the winning probability of $\mathcal{A}'$. First, note that $\mathcal{A}'_1$ runs in polynomial time and produces exactly the same output as $\mathcal{A}_1$. $\mathcal{A}'_2$ runs solver M at most $\overline{a}m$ times, since at most $\overline{a}m$ of the queries asked to $\mathcal{O}$ by $\mathcal{A}_2$ are simulated. By assumption, the execution of $\mathcal{A}_2$ costs less than $(1-\hat{\epsilon}(1+\overline{a},b,c))m\hat{t}<(1-\hat{\epsilon}(1,b,c)-\overline{a})m\hat{t}$ steps; and the additional executions of M by $\mathcal{A}'_2$ cost at most $\overline{a}m\hat{t}$ steps. Thus, using $q_{\mathcal{O}}=m$ queries to $\mathcal{O}$, the overall steps taken by $\mathcal{A}'_2$ in the MH game are less than

$$(1-\hat{\epsilon}(1,b,c)-\overline{a})m\hat{t}+\overline{a}m\hat{t}=(1-\hat{\epsilon}(1,b,c))m\hat{t}\,,$$

implying that $\mathcal{I}$ is not $(\hat{t},\hat{\epsilon},\hat{k})$-moderately hard, thus concluding the proof. □

On achieving moderate hardness. It is important to note that $(\hat{t},\hat{\epsilon},\hat{k})$-moderate hardness, for reasonable parameters, is (probably) not achievable for all families of DAG computations. In particular, to satisfy moderate hardness, we cannot allow for generic DAG computations.

To illustrate this, consider a family of DAG computations allowing for an instance to be crafted in the following way: a key pair of a trapdoor permutation is generated, the public key is embedded in the instance, and the exploration step is designed in such a way that it implies to compute the pre-image of a random nonce. Clearly, such a DAG computation would not be moderately hard in any reasonable way.

Although this example is extreme, it distinctly demonstrates the necessity to carefully restrict the characterization of a DAG-computation family. In contrast, moderate hardness seems to be a reasonable assumption for a large class of computations with sufficiently simple exploration and verification algorithms, e.g., for a large class of WalkSAT problems (since WalkSAT involves only very elementary computation steps). The adversary now can still craft problems trying to gain computational advantage in the DAG computation, but randomization can help to mitigate this effect to a large extent.

Further research is still required on characterizing DAG-computation families with strong evidence for moderate hardness (under reasonable parameters).

4 The PoUW Blockchain Protocol

In this section we describe and prove secure a generic PoUW blockchain protocol implementing the DPLS algorithm.

4.1 Protocol description

We first summarize some informal requirements that our protocol must satisfy to qualify as a candidate protocol for useful-work mining. We then describe our protocol while motivating the design choices by these requirements. The requirements are motivated from both sides: blockchain security, and, efficiency of the DPLS algorithm:

1. Blockchain security:

   (a) No grinding: the adversary cannot gain mining advantage by cherry-picking exploration steps of low complexity.

   (b) Precomputation resilience: problem instances cannot be adversarily manufactured such that the adversary gains access to faster block production. Computation before seeing the head of the chain to be extended cannot contribute towards computing the respective PoUW.

   (c) Adjustable mining difficulty: The block difficulty can be adjusted to the mining power applied by the network.

2. DPLS efficiency:

   (a) Frequent updates: Results about new points explored are published (relatively) fast.

   (b) Small overhead: The computational overhead of integrating exploration algorithm M into PoUW is small (implying that honest mining performs useful work).

The high-level architecture of the protocol is similar to Bitcoin, i.e., blocks are chained together by referencing each other by hash, and, during each round, a miner selects the longest chain from his view, and tries to extend it by a block. Two modifications are applied: standard PoW is replaced by PoUW, and we apply 2-for-1 PoW [24] in order to accommodate different types of blocks for reasons explained below. See Figure 2 for further reference.

The core of the mining algorithm consists of applying the exploration algorithm M, constituting the "useful part" of the PoUW. To defend against precomputation (Requirement 1a), the computation of M is prepended by hashing the candidate block (see first $\mathcal{H}$ box in Figure 2), thereby randomizing the computation to be performed by M. Furthermore, similarly to Nakamoto consensus, this "pre-hash" of the block must lie below an initial target T_1, to antagonize grinding for parameters of M that result in lower-than-average computation complexity: resampling new parameters must be more expensive than the worst-case complexity of M.

By Requirement 1c, we must be able to reduce the mining-success probability below the success probability of hashing against T_1—which is currently fully determined by the computational characteristics of the problem instance and unrelated to mining participation in the network. One possibility to address this issue would be to further lower the target T_1 to make pre-hashing as hard as required for ledger security; however, this would come against a big loss in usefulness, as miners would spend most of their time performing hashing. Instead, we have the miner feed the output of M into one *single* round of "post-hashing" (see second $\mathcal{H}$ box in the figure) that decides, against a threshold T_3, whether the block is eligible for publication. This second threshold adjusts the overall mining difficulty to a level required by the security analysis to guarantee good and secure blockchain characteristics. Note the additional effect of post-hashing to adapt mining difficulty: the miner only learns whether a PoUW attempt is successful *after* executing M, i.e., the computation cannot be cut short to speed up block creation.

A miner loops, many times, the computation sequence of pre-hashing (against T_1), useful work, and one post-hash, until the post-hash of a sequence lies below T_3, allowing for the block to be published. To preserve progress, the best point (by means of scoring algorithm g) from all recent computation sequences is stored for eventual inclusion in a future block to be published. Note that finding a *good* new point is decoupled from mining success, thus helping to establish Requirement 1b. Furthermore, only publishing the best one from a batch of new points, rather than greedily publishing all of them incrementally, helps to accommodate Requirement 2b.

Considering Requirement 2a under Bitcoin parameters, we cannot afford that a miner waits with his update until he mines a block. For this reason, we incorporate 2-for-1 PoW to allow for the publication of different types of blocks, so-called *ranking blocks* which are "standard" Bitcoin blocks of high difficulty (target T_2), and so-called *input blocks* of low difficulty (target T_3, i.e., hash range $T_2 < h \leq T_3$) which are not part of the chain but are rather handled like transactions to be eventually referenced by a ranking block. A miner now includes his best point explored whenever he hits either type of a block; and by setting the input-block difficulty low enough, the update rate per miner is high enough to distribute progress in the explored points fast, while having no considerable impact on the blockchain characteristics.

A block contains two points explored using M: the "winner" one that lead to the small post-hash, and the "best" one that is included to progress the DPLS algorithm. In order to accommodate 2b, we minimize the cost of block verification by having the miner append a SNARG proving correctness of both exploration points contributing to the block, i.e., a SNARG proving membership to the following language: $L = \{((\Lambda, G, z, r', x'), (\Lambda_b, G_b, z_b, r'_b, x')) | V_\Lambda(G \oplus (z, r', x')) = 1 \wedge V_{\Lambda_b}(G_b \oplus (z_b, r'_b, x'_b)) = 1\}$, where $G \oplus (z, r', x')$ denotes the graph G extended with vertex (z, r', x') as defined in Section 3.2.

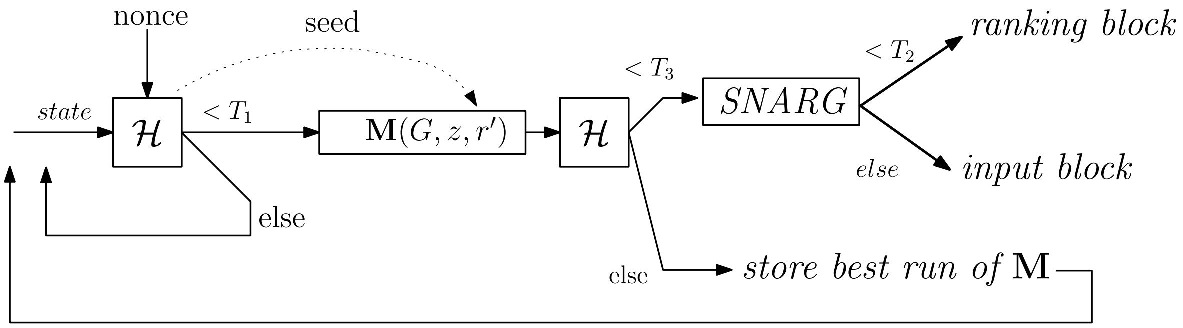


Figure 2: A diagram of the PoUW mining procedure.

A detailed description of the PoUW procedure is given in Algorithm 5. The mining algorithm is parametrized by the longest blockchain received $\mathcal{C}$, the message to be included in the block m, the problem instance Λ selected by the miner to work on,[4] the related transcript G extracted from $\mathcal{C}$, and the selected inner state z. The pre-hash input includes these parameters, the hash of the previous block s, and a random nonce r; and yields a unique seed r' for M. At this point, all parameters of M, Λ, G, z, and r', are fully determined based on the data initially hashed, thus establishing that each small pre-hash found by the adversary can only be used to perform one matching post-hash attempt. We note that if, in a round, a miner does not have enough steps to finish running the PoUW procedure, e.g., he only manages to find a small pre-hash, he continues the next round from the point it stopped.

Input blocks are processed on the application level of the protocol, as they are not essential for consensus. In particular, for an input block to be considered by miners, it must satisfy certain conditions: the related pre- and post-hashes are small enough, and the computation of M is correct. Note that ranking blocks are also treated as input blocks, and can be included in the payload of other ranking blocks. As in [24], an input block can be included in the payload of different ranking blocks in diverging chains, which ensures that all input blocks mined by an honest party will eventually be included in the main chain, and no progress is ever lost. The full protocol is presented in Appendix A.

Remark 1. (SNARG overhead) Note usefulness is not necessarily substantially impacted by a large SNARG-computation overhead as each state update involves a large number of exploration steps (on average) but SNARGs for only two of the M-computations performed. This average number of

[4] Even if clients do not post any such problem, we assume that miners can always generate a MH problem based on the hash of the block they are extending. For example, the fixed-time hash-based PoW of [5, 14] can be used. This amounts to a "fall-back" DPLS computation.

exploration steps can thus be raised in a trade-off against the state-update frequency in the system, helping to establish Requirement 2b.

Algorithm 5 The PoUW procedure is parameterized by hardness parameters $T_1, T_2, T_3 \in \mathbb{N}$, the SNARG system, hash function $H(\cdot)$, the explore algorithm M and scoring algorithm g.

```
var $(score_b, s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b) := (\infty, \bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot)$   ▷ Global variables related to the best run. They are reset either when the miner finds a block, or the instance he is working on changes
var $(z, com) := (\bot, \bot)$
function PoUW($\mathcal{C}$, $m$, $\Lambda, G$)
    $s := H(head(\mathcal{C}))$
    if $(z = \bot)$ then $z \leftarrow \mathsf{F}(\Lambda, G)$   ▷ Compute inner state
    $r \leftarrow \mathcal{U}_\lambda$   ▷ Sample nonce
    $h := H(s, m, com, \Lambda, G, z, r)$   ▷ Compute pre-hash
    if $(h < T_1)$ then
        $r' := H(s, m, com, \Lambda, G, z, r, h)$   ▷ Compute seed
        $x' := M_\Lambda(G, z, r')$   ▷ DAG computation
        $h' := H(s, m, com, \Lambda, G, z, r, h, x')$   ▷ Compute post-hash
        if $(h' < T_2$ or $T_2 \le h' < T_3)$ then   ▷ New block
            if $(s_b = \bot)$ then $(s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b) := (s, m, com, \Lambda, G, z, r, x')$
            $\pi := \text{SNARG.}\mathsf{P}(\Sigma, ((\Lambda, G, z, r', x'), (\Lambda_b, G_b, z_b, r'_b, x'_b)));$   ▷ Compute correctness proof
            $B := \langle (s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b), (s, m, com, \Lambda, G, z, r, x'), \pi \rangle$
            if $(h' < T_2)$ then $\mathcal{C} = \mathcal{C}B$   ▷ Return the new chain to the main function
            else Diffuse($(input, B)$)   ▷ Else diffuse the new input block
            $(score_b, s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b, z, com) := (\infty, \bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot, \bot)$
        else   ▷ No block
            if $(\Lambda \neq \Lambda_b$ or $g_\Lambda(x') > score_b)$ then   ▷ Working on a different instance, or found a better point
                $(s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b) := (s, m, com, \Lambda, G, z, r, x')$   ▷ Store best update
                $com := H(s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b)$   ▷ Commitment to previous best attempt
                $score_b := g_\Lambda(x')$
    return $\mathcal{C}$
```

4.2 Dealing with multiple instances

We now describe how to extend our design to handle multiple problem instances more effectively. The main motivation is that, for certain operations in the solving process, the protocol participants may want to establish a common view on the state of the algorithm in execution. For instance, the problem issuer may want to govern the solving process by further extending or stopping it, based on the quality of the solutions found. Moreover, agreement allows us to implement more complicated behavior of the solving algorithm itself, e.g., restarting the solving process if the initial parameters lead to a bad region of the solution space, a common technique in parallel stochastic local search. Without full agreement such operations may be hard to implement or lead to inefficiencies.

To achieve agreement we put forth the idea of *pipelining*. Analogously to pipelining in processor architecture, our computational machinery has two stages: running the solving algorithm for a problem, and establishing a common view on the state of the algorithm. Different problem instances are interleaved to cycle between these two stages during the protocol execution. In more detail, the protocol defines an epoch-based schedule determining which problem is solved at each epoch, where each epoch is defined by m consecutive blocks in the mainchain. For instance, the first epoch (blocks 1 to m) is assigned problem Λ_1, the second epoch (blocks $m+1$ to $2m$) problem Λ_2, the third epoch problem Λ_1 again, etc. An epoch that a given problem is not assigned to, gives the parties enough time to agree on all previous updates generated for the problem. While, in this example, only two problems are solved, the schedule can contain a large number of them to be run one after another, possibly with different frequencies. We note that different scheduling scenarios can be implemented by making use of the information posted in the blockchain, e.g., selecting as the next instance to be solved the one that offers the largest reward. Further, to achieve agreement, we allow related updates to be included in the main chain only for a limited number of blocks k after the end of the epoch that a problem was actively being solved. Parameter k must be large enough to ensure that any honest update is guaranteed to be included in the mainchain.

Surprisingly, pipelining also helps to deal with input-block DoS attacks, while at the same time allowing miners to consume unsettled input blocks during mining, thus boosting the performance of DPLS. In more detail, in an input-block DoS attack, the adversary pre-mines a large amount of input-blocks, and eventually releases them to flood the network. To deal with such an attack, Fruitchains [52] requires that valid ranking blocks only include recent input blocks, i.e., input blocks that reference a recent ranking block. This mechanism does not work in our setting where we want miners to refer to work by unsettled input blocks: the adversary can undetectedly backdate an input block to prevent its eventual settlement, while an honest miner will still reference it in his block—thus making the honest block invalid. Instead, following our architecture, we deal with DoS attacks by only requiring that valid ranking blocks include input blocks that (i) are "fresh", i.e., they reference a ranking block of the same epoch, and (ii) reference other input blocks that are part of the main chain. This has the effect that old input blocks that are not part of the common view established earlier are dismissed. Note, that despite these restrictions honest miners can still reference "fresh" unsettled input blocks while mining for a new block, as they can be sure that there is enough time for these blocks to be included in the main chain.

4.3 Incentives structure

Useful-work rewards in our protocol are given in epochs—periods of protocol execution measured in fixed (ranking) block intervals. Clients who submit DPLS instances Λ for processing will lock funds for executing the algorithm initializing a reward pool P_Λ for the problem. At the end of each epoch, the contribution of each miner towards solving a particular instance Λ will be measured as the fraction of input-blocks they contributed out of the total that is directed to that instance. Using this metric, miners will be rewarded proportionally from the reward pool of each problem. Note that the fraction of P_Λ to be allocated to a particular epoch will be determined by a function selected by the client that takes as input the total work invested in that problem for that particular epoch. Clients can keep increasing the reward pool P_Λ as it gets depleted to ensure the continuous interest of the miners to their problem instance.

The above mechanism essentially relies on the 2-for-1 mechanism and the fact that any set of miners will produce a subset of updates roughly proportional to their computational power. In this way our protocol is fair, and can be proven as such along the lines of Fruitchains [52]; we omit further details. We note that, in addition to the above, miners can be further rewarded by transaction fees and inflation of the base coin supply by creating an additional reward pool that is not tied to any optimization problem.

Using the above setup it is easy to see that rational miners have an incentive to stick to the protocol under the assumption that the pool rewards offset their operational costs. Note that a "griefing" miner can still deviate somewhat by not publishing their best update for the useful algorithm during the inner parallelization step and opting instead for, e.g., a randomly selected update. This is a type of griefing since the miner does not gain anything by doing this (as the work has been completed anyway). Even though dealing with such attacks is beyond the scope of the present exposition, we note that it is possible to assess blocks for such griefing behavior by statistically contrasting the "best" claimed update to the "winning" update associated with the small post-hash.

Distributing the native coin of the ledger is done with an inflation schedule that unlocks coins via a block reward for each ranking block. Recall that in case no other problem instance is available miners revert to the default problem instance (cf. footnote 4). Solving this instance does not have any external value but it gives a way to bootstrap the platform (as miners will engage at genesis with the intention of receiving tokens from the ranking blocks). The market that emerges around the platform's native asset has the problem setters buying the token in the open market from miners or speculators, so that they fund the problem instances they post in the platform.

5 Security Analysis

Next, we formally analyze the security of our protocol. First, we show that—assuming that the underlying DAG computation is moderately hard and that honest parties control the majority of the computational power in the network—our protocol implements a robust transaction ledger. Then, we define and analyze the usefulness rate of the protocol.

5.1 Ledger security

Let Π denote our blockchain protocol. The consistency analysis of the longest chain rule appearing in Π involves a number of new challenges, including an exotic Markov chain governing the mining dynamics and the possibility of "restarts" in this chain generated by the delivery of a new block, perhaps by the adversary. We adapt the language of [37, 6] to this setting and then develop the tools necessary for the associated probabilistic analysis. (Our treatment below does not require familiarity with these previous papers.)

For simplicity, in the main body of the paper, we discuss the case *without restarts*, which is to say that the protocol carried out by the honest parties does not restart the mining process when it learns of a longer chain, but rather completes the current computation. Intuitively, restarts *improve* the security properties of the blockchain, as they help ensure that honest parties are mining on current chains. However, the situation is somewhat complicated by the fact that restarts do permit the adversary to correlate the states of the honest parties in the Markov chain. Specifically, note that an adversary holding a chain that exceeds the length of those chains currently held by honest parties may strategically release the chain to honest players—perhaps with detailed knowledge about their current state—so as to achieve some short-term control over the distribution of honest mining successes. Despite such correlations, we show in Appendix D that the intuition above is correct: the adversarial advantage achieved by exposing adversarial blocks to honest miners is overshadowed by the fact that such exposures increase the length of the blockchain held by the honest recipient; in the language of the analysis below, such an exposure has an effect just as beneficial as an honest mining victory!

We adopt a discrete time model, dividing time into short "rounds" with duration c_H equal to the time taken to carry out a hash query. We reflect the essential block-generation events of an execution of the protocol with a *characteristic* string: this determines, for each round, the number of adversarial and honest ranking blocks generated. Thus our characteristic strings have the structure $w = w_1, \ldots, w_L$ where each $w_i = (h_i, a_i) \in \mathbb{N}^2$ and h_i and a_i denotes the number of honest and adversarial ranking block discoveries, respectively; here L is the lifetime of the protocol.

Ultimately, our protocol Π determines a blockchain of ranking blocks, which themselves refer to input blocks. Such a structure determines a linear order on the collection of input blocks referenced in the blockchain of ranking blocks (by ordering input blocks referenced in a particular ranking block according to the order of their references in the ranking block). Ultimately, we wish to establish the two fundamental ledger properties: *liveness* and *persistence*.

Persistence with parameter $k \in \mathbb{N}$. Once a node of the system proclaims a certain input block in the *stable* part of its ledger $\mathcal{L}$, the remaining nodes either report the input block in the same position of their ledgers, or report a stable ledger which is a prefix of $\mathcal{L}$. Here the notion of stability is a predicate that is parametrized by a security parameter k; specifically, an input block is declared *stable* if and only if it is in a (ranking) block that is more than k (ranking) blocks deep in the ledger.

Liveness with parameter $u \in \mathbb{N}$. If all honest nodes in the system attempt to include a certain input block then, after the passing of time corresponding to u rounds, all nodes report the input block as stable.

We establish these properties as consequences of three more elementary properties of the blockchain of ranking blocks, originally formulated in [24] (we use a slightly adapted formulation from [18]):

- **Common Prefix (CP); with parameter $k \in \mathbb{N}$.** The chains $\mathcal{C}_1, \mathcal{C}_2$ adopted by two honest parties at the onset of rounds $r_1 \leq r_2$ are such that $\mathcal{C}_1^{\lceil k} \prec \mathcal{C}_2$, where $\mathcal{C}_1^{\lceil k}$ denotes the chain obtained by removing the last k blocks from $\mathcal{C}_1$, and $\prec$ denotes the prefix relation.

- **Existential Chain Quality (ECQ); with parameter $s \in \mathbb{N}$.** Consider the chain $\mathcal{C}$ adopted by an honest party at the onset of a round and any portion of $\mathcal{C}$ spanning s prior rounds; then at least one honestly-generated block appears in this portion.

- **Chain Growth (CG); with parameters $\tau \in (0, 1]$ and $s \in \mathbb{N}$.** Consider the chain $\mathcal{C}$ possessed by an honest party at the onset of a round and any portion of $\mathcal{C}$ spanning s contiguous prior rounds; then the number of blocks appearing in this portion of the chain is at least τs. We call τ the *speed coefficient*.

One of the important conclusions of previous work is that these properties (CP, CG, and ECQ) directly imply liveness and persistence and—from an analytic perspective—can be guaranteed merely based on the characteristic string associated with a particular execution. This fact is fairly immediate for CG and ECQ, whereas identification of the properties of the characteristic string that guarantee CP is more delicate.

We give a summary of this theory in Appendix C, both so that the article is self-contained and so that we can describe the extension to restarts in Appendix D. Fortunately, it is possible to succinctly reflect the conclusions of this theory as they relate to our needs, which is the next order of business.

To continue, we first introduce two assumptions related to the level of moderate hardness of the underlying DAG-computation family $\mathcal{I}$ used by Π, and the complexity of the SNARG system used.

Assumption 1. For parameters $\hat{t}, \hat{\epsilon}, \hat{k}$, we assume that the DAG computation family $\mathcal{I}$ used in Π is $(\hat{t}, \hat{\epsilon}, \hat{k})$-moderately hard.

Assumption 2. For parameters $c_{\mathsf{P}}, c_{\mathsf{V}}, c_{\mathsf{S}}$, we assume that there exists a SNARG system *SNARG* where running the prover (resp., verifier, setup) takes c_{P} (resp. $c_{\mathsf{V}}, c_{\mathsf{S}}$) steps.

Let $w = w_1, \ldots, w_L$ be a characteristic string, as above. We fix a constant Γ, a time period with the following Γ-*serializing guarantee*: *if a ranking block B_2 is generated by an honest party P at least Γ rounds after the honestly-generated ranking block B_1 is diffused, then the full computation supporting B_2 (including the prehash) was carried out while P was aware of B_1.* In our setting, Γ can be set to $2 + \Delta + c_{\mathsf{P}}/c_H + \hat{t}/c_H$ (corresponding to the number of rounds taken to produce the prehash (≤ 1), useful work ($\leq \hat{t}/c_H$), post-hash (≤ 1), and SNARG (c_{P}/c_H) for block B_2 in addition to any network delay). With this in mind, we say that t is a Γ-*isolated uniquely successful round* if the region $w_{t-\Gamma} \ldots w_t \ldots w_{t+\Gamma}$ satisfies $h_t = 1$ and, furthermore, that the sum $\sum h_i = 1$ over this region (recall $w_i = (a_i, h_i)$). Note that a round cannot be isolated if it is not followed by at least Γ symbols. For each t define I_t to be an indicator variable for the event that t is an isolated uniquely successful round.

The basic quantities of interest are given by two conventions for accounting for the balance of adversarial and honest successes.

Definition 4 (The barrier walk; the free walk.)**.** Let $x = x_1, \ldots, x_n \in \mathbb{N}^*$. Define the *barrier walk* $B(x)$ by the recursive rule $B(\epsilon) = 0$ (for the empty string ϵ) and, for any $x \in \mathbb{N}^*$ and $a \in \mathbb{N}$, $B(xa) = \max(B(x) + a, 0)$. Likewise, define the *free walk* $F(x) = \sum_i x_i$.

Definition 5. For a characteristic string $w \in (\mathbb{N}^2)^L$ and $0 < t \leq L$, define the *margin effect* $w_t^* = a_t - I_t \in \mathbb{N}$ (and w^* to be the sequence of elements of $\mathbb{N}$ given by this rule). We then define $B^*(w) = B(w^*)$ and $F(w) = F(w^*)$. Finally, for a characteristic string $w = xy$ with $|x| = \ell$, we define the ℓ-*isolated margin* of w to be $\beta_\ell(w) = B(x^*) + F(y^*)$.

The role of ℓ-isolated margin is clarified by the following, which establishes a direct connection to common prefix.

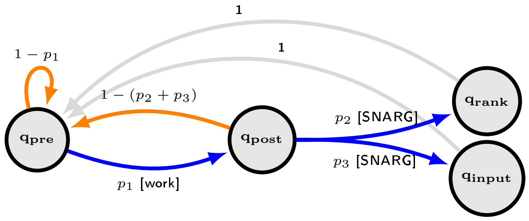
Theorem 6. *Let $w \in (\mathbb{N}^2)^L$ be the characteristic string associated with an execution satisfying the Γ-serializing guarantee. Suppose, further, that (i.) the execution satisfies $(k/s, s)$-CG, and (ii.) for any prefix xy of w for which $|y| \geq s$, we have $\beta_{|x|}(xy) < 0$. Then the execution satisfies k-CP.*

This is the major component in the following theorem; as noted, the details of this existing theory are discussed in Appendix C.

Theorem 7. *Let D_Π be a distribution on characteristic strings of length L (induced by a protocol Π), λ a security parameter, and $\alpha > \beta$ two constants corresponding to the rate of uniquely isolated blocks and the rate of adversarial blocks, respectively. Assume that for a constant $\delta < (\alpha - \beta)/2$, when w is drawn from D_Π, every interval of w of length poly(λ) has at least $\alpha - \delta$ uniquely isolated blocks and no more than $\beta + \delta$ adversarial blocks except with negligible probability. Then, except with negligible probability, the protocol satisfies (i.) CG with $s =$ poly(λ) and constant speed coefficient, (ii.) ECQ with $s =$ poly(λ), and (iii.) CP with parameter $k =$ poly(λ).*

5.1.1 Analysis of the Markov chain

In light of the description above, we are specifically interested in analyzing the sequence of (i.) adversarial mining successes and (ii.) uniquely isolated honest successes. The analysis is simplified by the fact that the time evolution of the honest parties is independent. We focus on the Markov chain pictured below, showing nodes for "pre-hash", "post-hash", and both "ranking" and "input" block production. It is convenient for us to further decorate our transitions with delays: orange edges are traversed in a single round (or c_H time, corresponding to hash queries), the gray edges are traversed instantaneously, and the blue edges have transition times given by the distribution of useful work (upper bounded by $\hat{t}$) and SNARG times (c_{P}). (Note that the timing delays indicated in this chain could be implemented with paths of individual states connected by edges with unit delay, so this presentation can be reflected with a standard Markov chain.) While the basic security properties of the protocol depends only the production of ranking blocks, the dynamics of the Markov chain depends on both ranking and input block production.



We begin by establishing that—despite the fact that honest parties begin the protocol synchronized (in "pre")—they quickly converge to mutually independent positions in the mining chain. Looking ahead, this mixing argument will be instrumental to establish bounds on uniquely isolated block production.

5.1.2 The mixing time; convergence to mutual independence

By a standard coupling argument we get the following:

Lemma 8. *Consider m particles $P_1, \ldots, P_m$ independently evolving on the Markov chain with any fixed initial states. Let $(S_1, \ldots, S_m)$ denote a random variable so that each coordinate is independent and stationary on the chain. Then letting $T = L(1 + (\hat{t} + c_{\mathsf{P}})/c_H)$,*

$$\|(P_1^T, \ldots, P_m^T) - (S_1, \ldots, S_m)\|_{t.v} \leq m(1 - p_{couple})^L\,,$$

where $\|X - Y\|_{t.v}$ denotes the distance in total variation between the random variables X and Y. Here $p_{couple} > 0$ is a constant that depends only on $c_{\mathsf{P}}/\hat{t}$.

Proof. We proceed with a standard coupling argument. Consider m particles (parties) $P_1, \ldots, P_m$, initially in the state q_{pre}, that carry out simultaneous, independent evolution according to the dynamics of the chain. We wish to show that the joint distribution of positions of all the particles quickly converges to m independent copies of the stationary distribution. For this purpose, consider m additional particles $R_1, \ldots, R_m$ on the chain, initially distributed independently according to the stationary distribution. We let P_i^t and R_i^t denote the positions of the particles at time t. We give a simple coupling C of the evolution of $P_1^t, \ldots, P_m^t$ with $R_1^t, \ldots, R_m^t$, and apply the standard "coupling lemma" which establishes convergence to the stationary distribution. The coupling C is described, at each time step, by a family of random variables U_i^t; for each $i \in \{1, \ldots, m\}$, $U_i^t : Q \to Q$ is a function where Q is the set of states of the chain (which is in fact larger than the diagram indicates as a result of implementing the "long" transitions). The "update functions" U_i are chosen so that the full ensemble of entries $(U_i^t(q))$ (over all t, i, and $q \in Q$) are independent and each $U_i(q)$ is distributed according to the defining distribution for the state q. Then P_i and R_i are updated according to the same update: $P_i^{t+1} = U_i(P_i^t)$ and $R_i^{t+1} = U_i(R_i^t)$. Observe that the dynamics of the P_i^t are as promised, each independently evolving according to the chain; the same is true of the R_i^t, which of course continue to be independent and stationary. Observe that if $R_i^t = P_i^t$ at some time t this property will be retained by the coupling in the future (as they are subject to the same update function). Now, consider any time period of length $E = 1 + (\hat{t} + c_{\mathsf{P}})/c_H$ rounds and any pair of particles P_i and Q_i. Observe that both particles must visit the state q_{pre} during this time period (as $\hat{t}$ and c_{P} are upper bounds on the transition times of the blue transitions); it follows that if the first of the two particles to visit q_{pre} remains in that state for the remainder of the E time steps then the two particles must couple (that is, coincide during this time period and forever after). Recalling that we take $T_1 \geq \hat{t}/c_H$, we find that the probability that that first particle remains in q_{pre} when the second one arrives is at least $p_{\text{couple}} := (1-p_1)^{E/c_H} = (1-p_1)^{(\hat{t}+c_{\mathsf{P}})/c_H} = [(1-p_1)^{\hat{t}/c_H}]^{(1+c_{\mathsf{P}}/\hat{t})} \geq [(1-1/T_1)^{T_1}]^{(1+c_{\mathsf{P}}/\hat{t})} \geq (1/e - O(1/T_1))^{(1+c_{\mathsf{P}}/\hat{t})}$. Thus p_{couple} is a constant larger than zero (and can be lower bounded as a function of the constant $c_{\mathsf{P}}/\hat{t}$). Note that the events that P_i couples with R_i (for distinct i) during such an epoch are independent, and it follows that after L such epochs the probability that there is a pair (P_i, R_i) that has not coupled is no more than $n(1-p_{\text{couple}})^L$. By the standard coupling lemma (see, e.g., [4, §12]), after L epochs the distance in total variation between $(P_1, \ldots, P_m)$ and the independent stationary distribution in each coordinate is no more than $m(1-p_{\text{couple}})^L$, which tends to zero exponentially quickly in L. This proves the lemma. □

5.1.3 Bounds on the events of interest

Consider, as above, the population of particles (players) $P_1, \ldots, P_n$ on the Markov chain. According to an evolution of these particles, given by the random variables P_i^t, we are interested in establishing upper bounds on the rate at which the adversary produces ranking blocks, and a lower bound on the rate at which the honest players produce uniquely isolated blocks.

Lemma 9. *Consider m parties, with arbitrary initial conditions but evolving independently on the Markov chain. Let $S = (\hat{t}+c_{\mathsf{P}})/c_H+1$ and consider any interval of R rounds, the first of which starts at least S steps after the evolution begins. Then the probability that a particular player generates at least k ranking blocks in this interval is no more than $\binom{R+S}{k}(p_1p_2)^k \leq (R+S)^k(p_1p_2)^k$.*

Proof. In order for a ranking block to be produced during the interval, the pre-hash associated with the ranking block must have succeeded no earlier than S rounds prior to the beginning of the interval. To analyze this event, let I denote the set of rounds consisting of all rounds referred to in the statement of the theorem and the previous S rounds. Let A_i^* and B_i^* denote two families of independent indicator random variables for which $\Pr[A_i^* = 1] = p_1$ and $\Pr[B_i^* = 1] = p_2$. Then define A_i to be the indicator random variable with the following definition: if the player completes a pre-hash query during the ith round, then A_i is the indicator variable for the success of this pre-hash; otherwise, there is no pre-hash completed and $A_i = A_i^*$. Similarly, B_i is defined with similar marginals: if round i contains a successful pre-hash, B_i is the indicator random variable for the event that the subsequent post-hash is successful; otherwise, there was no successful pre-hash in round i and $B_i = B_i^*$. Observe that if there were k ranking blocks successfully generated during R, then $A_iB_i = 1$ for at least k of these i. Furthermore, as a result of the conditional definitions above, the families A_i and B_i are independent, Bernoulli random variables. Thus the probability of k successes is no more than $\binom{R+S}{k}(p_1p_2)^k$, as desired. □

Lemma 10. *Consider m independent parties walking on the Markov chain in the stationary distribution. Let p^*_{rank} denote the stationary probability of q_{rank}, then*

$$\Pr[t \textit{ is a uniquely isolated round}] \geq m(1-(3\Gamma)p_1p_2)^m p^*_{rank}.$$

Proof. We consider the event that a particular honest player P_w generates a uniquely isolated block in round t. Let p^*_{rank} denote the probability of q_{rank} under the stationary distribution. Then the probability of a success in round t is exactly p^*_{rank}. In the event that P_w produces a ranking block in round t, consider the trajectory of P_w through the Markov chain in which the loop between round t

and the previous visit to q_{pre} is excised. The remaining trajectory follows the dynamics of the chain without conditioning, and it follows that the probability that P_w produces a second block in this region of size $2\Gamma + 1$ is no more than $3\Gamma(p_1 p_2)$ (note that the quantity S of the lemma above is no more than $(\hat{t} + c_{\mathsf{P}})/c_H + 1 \leq \Gamma - 1$). Again applying the previous lemma, the probability that the remaining $m-1$ honest players produce no blocks in this region is at least $(1-3\Gamma p_1 p_2)^{m-1}$. Noting that the events that distinct players play the role of P_w above are non-intersecting, we conclude that $\Pr[t \text{ is uniquely isolated}] \geq m(1-(3\Gamma)p_1p_2)^m p^*_{\text{rank}}$, as desired. □

In light of Lemma 8, the following is immediate.

Lemma 11. *Consider m players evolving according to the Markov chain, where the players are initially stationary and independent. Let p_{couple} denote the coupling constant of Lemma 8. Consider two rounds $\check{s} < s$ for which $|\check{s} - s| \geq L(1 + (\hat{t} + c_{\mathsf{P}})/c_H)$. Let I_s denote the indicator random variable for the event that s is uniquely isolated. Let C denote an arbitrary event depending only on the players trajectories prior to $\check{s}$. Then $|\Pr[I_s|C] - \Pr[I_s]| \leq (1 - p_{couple})^L$.*

Lemma 12. *Consider m players evolving on the Markov chain with any fixed initial states. Let p_{iso} denote the probability that a round is uniquely isolated under the stationary distribution, bounded below by Lemma 10. Fix a parameter $\sigma > 0$ and define $L = \ln(p_{iso}\sigma/2)/\ln(1-p_{couple})$ and $E = L(1 + (\hat{t} + c_{\mathsf{P}})/c_H)$. Let $\{R, \ldots, R+S-1\}$ be a sequence of rounds for which $R \geq E$. Let I_s be the event that the players produce a uniquely isolated block in round s. Then*

$$\Pr\left[\sum_s I_s \leq (1-\sigma)p_{iso}S\right] \leq E \exp\left(-\frac{(1-\sigma/2)\sigma^2 p_{iso} \cdot S}{8E}\right).$$

Proof. We prove a slightly more parameterized version. Consider an arbitrary value of $L > 0$ and define $\delta = (1 - p_{\text{couple}})^L$ and $E = L(1 + (\hat{t} + c_{\mathsf{P}})/c_H)$ (consistent with the statement of the lemma). We then show that for any $\gamma > 0$

$$\Pr\left[\sum_s I_s \leq (1-\gamma)(p_{\text{iso}} - \delta)S\right] \leq E\exp(-\gamma^2 S(p_{\text{iso}} - \delta)/2E)\,.$$

The statement of the lemma follows by choosing $\gamma = (1 - \sigma/2)$ and $L = \ln(p_{\text{iso}}\sigma/2)/\ln(1 - p_{\text{couple}})$ so that $\delta = p_{\text{iso}}\sigma/2$.

Observe that the variables I_s are not independent; however, for two sufficiently distant indices s and s', they are nearly independent as formulated in the lemma above and, in fact, the variable I_s is nearly independent of *any* conditioning on the variables $I_1, \ldots, I_{\check{s}}$ for $\check{s} < s - E$. To exploit this, we organize the random variables into E collections of "distant" variables: the ith collection consists of the variables $I_i, I_{E+i}, I_{2E+i}, \ldots$. Observe that all pairs of variables in a particular collection are at least E-distant from each other and the collections partition the complete set of variables. While the variables in a particular collection are not strictly independent, they do satisfy the requirement $\mathbb{E}[X_i | X_j, j < i] \geq p_{\text{iso}} - \delta$. Recall the basic Chernoff bound: if $X_1, \ldots, X_n$ are independent indicator random variables with $\mathsf{Exp}[X_i] = p$ then $\Pr[\sum X_i < (1-\gamma)np] \leq \exp(-\gamma^2 np/2)$. This very same Chernoff bound applies to the variables in a single collection via a standard stochastic dominance argument that compares them to i.i.d. variables with these same expectations. Observe, finally, that if $\sum_s I_s \geq (1-\epsilon)(p_{\text{iso}} - \delta)S/E$ for each collection, this same inequality applies over the whole set of variables. Taking the union bound over these E bad events concludes the argument. □

λ :	security parameter
n :	number of parties
t :	adversarial party corruption bound
t' :	amplified adversarial party corruption bound
c_H :	"mining" steps each party takes per round
$c_{\mathsf{P}}, c_{\mathsf{V}}$:	SNARG prover/verifier cost
$\hat{\epsilon}, \hat{t}, \hat{k}$:	MH DAG parameters
$T_1, p_1 = T_1/2^\lambda$:	target/success probability of prehash
$T_2, p_2 = T_2/2^\lambda$:	” of ranking block posthash
$T_3, p_3 = \frac{T_3 - T_2}{2^\lambda}$:	” of input block posthash
σ :	concentration-bound parameter
Δ, Γ :	network/serialization worst-case delay
β :	upper bound on ranking-block computation rate
δ_{MH} :	adversarial advantage in DAG computation rate
δ_{Steps} :	honest advantage in number of steps per round
δ_{tot} :	upper bound on the total block computation rate

Table 1: *The parameters of our analysis.*

5.1.4 Bounds on the number of adversarial mining successes

Next, we proceed to bound the rate of adversarial mining successes. Our analysis is going to depend on the level of moderate hardness of the underlying DAG computations family.

By Lemma 3 the speed-up the adversary gets by each extra query to oracle $\mathcal{O}$ is at most $\hat{t}$ steps. Thus, in order to protect our protocol from grinding attacks, we *set the pre-hash hardness* parameter p_1 to $c_H/((1+\sigma)\hat{t}+4)$, where $\sigma \in (0,1)$ is a parameter associated with the concentration bounds we use later in our analysis. Setting p_1 this way implies that computing a small pre-hash costs on expectation $c_H/p_1 = (1+\sigma)\hat{t} + 4 > \hat{t}$ steps; the extra steps added are related to costs occurring in our reduction later.

To simplify our presentation, we define

$$t' := t + (2n + 4(p_2 + p_3)(nc_{\mathsf{P}} + tc_{\mathsf{V}})) \cdot p_1/c_H$$

to be the increased corruption power the adversary gets, due to fact that our reduction from to the MH of the DAG computation is not tight, mainly because of the cost of generating and verifying SNARG proofs for the DAG computations. With foresight, we let β be an estimation of the rate at which the adversary produces ranking blocks

$$\beta := \frac{p_2}{(1 - \hat{\epsilon}(1, 2, 2n/t'))\hat{t} + (\frac{1}{(1+\sigma)p_1} + 1)(c_H - 4p_1)} .$$

As expected β^{-1}, the expected number of steps to find a block, is basically the number of attempts needed to find a small post-hash $(1/p_2)$, times the number of steps needed to find a small pre-hash (c_H/p_1) plus the time needed to perform the DAG computation $((1-\hat{\epsilon})\hat{t})$. The other constants of the formula are related to our security analysis, i.e., our reduction from an attacker against the blockchain to an attacker against the MH property. Finally, the parameters $0, 2, 2n/t'$ of $\hat{\epsilon}$ relate to the rate at which the adversary queries oracles $q_{\mathcal{O}}, q_{\mathcal{V}}, q_{\mathcal{M}}$ as explained in Section 3.2.

Let r.v. $Z(S)$ denote the maximum number of distinct blocks computed by the adversary during S, where the pre-hash query for each of these blocks was also issued to the RO during S. We prove that the adversary cannot mine fresh ranking blocks with rate and probability better than that of breaking the moderate hardness experiment. The main proof idea is to use an adversary that creates blocks fast, to create an adversary that breaks the moderate hardness of $\mathcal{I}$. A summary of our notation is given in Table 1.

Lemma 13. *For any set of consecutive rounds S, where $|S| \geq \hat{k}(1+\sigma)p_2/(\beta \cdot t'c_H)$, it holds that $Z(S) \geq (1+\sigma)\beta \cdot t'c_H|S|$ with probability $negl(\lambda)$.*

Proof. For the sake of contradiction, assume that the lemma does not hold. This implies that there exists a round interval $S = \{i'|i \leq i' < i+s\}$ such that the following event E occurs with non-negligible probability in λ: the adversary computed at least $(1+\sigma)\beta t'c_H|S|$ new blocks until round $i+s$. Using $\mathcal{A}$, we will construct an adversary $\mathcal{A}' = (\mathcal{A}'_1, \mathcal{A}'_2)$ that breaks the moderate hardness (Definition 2) of I with non-negligible probability. For the rest of the proof let $m := (1+\sigma)\beta \cdot t'c_H|S|$.

$\mathcal{A}'$ is going to run internally $\mathcal{A}$ and $\mathcal{Z}$, while at the same time simulating the work honest parties do using the oracles $\mathcal{M}$ and $\mathcal{V}$ provided by the moderate hardness experiment (Definition 2). It is also going to cheaply simulate the RO queries made by $\mathcal{A}_2$ and inject challenges generated by oracle $\mathcal{O}$ for $\mathcal{A}_2$ to solve. By a hybrid argument, we will show that the view of $\mathcal{A}, \mathcal{Z}$ is indistinguishable both in the real and the simulated run, and thus the probability that E happens will be the same in both cases.

Next, we describe the behavior of $\mathcal{A}'$ in more detail—separately for both stages. First, $\mathcal{A}'_1$ sets Ω as the common input for $\mathcal{A}$ and $\mathcal{Z}$, where Ω has been generated using the SNARG CRS generator S. Then, it perfectly simulates honest parties up to round $i-1$ and at the same time runs $\mathcal{A}$ and $\mathcal{Z}$ in a black-box way. To do that, whenever $\mathcal{A}$ queries the RO, it responds with a randomly sampled value; w.l.o.g, we assume, in our analysis, that $\mathcal{A}$ does not repeat the same oracle query twice. Finally, it outputs the contents of the registers of $\mathcal{A}$ and $\mathcal{Z}$. It can do all this, since in the moderate hardness experiment it has polynomial time on λ on its disposal. Note, that up until this point in the eyes of $\mathcal{A}$ and $\mathcal{Z}$ the simulated execution is indistinguishable compared to the real.

For the second stage, $\mathcal{A}'_2(st)$, is first using st to reset $\mathcal{A}$ and $\mathcal{Z}$ to their earlier state. We assume that this can be done efficiently, e.g., by having $\mathcal{A}$ and $\mathcal{Z}$ read from the registers where st is stored whenever they perform some operation on their registers.

Next, we describe how $\mathcal{A}'_2$ simulates honest parties. For each honest party P, it first samples a value l from the geometric distribution with parameter p_2, the number of post-hash queries P has to make until a small post-hash is found. Then, $\mathcal{A}'_2$ samples l values $(t_i)_{i\in[l]}$ from the geometric distribution with parameter p_1. These values are the number of pre-hash queries P has to make until l small pre-hashes are found. Next, $\mathcal{A}'_2$ queries oracle $\mathcal{M}$ l times, with appropriate inputs (G_i, r_i). G_i is determined by the view of P at the round it would supposedly make this computation, and r_i is sampled uniformly from the range of the hash. Finally, after $\sum_{i\in[l]} t_i + l \cdot (\hat{t}/c_H + 1)$ rounds, $\mathcal{A}'_2$ simulates P by diffusing a new block by programming the RO in a way that the challenges r_i, and the pre-hash and post-hash values make the block look valid. It also has to compute a SNARG proof for the $\mathcal{M}$ response that minimizes g and for the winning $\mathcal{M}$ response. The same process is repeated until the end of interval S, to find the next round that P diffuses a new block.

While simulating honest parties, $\mathcal{A}'_2$ has to deal with incoming network traffic coming from $\mathcal{A}_2$. In the best case, $\mathcal{A}'_2$ has to verify a number of SNARG proofs equal to the number of valid input and ranking blocks $\mathcal{A}_2$ can create. While $\mathcal{A}_2$ can potentially spam honest parties with incorrect SNARG proofs, we note that such attacks can be dealt with by requiring parties to produce an additional hash-based PoW based on the SNARG and the related block. The hardness of the PoW should be appropriately set, so that on the one hand the adversary can only produce valid SNARG/PoW pairs at a limited rate, while, on the other hand, honest parties do not spend a large amount of their power on hashing compared to running M. For simplicity, in our analysis we assume that $\mathcal{A}_2$ does not diffuse invalid blocks to the network.

$\mathcal{A}'_2$ also simulates queries made to the RO by $\mathcal{A}$ in a different way than before. Each time $\mathcal{A}$ queries the RO with string w, $\mathcal{A}'_2$ first checks if $H(w)$ has a hard-coded response. If yes, it returns the value it hard-coded earlier. Otherwise, (i) if w corresponds to a pre-hash string, it samples and outputs a random string in the range of the hash and if the value is smaller than T_1, it queries oracle $\mathcal{O}$ and hard-codes the response as the value of[5] $H(w||H(w))$, (ii) if w is a post-hash string, it uses oracle $\mathcal{V}$ to check if the DAG computation described in w is valid, and if yes it stores it. Then, again it samples and returns a random string. In both cases, if the string is smaller than the respective targets T_1, T_2, T_3, the result is stored and used in the simulation of honest parties to determine whether a block is valid.

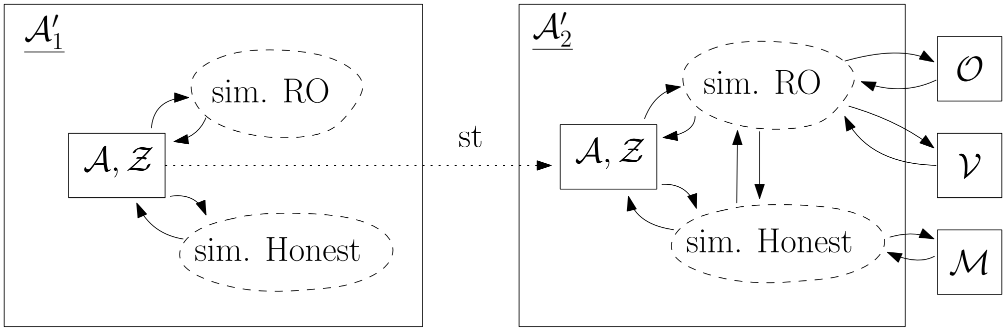


Figure 3: A schematic of the reduction. $\mathcal{A}'$ simulates both the honest parties and queries to the RO. In the second stage of the reduction, $\mathcal{A}'$ makes use of the oracles that it has access to in the MH experiment. It uses: $\mathcal{O}$ to inject new challenges to $\mathcal{A}$, $\mathcal{V}$ to quickly verify the validity of the DAG computations performed by $\mathcal{A}$, $\mathcal{M}$ to quickly simulate the DAG computations performed by honest parties.

Next, we analyze the probability of $\mathcal{A}'$ breaking the moderate hardness of $\mathcal{I}$. First, note that $\mathcal{A}$ and $\mathcal{Z}$ cannot distinguish between the real execution and the simulated one we described above, as queries to the RO are perfectly simulated and honest parties are perfectly simulated using oracle $\mathcal{M}$. Hence, E will occur in the simulated execution with non-negligible probability as well, i.e. $\mathcal{A}$ will compute at least m new blocks starting from round i and up to round $i+s$. This implies that $\mathcal{A}$ will issue at least m valid post-hash queries less than T_2. We are going to use this fact to lower-bound the probability of $\mathcal{A}'$ winning. Due to the law of total probability we have that:

$$\begin{aligned}
&\Pr[\mathcal{A}' \text{ breaks MH}] = \\
&= \Pr[\mathcal{A}' \text{ breaks MH} \wedge E] + \Pr[\mathcal{A}' \text{ breaks MH} \wedge \neg E] \\
&\geq \Pr[\mathcal{A}' \text{ breaks MH}|E] \Pr[E]\,.
\end{aligned}$$

Since $\Pr[E]$ is non-negligible, we next focus on lower bounding $\Pr[\mathcal{A}' \text{ breaks MH}|E]$. Let q_H be the number of queries $\mathcal{A}$ made to the simulated RO. For simplicity, we assume that the cost of simulating an RO query as described above is insignificant compared to c_H, the cost of an RO query in the real execution. Now, let random variable U (resp. U') denote the number of ranking and input blocks generated by honest parties (resp. the adversary) during S. Taking also in account the oracle calls made by $\mathcal{A}'_2$, the total number of steps $\mathcal{A}'_2$ takes in any execution, denoted by $\mathsf{Steps}_{\mathcal{A}'_2}$,

[5] This hard-coding is always possible, since the probability of $\mathcal{A}$ querying $H(w||H(w))$ before querying $H(w)$ is negligible in λ.

is at most:

$$\mathsf{Steps}_{\mathcal{A}_2'} \leq s \cdot tc_H - q_H c_H + (q_{\mathcal{O}} + q_{\mathcal{M}} + q_{\mathcal{V}}) + U c_{\mathsf{P}} + U' c_{\mathsf{V}} \,. \tag{2}$$

Next, we prove that $q_{\mathcal{M}}, q_{\mathcal{V}}, q_{\mathcal{O}}$, i.e., the number of queries made to oracles $\mathcal{M}, \mathcal{V}, \mathcal{O}$, as well as U, U', are upper-bounded as follows:

Claim 1. It holds that $q_{\mathcal{V}}, q_{\mathcal{O}} < 2q_H p_1$, $q_{\mathcal{M}} < 2p_1 ns$, and $U < 4p_1(p_2+p_3)ns, U' < 4p_1(p_2+p_3)ts$ with overwhelming probability in λ.

Proof. We first analyze the bound $q_{\mathcal{M}} < 2p_1 ns$. If some P_j finished running an invocation of M, it should be the case that it computed a small pre-hash before. Hence, we can upper bound $q_{\mathcal{M}}$, by computing an upper bound on the number pre-hashes that P_j successfully computed. Since, in each round, P_j can perform at most one hash query, it succeeds with probability p_1. Let r.v. $X_{i,j}$ be equal to 1 with probability p_1, and 0 otherwise. We can upper bound the total number of small pre-hashes computed by honest parties by $X = \sum_{i \in S, j \in [n]} X_{i,j}$. By an application of the Chernoff bound it easily follows that $\Pr[X \geq 2p_1 ns] < negl(\lambda)$.

In a similar fashion we can upper-bound the number of blocks generated by honest parties (and thus the number of SNARGs computed). Let random variable R_i' be equal to 1, if the i-th invocation of M made by an honest party during S leads to the generation of either a new ranking or a new input block. By our previous bound, we can define $R' := \sum_{i \in [2p_1 ns]} R_i'$, where $\mathbb{E}[R'] \leq 2p_1 ns(p_2+p_3)$ and R' is an upper bound on the number of blocks generated by honest parties during S with overwhelming probability. Again using the Chernoff bound, it follows that $U < 4p_1(p_2+p_3) \cdot ns$ with overwhelming probability in λ.

To establish the bound $q_{\mathcal{V}}, q_{\mathcal{O}} < 2q_H p_1$, first note $q_{\mathcal{V}}, q_{\mathcal{O}} \leq q_{pre}$, by the fact that oracle $\mathcal{O}$ is queried whenever a new valid successful pre-hash query is made, and distinct valid post-hash queries must contain distinct valid small pre-hashes. Let R_i^* be equal to 1 if the i-th distinct hash query was a valid pre-hash. Let $R^* := \sum_{i \in [q_H]} R_i^*$. It holds that $\mathbb{E}[R^*] \leq q_H p_1$. Moreover, random variables in $\{R_i^* | i \in [q_H]\}$ are mutually independent. Thus, we can apply the Chernoff bound (w.l.o.g. assume also that $q_H > \lambda$):

$$\begin{aligned} \Pr[R^* \geq (1+\sigma) q_H p_1] &\leq \Pr[R^* \geq (1+\sigma) q_H p_1] \\ &< e^{-q_H p_1 \sigma^2/3} < negl(\lambda) \,. \end{aligned}$$

It follows that with overwhelming probability $q_{\mathcal{V}}, q_{\mathcal{O}} \leq q_{pre} < 2q_H p_1$. In a similar way as before we can bound U' by $4q_{pre}(p_2+p_3) \leq 4q_H p_1(p_2+p_3) \leq 4p_1(p_2+p_3)ts$, since $q_H < ts$. □

Let D_1 be the event that $q_{\mathcal{V}}, q_{\mathcal{O}} < 2q_H p_1$, $q_{\mathcal{M}} < 2p_1 ns$, and $U < 4p_1(p_2+p_3)ns, U' < 4p_1(p_2+p_3)ts$. If D_1 holds, by Inequality 2 we have that:

$$\begin{aligned} \mathsf{Steps}_{\mathcal{A}_2'} &\leq s \cdot tc_H - q_H c_H + 4q_H p_1 \\ &\quad + sp_1(2n + 4(p_2+p_3)(nc_{\mathsf{P}} + tc_{\mathsf{V}})) \\ &= s \cdot tc_H - q_H(c_H - 4p_1) \\ &\quad + sp_1(2n + 4(p_2+p_3)(nc_{\mathsf{P}} + tc_{\mathsf{V}})) \,. \end{aligned}$$

By the definition of β we have that $1/\beta \leq \frac{c_H}{p_1 p_2}$. Using also the definition of m, we can bound the rate at which $\mathcal{A}'$ queries oracles $\mathcal{V}$ and $\mathcal{M}$:

$$\frac{q_{\mathcal{V}}}{m/((1+\sigma)p_2)} \leq \frac{2p_1 p_2 q_H}{\beta t' c_H s} \leq \frac{2p_1 p_2 st}{p_1 p_2 / c_H \cdot t' c_H s} \leq 2$$

and

$$\frac{q_{\mathcal{M}}}{m/((1+\sigma)p_2)} \leq \frac{2p_1p_2ns}{\beta t' c_H s} \leq \frac{2p_1p_2ns}{p_1p_2/c_H \cdot t' c_H s} \leq 2n/t' \,.$$

For the rest of the proof we use $\hat{\epsilon}(c)$ instead of $\hat{\epsilon}(1 + c/(m/((1+\sigma)p_2), 2, 2n/t')$, i.e., c denotes the number of extra queries made to oracle $\mathcal{O}$. By the definition of m we have that:

$$\begin{aligned} s \cdot tc_H &\leq m/((1+\sigma)\beta) - sp_1(2n + 4(p_2+p_3)(nc_{\mathsf{P}} + tc_{\mathsf{V}})) \\ &\leq \frac{m}{(1+\sigma)p_2}\left((1-\hat{\epsilon}(0))\hat{t} + \left(\frac{1}{(1+\sigma)p_1} + 1\right)(c_H - 4p_1)\right) \\ &- sp_1(2n + 4(p_2+p_3)(nc_{\mathsf{P}} + tc_{\mathsf{V}}))\,. \end{aligned}$$

Putting everything together, we get that:

$$\begin{aligned} \mathsf{Steps}_{\mathcal{A}_2'} \leq &\frac{m}{(1+\sigma)p_2}(1-\hat{\epsilon}(0))\hat{t} \\ &+ \left(\frac{m}{(1+\sigma)p_2}\left(\frac{1}{(1+\sigma)p_1} + 1\right) - q_H\right)(c_H - 4p_1)\,. \end{aligned}$$

Furthermore, due to Lemma 3 and the above inequality, it holds that for any $c \geq 0$:

$$\begin{aligned} \mathsf{Steps}_{\mathcal{A}_2'} \leq &\frac{m}{(1+\sigma)p_2}(1-\hat{\epsilon}(c))\hat{t} \\ &+ \left(\left(\frac{\frac{m}{(1+\sigma)p_2} + c}{(1+\sigma)p_1} + \frac{m}{(1+\sigma)p_2}\right) - q_H\right)(c_H - 4p_1)\,. \end{aligned} \tag{3}$$

It remains to show that the second term in the right-hand side of the inequality is 0. As a first step, we establish lower bounds on the number of pre-hash and post-hash queries to the RO issued by $\mathcal{A}$. Let q_{pre}, q_{post} denote the number of distinct valid small, i.e., smaller than T_1 and T_2, pre-hashes and post-hashes computed by $\mathcal{A}$, respectively (not to be confused with q_{pre}, q_{post} the states of the Markov Chain presented earlier). Let $q_{H,pre}, q_{H,post}$ denote the number of distinct valid pre-hash and post-hash queries made to the RO, respectively.

Claim 2. For $\sigma \in (0,1)$ and any $x \geq \lambda$ it holds that:

- $\Pr[q_{post} \geq x \wedge q_{H,post} \leq \frac{x}{p_2(1+\sigma)}] \leq e^{-\Omega(\lambda)}$;
- $\Pr[q_{pre} \geq x \wedge q_{H,pre} \geq \frac{x}{p_1(1+\sigma)}] \leq e^{-\Omega(\lambda)}$.

Proof. First, we bound the probability that $\mathcal{A}$ generated $x(\geq \lambda)$ or more blocks by querying the RO less than $\frac{x}{p_2(1+\sigma)}$ times with post-hash queries. W.l.o.g., assume that $q_{H,post} = \frac{x}{p_2(1+\sigma)}$. Let R_i be equal to 1 if the i-th post-hash query was successful, i.e., smaller than T_2, and 0 otherwise. Let $R := \sum_{i \in [q_{H,post}]} R_i$, where $q_{H,post}$ is the number of distinct post-hash queries. It holds that $q_{post} \leq R$ and $\mathbb{E}[R] = q_{H,post} \cdot p_2 = x/(1-\sigma)$. By an application of the Chernoff bound it holds that:

$$\Pr[R \geq x] = \Pr[R \geq (1+\sigma)\mathbb{E}[R]] \leq e^{-\frac{\mathbb{E}[R]\sigma^2}{3}} = e^{-\frac{x\sigma^2}{3(1+\sigma)}} \leq negl(\lambda)\,.$$

Hence, the event that $q_{post} \geq x$ and $q_{H,post} \leq \frac{x}{p_2(1+\sigma)}$ happens with probability $negl(\lambda)$.

Similarly, we bound the probability that $\mathcal{A}$ computed more than $x(\geq \lambda)$ pre-hashes in less than $\frac{x}{(1+\sigma)p_1}$ pre-hash RO queries. Let R'_i be equal to 1 if the i-th pre-hash query was successful, then following a similar reasoning as before we have that:

$$\Pr[R' \geq x] \leq \Pr[R' \geq (1+\sigma)\mathbb{E}[R']] \leq e^{-\frac{\mathbb{E}[R']\sigma^2}{3}}$$
$$= e^{-\frac{x\sigma^2}{(1+\sigma)3}} = e^{-\Omega(\lambda)}.$$

The claim follows. □

Let D_2 be the event where $q_{post} \geq m \implies q_{H,post} > \frac{m}{p_2(1+\sigma)}$ and for any $c \leq q_H$, $q_{pre} \geq \frac{m}{p_2(1+\sigma)} + c \implies q_{H,pre} > \frac{\frac{m}{p_2(1+\sigma)}+c}{p_1(1+\sigma)}$. By the previous claim and an application of the union bound we get that $\Pr[D_2] \geq 1 - negl(\lambda)$.

Since each of D_1 and D_2 occurs with overwhelming probability, and E occurs with non-negligible probability, by the definition of conditional probability it follows that $D_1 \wedge D_2 | E$ occurs with overwhelming probability:

$$\Pr[D_1 \wedge D_2|E] = 1 - \Pr[\neg(D_1 \wedge D_2)|E]$$
$$= 1 - \frac{\Pr[\neg(D_1 \wedge D_2) \wedge E]}{\Pr[E]}$$
$$\geq 1 - \frac{\Pr[(\neg D_1) \vee (\neg D_2)]}{\Pr[E]}$$
$$\geq 1 - \frac{\Pr[\neg D_1] + \Pr[\neg D_2]}{\Pr[E]} = 1 - negl(\lambda)$$

where the last inequality follows from an application of the union bound.

We will next show that conditioned on E, $D_1 \wedge D_2$ implies that $\mathcal{A}'$ wins in the MH game. Firstly, E implies that $\mathcal{A}$ issued m valid small post-hash queries. It follows that $q_{post} \geq m$, which in turn due to D_2 implies that $q_{H,post} > \frac{m}{p_2(1+\sigma)}$. Since each pair (G, r) defines a unique output x, it holds that each pre-hash query determines exactly one post-hash query (except with negligible probability). Thus, $q_{pre} \geq q_{H,post} > \frac{m}{p_2(1+\sigma)}$. Let $q_{pre} := \frac{m}{p_2(1+\sigma)} + c$, for $c \geq 0$. For each small pre-hash query, a query to oracle $\mathcal{O}$ must have been issued, thus $q_{\mathcal{O}} := \frac{m}{p_2(1+\sigma)} + c$. Finally, again by D_2, we get that $q_{H,pre} \geq \frac{\frac{m}{p_2(1+\sigma)}+c}{p_1(1+\sigma)}$.

Note now, that the pre-hash and post-has queries are distinct, hence $q_H \geq q_{H,pre} + q_{H,post}$, and by our previous analysis $\mathcal{A}_2'$ has issued at least

$$\frac{\frac{m}{(1+\sigma)p_2} + c}{(1+\sigma)p_1} + \frac{m}{(1+\sigma)p_2}$$

queries to the RO. Moreover, for each distinct small post-hash query $\mathcal{A}_2'$ has extracted a distinct valid DAG computation on a challenge issued by oracle $\mathcal{O}$, i.e., $\frac{m}{(1+\sigma)p_2}$ $(\geq \beta t' c_H |S|/(1+\sigma)p_2 \geq \hat{k})$ valid DAG computations. By our bound on q_H and Inequality 3, it also holds that $\mathsf{Steps}_{\mathcal{A}_2'} \leq (1-\hat{\epsilon}(c))\hat{t}\frac{m}{(1+\sigma)p_2}$. Hence, $\mathcal{A}'$ wins in the MH game, since it has managed to perform $\frac{m}{(1+\sigma)p_2}(\geq \hat{k})$ valid DAG computations in at most $(1-\hat{\epsilon}(c))\hat{t}\frac{m}{(1+\sigma)p_2}$ steps and querying oracle $\mathcal{O}$ at most $\frac{m}{(1+\sigma)p_2}+c$ times. Thus:

$$\Pr[\mathcal{A}' \text{ breaks MH}|E] \geq \Pr[D_1 \wedge D_2|E] \geq 1 - negl(\lambda)$$

which implies that

$$\Pr[\mathcal{A}' \text{ breaks MH}] \geq \Pr[\mathcal{A}' \text{ breaks MH}|E]\Pr[E] > negl(\lambda)$$

since both parts of the product are non-negligible. This contradicts Assumption 1, and the lemma follows. □

5.1.5 Putting everything together

Next, we show that the probability that a uniquely successful round happens is larger than the expected adversarial mining rate per round. Towards this purpose, our next assumption ensures that the computational power advantage of honest parties outperforms the moderate hardness advantage on the DAG computation of the adversary, while at the same time the rate at which blocks are produced is upper bounded.

Assumption 3. There exist $\delta_{\mathsf{MH}}, \delta_{\mathsf{Steps}}$ and $\delta_{tot} \in (0,1)$, such that for sufficiently large $\lambda \in \mathbb{N}$:

- $(n-t)(1-\delta_{\mathsf{Steps}}) \geq t'$ (steps gap)
- $p^*_{\text{rank}} \geq (1-\delta_{\mathsf{MH}})\beta c_H$ (MH gap)
- $\delta_{\mathsf{Steps}} - \delta_{\mathsf{MH}} \geq \delta_{tot}$ (steps vs. MH gap)
- $\delta_{tot} > 3\Gamma \cdot \beta c_H(n-t)$ (bounded block rate) .

As promised, and based on Assumption 3, we prove the following lemma.

Lemma 14. *It holds that $p_{iso} > (1+\delta_{tot})\beta t' c_H$.*

Proof. We have that:

$$\begin{aligned}
p_{\text{iso}} &\geq (n-t)(1-(3\Gamma)p_1p_2)^{(n-t)}p^*_{\text{rank}} && \text{(Bernoulli)}\\
&> (n-t)(1-(3\Gamma)p_1p_2 \cdot (n-t))p^*_{\text{rank}} && \text{(block rate)}\\
&\geq (n-t)(1-\delta_{tot})p^*_{\text{rank}} && \text{(MH gap)}\\
&\geq (n-t)(1-\delta_{tot})(1-\delta_{\mathsf{MH}})\beta c_H && \text{(steps gap)}\\
&\geq \frac{(1-\delta_{tot})(1-\delta_{\mathsf{MH}})}{1-\delta_{\mathsf{Steps}}} t' \beta c_H && \text{(block rate)}\\
&\geq (1+\delta_{tot})\beta t' c_H \,.
\end{aligned}$$

where the first inequality follows also from the fact that $p_1p_2 \leq \beta c_H$. □

Together with the appropriate concentration bounds proved in Lemma 13 and Lemma 12, Lemma 14 is sufficient to apply Theorem 7 for Π, which in turn implies that Π satisfies both Persistence and Liveness with overwhelming probability. Finally, in Appendix B, we argue that under ideal conditions, i.e. optimal MH, small SNARG costs, etc., Π can tolerate any dishonest minority.

A more detailed treatment of useful work completion times. The analysis above calibrates pre-hash hardness as a function of $\hat{t}$, the worst-case completion time of useful work. In certain settings of interest, the time complexity of the useful work task may satisfy a significantly stronger bound with very high probability, in which case this reduced bound can take the place of $\hat{t}$ with only minimal changes to the development above. Specifically, if the time complexity is $\bar{t} < \hat{t}$ except with negligible probability, the value $\bar{t}$ can be uniformly substituted for $\hat{t}$ above with the addition of negligible error terms in the theorems above.

5.2 DPLS security

Executing DPLS in our permissionless PoUW setting potentially implies substantial adversarial participation which can negatively influence the performance of the algorithm in multiple ways. In particular, the adversary does not have to follow Algorithm 1, e.g., by publishing the result of the worst execution of M, instead of the best one.

While the presentation of DPLS is agnostic to adversarial participation, we provide the respective defenses in its embedding PoUW protocol. We present two important quality guarantees of our implementation of DPLS by a PoUW protocol as long as the adversary only controls a minority of the computational power: (i) during any sufficiently large round interval, honest parties contribute new updates proportionally to their relative mining power—in particular, the honest parties contribute more updates than the adversary; (ii) the adversary cannot extensively manipulate the score of its updates, as we enforce each update to additionally include the result of a "random" execution of M from the batch, which is taken in account if the "best" execution has worse score.

Next, we proceed to formalize the aforementioned properties. We start with an observation about the rate at which the adversary produces updates in the context of the DPLS protocol, i.e., the total number of adversarial input and ranking blocks. Let $\bar{Z}(S)$ be the natural extension of $Z(S)$ that refers to both input and ranking blocks, and

$$\bar{\beta} := \frac{p_2 + p_3}{(1 - \hat{\epsilon}(1, 2, 2n/t'))\hat{t} + (\frac{1}{(1+\sigma)p_1} + 1)(c_H - 4p_1)}\,.$$

Following the same steps as in Lemma 13, we get that:

Lemma 15. *For any set of consecutive rounds S, where $|S| \geq \hat{k}(1+\sigma)(p_2+p_3)/(\bar{\beta} \cdot t'c_H)$, it holds that $\bar{Z}(S) \geq (1+\sigma)\bar{\beta} \cdot t'c_H|S|$ with probability $negl(\lambda)$.*

Using the previous lemma, we can show that the number of updates produced by the honest parties outperforms that of the adversary.

Lemma 16. *For any set of consecutive rounds S, where $|S| \geq \hat{k}(1+\sigma)(p_2+p_3)/(\bar{\beta} \cdot t'c_H)$, it holds that honest parties produce more updates than the adversary with overwhelming probability.*

Proof. We can lower bound the rate at which honest blocks are produced based on the stationary probability p^* that an honest party is in an input-block or ranking-block producing state. We can compute p^* in a similar way to how we computed p^*_{rank} in Appendix B. We get that:

$$p^* := \frac{p_2 + p_3}{1/p_1 + 1 + \hat{t}/c_H + c_{\mathsf{P}}/c_H(p_2 + p_3)}\,.$$

Based on Assumption 3, we have that the expected rate at which honest parties produce blocks outperforms that of the adversary, i.e., $(n-t)p^* \geq (1-\delta_{tot})t'\bar{\beta}c_H$. The lemma follows easily. □

Next, we turn our attention to the score of adversarial block updates. Since the blockchain protocol dictates that if the score of the winning attempt is greater that the score of the best attempt in a block, then the winning attempt should be considered in DPLS, we easily get the following corollary.

Corollary 17. *For any set of consecutive rounds S it holds that the score of the i-th adversarial update corresponding to the i-th block in $\bar{Z}(S)$, is larger or equal to the score of the run of M that corresponds to the related small post-hash attempt.*

To negatively influence the quality of its updates, the adversary can at best select to *not* diffuse some of them, at the cost of decreased rewards and influence to the protocol. Otherwise, its updates will not be much worse than uniform runs of M. Finally, notice that due to the pipeline design presented at Section 4.2, the adversary only has a limited window, equal to an epoch, to choose which of its blocks to drop.

5.3 Protocol usefulness

The goal of any PoUW-based blockchain protocol is to be used to solve some external to the blockchain, moderately hard, computational problem. We say that a protocol has a high *usefulness rate* if the total computational work spent to run the blockchain and solve the external problem is not much bigger than just solving the problem with the best algorithm for the setting we consider, denoted by A_{best}.

We study the usefulness rate of our protocol using two metrics. The first metric, U_{eng}, measures the overall ratio of computational steps that the engine directs towards running the DPLS algorithm. Intuitively this metric captures how effective the protocol is as a DPLS engine. We generically calculate U_{eng} as follows (assuming that that the runtime of M is fixed):

$$\begin{aligned}\mathsf{U}_{\text{eng}} &:= \mathbb{E}[\text{DPLS steps per block}]/\mathbb{E}[\text{total steps per block}]\\ &= \frac{\hat{t}/(p_2+p_3)}{\hat{t}/(p_2+p_3)+2\cdot c_{\mathsf{P}}+c_H/(p_1(p_2+p_3))}\\ &= [1+2\cdot c_{\mathsf{P}}\cdot(p_2+p_3)/\hat{t}+c_H/p_1\cdot 1/\hat{t}]^{-1} < 1/2\end{aligned}$$

where the last inequality follows from the fact that the expected time to find a small pre-hash (c_H/p_1) is close to M's solving time $\hat{t}$. Moreover, the smaller the SNARG proving cost (c_{P}) is compared to the expected DPLS work to generate a block ($\hat{t}/(p_2+p_3)$), the closer U_{eng} gets to $1/2$. Hence, by design, our protocol manages to direct approximately $1/2$ of the work spent to generate blocks, to the DPLS algorithm. We note that this rate can be improved by taking in account the exact security bounds regarding grinding attacks against the moderately hard DAG computation (see Remark 2).

The second metric, U_{alg}, compares the complexity of DPLS to algorithm A_{best}. Note, that for U_{alg} we only take into account the DPLS computation steps and no other steps related to the protocol, e.g, hashing, computing SNARGs.

$$\mathsf{U}_{\text{alg}} := \mathbb{E}[\text{total steps of } \mathrm{A}_{\text{best}}\ ]/\mathbb{E}[\text{total steps of DPLS}]$$

U_{alg} cannot be studied generically as it depends on the specific external problem solved as well as the computational model we consider. For example, we expect U_{alg} to be much larger when we consider the best algorithm in a distributed setting compared to the best one in the single machine setting. Instead, in Section 6, we perform an experimental analysis of U_{alg} for a DPLS variant of WalkSAT.

The two metrics that we introduced capture both costs associated with the ledger protocol (hashing and SNARGS) and costs that are induced by the specific algorithm we implement. In fact, in the case where blocks are computed using the honest mining algorithm, the product of the two metrics is a good approximation of the usefulness rate.

Remark 2. (Improved U_{eng}) To be able to prove the security of our protocol given any DAG computation, we set pre-hash hardness (T_1) to be approximately equal to the worst-case time complexity of the exploration algorithm M. While this has the effect that U_{eng} is less that $1/2$, we note that it is possible to improve U_{eng} by adjusting pre-hash hardness based on the MH of the specific DAG computation considered. Specifically, for our security arguments to remain valid, it suffices that:

$$c_H/p_1 := \max_{m,a}\left\{a\cdot(\hat{\epsilon}(1+a,2,2n/t')-\hat{\epsilon}(1,2,2n/t'))\hat{t}\right\}\ .$$

Given now an MH DAG computation where $\hat{\epsilon}(1+a,2,2n/t')-\hat{\epsilon}(1,2,2n/t')\leq\delta a$, for some $\delta\in[0,1]$, we get that

$$\mathsf{U}_{\text{eng}} = [1+2\cdot c_{\mathsf{P}}\cdot(p_2+p_3)/\hat{t}+\delta]^{-1}$$

which, for favorable parameters, i.e, $c_{\mathsf{P}} << \hat{t}/(p_2 + p_3)$ and $\delta \approx 0$, implies that U_{eng} is close to 1.

6 Applications

In this section we describe the applications of our protocol to concrete, practice-relevant optimization problems. First, we propose algorithms from the literature that can be cast as instantiations of the generic DPLS algorithm, followed by a discussion of several real-world problems that can be solved by our protocol. Finally, we evaluate the performance of a concrete example of a DPLS variant of WalkSAT.

6.1 Suitable algorithms

A prime example for DPLS is the WalkSAT algorithm [55, 56], which we described in Section 3. In fact, most well-known stochastic-local-search (SLS) algorithms [34] can be mapped to DPLS as follows: The Init function provides the initial information needed, e.g., a number of different starting locations for parallel executions of WalkSAT. Given the current location, M is set to explore a single location in its neighborhood and any randomness needed is provided by the seed. Consequently, Update can be interpreted as exploring one or more locations in the neighborhood depending on the post-hash parameters p_2 and p_3, and then returning the one that maximizes the scoring function g. This location can then serve as the next point in the random walk. Note that DPLS allows both for multiple-walk parallelism, in the sense that multiple walks can be executed concurrently by different miners coordinating through the blockchain, and for parallelism in each walk, i.e., before a step is made multiple locations in the neighborhood are explored [58].

As argued above, DPLS is rather expressive. Still, we expect even better performance when the following two conditions are met: (i) the number of neighborhood locations explored in each step is large, such that the expected DPLS work to generate a block is a lot larger than the work to create the respective SNARG proof—for the benefit of usefulness, as explained in Section 5.3, and, (ii) the total neighborhood size is sufficiently large, such that parties do not explore the same locations due to desynchronization and the fact that the locations searched are randomly determined. Furthermore, algorithms that search larger neighborhoods may also be better candidates regarding moderate hardness, since a small neighborhood makes precomputation attacks potentially easier.

A subclass of SLS algorithms that has these two characteristics are Very Large Scale Neighborhood search algorithms [3], where the algorithm (partially) searches a very large neighborhood before making its next step. For example, in the Large Neighborhood Search heuristic [53], exploring a neighborhood involves destroying part for the solution and then reconstructing it in a greedy way, to find a new maximum. Depending on the way the reconstruction happens, the time it takes to obtain a new solution can vary, e.g., if during reconstruction a Linear Programming problem is solved, the whole process may be in the order of seconds. Another example is variable-depth search methods [40, 3], where the k-exchange neighborhood is searched partially in an effort to find solutions close to the local minimum of the neighborhood. As before, parameter k can be used to adjust the time it takes for a single neighbor search.

6.2 Real-world problems

We now present some suitable real-world optimization problems for our protocol. One of the competitive advantages of our system is that it can provide access to huge amounts of computational power; Bitcoin's current power consumption compares to that of a medium-size country. An additional benefit is public auditability: the results of our algorithm executions are publicly agreed and come with a publicly verifiable correctness proof.

These properties are especially appealing for governance problems. For example, in 2016 the US Federal Communications Commission (FCC) ran a radio spectrum auction [15] that involved solving large instances of the station repacking problem [22]. The auction took place in rounds, and organizers set a constraint of one minute to solve each instance, as the solution of different instances directly influenced the subsequent phases of the auction. The solving process followed by FCC involved an SLS algorithm as its main component. Speed and public auditability would be strong arguments in favor adopting our framework in this case.

A second example is the residents/hospital-matching problem. Initially, both the applicants and the hospitals declare the order at which they prefer each other. Then, the relevant agency, e.g., NRMP in the USA, tries to find a matching between residents and hospitals minimizing vacant positions while maximally respecting the preference order set by the involved parties. The problem is a good candidate since several SLS algorithms [45, 54] have been proposed for solving it, and also due to the fact that there is public interest that the matching is obtained in a fair manner.

Another relevant class of hard optimization problems is athletic-events scheduling. As an example, the US National Football League uses an SLS algorithm [19] to create the schedule of each season [47]. The associated optimization problem is generated by taking in consideration how much each team has to travel, stadium availability and broadcasting channels' interests, among others. The hardness of the problem and the fact that multiple parties with conflicting interests are involved in the process makes our protocol a good fit.

6.3 Concrete example

In this section, we evaluate the performance of the DPLS variant of WalkSAT described in Section 3. We stress that our objective here is to provide a proof-of-concept of our ideas, and not to give a solution that can compete with the state-of-the-art.

On a high level each thread of DPLS runs the WalkSAT procedure (Section 3.1, Algorithms 2 and 3) for a bounded number of flips. The scoring function, used by the Update procedure to pick the best among these threads, outputs the number of satisfied clauses in the final configuration of the walk. In addition, multiple such walks are run concurrently, with parties selecting at random which walk to work on.

We experimentally evaluated the performance of the algorithm compared to the WalkSAT algorithm. Our test set consisted of Automated Planning problem instances [28], namely Blocks World Planning; more experiments on different data sets were executed showing similarly good behavior as demonstrated here. In more detail, the problem consists of finding a plan to move blocks in a table from an initial configuration to a goal configuration in a bounded number of moves, where the only move allowed is moving a block from the top of a pile of blocks either to the table or on top of another pile. For more information about the problem and the selected instances (instances *bw_ large.b, bw_ large.c, bw_ large.d* were used) we refer the reader to [1, 33].

For the sake of comparability, we choose to partially solve each of the three given instances by allowing solutions to leave at most T clauses unsatisfied, for some parameter T. This way we can obtain a similar hardness level for the three instances of approximately 200k to 300k flips when running plain WalkSAT. An upper bound of 10^6 flips is used in all our experiments. Following Figure 4, we determined T to be 1, 13, and 33, for the different instances.

We first examine the behavior of our algorithm assuming a single starting point and that all updates are honestly produced one after another. As described earlier, our algorithm proceeds in epochs, where, in each epoch, a number of different threads run WalkSAT for a bounded number

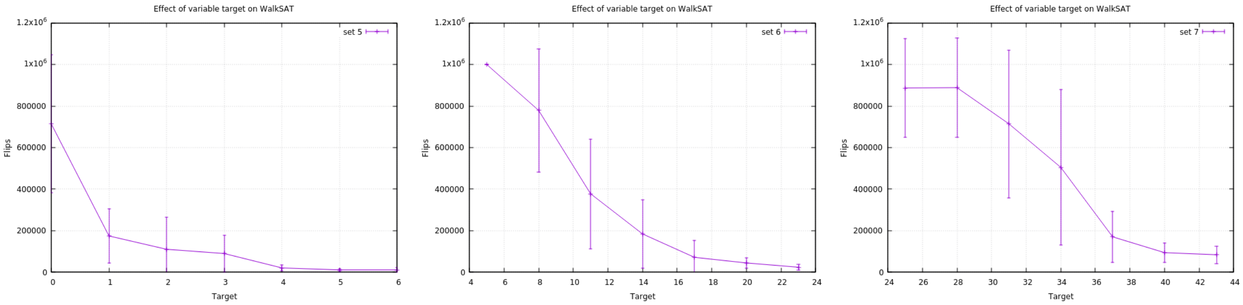


Figure 4: Hardness of planning instances. On the x-axis we set the target T, while on the y-axis we count the total number of flips that it took WalkSAT to solve the problem. The standard deviation of our experiments is shown as a vertical bar.

of flips. Here, we fix the total number of flips per epoch to 10^5, and study what happens when the (expected) number of threads changes. The quantities of interest are: (i) the total number of flips our algorithm made before reaching a solution, and (ii) the depth that the longest walk in our search reached. In Figure 5, we see that while increasing the number of threads has a negative effect on the total number of flips, it also leads to decreased depth, thus positively affecting how early a solution is found. For the rest of the experiments we set the (expected) number of threads to 20 and the flips per thread to 5000.

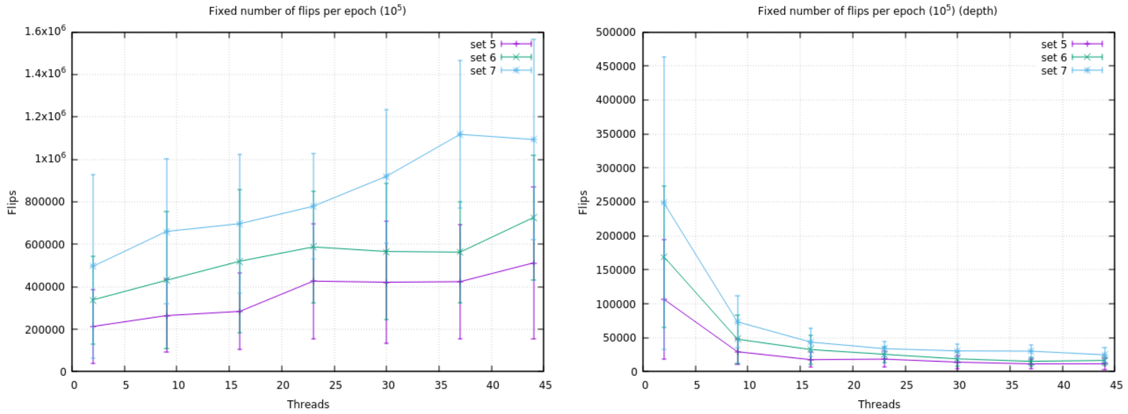


Figure 5: Varying the number of threads of DPLS for a fixed number of flips per epoch.

Next, in Figure 6, we compare the performance of our algorithm to that of WalkSAT, and see that our algorithm is approximately half as fast. Note, that this factor is not close to 1/20 (for 20 threads), implying that the extra threads have a noticeable effect in how fast a solution is found.

The next set of tests is concerned with the effect of faults in our algorithm. The type of faulty behavior we consider is the adversary picking a random run of function M instead of the best one when producing a new update. This behavior is quite realistic, as it is easy to detect if the adversary publishes a solution a lot worse than a randomly sampled one, since each block additionally contains the run of M that led to a small post-hash. The main thing we observe in Figure 6, is that the performance of our algorithm deteriorates faster after the 50% fault mark, suggesting some robustness to faults. Moreover, with faults at 50%, the performance of the algorithm suffers a factor of less than two.

Next, we examine the effect of multiple updates being produced in parallel in our algorithm.

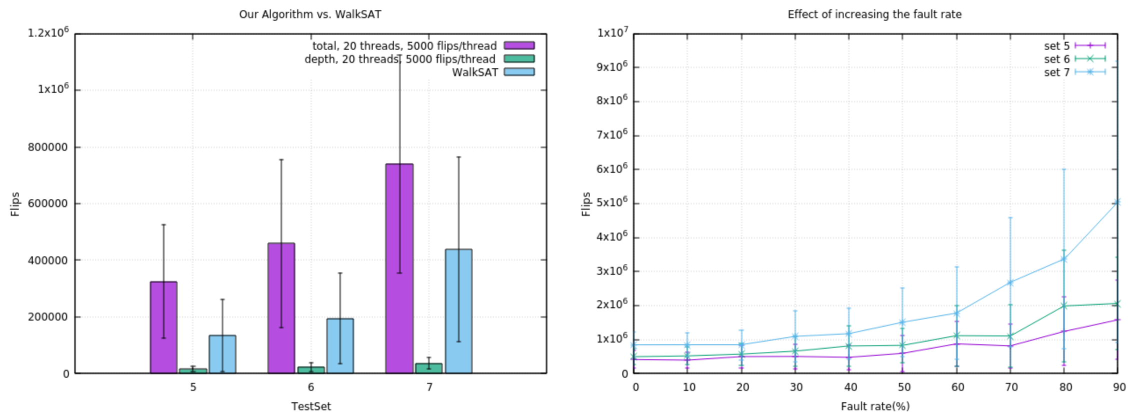


Figure 6: DPLS vs. WalkSAT and the effect of increasing the fault rate on DPLS.

Figure 7 depicts the performance of DPLS when multiple starting points are considered. Again, we are in the setting where all updates are computed honestly one after another. We observe that while increasing the number of starting points (x-axis) leads to an increased total number of flips, this increase is not directly proportional to the number of starting points; doubling the number of starting points does not double the number of steps. This implies that parallelization speeds up our search, in the sense that depth is decreased.

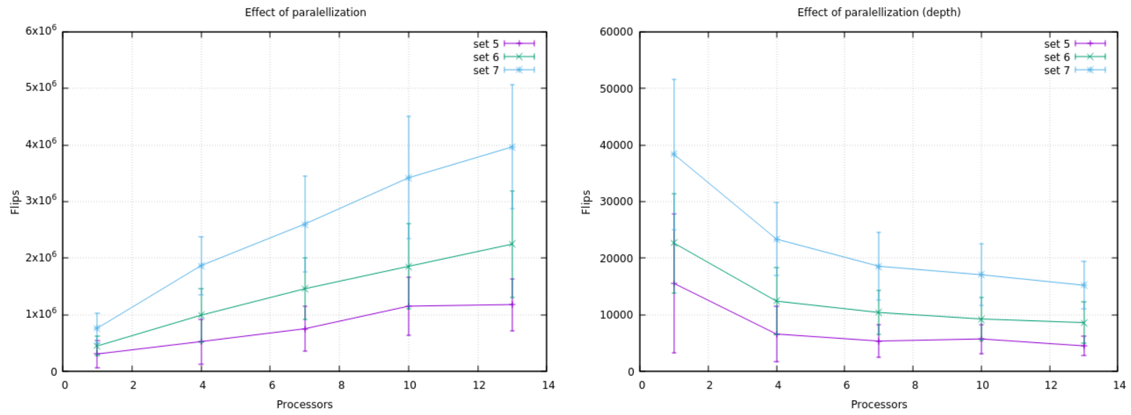


Figure 7: Varying the number of starting points of DPLS.

Finally, in Figure 8, we examine the effect of increased SNARG costs on DPLS. We plot the total cost of running DPLS with a single starting point assuming that the SNARG cost depends linearly[6] on the cost of the single thread of WalkSAT that is proven correct. The multiplicative constant appears on the horizontal axis of the graph. Observe that for our test cases the total cost doubles approximately when the SNARG costs 20 times more than the computation proven correct, i.e., close to the number of threads in our DPLS execution. This confirms our theoretical analysis where we showed that the larger the number of threads, the less significant the cost of SNARG is compared to the total cost of our algorithm.

[6] Spartan [57] is an example of a SNARG system with this property.

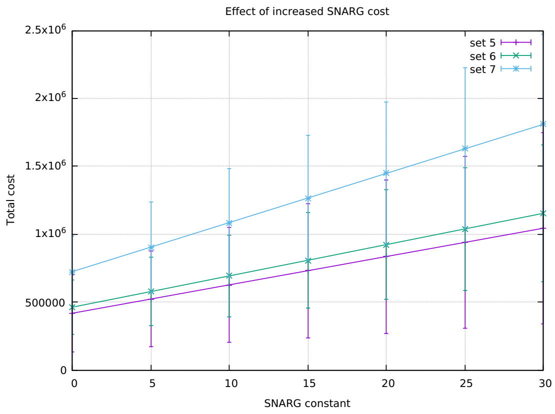


Figure 8: The effect of increased SNARG cost on the total running time of DPLS.

Acknowledgments. We thank Laurent Michel for providing us with valuable information about state-of-the-art stochastic local-search algorithms and their application to real-world problems.

References

[1] Sat-encoded blocks world planning problems. `https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/PLANNING/BlocksWorld/descr.html`.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of prams. *Theor. Comput. Sci.*, 71(1):3–28, 1990.

[3] R. K. Ahuja, Ö. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002.

[4] D. Aldous and J. A. Fill. Reversible markov chains and random walks on graphs, 2002. Unfinished monograph, recompiled 2014, available at `http://www.stat.berkeley.edu/~aldous/RWG/book.html`.

[5] M. Andrychowicz and S. Dziembowski. Distributed cryptography based on the proofs of work. Cryptology ePrint Archive, Report 2014/796, 2014. `http://eprint.iacr.org/`.

[6] C. Badertscher, P. Gazi, A. Kiayias, A. Russell, and V. Zikas. Consensus redux: Distributed ledgers in the face of adversarial supremacy. *IACR Cryptol. ePrint Arch.*, 2020:1021, 2020.

[7] C. Badertscher, U. Maurer, D. Tschudi, and V. Zikas. Bitcoin as a transaction ledger: A composable treatment. In J. Katz and H. Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 324–356, Cham, 2017. Springer International Publishing.

[8] A. Baldominos and Y. Saez. Coin. ai: A proof-of-useful-work scheme for blockchain-based distributed deep learning. *Entropy*, 21(8):723, 2019.

[9] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan. Proofs of work from worst-case assumptions. In *Annual International Cryptology Conference*, pages 789–819. Springer, 2018.

[10] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73, 1993.

[11] D. Boneh, J. Bonneau, B. Bünz, and B. Fisch. Verifiable delay functions. In *Advances in Cryptology - CRYPTO 2018.*

[12] K. Chatterjee, A. K. Goharshady, and A. Pourdamghani. Hybrid mining: exploiting blockchain's computational power for distributed problem solving. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 374–381, 2019.

[13] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 5-7, 2010. Proceedings.*

[14] F. Coelho. An (almost) constant-effort solution-verification proof-of-work protocol based on merkle trees. Cryptology ePrint Archive, Report 2007/433, 2007. `https://eprint.iacr.org/2007/433`.

[15] F. C. Commission. Expanding the economic and innovation opportunities of spectrum through incentive auctions. https://docs.fcc.gov/public/attachments/FCC-14-50A1.pdf, 2014.

[16] A. Coventry. Nooshare: A decentralized ledger of shared computational resources. `https://www.semanticscholar.org/paper/NooShare-%3A-A-decentralized-ledger-of-shared-Coventry/4616e9784009f1274a7f4bf6087a6870cd62f122`, 2012.

[17] P. Daian, R. Pass, and E. Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake. In *International Conference on Financial Cryptography and Data Security*, pages 23–41. Springer, 2019.

[18] B. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain. In *Advances in Cryptology - EUROCRYPT 2018.*

[19] B. N. Dilkina and W. S. Havens. The us national football league scheduling problem. In *AAAI*, pages 814–819, 2004.

[20] M. Dotan and S. Tochner. Proofs of useless work–positive and negative results for wasteless mining systems. *arXiv preprint arXiv:2007.01046*, 2020.

[21] S. Dziembowski, S. Faust, V. Kolmogorov, and K. Pietrzak. Proofs of space. In *Annual Cryptology Conference*, 2015.

[22] A. Fréchette, N. Newman, and K. Leyton-Brown. Solving the station repacking problem. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[23] Gapcoin. Gapcoin, 2014. `https://gapcoin.org/`.

[24] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology - EUROCRYPT 2015.*

[25] J. A. Garay, A. Kiayias, and G. Panagiotakos. Consensus from signatures of work. In *Topics in Cryptology – CT-RSA 2020.*

[26] J. A. Garay, A. Kiayias, and G. Panagiotakos. Blockchains from non-idealized hash functions. In *Theory of Cryptography*, 2020.

[27] P. Gazi, A. Kiayias, and A. Russell. Tight consistency bounds for bitcoin. In J. Ligatti, X. Ou, J. Katz, and G. Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 819–838. ACM, 2020.

[28] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: theory and practice.* Elsevier, 2004.

[29] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[30] J. Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology – EUROCRYPT 2016.*

[31] J. Groth, M. Kohlweiss, M. Maller, S. Meiklejohn, and I. Miers. Updatable and universal common reference strings with applications to zk-snarks. In *Annual International Cryptology Conference.*

[32] N. Gupta and D. S. Nau. On the complexity of blocks-world planning. *Artif. Intell.*, 56(2-3):223–254, 1992.

[33] H. H. Hoos and T. Stützle. Satlib: An online resource for research on sat. *Sat*, 2000:283–292, 2000.

[34] H. H. Hoos and T. Stützle. *Stochastic local search: Foundations and applications*. Elsevier, 2004.

[35] H. Kautz, B. Selman, and D. McAllester. Walksat in the 2004 sat competition. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, 2004.

[36] T. Kerber, A. Kiayias, and M. Kohlweiss. Mining for privacy: How to bootstrap a snarky blockchain. Cryptology ePrint Archive, Report 2020/401, 2020. `https://eprint.iacr.org/2020/401`.

[37] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Annual International Cryptology Conference*, pages 357–388. Springer, 2017.

[38] S. King. Primecoin: Cryptocurrency with prime number proof-of-work, 2013.

[39] A. Lihu, J. Du, I. Barjaktarevic, P. Gerzanics, and M. Harvilla. A proof of useful work for artificial intelligence on the blockchain. *arXiv preprint arXiv:2001.09244*, 2020.

[40] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[41] A. F. Loe and E. A. Quaglia. Conquering generals: an np-hard proof of useful work. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, pages 54–59, 2018.

[42] M. Maller, S. Bowe, M. Kohlweiss, and S. Meiklejohn. Sonic: Zero-knowledge snarks from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2111–2128, 2019.

[43] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing bitcoin work for data preservation. In *2014 IEEE Symposium on Security and Privacy*, pages 475–490. IEEE, 2014.

[44] T. Moran and I. Orlov. Simple proofs of space-time and rational proofs of storage. In *Advances in Cryptology – CRYPTO 2019*.

[45] D. Munera, D. Diaz, S. Abreu, F. Rossi, V. Saraswat, and P. Codognet. Solving hard stable matching problems via local search and cooperative parallelization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 29, 2015.

[46] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008.

[47] NFL. Creating the nfl schedule. https://operations.nfl.com/gameday/nfl-schedule/creating-the-nfl-schedule/.

[48] C. G. Oliver, A. Ricottone, and P. Philippopoulos. Proposal for a fully decentralized blockchain and proof-of-work algorithm for solving np-complete problems. *arXiv preprint arXiv:1708.09419*, 2017.

[49] C. H. Papadimitriou and J. D. Ullman. A communication-time tradeoff. *SIAM J. Comput.*, 16(4):639–646, 1987.

[50] S. Park, A. Kwon, G. Fuchsbauer, P. Gaži, J. Alwen, and K. Pietrzak. Spacemint: A cryptocurrency based on proofs of space. In *International Conference on Financial Cryptography and Data Security*, 2018.

[51] R. Pass, L. Seeman, and abhi shelat. Analysis of the blockchain protocol in asynchronous networks. Cryptology ePrint Archive, Report 2016/454.

[52] R. Pass and E. Shi. Fruitchains: A fair blockchain. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 315–324, 2017.

[53] D. Pisinger and S. Ropke. Large neighborhood search. In *Handbook of metaheuristics*, pages 99–127. Springer, 2019.

[54] M. Sartori. A local search algorithm for matching hospitals to residents. 2013.

[55] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, page 337âĂŞ343, USA, 1994.

[56] B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1996.

[57] S. Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, 2020.

[58] M. G. A. Verhoeven and E. H. Aarts. Parallel local search. *Journal of heuristics*, 1(1):43–65, 1995.

[59] F. Zhang, I. Eyal, R. Escriva, A. Juels, and R. Van Renesse. {REM}: Resource-efficient mining for blockchains. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1427–1444, 2017.

[60] W. Zheng, X. Chen, Z. Zheng, X. Luo, and J. Cui. Axechain: A secure and decentralized blockchain for solving easily-verifiable problems. *arXiv preprint arXiv:2003.13999*, 2020.

A The Full Protocol

Our protocols follows the same high-level structure (Algorithm 6) as Bitcoin, formalized in [24]. In each round, miners fetch any new chains diffused in the network, and pick the longest valid chain among the ones they have received (Algorithm 7 and 8). Then, based on the contents of the selected chain and other messages received from the network, they select their block input m, as well as information related to the DAG computation they are going to perform (Λ, G, z), and run the PoUW generation function (Algorithm 5). If a new block is produced, it is diffused in the network.

Algorithm 6 The main function of our protocol, parameterized by the *input contribution function* I$(\cdot)$ and the *chain reading function* R$(\cdot)$.

```
$\mathcal{C} := B_{\mathsf{Gen}}$                                    ▷ Initialize to the genesis block $B_{\mathsf{Gen}}$
$st := \varepsilon, round := 0$
while TRUE do
    $\tilde{\mathcal{C}} \leftarrow \mathsf{maxvalid}(\mathcal{C}$, any chain $\mathcal{C}'$ found in RECEIVE())
    $(st, m, \Lambda, G) \leftarrow \mathrm{I}(st, \tilde{\mathcal{C}}, round,$ INPUT(), RECEIVE())
    $\mathcal{C}_{\mathsf{new}} \leftarrow$ POUW$(\tilde{\mathcal{C}}, m, \Lambda, G)$
    if $\mathcal{C} \neq \mathcal{C}_{\mathsf{new}}$ then
        $\mathcal{C} \leftarrow \mathcal{C}_{\mathsf{new}}$
        DIFFUSE$(\mathcal{C})$
    $round \leftarrow round + 1$
    if INPUT() contains READ then
         write $\mathrm{R}(\mathbf{m}_{\mathcal{C}})$ to OUTPUT()
```

Algorithm 7 The function that finds the “best” chain, parameterized by function max$(\cdot)$. The input is $\{\mathcal{C}_1, \ldots, \mathcal{C}_k\}$.

```
function maxvalid$(\mathcal{C}_1, \ldots, \mathcal{C}_k)$
    $temp := \varepsilon$
    for $i = 1$ to $k$ do
        if validate$(\mathcal{C}_i)$ then
            $temp := \max(\mathcal{C}, temp)$
    return $temp$
```

The protocol is parametrized by functions $V(\cdot), R(\cdot), I(\cdot)$. The content validation predicate $V(\langle x_1, \ldots, x_m \rangle)$ is true if its input is a valid ledger, i.e., it is in $\mathcal{L}$. Valid ledgers for our protocol contain transactions that move funds around, exactly as in Bitcoin, and valid or invalid input blocks as described in Section 4.1. With foresight, we allow the existence of invalid input blocks in valid ledgers, in order to be able to decouple the verification of the "header" of the ranking block from its contents; verifying the header includes verifying the pre-hash, the post-hash, and the SNARG, and does not include verifying that G was computed correctly. Note, that just verifying the headers of the ranking blocks is sufficient to achieve consensus, according to our analysis in Section 5.1. Next, the chain reading function $R(\mathcal{C})$ returns the contents of the chain if they constitute a valid ledger, otherwise it is undefined. Finally, the input contribution function $I(st, \mathcal{C}, round, \textsc{Input}(), \textsc{Receive}())$ returns (st, m, Λ, G), where st is the new state and m is the largest subsequence of transactions and valid input blocks in the input and receive tapes that constitute a valid ledger, with respect to the contents of the chain the party already has, together with a randomly generated neutral transaction to avoid collisions.

$I(\cdot)$ also encodes part of the DPLS algorithm logic, as it is used to select Λ and G. In more detail, Λ is selected according to the scheduling procedure, as discussed in Section 4.2. G is selected to be the transcript of Λ extracted from C plus any valid points/input-blocks that the miner has already received and verified. While we allow a valid ledger to contain invalid input blocks, function I should only take into account valid input blocks where the pre and post hashes are small enough and the DAG computation as well as the transcript used is correctly computed based on the referenced chain pointer. Finally, we assume that I changes Λ and G only after either the miner successfully computes a new block, or a new epoch starts according to Section 4.2. While it is easy to extend our protocol to handle different transcripts at every round, this would lead to a more complicated Update function.

Algorithm 8 The validate procedure, parameterized by the hash function $H(\cdot)$, the *chain validation predicate* $V(\cdot)$, and the verification algorithm V of SNARG. The input is $\mathcal{C}$.

```
function validate($\mathcal{C}$)
    $b \leftarrow \mathrm{V}(\mathbf{m}_{\mathcal{C}}) \wedge (\mathrm{tail}(\mathcal{C}) = B_{\mathtt{Gen}})$
                                                  ▷ $\mathbf{m}_{\mathcal{C}}$ describes the contents of chain $\mathcal{C}$.
    if $b =$ False then
        return $b$
    $s' := H(B_{\mathtt{Gen}})$
    $\mathcal{C} \leftarrow \mathcal{C}^{1\rceil}$                                  ▷ Remove the genesis from $\mathcal{C}$
    while $(\mathcal{C} \neq \epsilon \wedge b = \mathrm{True})$ do
        $\langle(s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b), ...$
            $...(s, m, com, \Lambda, z, r, x'), \pi\rangle \leftarrow \mathrm{tail}(\mathcal{C})$
        $s'' := H(\mathrm{tail}(\mathcal{C}))$
        $h := H(s, m, com, \Lambda, G, z, r)$
        $r' := H(s, m, com, \Lambda, G, z, r, h)$
        $h' := H(s, m, com, \Lambda, G, z, r, h, x')$
        $b_0 := (s = s') \wedge (h < T_1) \wedge (h' < T_2)$
        $b_1 := (H(s_b, m_b, com_b, \Lambda_b, G_b, z_b, r_b, x'_b) = com)$
        $b_2 := \mathrm{SNARG}.\mathsf{V}(\Sigma, ((\Lambda, G, z, r', x'), ...$
            $...(\Lambda_b, G_b, z_b, r'_b, x'_b)), \pi)$
        if $(\bigwedge_{i \in \{0,\dots,2\}} b_i)$ then
            $s' \leftarrow s''$                                   ▷ Retain hash value
            $\mathcal{C} \leftarrow \mathcal{C}^{1\rceil}$                             ▷ Remove the tail from $\mathcal{C}$
        else
            $b \leftarrow$ False
    return $b$
```

B Security under Ideal Conditions

In this section, we argue that under ideal conditions, Π can tolerate any dishonest minority, i.e., the adversary can corrupt close to $1/2$ of the parties. By ideal conditions we mean that $\sigma, p_3, c_{\mathsf{P}}, c_V, \hat{\epsilon}, p_1/c_H$ are all close to 0.

First, we obtain a lower bound on the stationary probability of q_{rank}. In general, the stationary probability of a state in a (finite, irreducible, positive recurrent) Markov chain is the inverse of the mean recurrence time. In the case of q_{rank}, this mean recurrence time is easy to determine. Let R_{rank} denote the mean recurrence time for q_{rank}. Note that since the transition from q_{pre} to q_{rank} is zero-cost (we only count the hashing, SNARK and M costs), R_{rank} is equal to the mean time to transition to q_{rank} from q_{pre} (as there is a single outgoing transition from q_{rank}). So we focus on this mean transition time which we simply call R. Considering that returns to q_{pre} are of four kinds, we can expand R as follows:

$$\begin{aligned} R = {} & (1-p_1)(1+R) + p_1(1-p_2-p_3)[\bar{t}/c_H + 2 + R] \\ & + p_1p_2((\bar{t}+c_{\mathsf{P}})/c_H + 2) + p_1p_3((\bar{t}+c_P)/c_H + R + 2) \end{aligned}$$

(where $\bar{t}$ is the mean useful work time and c_P is treated as a constant). Thus

$$\begin{aligned} R = {} & \frac{(1-p_1) + p_1[(1-p_2-p_3)[\bar{t}/c_H + 2]}{p_1p_2} \\ & + \frac{(p_2+p_3)((\bar{t}+c_{\mathsf{P}})/c_H + 2)]}{p_1p_2} \end{aligned}$$

and $R_{\mathsf{rank}} = R$.

Next, we proceed to prove our lemma. We have that $p^*_{\mathsf{rank}} = 1/R_{\mathsf{rank}} = \frac{p_2}{1/p_1+\hat{t}/c_H+1+(p_2+p_3)c_{\mathsf{P}}/c_H}$. For $\sigma, np_1/c_H, c_P/c_H, c_V/c_H, \hat{\epsilon} << 1$, we have that $p^*_{\mathsf{rank}} \approx \frac{p_2}{\hat{t}/c_H+1/p_1+1}$. Similarly, $\beta c_H \approx \frac{p_2}{\hat{t}/c_H+1/p_1+1}$. Hence, $p^*_{\mathsf{rank}} \approx \beta c_H$, and thus $\delta_{MH} \approx 0$. By setting δ_{tot} close to 0, we get that $\delta_{\mathsf{Steps}}(= \delta tot + \delta_{MH})$ can be close to 0. Given also that under our assumptions $t' \approx t$, it is implied that $n - t \approx t$, or $t \approx n/2$, i.e., any dishonest majority can be tolerated.

C Characteristic-String Analysis: Forks, Margin, and Common Prefix

In this appendix we survey the general theory of forks and margin that is used to prove Theorems 6 and 7 of the main body. A full account can be found in [37, 6].

The basic bookkeeping tool of the theory is the Δ-fork. A Δ-fork is a graph-theoretic abstraction that maintains the topology of the blocktree constructed by an execution of a Nakamato-style consensus protocol. In particular, it determines the predecessor of each generated block, the round in which each block is produced, and whether or not the block producer was honest or adversarial. These details are sufficient for capturing the elementary liveness and persistence properties of the execution. The parameter Δ determines the time horizon at which the longest chain rule is guaranteed to operate for honest players: historically, this has been determined by network delays, which is to say that any honestly generated block must have depth exceeding that of any block honestly generated Δ rounds previously. In our setting, the Δ parameter (called Γ in the main body) has actually been used to reflect some further delays arising due to the Markov chain that dictates mining dynamics. In any case, the basic role that the parameter plays in the approach is identical.

Definition 18 (PoW Δ-fork)**.** Let Δ be a positive integer and $L \in \mathbb{N}$. A *PoW Δ-fork* for the string $w \in (\mathbb{N}^2)^L$ is a directed, rooted tree $F = (V, E)$ with a pair of functions

$$\mathsf{l}_{\#} : V \to \mathbb{N} \qquad \text{and} \qquad \mathsf{l}_{\mathsf{type}} : V \to \{\mathsf{h}, \mathsf{a}\}$$

satisfying the axioms below. Edges are directed "away from" the root so that there is a unique directed path from the root to any vertex. The value $\mathsf{l}_{\#}(v)$ is referred to as the *label* of v. The value $\mathsf{l}_{\mathsf{type}}(v)$ is referred to as the *type* of the vertex: when $\mathsf{l}_{\mathsf{type}}(v) = \mathsf{h}$, we say that the vertex is *honest*; otherwise it is *adversarial.*

(i) the root $r \in V$ is honest and has label $\mathsf{l}_{\#}(r) = 0$;

(ii) the sequence of labels $\mathsf{l}_{\#}()$ along any directed path is non-decreasing;

(iii) if $w_i = (h_i, a_i)$, there are exactly h_i honest vertices with the label i and no more than a_i adversarial vertices of F with the label i;

(iv) for any pair of honest vertices v, w for which $\mathsf{l}_{\#}(v) + \Delta \leq \mathsf{l}_{\#}(w)$, $\text{len}(v) < \text{len}(w)$, where len() denotes the depth of the vertex.

An advantage of this formalism is that it can easily reflect a persistence failure: two chains, each of equal length, that disagree in a particular round ℓ.

C.1 Fork notation, closure

We write $F \vdash_{\Delta} w$ to indicate that F is a Δ-fork for the string w. If $F' \vdash_{\Delta} w'$ for a prefix w' of w, we say that F' is a *subfork* of F, denoted $F' \sqsubseteq F$, if F contains F' as a consistently-labeled subgraph. A fork $F \vdash_{\Delta} w$ is *closed* if all leaves are honest. By convention the trivial fork, consisting solely of a root vertex, is closed. The *closure* of a fork F, denoted $\overline{F}$, is the maximal closed subfork of F.

C.2 Tines

A path in a fork F originating at the root is called a *tine* (note that tines do not necessarily terminate at a leaf). For a vertex v in F, $F(v)$ denotes the tine in F terminating in v. Given this one-to-one correspondence between vertices and tines of a fork, we routinely overload notation so that it applies to both tines and vertices. For example, we let $\text{len}(T)$ denote the *length* of the tine T, equal to the number of edges on the path; recall that $\text{len}(v)$ also indicates the depth of the vertex v. To emphasize the fork from which v is drawn, we sometimes write $\text{len}_F(v)$. We further overload len() to apply to forks: $\text{len}(F)$ denotes the length of the longest tine in a fork F. A tine is called *honest* if it terminates in some vertex v with $\mathsf{l}_{\mathsf{type}}(v) = \mathsf{h}$.

For two tines T, T' of a fork F, we write $T \sim_{\ell} T'$ if the two tines share a vertex with a label greater or equal to ℓ. Intuitively, $T \sim_{\ell} T'$ guarantees that the respective blockchains agree on the state of the ledger up to time ℓ. Looking ahead, the adversary can only make two honest parties disagree on the state of the ledger up to time ℓ if she makes them hold two chains corresponding to tines for which $T \not\sim_{\ell} T'$.

C.3 Fork trimming; dominance

For a characteristic string $w = w_1 \dots w_n \in \Sigma^n$ and a positive integer k, we let $w_{\lfloor k} = w_1 \dots w_{n-k+1}$ denote the string obtained by removing the last $k - 1$ symbols. For a fork $F \vdash_{\Delta} w_1 \dots w_n$ we let $F_{\lfloor k} \vdash_{\Delta} w_{\lfloor k}$ denote the fork obtained by retaining only those vertices labeled from the set

$\{1, \ldots, n-k+1\}$. Observe that honest tines appearing in $F_{\lfloor\Delta}$ are those that are necessarily visible to honest players at a round just beyond the last one described by the characteristic string. We say that a tine T in F is Δ-*dominant* if $\mathrm{len}(T) \geq \mathrm{len}(\overline{F_{\lceil\Delta}})$ and simply call it *dominant* if Δ is clear from the context.

C.4 Advantage and margin

We develop some tools for reasoning about the settlement game. For a Δ-fork $F \vdash_\Delta w$, we define the Δ-*advantage* of a tine $T \in F$ as $\alpha_F^\Delta(T) = \mathrm{len}(T) - \mathrm{len}(\overline{F_{\lfloor\Delta}})$. Observe that $\alpha_F^\Delta(T) \geq 0$ if and only if T is Δ-dominant in F. For $\ell \geq 1$, we define the quantity of interest

$$\beta_\ell^\Delta(F) = \max_{\substack{T \not\sim_\ell T^* \\ T^* \text{ is } \Delta\text{-dominant}}} \alpha_F^\Delta(T)\,,$$

this maximum extended over all pairs of tines (T, T^*) where T^* is Δ-dominant and $T \not\sim_\ell T^*$. Note that there might exist multiple such pairs in F, but under the condition $\ell \geq 1$ there will always exist at least one such pair, as the trivial tine T_0 containing only the root vertex satisfies $T_0 \not\sim_\ell T$ for any T and $\ell \geq 1$, in particular $T_0 \not\sim_\ell T_0$. For this reason, we will always consider β_ℓ^Δ only for $\ell \geq 1$. We overload the notation and let

$$\beta_\ell^\Delta(w) = \max_{F \vdash_\Delta w} \beta_\ell^\Delta(F)\,.$$

Intuitively, $\alpha_F^\Delta(T)$ captures the length advantage (or deficit) of the tine T against the longest honest tine created at least Δ rounds before the upcoming round, and hence now known to all honest parties. Consequently, $\beta_\ell^\Delta(F)$ records the maximal advantage of any tine T_a in F that potentially disagrees with some Δ-dominant tine T_h about the chain state up to round ℓ.

The crucial property motivating these definitions is that $\beta_\ell^\Delta()$ provides explicit control over persistence failure events. This is reflected in the lemma below.

Lemma 19. *Fix a parameter k, and consider the sequence of forks $F_1 \vdash w_1, F_2 \vdash w_1 w_2, \ldots$ associated with each step of a settlement game for characteristic string $w = w_1 w_2 \ldots$. Consider a tine T held by an honest party in round r_1, which is hence Δ-dominant in F_{r_1}; let ℓ be a round associated with a vertex (block) B that is buried by k vertices (blocks) in T. If $\beta_\ell(w_1 \ldots w_r) < 0$ for all $r_1 \leq r \leq r_2$, then any Δ-dominant tine T' of F_{r_2} contains the vertex B. In particular, persistence is guaranteed for this block.*

The proof of the lemma is a straightforward induction on r; for full details see [27]. Thus, in order to rule out persistence violations it suffices to establish that $\beta_\ell^\Delta(w) < 0$ for appropriate ℓ and $w_1 \ldots w_{\ell+t}$ (note that this bounds above $\beta_\ell^\Delta(F)$ for any relevant fork). Similar connections were originally established for PoS forks [37] and more recently also for PoW forks [27, 6]. This yields Theorem 6.

On the other hand, margin is well-behaved analytically. In the lockstep synchronous case, it essentially increases for each adversarial PoW discovery and decreases for every honest PoW discovery, with the understanding that it descends below zero prior to ℓ (thus motivating the barrier walk of the main body). The situation with Δ delays is more complex, but one fairly straightforward fact is that margin is decreased by one for every uniquely isolated honest PoW discovery, and can increase by no more than one for an adversarial PoW discovery; again one has the extra constraint that it does not descend below zero prior to ℓ. This is the content of the "barrier" and "free" walks defined in the main body. The full inductive proofs appear in [6].

Returning to the main claims required for the body of the paper, for the reader's convenience we recall how the basic chain properties are inferred from the assumptions of Theorem 7:

- CG over a large enough interval follows from the fact that the interval contains many uniquely isolated honest blocks. Observe that across (the Γ-region around) any uniquely isolated success, the height of every honestly held chain must increase.

- ECQ follows from a straightforward counting argument so long as (i.) the margin at the outset of the interval is bounded, so that there is no adversarial chain of length that significantly exceeds those held by the honest players and (ii.) the number of uniquely isolated blocks in the region exceeds the number of adversarial blocks.

- CP follows directly from control on margin and CG.

D The protocol with honest restarts

We give a sketch of the proof that "restarts" do not significantly change the security analysis and, as indicated in the main paper, in fact improve the security properties of the protocol. In a particular round, there are three sorts of events we now wish to record: (i.) discovery of an honest mining victory, (ii.) discovery of an adversarial mining victory, and (iii.) delivery of an adversarial block to an honest party that increases the length of the chain held by the (honest) party. We remark that events of this last variety are not reflected by previous treatments but are essential in our setting because these block delivery events can restart the mining process of honest parties (and, fortunately, effect relative margin in a fashion consistent with new honest blocks).

To formally articulate this, we will focus on a new event: an "advance". This is the event that an honest party adopts a new blockchain via the longest-chain rule. Note that these are precisely the events that cause parties in our existing analysis to "restart" the underlying Markov chain. We may naturally associate a particular block with an advance event; namely, the deepest block on the blockchain adopted by the honest party during the advance event. Considering network delays, a given block may in fact be responsible for multiple advance events, as it may be delivered to different honest parties in different rounds. However, two advance events associated with the same block can be no more than Δ rounds apart (as honest players are assumed to broadcast any new chains at the moment they adopt them and advance events must necessarily increase the depth of the chain held by the advancing party). While honest blocks are necessarily associated with at least one advance event—arising from the honest player that created the block—adversarial blocks may generate zero advance events even if they eventually appear on a chain adopted by an honest player (in which case they would necessarily not be the last block on such a delivered chain).

To reflect this, we work with a richer notion of Δ-fork that maintains, for each block, the set of rounds in which the block generates an advance event. These new "advance-annotated" forks will be associated with characteristic strings that likewise reflect these events. Specifically, we work with "advance-annotated" characteristic strings: our characteristic strings have the structure $w = w_1, \ldots, w_L$ where each $w_i = (h_i, a_i; d_i, D_i) \in \mathbb{N}^4$: intuitively, the first two coordinates have the same interpretation as their counterparts in standard characteristic strings, while the last two coordinates reflect advance events. Specifically,

- h_i denotes the number of honest proof-of-work discoveries,

- a_i denotes the number of adversarial proof-of-work discoveries,

- D_i denotes the number of blocks, adversarial or honest, associated with an advance event in this round, which is to say that an honest party adopts the blockchain terminating at the block via the longest-chain rule; and

- d_i denotes the number of blocks, adversarial or honest, associated with an advance event for the *first time* in this round.

Observe that $h_i \leq d_i \leq D_i$.

Definition 20 (Delivery Augmented PoW Δ-fork)**.** Let Δ be a positive integer and $L \in \mathbb{N}$. A *delivery-augmented PoW Δ-fork* for the string $w \in (\mathbb{N}^4)^L$ is a directed, rooted tree $F = (V, E)$ with a triple of functions

$$\mathsf{l}_{\#} : V \to \mathbb{N}, \qquad \mathsf{l}_{\mathsf{A}} : V \to 2^L\,, \quad \text{and} \quad \mathsf{l}_{\mathsf{type}} : V \to \{\mathsf{h}, \mathsf{a}\}$$

satisfying the axioms below. Edges are directed "away from" the root so that there is a unique directed path from the root to any vertex. The value $\mathsf{l}_{\mathsf{type}}(v)$ is referred to as the *type* of the vertex: when $\mathsf{l}_{\mathsf{type}}(v) = \mathsf{h}$, we say that the vertex is *honestly generated*; otherwise $\mathsf{l}_{\mathsf{type}}(v) = \mathsf{a}$ and it is *adversarially generated.* The value $\mathsf{l}_{\#}(v)$ is referred to as the *creation time* of the vertex v. When $\mathsf{l}_{\mathsf{A}}(v) \neq \emptyset$, the vertex is said to be *advancing* and, if $s \in \mathsf{l}_{\mathsf{A}}(v)$, we say that the vertex v has an *advance event* associated with round s. For an advancing vertex v, the *initial advance* is the round $\min \mathsf{l}_{\mathsf{A}}(v)$.

(i) the root $r \in V$ has type $\mathsf{l}_{\mathsf{type}}(r) = \mathsf{h}$, has creation time $\mathsf{l}_{\#}(r) = 0$ and $\mathsf{l}_{\mathsf{A}}(r) = \{0\}$;

(ii) for all $v \in V$ and all $s \in \mathsf{l}_{\mathsf{A}}(v)$, $\mathsf{l}_{\#}(v) \leq s$; if $\mathsf{l}_{\mathsf{type}}(v) = \mathsf{h}$ then $\mathsf{l}_{\#}(v) \in \mathsf{l}_{\mathsf{A}}(v)$;

(iii) the sequence of creation times $\mathsf{l}_{\#}()$ along any directed path is non-decreasing;

(iv) if $w_i = (h_i, a_i; d_i, D_i)$, then there are

  (a) exactly h_i honestly generated vertices with creation time i,

  (b) no more than a_i adversarially generated vertices with creation time i,

  (c) exactly D_i vertices for which $i \in \mathsf{l}_{\mathsf{A}}(v)$, and

  (d) exactly d_i vertices for which $i = \min \mathsf{l}_{\mathsf{A}}(v)$;

(v) if two vertices u and v satisfy the property that there are a pair of rounds $s \in \mathsf{l}_{\mathsf{A}}(v)$ and $t \in \mathsf{l}_{\mathsf{A}}(v)$ for which $s + \Delta \leq t$ then the depth of v exceeds that of u.

We remark that the last axiom, reflecting the combination of the longest chain rule with networking delays, can be reformulated in the following natural way. For a fork F, let $D_t(F)$ denote the induced subgraph given by the union of all paths terminating at vertices v for which $\min \mathsf{l}_{\mathsf{A}}(v) \leq t$—all vertices "delivered to honest parties" by time t. Then, if there is an element s of $\mathsf{l}_{\mathsf{A}}(v)$ for which $s \geq t + \Delta$, the depth of v exceeds that of $D_t(F)$. In terms of the longest chain rule, this captures the intuition that any party carrying out the longest chain rule at time t observes all chains held by honest parties at time $t - \Delta$.

As with Δ-forks, we adopt the notation $F \vdash^{A}_{\Delta} w$ to indicate that F is an advance-annotated Δ-fork for the (advance-annotated) characteristic string w.

The analysis hinges on a re-evaluation of the behavior of margin $\beta_\ell()$ with these new symbols. First of all, to reflect the new types appearing in the notion of fork above, the notion of "honest" in the standard notion of forks must be substituted with the notion of "advancing." To be precise:

- The notion of *closed* is changed to refer a tine ending in an advancing vertex (rather than an honest vertex). The closure of a fork then contains all vertices on chains adopted by honest players using the longest-chain rule. (As mentioned above, with the convention that honest players ignore delivered blocks that are not on chains that activate the longest-chain rule, the closed fork intuitively reflects all delivered blocks.)

- Fork trimming and dominance (as defined in Section C.3) must be suitably adapted to properly handle $\mathsf{I}_{\mathsf{A}}()$. Specifically, $F_{\lfloor k}$ is (still) defined to include all vertices for which $\mathsf{I}_{\#}(v) \leq n-k+1$. Of course, the definition of $\mathsf{I}_{\mathsf{A}}()$ is suitably updated to account for the trimmed set of rounds; that is $\mathsf{I}_{\mathsf{A}}(v) \cap \{1, \ldots, n-k+1\}$. Observe that restricting $\mathsf{I}_{\mathsf{A}}()$ in this way may mean that vertices which were advancing in F may no longer be advancing in $F_{\lfloor k}$ (as the slots in which they were to be delivered no longer exist).

With these conventions, the most important of which replaces "honest" vertices of the conventional theory with "advancing" vertices, the direct connection between $\beta_\ell()$ and persistence failures (Lemma 19) is retained.

We first discuss the lockstep synchronous case (with no network delays). In this case, $\mathsf{I}_{\mathsf{A}}(v)$ is either empty or a singleton and $D_i = d_i$. Observe that when $\beta_\ell(w) \leq 0$, there are no adversarial blocks of depth exceeding the longest honest chain, so no adversarial block delivery can force an honest party to abandon their chain. On the other hand, when $\beta_\ell(w) > 0$ the adversary may indeed deliver blocks of depth exceeding those held by the honest players; however, in this case the depth of the chain held by the honest player is increased by at least one: in particular, this decreases β_ℓ. For the case with network delays (or more exotic delays such as those considered in this paper), note that the height of the longest chain observed by any honest player must increase across any region with an honest PoW success *or* an adversarial block delivery (which necessarily increases the height of the recipient's chain) with at least Δ symbols on either side; this suffices to decrease margin over such an area, as desired. The full analysis then follows that of [6].