Simulating Bitcoin mining

Andreas Bech

University of Southern California

13 December, 2021

1 Introduction

Previous attempt at modeling the proof-of-work mining game as an MDP has made several simplifying assumption. Most notably, a step in the MDP process is a found block for any player. This is justified with the time to next block being memoryless (exponentially distributed) and hence any decision making between blocks is invariant to time. This assumption greatly reduces the size of the state space. This version of an MDP model of blockchain mining I will call the block domain.

What is lost with this assumption, however, is that it loses track of the profitability of a particular policy. Profitability is an explicit function of time and policies in this MDP framework thus cannot be properly evaluated.

The objective for any miner on the Bitcoin network should be to maximize the profit and loss (PnL) per unit of time. Profit and loss is the revenue R minus the cost C

$$PnL = R - C.$$

In this paper costs are assumed to be constant and independent of the mining strategy, and are normalized to $C = 0$.

Since explicit profitability cannot be calculated, the alternative evaluation used in the block domain is the share of blocks mined for the attacker, the reward ratio, which is not a function of time. This measure, however, fails to account for a given policy's effect on the total number of blocks mined per unit of time, and thus the realized profitability of a mining policy. That is, maximizing the reward ratio is not equivalent to maximizing the PnL per unit of time.

Furthermore, without an explicit notion of time, the block domain model cannot incorporate difficulty adjustments into the model. The Bitcoin protocol aims for a block to be created roughly every 10 minutes (sometimes called the target) and thus adjust the difficulty of the mining game whenever blocks are being found too (in)frequently. The adjustments take place every 2016 blocks (roughly two weeks, often called an epoch). Since time is not part of the state in the block domain, the difficulty adjustments cannot be calculated in the MDP. The difficulty adjustments

are crucial in evaluating the profitability of a mining strategy since they directly affect the transition probabilities in the (actual) MDP.

In this paper I try to evaluate mining policies using a discrete time MDP (call it the time domain). The primary purpose of the paper is to introduce a framework for approximating and evaluating the actual bitcoin mining process and to give context to the rich literature using the block domain approach to modeling the Bitcoin mining problem.

Primarily through simulations, this paper quantifies the true feasibility of alternative mining strategies/attacks on the Bitcoin network.

2 Literature Review

Some test. [3]

The selfish mining strategy (SM1) was originally suggested by Eyal and Sirer [3]. Using the block domain they show that this strategy yields a higher share of the total blocks to the attacker than the attacker's share of computing power. Several papers have built on the block domain model to devise slightly improved policies to SM1 using standard dynamic programming techniques, see Sapirshtein et al. [11] and Bar-Zur et al. [1].

Grunspan and Pérez-Marco [5], [6] prove with an analytic approach, however, that selfish mining is not profitable without difficulty adjustments. While selfish mining does give the attacker a larger share of blocks, the total number of block decreases. This is because selfish mining will "waste" a number of found blocks that do not end up in the main/settled chain. On the other hand, the authors also show that, since the difficulty will be lowered on the network, selfish mining will eventually be profitable, with the time to recoup initial losses depending crucially on a couple of parameters — the share of computing power and the attackers connectivity in the network.

Other alternative mining strategies have been proposed. Smart mining [4] manipulates the difficulty adjustment by not mining for certain periods and thus decreasing the difficulty while saving the variable costs of mining. Stubborn mining [8] augments selfish mining by combining it with certain network level attacks

Finally, SquirRL [7] utilizes deep Q-learning to look for strategies in blockchain protocols where the state space is too large for policy/value iteration (to be clear, SquirRL still uses the block domain but aims to be general framework to evaluate consensus mechanisms and blockchains that are more involved than the one for Bitcoin). And Wang et al. [12] uses reinforcement learning to look for optimal policies in a model-free framework where the transition probabilities are assumed to be unknown.

3 The Model

This MDP models the payoff and environment to a Bitcoin miner seeking to maximize profits by choosing the optimal timing to publish "found" blocks. Note that the environment here is the honest miners on the network choosing always to follow the "honest" strategy, also known in the Bitcoin protocol as the "longest chain rule". Thus, there is only one player, the attacker, and the problem reduces to a dynamic programming problem.

The general state of the environment is a "fork" that consist of two branches, one for the honest miners and a secret one for the attacker. The branches branch from the latest block that both the honest miners and the attacker agree on, i.e. the latest settled block. It is assumed that the attacker does not challenge any block before this latest forking block. It is also implicitly assumed that honest miners do not create forks among themselves - in this framework the honest miners are one logical unit. The following list is a brief overview of the MDP.

- State Variabels: h(onest), a(ttack), s(ettled), d(ifficulty), w(time), f(ork)
- Actions: adopt, override, match, wait
- Reward: Block rewards for every settled block for the attacker
- Transition: Two branches are incremented with Bernoulli processes, or reset
- Difficulty: Updated when $s > 2015$

3.1 The Action Space

You can think of many actions that a miner could take during mining on the Bitcoin network. In this case, we are designing a "small" MDP in which the state/action space encompasses the selfish mining strategy first introduced in [3]. There are four actions $\mathcal{U} = \{adopt, override, match, wait\}$

1. *adopt* means that the attacker accepts the branch built by the honest miners and discards its own secret branch
2. *override* means the attacker publishes its secret branch and the honest miners accept this branch if it is longer that the honest branch (we restrict the action space such that the attacker can only override if this is the case)
3. *match* imitates the scenario where the attacker learns of a new public block and quickly published h secret blocks to match. This is only feasible if $a \geq h$ and the latest public block was just published. In this case it puts the network into a forked (another kind of fork) state where some miners mine on the public branch and some mine on the attackers branch, since both branches are equally long. The share of each is determined by an exogenous parameter γ.
4. Finally, *wait* means taking no action during a time step, other that to continue mining.

Clearly, *wait* and *adopt* are available in all states, whereas *override* is only available when $a_t > h_t$. To make the action space simpler, the attacker does not choose how

many blocks to reveal when overriding. Instead, in the simple version, the attacker publishes $h_t + 1$ blocks such that the honest miners accept the new blocks, and the attacker retains $a_t - h_t - 1$ secret blocks. An alternative is to let the attacker choose how many to reveal.

3.2 The State

The state is defined by the following state variables: h_t = public blocks, a_t = private blocks, s_t = settled blocks, d_t = multiplicative difficulty parameter, w_t = time passed since last difficulty adjustment, and $fork_t$ = the current state of match. Denote by $x_t = (h_t, a_t, s_t, d_t, w_t, fork_t)$ the state at time t

h_t and a_t represent the length of the resp. honest miners and attacker's branches. As these can take any natural number, some kind of truncation will be needed. For example, you could force the attacker to take the action *adopt* if either branch reaches $100 (= maxfork)$.

The settled blocks variable s_t is needed as it determines when the difficulty update is done — $s_t = 2016$ in the official protocol. The difficulty d_t takes real numbers and so will need to be discretized. Finally it is needed to keep track of time in the state, w_t, since this enters in the difficulty adjustment calculation.

The fork variable $fork_t$ keeps track of when it is feasible to match, and whether the network is currently in a matched state. If a public block was just published, $fork_t = relevant$ since, if $a \geq h$, the attacker can quickly match with h blocks. If a public block was not just released in the previous time step, $fork_t = irrelevant$ and matching is not an option. Finally, if the attacker chooses to match it will put the network into a forked state, $fork_t = active$ and it will remain active until someone finds a new block, which settles the fork.

3.3 The Reward Function

The reward function is simple. The attacker is rewarded for adding settled blocks to the blockchain. This is happens when the attacker overrides — an action that is only available when $h_t < a_t$. In this case the attacker is rewarded for the number of blocks that it adds to the blockchain, $h_t + 1$, times the block reward, b, which will remain constant. It also can happen if the attacker matches and the next block is found by an honest miner mining on the attackers branch, and thus resolving the fork. In this case the attacker receives a reward of $h_t \cdot b$.

Let $u_t \in \mathcal{U}(x_t)$ be the action chosen at time t in state x_t, and let

$$r_t(x_t) = \begin{cases} b \cdot (h_t + 1) & \text{if } override \in \mathcal{U}(x_t) \ \& \ u_t = override \\ b \cdot h_t & \text{if } fork_t = active \ \& \text{ honest miner finds block on attackers branch} \\ 0 & otherwise \end{cases}.$$

3.4 The State Transition

The time it takes for the network to find the next block, denoted by T, is exponentially distributed with parameter λ (this is a feature of the SHA256 hash function). The expected time to find a block is then $T = \frac{1}{\lambda} = \frac{d \cdot C}{hashrate}$[1] given a difficulty, d, and a $hashrate$. With an attacker and an honest miner let $\frac{1}{\lambda} = \frac{1}{\alpha_h + \alpha_a}$, and $share_h = \frac{\alpha_h}{\alpha_h + \alpha_a}$ and $share_a = \frac{\alpha_a}{\alpha_h + \alpha_a}$. Here $share_h$ and $share_a$ are the relative hashrates of the honest miners and the attacker, respectively.

This model discretizes this process into time steps of size Δt. Time steps will be made small enough such that at every time step both the miners either find one block or they do not. As you make the Δt smaller the distribution of the number of blocks found for one player within a time step will be well approximated by a Bernoulli(p) (resp. q) distribution. Here $p = \Delta t \cdot \alpha_h$ gives the correct approximation. Likewise $q = \Delta t \cdot \alpha_a$. As an example, the transition probability from state (h, a) to $(h + 1, a)$ for action "wait" will be $p(1 - q)$. By letting the time interval be small enough, the probability for two found blocks in an interval vanishes, $p \cdot q \approx 0$, and this outcome can be disregarded in the model.

The transition is complicated by the fact that, when the network is in a forked state, some of the honest miners mine on the attackers branch (a share γ of the honest miners), which is of the same length as the honest branch. For that reason, consider the transition divided into two cases. Denote by $\text{Ber}(p)$ a bernoulli distributed random variable with parameter p. Also denote the honest miners probability by p and the attacker's by q. If the network is currently **not** in a forked state, then the transition is defined by the action taken by the attacker as

$$x' = (h', a', s', \cdot) = \begin{cases} (h + \text{Ber}(p), a + \text{Ber}(q), s, \cdot) & \text{if } u = wait \\ (\text{Ber}(p), \text{Ber}(q), s + h \mod 2016, \cdot) & \text{if } u = adopt \\ (\text{Ber}(p), a - h - 1 + \text{Ber}(q), s + h + 1 \mod 2016, \cdot) & \text{if } u = override \end{cases}.$$

The dot represents the w, d and $fork$ in the state. w and d update based on two cases.

- If s' does not exceed 2016 (and is thus not reset) then $w' = w + 1$ and $d' = d$, i.e. there is no difficulty update.
- In the case where s' exceeds 2016, there will be an update and $w' = 0$ and $d' = f(d, w)$ where f is the difficulty update function specified by the protocol[2].

while $fork$ is determined by whether the honest miners just found a block or not. If an honest miner finds a block, then $fork = relevant$ and the attacker has the option to $match$. Otherwise $fork = irrelevant$.

If the action taken is $match$, then there are four possible next states

[1] See Appendix A for definitions of d and C

[2] See appendix A

1. The attacker finds the next block w.p q. In this case a increases by 1 and the fork is assumed to be solved even though the new block is kept secret, that is $fork_t = irrelevant$
2. An honest miner finds a block on top of the attackers branch w.p $p \cdot \gamma$. The fork is resolved and $fork = relevant$. Furthermore $h + 1$ blocks are settled and $h' = 1$ and $a' = a - h$.
3. Honest miner finds block on honest branch w.p $p \cdot (1 - \gamma)$. $h' = h + 1$, $a' = a$ and $fork = relevant$
4. No block is found, the network remains in a forked state, $fork = active$, in which case a new action has to be taken by the attacker.

If the network **is** already in a forked state and $fork = active$ then the attacker can choose actions *wait*, *adopt* and *override*, however, we assume that the attacker cannot choose to *match* if the network is already in a forked state.

4 Simulation

By defining a fully contingent policy we can test it in a simulation. The simulator works by specifying all possible transitions given a state and then sampling one according to the probabilities associated with each transition.

While the state space is too large to store either a value vector or a transition matrix, the defined transition can simply the defined as a function that outputs all states with a non-zero transition probability given the current state.

The policies from the literature I have run through the simulator are honest mining (following the longest-chain protocol), selfish mining (SM1) and intermittent selfish mining (ISM).

Honest mining is what is prescribed by the Bitcoin protocol and is the assumed Nash Equilibrium of the game induced by the opportunity of mining on the Bitcoin network. Some authors have attempted to formally define this equilibrium as a Markov perfect equilibrium of a repeated coordination game [2]. The recipe is to always mine on the longest chain visible to the miner and to broadcast found blocks immediately to the rest of the network.

Selfish mining is based on the idea of withholding found blocks, that is, initially not broadcasting found blocks to the network. In this way, the rest of the network will continue to mine on the latest public block and thereby will have potentially wasted time and resources when the secret blocks are eventually published. The idea was first proposed in 2010 on the bitcointalk[3] forum and later formalized in 2013 by Sirer and

[3] https://bitcointalk.org/

Eyal [3]. The policy is defined as

$$SM_1(a,h,\cdot) := \begin{cases} adopt & \text{if } h > a \\ match & \text{if } h = a \ \& \ fork = relevant \\ override & \text{if } h = a - 1 \geq 1 \\ wait & otherwise \end{cases}.$$

So you adopt if the public chain is ahead, you match if the honest chain has caught up to you and you override if the honest chain is close to catching up with you.

Intermittent selfish mining from [9] is selfish mining but only every second epoch. The strategy is to attack with selfish mining until the difficulty adjustment and then, when the difficulty is lowered, revert to honest mining in order to collect as many blocks as possible.

The results from simulating the selfish mining policy are shown in Figure 1 for different combinations of α[4] and γ. The simulation is done with 1 second time steps and initialized from an unforked state. One data point every 10 minutes is plotted in the figures and the simulation is shown for about 8 weeks.

The results are in accordance with the literature. Comparing with [11] Fig.1, the thresholds for when selfish mining is profitable are the same. The threshold for α is different depending on the network connectivity, γ, and for $\gamma = 1$ selfish mining is always profitable. The overall conclusion from these simulations is that as long as selfish mining gives the attacker a larger share of the blocks, selfish mining will eventually become profitable as the difficulty is adjusted to the new lower block rate under the selfish mining attack. That is, the conclusions from [11] and [3] are correct, as long as it is assumed that the selfish mining attack can continue indefinitely, without new miners entering to take advantage of a new lower difficulty.

The final combination in Figure 1, with $\alpha = 0.1$ and $\gamma = 0.9$ is almost inconclusive after 8 weeks, and has not broken even yet. In Figure 2 below, the result is plotted for a full 20 weeks. It can be seen that the attack eventually becomes profitable and breaks even at about 10 weeks or about 5 difficulty adjustments (epochs).

Compare this with the analytical result from Grunspan & Pérez-Marco [6] Fig. 1 and proposition 3.7. The simulation is exactly in accordance with their theoretical conclusion. See also their paper's Fig. 2 for the exact dominance region in the (α, γ) parameter space.

Finally, alternative policies like intermittent selfish mining and a random policy are shown below. Intermittent selfish mining as defined in [9] alternates between the SM1 strategy and honest mining in an effort to game the difficulty adjustment mechanism. The random policy is an arbitrarily chosen policy only included for illustration purposes. The policy will override, randomly, when the attacking chain is in front. That is, with a small probability the attacker overrides and otherwise it waits.

[4] α is actually $share_a$ but in the literate it is defined as α.

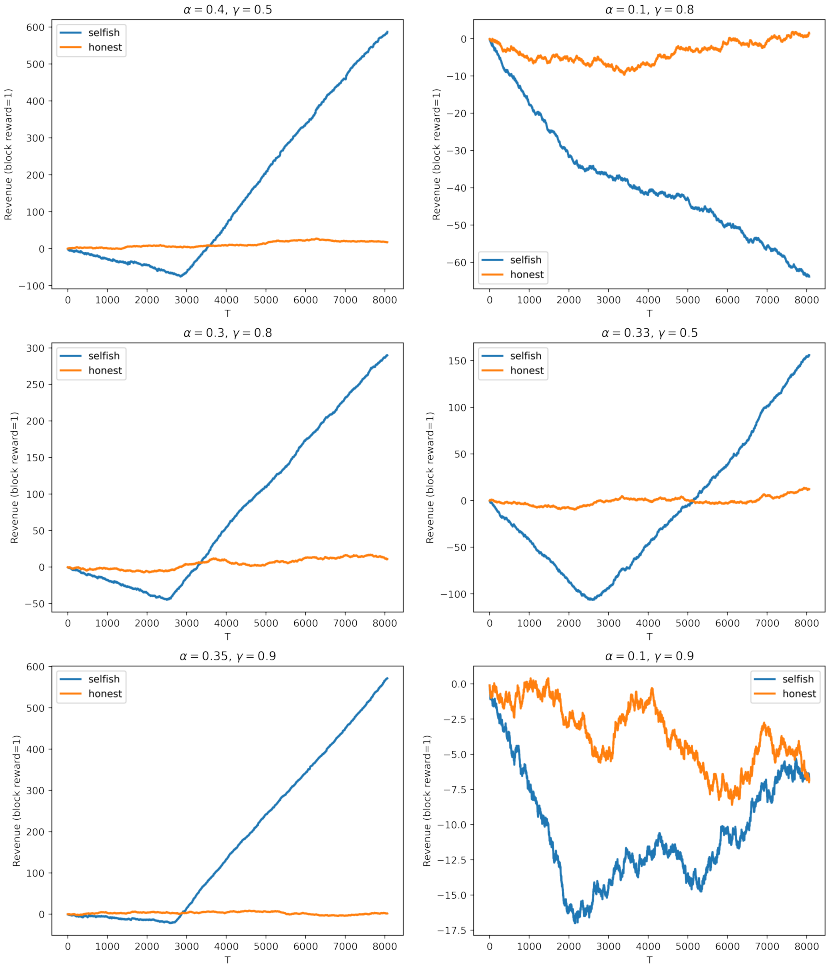

Figure 1: The results from the selfish mining strategy (SM1) after 8 weeks. Each data points is 10 minutes (600 seconds), while the timestep during the simulation was 1 second. The plots show the difference from the expected reward during the simulation, where the expected reward is the expected number of blocks times the share, α, times the block reward (in this example normalized to 1. The six plots present different combinations of α and γ, where γ is the connectivity.

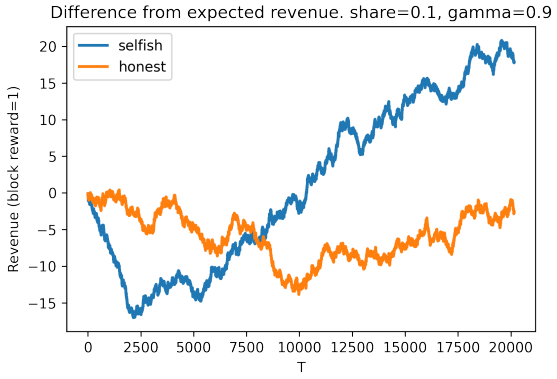

Figure 2: The results from the selfish mining strategy (SM1) after 20 weeks with $\alpha = 0.1$, $\gamma = 0.9$. Each data points is 10 minutes (600 seconds), while the timestep during the simulation was 1 second. The plots show the difference from the expected reward during the simulation

As can be seen from the simulation, however, intermittent selfish mining does not perform better. In fact, selfish mining performs better. While it is more profitable to mine honestly after the difficulty is lowered, it also has the consequence of increasing the difficulty back up to where it started. Repeating this pattern is not better than to keep on selfish mining perpetually. It is perhaps more realistic, however, to follow the intermittent selfish mining, as this strategy does permanently decrease the difficulty, and perhaps deters the entry of new miners.

5 Solving a smaller problem

Ideally we would like to find an optimal policy in this MDP framework (the time domain), but the state space is too big to apply any algorithm like policy iteration or value iteration. An option is to search for an optimal policy in a smaller problem.

The current size of the problem with state $x_t = (h_t, a_t, s_t, d_t, w_t, fork_t)$ is very big, especially due to s_t, d_t and w_t. A back-of-the-envelope calculation yields $|S| = 20 \cdot 20 \cdot 2016 \cdot (2016 \cdot 2000) \cdot 50 \approx 2^{47}$.

If instead we restrict the difficulty to be updated every 30 blocks and set a $maxfork$ length of 10 then $|S_{small}| = 10^2 \cdot 30^2 \cdot 2000 \cdot 50 \approx 2^{33}$. This is a reasonably sized problem that can be solved. This would be a logical next step in the framework.s

6 Discussion

One current issue with the simulator is that the state transition during a difficulty update is imperfectly specified. A difficulty update happens when the number of

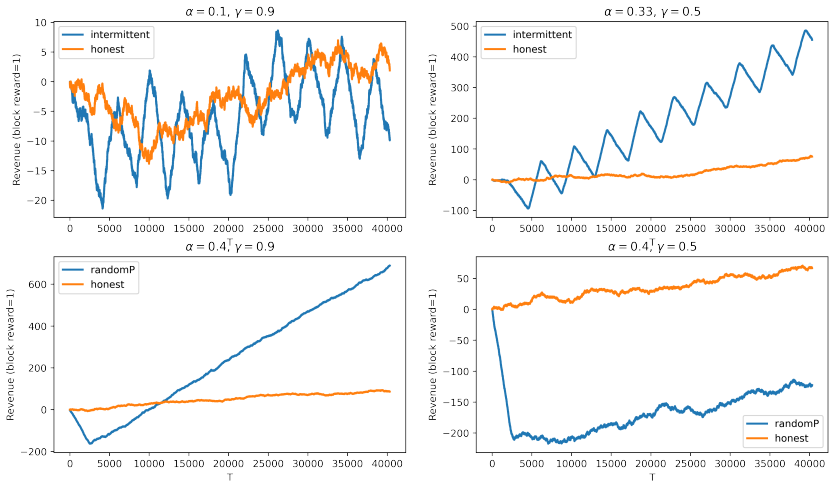

Figure 3: The results from the alternatively defined mining strategies after 20 weeks. Each data points is 10 minutes (600 seconds), while the timestep during the simulation was 1 second. The plots show the difference from the expected reward.

settled blocks s_t reaches 2016 and s_t is in turn increased whenever the action *adopt* or *override* is chosen, and sometimes when *match* is chosen. The issue is that s_t is often incremented by more than one block at a time - it is incremented by either h or $h+1$. The leads to ambiguity in when the update should take place in the MDP, that is when the difficulty should be updated and when the timer w should be reset. For example if $s_t = 2015$ and $a_t = 7$ and $h_t = 5$ and then the honest miners find a block such that $h_{t+1} = 6$. Under SM1 the attacker will choose to override and this will settle 7 blocks and give $s_{t+1} = 2022$. Currently in the simulator, this will trigger an update and set $s_{t+1} = 6$ and $w_{t+1} = 0$. This of course introduces a couple of biases. First of all, there are 6 settled blocks since the update but time since the update is 0. This will bias the time it takes to the next update towards zero. Second, the 6 settled blocks were mined using the previous difficulty level, which is not in accordance with the protocol.

The true way the transitions should be defined, the way the updates take place in the protocol, is to update on either branch as soon as $h + s = 2016$ or $a + s = 2016$. This however would create fork in the MDP as the state would have to keep track of both a difficulty parameter, d_t and the time since an update, w_t for both the public branch and the attackers branch. At this point I am not sure how to remedy this issue.

Another improvement that could be made is the introduction of costs per unit of time, both fixed costs and variable costs (incurred only when mining). This would give a richer strategy space and allow for strategies such as smart mining and alternate network mining [6].

Regarding the overall design of the MDP, there is a claim in the appendix of Hou et

al. [7] that the expected time during an epoch is

$$D = (M + S_a + S_h)\tau_0.$$

where D is the expected time, $M = 2016$ is the number of blocks for an epoch, and S is the number of stale/orphaned blocks during the epoch for, respectively, the attacker and the honest miner, and $T_0 = 600$ is the target between blocks in the protocol. If the claim is true then the framework could potentially be improved by having the difficulty update function depend on stale blocks instead of time. This would greatly reduce the state space and allow for an easier search for optimal policies.

Here $M + S_a + S_h = B_a + B_h$ is the total blocks mined during an epoch, where I denote B as the total number of blocks found for one player. Of course, M represents the blocks that are settled in the main chain. The issue is whether B is gamma (Erlang) distributed. If a player were to just mine on their own branch, and not mind what happened in the other branch, B would be gamma distributed as a sum of independent exponential distributions. However, in the mining game, if a branch is published (using override) or the public branch is adopted, a player will switch from mining on their own current block to the new longest chain. Intuitively a player has then "wasted" time on the previous block they were mining on, however, due to the memory less property of the distribution, there is no such thing as wasted time. As such, under any strategy of the attacker, the expected time to mine $M + S_a + S_h$ blocks, stale or settled, is indeed $D = (M + S_a + S_h)\tau_0$.

This means you could potentially replace the update function $f(d, w)$ with $f(d, S)$ in the MDP model, in effect replacing the actual time passed during the epoch with the expected time it would have taken to mine $M + S_a + S_h$ blocks. It is not entirely clear what implications this substitution would have for the MDP and any potential optimal policy.

Finally, it should be noted that a few crucial assumptions are present in almost all of the literature on the subject. Most importantly, the dynamics of entry and exit to and from the mining game is ignored. In order to fully analyze the economics of attacks on these decentralized monetary systems, explicit entry and exit conditions should be part of the dynamics. These conditions would likely depend on the difficulty parameter and the price/value of the underlying asset (like Bitcoin or Ether) relative to the US dollar. The paper by Prat Walter [10] is the first serious attempt at doing this, however, the context of the paper is not mining attacks on the network, but rather on the overall economics of the price, use of global electricity, and the future incentive compatibility of honest mining.

7 Appendix

7.1 A - Difficulty adjustment

This section gives a brief description of the difficulty adjustment in the Bitcoin protocol. Ignoring a few details , the difficulty level is simply $d = \frac{offset}{t}$ (offset is a normalizing

constant). The hash (SHA256) of a block is an integer in the range $[0, 2^{256} - 1]$. The target, t, is an integer in this range and if the hash, of a block is less than the target then the block has a valid proof-of-work. That is, if $hash < t$ then a block is found. The hashing algorithm SHA256 is completely random, implying that when "searching" for a block (by incrementing a nonce) the $hash$ is uniformly distributed on $[0, 2^{256} - 1]$ and thus

$$p = Pr(hash < t) = \frac{t}{2^{256} - 1}$$

The number of hashes you need to calculate in order to find a block is thus geometrically distributed with parameter p and the expected number of hashes you need is $1/p$

$$\mathbf{E}[\#hashes] = \frac{2^{256} - 1}{t} = \frac{(2^{256} - 1)d}{offset} = dC$$

where I have defined the constant $C = (2^{256} - 1)/offset$. The protocol sets the difficulty, d, at the end of every epoch such that the next block is expected to be found after 10 minutes (600 seconds) if the hashrate stays the same as in the previous epoch. Call this parameter the target $\tau_0 = 600$. It is thus set using the equation

$$\frac{\mathbf{E}[\#hashes]}{hashrate} = \frac{dC}{hashrate} = 600 \Leftrightarrow d = \frac{600 \cdot hashrate}{C}$$

where $hashrate$ is the total hashrate for the network (the sum of all the miner's hashrates).

The protocol does not know the hashrate, but can estimate it using the previous 2016 block by

$$\hat{hashrate} = \frac{2016 \cdot d \cdot C}{T}$$

where T is the time passed during the past 2016 blocks, in the framework of the MDP it is equivalent to $w_t/\Delta t$. The difficulty adjustment is then the recursive update

$$d' = \tau_0 \cdot \hat{hashrate} \cdot C = \frac{\tau_0 \cdot 2016 \cdot d}{T}$$

an explicit function of time and the previous difficulty level.

References

[1] Roi Bar-Zur, Ittay Eyal, and Aviv Tamar. Efficient MDP Analysis for Selfish-Mining in Blockchains. *arXiv:2007.05614 [cs]*, September 2020.

[2] Bruno Biais, Christophe Bisière, Matthieu Bouvard, and Catherine Casamatta. The Blockchain Folk Theorem. *The Review of Financial Studies*, 32(5):1662–1715, May 2019.

[3] Ittay Eyal and Emin Gun Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. *arXiv:1311.0243 [cs]*, November 2013.

[4] Guy Goren and Alexander Spiegelman. Mind the Mining. *arXiv:1902.03899 [cs]*, February 2019.

[5] Cyril Grunspan and Ricardo Pérez-Marco. On profitability of selfish mining. *arXiv:1805.08281 [cs, math]*, January 2019.

[6] Cyril Grunspan and Ricardo Pérez-Marco. Profit lag and alternate network mining. *arXiv:2010.02671 [cs, math]*, October 2020.

[7] Charlie Hou, Mingxun Zhou, Yan Ji, Phil Daian, Florian Tramer, Giulia Fanti, and Ari Juels. SquirRL: Automating Attack Analysis on Blockchain Incentive Mechanisms with Deep Reinforcement Learning. *arXiv:1912.01798 [cs]*, August 2020.

[8] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 305–320, Saarbrucken, March 2016. IEEE.

[9] Kevin Alarcón Negy, Peter R. Rizun, and Emin Gün Sirer. Selfish Mining Re-Examined. 12059:61–78, 2020.

[10] Julien Prat and Benjamin Walter. An Equilibrium Model of the Market for Bitcoin Mining. *Journal of Political Economy*, 129(8):2415–2452, August 2021.

[11] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal Selfish Mining Strategies in Bitcoin. *arXiv:1507.06183 [cs]*, July 2015.

[12] Taotao Wang, Soung Chang Liew, and Shengli Zhang. When Blockchain Meets AI: Optimal Mining Strategy Achieved By Machine Learning. *arXiv:1911.12942 [cs]*, January 2021.