WeRLman: To Tackle Whale (Transactions), Go Deep (RL)

Roi Bar-Zur
Technion, IC3
roi.bar-zur@campus.technion.ac.il

Ameer Abu-Hanna
Technion

Ittay Eyal
Technion, IC3

Aviv Tamar
Technion

***Abstract*—The security of proof-of-work blockchain protocols critically relies on incentives. Their operators, called *miners*, receive rewards for creating *blocks* containing user-generated *transactions*. Each block rewards its creator with newly minted tokens and with *transaction fees* paid by the users. The protocol stability is violated if any of the miners surpasses a *threshold* ratio of the computational power; she is then motivated to deviate with *selfish mining* and increase her rewards.**

Previous analyses of selfish mining strategies assumed constant rewards. But with statistics from operational systems, we show that there are occasional *whales* – blocks with exceptional rewards. Modeling this behavior implies a state-space that grows exponentially with the parameters, becoming prohibitively large for existing analysis tools.

We present the WeRLman[1] framework to analyze such models. WeRLman uses *deep Reinforcement Learning* (RL), inspired by the state-of-the-art AlphaGo Zero algorithm. Directly extending AlphaGo Zero to a stochastic model leads to high sampling noise, which is detrimental to the learning process. Therefore, WeRLman employs novel variance reduction techniques by exploiting the recurrent nature of the system and prior knowledge of transition probabilities. Evaluating WeRLman against models we can accurately solve demonstrates it achieves unprecedented accuracy in deep RL for blockchain.

We use WeRLman to analyze the incentives of a rational miner in various settings and upper-bound the security threshold of Bitcoin-like blockchains. We show, for the first time, a negative relationship between fee variability and the security threshold. The previously known bound, with constant rewards, stands at 0.25 [2]. We show that considering whale transactions reduces this threshold considerably. In particular, with Bitcoin historical fees and its future minting policy, its threshold for deviation will drop to 0.2 in 10 years, 0.17 in 20 years, and to 0.12 in 30 years. With recent fees from the Ethereum smart-contract platform, the threshold drops to 0.17. These are below the common sizes of large miners [3].

***Index Terms*—Blockchain, Selfish Mining, Bitcoin, Ethereum, Transaction Fees, Whale Transactions, Miner Extractable Value, Deep Reinforcement Learning, Monte Carlo Tree Search, Deep Q Networks**

I. Introduction

Proof of Work cryptocurrencies like Bitcoin [4], Ethereum [5] and Zcash [6], implement digital currencies and secure hundreds of billions of dollars [7]. Such cryptocurrencies are based on decentralized *blockchain* protocols, relying on *incentives* for their security. Blockchains are maintained by *miners* who use computational power to create new blocks. In return, the protocols distribute rewards to their miners in the form of virtual tokens. Ideally, every miner should get her fair share of the reward, based on how much computational power she controls [4], [8], [9].

However, many protocols are vulnerable to *selfish mining* [2], [8], [10], [11], where a miner deviates from the desired behavior to obtain more than her fair share. Such deviations destabilize the protocol, harming its safety and liveness guarantees [12]. The *security threshold* of a blockchain protocol is the minimum relative computational power required to perform selfish mining. The threshold should be high enough such that no miner is motivated to deviate.

Like previous work, we analyze the system as a *Markov Decision Process* (MDP) [2], [10], [13], a mathematical model for a single-player stochastic decision-making process that is played in stages [14]. The MDP can be used to find the optimal strategy of a rational miner. If the optimal strategy is not the desired one, then a rational miner should deviate. We can find the security threshold by considering different miner sizes. Thus, finding the security threshold amounts to solving a sequence of MDPs.

To our knowledge, previous work on mining incentives assumed that block rewards are constant [2], [8]–[10], [13], [15]–[17] or that the available rewards are constant [18], [19]. However, in practice, variability is significant (Fig. 1). Contention for block space results in an active market, where users vary the *fees* they offer for placing their *transactions*. Miners can then increase their rewards by including specific transactions in their block. These rewards are called *Miner Extractable Value* (*MEV*) [20].

In this work, we introduce a model of blockchain incentives that considers MEV. We focus on infrequent opportunities for rewards much higher than the average, as apparent in Fig. 1. Our model applies to *Nakamoto Blockchains*: Bitcoin and blockchains based on Bitcoin, e.g., Zcash, Litecoin and Bitcoin Cash. We model a single rational miner trying to optimize her revenue while all other miners behave honestly. To the best of our knowledge, this is the first analysis of honest mining as a Nash equilibrium when both the constant block reward and MEV opportunities are present.

Our model extends the selfish mining model of Sapirshtein et al. [2] to include transaction fees. We make the assumptions that all transaction fees are equal and that each block can contain a single transaction. Thus, high-reward blocks and low-reward blocks are captured as blocks with or without a transaction, respectively. Despite these easing assumptions, integrating MEV into the model poses new challenges. The model must synchronize the arrival of transactions with the creation of blocks, without allowing the miner to manipulate the transaction arrival rate. In addition, the extension yields a significantly larger model. In our configuration, the model contains over 10^8 states, prohibiting direct dynamic program-

[1] Pronounced werl-man. The Wellermen were employees of the 19th century whale-trading company Weller [1].

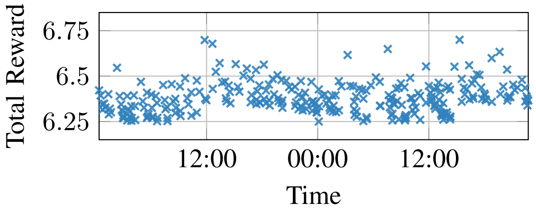


Fig. 1: Bitcoin block rewards, Sep. 1–2, 2021.

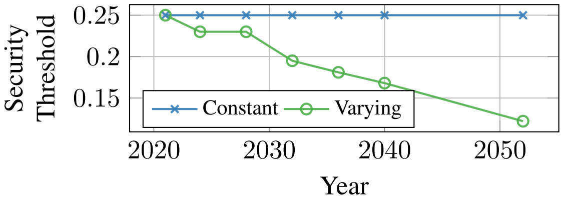


Fig. 2: Projected upper bound of Bitcoin's security threshold.

ming based solutions.

To analyze security in the presence of varying rewards, we use *deep reinforcement learning* (RL) – a popular method for approximately solving large MDPs [21]. However, we found that previous work on deep RL for blockchain [10] yields approximations that are not precise enough for our problem. Therefore, we present WeRLman[2] – a novel RL algorithm that is tailored for blockchain MDPs. WeRLman combines *deep Q networks* [22], a popular RL method, with *Monte Carlo Tree Search* (MCTS) [23], [24] and a novel variance reduction technique that exploits the structure of the blockchain problem. We show that WeRLman yields superior accuracy compared to the state-of-the-art, and compute security thresholds for our problem.

In addition to WeRLman, we present a simplified model of a Nakamoto blockchain with MEV by limiting the available actions of the miner. The simplified model requires a state space 2 orders of magnitude smaller compared to the full model, and thus can be solved using dynamic programming. Although the limitations of the model result in a lower bound for the revenue, this approach yields an accurate result. We thus use the simplified model to further improve the accuracy of the threshold: if the revenue found by WeRLman is lower than that of the simplified bound (due to approximation errors), we use the value given by the simplified model. We observe that for many cases this does not happen. However, we find it particularly useful for miner sizes close to threshold, since the security threshold is sensitive to any deviation from the honest strategy, even if it yields only a very minor advantage.

Our analysis of the Nakamoto blockchain shows it is more vulnerable to selfish mining than when assuming constant rewards, as in state-of-the-art [2]. Higher reward variability leads to a higher incentive to deviate when large fees are available, lowering the security threshold. The security guarantees of blockchains rely on rational miners following the prescribed protocol. Even temporary deviations, when rare MEV opportunities are present, revoke the assumptions their security relies on.

To estimate the effect on real systems, we choose model parameters based on publicly available data on operational systems rewards. A year of Bitcoin data reveals a significant fee variance. However, most of its rewards come from *subsidy* – newly minted tokens. Therefore, taking today's values, the threshold does not change significantly.

But, by design, Bitcoin's minting rate is halved every 4 years. Thus, fee variability will become more significant for the total reward in the future. Using historical fee data and simulating future halving events, we show the security threshold significantly decreases, as opposed to the state-of-the-art analysis of the constant rewards model (Fig. 2). For example, when simulating 3 halving events, corresponding to about 10 years, the threshold reduces from 0.25 today to 0.2. After another halving, from 0.2 to 0.18 and after another halving to 0.17. This trend continues until the threshold finally reduces to below 0.12. In this scenario, considering the common sizes of miners [3], several miners are incentivized to deviate from the prescribed protocol.

The MEV variability is even higher in smart-contract platforms (e.g., [5], [25]), possibly due to a lively DeFi market [26]. Although their protocols are outside the scope of this work, as they require a more complicated model, with current Ethereum rewards [27] WeRLman shows that the threshold in a Nakamoto blockchain drops to 0.17, also a common size for large miners.

In summary, our main contributions are:

1) A model of blockchains that includes occasional reward spikes and a simplified model that can be solved using traditional dynamic programming methods (§II),
2) WeRLman – a novel deep RL method for finding near-optimal selfish mining strategies in blockchain models with large state spaces (§III),
3) An empirical evaluation of WeRLman showing it achieves near-optimal results (§IV),
4) Finding that fee variability hurts blockchain security by evaluating the security threshold for both models and its sensitivity to their parameters (§V), and
5) An evaluation of model parameters based on realistic distributions of rewards and the resultant security thresholds, well below the known bound (§V-C).

We review previous work on selfish mining, deep RL and other forms of blockchain attacks (§VI) and then conclude (§VII) with a discussion of further open questions.

II. Model

We present our model of the mining process of a Nakamoto-based blockchain with varying fees. We first provide background on the general mathematical model, a Markov decision process (§II-A) and on selfish mining in general (§II-B). Then, we describe our `varying` model of the mining process with varying rewards as an MDP and compare it to the state-of-the-art `constant` model by Sapirshtein et al. [2] (§II-C). Afterwards, we present a simplified version of the varying model, which can be solved using exact methods (§II-D).

[2] Available on GitHub at: https://github.com/roibarzur/pto-selfish-mining

A. Preliminaries: Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical model for a step based single-player game [14], [28]. It consists of an *environment* and an *agent*. Let $\mathcal{S}$ be the *state space* and $\mathcal{A}$ be the *action space*. At every step t, the agent observes the *state* $s_t \in \mathcal{S}$ of the environment and chooses an *action* $a_t \in \mathcal{A}$. The environment then transitions stochastically to a new state s_{t+1} based only on the previous state and action, and receives a reward. Afterward, the process continues to the next step, and so on. An MDP is defined by *transition matrices* $P(s'|s,a)$ (one matrix for each action) which specify the probability distributions of moving from each state s to each state s' under action a, and a *reward matrix* $R(s,a)$ which defines the reward of performing action a in state s.

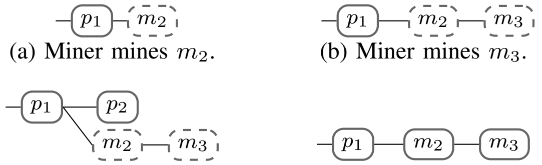
The agent strives to maximize some *objective function* of the per-step rewards by choosing an optimal *policy*. A Markov policy $\pi : \mathcal{S} \to \mathcal{A}$ defines which action to take in each state, independent of previous states or actions. Standard objective functions include the λ-discounted reward $\mathbb{E}\left[\sum_{t=1}^{\infty} \lambda^t R_t\right]$, and the stochastic shortest path $\mathbb{E}\left[\sum_{t=1}^{T} R_t\right]$, where T is a random variable denoting the step in which the process terminates [14].

B. Preliminaries: Selfish Mining

We focus on blockchains based on the original design of Bitcoin [4], which we refer to as Nakamoto blockchains (after their pseudonymous creator). The main goal of a blockchain [29] is to achieve consensus among untrusting parties using only peer to peer communication. Specifically, a blockchain protocol maintains a ledger, which keeps track of all the transactions that took place. A transaction is typically a cryptographically-signed statement ordering a state change, commonly a token transfer between parties. This ensures that participants can only spend tokens they have and cannot spend them more than once.

Principals called miners maintain this ledger. Miners gather transactions from the network and organize them into blocks. Each block points to the block that came before it by including its hash, thus forming a block chain. This creates a timeline of all the transactions. The miners are required to all extend the same chain to ensure there is a consensus. Therefore, miners are expected to extend the longest chain they know of.

To incentivize miners to mine blocks in the longest chain, each block rewards its miner with newly-minted tokens called subsidy. In addition, to incentivize miners to include user transactions, they receive fees from all the transactions they include in a block. Users, who publish transactions to be added to the blockchain, specify the fee to pay to the miner. The higher the fee is, the earlier the transaction will appear in the next few blocks as the block size is limited, so miners will be incentivized to include transactions with higher fees when available.



(a) Miner mines m_2. (b) Miner mines m_3.

(c) Someone else mines p_2. (d) Miner publishes p_2 and p_3.

Fig. 3: Selfish Mining illustration. Secret blocks are marked by a dashed outline.

Mining is made artificially hard by requiring a solution for a cryptographic puzzle, which must be solved to make a block valid. This ensures there will be time for a block to propagate through the network before a new block is created, and that an attacker cannot easily create a long chain to undermine the consensus on the current longest chain. To solve the puzzle, a miner has to guess a number, which when hashed with the block, returns a number lower by some parameter. Thus, the time until a solution is found is distributed exponentially [8], [12], [30] with a rate dependent on how many guesses the miner makes per unit of time.

Nakamoto blockchains also have a mechanism to adjust the difficulty of the puzzles to the amount of total computational power invested in the network by raising or lowering the threshold parameter of the puzzle. This is called a *difficulty adjustment* mechanism. It ensures that even when miners leave or join the network, or as computational power becomes cheaper and more abundant, the expected time between the creation of blocks remains constant. Without this mechanism, blocks would be created more frequently as more computational power joins the network over time. This would leave less time for a block to be propagated before another block is created and would lead to more disputes over the longest chain when two blocks point to the same block, creating a *fork* in the network.

The difficulty adjustment mechanism enables selfish mining [2], [8]. The mechanism ensures a constant average amount of blocks per unit of time, and a rational miner wants to maximize her reward per unit of time. Therefore, a rational miner would want to maximize her fraction of blocks in the main chain rather than the total number of blocks created. An example for a simple case of selfish mining is illustrated in Fig. 3. A miner may want to keep her newly mined block m_2 secret (Fig. 3a) in order to have a chance to find another block, m_3 (Fig. 3b) which points to it and then publish both blocks together. By doing so, other miners waste resources by trying to create block p_2 (Fig. 3c), which will not be part of the longest chain if the miner succeeds in creating m_3 and then publishing m_2 and m_3 together (Fig. 3d). The miner might not always succeed and might lose block m_2, but by sometimes wasting others' work, the miner can increase her fraction of blocks in the longest chain. More generally, we name any form of deviation from the prescribed protocol of following the longest chain and publishing blocks immediately, selfish mining. The main idea behind selfish mining is to risk some rewards but waste more resources for others, and thus increase the miner's share of rewards.

C. Varying Fees Model

We now present our `varying` model of a Nakamoto-based blockchain with varying fees. We model the process of mining from the perspective of a *rational* miner as an MDP [2], [13]. The rational miner mines on a single *secret* chain, and all other miners mine *honestly* on a single *public* chain. Let α be the fraction of the miner from the total computational power of the system. Let γ be the *rushing factor* [2], [8] of the miner: if a miner publishes a block b, the rational miner sees this block and can publish her own block b' that she kept in secret. The rushing factor γ denotes the fraction of mining power (controlled by other miners) that will receive b' before b.

We normalize the baseline block reward to be 1. The baseline includes the subsidy and the fees commonly available in the *mempool*, the set of all available transactions that were not included yet. Occasionally, there appear whale transactions, which offer an additional fee of F. We assume each block can hold a single whale transaction, therefore a block can give a reward of either 1 or $1 + F$. In practice, F would be the aggregation of multiple fees.

To represent the available whale transactions in the network, we introduce the transaction *memory pool* (represented as an integer value *pool*) into the state space. All new whale transactions arrive into the pool (by incrementing the value of *pool*), and can then be included in a block. Honest miners always include a transaction when available. However, the rational miner can choose whether to include a transaction or not when there is one available.

We now present the objective function of the MDP, the state space, the action space and the transitions. Then, we present how we confine the state space to be finite.

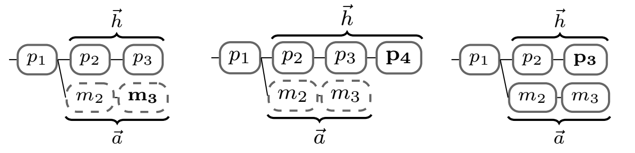
1) Objective Function: In general, the MDP can be modeled such that at step t the miner gains a reward of R_t and the difficulty adjustment mechanism advances by D_t [13]. The rational miner's incentive is to maximize average revenue per unit of time. The difficulty adjustment mechanism ensures that $\sum_{t=1}^{n} D_t$ grows at a constant rate. Thus, dividing the accumulated reward by the sum yields the miner's reward per unit of time.

Let the revenue for a policy π be $\mathbb{E}^{\pi}\left[\lim_{n\to\infty} \frac{\sum_{t=1}^{n} R_t}{\sum_{t=1}^{n} D_t}\right]$, and define the optimal revenue as the maximum of this value.

In previous work on Bitcoin with constant block rewards [2], [10], [13], R_t denoted the number of blocks that the miner added to the longest chain (assuming the block reward is normalized to 1), and D_t denoted the total number of blocks added to the longest chain. The objective then becomes the fraction of blocks in the longest chain mined by the miner.

In the `varying` rewards model, however, the reward R_t includes the transaction fees as well as the subsidy reward. The difficulty advancement D_t remains the same as above, as transactions do not affect the difficulty adjustment mechanism.

2) State Space: The state is a tuple of 4 elements $(\vec{a}, \vec{h}, \mathit{fork}, \mathit{pool})$. The vector $\vec{a}$ is a list of blocks of the rational miner since the last fork. Each item in the list denotes whether the block holds a whale transaction. Similarly, the list $\vec{h}$ keeps the blocks mined in the public chain. The element $\mathit{fork} \in \{\text{RELEVANT}, \text{IRRELEVANT}, \text{ACTIVE}\}$ is a ternary value denoting whether the last blocked was mined by the rational miner, another miner or whether a race is taking place, respectively. The element *pool* is the number of available transactions since the last fork.



(a) IRRELEVANT (b) RELEVANT (c) ACTIVE.

Fig. 4: The different cases of the *fork* parameter.

Note the differences between `varying` model and the `constant` model of Sapirshtein et al. [2]. While *fork* is the same in both models, the element *pool* is new in the `varying` model. In addition, the vectors $\vec{a}$ and $\vec{h}$ are more informative in the `varying` model, and keep track of all blocks with whale transactions, instead of just the length of the chains. Representing the element $\vec{a}$ requires at least $2^{|\vec{a}|}$ bits. This is an exponential cost with respect to the allowed length of the miner's chain. The same applies for $\vec{h}$. This very fact is the reason that the size of the state space in the `varying` model is significantly larger compared to the `constant` model.

Fig. 4 illustrates the different cases of *fork*. The last mined block is in bold. Blocks with a dashed outline are secret. First, in Fig. 4a, m_3 was the last mined block, so $\mathit{fork} = \text{IRRELEVANT}$. In addition, in this case, $|\vec{a}| = \left|\vec{h}\right| = 2$. Second, in Fig. 4b, $|\vec{a}| = 2$, $\left|\vec{h}\right| = 3$ and since p_4 was the last mined block then $\mathit{fork} = \text{RELEVANT}$. Lastly, in Fig. 4c, the miner published her private chain after block p_3 was mined. In this case, $|\vec{a}| = \left|\vec{h}\right| = 2$ and $\mathit{fork} = \text{ACTIVE}$.

3) Actions and Main Transitions: The action space in the `varying` model cannot be trimmed as in the `constant` model, and we must consider more possible actions. The available miner actions and the resultant transitions in the `varying` model are as follows. For clarity, the following explanation of the transitions is divided into 3 parts. The transitions of $\vec{a}$ and $\vec{h}$ are described with the actions, as these are the main changes in the state. Then, the transitions of *fork* are described separately, as they do not depend on $\vec{a}$ and $\vec{h}$. For a similar reason, the transitions of *pool* are described afterwards.

a) Adopt ℓ: The miner discards her private chain, adopts the first ℓ blocks of the public chain and continues mining privately on the last adopted block. Formally, (1) $\vec{a} \leftarrow [\ ]$, (2) $\vec{h} << \ell$ (shifted by ℓ, discarding the first ℓ blocks), and (3) $D_t \leftarrow \ell$ as the difficulty adjustment mechanism advances by ℓ blocks.

In the `constant` model, allowing the miner to adopt exactly $\left|\vec{h}\right|$ blocks suffices. However, in the `varying` model, since some block might hold an extraordinary reward, the miner might be incentivized to discard all her previous work and deliberately compete for a specific block.

b) Reveal ℓ: This action is legal only when the miner's private chain is at least as long as ℓ. The miner reveals the first ℓ blocks of her private chain, either causing an active fork when $\left|\vec{h}\right| = \ell$ and *fork* = RELEVANT or overriding the public chain when $\ell > \left|\vec{h}\right|$. In the latter case, the alternative public chain is discarded by all other miners, and they start mining on the end of the miner's revealed chain. The miner is rewarded by ℓ plus the number of fees in the first ℓ blocks. In addition, (1) $\vec{a} << \ell$, (2) $\vec{h} \leftarrow [\ ]$, and (3) $D_t \leftarrow \ell$.

In the constant model, the miner is only allowed to reveal either $\left|\vec{h}\right|$ blocks to cause an active fork or exactly $\left|\vec{h}\right| + 1$ blocks. However, in the varying model, the miner might want to also secure a block with a whale fee by preemptively revealing it in addition to $\left|\vec{h}\right| + 1$ blocks. The possibility to adopt or reveal an arbitrary number of blocks is the reason that the state must save $\vec{a}$ and $\vec{h}$ as vectors rather than only keep track of the number of transactions. Otherwise, after adopting or revealing some blocks, there is no way of knowing how many transactions are included in $\vec{a}$ and $\vec{h}$ in the next state.

c) Wait f: Keep mining. After this action is performed, a new block is mined and added to either the secret chain of the rational miner or to the public chain, depending on who mined this block. The binary parameter f specifies whether the rational miner attempts to include a whale transaction to the block she is trying to mine. So, it indicates that if she is the one to find a block next, that block will hold a transaction if there is one available.

When there is no active fork in the network, i.e., *fork* $\neq$ ACTIVE, the block is mined by the rational miner w.p. α or is mined by someone else w.p. $1 - \alpha$. When there is an active fork in the network, namely when *fork* = ACTIVE, for example, the case in Fig. 4c, a fraction γ of the honest miners received the rational miner's last published block (block m_3 in the example) first, so they try to extend it. The rest of the miners did not receive it first, but instead received a block mined by an honest miner first (block p_3 in the example). So, a fraction of $1-\gamma$ of the honest miners try to extend the honest chain.

Thus, there are 3 possible outcomes of an active fork. The first is that w.p. α the rational miner is the one to mine the next block. In that case, the active fork in the network remains, as the rational miner only extended her secret chain. Another possibility is that w.p. $\gamma(1 - \alpha)$ one of the honest miners manages to mine a block extending the rational miner's chain and publishes the block immediately. In this case, The miner keeps mining on his private chain and the rest mine on the newly created block, discarding the public chain and accepting the published part of the rational miner's chain (those would be blocks m_2 and m_3 in Fig. 4c). In this case, (1) $\vec{a} << \left|\vec{h}\right|$, and (2) $\vec{h} \leftarrow [\ ]$. The last option is that w.p. $(1-\gamma)(1-\alpha)$ one of the honest miners mines a block which extends the honest chain and publishes it. In this case, the rational miner keeps mining on his private chain while everybody else keeps mining on the newly created block.

Besides the option to add a fee to the miner's block or not, the Wait action and transitions are identical in both the constant and the varying model.

4) Transitions of the Fork State: The transitions of *fork* are the same as in the constant model. If there is no active fork and the last block was mined by the rational miner, then *fork* is IRRELEVANT, meaning that the miner cannot match as all other miners already received the last honest block. If the last block was mined by an honest miner, then *fork* is RELEVANT as the miner already has a block ready and can immediately publish it, so it will reach a fraction of γ of the miners before the other block. If there is an active fork and then an honest miner finds the next block, then *fork* becomes RELEVANT. Otherwise, if the rational miner finds the next block, then *fork* does not change since the miner doesn't publish her new block.

5) Transitions of the Pool: The *pool* element is not present in the constant model, and thus its transitions are novel in the varying model. We assume the block creation process is a Poisson process with rate δ, independent of the block creation process. However, our model is event-based, i.e. the miner makes decisions immediately after block creation events. Thus, in the model we assume that transactions can only arrive after each block creation event, with some probability. But, since the rate of block creation depends on the rational miner behavior, it is not trivial to synchronize the transaction creation in our model.

Our insight is that the difficulty adjustment mechanism ensures that the rate of block creation in the main chain is constant. Thus, if we define the height as the distance of the block from the first block in the blockchain, the height of the longest chain increases at a constant rate. Therefore, we sample a new transaction after a new block of a new height is created. For example, we would sample the process after the first block out of blocks m_3 and p_3 in Fig. 4a is created. Thanks to the difficulty adjustment mechanism, the time between samples is distributed exponentially with a mean of 1 unit of time. Sampling the Poisson process at a time of block creation is the same as sampling a geometric variable with mean δ (See Appendix A). In practice, however, since we only consider small δ values, we simulate this as only a Bernoulli experiment with a transaction arrival chance of δ and neglect terms of order δ^2 in the probability.

6) Bounding the State Space: In practice, it is easier to work with finite state space MDPs. Therefore, following previous work [2], [10], [13], we bound the state space of our model.

Similarly to the constant model, we bound the maximum fork length by enforcing that $\max\{|\vec{a}|, |\vec{h}|\} <$ *max_fork*. Practically, all actions that would lead to either $\vec{a}$ or $\vec{h}$ to be longer than the maximum value cannot be used, and instead the miner would have to reveal or adopt some blocks to reduce the length of the chains. This also discourages undesired behaviors of the miner, for example the ability to keep waiting forever and only extending her secret chain. Previous empirical results [13] show that increasing the maximum fork beyond a certain point gives a small and diminishing improvement of the revenue.

In addition, we bound the maximum possible pool size. Whenever a whale transaction arrives and causes the number of available transactions to exceed the limit, the new transaction is simply discarded and *pool* is left unchanged. We choose the maximum pool size to be the same as the maximum fork size. Note that in the `constant` model, there is no need to limit the pool size, as there is no pool. Increasing the maximum pool size also has a negligible effect, since whale transactions are rare and provide a strong incentive for the miner to include them, making it unlikely that many available transactions will be left unused and overflow the pool.

The elements $\vec{a}$ and $\vec{h}$ are binary vectors up to size *max_fork*. The *fork* value is ternary, and *pool* is an integer between 0 and the maximum allowed size. We get that the number of states in the `varying` model is:

$$|\mathcal{S}| = \left(2^0 + 2^1 + ... + 2^{\mathit{max\_fork}}\right)^2 \cdot 3 \cdot (\mathit{max\_pool} + 1) \approx \approx 12 \cdot \mathit{max\_pool} \cdot 4^{\mathit{max\_fork}} . \quad (1)$$

D. Simplified Varying Model

We introduce a `simplified` model, which is based on the `varying` model, but allows the rational miner only a subset of the possible actions. Instead of allowing the miner to reveal or adopt an arbitrary amount of blocks, the `simplified` model allows the miner to reveal only h or $h+1$ blocks or adopt exactly h blocks. Thanks to this constraint, the positions of transaction in the chain can be ignored and instead, only the number of included transactions needs to be known.

Thus, this model can be represented more efficiently and decreases the size of the state space to about $500,000$. We can then use PTO [13] to find the optimal solution in this model. Because the `simplified` model only reduces the freedom of the rational miner, optimal policies found in the `simplified` model provide a lower bound on the optimal policy possible in the full model. The full details of this model are described in Appendix B.

III. Method

To analyze whether selfish mining is beneficial, we must first find the miner's optimal strategy, and then check whether it achieves more than her fair share. To find the optimal policy, all the blockchain models described in the paper can be transformed into a standard MDP form with a linear objective function. The `constant` rewards model and the `simplified` model can then be solved using conventional dynamic programming methods, as in previous work [2], [10], [13]. The `varying` model, however, cannot be solved using standard techniques due to its large state space.

In this section, we present WeRLman, an algorithmic framework based on deep RL for solving the `varying` model for Nakamoto based blockchains. We begin the section by explaining the transformation we use to linearize the objective function (§III-A). We proceed to describe Q-learning (§III-B) and Monte Carlo Tree Search (MCTS) (§III-C), popular algorithms that WeRLman builds upon. We conclude by describing our RL algorithm that is specialized for transformed MDPs, and exploits domain knowledge for improved accuracy (§III-D). The full details of our implementation of WeRLman and the configuration used to obtain our results is in Appendix. C.

A. Transformation to Linear MDP

As described in Section II, the objective function of a rational miner is an expected ratio of two sums. This objective is different from standard MDP objectives that are based on a linear combination of the step rewards [28]. To overcome this difficulty, we transform our MDP to a linear objective MDP using Probabilistic Termination Optimization (PTO) [13]. In a nutshell, PTO replaces the denominator in the objective function with a constant, and in addition, modifies the MDP transitions with a chance to terminate the process at each step. The termination probability for each transition is given by $p = (1 - \frac{1}{H})^D$, where D depends on each transition, and H is a controlled parameter that is constant for all transitions and is termed the *expected horizon*. The expected horizon becomes the new denominator in the objective function, and the transformed MDP is a conventional stochastic shortest path problem [14]. The intuition behind this approach is that, on one hand, the process stops after the denominator reaches H in expectation. On the other hand, the memory-less termination probability guarantees that a Markov policy, (action depends on the current state only) is optimal, thus the state space does not need to be expanded to calculate an optimal policy. The stochastic shortest path can be further represented as a discounted MDP with a state-dependent discount factor [14], and we use this representation in our algorithms. This means that instead of giving the process a chance of p to terminate, we multiply all future rewards by $\lambda = 1 - p$. Thus, the expectation of the rewards stays the same and the terminal state is no longer necessary. In the remainder of this section, we describe RL algorithms for discounted MDPs, with the understanding that using the transformation above, they can be applied to our problem setting.

B. Q-Learning

Q-learning is an RL algorithm for solving MDPs that is based on learning state-action values, a.k.a. Q values [14]. The Q values $Q^*(s, a)$ are defined as the expected discounted reward when starting from state s, taking action a, and then proceeding optimally. The Q values are known to satisfy the Bellman Equation:

$$Q^*(s,a) = R(s,a) + \mathbb{E}_{s' \sim P(\cdot|s,a)} \left[\lambda \max_{a' \in \mathcal{A}} Q^*(s',a') \right] = = R(s,a) + \sum_{s' \in \mathcal{S}} \lambda P(s'|s,a) \max_{a' \in \mathcal{A}} Q^*(s',a') , \quad (2)$$

where $0 \leq \lambda < 1$ is the discount factor. Given the Q values, an optimal policy π^* can be easily obtained using:

$$\pi^*(s,a) = \arg\max_{a \in \mathcal{A}} \{Q^*(s,a)\} . \quad (3)$$

The Q-learning algorithm is based on this property. It requires a simulator of the MDP which is used to sample pairs of states and actions (s_n, a_n). It starts at an initial guess of the Q values Q_0 and iteratively updates the values using step sizes β_n. Formally, for $n = 1, 2, ...,$ it performs:

$$
\begin{aligned}
Q_n(s_n, a_n) \leftarrow Q_{n-1}(s_n, a_n) + \beta_n \Big(R(s_n, a_n) + \\
+ \mathbb{E}_{s' \sim P(\cdot|s_n, a_n)} \left[\lambda \max_{a' \in \mathcal{A}} Q_{n-1}(s', a') \right] - Q_{n-1}(s_n, a_n) \Big) .
\end{aligned}
\tag{4}
$$

When a stopping condition is satisfied, the algorithm stops and the policy can be extracted by Eq. 3. For finite state and action spaces, and as long as all state action pairs are sampled infinitely often, the algorithm is guaranteed to converge under a suitably diminishing sequence of step sizes β_n [14].

For large or infinite state spaces, RL algorithms approximate the Q values, as in the popular Deep Q Networks algorithm (DQN [22]), which employs deep neural networks as the function approximation. To train the network, a simulator of the MDP is again used to sample state-action pairs $\{(s_n, a_n)\}_{n=1}^{N}$. Then, *target values* for the network are calculated using the current network Q_θ, where θ denotes the parameters (weights and biases) of the neural network:

$$
t_n = R(s_n, a_n) + \mathbb{E}_{s' \sim P(\cdot|s_n, a_n)} \left[\lambda \max_{a' \in \mathcal{A}} Q_\theta(s', a') \right] . \tag{5}
$$

The target values are then used to train the neural network by minimizing the mean squared error loss, using stochastic gradient descent, typically in batches [31]. The loss function is then calculated for each batch $B \in \{1, 2, ..., n\}$: $L(\theta) = \frac{1}{|B|} \sum_{n \in B} (t_n - Q_\theta(s_n, a_n))^2$, and used to update θ iteratively. After the neural network is trained, the algorithm repeats by collecting more data and retraining the network.

In the typical RL scenario, the transition model of the MDP is not known to the agent, and the expectations in Equations 4 and 5 are replaced with a sample from the simulator $s' \sim P(\cdot|s_n, a_n)$, giving an unbiased estimate of the update rule [14], [22]. For example, Eq. 5 would be rewritten using the observed reward r_n and next state s'_n as $t_n = r_n + \lambda \max_{a' \in \mathcal{A}} Q_\theta(s'_n, a')$.

C. Multi-step Look-Ahead using MCTS

WeRLman exploits the full knowledge of the transition model by replacing the 1-step look-ahead in the Q-learning update rule with an h-step look ahead [32]. Specifically, the Q-learning targets in Eq. 5 are replaced by:

$$
\begin{aligned}
t_n = \max_{\pi_0, \dots \pi_h} \mathbb{E}_{|\cdot}^{\pi_0, \dots \pi_h} \Bigg[\sum_{t=1}^{h-1} \lambda^{t-1} R(s_n^{(t)}, \pi_t(s_n^{(t)})) + \\
+ \lambda^h Q(s_n^{(h)}, \pi_h(s_n^{(h)})) \Bigg]
\end{aligned}
\tag{6}
$$

where $s_n^{(0)} ..., s_n^{(t)}$ denote a possible trajectory of states and the notation $\mathbb{E}_{|\cdot}^{\pi_0, \dots \pi_h}$, indicates an expectation conditioned on the starting state $s_n = s_n^{(0)}$, the action $a_n = \pi_0(s_n^{(0)})$, and the choice of actions $\pi_1(s_n^{(1)}), ..., \pi_h(s_n^{(h)})$. This modification significantly improves the accuracy, as approximation errors in Q_θ have less impact as h is increased due to the discounting. However, the complexity of solving Eq. 6 quickly increases with the horizon (it is $O\big(|\mathcal{A}|^h \big)$ [33]). In order to handle a relatively large h, which we found is critical for the accuracy required for the blockchain domain, we propose a sampling scheme to approximate the optimization in (6), building on Monte Carlo Tree Search (MCTS).

MCTS is a sampling-based planning method that has seen wide success in solving large decision-making problems [23], [24], [34], [35]. The main idea is to plan ahead by randomly simulating many possible trajectories that start from the current state of the system, and use these simulated outcomes to select the best action to take. MCTS can be thought of as a sparse sampling method to calculate the h-step look ahead. Each trajectory can provide a sample for the value of an h step trajectory (the term inside the expectation in Eq. 6). Thus, the values of all trajectories starting from a specific action a can be averaged to obtain an estimate of the value of action a in the current state.

There are many flavors of MCTS [35]; here, we focus on the AlphaGo Zero implementation [23]. An optimistic exploration mechanism (described in the following paragraphs) steers the samples towards high-reward trajectories, approximating the $\max$ operation in (6). In addition, AlphaGo Zero uses samples more efficiently by maintaining a graph of all visited states. Each node represents a state s and stores, for each action a, the number of times the action was selected $N(s, a)$, the prior probability $P(s, a)$, the Q value of the neural network $Q_\theta(s, a)$, and the MCTS estimated Q value $W(s, a)$. When creating a node for a state s for the first time, $W(s, a)$ is initialized to be $Q_\theta(s, a)$. After a trajectory $s^{(0)}, \dots, s^{(h)}$ with actions $a^{(0)}, \dots, a^{(h-1)}$ and rewards $r^{(0)}, \dots, r^{(h-1)}$ is simulated, the algorithm performs a *backup* to update the values $W(s, a)$:

$$
\begin{aligned}
W(s^{(t)}, a^{(t)}) \leftarrow W(s^{(t)}, a^{(t)}) + \\
+ \beta(r^{(t)} + W(s^{(t+1)}, a^{(t+1)}) - W(s^{(t)}, a^{(t)})) .
\end{aligned}
\tag{7}
$$

Thus, the MCTS estimates from every simulation are preserved, and used later to update other estimates more accurately.

AlphaGo Zero requires the neural network to generate two types of output. The first is an estimate for the Q values $Q(s, a)$, which are the initial values for $W(s, a)$, and the second is the prior probability $P(s, a)$ of an action a in state s, which guides the action sampling during the trajectory simulation. In each of the simulated trajectory steps, an action that maximizes $W(s, a) + U(s, a)$ is chosen, where U is an exploration factor given by:

$$
U(s, a) = P(s, a) \cdot \frac{\sqrt{N(s)}}{1 + N(s, a)} , \tag{8}
$$

where $N(s)$ is the accumulated number of visits to the state s, and $N(s, a)$ the number of times action a was chosen in state s. Intuitively, the exploration factor starts at a high value and decreases for state action pairs that have been chosen more often.

Similarly to DQN, the neural network is trained by stochastic gradient descent. Unlike DQN, the AlphaGo Zero loss function has an additional term to train the prior probabilities, represented by another neural network with parameters ϕ. The prior probabilities are trained to minimize their cross entropy with respect to the actual action selection distribution in state s_n determined by $N(s, \cdot)$. The loss function is given by:

$$L(\theta, \phi) = \frac{1}{|B|} \sum_{n \in B} \Bigg[\left(t_n - Q_\theta(s_n, a_n)\right)^2 + \\ + \sum_{a \in \mathcal{A}} \frac{N(s_n, a)}{N(s_n)} \log P_\phi(s_n, a) \Bigg] . \quad (9)$$

The target value t_n should approximate the value of a state. In the original AlphaGo Zero algorithm, which was developed for turn-based games, t_n is given by the result (win/lose) of a full game simulation. In WeRLman, since the problem horizon is infinite (when treating the termination chance as a discounting factor), we instead use bootstrapping, and choose t_n to be $W(s_n, a_n)$, the approximate value function estimated during MCTS.

Additionally, we utilize our knowledge of rewards and transition probabilities in the WeRLman MCTS backup update, and replace Eq. 7 with:

$$W(s_t, a_t) \leftarrow W(s_t, a_t) + \beta \Big(R(s_t, a_t) + \\ + \mathbb{E}_{s' \sim P(\cdot|s_t, a_t)} \left[\max_{a'} W(s', a') \right] - W(s_t, a_t) \Big) . \quad (10)$$

D. Neural Network Scaling

The quality of the RL solution depends on the accuracy of the value function approximation Q_θ. Most deep RL studies focused on domains where the differences in value between optimal and suboptimal choices are stark, such as in games, where a single move can be the difference between winning or losing the game. In such domains, it appears that the accuracy of the neural network approximation is not critical, and most of the algorithmic difficulty lies in exploring the state space effectively. In the blockchain domain, however, the difference in value between a selfish miner and an honest one can be very small. We found that the neural network approximations in conventional deep RL algorithms are not accurate enough for this task. In this section, we propose two scaling tricks that we found are critical to obtain high-accuracy results in deep RL for blockchain.

1) Q Value Scaling: We use special domain knowledge about the transformed MDP to manually set the scale of the Q values. The following proposition provides the reasoning behind this approach.

Proposition 1. *Let $Q^*(s, a)$ be the optimal action values in an MDP obtained by a PTO transformation with expected horizon H. Let π^* be the optimal policy and s_{init} be the initial state of the MDP. Then, it holds that for any state $s \in \mathcal{S}$:*

$$|Q^*(s, \pi^*(s)) - Q^*(s_{\mathit{init}}, \pi^*(s_{\mathit{init}}))| = O\left(\frac{1}{H}\right) . \quad (11)$$

The proof is deferred to Appendix D.

Based on Proposition 1, we can expect the optimal Q values for all states and actions to be centered around the Q value of the initial state. Therefore, the first improvement we introduce is the *base value*: instead of using the neural network output as the values of state actions pairs, we interpret it as the difference from a base value ρ. In addition, Prop. 1 also implies that the standard deviation of the values is of similar magnitude to $\frac{1}{H}$. So, we divide the neural network output by the expected horizon. In practice, we change the initialization of W to be:

$$W(s, a) = \rho + \frac{Q(s, a)}{H}, \quad (12)$$

instead of simply $Q(s, a)$.
We calculate the base value as the average revenue in separate Monte-Carlo simulations utilizing MCTS according to the current neural network. To train the neural network appropriately, we first subtract the base value from all calculated target values, and then multiply them by the expected horizon. Formally, instead of taking $W(s, a)$ as the target value, we calculate the target value t_n by:

$$t_n = H \cdot W(s, a) - \rho . \quad (13)$$

This improvement accelerates the training by jump-starting the perceived values closer to the true values. Otherwise, many steps of bootstrapping would be necessary for the values to reach the same magnitude. In addition, this improvement focuses the neural network on learning the *relative differences* between state values, which are much more sensitive than the magnitude of the values.

At this point, the careful reader should ask whether the base value formulation of Q-learning impacts the convergence of the method. In Appendix E, we show that for finite state and action MDPs, our base value modification still converges to the optimal solution. In our experiments with deep neural networks, we did not encounter any convergence problems when using the base value method.

2) Target value normalization: Our second improvement is target value normalization. During training, we track the average of the target values. When sampling a batch for SGD, namely a set of states, actions and previously calculated target values $\{(s_n, a_n, t_n)\}_{n \in B}$, we subtract the average target value from all values in the batch and only then proceed to train the neural network. Formally, we calculate:

$$t'_n = t_n - \frac{1}{|B|} \sum_{n \in B} t_n . \quad (14)$$

Then, we use t'_n instead of t_n in the loss function (Eq. 9). This ensures the network learns values with an average of 0, so the magnitude of the action values is controlled solely by the base value. This is inspired by Pop-Art normalization, an adaptive normalization technique specialized for learning values in a scale invariant manner [36]. The main differences of WeRLman from Pop-Art normalization are that (1) in Pop-Art normalization the values are also divided by their standard deviation, and (2) the average and standard deviation

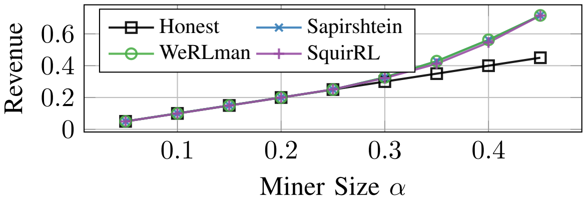


Fig. 5: The revenue of WeRLman in the `constant` model versus state-of-the-art results for different miner sizes.

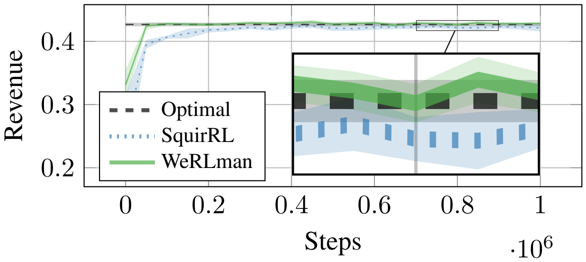


Fig. 6: Learning curves of WeRLman and SquirRL in the `constant` model averaged over 20 runs. The shaded areas represent 97.5% confidence intervals. Up to a 2.5% statistical error, WeRLman is consistently optimal, while SquirRL is suboptimal.

are learned from the sampled values rather than the previous blockchain specific scaling method we described.

IV. Evaluating WeRLman's Accuracy

To motivate the use of WeRLman for our blockchain security calculations, in this section, we show that WeRLman obtains near-optimal results in simpler problems where optimal values can be calculated exactly. We begin by demonstrating that WeRLman can be used on the `constant` model to recreate known optimal results and surpass state-of-the-art deep RL results (§IV-A). We then perform an ablation study of the techniques we presented, namely the manual neural network scaling and target normalization, and show these are crucial to our framework (§IV-B). Finally, we test WeRLman in the `varying` model and show it manages to obtain near-optimal results in a toy setting (§IV-C).

A. Constant-Rewards Model

To demonstrate the accuracy of WeRLman, we first use it in the `constant` reward model. For this model, the optimal revenues were calculated by Sapirshtein et al. [2]. We compare WeRLman with SquirRL, state-of-the-art deep RL for blockchain [10], for various miner sizes using $\gamma = 0.5$ and a maximum fork of 10 (Fig. 5). We use Monte Carlo simulations to evaluate the revenues of the policies obtained by both WeRLman and SquirRL. WeRLman outperforms SquirRL in the `constant` model. We do not draw the confidence intervals in Fig. 5, as they are too small at the plotted resolution.

In addition, we simulate the revenues of policies obtained by both WeRLman and SquirRL during their training processes (Fig. 6). The simulated revenues are averaged over 20 runs, all using $\alpha = 0.35, \gamma = 0.5$ and a maximum fork of 10. The shaded areas represent 97.5% confidence intervals. In this comparison only, to reduce the effect of computing power differences as much as possible, we use a weakened version of WeRLman with minimal multiprocessing – only 1 of each process type (see Appendix C). It is apparent that SquirRL usually plateaus at a close yet suboptimal policy. On the other hand, WeRLman consistently outperforms SquirRL and obtains revenues much closer to the optimal revenue[3]. This shows that WeRLman's consistently superior accuracy does not stem from additional running time or computing power, but rather from the inherent differences between the 2 methods, namely, the use of MCTS and the variance reduction techniques we introduce (§III-D). In addition to WeRLman's superior performance, when comparing SquirRL to WeRLman, another more delicate point should be kept in mind: WeRLman builds on PTO, and as such does not require any prior knowledge of the optimal revenue. SquirRL, on the other hand, builds on the iterative MDP solution of Sapirstein et al. [2], where the reward in each MDP is based on the previously computed optimal revenue. In practice, SquirRL replaces the iterative computation by plugging in an approximately optimal revenue from [8]. While this approach works relatively well for the `constant` model, it is not clear how to extend it to blockchains for which an approximate solution is not available.

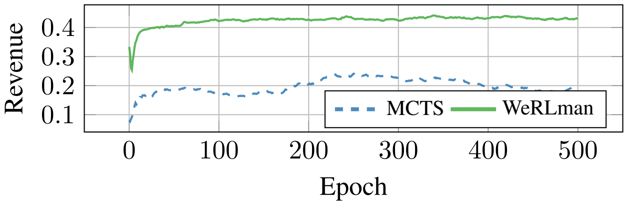


Fig. 7: The revenue of WeRLman compared to simple MCTS using the `constant` model.

B. Ablation Study

To test the effect of our novel scaling techniques on the performance of the algorithm, we compare the revenues obtained by WeRLman to the revenue obtained by simple MCTS. More precisely, we compare 2 runs of the algorithm, both starting from the same random seed and running for 500 epochs. The first run employs MCTS only. The second run utilizes MCTS and the novel manual scaling and normalization we introduce. Both runs use the `constant` model with $\alpha = 0.35, \gamma = 0.5$ and a maximum fork of 10.

The results are plotted in Fig. 7. The manual scaling of values, together with the target normalization, immediately transforms the Q values to the desired magnitude. Notably, an immediate improvement of the policies is attained overall, and WeRLman greatly surpasses simple MCTS.

C. Toy Setting of Varying Model

In the `varying` model, we first test WeRLman in a toy configuration. In this configuration only, we reduce the maximum fork size and pool size to be 3. This constraint limits the state space to approximately 2,000 states (Eq. 1)

[3] WeRLman cannot achieve more than the optimal revenue. But, it does get close enough to the optimal policy to sometimes appear as if it outperforms the optimum due to simulation noise.

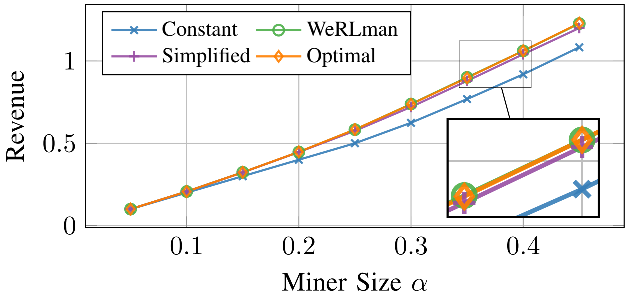


Fig. 8: The revenue as a function of the miner size α in a toy setting with a maximum fork of 3. We compare the `varying` and the `simplified` models with $F = 10$ and $\delta = 0.1$.

and thus allows us to find the optimal policy of the `varying` model precisely with PTO [13].

We compare the revenues obtained using WeRLman to the optimal revenue for different miner sizes (Fig. 8). In addition, we compare WeRLman to the results obtained in the `constant` model, scaled appropriately using the following proposition.

Proposition 2. *Let π be policy in the* `constant` *model and let π' be its fee-oblivious extension to the* `varying` *model, in which the miner always includes a transaction when available. If ρ_π, and $\rho_{\pi'}$ are the revenues of π and π' in the respective models, then $\rho_{\pi'} = (1 + \delta F)\rho_\pi$.*

The proof is deferred to Appendix F.

WeRLman achieves the optimal results under this model as well, consistently surpassing the results from the `simplified` and `constant` models. This suggests that WeRLman is capable of accurately obtaining optimal results in the more complex `varying` model with a larger maximum fork. WeRLman's accuracy in the two distinct models also suggests it may generalize well to other blockchain protocols.

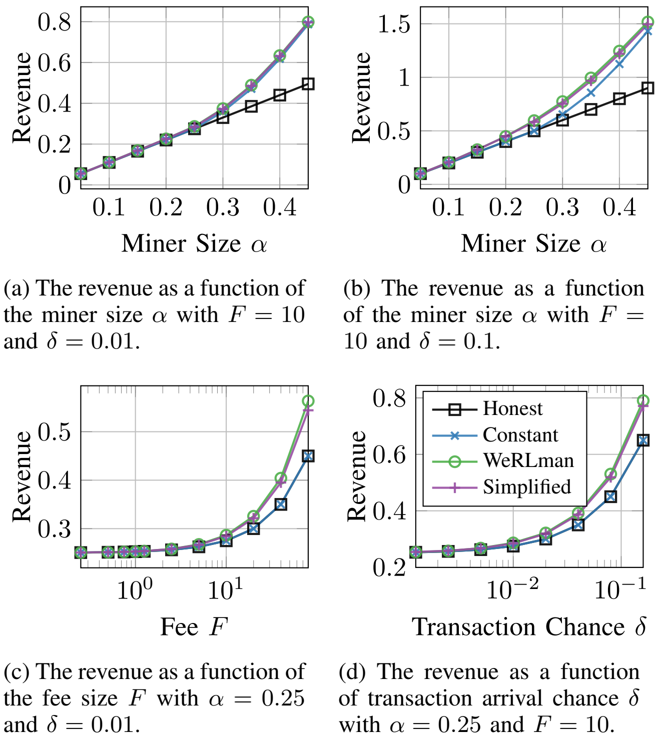
V. Security Threshold Analysis

After establishing WeRLman as a reliable tool for approximately solving blockchain MDPs, we now turn to analyze the security of a Nakamoto blockchain. We begin by using WeRLman for the `varying` model in a variety of configurations to test how model parameters affect the optimal miner revenue (§V-A). Next, we bound the security threshold of a Nakamoto blockchain in the `varying` model combining WeRLman and the `simplified` model (§V-B). Finally, we consider specific parameter values based on statistics from operational networks and find the resultant security thresholds (§V-C).

A. Revenue Optimization

We evaluate the policies obtained using WeRLman in the `varying` model with a maximum fork of 10. Increasing the maximum further yields only a negligible improvement, especially for miner sizes close to the security threshold [13].

We run WeRLman with a variety of configurations, testing its sensitivity to the miner size α (Fig. 9a and 9b), the whale fee amount F (Fig. 9c) and the whale fee frequency δ (Fig. 9d).



(a) The revenue as a function of the miner size α with $F = 10$ and $\delta = 0.01$.

(b) The revenue as a function of the miner size α with $F = 10$ and $\delta = 0.1$.

(c) The revenue as a function of the fee size F with $\alpha = 0.25$ and $\delta = 0.01$.

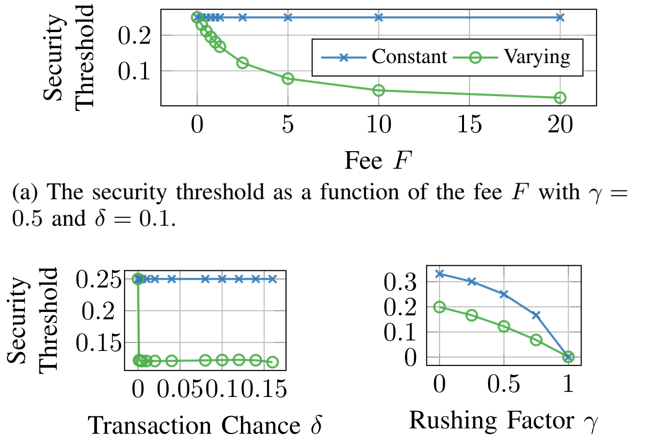
(d) The revenue as a function of transaction arrival chance δ with $\alpha = 0.25$ and $F = 10$.

Fig. 9: Revenue in the full model with $\gamma = 0.5$ and a maximum fork of 10.

For each configuration, we ran WeRLman on 64 CPU cores between 6 and 12 hours using our parallel python implementation (see Appendix C). The honest revenue was obtained based on Prop. 2. In contrast to the toy setting, the optimal revenue in the full model cannot be calculated precisely due to its large MDP with about 10^8 states (Eq. 1). However, like in the toy setting, WeRLman still surpasses the optimal revenues of the `constant` model (scaled using Prop. 2) and the `simplified` model. We observed several instances where the optimal policy utilized the richer action space in the `varying` model compared to the `simplified` model. The learned policy sometimes adopted all blocks but the last one or two blocks to try to overtake the public chain and steal a transaction fee (see Appendix G).

Figures 9a and 9b indicate the advantage of WeRLman is more notable when increasing the miner size. Furthermore, increasing the fee size also results in a significantly higher revenue for the results of WeRLman compared to the `constant` model (Fig. 9c). In addition, increasing δ to 0.1 results in a more significant incentive of a miner to mine selfishly in the `varying` model and the `simplified` model compared to the `constant` model. This is apparent directly from Fig. 9d or by observing the differences in Fig. 9b are more substantial than those in Fig. 9a. This can be explained by the increased frequency of whale fees, allowing more opportunities to deviate and obtain extra reward.

Fig. 9b also clearly shows a significant change in the security threshold. Under the `constant` method, the threshold is 0.25 [2]. This can be seen as the point in which the optimal



(a) The security threshold as a function of the fee F with $\gamma = 0.5$ and $\delta = 0.1$.

(b) The security threshold as a function of the transaction arrival chance δ with $\gamma = 0.5$ and $F = 2.5$.

(c) The security threshold as a function of the rushing factor γ with $F = 2.5$ and $\delta = 0.1$.

Fig. 10: Security thresholds in the `varying` model compared to the `constant` model with a maximum fork of 10.

revenue in the `constant` model surpasses the line for honest mining. In contrast, this deviation occurs at around 0.1 for the results of WeRLman and the `simplified` model. This significant threshold difference is due to the large F value.

B. Security Threshold

To obtain the security threshold of a blockchain protocol, we perform a binary search over the miner size, looking for the minimal size where the optimal strategy is not the prescribed one. With the `constant` and `simplified` models, we can compare policies directly.

However, in the `varying` model, directly comparing policies is intractable: WeRLman does not output an explicit policy, as this would be infeasible due to the size of the state space. Thus, we estimate the revenue of the policy via Monte Carlo simulations and compare revenues. The outputs of the simulations are noisy, and the deep RL training is not guaranteed to find the optimal strategy, introducing another source of uncertainty. Therefore, we can only strive to obtain an upper bound on the security threshold. We address these issues in two ways. First, we consider a policy to be better than honest mining only if its revenue advantage is statistically significant based on a Z-test with a confidence of 99%.

Second, instead of performing a binary search, we perform an iterative search where in each stage we sample 4 different sizes of the miner. We check the results in each stage are consistent, meaning that lower miner sizes provide honest policies and higher miner sizes provide selfish policies. If the stage results are consistent, we focus on a smaller region, similarly to a binary search. However, if the stage results are inconsistent, meaning we found $\alpha_1 < \alpha_2$, such that the Z-test for α_1 is positive (selfish mining is possible) but the Z-test for α_2 is negative (selfish mining is not possible), we checked an additional value $\alpha_1 < \alpha' < \alpha_2$ based on the current resultion. If the Z-test for α' is positive, we deem the Z-test for α_1 to be a true positive and the test for α_2 to be a false negative. Otherwise, if the test for α' is negative, we consider the tests to be a false positive and a true negative respectively. Then we proceed to focus on a smaller region in a similar manner. In practice, we only rarely encounter inconsistencies. We stop the search once we reach 3 digits of precision. A more accurate calculation of the threshold is only a matter of using more computational power.

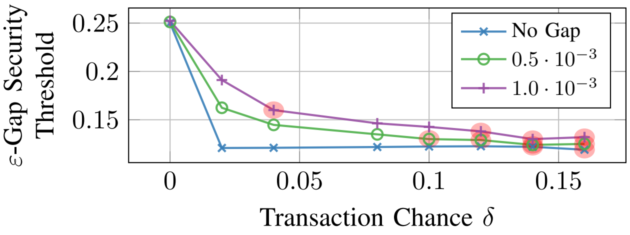


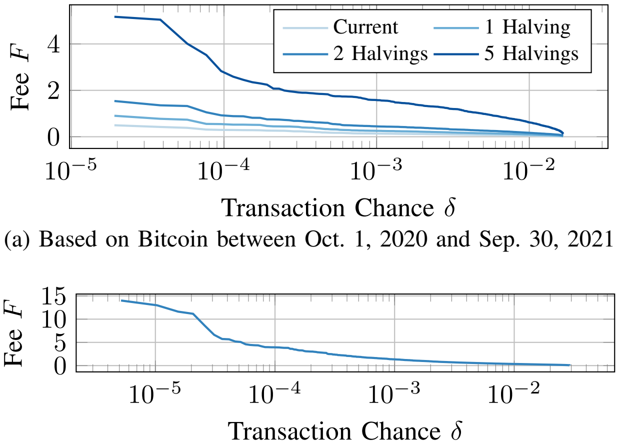
Fig. 11: Security thresholds in the `varying` model for $\gamma = 0.5$, $F = 2.5$ for various values of δ when considering different gaps. Points marked in red were obtained using WeRLman.

Thresholds obtained according to the `simplified` model also provide only an upper bound on the true security threshold because the `simplified` model constrains the rational miner. We utilize both upper bounds to yield a single upper bound of the `varying` model by taking the minimum of both. We compare the security thresholds of the `constant` model and the `varying` model and test their sensitivity with respect to the fee size F (Fig. 10a), the transaction arrival chance δ (Fig. 10b), and the rushing factor (Fig. 10c).

Fig. 10a shows that increasing the fee size rapidly decreases the threshold in the `varying` model. In contrast, increasing the transaction arrival chance (Fig. 10b) has almost no effect on the threshold of the `varying` model. This could be because the main difference of the optimal revenue between the `constant` and `varying` models depends primarily on the miner's behavior when there is a single fee available. When no fees are available, the behavior of the miner is similar to the `constant` model. However, when there is a single fee available, the miner might try to obtain it by mining selfishly (depending on the miner size) and when the miner has a good chance to succeed, it can increase its revenue regardless of the value of δ. The case when there is more than a single fee available is rare (happens with probability of less than δ^2) and thus has a low impact on the miner's revenue.

In many configurations, WeRLman's optimal strategies are significantly better than `simplified` (Fig. 9). However, the threshold obtained with `simplified` is often lower. The reason is that when using WeRLman, the search requires a revenue of a policy to be higher than the fair share by more than the confidence interval in order for us to determine that the optimal policy is selfish. However, close to the threshold, selfish mining nets a negligible gain in revenue, resulting in a false negative.

But aiming for the threshold point ignores the cost of using a nonstandard protocol. We therefore define the *ε-gap security threshold* to be the minimum miner size for which mining optimally nets at least ε more than the fair share. The 0-



(a) Based on Bitcoin between Oct. 1, 2020 and Sep. 30, 2021

(b) Based on Ethereum Sep. 1–30, 2021

Fig. 12: Parameters of the model applicable in practice.

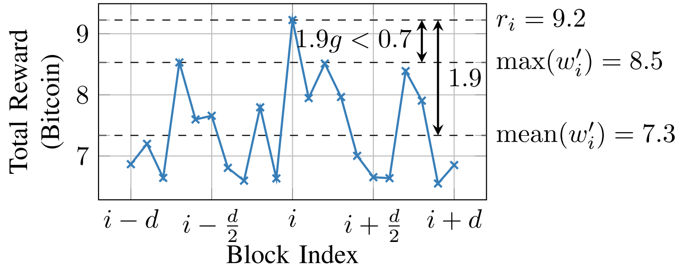


Fig. 13: Example of a window with radius $d = 10$ around a Bitcoin block ($i = 655790$) with an exceptional reward. The ratio between the spike reward minus the second-highest value in the window and the spike reward minus the mean reward is at least g.

gap threshold is identical to the original security threshold. Note that this definition is not just a technical aid for our evaluation method, but a necessary modification to make the threshold meaningful in the varying model. As Fig. 10b shows, increasing δ from 0 by an infinitesimal change, which leads to an infinitesimal increase in average revenue, can completely determine the threshold even for negligible ε. In such a case, the threshold loses any physical meaning. The ε-gap security threshold guarantees that we only consider *material* gains.

When using WeRLman, we make sure the confidence interval is smaller than ε. We calculate the ε-gap security thresholds by taking the minimum threshold of WeRLman and the `simplified` model for different gap sizes (Fig. 11). Thresholds obtained using WeRLman are circled in red. When the gap size is bigger, WeRLman finds a lower threshold than the `simplified` model more often because when considering larger gaps, WeRLman encounters less false negatives.

C. Real-World Parameters

To understand the implications of our results, we choose parameters based on an established operational system, namely Bitcoin. We first extract the total block rewards obtained between October 1, 2020, and September 30, 2021 [27]. We then calculate suitable values of the whale fee F and the transaction chance δ.

Scenario	Chance $\delta = 10^{-4}$		Chance $\delta = 10^{-3}$	
	Fee F	Threshold Upper Bound	Fee F	Threshold Upper Bound
Current	0.29	0.230	0.14	0.250
1 Halving	0.52	0.212	0.26	0.230
2 Halvings	0.89	0.195	0.45	0.230
3 Halvings	1.37	0.168	0.74	0.195
4 Halvings	1.87	0.168	1.14	0.181
5 Halvings	2.59	0.122	1.58	0.168
8 Halvings	4.91	0.078	3.20	0.122
Ethereum Stats	3.93	0.122	1.34	0.168

Table 1: Possible instantiations of the model and the resultant bounds for the ε-gap security thresholds ($\varepsilon = 10^{-6}$).

Since fees gradually vary (Fig.1), we consider whale transactions to be local spikes rather than global maxima. We search for local spikes in block rewards and take the ratio of the spike reward and the mean reward of its neighborhood. Thus, we iterate over all blocks and check two conditions for each block. The first condition asserts that the current block is a local maximum, and the second condition asserts that the block is a spike – larger than all nearby blocks by at least some factor. An example of such a block is illustrated in Fig. 13. The block in position i is the maximum, and the difference of its reward and the second-highest reward is large enough.

Formally, let the total block reward in each block be $r_1, \ldots r_n$. Define a *window radius*, d and a *margin threshold*, g. Then, consider all windows w_i of size $2d+1$ such that w_i is centered around r_i, i.e. $w_i = \{r_j\}_{j=i-d}^{i+d}$. Let w_i' be $w_i \setminus \{r_i\}$. We then go over all windows and check whether the block reward r_i is the maximum in w_i and if $r_i - max(w_i') > (r_i - mean(w_i')) \cdot g$. We extract the suitable fee parameter $F_i = r_i / mean(w_i') - 1$ from all applicable windows. Then, we go over all values of F_i and calculate δ_i to be the number of spikes j for which F is at least as big, i.e. $F_j >= F_i$, divided by n.

For Bitcoin, we perform the above process multiple times for different values of the subsidy. To simulate the subsidy after k halving events, we subtract $(1 - 2^{-k}) \cdot 6.25$ (6.25 is the subsidy at the time of writing [27]) from all r_i. We use $d = 10$ and $g = 0.25$ to plot all possible combinations of F and δ in Fig. 12. Each line in Fig. 12a represents the possible combinations of parameters given a certain number of halving events have occurred.

We consider possible choices of model parameters after various numbers of halving events and bound the ε-gap security threshold in each scenario (Table 1). Since the total rewards obtained by miners in the data we gathered was over 13 billion USD, we choose ε to be 10^{-6}.

For each scenario, we first take the appropriate F for $\delta = 10^{-3}$ and $\delta = 10^{-4}$ based on our previous analysis. Because of the large computational power required to run WeRLman, we compute the security threshold only for a set of values for F and using $\delta = 0.1$. Since the thresholds are decreasing in F (Fig. 10a), for each scenario, we take the threshold obtained in the run using the largest F smaller than the F in the scenario. Although the appropriate δ for the scenario is significantly smaller than 0.1, our previous analysis (Fig. 10b) shows it has almost no effect on the security threshold.

Based on the more conservative bound of $\delta = 10^{-3}$, we see that in 10 years (3 halving events from today), the threshold will reduce to 0.195. In about 20 years (5 halvings), it will reduce to 0.168 and in 30 years (8 halvings), to 0.122 (Fig. 2). These lower thresholds are dangerously close to the common sizes of the largest mining pools [3].

Bitcoin transactions only transfer tokens among accounts. In contrast, Ethereum transactions can perform more complex actions defined in smart contracts [5]. This enables a rich and diverse DeFi ecosystem [26], possibly explaining the higher variation of rewards in Ethereum. We perform a similar process based on data from Ethereum in September 2021, using the same parameters. When considering statistics based on Ethereum, we get that the model parameters matching Ethereum reward values are $F = 1.34$ and $\delta = 10^{-3}$. Notice that F is 10 times larger than the parameters based on Bitcoin today. Using these parameters, the resultant threshold is 0.168.

VI. Related Work

Selfish behavior in blockchains was discovered by Eyal and Sirer [8], followed by a stronger attack by Nayak et al. [15] and the optimal one by Sapirshtein et al. [2] that used iterative MDP optimization. Bar-Zur et al. [13] replaced the iterative approach with a single solution of a so-called probabilistic termination MDP, allowing to find the optimal selfish mining behavior in Ethereum. WeRLman addresses models with much larger state spaces, out of the reach of previous accurate solutions. Wang et al. [37], utilized a novel tabular version of Q-learning suitable for the non-linear objective function to recreate the optimal results of the `constant` model. This method still suffers from the drawbacks of the exact methods and cannot be used when the state space is too big.

Carlsten et al. [18] analyze a model where there is no subsidy and rewards are from gradually arriving transactions – a state that they expect Bitcoin to exhibit once its minting rate drops to zero. In their model, each block has enough capacity to include all pending transactions; then the available rewards gradually increase with the arrival of other transactions. They foresee a mining gap forming, as miners would not mine until the available rewards are sufficient. Tsabary and Eyal [19] quantify the formation of this graph even with a constantly available baseline reward. In contrast, we base our model on the current reward distribution (§V-C) where a baseline level of rewards is always available (due to many pending transactions) and sufficient to motivate mining, and where MEV spikes occur infrequently.

The analysis of Carlsten et al. also assumes miners may slightly deviate from the protocol, whereas we focus on the bound where the prescribed strategy is a Nash Equilibrium. Additionally, they evaluate a specific strategy that outperforms selfish mining, whereas WeRLman searches the entire strategy space for an optimal strategy.

Fruitchains [9] provides an algorithm that is resilient to selfish mining up to a threshold of 50%. However, it assumes constant block rewards by spreading fees over multiple blocks.

Other incentive-based attacks consider extended models taking into account network attacks [15], [38], additional network and blockchain parameters [39], and mining coalitions (pools) [16], [40], as well as exogenous attacker incentives [17], [41], [42] leading it to bribe miners to deviate. Those are outside the scope of this work.

The use of RL to approximately solve the MDPs that occur in selfish mining analysis was pioneered in the SquirRL algorithm [10], which is based on the framework of [2]. Using the PTO framework, and the techniques of Section III, WeRLman is significantly more accurate and can tackle more complex protocols, with significantly more actions and states such as the `varying` model.

Deep RL has been applied to various decision-making domains, including resource management [43] and games [22], [24]. By exploiting several characteristics special to blockchains, our method is able to provide results that are, to the best of our knowledge, the most accurate to date.

VII. Conclusion

We analyze the incentive-based security of Nakamoto-based blockchains when considering, for the first time, varying rewards. Our key discovery is that variance in rewards hurts security; The security threshold in the `varying` model is lower than the 0.25 threshold [2] of the `constant` model. With parameters taken from Bitcoin, the security threshold drops to 0.2 in a decade, to 0.17 in two and to 0.12 in 3 decades. If Ethereum's recent fees were manifested in a Nakamoto chain, the security threshold would be only 0.17.

We face two major challenges in this work. First, naively integrating the external arrival process of transactions into the model means that miner behavior affects the transaction arrival rate. We avoid this by sampling arrivals at the creation of blocks extending the longest chain, ensuring a constant sampling rate due to the difficulty adjustment mechanism. And second, extending deep RL to solve blockchain problems with high accuracy. Our novel method significantly outperforms state-of-the-art accuracy in simple problems and allows analyzing a complex model with significantly more states and actions.

The utilization of deep RL in WeRLman provides a foundation for analyzing other systems like Ethereum as well as novel architectures, enabling the design of more secure blockchains. In addition, the high-accuracy solutions that WeRLman demonstrates may inspire its use in other large, stochastic decision-making problems.

Acknowledgments

The research was supported by the Israel Science Foundation (1641/18, 759/19), an IC3 grant, the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate.

REFERENCES

[1] A. A. Jøn, "The whale road: Transitioning from spiritual links, to whaling, to whale watching in aotearoa new zealand," *Australian folklore*, vol. 29, 2014.

[2] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 515–532.

[3] A. E. Gencer, S. Basu, I. Eyal, R. Van Renesse, and E. G. Sirer, "Decentralization in bitcoin and ethereum networks," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 439–457.

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

[5] V. Buterin, "Ethereum white paper," *GitHub repository*, pp. 22–23, 2013.

[6] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox. (2021) Zcash protocol specification. [Online]. Available: https://github.com/zcash/zips/blob/main/protocol/protocol.pdf

[7] coinmarketcap.com. (2022) Cryptocurrency market capitalizations. [Online]. Available: https://coinmarketcap.com/

[8] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," 2013.

[9] R. Pass and E. Shi, "Fruitchains: A fair blockchain," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 315–324.

[10] C. Hou, M. Zhou, Y. Ji, P. Daian, F. Tramèr, G. Fanti, and A. Juels, "Squirrl: Automating attack analysis on blockchain incentive mechanisms with deep reinforcement learning."

[11] M. Davidson, T. Diamond *et al.*, "On the profitability of selfish mining against multiple difficulty adjustment algorithms." *IACR Cryptol. ePrint Arch.*, vol. 2020, p. 94, 2020.

[12] J. Garay, A. Kiayias, and N. Leonardos, "The bitcoin backbone protocol: Analysis and applications," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 281–310.

[13] R. Bar-Zur, I. Eyal, and A. Tamar, "Efficient mdp analysis for selfish-mining in blockchains," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 113–131.

[14] D. P. Bertsekas, *Dynamic programming and optimal control*. Athena scientific Belmont, MA, 1995, vol. 1, no. 2.

[15] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 305–320.

[16] Y. Kwon, D. Kim, Y. Son, E. Vasserman, and Y. Kim, "Be selfish and avoid dilemmas: Fork after withholding (faw) attacks on bitcoin," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 195–209.

[17] M. Mirkin, Y. Ji, J. Pang, A. Klages-Mundt, I. Eyal, and A. Juels, "Bdos: Blockchain denial-of-service," in *Proceedings of the 2020 ACM SIGSAC conference on Computer and Communications Security*, 2020, pp. 601–619.

[18] M. Carlsten, H. Kalodner, S. M. Weinberg, and A. Narayanan, "On the instability of bitcoin without the block reward," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 154–167.

[19] I. Tsabary and I. Eyal, "The gap game," in *Proceedings of the 2018 ACM SIGSAC conference on Computer and Communications Security*, 2018, pp. 713–728.

[20] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, "Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges," *arXiv preprint arXiv:1904.05234*, 2019.

[21] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[22] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[23] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[24] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[25] T. Rocket, "Snowflake to avalanche: A novel metastable consensus protocol family for cryptocurrencies," *Available [online].[Accessed: 4-12-2018]*, 2018.

[26] Y. Chen and C. Bellavitis, "Blockchain disruption and decentralized finance: The rise of decentralized business models," *Journal of Business Venturing Insights*, vol. 13, p. e00151, 2020.

[27] blockchair.com. (2021) Historical blockchain data. [Online]. Available: https://blockchair.com/

[28] M. L. Puterman, *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[29] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "Sok: Research perspectives and challenges for bitcoin and cryptocurrencies," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 104–121.

[30] R. Pass, L. Seeman, and A. Shelat, "Analysis of the blockchain protocol in asynchronous networks," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2017, pp. 643–673.

[31] L. Bottou, "Online algorithms and stochastic approxima-p tions," *Online learning and neural networks*, 1998.

[32] Y. Efroni, G. Dalal, B. Scherrer, and S. Mannor, "Beyond the one-step greedy approach in reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 1387–1396.

[33] M. Kearns, Y. Mansour, and A. Y. Ng, "A sparse sampling algorithm for near-optimal planning in large markov decision processes," *Machine learning*, vol. 49, no. 2, pp. 193–208, 2002.

[34] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European conference on machine learning*. Springer, 2006, pp. 282–293.

[35] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[36] H. P. van Hasselt, A. Guez, M. Hessel, V. Mnih, and D. Silver, "Learning values across many orders of magnitude," *Advances in Neural Information Processing Systems*, vol. 29, pp. 4287–4295, 2016.

[37] T. Wang, S. C. Liew, and S. Zhang, "When blockchain meets ai: Optimal mining strategy achieved by machine learning," *International Journal of Intelligent Systems*, vol. 36, no. 5, pp. 2183–2207, 2021.

[38] A. Gervais, H. Ritzdorf, G. O. Karame, and S. Capkun, "Tampering with the delivery of blocks and transactions in bitcoin," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 692–705.

[39] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[40] I. Eyal, "The miner's dilemma," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 89–103.

[41] P. McCorry, A. Hicks, and S. Meiklejohn, "Smart contracts for bribing miners," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 3–18.

[42] A. Judmayer, N. Stifter, A. Zamyatin, I. Tsabary, I. Eyal, P. Gazi, S. Meiklejohn, and E. Weippl, "Pay to win: Cheap, cross-chain bribing attacks on pow cryptocurrencies," *5th Workshop on Trusted Smart Contracts*, 2021.

[43] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proceedings of the 15th ACM workshop on hot topics in networks*, 2016, pp. 50–56.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

APPENDIX

A. Transaction Arrival Process

Lemma 3. *Let N be a Poisson process with rate δ and X be an independent random variable distributed exponentially with*

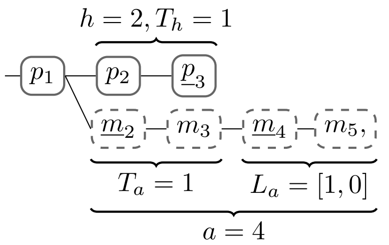


Fig. 14: An example of a state space in the `simplified` model. Blocks denoted with an underscore hold a whale fee.

rate μ. Then, $N(X)$ is a random variable with a geometric distribution with mean $\frac{\delta}{\mu}$.

Proof. By definition, X is distributed by the pdf:

$$p_X(x) = \begin{cases} \mu e^{-\mu x} & x \geq 0 \\ 0 & x < 0 \end{cases} .$$

For any time t, $N(t)$ is distributed by:

$$\Pr\left(N(t) = n\right) = \frac{(\delta t)^n}{n!} e^{-\delta t}, \;\; n = 0, 1, ... \geq 0 \, .$$

Since N and X are independent, for any $n = 0, 1, ...$:

$$\Pr\left(N(X) = n\right) = \int_{-\infty}^{\infty} \Pr\left(N(x) = n\right) p_X(x) dx \, .$$

By plugging the first two equations we get for any $n = 0, 1, ...$:

$$\Pr\left(N(X) = n\right) = \int_{0}^{\infty} \frac{(\delta x)^n}{n!} e^{-\delta x} \cdot \mu e^{-\mu x} dx =$$

$$= \frac{\delta^n \mu}{n!} \int_{0}^{\infty} x^n e^{-(\delta + \mu) x} dx = \frac{\delta^n \mu}{n!} \cdot \frac{n!}{(\delta + \mu)^{n+1}} =$$

$$= \frac{\mu}{\delta + \mu} \cdot \left(\frac{\delta}{\delta + \mu}\right)^n .$$

And this is exactly the probability function of a geometric distribution with mean $\frac{\delta}{\mu}$. □

B. Simplified Varying Model

Another approach to overcome the intractable state space of the `varying` model is to simply reduce it. We introduce a `simplified` model based on the `varying` model which allows the rational miner a subset of the possible actions. This allows the constricted model to be represented more efficiently and greatly decreases the size of the state space. We can then use PTO [13] to solve this model exactly. Because the constricted model only reduces the freedom of the rational miner, optimal policies found in the constricted model give a lower bound to the optimal policy possible in the full model.

The state space of the `simplified` model is represented as a 7-tuple of the form $(a, h, L_a, T_a, T_h, \mathit{fork}, \mathit{pool})$. The elements a and h represent the length of the miner chain and the public chain as in the `constant` model. The elements *fork* and *pool* are the same as in the `varying` model. The transactions in the rational miner's chain are represented in the elements L_a and T_a. The element T_a is the number of whale fees in the first h blocks in the miner's chain. The element L_a is a list of flags, representing whether each of the rest of the blocks in the miner's chain holds a whale fee. Similarly, T_h is the number of whale fees in the public chain. Fig. 14 illustrates a simple example of a state in the `simplified` model.

We calculate the number of states in our configuration by counting the number of appropriate combinations of elements and find it to be about 500,000.

We restrict the possible of actions of the miner to be the same as the `constant` model, adopt, reveal h, reveal $h+1$ and wait. The transitions of a, h are the same as defined in the `constant` model. The transitions of *fork* and *pool* are as defined in the `varying` model. When the miner adopts the public chain, $L_a \leftarrow [\;]$, $T_a \leftarrow 0$ and $T_h \leftarrow 0$. When the miner overrides the public chain by submitting $h+1$ blocks, those block become accepted by all so similarly, $T_a \leftarrow 0$ and $T_h \leftarrow 0$. However, in this case L_a is preserved except for the first block which is removed from it. When an active fork is resolved, that is when there is an active fork in the network and an honest miner extends the rational miner's chain, the first h blocks in the miner's chain are accepted by all. Therefore, $T_h \leftarrow 0$, T_a is reset and the first element of L_a is moved to T_a (if such an element exists). Whenever the rational miner finds a new block, a flag indicating whether it holds a whale transaction gets added to the end of L_a. Whenever another miner finds a new block, the first element of L_a is removed from the list and is added to T_a instead (if such an element exists). In all other cases, L_a, T_a and T_h remain the same. The rewards are as defined in the `varying` model. They can still be easily calculated from the state, even though the full lists $\vec{a}$ and $\vec{h}$ are unavailable.

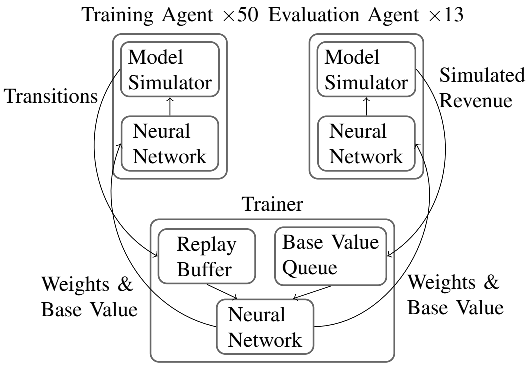


Fig. 15: The major parts of the framework and their data flow.

C. Implementation

We implemented the algorithm as a python application (about 7k LoC) designed to run on 64 CPU cores in parallel using PyTorch. Fig. 15 highlights the main elements of the application and their interactions. The application consists of 63 agents: 50 agents explore the model to gather state-transition data, and 13 agents simulate the current policy to approximate its revenue. The agents operate in a continuous loop of episodes. Once an episode ends, the gathered data are sent to the trainer and the neural network and base value are updated. The trainer also runs in a continuous loop, where in each iteration, we run an *epoch* of 50 batches of SGD. Before each epoch, the trainer gathers all the transitions and the simulated revenues, and then the neural network and the base value are updated.

The first epoch relies on the output of the first episodes. Thus, in order to allow the trainer to run in parallel with the agents, the first epoch has to run while the second episodes are running. Therefore, there is no need to update the neural network and base value after the first episode. Afterwards, the epochs and episodes are synchronized, meaning that in order to continue to the next epoch and episodes, the current epoch and all current episodes must end first. Except for the first episode, the neural network and base value are updated at the end of each episode to be the output of the epoch which was synchronized with the previous episode. For example, after the third episode, the agent is updated to the neural network reached after the first epoch.

1) Model: The algorithm uses the transformed model obtained by PTO with an expected horizon of 10,000. We chose a maximum fork and maximum pool size of 10 and $\gamma = 0.5$, respectively, except where it is specified otherwise.

2) Neural Network: The actual form of the state that is fed into the neural network has to be of a fixed size. Furthermore, a sparse representation with distinct differences between types of data is usually preferred as input for neural networks. To achieve this, we represent the lists $\vec{a}$ and $\vec{h}$ as two vectors with a length of the maximum possible fork and each element as two binary flags. The first flag indicates whether a block exists or not, and the second flag expresses whether a transaction is included in the respective block (if the block exists, otherwise this is meaningless). For example, to indicate that the public chain has two blocks and the second block holds a transaction, $\vec{h}$ would be $[(1, 0), (1, 1), (0, 0), (0, 0), ...]$. In addition, the state representation also includes the number of blocks and number of transactions in both chains, which can be extracted from $\vec{a}$ and $\vec{h}$,

The neural network is structured with two hidden layers, each with 256 neurons and a ReLU activation function. The dimension of the input is determined by the length of the state tuple. In our case, it is 46: both $\vec{a}$ and $\vec{h}$ are represented as a tuple of length 20 (twice the maximum fork), and there are 6 additional elements: *fork*,*pool*, the number of blocks and the number of transactions both in $\vec{a}$ and $\vec{h}$. The dimension of the output is twice the number of available actions. The first half of the output is interpreted as the values of all actions, and the second as the prior probability of the actions.

3) Agents: Each agent simulates an episode of 100 steps. In each step, the agents performs a Monte Carlo tree search to choose its next action. The agent simulates 25 potential trajectories separately. Each trajectory is of 5 steps ahead. To encourage the search to try all possible actions, in the first step of every trajectory, $0.5(P(s, a) + |\mathcal{A}|^{-1})$ is used instead of $P(s, a)$ in the term for the exploration factor (Eq. 8).

Similarly to AlphaGo Zero [23] we create a graph to save all the paths of all simulated trajectories. We initialize $W(s, a)$ using the appropriately scaled value of the neural network (Eq. 12). We update these values according to Eq. 10 with $\beta = 1/N(s, a)$. The graph is saved across the trajectories and steps and is only reset between episodes.

There are two types of agents, the training agents and the evaluation agents. After running the MCTS, a training agent chooses the action to take according to an ε-greedy policy (with $\varepsilon = 0.05$): it chooses the action with the maximum value estimate w.p. $1 - \varepsilon$ and w.p. ε chooses some valid action uniformly at random. Then, it saves a record of the state, the chosen action, the target value, and the empirical distribution of the actions selected in the state. The target value is calculated by Eq. 13. The evaluation agent always chooses the action greedily, and only at the end of an episode checks the revenue obtained and sends it to the trainer.

4) Trainer: In every epoch, the trainer gathers the transition records and target values from all the training agents into a replay buffer of 5000 transitions. The replay buffer is then shuffled, and the trainer samples 50 batches of 100 transitions each to train the neural network. The target values are first normalized by Eq. 14. The target values mean $\bar{t}$ is updated using an exponential moving average $\bar{t}_{\text{new}} = 0.9\bar{t}_{\text{old}} + 0.1 \cdot \frac{1}{|B|} \sum_{n \in B} t_n$, and is used instead of the calculated mean in Eq. 14.

The neural network is trained according to the loss function in Eq. 9, using the Adam Optimizer [44] with default parameters (in PyTorch), and a learning rate that starts at 0.0002, and is divided by 10 after every 1000 epochs.

In addition, the trainer gathers all revenue simulations from the evaluation agents. The base value is calculated by a sliding window average of the last 750 episodes. For the first epochs, the window is first initialized to be some predetermined initial parameter, repeated 750 times for reducing the dependence of the base value on the first few simulations. We chose the initial value to be the optimal revenue of the `constant` model scaled by $(1 + \delta F)$ as the default value.

5) After Training: Every 100 epochs, the state of the neural network and the base value are saved, and all 63 agents perform a revenue simulation (like the evaluation agents) in an episode of 1000 steps. To avoid high memory consumption, we prune the MCTS graph in the agents every 250 steps by removing all nodes which were not used more than once. The final policy of the algorithm is determined by the neural network and base value of the epoch with the best average revenue from the longer simulations. Then, to more accurately approximate the final policy's revenue, all agents run another simulation with the chosen neural network and base value in episodes of 100,000 steps. The final output is the average revenue across all agents and a confidence interval of 99% confidence.

To obtain reproducible results, we initialized the seed of all sources of all the sources of randomness in the algorithm to be

0 in all runs. All parallel results which were aggregated were sorted by their agent index before using them, to maintain reproducibility. Because of hardware dependent optimizations built into the linear algebra libraries used in NumPy and PyTorch, the results may vary slightly in different CPUs. We ran all of our tests using an AMD EPYC 7742 256-core processor.

D. Proof of Prop. 1

We restate and prove Prop. 1.

Proposition 4. *Let $Q^*(s,a)$ be the optimal action values in an MDP obtained by a PTO transformation with expected horizon H. Let π^* be the optimal policy and s_{init} be the initial state of the MDP. Then, it holds that for any state $s \in \mathcal{S}$:*

$$|Q^*(s,\pi^*(s)) - Q^*(s_{init},\pi^*(s_{init}))| = O\left(\frac{1}{H}\right). \tag{15}$$

Proof. Let π be a policy and let ρ_π be the revenue of π starting from state s and let ρ'_π be the revenue of π in the transformed MDP with an expected horizon of H starting from the same state. As a result of Theorem 2 of Bar-Zur et al. [13], we get that $|\rho_\pi - \rho'_\pi(H)| = O\left(1/H\right)$. Note that since $\rho'_\pi(H) = Q^*(s,\pi^*(s))$, we get that $|\rho_\pi - Q^*(s,\pi^*(s))| = O\left(1/H\right)$. Furthermore, the revenue ρ_π is invariant to the initial state chosen (Lemma 8 in [13]). Therefore, when using $s = s_{\text{init}}$, we get that $|\rho_\pi - Q^*(s_{\text{init}},\pi^*(s_{\text{init}}))| = O\left(1/H\right)$. Thus, we have that $|Q^*(s,\pi^*(s)) - Q^*(s_{\text{init}},\pi^*(s_{\text{init}}))| = O\left(1/H\right)$, by using the triangle inequality. □

E. Value Iteration with Base Value

The effects of target normalization are clearly understood from Proposition 1. The average of the true values should be the base value (up to $O\left(\frac{1}{H}\right)$) and thus normalizing the target values such that their new average is the base value makes sense. However, the effects of adding the base value and changing it from time to time are less clear.

We design a simplified algorithm which utilizes the base value feature in value iteration and assumes access to an oracle which given a policy outputs its exact revenue (Algorithm 1). The algorithm is based on value iteration, the synchronous version of Q-learning, and it can be extended to Q-learning as well. To avoid oscillations where the base value may go up or down, we enforce the base values to be increasing by taking the maximum between the old value and the new candidate base value returned by the oracle. We show that the algorithm is guaranteed to converge, just like value iteration.

Theorem 5. *The values calculated in Algorithm 1 converge to the optimal values and for a large enough N, the algorithm returns the optimal policy.*

$$\lim_{N\to\infty} V_N = V^* - \rho_N, \lim_{N\to\infty} \pi_N = \pi^*$$

Proof. The base value is bounded from above by the base value of the optimal policy, since it achieves the highest revenue. Because the base values ρ_n are a monotonically increasing sequence and bounded from above, we get that they converge to some ρ. Since, the number of possible candidates is limited by the number of policies, and the number of policies is finite (thanks to the practical considerations which ensure a finite state space size), there must be a step $M \in \mathbb{N}$ such that for all $n \geq M$, $\rho_n = \rho$. Then, for all $n > M$:

$$V_n(s) \leftarrow \max_{a\in\mathcal{A}}\{r(s,a) - \rho + \\ + \sum_{j\in\mathcal{S}\setminus\{s_{\text{term}}\}} p(s'|s,a)(V_{n-1}(s') + \rho)\}.$$

Since after every iteration, we subtract all values by ρ but add ρ before using the values again, in the subsequent iteration, the sequence behaves similarly to value iteration and converges. The only difference is that all values in the limit are shifted by $-\rho$ which is equal to ρ_N for a large enough N. This proves the first half of the theorem.

For $n \geq M$, the values are combined with ρ before choosing a policy, thus the choice of policy is similar to the policy induced from regular value iteration. Therefore, the chosen policy for a large enough N is the optimal policy π^*. □

Algorithm 1: Value Iteration With Base Value

```
Input : Transition matrices $P(s'|s,a)$
        Reward matrix $R(s,a)$
        Initial base value $\rho_0 \in \mathbb{R}$
        Terminal state $s_{\text{term}}$
        Initial value function $V_0$
        Base value oracle $\mathcal{O} : \mathcal{A}^{\mathcal{S}} \to \mathbb{R}$
        Number of iterations $N$
Output: Policy $\pi_N$
for $n \leftarrow 1$ to $N$ do
    For each $s \in \mathcal{S} \setminus \{s_{\text{term}}\}$, set:
        $\pi_n(s) \leftarrow \arg\max_{a\in\mathcal{A}}\{r(s,a) +$
            $+ \sum_{j\in\mathcal{S}\setminus\{s_{\text{term}}\}} p(s'|s,a)(V_{n-1}(s') + \rho_{n-1})\}$.
    For each $s \in \mathcal{S} \setminus \{s_{\text{term}}\}$, set:
        $V_n(s) \leftarrow \max_{a\in\mathcal{A}}\{r(s,a) - \rho_{n-1} +$
            $+ \sum_{j\in\mathcal{S}\setminus\{s_{\text{term}}\}} p(s'|s,a)(V_{n-1}(s') + \rho_{n-1})\}$.
    $\rho_n \leftarrow \max\{\mathcal{O}(\pi_n), \rho_{n-1}\}$
end
```

F. Relating Revenues between Models

Since the fees in the `varying` model introduce a new source of rewards, the total reward amount in the model increases. But, to be able to better interpret the revenue of a miner in the `varying` model, we need to relate it to the revenue of a policy in the `constant` model. First, given a policy in the `constant` model, we would want to know what revenue it will achieve in the model with fees when extended appropriately. We will assume that under the policy, the miner would behave the same and will be oblivious to fees. The only difference is that whenever a transaction is available, the rational miner will include it in the next block created. The following proposition is a restatement of Prop. 2.

Proposition 6. *Let π be policy in the* `constant` *model and let π' be its fee-oblivious extension to the* `varying` *model, in which the miner always includes a transaction when available. If ρ_{π}, and $\rho_{\pi'}$ are the revenues of π and π' in the respective models, then $\rho_{\pi'} = (1 + \delta F)\rho_{\pi}$.*

Proof. Since π' is oblivious to fees and just adds a transaction when possible, the choice of which block transactions appear in is independent of which block belongs to the rational miner. This means that on average, the chance that a block of the rational miner includes a transaction is the same as the chance any block includes a transaction and that is precisely δ. Therefore, blocks of the rational miner are worth $1 + \delta F$ in expectation, and thus the miner's revenue is the ratio of blocks which belong to her times the value of each block. Thus, $\rho_{\pi'} = (1 + \delta F)\rho_{\pi}$. □

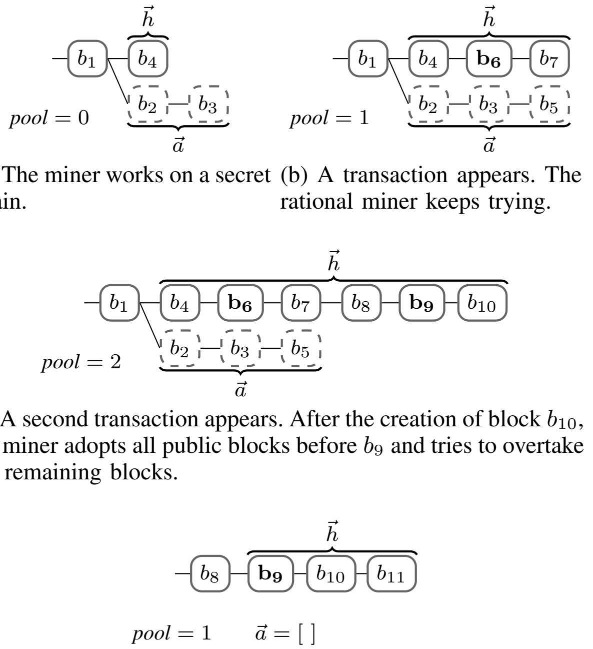
We can then immediately deduce the following corollary.

Corollary 7. *The fair share of an honest miner of size α in the* `varying` *model is $(1 + \delta F)\alpha$*

Proof. Follows immediately from Prop. 6 and the fact that the revenue of an honest miner in the `constant` model is simply α. □

G. Optimal Strategy Case Study

The rational miner can take advantage of the full `varying` model to obtain more revenue than in the `constant` and `simplified` model. To show this, we illustrate two cases where the learned strategy utilizes an action available only in the richer `varying` model. Both cases arose in our simulation of the final learned strategy with $\alpha = 0.35$, $F = 10$, $\delta = 0.1$. In the two cases, the miner adopts all blocks but the last one or two blocks in the public chain and tries to steal a transaction fee. Blocks are named according to the order they were mined. Blocks holding a transaction are marked in bold. The first case (happens around step 45507 in test agent 4) shows an unsuccessful attempt (Fig. 16) and the second case (happens around step 72010 in test agent 3) shows a successful one (Fig. 17).



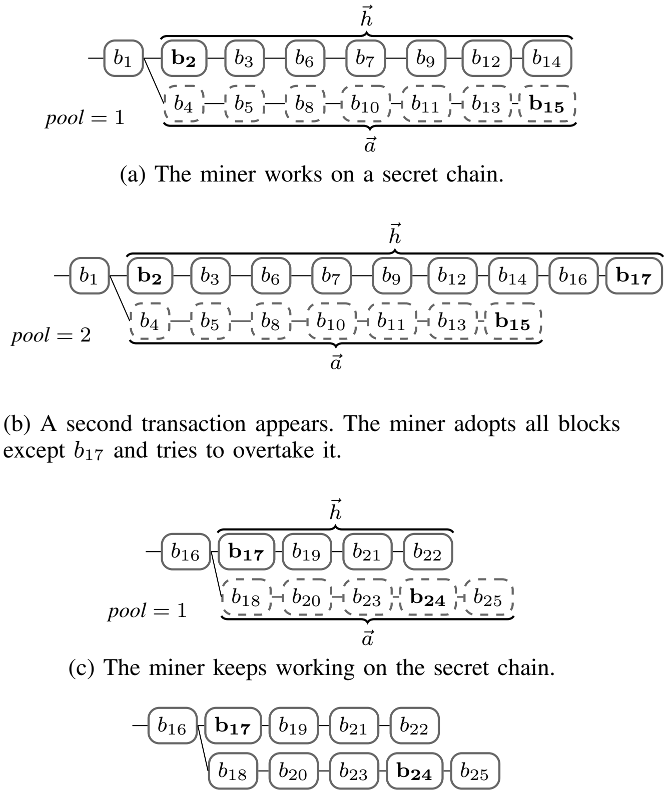
(a) The miner works on a secret chain.

(b) A transaction appears. The rational miner keeps trying.

(c) A second transaction appears. After the creation of block b_{10}, the miner adopts all public blocks before b_9 and tries to overtake the remaining blocks.

(d) The miner adopts the public chain after the creation of block b_{11}.

Fig. 16: An unsuccessful attempt of a partial adoption to steal a transaction fee.


$\vec{h}$
b_1 $\mathbf{b_2}$ b_3 b_6 b_7 b_9 b_{12} b_{14}
b_4 b_5 b_8 b_{10} b_{11} b_{13} $\mathbf{b_{15}}$
$pool = 1$
$\vec{a}$
$\vec{h}$
b_1 $\mathbf{b_2}$ b_3 b_6 b_7 b_9 b_{12} b_{14} b_{16} $\mathbf{b_{17}}$
b_4 b_5 b_8 b_{10} b_{11} b_{13} $\mathbf{b_{15}}$
$pool = 2$
$\vec{a}$
$\vec{h}$
b_{16} $\mathbf{b_{17}}$ b_{19} b_{21} b_{22}
b_{18} b_{20} b_{23} $\mathbf{b_{24}}$ b_{25}
$pool = 1$
$\vec{a}$
b_{16} $\mathbf{b_{17}}$ b_{19} b_{21} b_{22}
b_{18} b_{20} b_{23} $\mathbf{b_{24}}$ b_{25}

(a) The miner works on a secret chain.

(b) A second transaction appears. The miner adopts all blocks except b_{17} and tries to overtake it.

(c) The miner keeps working on the secret chain.

(d) The miner overtakes the chain and the state resets.

Fig. 17: A successful attempt of a partial adoption to steal a transaction fee.