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 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-
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, 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](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 \cite{14}, \cite{28}. It consists of an environment and an agent. Let \( S \) be the state space and \( A \) be the action space. At every step \( t \), the agent observes the state \( s_t \in S \) of the environment and chooses an action \( a_t \in A \). 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 \( \pi : S \rightarrow A \) defines which action to take in each state, independent of previous states or actions. Standard objective functions include the \( \lambda \)-discounted reward \( \mathbb{E}\left[\sum_{t=1}^{\infty} \lambda^{t-1} R_t\right] \), 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 \cite{14}.

B. Preliminaries: Selfish Mining

We focus on blockchains based on the original design of Bitcoin \cite{4}, which we refer to as Nakamoto blockchains (after their pseudonymous creator). The main goal of a blockchain \cite{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.

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 \cite{9}, \cite{12}, \cite{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 \cite{2}, \cite{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 miners on a single secret chain, and all other miners mine honestly on a single public chain. Let \( \alpha \) be the fraction of the miner from the total computational power of the system. Let \( \gamma \) 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 \( \gamma \) 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 \( \text{pool} \)) into the state space. All new whale transactions arrive into the pool (by incrementing the value of \( \text{pool} \)), 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 \( \sum_{i=1}^{n} D_t \) 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 \( \pi \) be
\[
E[\lim_{n \to \infty} \frac{\sum_{t=1}^{n} R_t}{\sum_{i=1}^{n} D_t}]\]
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 \( (\tilde{a}, \tilde{h}, \text{fork}, \text{pool}) \). The vector \( \tilde{a} \) 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 \( \tilde{h} \) keeps the blocks mined in the public chain. The element \( \text{fork} \in \{\text{RELEVANT}, \text{IRRELEVANT}, \text{ACTIVE}\} \) is a ternary value denoting whether the last block was mined by the rational miner, another miner or whether a race is taking place, respectively. The element \( \text{pool} \) is the number of available transactions since the last fork.

Note the differences between varying model and the constant model of Sapirshtein et al. [2]. While \( \text{fork} \) is the same in both models, the element \( \text{pool} \) is new in the varying model. In addition, the vectors \( \tilde{a} \) and \( \tilde{h} \) 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 \( \tilde{a} \) requires at least \( 2^{\lceil |\tilde{a}| \rceil} \) bits. This is an exponential cost with respect to the allowed length of the miner’s chain. The same applies for \( \tilde{h} \). 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 \( \text{fork} \). 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 \( \text{fork} = \text{IRRELEVANT} \). In addition, in this case, \( |\tilde{a}| = |\tilde{h}| = 2 \). Second, in Fig. 4b \( |\tilde{a}| = 2, |\tilde{h}| = 3 \) and since \( p_4 \) was the last mined block then \( \text{fork} = \text{RELEVANT} \). Lastly, in Fig. 4c the miner published her private chain after block \( p_3 \) was mined. In this case, \( |\tilde{a}| = |\tilde{h}| = 2 \) and \( \text{fork} = \text{ACTIVE} \).

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 \( \tilde{a} \) and \( \tilde{h} \) are described with the actions, as these are the main changes in the state. Then, the transitions of \( \text{fork} \) are described separately, as they do not depend on \( \tilde{a} \) and \( \tilde{h} \).

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

In the constant model, allowing the miner to adopt exactly \( \ell \) 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 \( \ell \): This action is legal only when the miner’s private chain is at least as long as \( \ell \). The miner reveals the first \( \ell \) blocks of her private chain, either causing an active fork.
when $\hat{h} = \ell$ and $\text{fork} = \text{RELEVANT}$ or overriding the public chain when $\ell > |\hat{h}|$. 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 $\ell$ plus the number of fees in the first $\ell$ blocks. In addition, (1) $\vec{a} \ll \ell$, (2) $\hat{h} \leftarrow \{\}$, and (3) $D_t \leftarrow \ell$.

In the constant model, the miner is only allowed to reveal either $\hat{a}$ blocks to cause an active fork or exactly $\hat{h} + 1$ 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 $\hat{h} + 1$ blocks. The possibility to adopt or reveal an arbitrary number of blocks is the reason that the state must save $\vec{a}$ and $\hat{h}$ 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 $\hat{h}$ 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., $\text{fork} \neq \text{ACTIVE}$, the block is mined by the rational miner w.p. $\alpha$ or is mined by someone else w.p. $1 - \alpha$. When there is an active fork in the network, namely when $\text{fork} = \text{ACTIVE}$, for example, the case in Fig. 4c, a fraction $\gamma$ 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. $\alpha$ 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) $\hat{a} \ll |\hat{h}|$, and (2) $\hat{h} \leftarrow \{\}$. 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 $\gamma$ 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 $\delta$, 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 $\delta$ (See Appendix A). In practice, however, since we only consider small $\delta$ values, we simulate this as only a Bernoulli experiment with a transaction arrival chance of $\delta$ and neglect terms of order $\delta^2$ 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 $\max\{|\vec{a}|, |\hat{h}|\} < \text{max~fork}$. Practically, all actions that would lead to either $\vec{a}$ or $\hat{h}$ 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:

\[
|S| = (2^0 + 2^1 + \ldots + 2^{max_{fork}})^2 \cdot 3 \cdot (max_{pool} + 1) \approx \approx 12 \cdot max_{pool} \cdot 4^{max_{fork}}. \tag{1}
\]

**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 \([3]\).

**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 (\( \text{III-A} \)). We proceed to describe Q-learning (\( \text{III-B} \)) and Monte Carlo Tree Search (MCTS) (\( \text{III-C} \)), 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 (\( \text{III-D} \)).

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 \( \text{II} \), 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 \([23]\). 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:

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

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

\[
\pi^*(s, a) = \arg \max_{a \in A} \{ Q^*(s, a) \} \tag{3}
\]

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 $\beta_n$. Formally, for $n = 1, 2, ...,$, it performs:

$$Q_n(s_n, a_n) \leftarrow Q_{n-1}(s_n, a_n) + \beta_n \left( R(s_n, a_n) + \mathbb{E}_{s' \sim p(s|s_n, a_n)} \left[ \lambda \max_{a' \in A} Q_{n-1}(s', a') \right] - Q_{n-1}(s_n, a_n) \right).$$

(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 $\beta_n$ [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 used to perform the Q-learning targets in Eq. 5, which guides the action sampling during the trajectory simulation. In each of the simulated trajectory steps, an action $a$ is chosen, which is then performed, and the rewards $r$ are collected. The return $r$ is calculated for each batch by minimizing the mean squared error loss, using stochastic gradient descent, typically in batches [31]. The loss function is then calculated for each batch $B \in \{1, 2, ..., n\}$: $L(\theta) = \frac{1}{|B|} \sum_{n \in B} (r_n - Q_\theta(s_n, a_n))^2$, and used to update $\theta$ 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 3 and 4 are replaced with a sample from the simulator $s' \sim p(s'|s_n, a_n)$, 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 $t_n = r_n + \lambda \max_{a' \in A} Q_\theta(s'_n, a')$.

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:

$$t_n = \max_{\pi_0, ..., \pi_h} \mathbb{E}_{\pi_0 \cdots \pi_h} \left[ \sum_{t=1}^{h-1} \lambda^{t-1} R(s_n^{(t)}, \pi_t(s_n^{(t)})) + \lambda^h Q(s_n^{(h)}, \pi_h(s_n^{(h)})) \right].$$

(6)

where $s_n^{(0)}$, ..., $s_n^{(t)}$ denote a possible trajectory of states and the notation $\mathbb{E}_{\pi_0 \cdots \pi_h}$ 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 $Q_\theta$ 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 $O(|A|^h)$ [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], [25]. 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 $s^{(0)}, ..., s^{(h)}$ with actions $a^{(0)}, ..., a^{(h-1)}$ and rewards $r^{(0)}, ..., r^{(h-1)}$ is simulated, the algorithm performs a backup to update the values $W(s, a)$:

$$W(s^{(t)}, a^{(t)}) \leftarrow W(s^{(t)}, a^{(t)}) + \beta (r^{(t)} + W(s^{(t+1)}, a^{(t+1)}) - W(s^{(t)}, a^{(t)})).$$

(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:

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

(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 \( \phi \). 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:

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

\[
W(s_t, a_t) \leftarrow W(s_t, a_t) + \beta \left( R(s_t, a_t) + \sum_{a' \sim P(\cdot|s_t, a_t)} \max_{a'' \in A} W(s', a') - W(s_t, a_t) \right).
\] (10)

**D. Neural Network Scaling**

The quality of the RL solution depends on the accuracy of the value function approximation \( Q^\theta \). 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 \( \pi^* \) be the optimal policy and \( s_{init} \) be the initial state of the MDP. Then, it holds that for any state \( s \in S \):

\[
|Q^*(s, \pi^*(s)) - Q^*(s_{init}, \pi^*(s_{init}))| = O \left( \frac{1}{H} \right).
\] (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 \( \rho \). 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:

\[
W(s, a) = \rho + \frac{Q(s, a)}{H},
\] (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:

\[
t_n = H \cdot W(s, a) - \rho.
\] (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:

\[
t'_n = t_n - \frac{1}{|B|} \sum_{n \in B} t_n.
\] (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
steps 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 B). 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. 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 [3]. 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.

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 3. This constraint limits the state space to approximately 2,000 states (Eq. 1).

\[\text{Revenue} = \alpha \cdot \text{Steps} - \beta \cdot \text{Epochs} + \gamma \cdot \text{Revenues}\]

\[\text{Steps} = 10^6 \cdot \text{Epochs} + 1\]

\[\text{Epochs} = \text{Steps} / 10^6\]

\[\text{Revenues} = \text{Optimal}\]

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.

Fig. 7: The revenue of WeRLman compared to simple MCTS using the constant model.
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 $\pi$ be policy in the constant model and let $\pi'$ be its fee-oblivious extension to the varying model, in which the miner always includes a transaction when available. If $\rho_0$ and $\rho_1$ are the revenues of $\pi$ and $\pi'$ in the respective models, then $\rho_1 = (1 + \delta F)\rho_0$.

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 $\alpha$ (Fig. 9a and 9b), the whale fee amount $F$ (Fig. 9c) and the whale fee frequency $\delta$ (Fig. 9d).
Varying revenue in the constant model surpasses the line for honest mining. In contrast, this deviation occurs at around 0.1 for varying models. We compare the security thresholds of the constant model and test their sensitivity with respect to the fee size $F$ (Fig. 10a), the transaction arrival chance $\delta$ (Fig. 10b), and the rushing factor $\gamma$ (Fig. 10c).

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.

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.

**B. Security Threshold**

rushing factor

(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 $\delta$ with $\gamma = 0.5$ and $F = 2.5$.

(c) The security threshold as a function of the rushing factor $\gamma$ 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.

Fig. 11: Security thresholds in the varying model for $\gamma = 0.5$, $F = 2.5$ for various values of $\delta$ when considering different gaps. Points marked in red were obtained using WeRLman.
The threshold loses any physical meaning. The infinitesimal increase in average revenue, can completely change the threshold meaningful in the varying model. As Fig. 10b shows, the 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 $\delta$ from 0 by an infinitesimal change, which leads to an infinitesimal increase in average revenue, can completely determine the threshold even for negligible $\epsilon$. In such a case, the threshold loses any physical meaning. The $\varepsilon$-gap security threshold guarantees that we only consider material gains.

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.

When using WeRLman, we make sure the confidence interval is smaller than $\varepsilon$. We calculate the $\varepsilon$-gap security thresholds by taking the minimum threshold of WeRLman and the simplified model for different gap sizes (Fig. 10b). 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 $\delta$.

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 $r_1, ... r_n$. 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 $r_i - \max(w'_i) > (r_i - \text{mean}(w'_i)) \cdot g$. We extract the suitable fee parameter $F_i = \frac{r_i}{\text{mean}(w'_i)} - 1$ from all applicable windows. Then, we go over all values of $F_i$ and calculate $\delta_i$ 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 $\delta$ 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 $\varepsilon$-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 $\varepsilon$ 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 $\delta$ for the scenario is

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

```plaintext
<table>
<thead>
<tr>
<th>Scenario</th>
<th>Chance $\delta = 10^{-4}$</th>
<th>Chance $\delta = 10^{-3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fee $F$</td>
<td>Threshold</td>
</tr>
<tr>
<td>Current</td>
<td>0.29</td>
<td>0.230</td>
</tr>
<tr>
<td>1 Halving</td>
<td>0.52</td>
<td>0.212</td>
</tr>
<tr>
<td>2 Halvings</td>
<td>0.89</td>
<td>0.195</td>
</tr>
<tr>
<td>3 Halvings</td>
<td>1.37</td>
<td>0.168</td>
</tr>
<tr>
<td>4 Halvings</td>
<td>1.87</td>
<td>0.168</td>
</tr>
<tr>
<td>5 Halvings</td>
<td>2.59</td>
<td>0.122</td>
</tr>
<tr>
<td>8 Halvings</td>
<td>4.91</td>
<td>0.078</td>
</tr>
<tr>
<td>Ethereum Stats</td>
<td>3.93</td>
<td>0.122</td>
</tr>
</tbody>
</table>
```
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 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.


rate $\mu$. 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:

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

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

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

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

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

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

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

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 $(a, h, L_a, T_a, T_h, \text{fork, pool})$. 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 $T_h$ 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 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, $L_a \leftarrow \emptyset$, $T_a \leftarrow 0$ 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 $a$ and $h$ are unavailable.

**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 agent 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, $0.5(P(s, a) + |A|^{-1})$ is used instead of $P(s, a)$ in the term for the exploration factor (Eq. 3).

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 $\varepsilon$-greedy policy (with $\varepsilon = 0.05$): it chooses the action with the maximum value estimate w.p. $1 - \varepsilon$ and w.p. $\varepsilon$ 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 $t$ is updated using an exponential moving average $t_{new} = 0.9t_{old} + 0.1 \cdot \frac{1}{|B|} \sum_{n \in B} t_{n}$, 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
Using the triangle inequality.

To the optimal values and for a large enough $N$ the values calculated in Algorithm 1 converge to some $\rho$. 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 $n > M$:

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

Since after every iteration, we subtract all values by $\rho$ but add $\rho$ 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 $\rho_N$ for a large enough $N$. This proves the first half of the theorem.

For $n \geq M$, the values are combined with $\rho$ 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 $\pi^*$.

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

**Proof.** Since \( \pi' \) 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 \( \delta \). 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 \( \alpha \) 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 \( \alpha \).

### 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).

---

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

(a) The miner works on a secret chain.

(b) A second transaction appears. The miner adopts all blocks before \( b_{10} \) and tries to overtake the remaining blocks.

(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 blocks.

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

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

(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 overtake the chain and the state resets.