RESEARCH BULLETIN MRL-0006

Difficulty Adjustment Algorithms in Cryptocurrency Protocols

12 October 2014

Surae Noether* and Sarang Noether

*Correspondence: lab@monero.cc
Monero Research Lab

Abstract

As of this writing, the algorithm employed for difficulty adjustment in the CryptoNote reference code is known by the Monero Research Lab to be flawed. We describe and illustrate the nature of the flaw and recommend a solution. By dishonestly reporting timestamps, attackers can gain disproportionate control over network difficulty. We verify this route of attack by auditing the CryptoNote reference difficulty adjustment code, which, we reimplement in the Python programming language. We use a stochastic model of blockchain growth to test the CryptoNote reference difficulty formula against the more traditional Bitcoin difficulty formula. This allows us to test our difficulty formula against various hash rate scenarios.

This research bulletin has not undergone peer review, and reflects only the results of internal investigation.

1 Introduction

For blockchain-based currencies such as Monero and Bitcoin, both transaction verifications and mining rewards occur upon block arrivals. The rate of block arrival is tied to both network hash rate and the block difficulty score. A good block difficulty adjustment method tracks network hash rate such that block arrival rate is kept on target; this ensures that currency rewards for mining are paid on schedule and that transaction verification does not stall.

The difficulty score of the next block to be added to the blockchain depends on the sequence of timestamps of the blocks preceding it, together with those difficulty scores. We have no way of validating a timestamp, and so timestamps are vulnerable to manipulation; the sequence of timestamps need not even be ordered. Hence, mining reward scheduling and transaction verification are vulnerable to manipulation by way of difficulty manipulation.

To our knowledge, at the time of this publication, the study by Kraft in [4] is the first and only analysis of blockchain difficulty adjustment. In part, [4] formally establishes the relationship between nonhomogeneous Poisson processes, block arrival rate, and network hash rate; we recapitulate some of those arguments less formally here in Section **??**. Kraft also develops a model of block arrival as a function of exponential hash rate and derives the desired constraint to place upon the model

to ensure block arrival times stably approach the target. Kraft refers to the desired constraint as the *time-ratio update*, and develops a fixed point iteration based on time-ratio updating as a new difficulty adjustment method. Under an exponential network hash rate with a constant growth, Kraft showed that the proposed fixed point iteration yields block arrival time that exponentially approach the prescribed target. Although Kraft assumed that network hash rate is an exponential function of time, this result and many of the other results from [4] may be naturally extended to piecewise continuous exponential functions (with either positive or negative growth rate).

In this research bulletin, we audit the difficulty assessment and adjustment components of the CryptoNote reference code, and we compare this code with the Bitcoin code; we conclude that the CryptoNote reference code is an inadequate solution to difficulty assessment and adjustment. We informally derive a stochastic process model of block arrival as a function of a piecewise constant network hash rate and we verify that, under our model, Kraft's time-ratio updating is a linear maximum-likelihood estimate of network hash rate based on block arrival rate. We propose a difficulty adjustment method for Monero similar to the one proposed by Kraft, but with a few differences. In particular, we utilize fixed point iterations and nonlinear dynamics; with nonlinear choices, we may ensure that difficulty adjustment is relatively insensitive to relatively small changes in block arrival rate, which are likely to correspond with stochastic noise.

We use stochastic simulations to compare our derived difficulty adjustment with the CryptoNote reference code and with the Bitcoin code under various piecewise constant hash rate scenarios. In particular, we investigate very large steps up and down. We consider this to be a "worst-case scenario" analysis, i.e. Lex Luthor's mining pool controlling half the network hash rate switches away to another coin. Rather than Kraft's approach, we implement our model of block arrival rate under a piecewise constant network hash rate for a few reasons. Piecewise constant functions are dense in L^2 on any compact interval, so these are mathematically convenient functions to work with. Simulating nonhomogeneous Poisson processes with piecewise constant rates of arrival is not difficult. Also, by assuming computers are either mining full-bore or they are not mining at all, we are actually closer to realistic dynamics under a piecewise constant scenario.

We also consider a sort of "timewarp" attack and demonstrate that the current CryptoNote reference code is vulnerable to a certain mode of attack. By this attack, a user manipulating their timestamps may choose to have no impact on difficulty whatsoever, or may choose to disproportionately increase their impact on difficulty.

The model we present herein is simplified in at least two critical ways compared to the true behavior of the network. First, we assume that there are no propagation delays of block discoveries, and second, we assume the usual Nakamoto parent coin selection rule. That is to say, although we will make some comments about parent coin selection rule in Section 2, we will not investigate variants of parent coin selection rules such as Sompolinsky's GHOST rule in this document.

The modeler is stuck with a dilemma. On one hand, as we have done herein, way may assume no propagation delays occur in the cryptocurrency network, which leads to unrealistic behavior. With such an assumption, everyone has perfectly accurate

data and thus every user will select the same parent coin, regardless of parent coin selection rule. Furthermore, in Poisson processes, arrivals may not occur simultaneously, and so we never have more than one chain in the blocktree. On the other hand, we may incorporate assumptions about propagation delays in a cryptocurrency network, but these assumptions are equivalent to assumptions about the cryptocurrency network structure (which is unobservable) and speed (which is estimable). In the former case, we are sacrificing realistic competitive behavior so that we are not forced to make unrealistic assumptions about network structure. In the latter case, we are gaining some realistic competitive behaviors but we are assuming much about network structure.

One could mitigate the problem of making assumptions about network structure by making empirical measurements such as average and standard deviation of block transmission times, inferring data about the network structure. However, if a researcher were to go down the route of estimating network structure, she would also need to study how rules regarding parent coin selection can impact blockchain dynamics in the presence of competing chains. Studying these things in the context of forking blockchains and selfish mining would certainly be a project worthy of effort. However, this is beyond the scope of this document, which is primarily concerned with the behavior of difficulty scores in response to varying network hash rates. Hence, we go with the former assumption, with no propagation delays and with the Nakamoto parent coin selection rule.

Before beginning, we define for the reader some notation we shall use in the sequel. When block arrival rate is viewed as a nonhomogeneous Poisson process, we denote the instantaneous rate of block arrivals on the network as $\lambda(t)$. We wish to keep instantaneous block arrival rate close to our target block arrival rate, λ^*, which is a known constant. We denote instantaneous network hash rate as $H(t)$, which is unknown and not directly observable (although we run simulations with piecewise constant $H(t)$). We may occasionally refer to a cryptographic hash function $\mathcal{H}$, and a nonce, x. We consider the blockchain to be a sequence of blocks, $\mathcal{B}_0$, $\mathcal{B}_1$, ... Each block, $\mathcal{B}_i$, consists of a difficulty score and a (possibly false) timestamp:

$$\mathcal{B}_0 = (t_0, d_0), \mathcal{B}_1 = (t_1, d_1), \mathcal{B}_2 = (t_2, d_2), \ldots$$

We begin counting the genesis block as height 0, so the block of height $n-1$ is the n^{th} block to arrive; the number of inter-arrival times corresponds with block height this way. The block of height n as $\mathcal{B}_n$, the difficulty of the block of height n as d_n. We may occasionally refer to a nonce, which we denote x.

For some $m \geqslant 2$, we will generate a sequence of sample block arrival rates, $\hat{\lambda}_i$, for $i = m{+}1, m{+}2, m{+}3, \ldots$, each with sample size m. If $m = 2$, we are only considering the inter-arrival time between the latest two blocks, for example. From these, we compute network hash rate estimates, $\hat{H}_i$. For some $\ell \leqslant m$, we also compute $\overline{H}_i$, for each $i = \ell, \ell+1, \ell+2, \ldots$, the moving average of the network hash rate estimates with sample size ℓ, $\overline{H}_i = \frac{1}{\ell}\sum_{j=1}^{\ell} \hat{H}_{n-j}$. If $\ell = 1$, then the moving average is simplified to our latest estimate of instantaneous hash rate. Notice that m/λ^* is the expected length of time to observe m arrivals from a homogeneous Poisson process

with rate λ^*; so, for example, if our block arrival target is $\lambda^* = (60.0s)^{-1}$ and we wish to use the last five minutes of data to compute our sample block arrival rate, we set $m = 5$. On the other hand, if we wish to use the last two hours of data, we set $m = 120$. Also note that the expected block arrival rate from a homogeneous Poisson process with constant rate λ^* is not the expected block arrival rate from the nonhomogeneous Poisson process with non-constant rate $\lambda(t)$, which is proportional to the time-varying network hash rate.

TODO: Verify that, under our model, Kraft's time-ratio updating is a linear maximum-likelihood estimate of network hash rate based on block arrival rate.

TODO: Ensure that difficulty adjustment is relatively insensitive to relatively small changes in block arrival rate

TODO: Use stochastic simulations to compare our derived difficulty adjustment with the CryptoNote reference code and with the Bitcoin code under various piecewise constant hash rate scenarios. In particular, we investigate very large steps up and down.

TODO: Consider a "timewarp" attack and demonstrate that the current CryptoNote reference code is vulnerable to a certain mode of attack. By this attack, a user manipulating their timestamps may choose to have no impact on difficulty whatsoever, or may choose to disproportionately increase their impact on difficulty.

2 Blockchain growth model

In this section, we define a stochastic process that models the growth of the blockchain over time, and we justify why this model is a good representation of blockchain growth. We also make a few comments about parent coin selection rules. In Section 2.1, we make a general criticism of traditional blockchain methods and describe a possibly novel route of attack based solely on blockchain dynamics.

First, we describe our model, which may be represented formally in the following way; let us worry about justification in a moment. We use as input an unknown, *a priori*, positive hash rate function $H(t) > 0$ with support containing the interval $[0, T)$ for some $T > 0$, where T is a constant denoting the time we stop modeling the network. We could apply more assumptions about $H(t)$ if we feel this is not sufficient. For example, we could also presume that $H(t)$ is bounded below away from zero, say $H(t) > 1$, but this won't change our model very much. We could also presume $H(t)$ is piecewise constant (modeling user hash rates as either on the network at full speed or not), or we could assume $H(t)$ is generated by some other stochastic process, which may or may not be dependent upon the current state of the blockchain. For a general derivation, we simply assume that $H(t)$ is positive with support containing $[0, T)$.

We set initial difficulty $d_0 = 1$. A nonhomogeneous Poisson process governing block arrivals with counting process $N(t)$ is observed. Recall that $N(t)$ corresponds to the total number of block arrivals on the time interval $[0, t)$. Denote the instantaneous rate of block arrivals at time $t < T$ as $\lambda(t) = H(t)/d_{N(t)}$. The nonhomogeneous Poisson process gives rise to block arrival times $t_0, t_1, t_2, \ldots$, which may then be manipulated by a malicious user, giving rise to *manipulated* block arrival times $\hat{t}_0, \hat{t}_1, \ldots$. The value of the denominator in the block arrival rate, $d_{N(t)}$, is block difficulty. This value is generated from a function that is determined by the

manipulated block arrival times and the difficulties of the top blocks. That is to say, we have some function ϕ and we set

$$d_{N(t)} = \phi((\hat{t}_i, d_i)_{i=0}^{N(t)-1})$$

We define a blockchain in this context, then, as the stochastic process consisting of the sequence of ordered pairs $(\hat{t}_i, d_i)_{i=0}^{N(t)}$, where the difficulty $d_n = \phi((\hat{t}_i, d_i)_{i=0}^{n-1})$. Observe two facts: first, the *latest* block (the block with $t_i = \max_j(t_j)$) is not necessarily the top block (which has height $n-1$) because the manipulated timestamps need not occur in the same order as their indices. Second, we have defined this notion so generally that it encompasses a variety of difficulty adjustment methods. For example, Bitcoin has a difficulty adjustment period of 2016 blocks. That is to say, for Bitcoin, the function ϕ is dependent upon n, so that if $n \cong 0(modp)$ for some integer p, then for $k = 0, 1, 2, \ldots, 2015$, $d_n = d_{n+k}$.

To justify this model, recall how the proof-of-work competition for block validation works. Users collect transactions into blocks for validation and try to hash nonces together with block data in order to find a hash smaller than a certain target. That is to say, if a user on the network finds some x such that $d_n \cdot \mathcal{H}(\mathcal{B}+x) < \texttt{fixed\_target}$, they have earned the right to declare a block as valid and they usually receive a block reward in the form of a coinbase transaction. Everyone who is working off of the same copy of the blockchain as each other will compute difficulty, d_n in the same way. When a user finds such an x, they publish $\mathcal{B}$, x, and a possibly untrustworthy timestamp. They then recompute their difficulty. When a user hears about a new block, they add it to their copy of the blockchain and recompute their difficulty before trying more hashes.

Since the goal of the proof-of-work game is to find x such that $d_n \cdot \mathcal{H}(\mathcal{B}+x) <$ `fixed_target` and since the output of a good hash function is, in practice, indistinguishable from a uniform distribution, this implies the probability that any given nonce is a success is precisely$\texttt{fixed_target}/d_n$. That is to say, each trial testing a nonce is a Bernoulli trial (weighted coin flip) with probability of success inversely proportional to the difficulty of the next block to be added. Without loss of generality, we may choose $\texttt{fixed\_target} = 1$; doing so calibrates the difficulty score such that $d_n = 1$ corresponds with a success on each and every nonce. Hence, each trial testing a nonce is a Bernoulli trial (weighted coin flip) with probabiltiy of success $1/d_n$.

Arrivals of heads-up coins in a sequence of coin flips, under suitable conditions, can be well approximated with a Poisson process. If we are given a constant $\lambda > 0$, a sequence of probabilities $\{p_n\}$ satisfying $np_n \longrightarrow \lambda$ and a few other suitable conditions, then a binomial random variable $\text{Bin}(n, p_n)$ can be roughly approximated with the Poisson distribution with rate λ. The suitable conditions are technical but a good rule of thumb is that when $n \geqslant 20$ and $p \leqslant 1/20$, or if $n \geqslant 100$ and $np \leqslant 10$, we may use the Poisson approximation. In our case, we are talking about cryptocurrency networks with n typically much greater than thousands of hashes per second, and with probability of success far lower than 1 in 20 nonces. Hence, we may approximate the proof-of-work block competition for validation by a Poisson process with rate $\lambda = np$. More technical arguments can be made toward this equivalence, but we shall be satisfied and proceed with a Poisson process model.

Note that, in the Poisson process described above, n is proportional to our global network hashrate, $H(t)$, and $p = 1/d_n$. The block arrival rate will be $\lambda(t) = np = \beta H(t)/d_n$. Consider β. If difficulty is 1, every nonce is a success, and if $H(t) = 1H/s$, then a block will arrive with probability 1 by time $t = 1.0s$, yielding a block arrival rate of $1.0s^{-1}$. Hence, we have $1.0s^{-1} = \beta\frac{1.0H/s}{1}$. We conclude $\beta = 1$ and we are free to use the relationship

$$\lambda = \frac{H}{d}$$

Furthermore, since users assign difficulty d_n according to some function of the preceding difficulties and timestamps, this is a nearly complete justification of our model in question. As mentioned in Section 1, we largely ignore parent-coin selection rules in this document, although many interesting questions may arise (see Section 6). For completeness, however, we shall discuss parent-coin selection for a moment.

Many decision rules for choosing parent coins could be constructed. One common misconception of the Bitcoin whitepaper, [5], is that Satoshi Nakamoto did not propose mining on the longest chain, but, in fact, the chain with the largest cumulative difficulty. In the case of Bitcoin, for which thousands of blocks in a row have the same difficulty score, this is often (although not always) equivalent to selecting a chain with the highest block height. Alternative proposals have been made, e.g in [6] Sompolinsky and Zohar recommend a greedy algorithm seeking the heaviest subtree approach. Users determine their parent coin by climbing the blocktree from the genesis block upward, and each time they are faced with a branch, they take the branch leading to the heaviest subtree. When they have finished climbing the blocktree, they have found their parent coin.

These approaches are, in fact, identical if we view as a generalized Nakamoto rule: assign a generalized "score" to each block and choose the parent block by selecting the block with the largest *cumulative* score of all preceding blocks. The only difference between these two methods is how the scores are computed. The original Nakamoto recommendation was to use difficulty of block $\mathcal{B}$ as the score function, whereas the recommendation by Sompolinsky and Zohar is to use the cumulative heaviness (in terms of difficulty) of the subtree that has a block $\mathcal{B}$ as its root. From this perspective, we see that the generalized Nakamoto rule may be sensitive to both choices of score function and difficulty equation.

This is all we shall say on parent-coin selection rules in this document. With that, we have justified the model of choice All that remains is the focus of this document, which is to decide how difficulty is to be computed under the model described above.

2.1 Consequences of the model: Stalling

In this section, we discuss how the model presented in Section 2 is sensitive to sudden changes in hash rate.

Consider a sudden change in hash rate; any traditional blockchain scheme is vulnerable to stalling if a large portion of the network suddenly withdraws its participation. We are creating a discrete object, the blockchain, from a continuous-time stochastic process; if the underlying forces giving rise to the creation of the

blockchain are in a very rapid state of change between blocks, the results may be disastrous. Consider the scenario in which true network hash rate drops by several orders of magnitude very suddenly. For example, if Lex Luthor has control of a very large proportion of some cryptocurrency network, say 99.9%, Lex has some options. He can certainly re-write the history of the blockchain and give himself all of the money (the usual 50% attack route). But he could also simply decide to switch off all of his machines.

Difficulty d_{n+1} remains unchanged, as it is based on an estimate of previous block arrival rates. After all, difficulty only updates and adjusts upon receipt of a new block timestamp. Now, however, block arrival rate is very close to zero blocks per second. Now the problem is that the network comes to a standstill; blocks are not arriving because difficulty was very high and a very large actor in the mining space took their equipment offline. Difficulty will not change before the next block arrives, which could take an arbitrarily long period of time. No one is mining because blocks are very rare, and no one is on the network, so blocks remain rare forever, killing transaction processing capabilities. Of course, a slow decay or smaller jumps in hash rate will not have this effect.

One standard assumption in cryptocurrencies is that no single attacker controls more than 50% of the network (otherwise, the ledger may be re-written, and presumably the currency will lose all value). Hence, we wish to construct a difficulty algorithm robust against a sudden halving or doubling of net hash rate. If hash rate is cut in half but difficulty remains the same, block inter-arrival times will be doubled, and if hash rate is doubled, the inter-arrival times will be halved. Hence, under the standard 50% attacker assumption, no network will be stalled forever. However, with sufficiently long block adjustment periods, this can still be disastrous for a currency. Consider a situation in which the Bitcoin network hash rate is cut in half immediately after a difficutly adjustment. Transactions will now be processed in 20 minute blocks, rather than 10 minute blocks, and thus transaction processing rate on the network will be halved. Furthermore, due to the two week adjustment period in the Bitcoin difficulty adjustment code, we can expect the network to remain in this state *for two weeks*. This would be an agonizingly long period of time for transaction processing speeds to be cut in half. Of course, the Bitcoin network need not be worried about such a scenario, for the size of the Bitcoin network is a good insulator against such attacks.

Due to this, we are interested in measuring the robustness of the two difficulty equations of interest (the CryptoNote reference code and our new difficulty equation) when exposed to sudden large changes in hash rate. In Section **??**, we will investigate hash rate functions that are piecewise constant, and we will investigae how rapidly the two difficulty adjustment algorithms respond to hash rate changes of varying magnitude.

2.2 Consequences of the model: Orphaned blocks

In this seciton, we discuss how block arrival rate relates to the production of orphan blocks. Indeed, we lament the speed of light's inevitable restriction on propagation of data and the resulting orphaned coins. Regardless of parent coin selection rule, regardless of network structure, no matter the speed of our network, we will still

occasionally see computers producing orphan blocks. If all users mine honestly, this is the primary source of orphan blocks, but there is nothing to prevent users from colluding in a selfish mining attack, as described in [2]. Due to some conflict in the community over the term "orphaned block," a more descriptive term perhaps could be *dead branches of the blocktree*.

Consider the following example modeled after Bitcoin with a block target of $\lambda_{\text{target}} = 1/600$ blocks per second. For the sake of argument, presume the network is a complete graph with a constant N nodes and with the same transmission speed between any two nodes. We could model transmission time in the network between node i and node j as $\mu + e_{ij}$ where each e_{ij} is a random variable with $E(e_{ij}) = 0$ and the mean propagation time is μ. In in [1], Decker and Wattenhofer measured the average Bitcoin propagation time between nodes to be $\mu \approx 6.5$ seconds. Although the mean propagation times reported in that publication are somewhat out of date, they are a sufficient starting point for this discussion. When a node finds a successful nonce at time t, they announce this fact on the network. By time $t + 6.5$, only half the network (on average) has heard of this new block. What about a computer in the second half of the network, the blind half? A node in that half of the network hashing between time t and $t + 6.5$ will have chosen a different parent coin because it does not have as much information as the nodes in the first half of the network. Furthermore, one or the other branch will become part of the main chain eventually; these two events are mutually exclusive. Hence the total number of coin flips wasted is, with high probability, bounded by

$$\text{Orphaned flips} \geqslant E\left[\text{Number of heads found by half the network in } [t, t+\mu)\right]$$

and if hash power is distributed approximately uniformly, we can take a simple average. This approximation works out to be $\mu \cdot \lambda_{\text{target}}$ flips, where λ_{target} denotes our target block arrival rate. In this example, this would be about 0.01083 orphaned blocks on average. In expectation, a Bitcoin miner can expect about 1.083 of their blocks orphaned for every 100 blocks mined (once every 16-ish hours). In probability, a Bitcoin miner can be 95% confident that at least one block has been orphaned for every 275 blocks mined. An individual with 1% of the Bitcoin network's hashing power can expect, on average, to receive 1 out of every 100 blocks rewards, which would occur with rate $\lambda_{\text{target}}/100 = 1/1000$ blocks per minute. Hence, to obtain 275 blocks would require about 190 days, and such a miner can be 95% confident she will see one orphaned block every 190 days.

On the other hand, if we have a coin modeled after Monero, with a block target time of $\lambda_{\text{target}} = 1.0$ blocks per minute (compared with Bitcoin's $\lambda_{\text{target}} = 0.1$ blocks per minute), but we choose an otherwise similar setup as above, we will see about 0.1083 orphaned blocks per mined block on average, corresponding to a 6.5-second propagation time. In expectation, a Monero miner can expect 1.083 blocks orphaned/wasted for every 10 mined blocks. In probability, a Monero miner can be 95% confident that at least one block has been orphaned every 26 blocks mined. If a miner has 1% of the Monero network's hashing power, such a miner can expect to receive 1 out of every 100 block rewards, which would occur with rate $\lambda_{\text{target}}/100 = 1/100$ blocks per minute. With 95% probability, such a miner can be 95% confident she will see one orphaned block every 1.8 days.

Before drawing any conclusions, note that these estimates rely upon *expectations*, which we justified by assuming uniformity in hashing power. This is, of course, very false in the case of Bitcoin, which can see many orders of magnitude difference in performance between various mining rigs. In the case of Monero, which has a somewhat egalitarian proof-of-work algorithm, this assumption is less problematic. Either way, these values should not be taken as particularly precise. Rather, these values are intended give a broad idea of a "first-glimpse" into orphan coin analysis. Having said that, notice that the rate of orphan block arrivals is proportional to the target block arrival rate: the arrival rate of Monero blocks is ten times the arrival rate of Bitcoin blocks, Monero miners can expect around 10% of blocks to be orphan blocks, and Bitcoin miners can expect around 1% of blocks to be orphan blocks. Hence, setting target block arrival rates lower (say, one block every two minutes or three minutes) will dramatically reduce the rate of orphans on the Monero network.

3 Difficulty Adjustment Formula

In this section we use the model defined in Section 2 to determine our difficulty adjustment function and we present it as a sequence of steps. Recall that the overall goal is to keep the true rate $\lambda(t)$ to be approximately λ^*, the target block arrival time. Say that a user is given a blockchain of height $n-1$, which has a sequence of (possibly out-of-order) timestamps and difficulties

$$\mathcal{B}_0 = (t_0, d_0), \mathcal{B}_1 = (t_1, d_1), \mathcal{B}_2 = (t_2, d_2), \ldots, \mathcal{B}_{n-1} = (t_{n-1}, d_{n-1})$$

Before mining, the user shall place the timestamps in order, i.e. the *order statistics*, and consider the top m of these:

$$t_{(n-1)} > t_{(n-2)} > \ldots > t_{(n-m)}$$

Define $\Delta_{n-1}T = t_{(n-1)} - t_{(n-m)}$ for each $n \geqslant m$. Notice that each $\Delta_n T$ is computed from the order statistics of a different sequence, and timestamps may occur out of order. We now compute the sample block arrival rate at height $n-1$

$$\hat{\lambda}_{n-1} := m/\Delta_{n-1}T$$

This is a running estimate of the instantaneous block arrival rate based on the maximum likelihood of a Poisson process with a constant rate/intensity. Each time a new block arrives, the *order statistics* are recomputed, and the sample block arrival rate is recomputed from the new *order statistics*. Our goal is to keep these samples very close to our target, λ^*. Further, since $\lambda = H/d$, each of these block arrival rates come equipped with an estimate of network hash rate:

$$\hat{H}_{n-1} = d_{n-1}\hat{\lambda}_{n-1} = md_{n-1}/\Delta_{n-1}T$$

While time series analysis tools may be used to make predictions of $\hat{H}_n$, we suspect that such predictions will not outperform simpler approaches[1]. In our case, for

[1] This may be the subject of a future research bulletin.

difficulty adjustment, it is unnecessary to predict future values of hash rate (which is a complicated problem) if we are capable of producing an accurate estimate of instantaneous hash rate (which is a less complicated problem). Thus we choose our sequence of difficulties d_n so as to allow difficulty to track the simple moving average of these network hashrate estimates; this is, in fact, equivalent to presuming network hash rate will remain static and match our current estimate until the next block arrives. We compute the sample mean of the last ℓ estimates of network hash rate

$$\begin{aligned}\overline{H}_{n-1} =& \frac{1}{\ell}\sum_{i=1}^{\ell}\hat{H}_{n-i}\\ =& \frac{1}{\ell}\sum_{i=1}^{\ell} d_{n-i}\hat{\lambda}_{n-i}\\ =& \frac{1}{\ell}\sum_{i=1}^{\ell} m d_{n-i}/\Delta_{n-i}T\end{aligned}$$

Since $\lambda = H/d$, we may keep $\lambda(t)$ close to λ^* by choosing

$$\begin{aligned}d_{n+1} =& \overline{H}_n/\lambda^*\\ =& \frac{\overline{H}_n}{\overline{H}_{n-1}}\frac{\overline{H}_{n-1}}{\lambda^*}\\ d_{n+1} =& \frac{\overline{H}_n}{\overline{H}_{n-1}}d_n\end{aligned} \tag{3.1}$$

That is to say, hash rate and difficulty should move in lockstep. At this point, we should be somewhat comforted. If average network hash rate has doubled, difficulty should probably double, and if it has halved, difficulty should halve. Notice that if we were to set $\ell = 1$, we would abandon the simple moving average and simply have the instantaneous estimate

$$d_{n+1} = \frac{\hat{H}_n}{\hat{H}_{n-1}}d_n$$

Returning to the more general case, we have

$$\begin{aligned}d_{n+1} =& \frac{\overline{H}_n}{\overline{H}_{n-1}}d_n\\ =& \frac{\frac{1}{\ell}\sum_{i=1}^{\ell} d_{n-i+1}\hat{\lambda}_{n-i+1}}{\frac{1}{\ell}\sum_{i=1}^{\ell} d_{n-i}\hat{\lambda}_{n-i}}d_n\\ =& \frac{\sum_{i=1}^{\ell} m d_{n-i+1}/\Delta_{n-i+1}T}{\sum_{i=1}^{\ell} m d_{n-i}/\Delta_{n-i}T}d_n\\ =& \frac{\sum_{i=1}^{\ell} d_{n-i+1}/\Delta_{n-i+1}T}{\sum_{i=1}^{\ell} d_{n-i}/\Delta_{n-i}T}d_n\end{aligned}$$

We may expand this a bit to write it as a difference equation:

$$\begin{aligned} d_{n+1} &= \frac{\frac{d_n}{\Delta_n T} + \frac{d_{n-1}}{\Delta_{n-1} T} + \cdots + \frac{d_{n-\ell+1}}{\Delta_{n-\ell+1} T}}{\frac{d_{n-1}}{\Delta_{n-1} T} + \cdots + \frac{d_{n-\ell}}{\Delta_{n-\ell} T}} d_n \\ &= \left(1 + \frac{\frac{d_n}{\Delta_n T} - \frac{d_{n-\ell}}{\Delta_{n-\ell} T}}{\frac{d_{n-1}}{\Delta_{n-1} T} + \cdots + \frac{d_{n-\ell}}{\Delta_{n-\ell} T}}\right) d_n \\ d_{n+1} - d_n &= \frac{\frac{d_n}{\Delta_n T} - \frac{d_{n-\ell}}{\Delta_{n-\ell} T}}{\frac{d_{n-1}}{\Delta_{n-1} T} + \cdots + \frac{d_{n-\ell}}{\Delta_{n-\ell} T}} d_n \end{aligned}$$

We will use these formulae interchangeably as our difficulty adjustment function, but for shorthand, we may remember the general inductive rule

$$d_{n+1} = \frac{\overline{H}_n}{\overline{H}_{n-1}} d_n$$

We now describe how to implement this difficulty adjustment algorithm. Presume we are given a blockchain of the form

$$(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1})$$

and a difficulty score of the next block to be added, d_n. Furthermore, presume some timestamp, t_n, has just been announced on the network and we wish to compute d_{n+1}. Denote the sequence of hash rate estimates

$$\texttt{HASH} = \hat{H}_{n-1}, \hat{H}_{n-2}, \ldots \hat{H}_{n-k}$$

where $k = \min(n, m)$. Denote the average of this sequence

$$\texttt{AVG\_HASH} = \frac{1}{600} \sum_{i=1}^{600} \hat{H}_{n-i}$$

Recall that we reject a block (and its timestamp, t_n) as illegitimate if its timestamp is too far away from the latest timestamp. In particular, the CryptoNote reference code rejects a block if $\text{median}(t_{n-i})_{i=1}^{m} > t_n$ or if $t_n > 7200 + \max(t_{n-i})_{i=1}^{m}$. We see no reason to change this window of acceptance. We execute the following procedure for any $n \geqslant 1$:

(1) Append (t_n, d_n) to the blockchain if the timestamp t_n is legitimate.
(2) Store the top m blocks' timestamps temporarily as, say, $\hat{t}_1, \hat{t}_2, \ldots, \hat{t}_m$. If $n < m$, store as many as we have.
(3) Sort these stored timestamps into their order statistics, $\hat{t}_{(1)} < \hat{t}_{(2)} < \ldots < \hat{t}_{(m)}$.
(4) Compute the span of time represented by these timestamps, $\Delta_n T = \hat{t}_{(m)} - \hat{t}_{(1)}$.
(5) If $n < m$, then set $\hat{\lambda}_n := n/\Delta_n T$. Otherwise, set $\hat{\lambda}_n := m/\Delta_n T$.
(6) Compute the estimated hash rate during this span of time, $\hat{H}_n = \hat{\lambda}_n d_n$.
(7) Prepend $\hat{H}_n$ to HASH.
(8) If $n \geqslant m$, remove $\hat{H}_{n-m}$ from HASH.

(9) Compute the new average of the hash rate estimates. If $n < m$, set $\texttt{NEW\_AVG\_HASH} = \frac{1}{n}\sum_{i=0}^{n-1}\hat{H}_{n-i}$. Otherwise, set $\texttt{NEW\_AVG\_HASH} = \frac{1}{m}\sum_{i=0}^{m-1}\hat{H}_{n-i}$.
(10) If $n < m$, set new difficulty $d_{n+1} = 1$. Otherwise, set new difficulty to $d_{n+1} = d_n \cdot \texttt{NEW\_AVG\_HASH}/\texttt{AVG\_HASH}$.
(11) Re-set $\texttt{AVG\_HASH} \leftarrow \texttt{NEW\_AVG\_HASH}$.
(12) Wait until a new block timestamp, t_{n+1}, to arrive and go back to step (1) when it does.

4 The Current CryptoNote Code

In Appendix 6.5, we present a re-implementation by Sarang Noether of the reference CryptoNote difficulty assessment method in python. We describe that difficulty assessment method here and make comparisons between their difficulty adjustment algorithm and ours.

The CryptoNote reference code computes the next difficulty score in the following manner:

(1) Store the top 720 blocks' timestamps temporarily as, say, $\hat{t}_1$, $\hat{t}_2$, ..., $\hat{t}_{720}$.
(2) Store the top 720 blocks' difficulty scores temporarily as, say, $\hat{d}_1$, $\hat{d}_2$, ..., $\hat{d}_{720}$.
(3) Sort the top timestamps in increasing order as $\hat{t}_{(1)}$, $\hat{t}_{(2)}$, ..., $\hat{t}_{(720)}$
(4) Eliminate outlying bottom 1/12 and outlying top 1/12 of each of these lists, leaving $(\hat{t}_{(61)}, \hat{t}_{(62)}, \ldots, \hat{t}_{(660)})$ and $(\hat{d}_{61}, \hat{d}_{62}, \ldots, \hat{d}_{660})$.
(5) Compute the span of time represented by these timestamps, $\Delta t = \hat{t}_{(660)} - \hat{t}_{(60)}$.
(6) Compute the sum of the difficulties, $D = \sum_{i=61}^{660}\hat{d}_i$.
(7) The difficulty of the next block, $\mathcal{B}_{H+1}$, given some target block arrival rate λ blocks per unit time, is then computed from the formula

$$\hat{d}_{721} := \frac{D/\lambda + \Delta T - 1}{\Delta T} \tag{4.1}$$

We make some observations about this approach. First, as usual, timestamps added to the blockchain need not occur in order. In fact, we have no reason to expect that they will. Sorting the timestamps seems reasonable (in fact, we do this in our proposed algorithm in Section 3). Further, removing the outlying elements from the list also seems reasonable. However, notice that the timestamps were sorted separately from the difficulties. Hence, while an outlying timestamp may be removed, it's associated difficulty score is still included in the computation and the associated difficulty score of some other block is subsequently removed as an outlier (although the timestamp associated with the removed difficulty score may not have been an outlier originally!). We discuss this problem in detail in Section 4.1.

Second, notice that the formula for $\hat{d}_{721}$ may seem a bit strange. Recall that, given n sample inter-arrival times $S_1, S_2, \ldots, S_n$, then the maximum likelihood estimate of the Poisson rate is $\hat{\lambda} = n/\sum_i S_i$ or rather, if we have n arrivals in a time interval of width ΔT, then $\hat{\lambda}^{-1} = n/\Delta T$. These are the formulas used in Section 3. Also notice that we may replace a sum, $D = \sum_i d_i$, with a product of the mean, $D = n\overline{d}$. Since the CryptoNote reference code considers the middle 600 blocks after slicing

outliers, we may write

$$\begin{aligned}\hat{d}_{721} =& \frac{D/\lambda + \Delta T - 1}{\Delta T} \\ =& \frac{\hat{\lambda}}{\lambda}\overline{d} + \frac{\Delta T - 1}{\Delta T}\end{aligned}$$

In the CryptoNote reference code, we have a block arrival target of $60.0s$ per block, so to observe 600 blocks should take, on average, $36000s$, so $\frac{\Delta T - 1}{\Delta T} \approx 0.99997$ and this formula may be written as, approximately,

$$\hat{d}_{721} = \frac{\hat{\lambda}}{\lambda} D + 1 \tag{4.2}$$

We provide further discussion of this in Section 4.2.

4.1 Criticisms of the CryptoNote Reference Code: Sorting Timestamps Alone

To see the gravity of the mistake made in re-ordering timestamps in isolation from the difficulties, consider throwing out the outlying measurements incorrectly from a more usual example. Let's say we are measuring the height and weight of everyone in Huntington-Ashland, West Virginia, the fattest city in America. We line everyone up and give them a number according to their position in line. This is their *index*. We then measure their heights and their weights, proceeding from index 1, the first person in line, down to *index* P, the last person in line, where P is the population of the town. These measurements yield the list of data (H_1, W_1), (H_2, W_2), and so on up to (H_P, W_P). Of course, H_i and W_i correspond to the height and weight, respectively, of the i^{th} person in line. Furthermore, let us presume that we wish to discard the top 1/12 of the list and the bottom 1/12 of the list, leaving 5/6 of the population in the list.

However, before analyzing weight by removing outliers, presume we follow a similar procedure as the CryptoNote reference code. We order heights separately from weights:

$$\begin{aligned}\hat{H}_1 =& \min\{H_1, H_2, \ldots, H_P\} \\ \hat{H}_2 =& \min\left\{H_1, H_2, \ldots, H_P \mid H_i \neq \hat{H}_1\right\} \\ \hat{H}_3 =& \min\left\{H_1, H_2, \ldots, H_P \mid H_i \neq \hat{H}_1, \hat{H}_2\right\} \\ &\vdots\end{aligned}$$

Now, $\hat{H}_1$ corresponds to the height of the shortest person in town, but W_1 still corresponds to the weight of the first person in line. Similarly, $\hat{H}_P$ corresponds to the height of the tallest person in town, but W_P still corresponds to the weight of the last person in line. The approach presented by the CryptoNote reference code would then exclude all pairs $(\hat{H}_i, W_i)$ if the index i is in the first $P/12$ or in the last $P/12$ indices.

What data do we have remaining if we do this? We have $\hat{H}_{P/12+1}$, $\hat{H}_{P/12+2}$, …, $\hat{H}_{5P/6}$, which correspond to the heights of the middle quantile. However, we also

have $W_{P/12+1}, \ldots, W_{5P/6}$, which corresponds to the weights of the people *who were in the middle of the line*. While our initial goal was to discard outliers based on their height and weight measurements, what we ended up doing was discarding outliers based on their position in line. While this may have yielded interesting statistical information about how rapidly overweight people make it into a queue, it is not relevant to relating height with weight.

With this simple description, we have demonstrated that the methods used in the CryptoNote reference code for discarding outliers are, at best, inappropriately applied.

4.2 Criticisms of the CryptoNote Reference Code: Lack of Equilibrium Solutions with Constant Hash Rate

Our intuitive notion of difficulty says we should expect difficulty to go up when hash rate has gone up, and for difficulty to go down when hash rate has gone down. If hash rate has not changed, then difficulty should not change. Intuitively, the CryptoNote difficulty assessment formula, Equation 4.1, does a satisfactory job. Indeed, when hash rate doubles, the sample rate $\hat{\lambda}$ doubles, and so the difficulty of the next block will be approximately double the average of the sample blocks.

However, there is a critical problem with the formula, which has to do with equilibrium solutions to the difficulty equation. Again, our intuitive notion is that, if hash rate remains unchanged, then difficulty should remain unchanged, and vice versa. Of course, we may not directly measure hash rate, but we can directly measure rate of block arrivals. So what happens if blocks have been arriving on target? This would be the best possible indication that hash rate has remained unchanged.

If blocks have been arriving on target, then $\hat{\lambda} \approx \lambda$ and so difficulty (approximately) goes up by one according to Equation 4.1. On the other hand, if difficulty is being held (approximately) constant at equilibrium, then $\hat{d}_{721} \approx \overline{d}$ and

$$\begin{aligned} \hat{d}_{721} =& \frac{\hat{\lambda}}{\lambda}\overline{d} + 1 \\ \overline{d} \approx& \frac{\hat{\lambda}}{\lambda}\overline{d} + 1 \\ (1 - \frac{\hat{\lambda}}{\lambda})\overline{d} \approx& 1 \end{aligned}$$

But then we must conclude that our sample rate, $\hat{\lambda}$, must be quite different from our target rate, λ, otherwise we would conclude $0 = 1$, which is absurd. Indeed, difficulty may never be held constant when blocks are arriving on target given Equation 4.1.

This analysis of an equilibrium solution suggests that we have verified that $\hat{\lambda} = \lambda$ does not directly correspond with a static difficulty in the CryptoNote reference code, violating our intuition about difficulty.

5 Comparing Difficulty Adjustment Performance

In Appendix **??**, we present Python code that takes as input a piecewise constant hashrate function and produces a stochastically generated blockchain in the form of a sequence of timestamp-difficulty ordered pairs. In this section, we use this code to

compare the CryptoNote reference code difficulty equation (see Section 4) with the proposed difficulty equation (see Section 3). Recalling our interest in comparing the robustness of these equations against sudden changes in hash rate, and recalling that $\lambda = H/d$ in the model from Section 2, we investigate several hash rate scenarios with piecewise constant hash rate functions. The overall goal of the difficulty equations is to keep observed block arrival rates, $\hat{\lambda}$, close to target block arrival rates, λ^*. Hence, this provides our metric for goodness of a difficulty equation: we compute the relative error between sample block arrival rate and target block arrival rate for each of our difficulty adjustment equations and for each hash rate function under investigation.

We look into four hash rate scenarios. In the first hash rate scenario, we investigate a piecewise constant hash function of the form $H(t) = \alpha^{\lfloor \beta t \rfloor}$ where $\lfloor x \rfloor$ denotes the usual floor function, i.e. $\lfloor x \rfloor = \max\{i \in \mathbb{Z} \mid i \leqslant x\}$, and where we take $\alpha > 1$ controls overall growth rate and $1/\beta > 0$ controls the period of time between hash rate jumps. This would be an exponential growth model. In the second hash rate scenario, we investigate a logistic growth model with $H(t) = K(1 + Ae^{-k\lfloor \beta t \rfloor})^{-1}$ where K is the carrying capacity of the network, k is the overall population growth rate, the constant A satisfies $A = (K - H(0))/H(0)$, and where $1/\beta > 0$ represents the period of time between hash rate jumps. In the third hash rate scenario, we investigate a hash rate that starts small, jumps to a very large value for a sufficiently long enough period of time for difficulty to come to equilibrium, and then drops back to a very small value for the remainder of the simulation (i.e. a tophat function). This way, we can investigate two population-based hash rate models, and we can also investigate the scenario presented in Section 2.1. In the final hash rate scenario, we investigate a square wave function to investigate what sort of effect periodic behavior in mining has upon network hash rate.

5.1 Exponential Growth in Hash Rate

We find that, for large hash rates, the CryptoNote reference code tracks hash rate very well. However, the term $(\Delta T - 1)/\Delta T$ leads to a roughly linear increase in hash rate over time when hash rate is small or constant. This is true regardless of the choice of β, which controls how long hash rate is held constant before changing. In all investigations of exponential growth in hash rate, we find that the difficulty equation presented in Section 3 does a better job of tracking true network hash rate compared to the CryptoNote reference code difficulty equation presented in Section 4. Not only is the relative error between observed block arrival rate and target block arrival rate generally smaller when we use our difficulty equation instead of the CryptoNote reference code, our difficulty equation appears to approach true network hash rate at a faster rate than the CryptoNote reference code.

We numerically validate these claims by investigating two hash rate functions, $H_1(t) = 1.001^{\lfloor t/600 \rfloor}$, representing a 0.1% increase in hash rate every ten minutes, and $H_2(t) = 1.005^{\lfloor t/120 \rfloor}$, representing a 0.5% increase in hash rate every two minutes. Using a given hash rate function, we generate one blockchain using our new difficulty adjustment equation described in Section 3:

$$\underline{\mathcal{B}}_{new} = (\mathcal{B}_{new,0}, \mathcal{B}_{new,1}, \mathcal{B}_{new,2}, \ldots, \mathcal{B}_{new,n_{new}-1})$$

and we generate another blockchain using the CryptoNote reference code difficulty adjustment equation described in Section 4:

$$\underline{\mathcal{B}}_{old} = (\mathcal{B}_{old,0}, \mathcal{B}_{old,1}, \mathcal{B}_{old,2}, \ldots, \mathcal{B}_{old,n_{old}-1})$$

From these blockchain objects, we may compute the sample block arrival rates, $\hat{\lambda}$, at each new timestamp, and we may compare the relative error between $\hat{\lambda}$ and the target block arrival rate, λ^* as a function of the latest timestamp (which may not be the top timestamp).

5.2 Logistic Growth in Hash Rate

We again find that, for large hash rates, the CryptoNote reference code does a decent job. However, when hash rate is small (i.e. the beginning of a logistic population) or held constant (i.e. the end of a logistic population), the linear growth in the CryptoNote reference difficulty equation becomes steadily more apparent as time goes on; the relative error increases regularly over time. The effect of this linear term on the upswing portion of the logistic curve is difficult to detect, but is still present. Not only is the relative error between observed block arrival rate and target block arrival rate generally smaller when we use our difficulty equation instead of the CryptoNote reference code, our difficulty equation appears to approach true network hash rate at a faster rate than the CryptoNote reference code.

We numerically validate these claims by investigating two hash rate functions. We first investigate $H_1(t) = 10^{\alpha}/(1+A^{-k\lfloor \beta t \rfloor}$, where $A = (10^{\alpha} - 10^{\gamma})/10^{\gamma}$, representing a logistic growth model with carrying capacity 10^{α}, initial hash rate 10^{γ}, growth rate k, and $1/\beta$ controls how often hash rate jumps. In particular, with a network carrying capacity of 1.0 petahash per second ($\alpha = 15$) and an initial network hash rate of 1.0 gigahash per second ($\gamma = 9$), with $\beta = 1/6h^{-1}$ so that hash rate changes every six hours, and with a maximum growth rate of $k = 1.0$ megahash per second per day. We also investigate $H_2(t)$ of the same form with a terahash per second carrying capacity ($\alpha = 12$), an initial hash rate of 1.0 gigahash per second ($\gamma = 9$), and with $\beta = 1/6h^{-1}$ so that hash rate changes every six hours, and with a maximum growht rate of 10^7 hashes per second, a ten fold increase over $H_1(t)$.

Using these given hash rate functions, we generate one blockchain using our new difficulty adjustment equation described in Section 3:

$$\underline{\mathcal{B}}_{new} = (\mathcal{B}_{new,0}, \mathcal{B}_{new,1}, \mathcal{B}_{new,2}, \ldots, \mathcal{B}_{new,n_{new}-1})$$

and we generate another blockchain using the CryptoNote reference code difficulty adjustment equation described in Section 4:

$$\underline{\mathcal{B}}_{old} = (\mathcal{B}_{old,0}, \mathcal{B}_{old,1}, \mathcal{B}_{old,2}, \ldots, \mathcal{B}_{old,n_{old}-1})$$

From these blockchain objects, we may compute the sample block arrival rates, $\hat{\lambda}$, at each new timestamp, and we may compare the relative error between $\hat{\lambda}$ and the target block arrival rate, λ^* as a function of the latest timestamp (which may not be the top timestamp).

5.3 Big Swings in Hash Rate

This is the fun, true test of the difficulty equations. We set a small constant $h_0 > 0$ and a large constant $h_1 > 0$. We choose a time interval $[t_1, t_2)$ on which network hash rate will blast up from $H(t) = h_0$ hashes per second to $H(t) = h_1$ hashes per second, and then back down to $H(t) = h_0$ again.

$$H(t) = \begin{cases} h_0 & t \in [0, t_1) \\ h_1 & t \in [t_1, t_2) \\ h_0 & t > t_2 \end{cases}$$

Using this function, we can assess the rate at which sample block arrival rate approaches target block arrival rate after a large jump in network hash rate, and we can also assess how gracefully a difficulty equation can respond to the converse situation when hash rate drops. We investigate only values of h_1, h_0 such that $h_0/h_1 > 1/100$. This would correspond to a user who controls 99% of the network turning off their equipment, and will lead to block arrival rates dropping by two orders of magnitude. Investigating smaller values of h_0/h_1 is unnecessary, for the consequence would be a large stall and no more blocks.

We again see that our difficulty adjustment equation almost always has a smaller relative error between sample and target block arrival rates, and that these values approach each other more rapidly with our difficulty adjustment equation than with the CryptoNote reference code difficulty equation.

5.3.1 Square Wave Hash Rate

If hash rate behaves in a square wave, we may imagine this as applying a periodic forcing function to a difference equation. We investigate this scenario for several reasons. One reason is the possibility that resonance could cause a blowup in network hash rate. Indeed, if a difference equation has a natural frequency, ω, and a periodic forcing function is applied with period P that is an integer multiple of ω, we run the risk of resonance. Another reason to investigate periodic hash rates is because the CryptoNote reference code uses a dubious method of discarding outliers. Indeed, in the CryptoNote reference code, the top 720 blocks are studied and the top and bottom 60 blocks from that list are discarded. Hence, a miner could hop on the network, mine for 60 blocks, and then hop off the network; such a miner would never personally experience an increase in difficulty as a response to their activity. This suggests some time delay properties between hash rates and difficulties which may be revealed with a periodic forcing function through the expressed phase angle.

To these ends, we investigate periodic square wave functions. We choose $h_1 > h_0 > 0$ and a period, P. We choose some timepoint $0 \leqslant t^* < P$ to represent the time in which network activity is "low." We then use a periodic extension of the function

$$H_0(t) = \begin{cases} h_0 & t \in [0, t^*) \\ h_1 & t \in [t^*, P) \end{cases}$$

with period P as our hash rate function. We investigate ranges of $1 > h_0/h_1 > 1/100$ as in the previous case, and we investigate several values for the period P ranging

from $P = 1.0s$ to $P = 2m/\lambda^*$, which represents the amount of time it takes to receive m blocks, where $m > \ell$ is the sample size used to estimate block arrival rates. If the frequency of the periodic hash rate is an integer multiple of difficulty adjustment period, or outlier discarding periods, or even block arrival rate sample size, this window should be wide enough to detect an effect.

5.4 Results

In response to exponential or logarithmic hash rate growth, the new difficulty adjustment equation performs uniformly better than the CryptoNote reference code; not only is relative error between sample block arrival rate and target block arrival rate usually smaller, it decays to zero more rapidly with the new difficulty equation. The CryptoNote reference code performs worse in the case of logistic growth, but sample block arrival rate is still usually kept to within a tolerance of BLAH BLAH INSERT NUMBER HERE compared to target block arrival rate.

In response to the top hat function, the two difficulty equations perform nearly as well as one another, although the new difficulty equation performed moderately better in terms of responding to a large decrease in hash rate. In response to the periodic function, we saw some interesting behaviors.

The most surprising result is that the CryptoNote reference code, which has only a weak resemblance to our difficulty equation (which was derived directly from a blockchain growth model), it does a fair to good job of keeping arrival rates roughly constant in time. The primary concern about this algorithm, it turns out, is not the response to general hash rate trends. In fact, it is a far bigger problem that users may exploit the way that outliers are discarded in order to mine without seeing difficulty change due to their own activity. Nevertheless, our difficulty equation is still uniformly better than the reference code.

6 Further Questions

In this section, we discuss several routes by which the above work could be expanded by interested and motivated readers. In general, it is to our benefit to produce many models and assess their precision and accuracy, and to compare and contrast multiple hypotheses to determine which model does the best job of representing the ground truth. In practice, exploring every possibility is unreasonable. We hope that others extend this work to develop yet better difficulty adjustment methods.

6.1 Time Series Models of Hash Rate

In deriving our difficulty equation, we presumed hash rate of the future will match the average hash rate of the past. The holy grail of any adaptive algorithm is prediction. Time series analysis tools may be used to make statistical predictions of instantaneous hash rate estimate $\hat{H}_n$ based on previous observations $\hat{H}_{n-1}, \hat{H}_{n-2}, \ldots$. An ambitious user could seek a transformation of this time series that is covariance-stationary and use a SARIMA model, for example, to try to predict the next hash rate.

In this document, although we are tempted to use heavy statistical machinery to solve the problem, parsimony suggests that we seek a simpler solution before we seek a more complicated solution. Models such as SARIMA should only be used

if ARIMA models are not doing particularly well, which should only be used if ARMA models are not doing very well, which should only be used if autoregressive (AR) or moving average (MA) models are insufficient. Furthermore, our difficulty adjustment algorithm will be unsupervised, whereas time series analysis problems usually require a modeler to actively tweak and play with data; time series analysis often does not perform well when fully automated and runs the risk of over-fitting. In our case, we decided upon using simple moving averages, but an efficient implementation of a fully automated SARIMA model that punishes overfitting would be a very interesting application of statistical analysis on a network.

6.2 Population Growth Models of Hash Rate

In this document, we generated hash rate functions in a deterministic way; we picked formulae for the hash rate functions directly. We did this in order to see how difficulty equations responded to specific scenarios. However, we could generate hash rate functions stochastically using population models and various stochastic implementations of those models.

For example, we may investigate the logistic growth, which obeys the differential equation $H' = kH(1 - H/H_{max})$ where k is maximum growth rate and H_{max} is the carrying capacity of the network. The solution to this equation takes the form $H(t) = K/(1 + Ae^{-kt})$ where $A = (K - H(0))/H(0)$, which is the form used in Section 5.2. However, rather than determining hash rate determinsitically from this function by plugging in the current time, we could implement a Gillespie algorithm from [3] to create a stochastic hash rate function whose expectation is the logistic solution above. This would allow a researcher to investigate how different assumptions about hash rate growth over time may influence blockchain growth.

One may ask, "why does the author recommend a complication of the hash rate function here, but in Section 6.1 wishes to argue for parsimony?" The answer to this is that in Section 6.1, we are discussing the way a user on the network estimates hash rate. This is the mechanism we are designing, a difficulty equation, which necessarily needs to be simple for computaitonal reasons. On the other hand, here, in Section 6.2, we are discussing which hash rate function we should use as input to our blockchain when testing a difficulty equation for robustness and accuracy. We are asking "how should I test our designed mechanism?" Testing our difficulty adjustment equation against a complicated hash rate is a stress test on the mechanism we have designed.

6.3 Graph Theoretic Modeling and Parent Coin Selection Rules

One other obvious route of extension is to model the cryptocurrency network itself and investigate the relationships between difficulty adjustment, parent coin selection rules, and network structure. One can representing a computer network with a weighted graph, where the weight between nodes represents a propagation delay. The choice of graph is important, for a five-node network with $1s$ separation between each pair of connected nodes, can result in a maximum five second propagation delay (if the nodes are lined up serially), or could result in a maximum one second propagation delay (if the nodes are arranged as a pentagram in a complete graph with five nodes, K_5). Any graph choice must be justified; equating the average

and standard deviation in propagation delay may be sufficient for some statistical purposes, but by no means capture the details of a complicated network.

The presence of propagation delays will necessarily lead to competing chains, which necessitates a choice of parent coin selection rules, such as the "longest chain" rule first proposed by Nakamoto in [5], and the so-called GHOST rule, first proposed by Sompolinsky in [6]. The GHOST rule selects the parent coin by seeking the heaviest observed sub-tree at each fork in the blocktree, and may exhibit certain advantages over the Nakamoto rule. It is possible that the choice of graph structure, parent coin selection rule, and difficulty adjustment formula are deeply inter-related. Given a particular graph structure and difficulty adjustment formula, a cryptocurrency network may prefer the GHOST rule over the Nakamoto rule, or vice versa. Given a particular graph structure and a particular parent coin selection rule, some difficulty adjustment formulae may make no sense, and some may perform very well. Finally, given a particular parent coin selection rule and difficulty adjustment formula, there may be a graph structure for which these choices perform optimally. These questions could lead to very efficient creations.

6.4 Stalling in the Event of Sudden Hash Rate Drop

The problem with stalling cryptocurrency networks in the event of a large hashrate drop can be mitigated in at least one of two ways. The reason the network may stall is because the current estimate of difficulty remains unchanged until a block is added. The usual maximum likelihood estimate (MLE) of a Poisson rate doesn't use all the data we have available. The first way we recommend mitigating a stalling event involves exploiting the fact that we know how long it has been since the last block arrival. This is additional information we have available that is not yet being used. The second way we recommend mitigating a stalling event involves contact with a third party to obtain information about their own, distinct blockchain, whcih is also additional information we have available. Either or both of these avenues could present interesting lines of inquiry.

Our first avenue is to incorporate the right-censored maximum likelihood estimate of a Poisson process rate. Indeed, in our derivation in Section 3, we assume network hash rate remains static and match our current estimate until more information arrives in the form of the next block to be added. However, at any given time, we actually have more information avaiable; that is to say, we know that some block *will be arriving* at some point *after the current time*. Assume that all users are reporting their timestamps honestly. Rather than declaring a blockchain to be a sequence of timestamp-difficulty ordered pairs together with a future difficulty of the form

$$(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1}), d_n$$

and with no *a priori* concept of time attached to the blockchain object (outside of it's timestamps), we could alternatively define a blockchain in the following way. Presuming some height n, we may define a blockchain to consist of a sequence of timestamp-difficulty ordered pairs together with a time, t, and a random variable,

τ, of the form

$$(t_0, d_0), (t_1, d_1), \ldots, (t_{n-1}, d_{n-1}), (\tau, d_n)$$

where the only information we have about τ is the distributional fact that $\tau > t$ with probability 1. If we wish to incorporate data about timestamp manipulation, this may be stored in any distributional assumptions we place on τ.

The way we could implement this is to take our list of known timestamps, $t_0, t_1, \ldots, t_{n-1}$, include the current time, t^*, and sort the results into their ordered statistics, $t_{(0)} < t_{(1)} < \ldots < t_{(n)}$. We then simply use the MLE as usual from here. If all timestamps are being reported honestly on a network with no propagation delays, then we automatically have the ordering $t_0 < t_1 < t_2 < \cdots < t_{n-1} < t^*$. However, since timestamps may occur out of order, it is possible that $t_{(n-i)} < t^* < t_{(n)}$ for some $i = 1, 2, \ldots$.

One problem with this approach is that now every user may use their own time, t^*, to determine current difficulty. As a consequence, two users with the same copy of the blockchain may compute different values for "next difficulty" and there is no way to verify which, if any, are correct. Users with the same data will only compute the same difficulty if they also have synchronized clocks. To worsen matters, it seems this approach leaves open the possibility of a vulnerability. It will be in the users' best interest to obtain a smaller estimate of block arrival rates, if possible, in order to obtain a small estimate of hash rate, and therefore a smaller difficulty. Hence, using the right-censored maximum likelihood approach, each user would set their local clock very far ahead in the future to take advantage of low-difficulty mining. This would cause an upward drift in timestamps compared to the true time. This upward drift may be bounded and slow, in which case the risk of blocks being rejected by the rest of the network will prevent the behavior from getting out of control. On the other hand, the upward drift may be very fast or unbounded. Timestamps on the top of the blockchain will eventually be so far out of whack with reality that no honest miners will get their blocks accepted by relaying nodes.

Our second avenue is to tie the estimate of block arrival rate on the network with the block arrival rates on other networks. For example, if a user has observed n Bitcoin blocks validated since the last Monero block has been validated, she may reasonably presume that either (a) she has a connectivity issue with the Monero network, (b) the Monero network hash rate has suddenly dropped by a factor of $10n$, or (c) the Bitcoin network hash rate has suddenly increased by a factor of $10n$. Due to the size of the Bitcoin network, option (c) is almost certainly false in general. This seems like an elegant solution, but any time we use a third party, verifiability and security become issues. Working out the details of either of these proposals could lead to an enormous improvement to the resilience of any cryptocurrency network.

6.5 Damping

We may wish to damp the amount of change we apply to our difficulty in the case that we believe that timestamps have recently been manipulated. To this end, recall the convenient property of Poisson processes with rate λ: the mean inter-arrival time

is $1/\lambda$ and the variance in the inter-arrival times is $1/\lambda^2$. Hence, if mean inter-arrival time and standard deviation of inter-arrival times are very different, this suggests that timestamps have been manipulated and the process underlying the timestamp creation is not a homogeneous Poisson process.

Hence, testing the squared mean against the variance is a test for whether a process is Poisson or not. Given a sequence of inter-arrival times, say $\{S_1, S_2, \ldots, S_n\}$, a test of whether these inter-arrival times came from a genuine Poisson process is to compare the squared sample mean $\overline{S}^2 = \left(n^{-1}\sum_{i=1}^n S_i\right)^2$ with the (unbiased) sample variance $\mathrm{Var}(S) = (n-1)^{-1}\sum_{i=1}^n (S_i - \overline{S})^2$. Indeed, taking the whole Monero blockchain as a sample, we observe about a 22% difference between mean and the standard deviation! This suggests that, throughout the history of Monero, quite a bit of timestamp manipulation has occurred, although at this point it is not clear to the Monero Research Lab how to quantify the amount or degree of manipulation.

When the squared mean and variance are vastly different, we distrust the notion that the underlying process is Poisson. Furthermore, if this lack of Poisson-icity is due to an attacker manipulating timestamps, then the mean inter-arrival time, which only utilizes $T_{(1)}$ and $T_{(N)}$, the first and last times of arrival, is particularly vulnerable to a *single* attacker changing their timestamp. On the other hand, the sample standard deviation utilizes all inter-arrival times equally in it's computation. Hence, rather than using $\hat{\lambda} = 1/\overline{S}$, another approach is to use $\hat{\lambda} = 1/\sqrt{\mathrm{Var}(S)}$.

Hence, rather than computing d_{n+1} by a multiplicative factor $\overline{H}_n/\overline{H}_{n-1}$, consider an equivalent additive change in difficulty, $d_{n+1} - d_n = (\frac{\overline{H}_n - \overline{H}_{n-1}}{\overline{H}_{n-1}})d_n$. The idea of this approach is to scale this change in difficulty by a factor α which will depend on both the squared sample mean and the sample variance of inter-arrival times. If squared sample mean and sample variance are approximately equal, we will set $\alpha \approx 1$, and if sample meean and sample standard deviation are different, $\alpha \to 0$. To this end, let $\overline{S}^2$ denote the squared sample mean of inter-arrival times and denote $\mathrm{Var}(S)$ denote the sample standard deviation. Many choices of α are reasonable. For example, we may choose

$$\alpha = \mathrm{Exp}\left[-\left|\overline{S}^2 - \mathrm{Var}(S)\right| / \overline{S}^2\right]$$

Notice that, since $\overline{S}^2$ is never zero, since all blocks must arrive with at least one second separating them, this expression is well defined. Also notice that α has the property that when $\overline{S}^2 = \mathrm{Var}(S)$, we have that $\alpha = 1$, and whenever $\left|\overline{S}^2 - \mathrm{Var}(S)\right| / \overline{S}^2 \to \infty$, we have that $\alpha \to 0$. The scaling in the denominator of the exponent provides a probabilistic guarantee that the momentum term α stays reasonable close to 1 under manipulation-free circumstances.

We can consider α a momentum term, where a close match between a true Poisson process and the observed inter-arrival times yields almost no momentum, or we can consider α a trust value, where a close match yields a high trust level. Using this interpretation, a 22% variance between squared sample mean and sample variance would yield an 80.25% trust rating. On the other hand, a mere five percent difference between squared sample mean, $\overline{S}^2$, and sample variance, $\mathrm{Var}(S)$, will provide a trust value of 0.95122, or rather, about a 95% trust rating. An unreasonably large

difference between squared sample mean and sample variance, say 50%, would yield a trust value of $0.5 \leqslant -\ln(\alpha)$, or rather $\alpha \geqslant 0.606$ or a 60.6% trust rating.

As we have previously observed, there is, historically, a 22% difference between squared sample mean, $\overline{S}^2$, and sample variance, $\mathrm{Var}(S)$. This provides a good reference point for examining this momentum term. A 22% difference would provide $-\ln(\alpha) \leqslant 0.22$, yielding $0.8025 \leqslant \alpha$. That is to say, our momentum term would cause our difficulty to adjust at around 80% it's maximal rate.

Appendices

CryptoNote Difficulty Reference Code

```
# DIFFICULTY.py
# Gives difficulty information over time for blocks with given
    timestamps
# Input: block_file WINDOW CUT LAG CHECK_WINDOW [MODE]
#   block_file format: one line per integer timestamp
#   WINDOW: blocks to be used when computing difficulty (720 in
    practice)
#   CUT: blocks on each side of the block window to exclude (60
    in practice)
#   LAG: how far behind we want to be (15 in practice)
#   CHECK_WINDOW: number of blocks to use for median cutoff (60
    in practice)
#   [MODE]: 0 = don't sort difficulties (default); 1 = sort
    difficulties
# Output: block information, one line per block
#   line format: block_id timestamp next_difficulty
    cumulative_difficulty

import sys
from math import floor

window = int( sys.argv[2] )
cut = int( sys.argv[3] )
lag = int( sys.argv[4] )
check_window = int( sys.argv[5] )
target = 60
mode = 0
try:
    if int( sys.argv[6] ) == 1:
        mode = 1
except:
    pass

print "# window is " + str( window ) + " and cut is " + str( cut
    )

# Read timestamps into an integer array
block_file = open( sys.argv[1], 'r' )
timestamps = []
for line in block_file:
    timestamps.append( floor( float( line.strip() ) ) )
cumulative_difficulties = []

print "# read " + str( len( timestamps ) ) + " blocks"

# Apply the median rule
for i in range( check_window, len( timestamps ) ):
    try:
        median = sum( timestamps[i-check_window:i] )
        if timestamp[i] < median:
            timestamp.pop( i )
    except:
        pass
```

```
# Compute the difficulty for the next block
def next_difficulty( timestamps, cumulative_difficulties ):
    if ( len( timestamps ) > window ):
        timestamps = timestamps[0:window]
        cumulative_difficulties = cumulative_difficulties[0:
            window]

    length = len( timestamps )

    # Run some sanity checks
    if len( timestamps ) > window or len( cumulative_difficulties
        ) > window or len( timestamps ) != len(
        cumulative_difficulties ):
        raise Exception( "Incorrect number of blocks" )
    if length <= 1:
        return 1
    if window < 2:
        raise Exception( "Window is too small" )
    if ( 2 * cut > window - 2 ):
        raise Exception( "Cut is too large" )

    timestamps.sort()
    if mode == 1:
        cumulative_difficulties.sort()

    # Compute the cut indices
    if ( length <= ( window - 2 * cut ) ):
        cut_begin = 0
        cut_end = length
    else:
        cut_begin = int( floor( ( len( timestamps ) - ( window -
            2 * cut ) + 1 ) / 2 ) )
        cut_end = cut_begin + window - 2 * cut

    time_span = timestamps[cut_end-1] - timestamps[cut_begin]
    if time_span == 0:
        time_span = 1

    total_work = cumulative_difficulties[cut_end-1] -
        cumulative_difficulties[cut_begin]
    if total_work < 0:
        raise Exception( "Cannot have negative total work" )

    # Assume high is zero; that is, no overflow
    low = total_work * target
    if low + time_span - 1 < low:
        return 0
    else:
        return int( floor( ( low + time_span - 1 ) / time_span )
            )

# Start feeding blocks into the difficulty algorithm
print "# block timestep next_difficulty cumulative_difficulty"
for i in range( len( timestamps ) ):
    offset = ( i + 1 ) - min( i + 1, window + cut )
    if offset == 0:
        offset = 1

    difficulty = next_difficulty( timestamps[offset:i],
        cumulative_difficulties[offset:i] )
```

```
    if i == 0:
        cumulative_difficulties.append( 0 )
    else:
        cumulative_difficulties.append( cumulative_difficulties[i
            -1] + difficulty )

    # Output
    print ' '.join( map( str, [ i, timestamps[i], difficulty,
        cumulative_difficulties[i] ] ) )
```

New Difficulty Reference Code

References

1. Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.
2. Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.
3. Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.
4. Daniel Kraft. Difficulty control for blockchain-based consensus systems. 2015.
5. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system.
6. Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains. *IACR Cryptology ePrint Archive*, 2013:881, 2013.