# The math of Nxt forging

mthcl[*]

June 29, 2014. Version 0.5.1

**Abstract**

We discuss the forging algorithm of Nxt from the probabilistic point of view, and obtain explicit formulas and estimates for several important quantities, such as the probability that an account generates a block, the length of the longest sequence of consecutive blocks generated by one account, and the probability that one concurrent blockchain wins over another one.

## 1 Forging algorithm

In this article we concentrate on the 1-block-per-minute regime, which is not implemented yet. Assume that there are $N$ forging accounts at a given (discrete) time, $B_1, \dots, B_N$ are the corresponding balances, and we denote by

$$b_k = \frac{B_k}{B_1 + \cdots + B_N}, \quad k = 1, \dots, N$$

the proportion of total forging power that the $k$th account has. Then, to determine which account will generate the next block, we take i.i.d. random variables $U_1, \dots, U_N$ with Uniform distribution on interval $(0,1)$, and the account which maximizes $b_k/U_k$ generates the block; i.e., the label $k_0$ of the generating account is determined by

$$k_0 = \underset{j\in\{1,\dots,N\}}{\arg\max} \frac{b_j}{U_j} = \underset{j\in\{1,\dots,N\}}{\arg\min} \frac{U_j}{b_j}. \tag{1}$$

[*]NXT: 5978778981551971141; author's contact information: `e.monetki@gmail.com`, or send a PM at `bitcointalk.org` or `nxtforum.org`

We refer to the quantity $b_k/U_k$ as the *weight* of the $k$th account, and to $b_{k_0}/U_{k_0}$ as the weight of the block, i.e., we choose the account with the maximal weight (or, equivalently, with the minimal inverse weight) for the forging. This procedure is called the *main algorithm* (because it is actually implemented in Nxt at this time), or the *U-algorithm* (because the inverse weights have Uniform distribution).

As a general rule, it is assumed that the probability that an account generates a block is proportional to the effective balance, but, in fact, this is only approximately true (as we shall see in Section 2). For comparison, we consider also the following rule of choosing the generating account: instead of (1), we use

$$k_0 = \arg\min_{j\in\{1,\dots,N\}} \frac{|\ln U_j|}{b_j}. \tag{2}$$

The corresponding algorithm is referred to as *Exp-algorithm* (since the inverse weights now have Exponential probability distribution).

**Important note:** for all the calculations in this article, we assume that all accounts are forging and all balances are effective (so that $B_1 + \cdots + B_N$ equals the total amount of NXT in existence). In the real situation when only the proportion $\alpha$ of all money is forging, one can adjust the formulas in the following way. Making a reasonable assumption that *all* money of the bad guy is forging and his (relative) stake is $b'$, all the calculations in this article are valid with $b = \alpha^{-1}b'$.

## 2 Probability of block generation

Observe that (see e.g. Example 2a of Section 10.2.1 of [10]) the random variable $|\ln U_j|/b_j$ has Exponential distribution with rate $b_j$. By (2), the inverse weight of the generated block is also an Exponential random variable with rate $b_1+\cdots+b_N = 1$ (cf. (5.6) of [11]), and the probability that the $k$th account generates the block is exactly $b_k$ (this follows e.g. from (5.5) of [11]).

However, for U-algorithm the calculation in the general case is not so easy. We concentrate on the following situation, which seems to be critical for accessing the security of the system: $N$ is large, the accounts $2,\dots,N$ belong to "poor honest guys" (so $b_2,\dots,b_N$ are small), and the account 1 belongs to a "bad guy", who is not necessarily poor (i.e., $b := b_1$ need not be very small).

We first calculate the probability distribution of the largest weight among the good guys: for $x \gg \max_{k\geq 2} b_k$ let us write

$$\begin{aligned}
\mathbb{P}\Big[\max_{k\geq 2}\frac{b_k}{U_k} < x\Big] &= \prod_{k\geq 2}\mathbb{P}\Big[U_k > \frac{b_k}{x}\Big] \\
&= \prod_{k\geq 2}\Big(1-\frac{b_k}{x}\Big) \\
&= \exp\sum_{k\geq 2}\ln\Big(1-\frac{b_k}{x}\Big) \\
&\approx e^{-\frac{1-b}{x}}, \qquad (3)
\end{aligned}$$

since $\ln(1-y) \sim -y$ as $y \to 0$ and $b_2+\cdots+b_N = 1-b$. We calculate now the probability $f(b)$ that the bad guy generates the block, in the following way. Let $Y$ be a random variable with distribution (3) and independent of $U_1$, and we write (conditioning on $U_1$)

$$\begin{aligned}
f(b) &:= \mathbb{P}\Big[\frac{b}{U_1} > Y\Big] \\
&= \int_0^1 \mathbb{P}\Big[Y < \frac{b}{z}\Big]\,dz \\
&= \int_0^1 e^{-\frac{1-b}{b}z}\,dz \\
&= \frac{b}{1-b}\Big(1-e^{-\frac{1-b}{b}}\Big). \qquad (4)
\end{aligned}$$

It is elementary to show that $f(b) > b$ for all $b \in (0,1)$ (see also Figure 1), and (using the Taylor expansion) $f(b) = b + b^2 + O(b^3)$ as $b \to 0$.

Let us remark also that, since $f(b) \sim b$ as $b \to 0$ and using a calculation similar to (3), if *all* relative balances are small, the situation very much resembles that under Exp-algorithm (see also (9) below).

## 2.1 Splitting of accounts

Here we examine the situation when an owner of an account splits it into two (or even several) parts, and show that, in general, this strategy is not favorable to the owner.

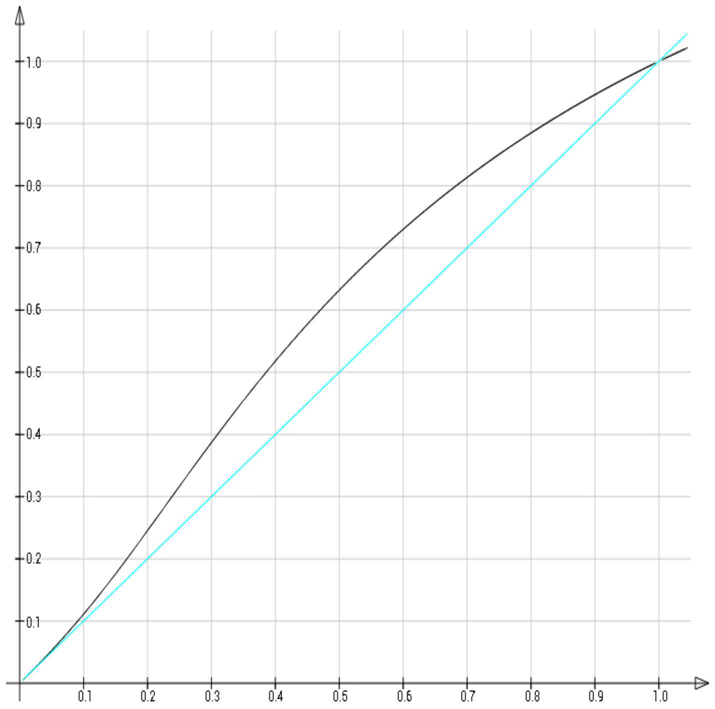

Figure 1: The plot of $f(b)$ (black curve)

First of all, as discussed in the beginning of Section 2, for the Exp-algorithm, the probability that one of the new (i.e., obtained after the splitting) accounts will generate the next block does not change at all. Indeed, this probability is exactly the proportion of the total balance owned by the account, and any splitting does not change this proportion (i.e., all the new accounts forge *exactly* as the old one).

Now, let us consider the case of U-algorithm. We shall prove that splitting is *always* unfavorable for a Nxt holder. Namely, we can prove the following result[1]:

**Theorem 2.1.** *Assume that a person or entity controls a certain number of Nxt accounts, and let $p$ be the probability of generating the next block (i.e., the account that forges the block belongs to this person or entity). Suppose now that one of these accounts is split into two parts (while the balances of all other accounts stay intact), and let $p'$ be the probability of block generation in this new situation. Then $p' < p$.*

By induction, one easily obtains the following

**Corollary 2.2.** *Under the U-algorithm, in order to maximize the probability of generating the next block, all NXT that one controls should be concentrated in only one account.*

*Proof of Theorem 2.1.* Let $b_1, \dots, b_\ell$ be the relative balances of accounts controlled by that person or entity, and let $b_{\ell+1}, \dots, b_n$ be the balances of the other active accounts. Assume without restriction of generality that the first account is split into two parts with (positive) relative balances $b'_1, b''_1$ (so that $b'_1 + b''_1 = b_1$).

Let $U_1, \dots, U_n$, $U'_1$, $U''_1$ be i.i.d. Uniform$[0,1]$ random variables. Let

$$\begin{aligned} Y &= \min_{j=1,\dots,\ell} \frac{U_j}{b_j}, \\ Y' &= \min\Big(\frac{U'_1}{b'_1}, \frac{U''_1}{b''_1}, \min_{j=2,\dots,\ell} \frac{U_j}{b_j}\Big), \\ Z &= \min_{j=\ell+1,\dots,n} \frac{U_j}{b_j}. \end{aligned}$$

[1]The author is happy that he is able to add at least one theorem to this text. Without theorems, he had a strong feeling of doing something *unusual*.

Let us denote $x^+ := \max(0, x)$ for $x \in \mathbb{R}$. Analogously e.g. to (3), we have for $t > 0$

$$\begin{aligned}\mathbb{P}[Y > t] &= \prod_{j=1,\dots,\ell} (1 - b_j t)^+,\\ \mathbb{P}[Y' > t] &= (1 - b_1' t)^+ (1 - b_1'' t)^+ \prod_{j=2,\dots,\ell} (1 - b_j t)^+,\end{aligned}$$

and a similar formula holds for $Z$; we, however, do not need the explicit form of the distribution function of $Z$, so we just denote this function by $\zeta$.

Observe that for $0 < t < \min\left(\frac{1}{b_1'}, \frac{1}{b_1''}\right)$ it holds that

$$\begin{aligned}(1 - b_1' t)(1 - b_1'' t) &= 1 - b_1 t + b_1' b_1'' t^2\\ &> 1 - b_1 t,\end{aligned}$$

so

$$(1 - b_1' t)^+ (1 - b_1'' t)^+ \geq (1 - b_1 t)^+$$

for all $t \geq 0$ (if the left-hand side is equal to 0, then so is the right-hand side), and, moreover, the inequality is strict in the interval $\left(0, \min\left(\frac{1}{b_1'}, \frac{1}{b_1''}\right)\right)$.

Then, conditioning on $Z$, we obtain

$$\begin{aligned}1 - p &= \mathbb{P}[Y > Z]\\ &= \int_0^\infty \prod_{j=1,\dots,\ell} (1 - b_j t)^+ \, d\zeta(t)\\ &= \int_0^\infty (1 - b_1 t)^+ \prod_{j=2,\dots,\ell} (1 - b_j t)^+ \, d\zeta(t)\\ &< \int_0^\infty (1 - b_1' t)^+ (1 - b_1'' t)^+ \prod_{j=2,\dots,\ell} (1 - b_j t)^+ \, d\zeta(t)\\ &= \mathbb{P}[Y' > Z]\\ &= 1 - p',\end{aligned}$$

and this concludes the proof of the theorem. □

One should observe, however, that the disadvantage of splitting under the U-algorithm is not very significant. For example, if one person controls 5% of total active balance and has only one account, then, according to (4), the forging probability is approximately 0.0526. For *any* splitting, this probability cannot fall below 0.05 (this value corresponds to the the extreme situation when all this money is distributed between many small accounts).

**Conclusions:**

1. Under Exp-algorithm, the probability that an account with relative active balance $b$ generates the next block is exactly $b$; if all relative balances are small, then the U-algorithm essentially works the same way as the Exp-algorithm.

2. For the U-algorithm, if an account has proportion $b$ of the total active balance and the forging powers of other accounts are relatively small, then the probability that it generates the next block is given by $f(b)$ of (4).

3. With small $b$, $f(b) \approx b + b^2$, i.e., the block generating probability is roughly proportional to the effective balance with a quadratic correction.

4. It is also straightforward to obtain that the probability that a good guy $k$ generates a block is $b_k(1 - f(b))$, up to terms of smaller order.

5. In general, splitting has no effect on the (total) probability of block generation under Exp-algorithm, and this probability always decreases under U-algorithm. However, the difference is usually not very significant (even if the account is split to many small parts).

6. Thus, neither algorithm encourages splitting (anyhow, there is some cost in maintaining many forging accounts, so, in principle, there is no reason to increase too much the number of them in the case of Exp-algorithm as well). The reader should be warned, however, that all the conclusions in this article are valid for *mathematical models*, and the real world can introduce some corrections.

7. In particular, it should be observed that, if the attacker could harm the network by splitting his account into many small ones, then a very small gain that he achieves by not splitting would not prevent him from attacking the network. If this attacker's strategy presents any real danger, we may consider introducing a *lower limit* for forging (e.g., only accounts with more than, say, 100 NXT are allowed to forge).

## 3 Longest run

We consider a “static” situation here: there are no transactions (so that the effective balances are equal to full balances and do not change over time). The goal is to be able to find out, how many blocks in a row can be typically generated by a given account over a long period $n$ of time.

So, assume that the probability that an account generates the next block is $p$ (see in Section 2 an explanation about how $p$ can be calculated). It is enough to consider the following question: let $R_n$ be the maximal number of consecutive 1's in the sequence of $n$ Bernoulli trials with success probability $p$; what can be said about the properties of the random variable $R_n$?

The probability distribution of $R_n$ has no tractable closed form, but is nevertheless quite well studied, see e.g. [12] (this article is freely available in the internet). The following results are taken from [9]: we have

$$\mathbb{E}R_n = \log_{1/p} qn + \frac{\gamma}{\ln 1/p} - \frac{1}{2} + r_1(n) + \varepsilon_1(n), \tag{5}$$

$$\mathrm{Var}R_n = \frac{\pi^2}{6\ln^2 1/p} + \frac{1}{12} + r_2(n) + \varepsilon_2(n), \tag{6}$$

where $q = 1-p$, $\gamma \approx 0.577\ldots$ is the Euler-Mascheroni constant, $\varepsilon_{1,2}(n) \to 0$ as $n \to \infty$, and $r_{1,2}(n)$ are uniformly bounded in $n$ and very small (so, in practice, $r_{1,2}$ and $\varepsilon_{1,2}$ can be neglected).

In the same work, one can also find results on the distribution itself. Let $\Gamma_p$ be a random variable with Gumbel-type distribution: for $y \in \mathbb{R}$

$$\mathbb{P}[\Gamma_p \leq y] = \exp(-p^{y+1}).$$

Then, for $x = 0, 1, 2, \ldots$ it holds that

$$\mathbb{P}[R_n = x] \approx \mathbb{P}[x - \log_{1/p} qn < \Gamma_p \leq x + 1 - \log_{1/p} qn], \tag{7}$$

with the error decreasing to 0 as $n \to \infty$. So, in particular, one can obtain that

$$\begin{aligned}\mathbb{P}[R_n \geq x] &\approx 1 - \exp(-p^{x+1}qn)\\ &\approx p^{x+1}qn \end{aligned} \tag{8}$$

if $p^{x+1}qn$ is small (the last approximation follows from the Taylor expansion for the exponent).

For example, consider the situation when one account has 10% of all forging power, and the others are relatively small. Then, according to (4), the probability that this account generates a block is $p \approx 0.111125$. Take $n = 1000000$, then, according to (5)–(7), we have

$$\begin{aligned} \mathbb{E}R_n &\approx 6.00273, \\ \mathrm{Var}R_n &\approx 0.424, \\ \mathbb{P}[R_n \geq 7] &\approx 0.009\,. \end{aligned}$$

**Conclusions:**

1. The distribution of the longest run of blocks generated by one particular account (or group of accounts) is easily accessible, even though there is no exact closed form. Its expectation and variance are given by (5)–(6), and the one-sided estimates are available using (8).

## 4 Weight of the blockchain and concurrent blockchains

First, let us look at the distribution of the inverse weight of a block. In the case of Exp-algorithm, everything is simple: as observed in Section 2, it has the Exponential distribution with rate 1. This readily implies that the expectation of the sum of weights of $n$ blocks equals $n$.

As for the U-algorithm, we begin by considering the situation when all relative balances are small. Analogously to (3), being $W$ the weight of the block, for $x \ll (\max_k b_k)^{-1}$ we calculate

$$\begin{aligned} \mathbb{P}\Big[\frac{1}{W} > x\Big] &= \mathbb{P}\Big[\max_k \frac{b_k}{U_k} < \frac{1}{x}\Big] \\ &= \prod_k \mathbb{P}\Big[U_k > xb_k\Big] \\ &= \prod_k (1 - xb_k) \\ &= \exp \sum_k \ln(1 - xb_k) \\ &\approx e^{-x}, \end{aligned} \tag{9}$$

so also in this case the distribution of the inverse weight is approximately Exponential with rate 1.

We consider now the situation when all balances except the first one are small, and $b := b_1$ need not be small. For the case of U-algorithm, similarly to (9) we obtain for $x \in (0, 1/b)$

$$\begin{aligned}\mathbb{P}\Big[\frac{1}{W} > x\Big] &= \prod_k (1 - xb_k) \\ &= (1 - bx) \exp \sum_{k \geq 2} \ln(1 - xb_k) \\ &\approx (1 - bx)e^{-(1-b)x}, \end{aligned} \tag{10}$$

so

$$\begin{aligned}\mathbb{E}\frac{1}{W} &\approx \int_0^{1/b} (1 - bx)e^{-(1-b)x}\, dx \\ &= \frac{be^{-\frac{1-b}{b}} + 1 - 2b}{(1 - b)^2}. \end{aligned} \tag{11}$$

One can observe (see Figure 2) that the right-hand side of (11) is strictly between 1/2 and 1 for $b \in (0, 1)$.

Let us consider now the following attack scenario: account 1 (the "bad guy", with balance $b$) temporarily disconnects from the network and forges its own blockchain; he then reconnects hoping that his blockchain would be "better" (i.e., has smaller sum of weights). Then, while the account 1 is disconnected, the "good" part of the network produces blocks with weights having Exponential distribution with rate $1-b$, and thus each inverse weight has expected value $\frac{1}{1-b}$.

Let $X_1, X_2, X_3, \dots$ be the inverse weights of the blocks produced by the "good part" of the network (after the bad guy disconnects), and we denote by $Y_1, Y_2, Y_3, \dots$ the inverse weights of the blocks produced by the bad guy. We are interested in controlling the probability of the following event (which means that the blockchain produced by the bad guy has smaller sum of inverse weights and therefore wins)

$$H_m = \{X_1 + \dots + X_m - Y_1 - \dots - Y_m \geq 0\}$$

for "reasonably large" $m$ (e.g., $m = 10$ or so). If the probability of $H_m$ is small, this means that the bad guy just does not have enough power to

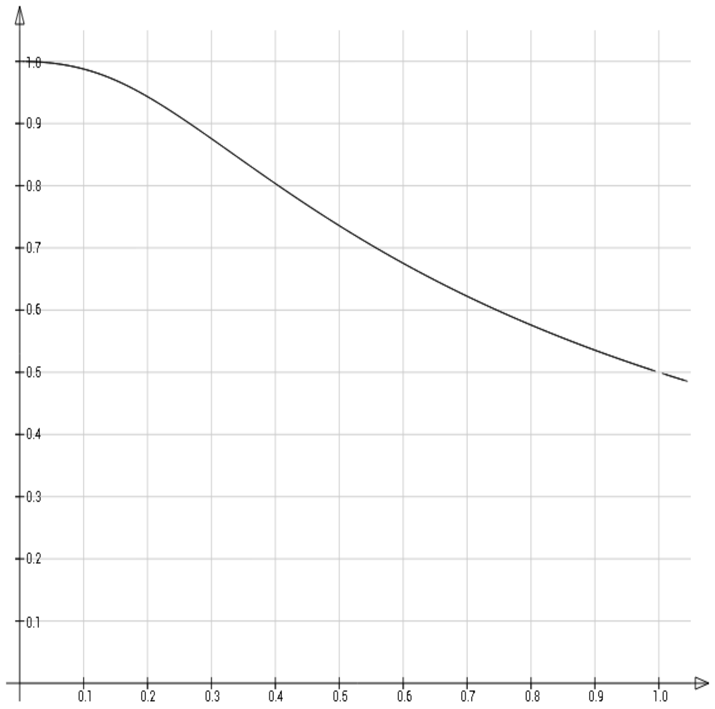

Figure 2: Expectation of the inverse weight (as a function of $b$)

attack the network; on the other hand, if this probability is not small, then the system should be able to fence off the attack by other means, which we shall not discuss in this note.

We obtain an upper bound on the probability of the event $H_m$ using the so-called Chernoff theorem (see e.g. Proposition 5.2 of Chapter 8 of [10]). More specifically, using Chebyshev's inequality and the fact that the random variables are i.i.d., we write for any fixed $t > 0$

$$\begin{aligned}\mathbb{P}[H_m] &= \mathbb{P}[X_1 + \cdots + X_m - Y_1 - \cdots - Y_m \geq 0]\\ &= \mathbb{P}\big[\exp\big(t(X_1 + \cdots + X_m - Y_1 - \cdots - Y_m)\big) \geq 1\big]\\ &\leq \mathbb{E}\exp\big(t(X_1 + \cdots + X_m - Y_1 - \cdots - Y_m)\big)\\ &= \big(\mathbb{E}e^{t(X_1-Y_1)}\big)^m.\end{aligned}$$

Since the above is valid for all $t > 0$, we have

$$\mathbb{P}[H_m] \leq \delta^m, \qquad \text{where} \qquad \delta = \inf_{t>0} \mathbb{E}e^{t(X_1-Y_1)}. \tag{12}$$

It is important to observe that this bound is nontrivial (i.e., $\delta < 1$) only in the case $\mathbb{E}X_1 < \mathbb{E}Y_1$.

For U-algorithm, $X_1$ is Exponentially distributed with rate $1-b$, and $Y_1$ has Uniform$(0, b^{-1})$ distribution. So, the condition $\mathbb{E}X_1 < \mathbb{E}Y_1$ is equivalent to $(1-b)^{-1} < (2b)^{-1}$, that is, $b < 1/3$. Then, for $b < 1/3$, the parameter $\delta$ from (12) is determined by

$$\delta = \delta(b) = b(1-b) \inf_{0<t<1-b} \frac{1-e^{-t/b}}{t(1-b-t)} \tag{13}$$

(see the plot of $\delta(b)$ on Figure 3), so we have

$$\mathbb{P}[H_m] \leq \delta(b)^m. \tag{14}$$

For example, for $b = 0.1$ we have $\delta(b) \approx 0.439$. We have, however, $\delta(b) \approx 0.991$ for $b = 0.3$, which means that one has to take very large $m$ in order to make the right-hand side of (14) small in this case.

For the Exp-algorithm, the bad guy would produce blocks with inverse weights having Exponential distribution with rate $b$, so each inverse weight has expected value $\frac{1}{b}$. Similarly to the above, one obtains that the condition $\mathbb{E}X_1 < \mathbb{E}Y_1$ is equivalent to $b < 1/2$, and

$$\mathbb{P}[H_m] \leq \big(4b(1-b)\big)^m \tag{15}$$

(that is, $\delta$ can be explicitly calculated in this case and equals $4b(1-b)$; observe that $4b(1-b) < 1$ for $b < 1/2$).

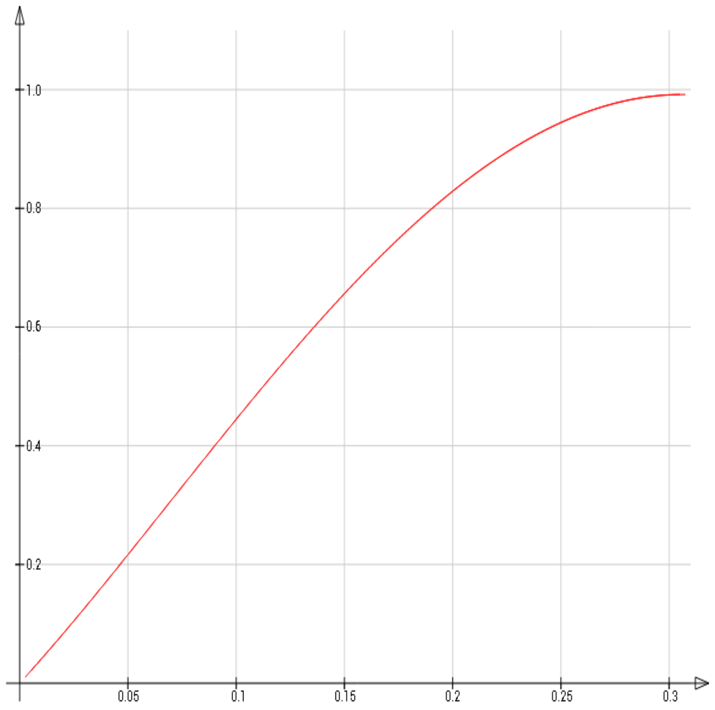

Figure 3: The plot of $\delta(b)$

**Conclusions:**

1. We analyze an attack strategy when one account (or a group of accounts) temporarily disconnects from the main network and tries to forge a "better" blockchain than the one forged by other accounts, in the situation when one bad rich guy has proportion $b$ of total amount of NXT, and the stakes of the others are relatively small.

2. The probability that the bad guy forges a better chain of length $m$ can be controlled using (14) (for the U-algorithm) or (15) (for the Exp-algorithm).

3. It should be observed that this probability does not tend to 0 (as $m \to \infty$) if the bad guy has at least 1/3 of all *active* balances in the network in the case of U-algorithm (correspondingly, at least 1/2 in the case of Exp-algorithm). There should exist some specific methods for protecting the network against such an attack in the case when there is risk that (active) relative balance of the bad guy could become larger than the above threshold.

4. For the current realization of the U-algorithm, the author expects that this analysis can be performed in a quite similar way (because the inverse weight is then proportional to the time to the next block, and the longest blockchain wins), with an additional difficulty due to the oscillating `BaseTarget`.

5. It may be a good idea to limit the forging power of accounts by some fixed threshold, e.g., if an account has more than, say, 1M NXT, then it forges as if it had exactly 1M NXT. Of course, a rich guy can split his fortune between smaller accounts, but then all those accounts would forge roughly as one big account (without threshold) under Exp-algorithm. So, one can use the computationally easier U-algorithm without having its drawbacks (the 1/3 vs. 1/2 issue) discussed in this section.

## 5 More on account splitting

In this section we analyze the following attack strategy: if the bad guy wins the forging lottery at the current step but owns *several* accounts with inverse weights less than all the accounts of good guys (i.e., these accounts of the bad guy are first in the queue), then he chooses which of his accounts will forge. Effectively, that means that he has several independent tries for choosing the hash of the block, and so he may be able to manipulate the lottery in his favor.

Of course, following this strategy requires that the balance of the winning accounts should be small (because of the ban for non-forging), but, as we shall see below, splitting into small parts is exactly the right strategy for maximizing the number of the best accounts in the queue.

First of all, let us estimate the probability that in the sequence of accounts ordered by the inverse weights, the first $k_0$ ones belong to the same person or entity, who controls the proportion $b$ of all active balance. We will do the calculations for the case of Exp-algorithm, since the attacker would have to split his money between many (or at least several) accounts anyway, and, as observed in Section 2, in this situation both algorithms work essentially in the same way.

One obvious difficulty is that we do not know, how exactly the money of the attacker are distributed between his accounts. It is reasonable, however, to assume that the balances of the other accounts (those not belonging to the attacker) are relatively small. Let us show the following remarkable fact:

**Proposition 5.1.** *The best strategy for the attacker to maximize the probability that the best $k_0$ accounts belong to him (under the Exp-algorithm), is to split his money uniformly between many accounts.*

*Proof.* We assume that $r$ is the *minimal* relative balance per account that is possible, and let us assume that the attackers money are held in accounts with relative balances $n_1 r, \dots, n_\ell r$, where $\ell \geq k_0$. Denote also $m = n_1 + \dots + n_\ell$, so that $b = mr$. Now, we make use of the elementary properties of the Exponential probability distribution discussed in the beginning of Section 2. Consider i.i.d. Exponential($r$) random variables $Y_1, \dots, Y_m$, and let $Y_{(1)}, \dots, Y_{(k_0)}$ be the first $k_0$ order statistics of this sample. Then, abbreviating $s_j = n_1 + \dots + n_j$, $s_0 := 0$, we have that

$$Z_j = \min_{i=s_{j-1},\dots,s_j} Y_i, \qquad j = 1, \dots, \ell$$

are independent Exponential random variables with rates $n_1 r, \dots, n_\ell r$. So, the orders statistics of $Z$-variables form a subset of the order statistics of $Y$-variables; since the inverse weights of the attacker's accounts are the first $k_0$ order statistics of $Z_1, \dots, Z_\ell$, the claim follows. $\square$

The above proposition leads to a surprisingly simple (approximate) upper bound for the probability that the best $k_0$ accounts belong to the attacker. Assume that *all* the accounts in the network have the minimum relative balance $r$; then each account in the (ordered with respect to the inverse weights) sequence has probability $b$ to belong to the attacker. Since $k_0$ should be typically small compared to the total number of accounts, we may assume that the ownerships of the first $k_0$ accounts are roughly independent, and this means that the probability that all the best $k_0$ accounts belong to the attacker should not exceed $b^{k_0}$.

Now, let us estimate the conditional probability $p^*(b)$ that the attacker wins the forging lottery on the next step, given he was chosen to forge at the current step. The above analysis suggests that the number of the best accounts in the queue that belong to the attacker can be approximated by a Geometric distribution with parameter $b$. Now, given that the attacker owns $k$ best accounts in the queue, the probability that he wins the next forging lottery is $1-(1-b)^k$ (since there are $k$ independent trials at his disposal, and the probability that all will fail is $(1-b)^k$).

Using the memoryless property of the Geometric distribution (i.e., as one can easily verify, $\mathbb{P}[X=k]=\mathbb{P}[X=k+n \mid X\geq n]$ if $X$ has the Geometric law) we have that, given that the winning account belongs to the attacker, he also owns the next $k-1$ ones with probability $(1-b)b^{k-1}$. So,

$$\begin{aligned} p^*(b) &= \sum_{k=1}^{\infty}(1-b)b^{k-1}(1-(1-b)^k) \\ &= (1-b)\sum_{j=0}^{\infty} b^j + (1-b)^2\sum_{j=0}^{\infty}(b(1-b))^j \\ &= 1-\frac{(1-b)^2}{1-b(1-b)}, \end{aligned} \tag{16}$$

see the plot of the above function on Figure 4. The quantity $p^*(b)$ is almost $b$ for small stakes (e.g., 0.1099 for $b=0.1$), but dramatically increases for large $b$. For $b=0.9$, for instance, this probability becomes 0.989, i.e., the attacker will be able to forge typically around 90 blocks in a row, instead of just 10.

Observe also, that this calculation applies to the following strategy of the attacker: take the *first* of his accounts that assures that he forges on the next step (so that the attacker minimizes the number of his accounts that

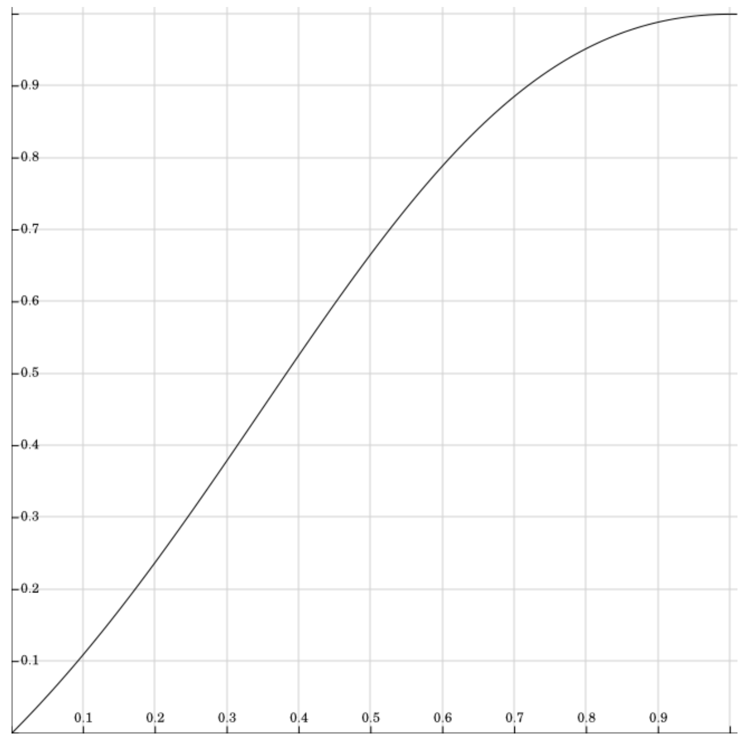

Figure 4: The plot of $p^*(b) = 1 - \frac{(1-b)^2}{1-b(1-b)}$.

will get banned). For this strategy, the attacker looks only one step to the future. One can consider also a more advanced strategy: since the future is (for now) deterministic, the attacker can try to calculate deeper into the future. We shall prove now that, under current implementation of the forging algorithm, *an attacker who owns more than* 50% *of all NXT can eventually forge all the blocks* (i.e., at some moment he starts forging and never stops). To prove this, we construct a *Galton-Watson* branching process (cf. e.g. [8] for the general theory) in the following way. Assume that the block $\ell_0 + 1$ is about to be forged. Let $Z_0 := 1$, and let $Z_1$ be the number of attacker's accounts that are first in the queue (i.e., win over any account not belonging to the attacker). Now, the attacker can choose which of these $Z_1$ accounts will forge. Let $Z_2^{(j)}$, $j = 1, \ldots, Z_1$ be the number of attacker's accounts that are first in the queue for the block $\ell_0 + 2$, provided he has chosen the $j$th account to forge at the previous step. Let $Z_2 = Z_2^{(1)} + \cdots + Z_2^{(Z_1)}$. Then, we define $Z_3, Z_4, Z_5, \ldots$ in an analogous way; it is then elementary to see that $(Z_n, n \geq 0)$ is a Galton-Watson branching process with the offspring law given by $p_k = (1-b)b^k, k \geq 0$. The mean number of offspring

$$\mu = \sum_{k=1}^{\infty} k(1-b)b^k = \frac{b}{1-b}$$

is strictly greater than 1 when $b > \frac{1}{2}$. Since a supercritical branching process survives with positive probability (in fact, one can calculate that in this case the probability of survival equals $\frac{1-b}{b}$), the attacker can choose an infinite branch in the genealogical tree of the branching process, and follow it.

The attacker can also use the same strategy with $b \leq \frac{1}{2}$; this, of course, will not permit him to forge all the blocks, but there is still a possibility to increase the number of generated blocks. Let us do the calculations. The probability generating function of the number of offspring (corresponding to the Geometric distribution) is

$$g(s) = \sum_{j=0}^{\infty} s^j (1-b)b^j = \frac{1-b}{1-sb},$$

so $a_n(b) := \mathbb{P}[Z_n = 0]$ satisfies the recursion

$$a_1(b) = 1 - b, \qquad a_{n+1}(b) = \frac{1-b}{1-ba_n(b)},$$

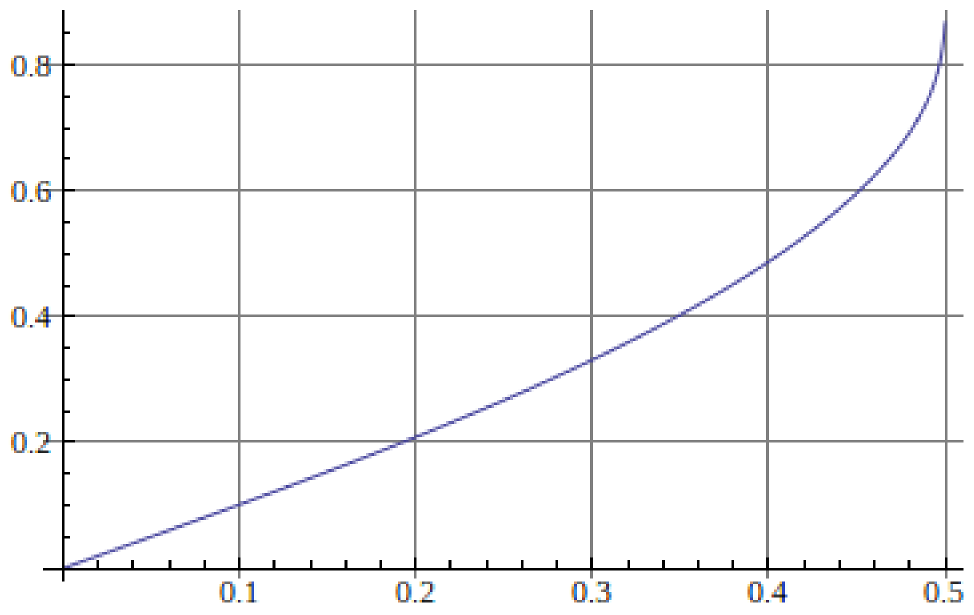

Figure 5: The plot of $\frac{h(b)}{1+h(b)}$ (it is not very evident from this picture, but $\frac{h(b)}{1+h(b)} \to 1$ as $b \to \frac{1}{2}$).

and the mean lifetime $h(b)$ of the branching process is

$$h(b) = \sum_{n=1}^{\infty}(1 - a_n(b)).$$

Unfortunately, usually there is no closed form expression for the expected lifetime of a subcritical branching process, but, as a general fact, it holds that $h(b) \sim b$ as $b \to 0$ and $h(b) \to \infty$ as $b \to \frac{1}{2}$. Since each streak of attacker's blocks has the expected length $b^{-1}h(b)$ and the expected length of each streak of good guy's blocks is $b^{-1}$, the attacker is able to forge the proportion $\frac{h(b)}{1+h(b)}$ of all blocks, see Figure 5 (the author thanks `Mathematica` for doing the computations).

**Conclusions:**

1. The probability that the attacker controls the best $k_0$ accounts can be bounded from above by $b^{k_0}$, where $b$ is the attacker's stake.

2. Under current implementation of the forging algorithm, an attacker who owns more than 50% of all NXT can eventually forge all the blocks (i.e., at some moment he starts forging and never stops).

3. For additional network protection, we can recommend to have also a *lower limit* for forging, i.e., an account that has less than (say) 500 NXT does not forge at all.

4. In fact, it may be a good idea to change this lower limit dynamically: In normal situation (there are not many non-forging events) it can be relatively low. However, if someone is starting playing games (and so there are many non-forging events), then the lower limit increases in order to protect the network. Incidentally, this increase of the lower limit will greatly decrease the attacking strength of the bad guy, since most of his accounts suddenly are unable to forge at all.

5. The community should be warned that if someone is advertising a forging pool but makes the forgers link to many different accounts, then it is a *very* suspicious behavior.

## 6 On the "difficulty" adjustment algorithm

In the current forging implementation (as of June 2014) the intervals between the blocks are not fixed to 1 minute, but are rather random (it is sometimes referred to as the *Blind Shooter* algorithm, cf. [2]). The algorithm of block generation can be described in the following way (to simplify notations, we assume that the total amount of NXT equals 1, and the time is measured in minutes). As before, let $b_i$ be the (relative) balance of $i$th account, and in this section we refer to the Uniform random variable $U_i$ as the *hit* of $i$th account.

Let us just start at time 0 (effectively, the moment of generation of the last block, say, of height $n$), and see when the next block of height $n+1$ appears and which account produces it. The *target* value $T_i(t)$ of $i$th account at time $t$ is defined as

$$T_i(t) = \Lambda_n b_i t, \tag{17}$$

where $\Lambda_n$ is the parameter called *baseTarget*, which will be explained later. When the value of the target of an account reaches its hit value $U_i$, this account may forge a block; this block, however will only be accepted by the network when no other account managed to forge before. That is, the time $t_i$ when the $i$th account has the right to forge is calculated by

$$t_i = \frac{U_i}{\Lambda_n b_i},$$

and (since $\Lambda_n$ is the same for all accounts) the account which minimizes $U_i/b_i$ will effectively forge. Thus, regarding the question of who-will-forge, we recover the algorithm of Section 1.

Observe also that the time to the next block is proportional to the minimal inverse weight, with the factor $\Lambda_n^{-1}$. As noted in Section 4 (see formula (9)), at least in the situation when all relative balances are small, the minimal inverse weight has approximately Exponential(1) distribution, so the time to the next block is approximately Exponential($\Lambda_n$). In particular, the *expected* time to the next block is roughly $\Lambda_n^{-1}$.

Now, the goal is to produce a chain with 1 minute average time between the blocks, so why not just set $\Lambda_n$ to 1 and forget about it? The reason for having this `baseTarget` variable is the following: not all accounts are online, and so sometimes the first account in the queue just would not produce a block because of that. When only $\alpha < 1$ of all NXT are online (i.e., ready to forge and participating in the forging race) arguments similar to the above show that the time to the next block will have approximately Exponential($\alpha$) distribution, with expectation $\alpha^{-1}$. So, in such circumstances it would be desirable to compensate that by setting $\Lambda_n := \alpha$, thus restoring 1 minute average blocktime. However, the problem is that it is quite difficult to determine the current active balance of the network: it would require pinging *every* node, which is clearly impractical.

So, an algorithm for automatic adjustment of the `baseTarget` was proposed. It takes as an input only the last blocktime; intuitively, if the last blocktime was too long, one should increase the `baseTarget` to make the next blocktime shorter, and vice-versa. (Such procedure also introduces negative correlation between the blocktimes, which is good for variance reduction.)

This algorithm can be described in the following way.

- $\Lambda_0 := 1$;
- let $X$ be an Exponential random variable with rate $\Lambda_n$ (one may put $X = \Lambda_n^{-1} \ln U$, where $U$ is a Uniform[0,1] random variable);
- if $X \geq 2$, then $\Lambda_{n+1} = 2\Lambda_n$;

- if $X \leq 1/2$, then $\Lambda_{n+1} = \Lambda_n/2$;
- if $X \in (1/2, 2)$, then $\Lambda_{n+1} = X\Lambda_n$ (note that $X\Lambda_n = \ln U$, so $X\Lambda_n$ is an Exponential(1) random variable).

In the above algorithm, the variable $X$ represents the time to the next block. The idea was $\Lambda_n$ should fluctuate around 1, so we'll have one block per minute in average. In fact, it turns out that this is not the case. Additionally, with this algorithm the rate occasionally becomes rather close to 0, leading to long intervals between the blocks (recall that the expectation of an Exponential($r$) random variable equals $r^{-1}$). The first problem (average blocktime is not 1) is easy to correct by a simple rescaling, but the second one (occasional very long block time) is more serious, since it is an inherent feature of this algorithm.

So, we propose a modified version of this algorithm. Let $\gamma \in (0, 1]$ be a parameter ($\gamma = 1$ corresponds to the current version of the algorithm). Then (abbreviating also $\beta = (1 - \frac{\gamma}{2})^{-1}$)

- $\Lambda_0 := 1$;
- let $X$ be an Exponential random variable with rate $\Lambda_n$;
- if $X \geq 2$, then $\Lambda_{n+1} = 2\Lambda_n$;
- if $X \leq 1/2$, then $\Lambda_{n+1} = \Lambda_n/\beta$;
- if $X \in (1, 2)$, then $\Lambda_{n+1} = X\Lambda_n$;
- if $X \in (1/2, 1)$, then $\Lambda_{n+1} = (1 - \gamma(1 - X))\Lambda_n$.

In words, we make it easier to increase the rate than to decrease it (which happens when $X$ is "too small"). This is justified by the following observation: if $Y$ is an Exponential(1) random variable, then $\mathbb{P}[Y < 1/n] = 1 - e^{-1/n} \approx 1/n$ and $\mathbb{P}[Y > n] = e^{-n}$, and the latter is much smaller than the former. In other words, the Exponential distribution is quite "asymmetric"; even for $n = 2$, we get $\mathbb{P}[Y < 1/2] \approx 0.393$ and $\mathbb{P}[Y > 2] \approx 0.135$.

Now, to evaluate how does the modified algorithm work, there are two methods. First, one can do simulations. Otherwise, we can write the balance equation for the density $f$ of the stationary measure of the above process in

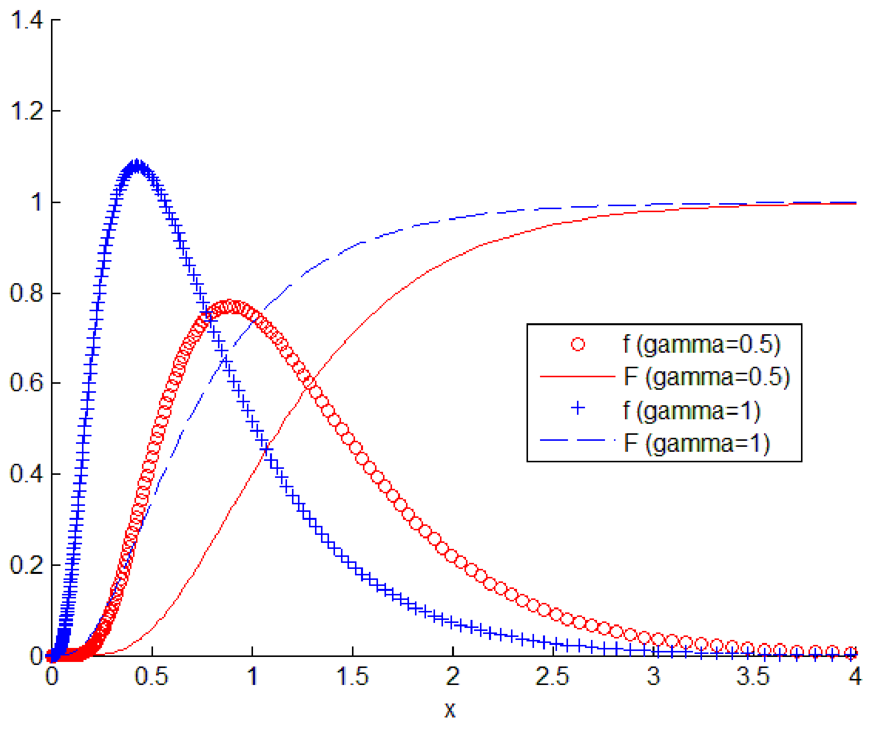

Figure 6: The densities of the stationary measures (with their corresponding cumulative distribution functions) for $\gamma = 1$ and $\gamma = 1/2$

the following way:

$$f(x) = \frac{1}{2}e^{-x}f(x/2) + \beta(1 - e^{-\beta x/2})f(\beta x) + e^{-x}\int_{x/2}^{x} f(s)\,ds + \gamma^{-1}e^{-x/\gamma}\int_{x}^{\beta x} e^{(1-\gamma)s/\gamma}f(s)\,ds. \qquad (18)$$

Here, $f(x)$ is positive for $x > 0$ and $\int_0^\infty f(s)\,ds = 1$ (since $f$ is a density).

There is little hope to obtain a closed form analytic solution to (18), but it is possibly to find a numerical one (many thanks to `Sebastien256`!), see Figure 6. The average blocktime (which equals $\int_0^\infty s^{-1}f(s)\,ds$) can be approximately calculated using the above numerical solution and is roughly 1.954 for the currently used algorithm (with $\gamma = 1$). In comparison, for $\gamma = 1/2$, the average blocktime is 1.015, i.e., quite close to 1 minute. In addition, long blocktimes are also much less probable for this modified algorithm with $\gamma = 1/2$. In the following table, we give probabilities that the `baseTarget` falls below a small parameter $\varepsilon$, thus potentially leading to long (of order $\varepsilon^{-1}$) blocktimes:

| | $\varepsilon = 0.05$ | $\varepsilon = 0.1$ | $\varepsilon = 0.2$ |
|---|---|---|---|
| $\gamma = 1/2$ | $2.243 \cdot 10^{-9}$ | $2.491 \cdot 10^{-6}$ | $5.331 \cdot 10^{-4}$ |
| $\gamma = 1$ | $2.278 \cdot 10^{-4}$ | $4.559 \cdot 10^{-3}$ | $4.574 \cdot 10^{-2}$ |

Now, it's time for another important observation regarding the `baseTarget` adjustment algorithm. In fact, there is a very important difference between `baseTarget` in PoS and *difficulty* in PoW. In PoW, there are no limits on the hashing power of the network: it can be very close to 0, and also it can grow really a lot. However, in PoS there is a clear upper limit on the forging power: just 100% of all available stake.

So, there is no reason at all, why we should even allow the `baseTarget` $\Lambda_n$ to become close to 0! We can introduce another parameter $w > 0$, and never let the `baseTarget` to decrease below $w$ (i.e., if $\Lambda_n$ "attempts" to fall below $w$, we just set it equal to $w$). It is elementary to write down the balance equation for the stationary measure of this version of the algorithm; this stationary measure, however, will be of a "mixed" type (density on $[w, +\infty)$ and an atom in $w$). The numerical analysis similar to the above may then be performed.

We refer also to [1] for related discussions.

**Conclusions:**

1. The current version of the `baseTarget` adjustment algorithm has some disadvantages: it does not lead to 1 minute average blocktimes, and the occasional long blocktimes are quite annoying.

2. An easy-to-implement algorithm (in fact, only a few lines of code need to be modified) with better properties has been proposed in this section. Further research may be needed to find an even better versions.

## 7 On predictability and randomness

All the previous discussion in this paper was restricted to the mathematical model of Section 1, defined in terms of i.i.d. Uniform random variables

$U_1, \dots, U_n$. This, however, is only an approximation of reality (similarly to all other mathematical models in this world); in fact, the $U$-variables are *pseudorandom*, i.e., they are deterministically computed from the information contained in the previous block together with the account hashes. That is, in the static situation (no transactions and the nodes do not connect/disconnect to/from the network) one can precisely determine who will generate the subsequent blocks. Some people have expressed concern whether this situation is potentially dangerous, i.e., an attacker could somehow exploit this in order to forge "too many" (= more than the probability theory permits) blocks in a row. To fence off this kind of treat, we may want to introduce some "true randomness" to the system; i.e., we want the network to produce a "truly unpredictable" random number $U$ (say, with Uniform distribution on $[0, 1]$) in a decentralized way, with no central authority. Of course, one is not obliged to use this random number for forging (see the first remark in "conclusions" below), but nevertheless the ability to produce random numbers may be useful for other applications, e.g., lotteries on top of the Nxt blockchain.

In principle, this is not an easy task, since the nodes controlled by the attacker can cheat by producing some carefully chosen nonrandom numbers, and it is not clear, how does the network recognize who cheats, and who does not. To deal with this problem, we first consider the following algorithm[2]:

- each account obtains a random number (say, in the interval $[0, 1]$) using some *local* randomizing device (e.g., `rand()` or whatever), and publishes the *hash* of this number;

- we calculate the inverse weights of all accounts (using U-algorithm or Exp-algorithm);

- then, first $k_0$ accounts (with respect to the inverse weights, in the increasing order) publish the numbers themselves;

- if at least one the published number does not correspond to its hash or at least one chosen account does not publish its number at all, the corresponding account is penalized (i.e., not allowed to forge during some time), and the whole procedure must be repeated (immediately, or somewhat later);

[2]a similar procedure was proposed in [5].

- we then "mix" these numbers (e.g., by summing them modulo 1[3]), to obtain the random number we are looking for.

The parameter $k_0$ is supposed to be large enough so that the attacker would never control *exactly all* of $k_0$ best (with respect to the inverse weights) accounts. Below we will discuss the question how $k_0$ should be chosen, depending on the maximal amount of active balance that the attacker can obtain. At this point it is important to observe that even one "honest" account in $k_0$ is enough; indeed, for the above mixing method, this follows from the fact that if $U$ is a Uniform$[0, 1]$ random variable and $X$ is *any* random variable independent of $U$, then $(X + U \mod 1)$ is also a Uniform$[0, 1]$ random variable.

Note that this two-step procedure (first publish the hash, and only then the number itself) is necessary. If we do not obscure the numbers, then the attacker can see the $k_0 - 1$ numbers that are already published, and then publish something nonrandom that suits him. If we obscure them first, then the attacker cannot manipulate the procedure.

Let us explain also why the procedure should be restarted when at least one account *attempts* to cheat. It seems to be more "economical" to just pick the next account in the queue if some previous account excludes itself for whatever reason. However, this opens the door for the following attacking strategy. Assume, for example, that first $k_0-1$ accounts have already revealed their numbers, and the $k_0$th and $(k_0 + 1)$th accounts belong to the attacker. Then, he can actually *choose*, which of the two numbers will be published; this, obviously, creates a bias in his favor. In fact, we will see below that the best strategy for the attacker is to have many small accounts, so, invalidating one round of this procedure would not cost much to him. However, still each attempt costs one banned account, and, more importantly, if many rounds of the procedure are invalidated, it is likely that the identity of the attacker could be revealed (one can analyze the blockchain to investigate the origin of the money in offending accounts).

As discussed in Section 5, the probability that all the best $k_0$ accounts belong to the attacker can be bounded from above by $b^{k_0}$. For example, if $b = 0.9$ (i.e., the attacker has 90% of all NXT) and $k_0 = 150$, then $b^{k_0} \approx 0.00000013689$.

[3]By definition, $(x \mod 1)$ is the fractional part of $x$, i.e., set the integer part to 0 and keep the digits after the decimal point.

There are, however, questions about the practical implementation of this algorithm: the difficulty about obtaining consensus on who are the top accounts in the lottery, if such a procedure can slow down the network, etc. Nevertheless, it is conceptually important; in the next section we'll discuss a more "practical" variant of this algorithm.

The contents of this and the next sections is based on private discussions with `ChuckOne` and `mczarnek` at `nxtforum.org`; see also [4].

**Conclusions:**

1. The current forging algorithm is only pseudorandom (deterministic but unpredictable), and there is concern whether this situation could be potentially dangerous.

2. Nevertheless, it is possible to propose an extra randomization algorithm, i.e., the network can achieve a consensus on a Uniform[0,1] random number independent of the previously published data.

3. The procedure for obtaining this random number can be roughly described in the following way. First $k_0$ accounts (with respect to the weights) choose some "random" numbers locally (e.g., take a local output of `rand()`), and publish their hashes. Then, they publish numbers themselves; if the published number does not correspond to the hash or is not published at all, then the corresponding account is penalized. If that happens for at least one account, the whole procedure is invalidated (and we wait for the next try).

4. One can then "mix" the $k_0$ numbers (e.g., by summing them modulo 1); if at least one of the best $k_0$ accounts does not belong to the attacker, the result is "truly random" (it cannot be manipulated, even if we suppose that the attacker controls the other $k_0 - 1$ accounts and cheats by choosing their numbers at will).

5. Because of issues related to the practical implementation, we'll propose a modified version of this algorithm in the next section.

## 8 A proposal for the Transparent Forging algorithm

In this section we describe a forging algorithm which is intended to make a full use of the following advantage that PoS has over PoW: the forgers of the next blocks in PoS are known (at least to some extent), while in PoW they are completely unpredictable. This allows to implement a number of interesting features, which would be hardly accessible in PoW systems, see [6, 7].

We also use a modified version of the randomization algorithm; what we achieve is

- the (main) blocks are forged in a deterministic way, one per minute;
- one can (relatively) accurately predict who will forge the next $L$ blocks;
- however, the forgers of the subsequent blocks after the $L$th one from now, are completely unpredictable.

In the following, a hashchain of length $\ell$ is the sequence $x_1, x_2 = h(x_1), x_3 = h(x_2) \dots, x_\ell = h(x_{\ell-1})$, where $h$ is a hash function (such as, for example, `sha256`), and $x_1$ is the initial seed. The algorithm is then described in the following way:

(a) each forging account must maintain two hashchains of fixed length $S$: active hashchain, reserve hashchain (actually, the nodes must maintain them for their accounts), and publish the last hash of the active hashchain;

   - if active hashchain is *depleted* at block $B$, then the reserve hashchain becomes the active hashchain, next reserve hashchain must be created and its hash must be published in $B$;
   - if forger *invalidates* the active hashchain at block $B$, then the reserve hashchain becomes the active hashchain, next reserve hashchain must be created and its hash must be published in $B$, and the non-forging penalty is applied.

(b) at blocks $N\ell, \ell = 1, 2, 3, \dots$, the list of $K$ richest accounts with respect to the effective balance is formed, and this list becomes *valid* for blocks $N\ell + L, \dots, N(\ell + 1) + L - 1$

(c) special "randomizing" blocks ($R_n, n \geq 1$) are forged just in between of every two main blocks (this can be adjusted, maybe does not need to be so frequent);

(d) randomizing blocks are forged by the accounts from the valid list, e.g., in the cyclic order; they contain the hash from the hashchain preceding to the one already published (so the forger cannot cheat);

(e) for each main block $B_n$, the forging queue of accounts $A_{1,n}, A_{2,n}, A_{3,n}, \ldots$ is formed, according to the algorithm of Section 1 (to limit the number of accounts that need to be considered for the minimization of the inverse weight, the lower forging limit should be introduced);

(f) the accounts in the queue submit their versions of the main block, and the one with the highest *legitimacy* (i.e., the position of the forging account in the forging queue) wins;

(g) to determine the forger of the next block ($B_{n+1}$), the random number we use for $j$th account is `sha256(generating signature of` $B_n$`, public key of the account, sum of hashes in` $L'$ `last randomizing blocks published before the block` $n - L$`)`;

(h) to calculate the generating signature of $B_n$, we always use the public key of $A_{1,n}$ (the account with highest legitimacy at time $n$), even if the block was effectively forged by another account (as discussed below, this is for getting rid of the forks).

One may take e.g. $N = 1440, K = 100, L = 10, L' = 50$, but, of course, these constants may be adjusted. We stress that, to avoid forks, it is essential that the "random number" we use to determine the next forging queue is calculated using the public key of the top account in the queue (i.e., not necessarily of the one who actually forged the block). The same idea may be used for the randomizing blocks as well.

We refer to Figure 7 for an illustration of the blockchains we use for the TF. It remains to explain the chain (actually, the DAG) in the lower part of the picture. As proposed in [3], we may use a "fast" chain or DAG for storing certain kind of information; the set of forgers of this DAG can be the same as the set of forgers of the randomizing blockchain, and we may use the (modified) Blind Shooter algorithm (cf. Section 6) to produce it.

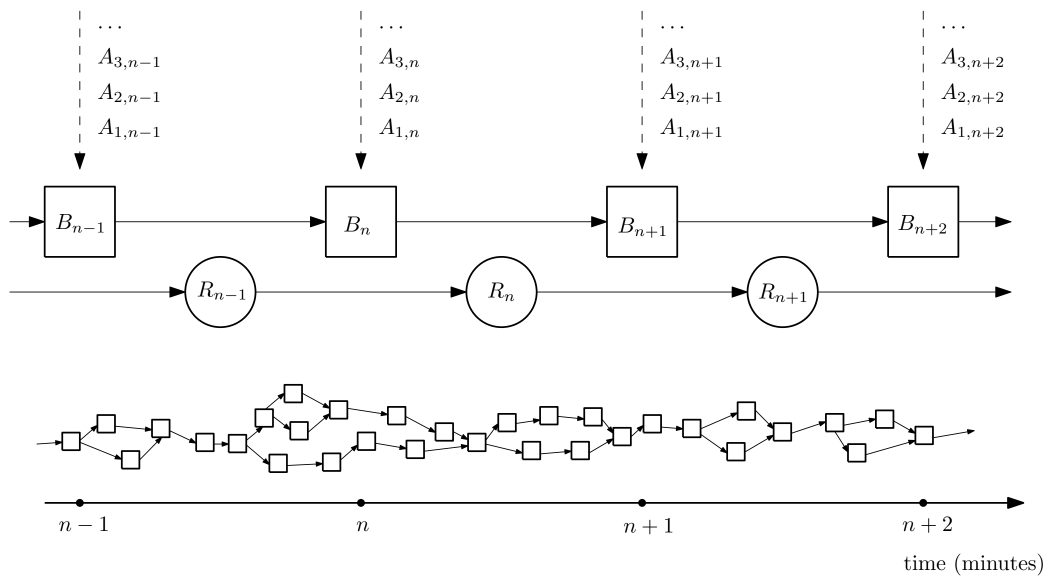

Figure 7: The TF blockchains: main, randomizing, registration (DAG)

This DAG may be used for various purposes, but its importance for the main blockchain construction can be explained in the following way. Assume that the account $A_{1,n}$ (the first one in the forging queue for the $n$th main block) intentionally delays publishing his block, and reveals it only a bit later, when the majority of nodes have already accepted the block forged by $A_{2,n}$ (the second account in that forging queue). For the algorithm currently in use, each forger in some sense validates the previous block; but for the algorithm proposed in this section it can be a problem, since the next forger already assumed that the previous block was forged by $A_{2,n}$. How can we *prove* that $A_{1,n}$ did not publish his block in time? The proposed solution is the following: the nodes that forge the DAG, act as supervisors: if they see the block published by the accounts in the forging queue, they report that in their "registration" blocks. If no supervisor reported the block of $A_{1,n}$ before certain deadline, the network assumes that $A_{1,n}$ did not forge.

A few concluding remarks and explications. An account that must forge the current block but does not do it, is penalized. We may consider also that an account may *declare* on the blockchain if it goes online or offline.

About the penalizations: it is probably a good idea not to apply it immediately, but only after some time, to help the network achieve consensus on who is penalized, and who is not. If an account was penalized on block $n$ for whatever reason, the penalization only becomes effective at the block $n + L''$ (and this account still can legitimately forge between $n + 1$ and $n + L'' - 1$).

With this algorithm, by the way, we can predict the next $L$ forgers (but not more!), which was also a desirable feature of TF, as far as the author remembers.

The point of having the richest accounts do the randomization job is the following: it is probably impossible for the attacker to control e.g. top 100 accounts, that is just too expensive. And another point: since the accounts must be big, cheating by not publishing the random number now becomes very expensive as well (an account that does not forge the randomizing block when it must to, is banned and so unable to forge normal blocks for some period). Anyhow, it seems reasonable that the accounts that have something to lose, do some additional job, like the randomization and the supervision. Also, the author is not convinced that accounts that forge the main chain should be penalized for nonforging: the bad guy can split his balance into many small accounts, retaining his forging power and making those penalizations inefficient.

## Acknowledgments

Participating in this amazing Nxt community is a very rewarding and unique experience, and the author thanks all people who contributed to this article with suggestions and comments. In particular, thanks to `CfB` and `CIYAM` for having enough patience in answering the author's mostly naive questions and so helping him to understand how the forging algorithm works. Many thanks also to `ChuckOne` for very useful discussions and careful reading of this paper and to `BloodyRookie` for helping with some technical issues. `Sebastien256` made a very good job providing a numerical solution to (18). And, last but not least, the author is very grateful for the generous donations he received from the community in support of his work.

## References

[1] MTHCL. BaseTarget adjustment algorithm.
`https://nxtforum.org/proof-of-stake-algorithm/`

basetarget-adjustment-algorithm/ (registration on nxtforum.org required)

[2] Come-from-Beyond. "Blind Shooter" algorithm.
https://nxtforum.org/proof-of-stake-algorithm/'blind-shooter'-algorithm/

[3] mthcl. DAG, a generalized blockchain.
https://nxtforum.org/proof-of-stake-algorithm/dag-a-generalized-blockchain/ (registration on nxtforum.org required)

[4] ChuckOne. JFTR: randomness without cheating.
https://nxtforum.org/transparent-forging/jftr-randomness-without-cheating

[5] mczarnek. Transparent forging algorithm.
nxtforum.org/transparent-forging-*/transparent-forging-algorithm-245/

[6] Come-from-Beyond. Transparent mining, or What makes Nxt a 2nd generation currency.
https://bitcointalk.org/index.php?topic=364218.0

[7] Come-from-Beyond. Transparent mining 2, or What part of Legacy should be left behind.
https://bitcointalk.org/index.php?topic=458036.0

[8] K.B. Athreya, P.E. Ney (1972) *Branching Processes.* Springer-Verlag Berlin–Heidelberg–New York.

[9] L. Gordon, M.F. Schilling, M.S. Waterman (1986) An extreme value theory for long head runs. *Probab. Theory Relat. Fields* **72**, 279–287.

[10] Sheldon M. Ross (2009) *A First Course in Probability.* 8th ed.

[11] Sheldon M. Ross (2012) *Introduction to Probability Models.* 10th ed.

[12] Mark Schilling (1990) The Longest Run of Heads. *The College Math J.*, **21** (3), 196–206.