# Constant Time Fee Redistribution Smart Contract

Bogdan BATOG

December 11, 2017

## 1 Introduction

A "savings account" for cryptocurrencies is proposed in TokenBNK whitepaper, where funds from withdrawal fees are used to reward holders. I explore the practical implementation of this redistribution logic in a smart contract over Ethereum blockchain.

## 2 Economic model

Users can deposit and withdraw funds from the smart contract at any time. At withdrawal time, a fixed percentage of the deposit will be retained and then redistributed proportionally to the remaining deposits. The system thus incentivizes long term holding. In addition, the last one to withdraw will not pay any fee, since there's no other destination to redistribute to.

Note that duration of deposit is not used to determine the fee neither the reward. It's only the relative order between withdrawals that impacts the redistribution.

Every user will pay a fixed percentage fee, so there's a clear bound on the potential loss that may be encountered. Potential gain however, has no bound, as unlimited amounts can be deposited by other users and then withdrawn, thus generating more reward for holders.

## 3 Implementing the fee redistribution

Let's start by considering only one deposit per address. Later I'll also discuss multiple deposits per address (successive addition to the original deposit). Instead, I'll only consider full withdrawals. The withdrawal amount is computed as

$$withdraw = principal - fee + reward \tag{1}$$

Let $principal_t$ denote the original deposited amount that is now asked for withdrawal at moment $t$. Let $T_t$ be the sum of active deposits at moment t (before this withdrawal is processed).

Let $0 < F_P < 1$ be a constant percentage, then the fee paid when withdrawing this deposit is

$$fee_t = F_P * principal_t \tag{2}$$

The redistribution occurs at the withdrawal moment and it reallocates the collected fee to all other active deposits, proportional to their principal amount. This means another deposit $deposit_j$ will get a proportional reward of:

$$reward_{j,t} = principal_j * \frac{fee_t}{T_t - principal_t} \tag{3}$$

A simple implementation is to iterate over all active deposits and increase their stored reward. But such a loop will require more gas per contract call as more deposits are created, making it a costly approach. A more efficient implementation is possible, in O(1) time.

### 3.1 Factor out reward computation

The total reward earned by one deposit is the sum of proportional rewards it extracted from every other withdrawal, while it was active.

$$reward_j = \sum_t reward_{j,t} = principal_j * \sum_t \frac{fee_t}{T_t - principal_t} \tag{4}$$

where $t$ iterates over all withdrawals performed while deposit $j$ was active. Let's note this sum, from beginning of time until time $t$:

$$\sum_t \frac{fee_t}{T_t - principal_t} = S_t \tag{5}$$

This means that if deposit $j$ is created at moment $t_1$ and then withdrawn at moment $t_2 > t_1$, its reward is:

$$reward_j = principal_j * \sum_{t=t_1+1}^{t_2} \frac{fee_t}{T_t - principal_t} = principal_j * (S_{t_2} - S_{t_1}) \tag{6}$$

This form can be implemented in constant time by noting that $S_t$ is monotonic.

### 3.2 Constant time algorithm

The contract will expose two methods, **deposit** and **withdraw**.

**Algorithm 1:** Constant Time Fee Redistribution

function Initialization();
**begin**
    $T = 0$;
    $S = 0$;
    $deposit = \{\}$;
    $S_0 = \{\}$;
**end**

function Deposit $(address, d)$;
**Input :** Amount $d$ to be deposited into $address$
**begin**
    $deposit[address] = d$;
    $S_0[address] = S$;
    $T = T + d$;
**end**

function Withdraw $(address)$;
**Input :** $address$ to withdraw from
**Output:** $amount$ withdrawn
**begin**
    $principal = deposit[address]$;
    $fee = F_p$ * $principal$;
    $reward = principal$ * ($S$ - $S_0[address]$);
    **if** $principal = T$ **then**
        $fee = 0$;
    **else**
        $S = S + fee$ / $(T - principal)$;
    **end**
    $T = T$ - $principal$;
    $deposit[address] = 0$;
    **return** $principal - fee + reward$
**end**

# 4 Contract security

In light of recent Ethereum smart contract exploits it's obvious that even such simple contracts need rigorous security analysis. Let's see what could go wrong.

## 4.1 Front running attack

As in front-running Bancor[1], an entity observing newly placed orders may be able to profit, if it manages to place its own order such that it will be executed before the observed one. This is a general concern with smart contracts where the contract's logic is publicly known and freshly created orders can be observed before they are executed.

In our case, an attacker may observe a large withdrawal has been submitted to the txpool, craft its own deposit order and broadcast it with a greater gas budget, such that it will execute before the withdrawal. As the attacker's deposit amount can be chosen to maximize its share of the large withdrawal, in some cases, it may bring the attacker risk free profit (unless there are other attackers at the same time, so they have to share the potential reward).

Unlike Bancor, where the system can be exploited on both a buy or a sell order, in our case only a withdrawal order can be exploited. This makes the mitigation much simpler: place all deposits from the current block in a queue and actually process them once a new block arrives, effectively delaying them after all withdrawals in their originating block were executed. First order to be executed in a new block will encounter an additional gas expense as the contract will iterate over pending deposits. But the maximum number of transactions in a block is still very small when compared to the (unbounded) number of active deposits, thus the O(1) time per order still stands.

## 4.2 Overflow

The contract accumulates the total amount of fees, in wei, to be distributed to a deposited unit (one billionth of an Ether). It can theoretically grow indefinitely, but will probably need the entire Ether supply to be circulated through our contract for millions of times.

[1] https://hackernoon.com/front-running-bancor-in-150-lines-of-python-with-ethereum-api-d5e2bfd0d798