Evaluation of PoS selection bias attacks on Massa

Rationale

In Massa, we perform PoS draws using the roll distribution from cycle $C-3$ and random seed from $C-2$. That way, attackers can't manipulate the distribution knowing the seed (because the seed is determined later) in order to bias the PoS draws of cycle C. However, attackers can do the opposite: they can manipulate the seed knowing the distribution. To mitigate this attack, in Massa the seed is currently derived by hashing the concatenation of only the first bit of the hash of every slot of cycle $C-2$. That way, each block only brings 1 bit of entropy (and possible manipulation) to the seed. Attackers don't control honest blocks, therefore series of honest blocks would drown the attack in noise. However, attackers being lucky enough to be selected as block producers for the k last slots of cycle $C-2$ can manipulate 1 bit of seed entropy per block, therefore choose the most advantageous draw out of 2^k for the subsequent cycle. Note that this attack is made more difficult in practice because previously produced honest blocks might still be propagating during the attack.

Once the attacker was selected to produce the k last blocks of a given cycle, they can increase their probability of being selected to produce the last blocks of a later cycle by choosing the most favorable draw out of 2^k. Then do it again and again to boost their attack over cycles.

The question is: can attackers significantly boost their probability of getting selected?

Model

In our model, we assume that if attackers of power ρ (ratio of the total stake belonging to the attackers) have the k last blocks of cycle C, they can choose among 2^k possible PoS draws at cycle $C+1$.

On a single draw, we compute the probability p_S that the attacker gets exactly the k last blocks of a cycle. We also make sure that the slot just before (if any) is not the attacker's. This yields a $(1-\rho)$ factor and avoids counting the probability for $k=5$ within the probability of $k=4$ for example, because in both cases the last 4 blocks are the attacker's. N_C is the number of slots in the cycle.

$$p_S(k) = \begin{cases} \rho^k(1-\rho) & \text{if } k < N_C \\ \rho^{N_C} & \text{if } k = N_C \end{cases} \tag{1}$$

We sum $p_S(k)$ over $k = \{0, 1, 2, 3, ..., K\}$ to get the probability $p_L(K)$ that a single draw results in a number of end-of-cycle attacker slots lower or equal to K.

$$p_L(K) = \sum_{k=0}^{K} p_S(k) = \begin{cases} \sum_{k=0}^{K} \rho^k(1-\rho) = 1-\rho^{K+1} & \text{if } K < N_C \\ 1 & \text{if } K = N_C \end{cases} \tag{2}$$

We assume that the attacker had b bits of control during the previous cycle and can therefore attempt 2^b draws and keep the most favorable one. We compute probability $P_D(b, K)$ that all of the 2^b draws result in the attacker getting at most K last cycle blocks, so that selecting the best draw results in at most K last cycle blocks.

$$P_D(b, K) = \left(p_L\left(K\right)\right)^{2^b} = \begin{cases} (1-\rho^{K+1})^{2^b} & \text{if } K < N_C \\ 1 & \text{if } K = N_C \end{cases} \tag{3}$$

The probability $P_E(b, K)$ of the attacker getting exactly K of the last cycle blocks after choosing the best outcome among 2^b draws is:

$$P_E(b, K) = \begin{cases} P_D(b, K) - P_D(b, K-1) = (1-\rho^{K+1})^{2^b} - (1-\rho^K)^{2^b} & \text{if } 0 < K < N_C \\ P_D(b, N_C) - P_D(b, N_C-1) = 1 - (1-\rho^K)^{2^b} & \text{if } K = N_C \\ P_D(b, 0) = (1-\rho)^{2^b} & \text{if } K = 0 \end{cases} \tag{4}$$

$P_E(b, K)$ is therefore the probability of the attacker getting the last K blocks of a cycle if they had the b last blocks previously.

We define a Markov chain where the states labelled $0, 1, 2, 3, ..., \Delta F_0$ represent the number of end-of-cycle blocks that the attacker has at the current cycle. The ΔF_0 state is absorbing and signifies a successful finality fork attack. The transition probabilities $P_T(b \to K)$ from state b to state K are defined as:

$$P_T(b \to K) = \begin{cases} 1 & \text{if } b = K = \Delta F_0 \\ 0 & \text{if } b = \Delta F_0 \text{ and } K \neq \Delta F_0 \\ P_E(b, K) & \text{if } b \neq \Delta F_0 \text{ and } K \neq \Delta F_0 \\ \sum_{k=\Delta F_0}^{N_C} P_E(b, k) & \text{if } b \neq \Delta F_0 \text{ and } K = \Delta F_0 \end{cases} \tag{5}$$

The initial state of the Markov chain (intial probability of state K) is:

$$P_S(K) = \begin{cases} P_E(0, K) & \text{if } K \neq \Delta F_0 \\ \sum_{k=\Delta F_0}^{N_C} P_E(0, k) & \text{if } K = \Delta F_0 \end{cases} \tag{6}$$

This Markov chain effectively simulates the attackers maximizing the number of blocks they get at the end of each cycle by manipulating draws. Each iteration of the Markov Chain represents a cycle.

Results

We simulate 100 years worth of cycles and different attacker powers ρ. A reference "Without Boosting" distribution that does not take into account the probability bias boosting is also computed for comparison. We assume $N_C = 4096$, $\Delta F_0 = 64$ and a throughput of 2 blocks per second.

Results show that for an attacker with power $\rho = 0.3$, bias boosting multiplies the probability of reaching a bias of ΔF_0 by 2.39 but it remains in the order of 1.26×10^{-27}. Bias boosting also multiplies the expected number of attacker cycle last blocks by 1.63 but it remains around 0.7, therefore still lower than 1.

Conclusion

Massa does not seem vulnerable to bias boosting attacks under the defined model.

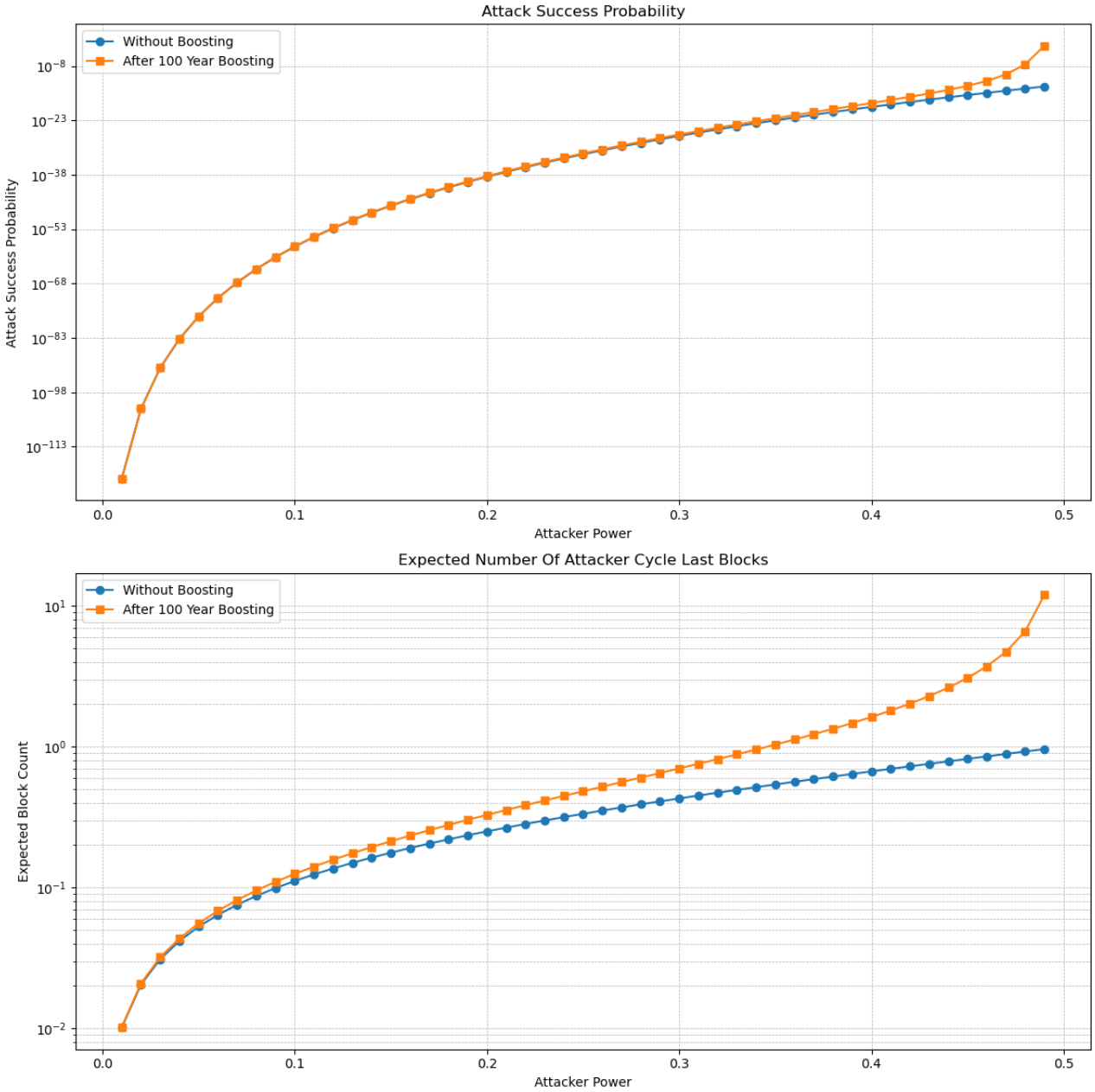

Figure 1: Simulation results with and without PoS bias boosting.

Code

```
import numpy as np
import matplotlib.pyplot as plt
from decimal import Decimal, getcontext
import math
import pandas as pd
from concurrent.futures import ProcessPoolExecutor

# Set the precision to a large enough value to handle the operations
getcontext().prec = 200


def proba(score, trials, cycle_length, bias):
    bias = Decimal(bias)
    if score == 0:
```

```
        return (Decimal(1) - bias) ** trials
    elif score == cycle_length:
        return Decimal(1) - (Decimal(1) - bias ** Decimal(cycle_length)) ** trials
    else:
        return (Decimal(1) - bias ** Decimal(score + 1)) ** trials - (Decimal(1) - bias ** Decimal(scor


def compute_attack_probabilities(bias, cycle_length, deltaF0, attack_cycles):
    # initial distribution
    distrib = np.array([float(proba(score, Decimal(1), cycle_length, bias))
                        for score in range(deltaF0)])
    distrib = np.append(distrib, float(Decimal(bias) ** Decimal(deltaF0)))
    assert np.allclose(np.sum(distrib), 1)

    # Probability to have at least a score of deltaF0 in the initial distribution
    initial_attack_proba = distrib[deltaF0]

    # Expected value
    normal_expected_value = math.fsum(
        [i * distrib[i] for i in range(deltaF0)])/math.fsum([distrib[i] for i in range(deltaF0)])

    # Compute the probability of the attack succeeding after 100 years
    normal_attack_proba = Decimal(
        1) - (Decimal(1) - Decimal(initial_attack_proba)) ** Decimal(attack_cycles)

    # Transition matrix
    transition_matrix = np.zeros((deltaF0+1, deltaF0+1), dtype=np.float64)
    for score_from in range(deltaF0+1):
        trials = Decimal(2) ** Decimal(score_from)
        for score_to in range(deltaF0+1):
            if score_from == deltaF0:
                if score_to == deltaF0:
                    transition_matrix[score_to, score_from] = 1
                else:
                    transition_matrix[score_to, score_from] = 0
            elif score_to == deltaF0:
                transition_matrix[score_to, score_from] = Decimal(
                    1) - (Decimal(1) - Decimal(bias) ** Decimal(deltaF0)) ** trials
            else:
                transition_matrix[score_to, score_from] = float(
                    proba(score_to, trials, cycle_length, bias))
    assert np.allclose(np.sum(transition_matrix, axis=0), 1)

    # apply the transition matrix to the initial distribution over 100 year worth of cycles
    for _ in range(attack_cycles):
        distrib = np.dot(transition_matrix, distrib)
        assert np.allclose(np.sum(distrib), 1)

    # Probability to have at least a score of deltaF0 in the final distribution
    final_attack_proba = distrib[deltaF0]

    # Expected value
    boosted_expected_value = math.fsum(
        [i * distrib[i] for i in range(deltaF0)])/math.fsum([distrib[i] for i in range(deltaF0)])

    return bias, normal_attack_proba, final_attack_proba, normal_expected_value, boosted_expected_value
```

```
def run():
    cycle_length = 4096
    deltaF0 = 64
    dt = 0.5  # time step between blocks
    attack_duration = 100 * 365 * 24 * 3600  # 100 years in seconds
    biases = np.arange(0.01, 0.500, 0.01)

    attack_cycles = math.ceil(attack_duration / (cycle_length * dt))

    results = []
    with ProcessPoolExecutor() as executor:
        futures = [executor.submit(compute_attack_probabilities, bias,
                                   cycle_length, deltaF0, attack_cycles) for bias in biases]
        for future in futures:
            results.append(future.result())

    biases, normal_attack_probas, boosted_attack_probas, normal_expected_value, boosted_expected_value
        *results)

    # Save data to CSV
    data = pd.DataFrame({
        'Attacker_Power': biases,
        'Normal_Attack_Probability': normal_attack_probas,
        'Boosted_Attack_Probability': boosted_attack_probas,
        'Normal_Expected_Value': normal_expected_value,
        'Boosted_Expected_Value': boosted_expected_value
    })
    data.to_csv('attack_probabilities.csv', index=False)

    # Create 2 subplots vertically
    fig, ax = plt.subplots(2, 1, figsize=(12, 12))

    # First subplot
    ax[0].semilogy(biases, normal_attack_probas,
                   label='Without Boosting', marker='o')
    ax[0].semilogy(biases, boosted_attack_probas,
                   label='After 100 Year Boosting', marker='s')
    ax[0].set_xlabel('Attacker Power')
    ax[0].set_ylabel('Attack Success Probability')
    ax[0].set_title('Attack Success Probability')
    ax[0].grid(which='both', linestyle='--', linewidth=0.5)
    ax[0].legend()

    # Second subplot
    ax[1].semilogy(biases, normal_expected_value,
                   label='Without Boosting', marker='o')
    ax[1].semilogy(biases, boosted_expected_value,
                   label='After 100 Year Boosting', marker='s')
    ax[1].set_xlabel('Attacker Power')
    ax[1].set_ylabel('Expected Block Count')
    ax[1].set_title('Expected Number Of Attacker Cycle Last Blocks')
    ax[1].grid(which='both', linestyle='--', linewidth=0.5)
    ax[1].legend()

    plt.tight_layout()
    plt.show()
```

```
run()
```

Data

```
Attacker_Power,Normal_Attack_Probability,Boosted_Attack_Probability,Normal_Expected_Value,Boosted_Expec
0.01,1.539844000000002056226480875769055039755661668934702851623042135143793348465900E-122,1.5557180268
0.02,2.840510818143724872927155429973948773881855950586609203453542634354197272371177435116290836280623
0.03,5.287337428574491294217470947006338274652921597993664921779040645976831361097297355429391905820690
0.04,5.239817610090062670309104469246656387463193364913240367979183342939702867391670179811926279656243
0.05,8.347511050443874727162378179507834457032769407319439584363414437631530510101104774858907244168484
0.060000000000000005,9.753416037625991668905481854992834318349122311153437760939637247750102690054104 70
0.06999999999999999,1.878240868323103526985503792099055069688152575781837817186553589445782490861438519
0.08,9.665757444624780961250768781927266978617104105218747390983900671697801414551271791391101990044167
0.09,1.815504498092324723317469757684726288137286519245130798719076074200683674461643610959180224636216
0.09999999999999999,1.539843999999991894839486259256782175583763054561420899113815444126009895957812151
0.11,6.864494719323106271319162868365505378447225135348684306737008007999519507206767932966025377280583
0.12,1.799187694904996312126505331212095585361341838467018541747520081803277033725478919044628594269251
0.13,3.018917693610495030370008353363643956465807288809227924093683345234089329234052539200461846026342
0.14,3.464742860673872734193371720494059180295184620639793723310751881860200460488518839438496638852843
0.15000000000000002,2.866271363362225879648776268970254525065192161908845609985264777946577019265794316
0.16,1.783017538595461577205734055394117580253525266095516265489267342263405754601985782988206955360909
0.17,8.633913500895657695472855393098525712939474160831164120855604579760012543235602727624985182414913
0.18000000000000002,3.349014684097795725198999024234918010105480939454569933419055865282752175110292426
0.19,1.065877233707064587410940043252213904393993843703028385352533329797496789050754282760631869973099
0.2,2.840510818143731042905740628138941080247531826511426092678401521141187735333489199361263363643189 3
0.21000000000000002,6.449285280173281613424046610436145539477720186100114275362328079271395407965659199
0.22,1.266275772826840225035250740142973119223642472517930889704641542286806114466832238206989327985329
0.23,2.178001779570516129564429332866899403766881170925834942853522075140232889306625352446892733504371
0.24000000000000002,3.318915494848013447401762259953305476662073440547103686134459899800170439220153140
0.25,4.525194807868986213551527861056118945100137707956145473134748928496702882015461265841573132502509
0.26,5.568920207362630710014622621272874663007378809135542425940476624773074735310618685328210723734261
0.27,6.233868420767935666960871211466248506894421395650810813088407562032811949603707886348425002240051
0.28,6.391322483206324054125349346823059929380507169641904581557509446690359551007232107481626782174924
0.29000000000000004,6.038771415829553214026012278035456570467441337725060301168269271068154461711646215
0.3,5.287337428574491149787972393816324230784636240757384656184093019668937999198593515121688430285394 5
0.31,4.311467637131350056286314333482261778684877190394735197379096879946447555842663299427189933023050
0.32,3.289086821340602256988585817942367487259712728530366039444152776152916218379011616621498290968535
0.33,2.357050445222320529697497113628508099775901906257923272179037052337438499988200400594293118391872
0.34,1.592675927055678610623653386894838496209165663078505995832617363555184217401574616633481859195660
0.35000000000000003,1.018196414943497508664044778585148665516666609779297376027048757293646086831452454
0.36000000000000004,6.177841677664727942728678416954065681777465174936801633364350514359277335543886835
0.37,3.567689624978236232813957245452299628085345917121972581044544144169612447431558824018050347992046
0.38,1.966196454418772440966607637690108899119224822470037869847084295942696721327874504902161120799041
0.39,1.036600883934514513225664726709268592402200343024640281609713504385286150720114294256409478576578
0.4,5.239817610090074051773416257371870852341848683218159943030154823952456410584628476164496567870915 2
0.41000000000000003,2.544740813295697802177783061356532690925602065776136467800122588177400251557278343
0.42000000000000004,1.189683150216987277372146198375036611168230811205982286959152226307959949053551090
0.43,5.363618526856141811070894093096416278496548565948599425449698476714291889728658433063180118291460
0.44,2.335866510807549712458449053068391167543435384293526909934067637935135631390755384591858008320556
0.45,9.841869604944558423440451206655701030601570603166685245492878061345873074247815216659130141045735
0.46,4.017704141982126750997062754589914020123579982443836484871232357444221122813502971818878012422503
0.47000000000000003,1.591254456754679430903399345622655007272639098013784483764858676100355817341537904
0.48000000000000004,6.122318473573020853562385324438907289046003307188625931302537374571116661943351292
0.49,2.291003997443330066934489875796771415716360234669781715255822636644061936536323778589877950387849
```