# Immunefi Audit Comp | Folks: Liquid Staking

Report of the review of Folks Finance's xAlgo liquid staking smart contracts.

*Keywords – audit competition, review, smart contract, Folks Finance, xALGO, Immunefi*

Valar Solutions GmbH

21 January, 2025

# Disclaimer

This report has been prepared with respect to the Immunefi Audit Competition for Folks Finance Liquid Staking protocol, published at Immunefi website. The report was initially submitted via Immunefi platform on 16 December 2024. The report is hereby made publicly available as the competition ended and all the findings acknowledged and/or resolved by Folks Finance.

The findings and recommendations provided in this report are based on the information available at the time of the review and the specific scope of the audit competition, i.e. the xALGO *consensus_v2.py* contract, published at Folks Finance public Github repository. The provided design overview, published as Google Docs document, served to assess the intent of the smart contracts under the scope. This report does not constitute legal or investment advice. This report was made only for the purposes of the audit competition. The report represents a high-level security assessment based on the information and code provided at the time of the review. The responsibility for the implementation and execution of recommended actions lies solely with the project team. The authors of this report assume no liability for any and all potential consequences of the deployment or use of the contract or any other actions taken based on its findings. Any changes made by the project team, including any changes made based on the findings of this report, have not been reviewed. The competition organizers have published their summary of all reports submitted, which is accessible here.

Classification of severity of any vulnerabilities has been assigned based on Immunefi Vulnerability Severity Classification System - v2.3. Reporting of any insights, as per Immunefi's guidelines for Audit Competitions, is also included. The classification of the severity level of any reported findings in this report may be assessed differently based on additional information available to the individual assessor, e.g. based on additional information about the intended functionality.

# Executive Summary

This report is for Immunefi Audit Competition for a smart contract by Folks Finance, which is to upgrade a smart contract that is already deployed on the Algorand Mainnet (application ID: 1134695678). The new contract is written in PyTEAL. Its purpose is to provide a liquid staking solution for Algorand, enabling users to earn staking rewards while also having the option to use their stake in other applications, e.g. in decentralized finance applications. This is done through the issuance of a token called xALGO, which represents the user's portion of the stake and earned staking rewards in the protocol. With respect to earning staking rewards, the contract aims to take into account Algorand's consensus participation mechanism, where the committed stake starts to participate and earn staking rewards only after a 320-round delay. To take this mechanism into account, the liquid staking protocol includes two minting options:

- Immediate mint: where the user receives xALGO immediately in exchange for the sent ALGO but has to pay a premium, receiving less xALGO than the current exchange rate in order to compensate for the fact that the sent stake will only earn staking rewards after the 320-round delay.
- Delayed mint: where the user receives xALGO in exchange for the sent ALGO at the earliest after a 320-round delay, but at the full exchange rate, as the stake is now already earning rewards. The delayed mint must be explicitly claimed before the user starts earning the staking rewards. Until it is claimed, the rewards are forfeited. The claiming can be initiated by anyone.

The contract aims to minimize the centralization related to liquid staking solutions by enabling multiple selected entities to participate with a portion of the stake committed to the liquid staking protocol. The selection and withdrawal of participation rights rests with a privileged admin account. The privileged admin account collects fees, which are deducted from the earned interest. These can be set to amount to 100%, i.e. the admin may collect all the rewards from the users. The upgraded contract remains upgradable by a privileged admin account. These aspects are in line with the provided documentation.

The review revealed one **high-level** security vulnerability as a portion of the funds can be temporarily frozen, i.e. a part of the minted xALGO cannot be burned, leading to users losing their funds. Because the smart contract remains upgradable, a future smart contract upgrade could possibly recover these funds.

The review also found two security vulnerabilities that could be classified as **low-level**, as the contract fails to deliver the promised returns to the users in a broader sense. However, due to the scarce documentation provided, it is unclear whether these are indeed errors or intentional design choices. These include:

- The protocol over-charging users for the necessary storage fees when using the delay minting option of xALGO, effectively reducing the users' returns.
- The privileged admin account having the potential for monopolizing any decentralized finance (DeFi) arbitrating activities, potentially leading to worse capital efficiency of minted xALGO, thus reducing users' returns for using the xALGO.

Besides the listed security vulnerabilities, the review found four insights that do not result in users losing funds or reduced returns but relate to smart contracts not fully or optimally implementing

the functionality intended based on the available documentation. These include:

- The code not being transparently upgradable through scheduling an upgrade, as well as the upgrade potentially failing - without blocking the protocol.
- The code documentation imprecisely stating it is performing checks on the existence of the proposer and/or performing redundant checks.
- The code not taking advantage of Algorand Virtual Machine version 11, with opcodes related to the staking rewards, e.g. about the required maximum account stake for eligibility of the account to receive staking rewards and whether an account has paid the opt-in fee to be eligible for rewards. With AVM11, it would be possible to reduce the users' risks with respect to an inept admin or proposer, thus improving the decentralization aspect of the protocol.
- The minting and burning methods not taking advantage of OpUp budget increase utility to reduce the number of application calls required at the outer group transaction level, increasing the modularity and usefulness of the protocol for DeFi applications - which is one of its main applications.

The rest of the document addresses the details of these findings.

# Contents

# Chapter 1

# Scope and Methodology

There is only one smart contract under the scope of this audit competition, found within the repository. It implements a planned upgrade to the smart contract already deployed by Folks Finance on Algorand Mainnet (application ID: 1134695678). The smart contract is written in PyTEAL. It relies on a few custom-written libraries, which have not been part of the audit competition scope.

The repository with the code includes basic instructions on how to compile and deploy the contract. It also includes some automated tests. The tests depend on the state of other tests in the suite, i.e. they depend on the order they are executed. This has been already raised by auditors of the currently deployed version of the protocol. The tests miss some relevant edge cases, e.g. with the maximum number of possible proposers added or the maximum amount of xALGO that can be burnt. Besides manual code review, adding such cases helped catch some of the found issues. For easier testing and reviewing, the whole code in the repository was migrated into AlgoKit - the current standard for developing Algorand smart contracts, which simplified the audit process.

The repository also includes the smart contracts for deploying a simplification of the version of the smart contract that is currently deployed. The current version and its simplification for testing were not part of the audit competition scope. For the purpose of this review, the simplification was modified to reflect the current state of the application deployed on Algorand Mainnet to better reflect the conditions of the upgrade.

# Chapter 2

# High-Level Security Vulnerability

The review revealed one **high-level** security vulnerability as a portion of minted xALGO cannot be burned, leading to users losing their funds. Because the smart contract remains upgradable, a future smart contract upgrade could possibly recover these funds. Therefore, the vulnerability is not deemed to fall within the critical-level scope.

## 2.1 Not all xALGO can be burned

The problem arises in `burn()` method (L793) due to a possible underflow at L824. The underflow happens because the ALGO to be returned during the burn is allocated only from the `total_active_stake`, while the xALGO in reality represents also the portion of ALGO that have been received by the protocol as part of the staking rewards `total_reward`. The amount of ALGO that cannot be burnt is increasing with the reward that the protocol is getting, i.e. $xALGO_{lost} = xALGO_{minted} * (R-U)/(A+R-U)$, where $A$ is the `total_active_stake`, $R$ the `total_reward`, and $U$ the `total_unclaimed_fees`. Based on the current state of the Algorand Mainnet (application ID: 1134695678), this would result in about 111,000 ALGO being lost by users that are last to burn their xALGO. The case has been demonstrated in Appendix A.1.

# Chapter 3

# Low-Level Security Vulnerabilities

The review found two security vulnerabilities that could be classified as **low-level**, as the contract fails to deliver the promised returns to the users in a broader sense. However, due to the scarce documentation provided, it is unclear whether these are indeed errors or intentional design choices.

## 3.1 Over-charging users on delayed mint

When a user calls `delayed_mint()` method (L695), it stores the user's mint information in box storage. The user has to pay the protocol ALGO deposit (i.e. increase in the minimum balance requirement of the application's account) for the creation of this box. The deposit is charged according to the number of bytes stored in the box and its name. The box gets deleted upon claiming the minted xALGO with `claim_delayed_mint()` (L732), freeing the storage ALGO deposit. The storage ALGO deposit gets returned to the caller of the `claim_delayed_mint()` method, which can be anyone - not necessarily the user who paid for the creation of the box. This is meant to reward anyone who automates the claiming process for the users.

However, the protocol is over-charing the users for this feature. The box namely includes unnecessary redundancy, which is costing users more ALGO than necessary. The box stores the address of the minter both as part of the box name (L733) and its contents (L736). This results in the minter unnecessarily paying for 32 bytes (i.e. length of Algorand address) of storage. The costs of this amount to 0.0128 ALGO, as per the current Algorand costs, on each mint. Assuming Folks Finance reaches the same levels of popularity as Lido - the most prominent liquid staking provider on Ethereum, which processes about 5M withdrawals per year (as per Lido analytics) and assuming the number of mints is in the same range as the number of withdrawals, this amounts to 64,000 ALGO over-charged to xALGO users per year, effectively (covertly) reducing their return.

The issue could be easily solved by removing the redundancy in the box content information. The case has been demonstrated by correcting the smart contract (V2.1) and performing a successful test Appendix A.2.

## 3.2 Admin monopolizing xALGO arbitrage activities

The privileged admin account has the option to change the `premium` charged for immediate minting of xALGO. The admin can do this by calling the `update_premium()` method (L514). The change can be done at any time, even multiple times within the same block. This allows the admin to wrap an xALGO immediate mint operation `immediate_mint()` within two `update_premium()` calls. This gives the admin an advantage corresponding to the size of the `premium` in any DeFi arbitrage or other opportunities. This advantage will likely discourage other participants to try to resolve any arbitrage opportunities as they cannot compete with the admin that can mint the xALGO at a discount compared to them. As one of the main purposes of the xALGO liquid staking token is to be used in DeFi applications and generate additional yield for users, this will likely result in worse liquidity, capital efficiency, and overall reduced returns for the xALGO users.

The issue could be easily solved by having a time limit on how frequently the `update_premium()` call can be made. The case has been demonstrated in Appendix A.3.

# Chapter 4

# Insights

The following sections provide details about code insights that do not result in users losing funds or reduced returns but relate to smart contracts not fully or optimally implementing the functionality intended based on the available documentation.

## 4.1 Issues with contract upgrades

The code can be upgraded by admin via a two-step process, i.e. by calling `schedule_update_sc` method to schedule an update and then calling the method `update_sc` to deploy it. The scheduling makes commitments in the form of hashes of the smart contract code, i.e. the approval and clear programs, which are to be uploaded later. If the method `update_sc` is called with programs that do not meet the hash commitments, the update will be rejected. The update can succeed only if the call is made after a predefined amount of time, which is set to 1 day.

The purpose of this two-step process is to give users the option to review the new contract changes and act by burning their xALGO before the new update goes live e.g. if they do not agree with the changes or find them malicious. The downside of the current implementation is that it still requires the admin to publish somewhere the code, which corresponds to the commitments made, for the users to review.

Moreover, the implementation includes an error. The maximum size of programs on Algorand is 8192 bytes. However, the maximum byte width of the Algorand Virtual Machine is 4096 bytes. This means that when verifying whether the correct programs are to be deployed after they have been scheduled, the calculation of hashes of programs larger than 4096 bytes will fail, i.e. at L353. Because a pending upgrade can be overwritten by the admin calling again `schedule_update_sc()` method, the error will not result in a blocked contract. The error has been demonstrated in Appendix A.4.

To resolve both of these limitations simultaneously, the contract could take advantage of box storage to store the code for the new upgrade already in the update scheduling step.

## 4.2 Documentation and checks on proposer existence

At multiple places in the code, a reoccurring comment reads `## check proposer exists` for line `Assert(proposer_index.get() < App.globalGet(num_proposers_key)),`, e.g. as in L481. This does not strictly check the existence of the proposer. That is done later when accessing the proposer's box, e.g. L486. Moreover, the check is always performed in calls where the call to `get_proposer()` method is later made. The `get_proposer()` method itself includes the same check at L89. Therefore, all these lines could be improved.

## 4.3 Not using Algorand Virtual Machine version 11

The contracts use Algorand Virtual Machine (AVM) version 10, while AVM version 11 is scheduled to be released. The AVM11 includes opcodes related to the staking rewards. These could be used e.g. to check if the admin has set sensible limits for `max_proposer_balance` at L400, and if a proposer's account has been correctly brought online to be eligible for incentives in method `register_online()` (L514). With these, the users' risks with respect to an inept admin setting a too large maximum proposer blanace, or an inept proposer admin bringing an account online without paying the incentives eligible fee, could be reduced, thus the decentralization aspect of the platform improved.

## 4.4 Not using OpUp budget increase utility

On Algorand, each smart contract call receives a budget of 700 for the execution of opcodes. Depending on the program's complexity, this can be insufficient and must be combined with additional smart contract calls as multiple smart contract calls within the same atomic transaction group share the total opcode budget. The minting and burning methods already require multiple smart contract calls in case of the maximum number of proposers possible to provide all the necessary resources to the AVM. However, these calls do not provide sufficient opcode budget and fail. Therefore, additional calls must be issued within the same group transaction to get a sufficient opcode budget.

With the so-called OpUp budget increase utility, an inner application transaction is issued to increase the opcode budget instead of requiring additional transaction to be issued at the outer group transaction level. This would enable a reduction in the number of application calls required on minting and burning operations in the case of the maximum number of proposers. With this, there would be more space to issue additional transactions within the same atomic group, increasing the modularity and usefulness of the application, especially for DeFi purposes - which is one of its main applications. The issue has been demonstrated in Appendix A.5.

# Appendix A

# Proof-of-Concept Code

The following sections provide code excerpts that demonstrate the relevant issues found during the review. The full demonstration suite was made available to the project team.

## A.1 Not all xALGO can be burned

The test demonstrating that not all xALGO can be burned is implemented in `burn_test.py`.

```
import pytest
from algokit_utils import (
    TransactionParameters,
)
from algokit_utils.beta.account_manager import AddressAndSigner
from algokit_utils.beta.algorand_client import AlgorandClient
from algokit_utils.beta.composer import AssetTransferParams
from algosdk.atomic_transaction_composer import (
    AtomicTransactionComposer,
    TransactionWithSigner,
)
from algosdk.error import AlgodHTTPError

from tests.consensus.conftest import BOX_PROPOSERS_PREFIX, Setup
from tests.utils import (
    available_balance,
    get_sp,
)


def test_burn_all_fails(
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
    setup: Setup,
) -> None:

    with pytest.raises(AlgodHTTPError) as e:
        # Get all xALGO in circulation
```

```
        burn_amt = available_balance(algorand_client, dispenser.address, setup.xalgo)
        atc = AtomicTransactionComposer()
        send_xalgo = TransactionWithSigner(
            algorand_client.transactions.payment(
                AssetTransferParams(
                    sender=dispenser.address,
                    asset_id=setup.xalgo,
                    signer=dispenser.signer,
                    receiver=setup.client.app_address,
                    amount=burn_amt,
                )
            ),
            signer=dispenser.signer,
        )

        setup.client.compose(atc).burn(
            send_xalgo=send_xalgo,
            min_received=1,
            transaction_parameters=TransactionParameters(
                sender=dispenser.address,
                signer=dispenser.signer,
                suggested_params=get_sp(algorand_client, 3),
                accounts=[setup.proposer.address],
                boxes=[(0, BOX_PROPOSERS_PREFIX)],
                foreign_assets=[setup.xalgo],
            ),
        ).build().execute(setup.client.algod_client, 1)
    assert "logic eval error: - would result negative" in str(e.value) and
        "opcodes=app_global_get; load 42" in str(e.value) # noqa: E501
    # Incorrectly fails on L824 of `consensus_v2.py` due to underflow

    return
```

## A.2 Over-charging users on delayed mint

The test showing overcharging of users is implemented in `claim_delayed_test.py`.

```
from algokit_utils import (
    CreateTransactionParameters,
    TransactionParameters,
)
from algokit_utils.beta.account_manager import AddressAndSigner
from algokit_utils.beta.algorand_client import AlgorandClient
from algokit_utils.beta.composer import AssetTransferParams, PayParams
from algosdk.abi import AddressType
from algosdk.atomic_transaction_composer import (
    AtomicTransactionComposer,
    TransactionWithSigner,
```

```
)
from algosdk.transaction import ApplicationUpdateTxn

import smart_contracts.artifacts.consensus_v_one.consensus_client as cv1
import smart_contracts.artifacts.consensus_v_two_one.consensus_client as cv21
from tests.consensus.conftest import (
    BOX_PROPOSER_ADMIN_PREFIX,
    BOX_PROPOSERS_PREFIX,
    BOX_USER_DELAY_MINT_PREFIX,
    CONSENSUS_DELAY,
    MBR_ACCOUNT,
    MBR_ASSET,
    MBR_PROPOSER_ADMIN_EMPTY_BOX,
    MBR_PROPOSERS_BOX,
    MBR_USER_DELAY_MINT_BOX,
    MBR_USER_DELAY_MINT_BOX_NEW,
    Defaults,
    Setup,
)
from tests.utils import (
    create_and_fund_account,
    get_approval_and_clear_bytes,
    get_sp,
    wait_for_rounds,
)

mint_amt = 10**9

def test_v2_0(
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
    setup: Setup,
) -> None:

    atc = AtomicTransactionComposer()
    # Mint with delay
    mbr_txn = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=setup.client.app_address,
                amount=MBR_USER_DELAY_MINT_BOX,
            )
        ),
        signer=dispenser.signer,
    )
    atc.add_transaction(mbr_txn)

    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
```

```
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=setup.client.app_address,
                amount=mint_amt,
            )
        ),
        signer=dispenser.signer,
    )

    nonce = 0
    nonce_bytes = nonce.to_bytes(2, "big")
    boxes = [
        (0, BOX_PROPOSERS_PREFIX),
        (0, BOX_USER_DELAY_MINT_PREFIX + AddressType().encode(dispenser.address) +
            nonce_bytes),
    ]

    atc = setup.client.compose(atc).delayed_mint(
        send_algo=send_algo,
        nonce=nonce_bytes,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 2),
            accounts=[setup.proposer.address],
            boxes=boxes,
            foreign_assets=[setup.xalgo],
        ),
    ).build().execute(setup.client.algod_client, 1)

    # Wait for 320 rounds to pass
    wait_for_rounds(algorand_client, CONSENSUS_DELAY, dispenser)

    # Claim mint
    setup.client.claim_delayed_mint(
        receiver=dispenser.address,
        nonce=nonce_bytes,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 3),
            accounts=[setup.proposer.address],
            boxes=boxes,
            foreign_assets=[setup.xalgo],
        ),
    )

    return
```

```
def test_v2_1(
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
) -> None:

    # Setup V2.1
    # ------------------------------------

    # Create V1 client
    client = cv1.ConsensusClient(algorand_client.client.algod)

    # ------------------------------
    # ----- Create V1 contract -----
    # ------------------------------
    client.create_create(
        admin=dispenser.address,
        register_admin=dispenser.address,
        min_proposer_balance=0,
        max_proposer_balance=Defaults().max_proposer_balance,
        premium=Defaults().premium,
        fee=Defaults().fee,
        transaction_parameters=CreateTransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            extra_pages=3,
        ),
    )

    # Fund 1st proposer with the total stake recorded in SC on mainnet + total
        rewards recorded in SC + for rekey fee + MBR # noqa: E501
    proposer = create_and_fund_account(algorand_client, dispenser,
        algo_amount=9505527975627 + 123438790257 + 1000 + MBR_ACCOUNT) # noqa: E501

    # Rekey proposer to app
    algorand_client.send.payment(
        PayParams(
            sender=proposer.address,
            signer=proposer.signer,
            receiver=proposer.address,
            amount=0,
            rekey_to=client.app_address,
        )
    )

    # Fund the application with MBR needed later to create boxes and opt into asset
    algorand_client.send.payment(
        PayParams(
            sender=dispenser.address,
            signer=dispenser.signer,
            receiver=client.app_address,
```

```
            amount=MBR_ACCOUNT + MBR_ASSET + MBR_PROPOSERS_BOX +
                MBR_PROPOSER_ADMIN_EMPTY_BOX,
        )
    )

    # ------------------------------
    # --- Initialize V1 contract ---
    # ------------------------------
    boxes = [
        (0, BOX_PROPOSERS_PREFIX),
        (0, BOX_PROPOSER_ADMIN_PREFIX + AddressType().encode(proposer.address)),
    ]
    res = client.initialise(
        proposer=proposer.address,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 2),
            boxes=boxes,
        ),
    )

    xalgo_id = res.tx_info["inner-txns"][0]["asset-index"]

    # Opt dispenser into xALGO
    algorand_client.send.asset_transfer(
        AssetTransferParams(
            sender=dispenser.address,
            receiver=dispenser.address,
            amount=0,
            asset_id=xalgo_id,
            signer=dispenser.signer,
        )
    )

    # ------------------------------
    # ----- Mint on V1 contract ----
    # ------------------------------
    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=proposer.address, # Payment for mint goes to the proposer -
                    just in V1!
                amount=0,
            )
        ),
        signer=dispenser.signer,
    )
```

```
res = client.mint(
    send_algo=send_algo,
    transaction_parameters=TransactionParameters(
        sender=dispenser.address,
        signer=dispenser.signer,
        suggested_params=get_sp(algorand_client, 2),
        accounts=[proposer.address],
        foreign_assets=[xalgo_id],
        boxes=[(0, BOX_PROPOSERS_PREFIX)],
    ),
)

# ------------------------------
# ---- Deploy V2.1 contract ----
# ------------------------------
ap, cp = get_approval_and_clear_bytes(client.algod_client,
    "consensus_v_two_one/ConsensusV21")

txn = TransactionWithSigner(
    txn=ApplicationUpdateTxn(
        sender=dispenser.address,
        index=client.app_id,
        approval_program=ap,
        clear_program=cp,
        sp=get_sp(algorand_client, 1),
    ),
    signer=dispenser.signer,
)
atc = AtomicTransactionComposer()
atc.add_transaction(txn).execute(client.algod_client, 1)

# After deploy of new contract, switch to new client (same app ID)
client = cv21.ConsensusClient(client.algod_client, app_id=client.app_id)

# ------------------------------
# ---Initialize V2.1 contract --
# ------------------------------
client.initialise(
    transaction_parameters=TransactionParameters(
        sender=dispenser.address,
        signer=dispenser.signer,
    ),
)

# ------------------------------
# ---- Enable delay minting ----
# ------------------------------
client.pause_minting(
    minting_type="can_delay_mint",
    to_pause=False,
    transaction_parameters=TransactionParameters(
```

```
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 1),
        ),
    )




    # -------------------------------------

    atc = AtomicTransactionComposer()
    # Mint with delay
    mbr_txn = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=client.app_address,
                amount=MBR_USER_DELAY_MINT_BOX_NEW,
            )
        ),
        signer=dispenser.signer,
    )
    atc.add_transaction(mbr_txn)

    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=client.app_address,
                amount=mint_amt,
            )
        ),
        signer=dispenser.signer,
    )

    nonce = 0
    nonce_bytes = nonce.to_bytes(2, "big")
    boxes = [
        (0, BOX_PROPOSERS_PREFIX),
        (0, BOX_USER_DELAY_MINT_PREFIX + AddressType().encode(dispenser.address) +
            nonce_bytes),
    ]

    atc = client.compose(atc).delayed_mint(
        send_algo=send_algo,
        nonce=nonce_bytes,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
```

```
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 2),
            accounts=[proposer.address],
            boxes=boxes,
            foreign_assets=[xalgo_id],
        ),
    ).build().execute(client.algod_client, 1)

    # Wait for 320 rounds to pass
    wait_for_rounds(algorand_client, CONSENSUS_DELAY, dispenser)

    # Claim mint
    client.claim_delayed_mint(
        receiver=dispenser.address,
        nonce=nonce_bytes,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 3),
            accounts=[proposer.address],
            boxes=boxes,
            foreign_assets=[xalgo_id],
        ),
    )

    return
```

## A.3 Admin monopolizing xALGO arbitrage activities

The test showing admin arbitrage monopoly is implemented in update_premium_test.py.

```
from algokit_utils import (
    TransactionParameters,
)
from algokit_utils.beta.account_manager import AddressAndSigner
from algokit_utils.beta.algorand_client import AlgorandClient
from algokit_utils.beta.composer import AssetTransferParams, PayParams
from algosdk.atomic_transaction_composer import (
    TransactionWithSigner,
)

from tests.consensus.conftest import BOX_PROPOSERS_PREFIX, Defaults, Setup
from tests.utils import create_and_fund_account, get_sp

mint_amt = 10**12
min_received = 1

def test_arbitrage_monopoly(
```

```
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
    setup: Setup,
) -> None:

    # Create an account and to opt-it in to xALGO so that it can later accept xALGO
    tmp_account = create_and_fund_account(algorand_client, dispenser, [setup.xalgo])

    # Admin updates premium to 0
    atc = None
    atc = setup.client.compose(atc).update_premium(
        new_premium=0,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 1),
        ),
    ).build()
    # Prepare a call to immediately mint xALGO with 0 premium
    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=setup.client.app_address,
                amount=mint_amt,
            )
        ),
        signer=dispenser.signer,
    )
    atc = setup.client.compose(atc).immediate_mint(
        send_algo=send_algo,
        min_received=min_received,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 2),
            accounts=[setup.proposer.address],
            boxes=[(0, BOX_PROPOSERS_PREFIX)],
            foreign_assets=[setup.xalgo],
        ),
    ).build()
    # The user does something with the minted xALGO - e.g. send it to a user
    send_xalgo = TransactionWithSigner(
        algorand_client.transactions.payment(
            AssetTransferParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=tmp_account.address,
                asset_id=setup.xalgo,
                amount=min_received,
```

```
            )
        ),
        signer=dispenser.signer,
    )
    atc.add_transaction(send_xalgo)
    # Admin updates premium back
    atc = setup.client.compose(atc).update_premium(
        new_premium=Defaults().premium,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 2),
        ),
    ).build()
    # Execute call
    res = atc.execute(setup.client.algod_client, 1)

    # Check
    assert res.confirmed_round

    # By comparing the minted amount of `test_arbitrage_monopoly` and
        `test_prove_arbitrage_monopoly`
    # it is seen that user with compliance of admin can have monopoly regarding
        arbitrage

    return

def test_prove_arbitrage_monopoly(
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
    setup: Setup,
) -> None:

    # Create an account and to opt-it in to xALGO so that it can later accept xALGO
    tmp_account = create_and_fund_account(algorand_client, dispenser, [setup.xalgo])

    # Prepare a call to immediately mint xALGO with premium
    atc = None
    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=setup.client.app_address,
                amount=mint_amt,
            )
        ),
        signer=dispenser.signer,
    )
    atc = setup.client.compose(atc).immediate_mint(
        send_algo=send_algo,
```

```
        min_received=min_received,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 3),
            accounts=[setup.proposer.address],
            boxes=[(0, BOX_PROPOSERS_PREFIX)],
            foreign_assets=[setup.xalgo],
        ),
    ).build()
    # The user does something with the minted xALGO - e.g. send it to a user
    send_xalgo = TransactionWithSigner(
        algorand_client.transactions.payment(
            AssetTransferParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=tmp_account.address,
                asset_id=setup.xalgo,
                amount=min_received,
            )
        ),
        signer=dispenser.signer,
    )
    atc.add_transaction(send_xalgo)
    # Execute call
    res = atc.execute(setup.client.algod_client, 1)

    # Check
    assert res.confirmed_round

    # By comparing the minted amount of ‘test_arbitrage_monopoly‘ and
        ‘test_prove_arbitrage_monopoly‘
    # it is seen that user with compliance of admin can have monopoly regarding
        arbitrage

    return
```

## A.4 Issues with contract upgrades

The test showing issues with upgrades is implemented in schedule_and_update_sc_test.py.

```
from hashlib import sha256

import pytest
from algokit_utils import (
    TransactionParameters,
)
from algokit_utils.beta.account_manager import AddressAndSigner
from algokit_utils.beta.algorand_client import AlgorandClient
```

```
from algokit_utils.beta.composer import PayParams
from algosdk.abi import ArrayStaticType, ByteType, TupleType, UintType
from algosdk.abi.method import Method, Returns
from algosdk.atomic_transaction_composer import (
    AtomicTransactionComposer,
    TransactionWithSigner,
)
from algosdk.error import AlgodHTTPError
from algosdk.transaction import ApplicationCallTxn, OnComplete

from tests.consensus.conftest import BOX_UPDATE_PREFIX, MBR_UPDATE_BOX, Setup
from tests.utils import (
    advance_time,
    get_approval_and_clear_bytes,
    get_box,
    get_latest_timestamp,
    get_sp,
)


def test_fails_when_update_too_large(
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
    setup: Setup,
) -> None:

    ap, cp = get_approval_and_clear_bytes(setup.client.algod_client,
        "consensus_v_two/ConsensusV2")
    approval_program = ap + ap # Double the size of program
    clear_program = cp
    approval_sha256 = sha256(approval_program).digest()
    clear_sha256 = sha256(clear_program).digest()

    # Schedule update
    atc = AtomicTransactionComposer()
    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=setup.client.app_address,
                amount=MBR_UPDATE_BOX,
            )
        ),
        signer=dispenser.signer,
    )
    atc.add_transaction(send_algo)

    atc = setup.client.compose(atc).schedule_update_sc(
        approval_sha256=approval_sha256,
        clear_sha256=clear_sha256,
```

```
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client),
            boxes=[(0, BOX_UPDATE_PREFIX)],
        ),
    ).build().execute(setup.client.algod_client, 1)

    # Wait for update
    time_cur = get_latest_timestamp(algorand_client)

    box_raw = get_box(algorand_client,BOX_UPDATE_PREFIX, setup.client.app_id)
    data_type = TupleType(
        [
            UintType(64),
            ArrayStaticType(ByteType(), 32),
            ArrayStaticType(ByteType(), 32),
        ]
    )
    decoded_tuple = data_type.decode(box_raw[0])
    time_target = decoded_tuple[0]

    time_delta = time_target - time_cur + 1 # One more than the min difference to
        wait
    advance_time(algorand_client, time_delta, dispenser)

    # Try to update
    with pytest.raises(AlgodHTTPError) as e:
        atc = AtomicTransactionComposer()
        txn_unsigned = ApplicationCallTxn(
            sender=dispenser.address,
            sp=get_sp(algorand_client, 2),
            index=setup.client.app_id,
            on_complete=OnComplete.UpdateApplicationOC,
            approval_program = approval_program,
            clear_program = clear_program,
            app_args=[
                Method(
                    name="update_sc",
                    args=[],
                    returns=Returns(arg_type="void"),
                ).get_selector()
            ],
            boxes=[(0, BOX_UPDATE_PREFIX)],
        )

        txn = TransactionWithSigner(
            txn=txn_unsigned,
            signer=dispenser.signer,
        )
        atc.add_transaction(txn)
```

```
        atc.execute(setup.client.algod_client, 1)
    assert "txn produced a too big" in str(e.value) and "byte-array" in str(e.value)

    return
```

## A.5 Not using OpUp budget increase utility

The test showing issues with number of calls is implemented in immediate_mint_test.py.

```
import random

from algokit_utils import (
    TransactionParameters,
)
from algokit_utils.beta.account_manager import AddressAndSigner
from algokit_utils.beta.algorand_client import AlgorandClient
from algokit_utils.beta.composer import PayParams
from algosdk.abi import AddressType
from algosdk.atomic_transaction_composer import (
    AtomicTransactionComposer,
    TransactionWithSigner,
)

from tests.consensus.conftest import (
    BOX_PROPOSER_ADMIN_PREFIX,
    BOX_PROPOSERS_PREFIX,
    MBR_PROPOSER_ADMIN_EMPTY_BOX,
    Setup,
)
from tests.utils import (
    ALGO_ASA_ID,
    balance,
    create_and_fund_account,
    get_sp,
)

NUM_ADDITIONAL_CALLS = 2 # Without two additional app calls, the opcode budget is
    too small and the call fails

def test_mint_w_30_proposers(
    algorand_client: AlgorandClient,
    dispenser: AddressAndSigner,
    setup: Setup,
) -> None:

    proposers = [setup.proposer]
    # Create 29 accounts for new proposers
    to_fund = balance(algorand_client, setup.proposer.address, ALGO_ASA_ID) + 1_000
    n = 29
```

```
for _ in range(n):
    proposer = create_and_fund_account(algorand_client, dispenser,
        algo_amount=to_fund)
    proposers.append(proposer)

# Fund app with enough to cover MBR for all (empty) box creations
algorand_client.send.payment(
    PayParams(
        sender=dispenser.address,
        signer=dispenser.signer,
        receiver=setup.client.app_address,
        amount=MBR_PROPOSER_ADMIN_EMPTY_BOX * n,
    )
)

# Add 29 proposers
p_idx_start = 1
txn_cnt = 16
for p_i in range(n):
    if txn_cnt == 16:
        txn_cnt = 0
        atc = AtomicTransactionComposer()

    proposer_idx = p_idx_start + p_i
    proposer = proposers[proposer_idx]

    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=proposer.address,
                signer=proposer.signer,
                receiver=proposer.address,
                amount=0,
                rekey_to=setup.client.app_address,
            )
        ),
        signer=proposer.signer,
    )
    atc = atc.add_transaction(send_algo)

    proposer_idx = p_idx_start + p_i
    atc = setup.client.compose(atc).add_proposer(
        proposer=proposer.address,
        transaction_parameters=TransactionParameters(
            sender=dispenser.address,
            signer=dispenser.signer,
            suggested_params=get_sp(algorand_client, 1),
            boxes=[
                (0, BOX_PROPOSERS_PREFIX),
                (0, BOX_PROPOSER_ADMIN_PREFIX +
                    AddressType().encode(proposer.address)),
```

```
                ],
            ),
        ).build()

        txn_cnt += 2
        if txn_cnt == 16 or p_i == n - 1:
            atc.execute(setup.client.algod_client, 0)

    # Mint xALGO
    mint_amt = 10**9
    atc = AtomicTransactionComposer()

    send_algo = TransactionWithSigner(
        algorand_client.transactions.payment(
            PayParams(
                sender=dispenser.address,
                signer=dispenser.signer,
                receiver=setup.client.app_address, # Payment for mint goes to the
                    application
                amount=mint_amt,
            )
        ),
        signer=dispenser.signer,
    )

    num_proposers = len(proposers)

    if num_proposers <= 4:
        foreign_accounts = [p.address for p in proposers]
    else:
        num_dummy = num_proposers // 4
        for di in range(num_dummy):
            foreign_accounts = [p.address for p in proposers[di * 4 : (di + 1) * 4]]

            atc = setup.client.compose(atc).dummy(
                    transaction_parameters=TransactionParameters(
                        sender=dispenser.address,
                        signer=dispenser.signer,
                        suggested_params=get_sp(algorand_client, 1),
                        accounts=foreign_accounts,
                    ),
                ).build()

        foreign_accounts = [p.address for p in proposers[num_dummy * 4 :]]

    # Add other dummy calls to increase opcode budget
    for di in range(NUM_ADDITIONAL_CALLS):
        atc = setup.client.compose(atc).dummy(
            transaction_parameters=TransactionParameters(
                sender=dispenser.address,
                signer=dispenser.signer,
```

```
            suggested_params=get_sp(algorand_client, 1),
            lease=hash(di + random.random()).to_bytes(
                32, "big"
            ),  # Because txn must be unique
        ),
    ).build()

atc = setup.client.compose(atc).immediate_mint(
    send_algo=send_algo,
    min_received=1,
    transaction_parameters=TransactionParameters(
        sender=dispenser.address,
        signer=dispenser.signer,
        suggested_params=get_sp(algorand_client, 2 + num_proposers),
        accounts=foreign_accounts,
        boxes=[(0, BOX_PROPOSERS_PREFIX)],
        foreign_assets=[setup.xalgo],
    ),
).build()

res = atc.execute(setup.client.algod_client, 1)

# Check
assert res.confirmed_round

return
```